Aprendendo Java na marra - Avançado

Alexandre Henrique de Souza Torres
Roberto Silva Cantanhede
Rafael Henrique Santos Soares

Novembro 2015

Conteúdo

Prefácio

Introdução

Capítulo 1

Como programar com classe!

Vamos começar revendo um conceito apresentado no livro anterior: o Registro. Naquela ocasião definimos um registro como "um conjunto de valores diferentes em uma variável, mas seus valores podem ser de tipos diferentes e eles são distinguidos pelo nome, normalmente chamados campos".

Contudo, o que dissemos que era um registro, não era bem isso. Usamos esse recurso didático para introduzir os conceitos paulatinamente. Lembre-se que quando apresentamos o código correspondente à criação do registro, usamos o comando `class` porque estávamos, na verdade, escrevemos uma classe.

Então, após tanto tempo juntos, já é hora de você saber a verdade sobre o comando `class`. Digite o código a seguir e vamos conversar.

```
1  class CalculadoraIMC
2  {
3    double altura;
4    double peso;
5
6    void mostraIMC() {
7      double imc;
8      imc = peso / (altura*altura);
9      System.out.println("Valor do IMC: " + imc);
10   }
11
12   void mostraClassificacaoIMC() {
13     double imc;
14     imc = peso / (altura*altura);
15
16     System.out.print( "Categoria de IMC: " );
17     if ( imc < 18.5 )
18     {
19       System.out.println( "Sobpeso" );
20     }
21     else if ( imc < 25.0 )
22     {
23       System.out.println( "Peso normal" );
24     }
25     else if ( imc < 30.0 )
26     {
27       System.out.println( "Sobrepeso" );
28     }
29     else
30     {
31       System.out.println( "Obesidade" );
32     }
33   }
34 }
35
36 public class UtilizandoIMC
37 {
38   public static void main( String[] args )
39   {
40     CalculadoraIMC calc;
41     calc = new CalculadoraIMC();
42
43     calc.altura = 1.83;
44     calc.peso = 91;
45
46     calc.mostraIMC();
```

```
47      calc.mostraClassificacaoIMC();
48
49    }
50 }
```

O que você deve ver

```
Valor do IMC: 27.173101615455817
Categoria de IMC: Sobrepeso
```

Classes são a base da programação orientada a objetos e, portanto, da linguagem Java. Exploraremos as classes ao máximo durante o livro. Por enquanto, você precisa saber o seguinte:

- O Java, como qualquer outra linguagem, tem *tipos primitivos*. Esse são os tipos que você vem usando com frequência:`int, double` e `boolean` [1]. Esses tipos geram variáveis que contém apenas um valor e mais nada.
- *Classes* são tipos complexos. Elas podem ter mais de um valor diferente armazenado, e também "fazem coisas", isto é, podemos pedir que ela produza algum resultado novo.
- A partir de agora, nós vamos chamar as *variáveis de tipos complexo* de "objetos".
- O nome de classe sempre começa com letra maiúscula. Isto é uma convenção, o que quer dizer que se você fizer diferente o seu programa irá copilar e funcionar, mas os outros programadores Java irão te considerar um rebelde que está indo contra o sistema e terão dificuldades de ler seu programa.
- As classes possuem atributos e métodos. Atributos são variáveis da classe que guardam informações. Métodos são funções da classe que podem ser invocadas para executar ações.

O comando `class` na linha 1 descreve uma classe. Essa classe está contida entre as chaves que são abertas na linha 2 e fechadas na linha 34. Essa classe recebe o nome *CalculadoraIMC*.

Observe que na linha 36 temos outra classe sendo criada. Ela está recebendo o nome *UtilizandoIMC* e é precedida pela palavra reservada `public`, que define a visibilidade dessa classe como **pública**.

Se a visibilidade de *UtilizandoIMC* é pública, qual a visibilidade de *CalculadoraIMC*? Ela é privada. A diferença entre classes públicas e privadas será explorada mais a frente. Agora, eu preciso te falar apenas que classes privadas não são muito comuns e nos próximos capítulos vamos mudar isso.

Vamos voltar a estudar a classe *CalculadoraIMC*. Perceba que as linhas 3 e 4 declaram variáveis, que daqui pra frente vamos chamar de atributos, pois pertencem a uma classe. Essas variáveis são do tipo `double`. Até aqui nenhuma novidade para quem já estudou "registros".

Na linha 6 e na linha 12 as coisas começam a ficar interessantes. Veja que há, nessas linhas, a definição de duas funções, que daqui em diante chamaremos de métodos, pois pertencem a uma classe. O método `mostraIMC` é definido na linha 6 e vai até a linha 10. O método `mostraClassificacaoIMC` começa na linha 12 e vai até a linha 33. Perceba que ambos pertencem à classe `CalculadoraIMC` por estarem entre as chaves que definem essa classe (a chave na linha 2 inicia e a chave na linha 34 termina a definição da classe).

Assim como quando definíamos "registros"(que a essa altura você já percebeu que eram apenas classes comuns, sem nenhum método definido), éramos capazes de utilizar as variáveis definidas no registro usando um ponto, no formato: `nomeDoRegistro.nomeDaVariavel`, também somos capazes de chamar métodos de uma classe simplesmente usando o nome do objeto criado a partir dessa classe e o nome do método separados por ponto, no formato: `nomeDoObjeto.nomeDoMétodo`.

É exatamente isso que é feito na linha 46 e na linha 47.

[1] Existem outros, tais como `short, byte, long, float`... Basicamente a diferença entre eles é o tamanho do número que eles armazenam. Nós não iremos trabalhar com eles neste livro, mas vale a pena você consultar a documentação do Java.

Na linha 36 temos a classe pública `UtilizandoIMC` e logo na linha 38 temos a função `main`. Você sabe que essa é uma função especial. Quando eu executar meu programa ele começara a execução por aí. Então é aqui que o show acontece.

Na linha 45 dizemos ao Java que queremos acessar um método chamado `mostraIMC()` que é oferecido pelo objeto `calc`.

Na linha 46 dizemos ao Java que queremos acessar um método chamado `mostraClassificacaoIMC()` que também é oferecido pelo objeto `calc`.

O objeto `calc`, por sua vez é um objeto do tipo `CalculadoraIMC`. Isso está definido na linha 40.

Perceba que, quando declaramos uma variável utilizamos a seguinte estrutura: tipo da variável espaço nome da variável. Nessa estrutura, posso dar o nome que eu quiser para a variável, mas o tipo deve ser um tipo existente.

Até agora vínhamos utilizando tipos primitivos (`int`,`double`,`boolean`) e tipos complexos fornecidos pelo Java (`Scanner`, `String`). Na linha 40 estamos usando um tipo complexo que nós mesmos criamos: o tipo `CalculadoraIMC`! Legal demais, não é?

Capítulo 2

Mais classe e seus objetos

Se você é um dos felizardos que leu nosso livro anterior[1], irá perceber que começamos esse capítulo revisitando um exercício que já havíamos feito, só que agora dando uma incrementada. Digite o programa:

```
1  class Endereco
2  {
3    String rua;
4    String cidade;
5    String estado;
6    int CEP;
7
8    public String toString(){
9      return rua + " " + cidade + ", " + estado + " - " + CEP;
10   }
11
12   public boolean pertoDe( Endereco end)
13   {
14     boolean perto = false;
15
16     if ( (end.CEP - this.CEP >= -100) && (end.CEP - this.CEP <= 100) ) {
17       perto = true;
18     }
19
20     return perto;
21   }
22 }
23
24 public class EnderecoPostal
25 {
26   public static void main(String[] args)
27   {
28     Endereco um, dois, tres;
29
30     um = new Endereco();
31     um.rua = "Rua Jardineira, número 38";
32     um.cidade = "Taboquinha";
33     um.estado = "CE";
34     um.CEP = 33179180;
35
36     dois = new Endereco();
37     dois.rua = "Av. das Araras, número 4004";
38     dois.cidade = "Taboquinha";
39     dois.estado = "CE";
40     dois.CEP = 33179230;
41
42     tres = new Endereco();
43     tres.rua = "SQS 317 BL Y AP 701";
44     tres.cidade = "Brasília";
45     tres.estado = "DF";
46     tres.CEP = 74404010;
47
48     System.out.println( "O endereço " + um +"\ne o endereco " + dois);
49
50     if (um.pertoDe(dois)) {
51       System.out.println("são próximos. ");
52     } else {
53       System.out.println("não são próximos. ");
```

[1] "Aprenda Java na Marra", disponível em javanamarra.com.br.

```
54      }
55
56    }
57  }
```

O que você deve ver

```
O endereço Rua Jardineira, número 38 Taboquinha, CE - 33179180
e o endereco Av. das Araras, número 4004 Taboquinha, CE - 33179230
são próximos.
```

Como você já aprendeu, a linha 1 contém a palavra `class` que indica que estamos criando uma nova classe. Essa nova classe está contida entre as chaves que são abertas na linha 2 e fechadas na linha 22. Esse classe recebe o nome de `Endereco`.

Na linha 24 também temos uma classe sendo criada. Ela está recebendo o nome de `EnderecoPostal` e é precedida pela palavra reservada `public`, que define a classe como pública.

Voltemos para a classe `Endereco`. As linhas 3 a 6 definem quais os atributos da classe.

As linhas 8 a 10 definem uma função chamada `toString`. Como você deve se lembrar, ela é uma função especial que é utilizada sempre que um objeto dessa classe precisa ser mostrado como uma String[2].

Nas linhas 12 a 21 nós criamos uma outra função chamada `pertoDe`. Você já é grandinho o suficiente para entender a lógica que está escrita ali.

Portanto, nesse ponto do programa nós definimos uma classe chamada `Endereco`, que pode receber quatro tipos de valores diferentes e pode fazer duas coisas: ser transformada em uma String e verificar se um endereço é perto de outro. Porém, nós ainda não *fazemos* nada. As coisas foram definidas, mas não foram utilizadas.

Na linha 24 começamos a classe pública `EnderecoPostal`.

Na linha 28 eu defino três variáveis do tipo `Endereco`. Ou melhor, eu defino três... objetos. Vamos começar a dar nomes mais sofisticados para as coisas.

Na linha 30 temos uma coisa bacana, o objeto `um` recebe `new Endereco()`.

Quando declaramos uma variável de um tipo complexo como fizemos na linha 30, ela "nasce"em branco. O objeto `um` está apenas definido, mas não foi construído e contém "nada"dentro dele, que em Java quer dizer que ele é `null`.

Para que um objeto saia de `null` e passe a ter a estrutura complexa que foi definida pela classe, precisamos fazer duas coisas:

1. usar a palavra reservada `new`. Ela indica que um novo objeto está sendo criado.
2. colocar o nome da classe com um abre-e-fecha parêntesis (no nosso caso ficou `Endereco()`). Isso diz ao Java a partir de qual classe o objeto vai ser construído.

Nas linhas 30 a 46 nós criamos três objetos e colocamos valores em seus atributos usando a estrutura `nomeDoObjeto.nomeDoAtributo` que já vimos.

A linha 48 é interessante. Ali a função `toString` dos objetos `um` e `dois` é chamada implicitamente. Consegue ver isso?

Na linha 50 nós chamamos a função `pertoDe` do objeto `um`. Como parâmetro nós passamos o objeto `dois`. Legal, né?

O resto da lógica você consegue entender. Mas antes de terminar, temos mais uma coisinha para aprender.

[2]Isso acontece, por exemplo, na linha 48. Perceba que nosso código "soma"uma `String` ("O endereço ") com um objeto (`um` do tipo `Endereço`). Isso só é possível por que todo objeto em Java tem um método `toString` que é invocado sempre que usamos esse artifício de "somar"(na verdade, concatenar) `String`s e objetos. No exemplo, ajustamos o retorno desse método ao nosso gosto, por meio da instrução descrita nas linhas 8 a 10.

Vamos voltar lá para a linha 16, dentro da função `pertoDe()`. Observe a linha:

```
if ( (end.CEP - this.CEP >= -100) && (end.CEP - this.CEP <= 100) ) {
```

Eu quero destacar a palavra `this`. Ela é usada para um objeto se referenciar a algo dele próprio, ou seja, essa linha esta dizendo "compare a variável CEP de `end` com a minha própria variável CEP". Isso é especialmente útil nessa situação para deixar claro de qual `CEP` estamos falando. Perceba que dentro do método `pertoDe` há duas variáveis chamadas `CEP`: Uma pertencente ao `Endereco` recebido como parâmetro no métodos e outra pertencente ao próprio `Endereco` dono do método `pertoDe` invocado. Assim, da mesma forma que a palavra `end` em `end.CEP` indica que estamos acessando a variável `CEP` pertencente ao objeto `end`, a palavra `this` indica que estamos nos referindo à variável `CEP` pertencente ao objeto utilizado quando da execução de `pertoDe`.

Nós ainda iremos usar bastante o `this`. Tudo se tornará claro como o tempo.

Desafios para estudo

1. Nas linhas 48 e 50, troque `dois` por `tres`. Veja o resultado. Entendeu o que aconteceu?
2. Crie mais um objeto e utilize sua função `pertoDe`.

Capítulo 3

Explicações (chatas) sobre classes e objetos

Neste capítulo vamos ter um pouco de teoria chata sobre classes e objetos. Mas antes vamos fazer um programa para ver a teoria chata em ação. Aliás, dessa vez vamos fazer dois programas. Eu não preciso dizer qual o nome dos arquivos, preciso?[1]

Primeiro, digite o seguinte programa, mas não compile ainda:

```
1  public class Livro
2  {
3    String titulo;
4    String editora;
5    int anoPublicacao;
6    int paginas;
7
8    public String geraReferenciaBibliografica()
9    {
10     return titulo + ", " + paginas +" pgs. Publicado por: " + editora + " em " +
       anoPublicacao;
11   }
12
13 }
```

Agora, digite este outro, mas também não compile, até eu dar algumas explicações:

```
1  public class ControleLivro
2  {
3    public static void main(String[] args) {
4      Livro lv = new Livro();
5
6      lv.titulo = "Aprendendo Java na marra";
7      lv.editora = "Edição Independente";
8      lv.anoPublicacao = 2015;
9      lv.paginas = 152;
10
11     System.out.println( lv.geraReferenciaBibliografica() );
12   }
13 }
```

O que você deve ver

```
Aprendendo Java na marra, 152 pgs. Publicado por: Edição Independente em 2015
```

Ok, agora, vamos a uma pequena mágica. Compile apenas o programa `ControleLivro.java`. Após compilar sem erro, verifique os arquivos `.class` do seu diretório. Viu que você apareceu um `Livro.class`? Isto porque quando o Java encontra a classe `Livro` na linha 4, ele procura um arquivo `Livro.java` no diretório e compila esse arquivo também automaticamente. Inteligente, não?

Porém, agora você deverá ter um cuidado a mais. Quando receber um erro de compilação, observe bem em qual arquivo o erro ocorreu. Mesmo que seu programa principal esteja livre de

[1] Os arquivos devem ter sempre o nome da classe pública acrescido de `.java`.

erros, se um programa auxiliar tiver algo errado isso impedirá a compilação do projeto como um todo.

Esses dois programas não fazem nada de tão maravilhoso e você consegue entender todas as linhas de código. O mais importante é que nós definimos uma segunda classe pública em um arquivo separado (classe `Livro` no arquivo Livro.java) e é assim que nós fazemos em 99%[2] das vezes.

Vamos dar uma rápida olhada nos programas. O programa que contém a classe `Livro` é bastante simples e você consegue entendê-lo.

Vamos olhar `ControleLivro`. A linha 4 declara e inicializa um objeto `lv` que é do tipo `Livro`.

As linhas de 6 a 9 colocam valores nos atributos desses objetos.

E a linha 11 chama o método `geraReferenciaBibliofrafica()` desse objeto.

Mas afinal, o que é um objeto? Gosto de dar aos meus alunos a minha definição pessoal de objetos, que até este momento você não encontraria em nenhum livro:

"Objeto é uma coisa. Mas não é qualquer coisa. É uma coisa que tenha significado para a sua aplicação."

Por mais que pareça brincadeira, essa é a mais absoluta verdade. No tempo dos dinossauros, antes de haver programação orientada a objetos, os programadores amontoavam seu código em diversos arquivos organizados de qualquer maneira. Um arquivo famoso que sempre podíamos encontrar era o `util.xxx`. Esse arquivo continha várias funções úteis, tais como o cálculo de dígito verificador e outras coisas bem comuns.

Porém, um dia alguém teve a maravilhosa ideia de pegar os conceitos da aplicação e começar a organizar os programas a partir desses conceitos. Se você estiver fazendo um programa para controlar uma biblioteca, estes seriam alguns conceitos que provavelmente você iria ter:

- Livro;
- Editora;
- Autor;
- Estante;
- Sócio;

Esses conceitos muito provavelmente serão classes em um sistema de controle de biblioteca porque são "coisas que têm significado nessa aplicação".

Se você prestou atenção, o parágrafo anterior ficou meio confuso porque eu estava falando sobre "objetos" e de repente me referi a "classes". Então, antes de prosseguirmos, vamos definir o que é uma classe:

"Uma classe é um molde para a construção de objetos"

Talvez essa definição não tenha sido tão chique quanto você esperava, mas ela é verdadeira. O que nós *definimos* no programa são as classes e a partir dela nós *construímos* objetos. Veja seu arquivo `Livro.java`. Ali nós definimos uma classe e só isso. Nada acontece, pois não existem objetos sendo criados e manipulados.

A partir da linha 4 do `ControleLivro.java` nós criamos um objeto chamado `lv`. Esse objeto é uma réplica exata do que foi definido na classe Livro. A classe serviu de molde para a criação do objeto. Nós dizemos que o objeto é uma "instância" da classe.

É só por agora. Vamos fazer um pequeno exercício e torcer para que o próximo capítulo seja mais divertido.

[2]Essa é uma estatística inventada. Aliás, 92,35% das estatísticas são inventadas.

Desafios para estudo

Apenas para não perder a prática, crie mais dois objetos do tipo `Livro` e imprima seu conteúdo na tela.

Capítulo 4

Agora você vê, agora você não vê - Entendendo a visibilidade

A partir de agora, sempre teremos um ou mais programas representando as classes e um programa principal responsável por criar os objetos e fazer as engrenagens funcionarem. Esse programa é o que tem o `public static void main...` e nós vamos chamá-lo de "programa principal" ou "controlador", certo?

Vamos digitar os programas, mas observe que o seu programa principal contém um erro e não vai compilar. Tudo bem, são coisas da vida e daqui a pouco eu explico o que está acontecendo.

Primeiro a classe Funcionário:

```
1  public class Funcionario
2  {
3    String nome;
4    private double salario;
5
6    public void defineSalarioInicial(){
7      salario = 1000;
8    }
9
10   public void alteraSalario(double percentualReajuste)
11   {
12     this.salario *= (1 + percentualReajuste/100);
13     System.out.println("Salário reajustado em " + percentualReajuste + "%");
14   }
15
16   public String toString()
17   {
18     return "Nome:\t\t" + nome +"\nSalário:\t" + salario;
19   }
20 }
```

Agora o programa principal:

```
1  public class SalarioFuncionario
2  {
3    public static void main(String[] args) {
4      Funcionario f = new Funcionario();
5
6      f.nome = "Antonio";
7
8      // ESSA LINHA VAI DAR ERRO
9      f.salario = 1000;
10
11     System.out.println("Dados do funcionário\n" + f);
12   }
13 }
```

O que você deve ver (por enquanto)

```
SalarioFuncionario.java:9: error: salario has private access in Funcionario
    f.salario = 1000;
     ^
1 error
```

No programa principal nós criamos um objeto `f` do tipo `Funcionario` na linha 4. Veja que nós declaramos e inicializamos o objeto com o `new`, então o nosso objeto está definido e pronto para uso.

Na linha 6 nós colocamos um valor no atributo `nome` e tudo funciona bem, mas quando chega na linha 9, e nós queremos colocar um valor no atributo `salario`, o Java bate o pé, faz bico e te mostra a mensagem de erro na compilação. Por quê? Para entender, vamos olhar a classe `Funcionario`.

Nas linhas 3 e 4 temos a declaração dos atributos, mas se você observar bem, a linha 4 começa com a palavra `private`. Essa palavra está definindo a visibilidade do atributo como *privada*. Vamos entender o que isso quer dizer:

- **private** define a visibilidade privada. Isto é, apenas o próprio objeto tem acesso ao atributo. Para outros objetos é como se esse atributo não existisse.
- **public** define a visibilidade como pública. Isto quer dizer que todos os outros objetos podem ver e alterar o valor desse atributo.

Métodos também podem ter a visibilidade alterada e funcionam da mesma forma: `private`, só o próprio objeto chama o método. `public` o método está disponível para ser chamado por qualquer outro objeto[1].

Antes de avançarmos mais, grave uma coisa: atributos *sempre* devem ser privados e métodos *sempre* devem ser públicos[2].

No nosso exemplo, na classe `Funcionario`, linha 3, não temos nenhuma alteração de visibilidade para o atributo `nome`. Isso quer dizer que ele é público por *default* [3], e `nome` pode ser alterado por qualquer outro objeto.

Já a linha 4 começa com `private`, dizendo que `salario` só pode ser alterado pelo próprio objeto dono do atributo.

Agora que já sabemos *o que* é a visibilidade temos que entender *porque* fazer assim e *como* resolver o problema do programa.

Vamos ao porquê. Para isso, pense que a sua classe `Funcionario` será usada em um grande sistema de milhares de linhas de código. Você gostaria que um programador de segunda categoria, que não leu esse livro, saísse alterando o salário dos funcionários de qualquer maneira? Claro que não. A alteração de salário deve ser feita dentro de um processo específico e apenas em algumas ocasiões.

Como garantir que isso seja seguido? Colocando o atributo `salario` como privado e criando um método público que controle a alteração. Sacou?

Foi o que fizemos. Nas linhas de 6 a 9 temos o método `public void alteraSalario()` que tem visibilidade pública e faz a alteração do jeito que queremos (que nesse exemplo não é grande coisa, mas use sua imaginação).

Agora é hora do *como* fazer funcionar. Isso é fácil e acho que você consegue pensar na maneira de compilar o programa e rodar de forma que ao final o resultado seja:

```
Dados do funcionário
Nome:    Antonio
Salário:   1000.0
```

Desafios para estudo

1. Nós dissemos que atributos têm que ser privados, não dissemos? Então acrescente `private` na linha 3 da classe `Funcionario` para alterar a visibilidade de `nome`.

[1] Classes também podem ser definidas como `public` ou `private`, mas não trataremos disso agora. Fiquemos com métodos e atributos.

[2] Como várias coisas nesse livro, não é *sempre* assim, mas em aproximadamente 99% das vezes. Falaremos disso no próximo capítulo

[3] Essa é uma palavra que programadores usam muito e significa "a situação padrão se nada for alterado". Pronuncia-se "defou" e é uma palavra legal e descolada para você usar por aí.

2. Crie um método em `Funcionario` chamado `setNome` que receba uma String e não retorne nada. Esse método deve colocar o valor da String recebida como parâmetro no atributo `nome`.

3. Altere a linha 6 de `SalarioFuncionario` para usar o método que você criou. Deixe tudo funcionando e rodando. Vamos usar isso no próximo capítulo.

Capítulo 5

Encapsulamento

Se você não completou os *Desafios para estudo* do capítulo anterior é importante que você volte e faça tudo corretamente. Vamos precisar disso agora.

Vamos recapitular os programas que usamos no capítulo anterior, agora atualizados. Verifique as linhas que precisam ser acrescentadas.

A classe `Funcionario`:

```
1  public class Funcionario
2  {
3    private String nome;
4    private double salario;
5    private boolean ativo = true;
6
7    public void defineSalarioInicial(){
8      salario = 1000;
9    }
10
11   public void alteraSalario(double percentualReajuste)
12   {
13     this.salario *= (1 + percentualReajuste/100);
14     System.out.println("Salário reajustado em " + percentualReajuste + "%");
15   }
16
17   public String toString()
18   {
19     String texto =  "Nome:\t\t" + this.getNome() +"\nSalário:\t" + this.getSalario()
       ;
20
21     if (this.isAtivo())
22       texto += "\nO funcionário está ativo";
23     else
24       texto += "\nO funcionário está inativo";
25
26     return texto;
27   }
28
29   public void setNome(String nome)
30   {
31     this.nome = nome;
32   }
33
34   public String getNome()
35   {
36     return this.nome;
37   }
38
39   public void setAtivo(boolean ativo)
40   {
41     this.ativo = ativo;
42   }
43
44   public boolean isAtivo()
45   {
46     return this.ativo;
47   }
48
49   public double getSalario()
50   {
```

```
51      return this.salario;
52    }
53  }
```

Agora o programa principal:

```
1  public class SalarioFuncionario
2  {
3    public static void main(String[] args) {
4      Funcionario f = new Funcionario();
5
6      f.setNome("Antonio");
7
8      // ESSA LINHA VAI DAR ERRO
9      //f.salario = 1000;
10
11     f.defineSalarioInicial();
12
13     System.out.println("Dados do funcionário\n" + f);
14
15     f.alteraSalario(10.0);
16     f.setAtivo(false);
17
18     System.out.println("\nDados do funcionário\n" + f);
19   }
20
21 }
```

O que você deve ver

```
Dados do funcionário
Nome:    Antonio
Salário:  1000.0
O funcionário está ativo
Salário reajustado em 10.0%

Dados do funcionário
Nome:    Antonio
Salário:  1100.0
O funcionário está inativo
```

Uau! Nós temos muitas coisas novas acontecendo nessas classes agora.

Vamos olhar com calma para a classe `Funcionario` porque ela que contém toda a mágica da orientação a objetos.

Até a linha 4 temos os mesmos atributos de antes, todos privados agora. Na linha 5 eu introduzi um novo atributo `boolean` que indica se o funcionário está ativo ou não. Faremos algumas coisinhas com ele daqui a pouco.

Nas linhas que vão da 17 até a 27, temos um `toString` incrementado. Vou pular o comentário dele por enquanto.

Nas linhas de 29 a 32 temos um método `setNome`. Vamos ver algumas coisas sobre esse método:

1. Como todo bom método ele tem visibilidade `public`.
2. O retorno dele é `void`, já que esse método *faz alguma coisa* e não *devolve algum valor*.
3. O nome do método é formado pela junção da palavra `set` com o nome do atributo.
4. O método recebe um parâmetro, do mesmo tipo do atributo. Esse será o valor colocado no atributo.

Esse método faz parte de uma categoria especial de métodos chamada *métodos set*. Sempre que você tem um atributo que pode ter o seu valor alterado, você cria um método *set*. Você pode estar se perguntando: "se o atributo pode ser alterado, não bastaria ele ser público?". A resposta

é NÃO. Atributos são privados. Isso pode parecer estranho, mas seguir essa regra lhe permitirá construir programas muito mais confiáveis[1].

Olhe com atenção a linha 31. Ela tem uma estrutura estranha. Veja que o método recebeu um parâmetro chamado `nome`, mas a classe também tem um atributo `nome`. Então se a linha fosse:

```
nome = nome;
```

ficaria confuso para o Java saber quem é o parâmetro e quem é o atributo. Então quando fazemos

```
this.nome = nome;
```

Estamos dizendo ao Java: "meu atributo `nome` recebe o valor do parâmetro `nome`" e resolvemos a ambiguidade.

Nós já sabemos o que significa a palavra `this` e ela aparece em vários lugares na classe, mas ela só é obrigatória na linha 32 e na 41, porque existem ambiguidades do tipo que acabamos de ver. Nos outros lugares ela aparece para melhorar a leitura do programa. Essa é uma "boa prática de programação" e como nós somos programadores do bem, vamos usá-la.

Nas linhas que vão de 34 a 37 temos um método `getNome`. Ele é o irmão-oposto do `setNome`. Vamos ver algumas coisas sobre esse método:

1. Como todo bom método ele tem visibilidade `public`.
2. O nome do método é formado pela junção da palavra `get` com o nome do atributo.
3. O retorno dele é do mesmo tipo do atributo a que se refere, nesse caso, `String`
4. O método não recebe parâmetro, já que ele é um método que *retorna alguma coisa* e não um método que *faz alguma coisa*.

Esse método também faz parte de uma categoria especial de métodos chamada *métodos get*. Sempre que você tem um atributo que pode ter o seu valor lido, você cria um método *get*.

Métodos *gets e sets* frequentemente aparecem em pares, mas isso não é uma regra.

Nas linhas 39 a 42 temos o método *set* para o atributo `ativo`.

Nas linhas 44 a 47 temos um outro método especial chamado `isAtivo`. Esse método é essencialmente a mesma coisa que um método *get*, porém para atributos do tipo `boolean`, nós não usamos *get + nome_do_atributo*, ao invés disso nós usamos *is + nome_do_atributo*.

Parece confuso, mas se você lembrar das aulinhas de inglês sobre o verbo "to be", você consegue entender que *isAtivo* quer dizer *está ativo?*, o que faz todo o sentido. Isso será usado lá no método `toString`, na linha 21, para definir o que deverá ser impresso.

Agora, vamos comentar o `toString`. Veja que sempre que ele utiliza valores de atributos do próprio objeto, ele está utilizando os métodos *get, set e is*. Você verá a vantagem disso no Desafio...

O resto do programa é bem tranquilo, mas observe que temos nas linhas de 49 a 52 um `getSalario`, mas não temos um `setSalario` no programa. Tudo bem.

Alguns atributos, como o `salario`, têm regras de alteração complexas. No nosso exemplo existe uma regra para definir o salário inicial e uma outra para reajustar o salário. Então isso deve ser feito por métodos próprios, e não por um método *set* genérico.

Agora vamos entender o nome desse capítulo. Sem que você percebesse, você acabou de aprender um conceito bem bacana da orientação a objetos: **Encapsulamento**[2].

Podemos defini-lo como: "Um jeito de fazer as classes, colocando os atributos privados e escrevendo métodos que controlem o acesso aos dados."

[1] Além disso, o padrão *gets e sets* é usado para fazer uma coisa chamada *beans*. Isso é uma coisa obscura e complexa que você irá aprender quando for um programador adulto.

[2] Toda vez que dou aula desse assunto, algum aluno acaba chamando de "Encapsulação", o que sempre rende boas risadas na sala de aula.

E é exatamente isso que fizemos na linha 6 do programa principal. Compare a linha 6 da classe `SalarioFuncionario` deste capítulo com a mesma linha 6 da mesma classe no capítulo anterior. No anterior fazíamos uma atribuição direta do valor do nome ao atributo, pois ele era público. Agora, para alterar o atributo `nome` do objeto `f`, precisamos invocar seu método `setNome` passando como parâmetro para ele o valor do nome que desejamos atribuir.

Show de bola!

Desafios para estudo

Vamos ver o encapsulamento em ação.

1. Crie um novo atributo chamado `sobrenome`. Não preciso dizer que ele deve ser `private`.
2. Crie um método `setSobrenome` usando o que aprendemos, mas não crie o `getSobrenome`.
3. Altere o método `getNome` para retornar `nome + ‘‘ ’’ + sobrenome`.
4. Execute o programa principal, dando um sobrenome ao funcionário.

Você consegue observar que seu programa, incluindo o `toString` funcionou de um jeito atualizado, sem que você tivesse que corrigir muitas coisas?

Capítulo 6

Associação - Classes que têm outras classes

No capítulo de hoje vamos revisitar nosso exemplo de controle de livros. Além disso, não teremos apenas dois programas. Dessa vez serão três!

Primeiro crie a classe `Editora`.

```
1  public class Editora
2  {
3    private String nome;
4    private String cidade;
5    private String uf;
6
7    public String toString()
8    {
9      return nome + ":"+ cidade + ", " + uf;
10   }
11
12   public void setNome(String nome)
13   {
14     this.nome = nome;
15   }
16
17   public String getNome()
18   {
19     return this.nome;
20   }
21
22   public void setCidade(String cidade)
23   {
24     this.cidade = cidade;
25   }
26
27   public String getCidade()
28   {
29     return cidade;
30   }
31
32   public void setUf( String uf)
33   {
34     this.uf = uf;
35   }
36
37   public String getUf()
38   {
39     return uf;
40   }
41 }
```

Agora, atualize a classe `Livro`. Eu destaquei as mudanças.

```
1  public class Livro
2  {
3    String titulo;
4    private Editora editora; // linha modificada
5    int anoPublicacao;
6    int paginas;
7
8    // Modificado a partir daqui
```

```
9   public void setEditora(Editora editora)
10  {
11    this.editora = editora;
12  }
13
14  public Editora getEditora()
15  {
16    return this.editora;
17  }
18
19  public String geraReferenciaBibliografica()
20  {
21    return titulo + ", " + paginas +" pgs. Publicado por: " + this.getEditora().
      getNome() + " em " + anoPublicacao;
22  }
23
24 }
```

E por fim, a classe de controle. Como foram muitas modificações, talvez valha a pena você digitar novamente.

```
1  public class ControleLivro
2  {
3    public static void main(String[] args) {
4      Livro lv = new Livro();
5      Editora ed = new Editora();
6
7      ed.setNome( "Edição independente");
8      ed.setCidade("Brasília");
9      ed.setUf("DF");
10
11     lv.titulo = "Aprendendo Java na marra";
12     lv.setEditora( ed );
13     lv.anoPublicacao = 2015;
14     lv.paginas = 152;
15
16     System.out.println( lv.geraReferenciaBibliografica() );
17   }
18 }
```

O que você deve ver

```
Aprendendo Java na marra, 152 pgs. Publicado por: Edição independente:Brasília, DF
    em 2015
```

A classe `Editora` não tem nenhum mistério para você a essa altura.

Vamos dar uma olhada na classe `Livro`.

Na linha 4 nós temos um atributo do tipo `Editora`! Isso não é incrível? Um atributo é um objeto complexo, e não um tipo primitivo. O que eu estou dizendo é que o objeto do tipo `Livro` terá em seus atributos um outro objeto do tipo `Editora`.

Esse tipo de relação entre dois objetos nós chamamos de "associação". Definindo associação: *"associação é uma relação estrutural entre objetos onde, dado um objetos, podemos navegar para outro objeto relacionado."*.

Não queime a cabeça com essa definição. Você usará muito esse negócio e logo isso se tornará natural.

Como esse é um atributo, ele é privado e, da linha 9 até a 17, temos os métodos *get e set* para ele.

Da linha 19 até a 22 nós temos o método `geraReferenciaBibliografica()`. Ele está um pouco mais incrementado do que vimos em um capítulo anterior e tem uma coisa bem interessante aqui.

Observe a linha 21, onde está escrito `this.getEditora().getNome()`. Essa é uma construção com três partes unidas por dois pontos da seguinte forma `objeto.primeiroMetodo.segundoMetodo`. Vamos explicar o que esta acontecendo aqui bem devagar.

O *objeto* está representado por `this`, então ele é um objeto do tipo `livro`. Qualquer objeto desse tipo tem um método `getEditora()`, por isso a parte `this.getEditora()` funciona.

Se você observar o método `getEditora()` você verá que ele retorna um objeto do tipo `Editora`. Nós sabemos que qualquer objeto *Editora* tem um método `getNome()`. Portanto, essa construção está dizendo "objeto livro (`this`), pegue o seu objeto editora (`getEditora()`) e depois pegue o nome dessa editora (`getNome()`)".

A classe `ControleLivro` é bem tranquila.

Nas linhas 5 a 9 nós criamos um objeto `ed` do tipo `Editora` e setamos[1] seus valores.

Eu tenho que ter esse objeto pronto para poder associá-lo ao objeto `lv`. Isso acontece na linha 12. Antes da execução da linha 12 o objeto `lv` já estava criado (linha 4) e já tinha um título (linha 11), mas estava sem editora. Ou seja, sua referencia para editora era nula (`null`). A chamada na linha 12 invoca o método `setEditora()` que recebe uma Editora "pronta"(Criada, linha 5, e preenchida, linhas 7 a 9) e associa à referência de editora interna do objeto `lv`.

Na linha 16 nós imprimimos a referência do livro.

Desafios para estudo

Hoje os "Desafios" serão mais *trabalhosos* que desafiadores.

A classe `Livro` foi escrita originalmente antes de aprendermos sobre encapsulamento. Corrija isso.

1. Coloque todos os atributos como `private`.
2. Crie os métodos *gets e sets* para os atributos.
3. Finalmente, remova, temporariamente, a linha 12 do ControleLivro.java e execute o código. Observe o comportamento do programa e a linha que vai gerar erro. Entende o que está acontecendo? [2]

[1] Eu sei que essa palavra não existe em português, mas lembre-se que aqui nós falamos a língua dos programadores. Bem vindo à irmandade!

[2] Apesar de criar a Editora na linha 5 e preenchê-la nas linhas 7 a 9, ao fazer o que o desafio sugere você está deixando de associar essa editora ao livro. Assim, quando o livro tenta gerar sua referencia bibliográfica, a sua editora esta `null` e, acontece o erro.

Capítulo 7

Exercitanto Associação - Uma estante que só cabe um livro

Neste capítulo, vamos exercitar um pouco mais o que aprendemos sobre classes e objetos. Você vai escrever uma classe chamada `Estante`, que representa uma estante de livros em uma biblioteca.

Veja que coisa legal: a `Estante` tem um livro[1], e esse livro é a classe que você já escreveu. Espero que você esteja colocando todos os programas na mesma pasta, senão isso não vai funcionar.

```
1  public class Estante
2  {
3    private int numero;
4    private Livro livro;
5
6    public void setNumero(int numero)
7    {
8      this.numero = numero;
9    }
10
11   public int getNumero()
12   {
13     return this.numero;
14   }
15
16   public void setLivro(Livro livro)
17   {
18     this.livro = livro;
19   }
20
21   public Livro getLivro()
22   {
23     return this.livro;
24   }
25
26   public String obtemLivros()
27   {
28     return "Estante número: " + getNumero() + "\nLivro:\n " + livro.
       geraReferenciaBibliografica();
29   }
30
31 }
```

Vamos mudar a classe de controle para utilizar as novas funcionalidades:

```
1  public class ControleLivro
2  {
3    public static void main(String[] args) {
4      Livro lv = new Livro();
5      Editora ed = new Editora();
6
7      ed.setNome( "Edição independente");
8      ed.setCidade("Brasília");
9      ed.setUf("DF");
10
11     lv.setTitulo("Aprendendo Java na marra");
12     lv.setEditora( ed );
```

[1] Eu sei que uma estante pode ter vários livros, mas por enquanto, nossa estante só consegue ter um. No próximo capítulo vamos consertar essa estante. Pegue seu martelo e serrote!

```
13      lv.setAnoPublicacao(2015);
14      lv.setPaginas(152);
15
16      // mudando a partir daqui
17      Estante estante = new Estante();
18
19      estante.setNumero(1);
20      estante.setLivro(lv);
21
22      System.out.println( estante.obtemLivros() );
23    }
24  }
```

O que você deve ver

```
Estante número: 1
Livro:
 Aprendendo Java na marra, 152 pgs. Publicado por: Edição independente em 2015
```

Vamos dar uma olhada na classe `Estante`.

Nas linhas 3 e 4 definimos seus atributos: um número, representando o número dessa estante, e um livro que será colocado nela.

As linhas de 6 até 24 temos os *gets* e *sets* e você já é mestre neles.

As linhas de 26 a 29 definem o método `obtemLivros()` que existe para... obter os livros. Olha que na linha 28 nós utilizamos um método `geraReferenciaBibliografica()` que você programou no capítulo anterior.

Se você olhar com cuidado, esse método mostra parte do objeto `estante`. Esse objeto mostra partes do seu objeto `livro` e esse objeto *livro* mostra parte do seu objeto `editora`. Andamos por três objetos em uma única linha.

Ao programar em uma linguagem orientada a objetos, esperamos que com o tempo nós construamos uma biblioteca de classes que possa ser constantemente reutilizada. Com o tempo, seu programa passa a ser um grande brinquedo de pegar-e-encaixar objetos[2].

Bem, por ora é só isso. Vá relaxar um pouco.

[2] #maisOuMenos #soQueNão #ahSeFosseFacilAssim

Capítulo 8

Relembrando Vetores

Nesse capítulo vamos relembrar e fazer um aquecimento no assunto “Vetores”.

Como você se lembra, um “vetor” é um conjunto de variáveis acessadas pelo mesmo identificador mais um índice.

Antes de irmos ao programa, observe que a classe *Livro* que está em `Livro.java` vem sendo incrementada programa após programa, da mesma forma que outras classes usadas como exemplo, mas a classe de controle em `ControleLivro.java` é meio descartável, e nós vamos mudando ela de acordo com o que precisamos em cada exemplo.

Então vamos ao que interessa. A classe `Livro` não muda nada. Deixe ela do jeito que está. Vamos usá-la de um jeito novo.

Reescreva sua classe de controle da seguinte forma:

```
1  import java.util.Scanner;
2
3  public class ControleLivro
4  {
5    public static void main(String[] args) {
6      Livro[] livros = new Livro[3];
7
8      livros[ 0 ] = new Livro();
9      livros[ 1 ] = new Livro();
10     livros[ 2 ] = new Livro();
11
12     Editora ed = new Editora();
13
14     ed.setNome( "Edição independente");
15     ed.setCidade("Brasília");
16     ed.setUf("DF");
17
18     livros[ 0 ].setTitulo("Aprendendo Java na marra");
19     livros[ 0 ].setEditora( ed );
20     livros[ 0 ].setAnoPublicacao(2015);
21     livros[ 0 ].setPaginas(152);
22
23     livros[ 1 ].setTitulo("Aprendendo Java na marra - Livro II");
24     livros[ 1 ].setEditora( ed );
25     livros[ 1 ].setAnoPublicacao(2015);
26     livros[ 1 ].setPaginas(200);
27
28     livros[ 2 ].setTitulo("Aprendendo Java na marra - Livro III");
29     livros[ 2 ].setEditora( ed );
30     livros[ 2 ].setAnoPublicacao(2015);
31     livros[ 2 ].setPaginas(183);
32
33     for (Livro l:livros)
34     {
35       System.out.println( l.geraReferenciaBibliografica() + "\n");
36     }
37
38   }
39 }
```

O que você deve ver

```
Aprendendo Java na marra, 152 pgs. Publicado por: Edição independente em 2015

Aprendendo Java na marra – Livro II, 200 pgs. Publicado por: Edição independente em
    2015

Aprendendo Java na marra – Livro III, 183 pgs. Publicado por: Edição independente em
    2015
```

Na linha 6 declaramos e definimos uma variável chamada `livros`. Veja que ela é um vetor, já que tem os colchetes. Mas diferente de vetores de Strings, que estávamos acostumados a fazer, esse é um vetor de livros, ou seja, é um vetor de objetos[1].

Ao usarmos a palavra `new` estamos inicializando o vetor com três espaços que poderão conter `livro`, indo do índice 0 até o índice 2.

Nas linhas 8, 9 e 10 nós criamos os objetos do tipo `livro` em cada um dos espaços do vetor. Vale uma recaptulação:

- O `new` da linha 6, cria um vetor com três espaços que conterão `livro`, mas que ainda estão vazios;
- O `new` das linhas de 8 a 10 criam um objeto `livro` cada uma, colocando nos espaços do vetor.

Nas linhas de 12 a 16 nós criamos uma editora. Isso mesmo, todos os nossos livros serão da mesma editora nesse exemplo.

Nas linhas de 18 a 21 nós vamos colocar os dados no primeiro livro, aquele que se encontra na posição `livros[ 0 ]`, utilizando os métodos `set` da classe `livro`.

Nas linhas de 23 até a 31 fazemos a mesma coisa com os outros dois objetos.

Nas linhas de 33 a 36 temos o comando `for` que navega no vetor lendo cada objeto e chamando o método `geraReferenciaBibliografica()` de cada um. Lembra-se desse tipo de comando `for` ? A leitura dele pode ser feita assim: "Para cada `Livro` (que a cada rodada eu chamarei de `l`) armazenado na lista `livros` execute o código a seguir, entre as chaves.".

Ok, tudo relembrado. Chega de moleza e vamos em frente. As coisas ficarão cada vez mais interessantes.

[1] Talvez já seja hora de abrir seus olhos para algo: uma String é uma classe, isto é, um tipo complexo. Então, no fim das contas, não estamos fazendo nada que você já não tenha feito.

Capítulo 9

Introdução a Coleções - Reformando a estante para caber muitos livros

Vamos pegar o que vimos sobre vetor no capítulo anterior e consertar aquela nossa velha estante que só cabia um livro.

No programa abaixo, fizemos várias alterações. Então preste atenção ao evoluir a classe em `Estante.java`.

```
1  public class Estante
2  {
3    private int numero;
4    private Livro[] livros ;
5
6    public void setNumero(int numero)
7    {
8      this.numero = numero;
9    }
10
11   public int getNumero()
12   {
13     return this.numero;
14   }
15
16   public void setLivros(Livro[] livros)
17   {
18     this.livros = livros;
19   }
20
21   public Livro[] getLivros()
22   {
23     return this.livros;
24   }
25
26   public String obtemLivros()
27   {
28     String texto = "Estante número: " + getNumero() + "\nLivro:\n";
29     for (Livro l:livros)
30     {
31       texto += l.geraReferenciaBibliografica() + "\n";
32     }
33
34     return texto;
35   }
36
37 }
```

Agora, atualize o nossa classe "descartável":

```
1  import java.util.Scanner;
2
3  public class ControleLivro
4  {
5    public static void main(String[] args) {
6      Livro[] livros = new Livro[5];
7      Estante estante = new Estante();
8
9      for (int i=0;i<5;i++){
10       Scanner teclado = new Scanner(System.in);
```

```
11
12        Livro lv = new Livro();
13        Editora ed = new Editora();
14
15        System.out.println("Digite o título do livro: ");
16        String titulo = teclado.nextLine();
17        lv.setTitulo( titulo);
18
19        System.out.println("Qual a editora?");
20        ed.setNome( teclado.nextLine());
21
22        // define dados fixos para a editora
23        ed.setCidade("Brasília");
24        ed.setUf("DF");
25
26        // define a editora do livro
27        lv.setEditora( ed );
28
29        // seta dados fixos para o livro
30        lv.setAnoPublicacao( 2015 );
31        lv.setPaginas( 152 );
32
33        // coloca o livro no vetor
34        livros[ i ] = lv;
35      }
36
37      estante.setNumero(1);
38      estante.setLivros(livros); // coloca os livros na estante
39
40      System.out.println( estante.obtemLivros() );
41    }
42 }
```

O que você deve ver

```
Digite o título do livro:
O Senhor dos Anéis - A sociedade do anel
Qual a editora?
Ed. Completa
Digite o título do livro:
O Senhor dos Anéis - As duas torres
Qual a editora?
Ed. Completa
Digite o título do livro:
O Senhor dos Anéis - O retorno do rei
Qual a editora?
Ed. Completa
Digite o título do livro:
O Hobbit
Qual a editora?
Ed. Completa
Digite o título do livro:
O Silmarillion
Qual a editora?
Ediouro
Estante número: 1
Livro:
 O Senhor dos Anéis - A sociedade do anel, 152 pgs. Publicado por: Ed. Completa em
    2015
O Senhor dos Anéis - As duas torres, 152 pgs. Publicado por: Ed. Completa em 2015
O Senhor dos Anéis - O retorno do rei, 152 pgs. Publicado por: Ed. Completa em 2015
O Hobbit, 152 pgs. Publicado por: Ed. Completa em 2015
O Silmarillion, 152 pgs. Publicado por: Ediouro em 2015
```

Olhemos para a estrela do capítulo, a classe *Estante*.

Após as declarações iniciais da classe, na linha 3 temos o atributo `numero` e na linha 4 temos o atributo `livros`. Observe que o tipo dele não é simplesmente `Livro`, mas `Livro[]`, ou seja, ele é um vetor de livros. A nossa estante agora consegue ter vários livros. Nós também mudamos o nome do atributo de `livro` para `livros` porque agora ele é um conjunto de valores.

As linhas de 6 a 14 dispensam explicações. São o *getter* e o *setter* do atributo `numero`.

Na linha 16 começamos o método `setLivros()`. Como `livros` é um vetor, o método *set*

tem que ser alterado. Ele agora recebe um outro vetor como parâmetro.

Na linha 21 começamos o método `getLivros()`. Como `livros` é um vetor, o método *get* tem que ser alterado. Ele agora devolve o vetor de livros como retorno.

Da linha 26 até a 35 temos o método `obtemLivros()`. Ele agora tem que caminhar por todo o vetor para ver os livros da estante. Isso acontece no laço `for` que começa na linha 29 e você já consegue se virar para entender a lógica.

Antes de terminarmos, que tal aprender uma coisa nova?

Nós chamamos o vetor `livros` de "um conjunto de objetos". Isso tem um nome, se chama "coleção". Então, a partir de agora iremos chamar um conjunto de objetos de **coleção**. Mais nomes sofisticados para impressionar o pessoal!

Não vamos gastar muito tempo com a nova versão da classe `ControleLivros`, você consegue entendê-la. Ela tem um laço que lê cinco livros. Nós só perguntamos ao usuário o título do livro e o nome da editora, o resto deixamos fixo para a digitação ficar mais simples. A cada execução do laço, criamos um objeto `Livro` na linha 12 e colocamos no vetor na posição adequada (índice `i`) na linha 34.

Observe que a linha 6 cria um vetor de cinco posições que será vinculado à estante lá na linha 38. Agora nossa estante consegue conter cinco livros[1]!

A linha 40 simplesmente exibe na tela os livros da estante.

Desafios para estudo

Já que você está dando uma de marceneiro e reformando a estante, que tal perguntar para o ser humano qual o tamanho da estante que ele quer? Consegue fazer isso?

[1] Eu sei que não é grande coisa, mas se fizéssemos uma estante com 100 livros iríamos passar o resto do dia digitando títulos de livro no programa.

Capítulo 10

ArrayList, porque vetores de tamanho fixo é para os fracos

Você, a essa altura, já está dominando o uso de vetores. Mas também já deve ter visto que eles têm uma grande limitação: precisamos dizer qual é o tamanho do vetor. Infelizmente, a vida não se molda às nossas linguagens de programação e é muito comum termos coleções de tamanho indefinido.

Mas não nos desesperemos, o pessoal do Java já pensou nisso também. Martelo e serrote na mão, vamos dar mais uma reformada na estante:

```
1  import java.util.ArrayList;
2
3  public class Estante
4  {
5    int numero;
6    ArrayList<Livro> livros = new ArrayList<>();
7
8    public void setNumero(int numero)
9    {
10     this.numero = numero;
11   }
12
13   public int getNumero()
14   {
15     return this.numero;
16   }
17
18   // o método setLivros foi apagado
19
20   public ArrayList<Livro> getLivros()
21   {
22     return this.livros;
23   }
24
25   public String obtemLivros()
26   {
27
28     String texto = "Estante número: " + getNumero() + "\nLivro:\n";
29     for (Livro l:livros)
30     {
31       texto += l.geraReferenciaBibliografica() + "\n";
32     }
33
34     return texto;
35   }
36
37 }
```

Mude também a classe de controle:

```
1  import java.util.Scanner;
2
3  public class ControleLivro
4  {
5    public static void main(String[] args) {
6      //Livro[] livros = new Livro[5];
7      boolean sair = false;
8      Estante estante = new Estante();
```

```
9
10      do{
11        Scanner teclado = new Scanner(System.in);
12
13        Livro lv = new Livro();
14        Editora ed = new Editora();
15
16        System.out.println("Digite o título do livro: ");
17        String titulo = teclado.nextLine();
18        lv.setTitulo( titulo);
19
20        System.out.println("Qual a editora?");
21        ed.setNome( teclado.nextLine());
22
23        // define dados fixos para a editora
24        ed.setCidade("Brasília");
25        ed.setUf("DF");
26
27        // define a editora do livro
28        lv.setEditora( ed );
29
30        // seta dados fixos para o livro
31        lv.setAnoPublicacao( 2015 );
32        lv.setPaginas( 152 );
33
34        // coloca o livro no vetor
35        estante.getLivros().add(lv);
36
37        System.out.println("Deseja sair? (S/N)");
38
39        sair = teclado.next().toUpperCase().equals("S");
40      }while (!sair);
41
42      estante.setNumero(1);
43      //estante.setLivros(livros); // coloca os livros na estante
44
45      System.out.println( estante.obtemLivros() );
46    }
47  }
```

O que você deve ver

```
Digite o título do livro:
O Guia do Mochileiro das Galáxias
Qual a editora?
Sextante
Deseja sair? (S/N)
n
Digite o título do livro:
O restaurante no fim do universo
Qual a editora?
Sextante
Deseja sair? (S/N)
n
Digite o título do livro:
A vida, o universo e tudo mais
Qual a editora?
Sextante
Deseja sair? (S/N)
n
Digite o título do livro:
Até mais, e obrigado pelos peixes!
Qual a editora?
Sextante
Deseja sair? (S/N)
s
Estante número: 1
Livro:
O Guia do Mochileiro das Galáxias, 152 pgs. Publicado por: Sextante em 2015
O restaurante no fim do universo, 152 pgs. Publicado por: Sextante em 2015
A vida, o universo e tudo mais, 152 pgs. Publicado por: Sextante em 2015
Até mais, e obrigado pelos peixes!, 152 pgs. Publicado por: Sextante em 2015
```

Você deve ter percebido que as alterações tornaram a classe *Estante* mais simples!

A principal mudança está na linha 6. Nós substituímos o antigo vetor de livros `Livro[] livro;` por uma estrutura mais sofisticada chamada `ArrayList`. Precisaremos gastar um pouco de teoria agora, antes de prosseguirmos no programa.

O Java provê uma série de estruturas de coleção de objetos. Cada uma com suas características próprias. Algumas são coleções que têm itens identificados por uma espécie de chave, outra são coleções ordenadas e por aí vai. O `ArrayList` é uma dessas *classes de coleção*. Nós não vamos nos alongar em todas as classes de coleção nesse livro, mas se você quiser pesquisar, todas são derivadas de `Collection` e você pode achar tudo o que quiser no `Javadoc`.

Voltando ao `ArrayList`, ele é basicamente um vetor com tamanho variável e cheio de funcionalidades úteis. Um objeto `ArrayList` é um objeto que tem objetos. Eu posso colocar qualquer tipo de objeto no ArrayList, mas tem um porém. Eu preciso decidir qual tipo de objeto vou colocar e depois eu não posso mais mudar. É mais ou menos como sua mãe decidindo qual a sua gaveta de meias. Você poderia ter guardados suas camisetas ali, mas depois que sua mãe decide que serão apenas meias, nada de camisetas.

No nosso caso, nosso `ArrayList` guarda objetos do tipo `Livro` e é exatamente o que quer dizer a palavra *Livro* entre menor-e-maior na linha 6.

```
ArrayList<Livro>
```

Isto é o equivalente a dizer "esse ArrayList guardará objetos Livro".

Depois eu dou o nome para o *ArrayList* (nesse caso `livros`) e depois eu inicializo com `= new ArrayList<>()`. Veja que eu tenho novamente um menor-e-maior só que agora vazio. É como se eu dissesse "Crie um novo ArrayList que guardará o que eu já disse antes".

Continuando o programa, nós apagamos o método `setLivros()` porque não precisaremos mais dele.

Na linha 20 nós mudamos o retorno do método `getLivros()`. Agora ao invés de retornar um vetor, ele retorna um `ArrayList` de `Livro`.

E nessa classe foi só isso. Vamos ver as brincadeiras que fizemos na classe de controle.

A linha 6 está comentada, para você ver que não criaremos mais o vetor.

Na linha 7 criamos uma variável `boolean` para controlar o laço e na linha 8 criamos a estante.

Da linha 10 até a linha 40 temos um laço que controla a leitura dos dados. Como nós não precisamos nos preocupar com o tamanho do vetor, o ser humano poderá digitar quantos livros ele quiser. Então, ao invés de um `for`, temos um `do-while`.

Da linha 11 até a linha 32 não temos nada de novo.

Na linha 35 fazemos uma coisa legal. Nós pegamos o `ArrayList` do objeto `estante` e adicionamos o livro (`lv`) usando o método `add()`. E que método é esse? Esse é o método usado para colocar algo dentro da coleção. O método `add(` *`objeto`*`)` adiciona o objeto no fim do `ArrayList`.

Na linha 39 nós fizemos uma construção bem bacaninha (algumas linhas de código realmente nos dão orgulho). A variável booleana `sair` é igual ao resultado da comparação do maiúsculo do texto digitado pelo usuário com a letra "S". Entendeu?

O resto do programa você domina.

Não temos desafio nesse capítulo, então, passe logo para o outro. Não vamos descansar agora.

Capítulo 11

Fazendo coisas interessantes com o ArrayList

Vamos trabalhar um pouco mais com `ArrayList`. Nesse capítulo só iremos trabalhar com a classe de controle. Para facilitar eu assinalei onde as alterações foram incluídas:

```
1  import java.util.Scanner;
2  import java.util.ArrayList; // incluído
3
4  public class ControleLivro
5  {
6    public static void main(String[] args) {
7      boolean sair = false;
8      Estante estante = new Estante();
9
10     do{
11       Scanner teclado = new Scanner(System.in);
12
13       Livro lv = new Livro();
14       Editora ed = new Editora();
15
16       System.out.println("Digite o título do livro: ");
17       String titulo = teclado.nextLine();
18       lv.setTitulo( titulo);
19
20       System.out.println("Qual a editora?");
21       ed.setNome( teclado.nextLine());
22
23       // define dados fixos para a editora
24       ed.setCidade("Brasília");
25       ed.setUf("DF");
26
27       // define a editora do livro
28       lv.setEditora( ed );
29
30       // seta dados fixos para o livro
31       lv.setAnoPublicacao( 2015 );
32       lv.setPaginas( 152 );
33
34       // coloca o livro no vetor
35       estante.getLivros().add(lv);
36
37       System.out.println("Deseja sair? (S/N)");
38
39       sair = teclado.next().toUpperCase().equals("S");
40     }while (!sair);
41
42     estante.setNumero(1);
43
44     System.out.println( estante.obtemLivros() );
45
46     /* ------------------------
47        Incluindo a partir daqui
48        ------------------------ */
49     Scanner teclado = new Scanner(System.in);
50     ArrayList<Livro> livros = estante.getLivros();
51
52     System.out.println("Quer o livro de qual posição (começando de 0)?");
53     int posicao = teclado.nextInt();
54
```

```
55      if (posicao >= livros.size())
56      {
57        System.out.println("Posição inválida");
58      } else {
59        Livro l = livros.get( posicao );
60        System.out.println( l.geraReferenciaBibliografica());
61
62
63        System.out.println("Deseja excluir o livro? (S/N)");
64
65        if ( teclado.next().toUpperCase().equals("S"))
66        {
67          livros.remove(posicao);
68          System.out.println("Livro removido. Nova lista:\n");
69          System.out.println( estante.obtemLivros() );
70        }
71      }
72
73    }
74  }
```

O que você deve ver

```
... parte da digitação foi retirada...

Estante número: 1
Livro:
O Senhor dos Anéis - A sociedade do anel, 152 pgs. Publicado por: Ed. Completa em
    2015
O Senhor dos Anéis - As duas torres, 152 pgs. Publicado por: Ed. Completa em 2015
O Senhor dos Anéis - O retorno do rei, 152 pgs. Publicado por: Ed. Completa em 2015
O Hobbit, 152 pgs. Publicado por: Ed. Completa em 2015
O Silmarillion, 152 pgs. Publicado por: Ediouro em 2015

Quer o livro de qual posição (começando de 0)?
2
O Senhor dos Anéis - O retorno do rei, 152 pgs. Publicado por: Ed. Completa em 2015
Deseja excluir o livro? (S/N)
s
Livro removido. Nova lista:

Estante número: 1
Livro:
O Senhor dos Anéis - A sociedade do anel, 152 pgs. Publicado por: Ed. Completa em
    2015
O Senhor dos Anéis - As duas torres, 152 pgs. Publicado por: Ed. Completa em 2015
O Hobbit, 152 pgs. Publicado por: Ed. Completa em 2015
O Silmarillion, 152 pgs. Publicado por: Ediouro em 2015
```

Na linha 2 nós incluímos a importação do `ArrayList`. Depois disso nada muda até a linha 44. A partir dali começa a nossa brincadeira.

Na linha 49 nós declaramos o objeto `teclado` do tipo `Scanner`. "Espere aí!" dirá você, "nós já não haviamos declarado teclado lá na linha 11"? Sim e não. Sim, nós declaramos, mas como isso foi feito dentro do bloco do comando `while` essa variável não existe fora do bloco, lembra? Nós já falamos sobre escopo de variáveis no primeiro livro[1].

Na linha 50 criamos um `ArrayList` que recebe a coleção do objeto `estante`. Esse não é um passo obrigatório, mas simplifica o código porque ao invés de termos que ficar escrevendo `estante.getLivros()` que resulta no ArrayList, nós escrevemos apenas `livros`. O resultado é o mesmo e o programa fica mais limpo e fácil de entender[2].

Linha 52 e 53 são tediosas...

Na linha 55 temos `livros.size()` que é a chamada ao método `size()` do arraylist. Esse método, como você já deve ter deduzido, retorna o tamanho da coleção e o que estamos vendo no `if` é se a posição escolhida está dentro do intervalo possível.

[1] Uma outra solução seria mover a declaração do `teclado` da linha 11 para a linha 8, por exemplo, mas eu quis manter o corpo da classe sem alteração.

[2] Programadores gostam de chamar códigos limpos e fáceis de "elegantes". Somos um povo muito estranho, não?

Linha 56, chata... 57, simples... 58, blah....

Na linha 59 temos `livros.get(posicao)`. O método `get()` recebe um número inteiro como parâmetro e retorna o objeto que está naquela posição.

Você consegue entender até a linha 66.

Na linha 67 nós usamos `livros.remove(posicao)`. O método `remove()` recebe um inteiro e remove o objeto da coleção deixando `null` no lugar.

O resto é tranquilo.

Resumindo o que vimos de novo aqui:

- Método `size()`: retorna o tamanho do arraylist;
- Método `get( <int> )`: retorna o objeto na posição informada no parâmetro;
- Método `remove(<int>)`: remove o objeto do arraylist na posição informada no parâmetro;

Desafios para estudo

Existem muitas outras funções no `ArrayList`. Nosso objetivo aqui não é fazer um guia completo de uso da coleção. Você pode consultar a especificação no Javadoc [3].

Vamos fazer o seguinte.

1. Consultando o Javadoc, descubra qual o método permite a adição de um elemento em uma posição específica.
2. Deixe o usuário digitar um novo objeto e inclua na posição do objeto que foi excluído.
3. Peça para o usuário digitar o nome de um livro.
4. Descubra qual o método faz a busca dentro do arraylist e diga ao usuário se o livro existe ou não na estante.

[3] http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html

Capítulo 12

Construtores - O que são? O que comem? Descubra hoje!

Vamos começa um novo assunto, com um novo exemplo. Chega de livros e estantes por enquanto.

E eu quero tentar uma coisa nova. Vou te lançar em um vôo solo: ao invés de te dar a classe, vou pedir para você criar uma do zero (ela está no fim do capítulo, mas nada de colar).

Crie uma classe chamada Crianca, com um atributo nome do tipo String e um atributo dataDeNascimento do tipo Date. Faça a importação de java.util.Date, para isso. Crie os gets e os sets.

Agora crie a seguinte classe de controle:

```
1  public class CriancaNascendo{
2    public static void main(String[] args) {
3      Crianca crianca = new Crianca();
4
5      if (crianca.getNome() == null) {
6        System.out.println("Meu Deus! Uma criança sem nome!");
7      } else {
8        System.out.println("A criança se chama " + crianca.getNome() + " e nasceu em "
       + crianca.getDataDeNascimento());
9      }
10   }
11 }
```

Será que a sua classe `Crianca` ficou certa? Dê uma espiadinha ali no final só para garantir.

O que você deve ver

```
Meu Deus! Uma criança sem nome!
```

Esse programa é bem simples. Só vamos destacar que na linha 5 nós verificamos se o nome é `null` e mostramos a mensagem.

Neste momento, creio que você está revoltado e dizendo "que absurdo! Ele esqueceu de preencher os atributos do objeto `crianca`!". Na verdade eu não esqueci, eu propositalmente deixei assim pois "todos os meus movimentos são friamente calculados".

Estamos simulando um erro que pode ocorrer em um grande programa. Um objeto é criado, algum atributo obrigatório não é preenchido e um erro explode lá na frente.

Não seria bom que houvesse uma maneira de garantir que os atributos obrigatórios sempre fossem preenchidos quando um novo objeto é criado? E é claro que há!

Vamos fazer uma pequena alteração na classe criança. Inclua o seguinte, logo depois da declaração dos atributos[1].

```
  public Crianca(String nome, Date dataDeNascimento){
```

[1]Na verdade, a posição não importa, mas esse tipo de método costuma ser o primeiro da classe por uma questão de boa prática.

```
    this.setNome( nome );
    this.setDataDeNascimento( dataDeNascimento );
  }
```

Este método é um método especial, chamado de "método construtor". Vamos analisar as suas características:

1. Ele é um método público;
2. Ele não tem tipo de retorno! Nem `void`, nem qualquer outro! Isso é muito estranho.
3. O nome dele é exatamente igual ao nome da classe, incluíndo as letras maiúsculas e minúsculas.

Para que serve um método construtor?

Um método construtor serve para criar novos objetos seguindo as regras que você determinar.

E qual a regra que esse nosso construtor está assegurando? "Todo objeto `Crianca` que for criado, terá que fornecer o nome e a data de nascimento".

O Java já te fornecia um método construtor antes, que é o construtor default. Na linha 3 da classe de controle você pode vê-lo:

```
Crianca crianca = new Crianca();
```

Toda classe tem um construtor default. Ele não recebe parâmetros e não assegura nenhuma regra.

Agora, execute seu programa. O que aconteceu? Um erro de compilação!

```
CriancaNascendo.java:3: error: constructor Crianca in class Crianca cannot be
    applied to given types;
    Crianca crianca = new Crianca();
                      ^
  required: String,Date
  found: no arguments
  reason: actual and formal argument lists differ in length
1 error
```

Quando você cria um construtor, o construtor default deixa de existir.

Isso faz todo sentido não faz? Se você criou uma regra de construção, ela tem que ser seguida!

Substitua a linha 3 da classe de controle por essas duas:

```
Date data = new Date(2015 - 1900, 08, 20);
Crianca crianca = new Crianca("Marco Antonio Ferreira", data);
```

O que você deve ver agora

```
A criança se chama Marco Antonio Ferreira e nasceu em Sun Sep 20 00:00:00 BRT 2015
```

Tudo funcionando! Eu vi que a data está feia. Depois arrumamos isso. Por hoje já foi muita coisa. Vá tomar uma coca gelada.

Classe Crianca

```
1 import java.util.Date;
2
3 public class Crianca {
4   private String nome;
5   private Date dataDeNascimento;
6
7   public Crianca(String nome, Date dataDeNascimento){
8     this.setNome( nome );
9     this.setDataDeNascimento( dataDeNascimento );
```

```
10    }
11
12    public String getNome(){
13      return nome;
14    }
15
16    public void setNome(String nome){
17      this.nome = nome;
18    }
19
20    public Date getDataDeNascimento(){
21      return dataDeNascimento;
22    }
23
24    public void setDataDeNascimento(Date data){
25      this.dataDeNascimento = data;
26    }
27  }
```

Capítulo 13

Sobrecarregando seus métodos

Se você é ligeiramente obsessivo-compulsivo, ficou arrepiado com aquela data desformatada do capítulo anterior. Vamos resolver isso e de quebra aprender mais duas coisas.

Altere a classe `Crianca` de acordo com o programa abaixo (é só um import e um método a mais):

```
1  import java.util.Date;
2  import java.text.SimpleDateFormat; // inclua esse import
3
4  public class Crianca {
5    private String nome;
6    private Date dataDeNascimento;
7
8    public Crianca(String nome, Date dataDeNascimento){
9      this.setNome( nome );
10     this.setDataDeNascimento( dataDeNascimento );
11   }
12
13   public String getNome(){
14     return nome;
15   }
16
17   public void setNome(String nome){
18     this.nome = nome;
19   }
20
21   public Date getDataDeNascimento(){
22     return dataDeNascimento;
23   }
24
25   public void setDataDeNascimento(Date data){
26     this.dataDeNascimento = data;
27   }
28
29   // novo método
30   public String getDataDeNascimento( String formato){
31     SimpleDateFormat formatoData = new SimpleDateFormat( formato );
32
33     String dataFormatada = formatoData.format(this.getDataDeNascimento());
34
35     return dataFormatada;
36   }
37 }
```

Agora, altere a classe de controle. Observer que é apenas um detalhes na linha 12:

```
1  import java.util.Date;
2
3  public class CriancaNascendo{
4    public static void main(String[] args) {
5      Date data = new Date(2015 - 1900, 8, 20);
6      Crianca crianca = new Crianca("Marco Antonio Ferreira", data);
7
8      if (crianca.getNome() == null) {
9        System.out.println("Meu Deus! Uma criança sem nome!");
10     } else {
11       // Essa linha foi alterada
12       System.out.println("A criança se chama " + crianca.getNome() + " e nasceu em "
       + crianca.getDataDeNascimento( "dd/MM/YYYY"));
```

```
13      }
14    }
15 }
```

O que você deve ver

```
A criança se chama Marco Antonio Ferreira e nasceu em 20/09/2015
```

A data está arrumada, mas vamos ver como fizemos isso. A principal mágica acontece no novo método da classe `Crianca`.

Na linha 30 temos:

```
public String getDataDeNascimento( String formato)
```

Temos dois métodos chamados `getDataDeNascimento`? Pode isso Arnaldo? Sim! Podemos ter quantos métodos quisermos com o mesmo nome, *desde que tenham parâmetros com tipos diferentes ou quantidade diferentes de parâmetros.* Isso nós chamamos de "sobrecarga de métodos". Vamos falar de novo, só para fixar:

Sobrecarga de métodos é quando temos mais de um método, com o mesmo nome e com parâmetros diferentes.

Esse conjunto de coisas, nome do método, tipo de retorno e parâmetros, nós chamamos de "assinatura do método".

Na nosso classe `Crianca` nós podemos pegar a data de duas maneiras: como uma data desformatada ou como uma String formatada. Isso é bem prático e para saber qual método usar, basta usar a assinatura certa, com ou sem uma String sendo passada como parâmetro.

Trabalhar com formatação de datas no Java não é tão simples, mas depois que você pega o jeito, fica tranquilo.

Na linha 31 nós usamos uma classe chamada `SimpleDateFormat` que é fornecida pelo Java e serve exatamente para formatar datas. Aliás, você colocou o `import` no início do programa, não colocou? O que estamos fazendo é criando um objeto com um tipo específico de formato de data. O formato nós recebemos como parâmetro passado lá pela classe `CriancaNascendo` e nesse caso é "dd/MM/yyyy" onde "dd" é o dia da semana, "MM" é o mês do ano e "yyyy" é o ano com quatro dígitos. Consulte o Javadoc do `SimpleDateFormat` para ver a infinidade de formatações possíveis.

Na linha 33 nós usamos o método `format` do objeto `dataFormatada` que é nosso objeto `SimpleDateFormat`. Esse método recebe uma data do tipo `Date` e devolve uma String formatada.

Desafios para estudo

Vamos ver se você pegou o jeito da formatação da data.

1. Mostre a data de nascimento no formato americano. O resultado será "8/20/15".

Capítulo 14

Métodos estáticos não são métodos parados

A temperatura por aqui é medida na escala Celsius, onde a água congela a 0 graus e evapora a 100 graus, o que faz todo o sentido. Mas lá pelas terras do tio Sam eles usam a escala Fahrenheit, onde a água congela a 32 graus, e evapora a 212, e é completamente estúpida.

Talvez tenha sido o calor ou a secura do tempo que me levou a escrever essa introdução, mas ela tem tudo a ver com o exemplo abaixo e o que vamos aprender agora. Vamos aos programas:

```
1  public class Temperatura{
2
3    public static float celsius(float f){
4      // a fórmula de conversao de celsius para fahrenheit é ([Â°F] - 32) Ã— 5/9
5      return (f - 32) * 5 / 9;
6    }
7
8    public static float fahrenheit(float c){
9      // a fórmula de conversao de celsius para fahrenheit é [Â°C] Ã— 9/5 + 32
10     return c * 9/5 + 32;
11   }
12
13 }
```

E agora a classe de controle:

```
1  import java.util.Scanner;
2
3  public class ControleTemperatura{
4    public static void main(String[] args) {
5      Scanner teclado = new Scanner(System.in);
6
7      System.out.print("Digite a temperatura: ");
8      float temp = teclado.nextFloat();
9      System.out.println("Digite C se ela estiver em Celsius e F se Fahrenheit");
10     boolean isCelsius = teclado.next().equalsIgnoreCase( "C");
11
12     if (isCelsius){
13       float tempConvertida = Temperatura.fahrenheit( temp );
14       System.out.println("A temperatura em fahrenheit é " + tempConvertida);
15     } else {
16       float tempConvertida = Temperatura.celsius( temp );
17       System.out.println("A temperatura em Celsius é " + tempConvertida);
18     }
19   }
20 }
```

O que você deve ver

```
Digite a temperatura: 28
Digite C se ela estiver em Celsius e F se Fanhreinheit
C
A temperatura em fahrenhit é 82.4
```

A classe `Temperatura` tem dois métodos.

Da linha 3 até a 6 temos o método `celsius` que recebe um `float` com a temperatura em fahrenheit e devolve um `float` com a temperatura em celsius. Usamos `float` apenas para variar um pouco. Ele é a mesma coisa que `double` só que menor.

Da linha 8 até a 11 temos o método `fahrenheit` que recebe um `float` com a temperatura em celsius e devolve um `float` com a temperatura em fahrenheit. Tudo bem simples.

Observe que na declaração de ambos os métodos temos a palavra `static` (nas linhas 3 e 8). Já vamos explicar o que ela está fazendo aí.

O nosso programa de controle `ControleTemperatura.java` faz uma coisa simples: pega a temperatura em uma unidade e converte para outra. Não vamos explicar linha-a-linha, você já está entendendo as coisas.

O interessante aqui é que *não precisamos criar um objeto `Temperatura` para fazer a conversão.*

Em capítulos anteriores tínhamos, por exemplo, uma estante que continha livros. A estante é uma instância única, e métodos que colocam livros na estante se referem a essa instância em particular. Da mesma maneira, quando eu mostro a editora de um livro, essa é uma ação executada pelo objeto livro.

Mas converter um temperatura é uma ação geral, que não depende da existência de um objeto específico. Entendeu a diferença? Vou falar novamente, um pouco diferente:

- Existem métodos que precisam ser executados por objetos específico, por que buscam, usam, ou alteram dados desses objetos;
- Existem outras situações que precisamos executar coisas que não se relacionam com um objeto específico. A conversão de dados é um exemplo disso.

Quando um método não precisa de um objeto, chamamos esse método de **método estático** e ele é definido pela utilização da palavra `static`.

Portanto, definindo em palavras mais bonitas: *"Métodos estáticos são aqueles que existem na classe, e não em sua instância (objeto)."*

Por isso podemos fazer:

```
float tempConvertida = Temperatura.celsius( temp );
```

Sem termos feito algo como:

```
Temperatura t = new Temperatura();
```

...o que criaria um objeto.

Você já deve ter percebido que nós estamos usado `static` há muito tempo. Todo método `main` é estático. Vamos, finalmente, entender todos os componentes desse método especial:

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

1. `public` - ele é um método de visibilidade pública;
2. `static` - existe na instância da classe, o que faz todo sentido, já que ele é o primeiro método chamado e não há como criar um objeto antes disso;
3. `void` - ele não retorna valor;
4. `main` - o nome especial, indicando que é o primeiro método a ser executado;
5. `String[] args` - ele recebe um vetor de `String` como parâmetro. Esses parâmetros podem ser passados via linha de comando.

Finalmente entendemos todo o `main`!

Capítulo 15

Finalmente o `final`

Rápido, sem consultar, responda: em quantos graus fahrenheit a água congela? Não lembrou? Vamos resolver isso. Altere o programa:

```
1  public class Temperatura{
2    // inserimos essas quatro linhas
3    public final static float CONGELA_C = 0;
4    public final static float EVAPORA_C = 100;
5    public final static float CONGELA_F = 32;
6    public final static float EVAPORA_F = 212;
7
8    public static float celsius(float f){
9      // a fórmula de conversao de celsius para fahrenheit é ([Â°F] - 32) Ã— 5/9
10     return (f - 32) * 5 / 9;
11   }
12
13   public static float fahrenheit(float c){
14     // a fórmula de conversao de celsius para fahrenheit é [Â°C] Ã— 9/5 + 32
15     return c * 9/5 + 32;
16   }
17
18 }
```

E agora a classe de controle, que ficou mais simples:

```
1  import java.util.Scanner;
2
3  public class ControleTemperatura{
4    public static void main(String[] args) {
5      Scanner teclado = new Scanner(System.in);
6
7      System.out.println("A temperatura que a agua congela em fahrenheit é " +
       Temperatura.CONGELA_F);
8      System.out.println("E que evapora é " + Temperatura.EVAPORA_F);
9      System.out.println("Que convertido em C é " + Temperatura.celsius( Temperatura.
       EVAPORA_F ));
10
11   }
12 }
```

O que você deve ver

```
A temperatura que a agua congela em fahrenheit é 32.0
E que evapora é 212.0
Que convertido em C é 100.0
```

Vamos dar uma olhada na classe `Temperatura`.

Na linha 3 escrevemos:

```
public final static float CONGELA_C = 0;
```

Como as linhas de 3 a 6 da classe `Temperatura` fazem basicamente a mesma coisa, vamos detalhar apenas a 3.

Você já sabe que estamos definindo um atributo chamado `CONGELA_C`, do tipo `float` e que é inicializado com o valor zero.

A palavra `public` declara o atributo como público. Mas um atributo público? Nós não aprendemos que métodos são públicos e atributos são privados?!

É... bem... veja bem... Nem sempre... Mas vamos entender porque.

A palavra `static` você já sabe o que é. Da mesma maneira que temos métodos estáticos, que existem na classe e não precisam de objetos, também temos atributos estáticos, que existem na classe e não precisam de objetos.

E finalmente, a palavra `final`. Ela pode ser aplicada a atributos, métodos e classes, e em cada um deles opera uma função diferente. Antes de explicar todos vamos ver o caso do atributo:

Um atributo declarado como `final`, após inicializado, não poderá ter seu valor alterado. Ele se torna uma constante.

Por convenção, as constantes em Java são escritas com todas as letras em maiúsculas. Você usará mais algumas constantes nesse livro. O Java tem várias constantes e elas servem, basicamente, para você não ter que lembrar qual é o valor. Em quantos graus Fahrenheit a água evapora? Sei lá, use `Temperatura.EVAPORA_F`. Qual o valor de PI com alta precisão? `Math.PI` e está resolvido.

É por isso que constantes são atributos estáticos e públicos, porque eles serão usados para substituir valores.

Antes de terminar, vamos à explicação completa sobre `final`:

- Quando aplicado a um atributo, como por exemplo `public final static float CONGELA_C = 0;`, determina que o atributo se torna uma constante;
- Quando aplicado a uma classe, como por exemplo `public final class Dummy`, impede que a classe seja estendida;
- Quando aplicado a um método, como por exemplo `public final int calcula()`, define que este método não poderá ser sobre-escrito por classes filhas.

Capítulo 16

Classes que são outras classes - Introdução a herança

Vamos aprender agora um dos conceitos mais importantes da orientação a objetos: Herança.

Herança em Java funciona de modo muito parecido com a herança no mundo real só que aqui ninguém precisa morrer para que ela aconteça! A analogia aqui é que tudo que o pai tem, por efeito da herança, o filho também tem. Isso não impede o filho de ter coisas que o pai não tem. Digite o código abaixo e vamos explorar o assunto.

```
1  import java.util.Scanner;
2
3  public class ControleCorreio
4  {
5    public static void main(String[] args)
6    {
7      Carteiro entregador = new Carteiro();
8
9      Correspondencia carta = new Correspondencia();
10     carta.setRemetente("Alice");
11     carta.setDestinatario("Bob");
12     carta.setEnderecoEntrega("Rua dos loucos numero 0.");
13
14     String msg = entregador.entregar(carta);
15     System.out.println("O entregador diz: " + msg);
16
17
18     Pacote encomenda = new Pacote();
19     encomenda.setRemetente("Bob");
20     encomenda.setDestinatario("Rudy");
21     encomenda.setEnderecoEntrega("Travessa travêssa numero 1 ap. 102.");
22
23     String msg2 = entregador.entregar(encomenda);
24     System.out.println("O entregador diz: " + msg2);
25
26   }
27 }
```

```
1  class Carteiro {
2      public String entregar(Correspondencia correspondencia){
3      correspondencia.setEntregue(true);
4          String mensagem = "Correspondencia de " + correspondencia.getRemetente() + "
       para " + correspondencia.getDestinatario() + " ENTREGUE no endereco: " +
      correspondencia.getEnderecoEntrega();
5      return mensagem;
6      }
7  }
```

```
1  public class Correspondencia {
2
3      private String destinatario;
4      private String remetente;
5      private String enderecoEntrega;
6    private boolean entregue = false;
7
8      public String getDestinatario() {
9          return destinatario;
10     }
11
```

```
12      public void setDestinatario(String destinatario) {
13          this.destinatario = destinatario;
14      }
15
16      public String getRemetente() {
17          return remetente;
18      }
19
20      public void setRemetente(String remetente) {
21          this.remetente = remetente;
22      }
23
24      public String getEnderecoEntrega() {
25          return enderecoEntrega;
26      }
27
28      public void setEnderecoEntrega(String enderecoEntrega) {
29          this.enderecoEntrega = enderecoEntrega;
30      }
31
32    public boolean isEntregue(){
33      return entregue;
34    }
35
36      public void setEntregue(boolean entregue) {
37          this.entregue = entregue;
38      }
39  }
```

```
1  public class Pacote extends Correspondencia {
2      int peso;
3      int largura;
4      int altura;
5      int profundidade;
6
7      public int getPeso() {
8          return peso;
9      }
10
11     public void setPeso(int peso) {
12         this.peso = peso;
13     }
14
15     public int getLargura() {
16         return largura;
17     }
18
19     public void setLargura(int largura) {
20         this.largura = largura;
21     }
22
23     public int getAltura() {
24         return altura;
25     }
26
27     public void setAltura(int altura) {
28         this.altura = altura;
29     }
30
31     public int getProfundidade() {
32         return profundidade;
33     }
34
35     public void setProfundidade(int profundidade) {
36         this.profundidade = profundidade;
37     }
38 }
```

O que você deve ver

```
O entregador diz: Correspondência de Alice para Bob ENTREGUE no endereço: Rua dos
    loucos numero 0.
O entregador diz: Correspondência de Bob para Rudy ENTREGUE no endereço: Travessa
    travêssa numero 1 ap. 102.
```

Descanse os dedos por que esse foi grande! Vamos aproveitar essas classes por alguns capítulos para o esforço valer a pena.

Dê uma olhada na classe em *Correspondencia.java*. Nada muito emocionante por ali...

Agora olhe a classe *Pacote.java*. Aqui, se você for observador, percebeu que já na linha 1 há algo novo que você ainda não conhecia: a palavra chave `extends`. Essa palavra chave comunica ao Java que essa classe (`Pacote`) é "herdeira"de `Correspondencia`. Alguns lêm isso dizendo que `Pacote` *é filha de* `Correspondencia`.

Vejamos agora a classe `Carteiro` em *Carteiro.java*. Ali temos apenas um método `entregar`, na linha 2, e sua assinatura [1]prevê o recebimento de um objeto do tipo `Correspondencia`. Guarde essa informação. Esse método apenas altera altera o atributo booleano `entregue` da correspondência.

Agora vamos para o nosso programa principal, o *ControleCorreio.java*. O objetivo desse programa é enviar cartas com a ajuda de um carteiro. Para isso temos um objeto `entregador` do tipo `Carteiro` declarado e instanciado na linha 7.

Em seguida, nas linhas 9 a 12 criamos um objeto `carta` do tipo `Correspondencia` e preenchemos as informações de remetente, destinatário e endereço de entrega. Essas informações são básicas para qualquer tipo de correspondência, certo?!

Pois bem, na linha 14 nosso entregador entrega a carta. Esse objeto faz isso chamando o método `entregar`.

Agora, na linha 18 declaramos uma variável do tipo `Pacote` chamada `encomenda` e na linha 23 pedimos novamente ao entregador para entregar a encomenda.

Ôpa! Espere um pouco. O método `entregar` espera um objeto do tipo `Correspondencia` e nós estamos passando um objeto do tipo `Pacote`. Pode isso?

Claro que sim, porque, como vimos, `Pacote extends Encomenda`, ou seja *"Um pacote é um objeto filho se encomenda"* e como "filho de peixe, peixinho é", a chamada na linha 23 é válida.

Então, respire um pouco e reflita sobre o que acabamos de aprender: Se uma classe é filha de outra, então podemos usar um objeto da classe da filha onde é esperado um objeto do tipo do pai. No nosso exemplo, pudemos atribuir `Pacote` a um método que esperava uma `Correspondencia`, porque todo `Pacote` é, também, uma `Correspondencia`, já que a classe `Pacote extends Correspondencia`.

Se todo `Pacote` é também uma `Correspondencia`, então nosso entregador não vê problema em entregar nosso `Pacote`. Assim, após preenchermos os atributos de nossa encomenda nas linhas 19 a 21, na linha 23 o entregador entrega o pacote como se fosse uma `Correspondencia` como outra qualquer. O `Carteiro` trata o `Pacote` da mesma maneira que trata uma `Correspondencia`, ou seja, utiliza os atributos `remetente`, `destinatário` e `enderecoEntrega` para montar a mensagem e altera o atributo booleano `entregue` normalmente.

Observe novamente o arquivo *Pacote.java* e perceba que nele não há declaração dos atributos `remetente`, `destinatário` e `enderecoEntrega` e, mesmo assim, pudemos preenchê-los nas linhas 19 a 21 e o `Carteiro` pôde utilizá-los normalmente quando chamamos o método `entregar`. Isso é o efeito da herança! a classe `Pacote` herdou tudo que seu pai, a `Correspondencia`, possui. No caso, herdou esses três atributos e seus *setters* e *getters*. Não precisamos repeti-los no coódigo da classe `Pacote`.

Isso não impediu que a classe `Pacote` disponibilizasse atributos próprios, que dizem respeito somente a ela. Informações de peso e tamanho não se aplicam a uma correspondência comum, mas fazem todo sentido em um pacote. Assim, a classe `Pacote` estendeu as funcionalidades da classe `Correspondencia` (daí a palavra chave `extends`). Contudo, como o `Carteiro` está preparado para lidar com `Correspondencia` genérica, ele não faz uso dessas funcionalidades extras.

Desafios para estudo

1. Crie um novo tipo de correspondência, uma multa.

[1]Assinatura de um método é composta de quatro informações básicas: visibilidade, retorno, nome e parametros. no exemplo, a visibilidade é `public`, o retorno é um texto (`String`), o nome é `entregar` e o parâmetro é `corespondencia` do tipo `Correspondencia`

2. Altere o *ControleCorreio* adicionando código para criar uma multa, preenchê-la e faça o entregador entregá-la.

Capítulo 17

Entendendo o sentido da herança e suas limitações

Você aprendeu no capítulo anterior a identificar quando uma classe herda de outra, quando uma é filha da outra, ou em uma linguagem um pouco mais técnica, quando uma classe estende a outra.

Também aprendeu que uma variável de um tipo hierárquico inferior pode ser passado como parâmetro onde um objeto de hierarquia superior é esperado. Ou seja, se `Filho` estende `Pai`, então um método que espera `p` do tipo `Pai` pode receber uma f do tipo `Filho`.

Será que o contrário é verdade?

Digite novamente o arquivo *ControleCorreio.java* pois foi bastante alterado. No arquivo *Carteiro.java* adicione as linhas 6 a 9 a seguir. Os demais arquivos do capítulo anterior serão usados sem alterações.

```
1  import java.util.Scanner;
2
3  public class ControleCorreio
4  {
5    public static void main(String[] args)
6    {
7      Carteiro entregador = new Carteiro();
8
9      Pacote encomenda = new Pacote();
10     encomenda.setRemetente("Bob");
11     encomenda.setDestinatario("Rudy");
12     encomenda.setEnderecoEntrega("Travessa travêssa numero 1 ap. 102.");
13
14     //linhas adicionadas!!!
15     encomenda.setAltura(10);
16     encomenda.setLargura(5);
17     encomenda.setProfundidade(7);
18     encomenda.setPeso(50);
19
20     String msg = entregador.entregarPacote(encomenda);
21     System.out.println("O entregador diz: " + msg);
22
23     Pacote carta = new Correspondencia(); //VAI DAR ERRO AQUI!
24     carta.setRemetente("Alice");
25     carta.setDestinatario("Bob");
26     carta.setEnderecoEntrega("Rua dos loucos numero 0.");
27
28     msg = entregador.entregarPacote(carta);
29     System.out.println("O entregador diz: " + msg);
30
31   }
32 }
```

```
1  class Carteiro {
2      public String entregar(Correspondencia correspondencia){
3      correspondencia.setEntregue(true);
4          return "Correspondencia de " + correspondencia.getRemetente() + " para " +
       correspondencia.getDestinatario() + " ENTREGUE no endereco: " + correspondencia.
       getEnderecoEntrega();
5
6      }
7    //NOVO MÃ%dTODO!
```

```
8   public String entregarPacote(Pacote pac){
9     pac.setEntregue(true);
10        String mensagem = this.entregar(pac);
11    mensagem += " \nEsse pacote pesa: " + pac.getPeso() + " e suas dimensoes sao: "
     + pac.getAltura() + "x" + pac.getLargura() +  "x" + pac.getProfundidade();
12    return mensagem;
13    }
14 }
```

O que você deve ver

```
ControleCorreio.java:23: error: incompatible types: Correspondencia cannot be co
nverted to Pacote
                Pacote carta = new Correspondencia();
                               ^
1 error
```

Pergunta respondida, certo?!

Não foi possível atribuir uma instancia da classe `Correspondencia` a uma variável do tipo `Pacote`. Isso por que é `Pacote` quem estende `Correspondencia` e não o contrário. Então, podemos dizer que um `Pacote` é uma `Correspondencia`, mas uma `Correspondencia` não é um `Pacote`.

Vamos tentar entender o motivo de isso não ser possível no Java. Vá até o arquivo *Carteiro.java* e observe o novo método que criamos na linha 8. Ele recebe um `Pacote`. Sendo assim, ele acredita que o objeto recebido como parâmetro, por ser um `Pacote`, possui métodos como `getPeso()`, `getAltura()`, etc... Se o java permitisse que passássemos uma `Correspondencia` para esse método, como ele iria funcionar chamando esses métodos se uma `Correspondencia` **não tem esses métodos**?! Impossível. Para que o método `entregarPacote` funcione ele precisa receber um objeto que possua `getPeso()`, `getAltura()`, `getLargura()` e `getProfundidade()`.

Por causa desse tipo de situação que o java não nos deixa compilar a linha 25 do arquivo *ControleCorreio.java*. Um pai não tem tudo que um filho tem, então o pai não pode ser usado onde o filho é usado. Por isso são tipos incompatíveis. O contrário, não custa lembrar, é válido, afinal, tudo que o pai tem o filho tem, então não corremos perigo de algum método esperar que tenhamos um recurso que não temos.

Para corrigir o programa, altere a atribuição feita à variável `carta` na linha 25. A linha vai ficar parecida com o seguinte:

```
25 Pacote carta = new Pacote();
```

Compile o código e execute. Você deve ver:

```
O entregador diz: Correspondencia de Bob para Rudy ENTREGUE no endereco: Travessa
    travêssa numero 1 ap. 102.
Esse pacote pesa: 50 e suas dimensoes sao: 10x5x7
O entregador diz: Correspondencia de Alice para Bob ENTREGUE no endereco: Rua dos
    loucos numero 0.
Esse pacote pesa: 0 e suas dimensoes sao: 0x0x0
```

Tranquilo entender o que se passa agora não é?

Primeiramente no programa criamos um novo `Pacote` e setamos os atributos herdados do pai nas linhas 10 a 12. Nas linhas 15 a 18 setamos os atributos próprios de um `Pacote`.

Na linha 20 chamamos o novo método da classe `Carteiro` por meio do objeto `entregador` criado na linha 7. Já discutimos que esse novo método recebe apenas `Pacote`s.

O trecho seguinte na linha 25, que você corrigiu, cria um `Pacote` e define apenas os atributos herdados de `Correspondencia`. Nenhum problema quanto a isso. Os métodos próprios de `Pacote` não informados vão zerados para o `entregador` que exibe tudo zerado, como vimos.

Desafios para estudo

1. Altere novamente a linha 25 do *ControleCorreio.java* para que seja declarada uma variável do tipo `Correspondencia` e que a atribuição também seja feita com essa classe. Algo como: `Correspondencia carta = new Correspondencia();`

2. Compile e observe onde java aponta o erro. Não será mais na linha 25 como foi na versão original do exercício. Você entende por que o Java está reclamando? Qual método do `entregador` temos que usar para que o código funcione com essa `Correspondencia` ?

Capítulo 18

Sobre-escrita, fazendo várias coisas em uma linha só

No capítulo anterior tivemos nosso contato com herança. Esse conceito pode ser um tanto complicado e nós vamos tentar andar com calma... depois. Agora é hora de aprender coisas novas.

Primeiro, vamos criar uma nova classe:

```
1  public class Carta extends Correspondencia {
2      boolean seloColado;
3
4      public void setSeloColado(boolean selo ) {
5          this.seloColado = selo;
6      }
7
8      public boolean isSeloColado(){
9      return seloColado;
10   }
11
12   public boolean verificaEntregaAutorizada(){
13     return isSeloColado();
14   }
15
16 }
```

Depois, altere a classe `Correspondencia`. Apenas o método que começa na linha 41 foi incluído:

```
1  public class Correspondencia {
2
3      private String destinatario;
4      private String remetente;
5      private String enderecoEntrega;
6    private boolean entregue = false;
7
8      public String getDestinatario() {
9          return destinatario;
10     }
11
12     public void setDestinatario(String destinatario) {
13         this.destinatario = destinatario;
14     }
15
16     public String getRemetente() {
17         return remetente;
18     }
19
20     public void setRemetente(String remetente) {
21         this.remetente = remetente;
22     }
23
24     public String getEnderecoEntrega() {
25         return enderecoEntrega;
26     }
27
28     public void setEnderecoEntrega(String enderecoEntrega) {
29         this.enderecoEntrega = enderecoEntrega;
30     }
31
```

```
32   public boolean isEntregue(){
33     return entregue;
34   }
35
36     public void setEntregue(boolean entregue) {
37         this.entregue = entregue;
38     }
39
40   // novo método
41   public boolean verificaEntregaAutorizada(){
42     boolean destinatarioValido = (destinatario != null) && (!destinatario.equals("")
       );
43     boolean enderecoEntregaValido = (enderecoEntrega != null) && (!enderecoEntrega.
       equals(""));
44
45     return (destinatarioValido) && (enderecoEntregaValido);
46   }
47
48 }
```

Agora, altere o método `entregar()` da classe `Carteiro`:

```
1  class Carteiro {
2
3    // Método alterado
4      public String entregar(Correspondencia correspondencia){
5
6      if ( correspondencia.verificaEntregaAutorizada() ){
7        correspondencia.setEntregue(true);
8            String mensagem = "Correspondencia de " + correspondencia.getRemetente() +
       " para " + correspondencia.getDestinatario() + " ENTREGUE no endereco: " +
       correspondencia.getEnderecoEntrega();
9        return mensagem;
10
11     } else {
12       correspondencia.setEntregue(false);
13       return "Entrega não autorizada";
14     }
15
16     }
17 }
```

Por fim, redigite a classe de controle, pois tivemos muitas alterações:

```
1  import java.util.Scanner;
2
3  public class ControleCorreio
4  {
5    public static void main(String[] args)
6    {
7      Carteiro entregador = new Carteiro();
8
9      Pacote encomenda = new Pacote();
10     encomenda.setRemetente("Bob");
11     encomenda.setDestinatario("Rudy");
12     encomenda.setEnderecoEntrega("Travessa travêssa numero 1 ap. 102.");
13
14     encomenda.setAltura(10);
15     encomenda.setLargura(5);
16     encomenda.setProfundidade(7);
17     encomenda.setPeso(50);
18
19     String msg = entregador.entregar(encomenda);
20     System.out.println("O entregador diz: " + msg);
21     System.out.println("Encomenda entregue: " + encomenda.isEntregue());
22
23     Carta carta = new Carta();
24     carta.setRemetente("Alice");
25     carta.setDestinatario("Bob");
26     carta.setEnderecoEntrega("Rua dos loucos numero 0.");
27
28     msg = entregador.entregar(carta);
29     System.out.println("O entregador diz: " + msg);
30     System.out.println("Carta entregue: " + carta.isEntregue());
31   }
32 }
```

O que você deve ver

```
O entregador diz: Correspondencia de Bob para Rudy ENTREGUE no endereco: Travessa
    travêssa numero 1 ap. 102.
Encomenda entregue: true

O entregador diz: Entrega não autorizada
Carta entregue: false
```

Funcionou! Já é um grande avanço. Apesar do resultado simples, temos uma coisa bem legal acontecendo aqui. Vamos entender.

Você criou uma nova classe chamada `Carta` que estende `Correspondencia`. Essa classe é bem simples e só tem um atributo a mais que indica se existe selo na carta (`boolean seloColado`).

Lembre que a classe `Correspondencia` é uma classe pai. As classes `Pacote` e `Carta` são classes filhas.

Quando você escreve um método na classe pai, esse é o comportamento genérico. Vale para o pai e para toda a hierarquia. No caso do método você está dizendo "qualquer correspondência vai verificar a entrega autorizada desta maneira".

Quando você coloca atributos na classe pai, esses são os atributos genéricos. Valem para o pai e para toda a hierarquia. Você está dizendo "qualquer correspondência terá esses três atributos".

Quando você coloca coisas nas classes filhas, você está definindo comportamentos (para métodos) e estrutura (para atributos) específicos. Como se dissesse "toda correspondência tem destinatário, mas só cartas têm também selo colado". O pai não conta com essas funcionalidades.

Tudo bem até aqui? Ainda está acordado[1]? Então sigamos...

Na classe `Correspondencia`, você incluiu o método `verificaEntregaAutorizada()` na linha 41. Ele verifica se o endereço e se o destinatário são validos, isto é, se eles não estão vazios. Ele será usado pelo `Carteiro` para determinar se uma correspondência pode ou não ser entregue. Veja lá, a partir da linha 4 da classe `Carteiro`. Antes de enviar a `Correspondencia` o `Carteiro` verifica se essa entrega está autorizada (linha 6) e só entrega se estiver. Do contrário, ele nega a entrega e retorna uma mensagem nas linhas 12 e 13.

O método `verificaEntregaAutorizada()` define o comportamento para objetos criados a partir da classe pai `Correspondencia`.

Porém, você também escreveu um método `verificaEntregaAutorizada()` na classe `Carta`. Então, se o objeto for do tipo `Carta`, esse será o método utilizado na validação. Isso se chama "sobre-escrita".

Vamos definir direitinho:

Sobre-escrita é quando escrevemos um método em uma classe filha que tem a mesma assinatura de um método de uma classe pai.

Agora, falando difícil para ficar mais bonito: *Sobre-escrita é quando substituímos um comportamento genérico por um comportamento específico.*

É como se disséssemos "toda correspondência verifica a entrega autorizada do mesmo jeito, menos a carta. Ela tem um jeito próprio de fazer isso".

Então um objeto do tipo `Correspondencia` verifica o endereço e o destinatário usando o método `verificaEntregaAutorizada()` da classe `Correspondencia`. Um objeto `Pacote` também, por que ela é uma classe filha. Mas um objeto `Carta` não, porque apesar de ser uma classe filha, ele sobre-escreve o comportamento da classe pai.

Acho que você ainda não viu como isso é uma coisa bonita. Então vou te ajudar.

A classe `ControleCorreio` chama o método `entregador.entregar()` duas vezes, na linha 19 e na linha 28. Na primeira vez passa um "pacote", na segunda passa uma "carta" como parâmetro.

[1] Se não, tome café. Se você não gosta de café, aprenda a gostar. Isso é tão importante para um programador como o ar que respira. Aliás, já viu por que o Java chama Java? Tudo a ver com café.

Esses dois objetos tem comportamentos diferentes na hora de verificar se a entrega pode ser feita (método `verificaEntregaAutorizada()`).

O método `Carteiro.entregar` recebe um objeto do tipo `Correspondencia` que é a classe pai. E na linha 6 chama o método `verificaEntregaAutorizada()` sem se preocupar com qual objeto ele recebeu. Independentemente do objeto recebido, seja um pacote (linha 19) ou uma carta (linha 28) o `Carteiro` chama o mesmo método `verificaEntregaAutorizada()`, mas em cada um desses casos o comportamento é diferente.

Preste atenção! A linha 6 da classe `Carteiro` ora executa o método de `Correspondencia`, ora executa o método de `Carta` e nós não precisamos nos preocupar em decidir quando executar qual, o Java faz isso para nós.

É lindo ou não é?

Desafios para estudo

1. Crie o método `verificaEntregaAutorizada()` na classe `Pacote`, lembrando de manter a mesma assinatura do Pai(`public boolean verificaEntregaAutorizada()`).
2. Escreva o código desse método da seguinte maneira: Se o peso do pacote for maior do que 30, a entrega não pode ser autorizada. Do contrário, a entrega poderá ser feita.
3. Execute o código e observe se agora o pacote criado na linha 9 será entregue.

Capítulo 19

Chamando métodos do pai - `super` legal

Tudo bem até aqui? Vimos herança, sobre-escrita, isso é muita coisa. Então esse capítulo vai ser leve. No próximo vamos exercitar um pouco antes de acelerarmos novamente.

Você vai fazer uma única pequena alteração no programa e eu já te explico:

```
1  public class Carta extends Correspondencia {
2      boolean seloColado;
3
4      public void setSeloColado(boolean selo ) {
5          this.seloColado = selo;
6      }
7
8      public boolean isSeloColado(){
9      return seloColado;
10   }
11
12   public boolean verificaEntregaAutorizada(){
13     return super.verificaEntregaAutorizada() && isSeloColado(); // método modificado
14   }
15
16 }
```

O que você deve ver

```
O entregador diz: Correspondencia de Bob para Rudy ENTREGUE no endereco: Travessa
    travêssa numero 1 ap. 102.
Encomenda entregue: true

O entregador diz: Entrega não autorizada
Carta entregue: false
```

Você já sabe que o método `verificaEntregaAutorizada()` está definido em `Correspondencia` e sobre-escrito em `Carta`. Em `Correspondencia` ele verifica o endereço e o destinatário e em `Carta` ele verifica o selo.

Mas e se a classe filha `Carta` quisesse verifica *tudo o que a classe pai verifica* e mais o selo?

Olhe a linha 13 da classe `Carta`, logo depois do `return` temos uma nova palavra: `super`. Lembra que a palavra `this` quer dizer algo como "eu mesmo"? Pois então, a palavra `super` quer dizer algo como "meu pai".

Então `super.verificaEntregaAutorizada()` é "chame o método `verificaEntregaAutorizada()` do meu pai e depois continue o programa".

Simples assim.

Sem desafios por hoje, beba uma coca gelada e relaxe um pouco.

Capítulo 20

Exercitando classes e herança

Neste capítulo vamos exercitar um pouco mais as classes e a herança. Vamos tentar uma coisa diferente. Eu vou começar com um problema hipotético e depois construímos a solução, ok?

O problema

"Uma escola tem professores e alunos. Ambos têm nome. Professores têm a matrícula funcional (que na escola chamam de MF) como um dos seus dados e alunos têm o registro de aluno (que chamam de RA). Precisamos de um modelo de classes que permita mostrar o nome e a identificação, que pode ser o MF ou o RA, dependendo do tipo da pessoa."

A solução

Como ficou claro que professores e alunos têm partes em comum (nome) e partes específicas (a identificação de cada um), temos aqui uma boa estrutura para fazer uma herança. Vamos construir uma classe genérica, ou pai, chamada `Pessoa`, e as duas mais específicas, ou filhas: `Professor` e `Aluno`.

A classe Pessoa fica assim:

```
1  public class Pessoa{
2    private String nome;
3
4    // Método construtor
5    public Pessoa(String nome){
6      this.nome = nome;
7    }
8
9    public String getNome(){
10     return nome;
11   }
12
13
14 }
```

A classe Professor:

```
1  public class Professor extends Pessoa{
2    private int mf;
3
4    public Professor( String nome, int mf){
5      super(nome);
6      this.mf = mf;
7    }
8
9    public int getMf(){
10     return mf;
11   }
12 }
```

A classe Aluno:

```
1  public class Aluno extends Pessoa {
2    private int ra;
3
4    public Aluno(String nome, int ra){
5      super(nome);
6      this.ra = ra;
7    }
8
9    public int getRa(){
10     return ra;
11   }
12 }
```

E finalmente, para fazer as coisas acontecerem, a classe de controle:

```
1  public class ControleCurso{
2    public static void main(String[] args) {
3
4      Professor p = new Professor("Asdrúbal Maltêz", 28287);
5
6      Aluno a = new Aluno("José ferreira", 92881);
7
8      System.out.println( "Professor " + p.getNome() + " Matrícula: " + p.getMf());
9      System.out.println( "Aluno " + a.getNome() + " RA: " + a.getRa());
10   }
11 }
```

O que você deve ver

```
Professor Asdrúbal Maltêz Matrícula: 28287
Aluno José ferreira RA: 92881
```

As classes são uma estrutura de herança, com atributos, *gets* e *sets*. A classe de controle também é bem simples e vamos melhorá-la no próximo capítulo. Mas deixe-me fazer um destaque.

A classe `Pessoa` tem um construtor a partir da linha 5, que exige que toda pessoa tenha um nome na criação do seu objeto.

Agora, raciocine comigo, se uma Pessoa tem uma regra de construção, e um Professor é uma Pessoa, então ele deverá cumprir essa mesma regra, concorda?

Por isso o Java exige que *quando uma classe pai tem um construtor personalizado, todas as classes filhas têm que ter construtores que chamem o construtor do pai.*

É isso que estamos fazendo a partir da linha 4 da classe `Professor`. Ali temos um construtor. E na linha 4 usamos `super()` que é a chamada ao construtor da classe pai.

Fazemos a mesma coisa na classe `Aluno`.

Ok? Recomendo uma nova lida no capítulo. Garanta que você entendeu a estrutura da herança e porque utilizamos o `super()`. No próximo capítulo vamos melhorar esse modelo todo.

Capítulo 21

Entendendo métodos e classes abstratas

Agora vamos tratar de um tópico complexo, mas tentarei caminhar passo-a-passo para torná-lo simples e você irá acumular mais conhecimento em sua jornada para se tornar um cavaleiro Jedi em Java.

Vamos digitar os programas abaixo e logo passamos para as explicações. São só algumas poucas alterações que estão indicadas.

A classe Pessoa:

```
1  public abstract class Pessoa{ //mudança nessa linha
2    private String nome;
3
4    // Método construtor
5    public Pessoa(String nome){
6      this.nome = nome;
7    }
8
9    public String getNome(){
10     return nome;
11   }
12
13   public abstract String getIdentificacao(); // incluir essa linha
14
15 }
```

A classe Professor:

```
1  public class Professor extends Pessoa{
2    private int mf;
3
4    public Professor( String nome, int mf){
5      super(nome);
6      this.mf = mf;
7    }
8
9    public int getMf(){
10     return mf;
11   }
12
13   // incluir esse método
14   public String getIdentificacao(){
15     return " Matrícula: " + getMf();
16   }
17 }
```

A classe Aluno:

```
1  public class Aluno extends Pessoa {
2    private int ra;
3
4    public Aluno(String nome, int ra){
5      super(nome);
6      this.ra = ra;
7    }
8
9    public int getRa(){
10     return ra;
11   }
```

```
12
13   // incluir esse método
14   public String getIdentificacao(){
15     return " Registro: "+ getRa();
16   }
17 }
```

A classe de controle:

```
1  public class ControleCurso{
2    public static void main(String[] args) {
3
4      Professor p = new Professor("Asdrúbal Maltêz", 28287);
5
6      Aluno a = new Aluno("José ferreira", 92881);
7
8      System.out.println( "Professor " + p.getNome() + p.getIdentificacao());
9      System.out.println( "Aluno " + a.getNome() + a.getIdentificacao());
10   }
11 }
```

O que você deve ver

```
Professor Asdrúbal Maltêz Matrícula: 28287
Aluno José ferreira RA: 92881
```

A saída do programa não é o mais importante, por agora. Apesar de pequenas, as alterações sim, são importantíssimas.

Mas antes uma pequena revisão. Nós vimos que em uma herança temos o seguinte:

- **Métodos que existem apenas na classe pai.** Esse métodos tem um comportamento genérico para todas as classes filhas, isto é, todas as classes filhas se comportam da mesma maneira;
- **Métodos que existem apenas em algumas classes filha.** Esses métodos são muito específicos e só fazem sentido em algumas classes filha;
- **Métodos que existem na classe pai e são sobre-escritos.** Esse é uma casso onde a regra geral (o método da classe pai) tem alguma exceção (o método sobre-escrito pela classe filha).

Porém, e se uma herança precisa que cada classe filha tenha um comportamento específico (ou seja, escrito na classe filha), e que seja obrigatório para todas as classes filha?

Entendeu? É complicado mesmo, mas vou tentar exemplificar com nosso modelo.

O comportamento de dizer qual é a sua identificação é da classe pai ou das classes filha? Das filhas, certo? O Professor tem que dizer qual a matrícula dele e o Aluno tem que dizer qual é o seu registro. O importante é que *todas as classes filha terão que saber apresentar uma identificação*. Então, se no futuro criarmos uma nova classe que estenda `Pessoa`, por exemplo `Funcionario`, temos que garantir que ele saiba mostrar algum tipo de identificação[1], ok?

Para garantir essa regra, precisamos de métodos e classes `abstract`.

Na classe `Pessoa`, na linha 19 temos `public abstract String getIdentificacao();`. O que essa linha quer dizer? Ela está definindo um método abstrato. E o que é um método abstrato?

1. Um método abstrato, não é um método implementado. Ele é apenas a declaração de um método (nome, retorno e parâmetros);
2. Esse método abstrato não faz nada. Ele apenas define uma regra.
3. Mas qual regra? A regra que diz "eu sou uma classe pai e todas as classes filha terão que implementar esse método"

[1] As razões de porque isso é importante, ficarão mais claras no próximo capítulo. Por hora, fique firme e tente entender tudo. Aliás, lembre-se que o mestre Yoda disse: "tentar não! Ou faça, ou não faça".

Então, por causa da linha 13, qualquer classe que seja filha de `Pessoa` terá que ter o método `getIdentificacao` exatamente com o mesmo retorno e os mesmos parâmetros (neste caso específico, o método não tem parâmetros).

Até aqui tudo bem?

Agora, olhe esse código hipotético abaixo:

```
...
Pessoa p = new Pessoa("Antonio");
System.out.println( p.getIdentificacao() );
...
```

Aí nós estamos criando um objeto `Pessoa` e chamando o método `getIdentificacao()`, certo? Agora a pergunta: `Pessoa` tem esse método? Não! Ele é abstrato, e se é abstrato não é um método que pode ser chamado.

Portanto, não deveria ser possível criar um objeto `Pessoa`, mas apenas `Professor` ou `Aluno`. `Pessoa` é um conceito genérico, mas não pode ser um objeto concreto.

Por isso, logo na primeira linha da classe `Pessoa` existe `public abstract class`. Esse comando está definindo a classe como abstrata. O que é uma classe abstrata?

Uma classe abstrata é um classe que não pode ser instanciada por um objeto. Ela é um conceito genérico.

Ufa... que lambança de conceitos. Vamos resumir:

- Um **método abstrato**, é um método que não tem implementação, ele define uma regra de construção para as classes filhas;
- Uma **classe abstrata** é uma classe que não pode ser instanciada. Se a classe tem um método abstrato, obrigatoriamente ela deverá ser abstrata.

Desafios para estudo

Estava com saudade dos desafios para estudo? Eu estava. Vamos lá:

1. Crie uma nova classe `Funcionario` que estenda `Pessoa`. Coloque nessa classe um atributo `private int registroFuncional`.
2. Lembre que você terá que criar um construtor.
3. Compile. Funcionou? Provavelmente você deve estar tendo o erro abaixo:

   ```
   Funcionario.java:1: error: Funcionario is not abstract and does not override
     abstract method getIdentificacao() in Pessoa
   public class Funcionario extends Pessoa{
          ^
   1 error
   ```

   O Java está te dizendo que como você estende `Pessoa`, você tem que escrever o método `getIdentificacao()`.
4. Crie o método e veja se está tudo funcionando.

Gabarito

Segue a classe `Funcionario`.

```
1 public class Funcionario extends Pessoa{
2   private int registroFuncional;
3
4   public Funcionario( String nome, int registro){
5     super(nome);
6     this.registroFuncional = registro;
```

```
7     }
8
9
10 }
```

Capítulo 22

Polimorfismo. Agora a coisa ficou séria!

Vamos para mais um daqueles capítulos onde o programa é simples, o resultado é mais simples ainda, mas a explicação parece saída de um capítulo de "Além da Imaginação".

Aproveite os programas anterios, e faça apenas essa pequena alteração na classe de controle:

```
1  public class ControleCurso{
2    public static void main(String[] args) {
3
4      Professor p = new Professor("Asdrúbal Maltêz", 28287);
5      Aluno a = new Aluno("José ferreira", 92881);
6
7      // Modificado a partir daqui
8      mostraId( p );
9      mostraId( a );
10   }
11
12   // Método novo
13   public static void mostraId( Pessoa pessoa ){
14     System.out.println(  pessoa.getNome() + pessoa.getIdentificacao());
15   }
16 }
```

O que você deve ver

```
Professor Asdrúbal Maltêz Matrícula: 28287
Aluno José ferreira RA: 92881
```

Não é grande coisa, certo? Errado! Coisas incríveis estão acontecendo "debaixo do capô".

As linhas 4 e 5 criam dois objetos,`p` do tipo `Professor` e `a` do tipo `Aluno`.

As linhas 8 e 9, chamam o método `mostraId()` duas vezes.

Nas linhas de 13 a 15 o método `mostraId()` está definido. Ele recebe como parâmetro um objeto do tipo `Pessoa`.

Na linha 14 nos mostramos alguns dados da pessoa, mas temos uma chamada bastante peculiar ali. Observe `pessoa.getIdentificacao()`. Pergunta: um objeto `Pessoa` tem um método `getIdentificacao()` que possa ser chamado? Resposta: Não! O método é abstrato.

Então o que está acontecendo nessa linha?

O que está acontecendo é uma chamada polimórfica. Uma chamada polimórfica acontece quando, a partir de um objeto genérico (classe pai), você chama um método específico (da classe filha). Entendeu?

Quando você recebe uma `Pessoa` no parâmetro, o Java sabe se essa pessoa é um professor ou um aluno. Na hora de chamar o método `getIdentificacao()` o Java resolve qual método ele chama, se o de aluno, ou se o de professor. Como essa decisão só ocorre na hora que o programa executa, dizemos que ela é resolvida "em tempo de execução". Veja que todas as outras linhas do programa são resolvidas "em tempo de compilação", isto é, quando o programa é compilado. Você pode lê-las e dizer exatamente o que vai acontecer. A linha 14 não. Depende do parâmetro recebido.

O polimorfismo é um recurso extremamente poderoso de uma linguagem OO e com o tempo você vai aprender a usá-lo. No próximo capítulo vamos fazer um programa bacana com ele.

Antes de encerrar, deixe-me falar mais umas coisinhas. O polimorfismo acontece em duas situações:

- Pela chamada de um método abstrato em uma herança, a partir da classe genérica
- Pela chamada de um método definido em uma inteface. Você não sabe o que é isso, ainda.

Desafios para estudo

- No capítulo anterior você fez a classe `Funcionario`? Se não, faça.
- Mostre a identificação do funcionário, usando a chamada polimórfica.

Capítulo 23

Polimorfismo com ArrayList

Nesse capítulo vamos reforçar o uso da chamada polimórfica. Vamos usar as mesmas classes do capítulo anterior, mudando apenas a classe de controle.

Digite o programa abaixo:

```
1  import java.util.ArrayList;
2  import java.util.Scanner;
3  import java.util.InputMismatchException;
4
5  public class ControleCurso{
6    public static void main(String[] args) {
7      String continuar, nome;
8      int id;
9      ArrayList<Pessoa> pessoas = new ArrayList<>();
10     Pessoa p;
11     Scanner teclado = new Scanner(System.in);
12
13     System.out.println("Vamos adicionar várias pessoas. ");
14
15     do{
16       System.out.println("Digite o nome");
17       nome = teclado.next();
18
19       do{
20         try{
21           System.out.println("Digite a identificacao");
22           id = teclado.nextInt();
23           break;
24         }catch(InputMismatchException e){
25           System.out.println("Você não digitou um número válido. Tente novamente.");
26           teclado.next();
27         }
28       } while( true);
29
30       System.out.println("Digite P, para Professor ou A, para Aluno");
31       String opcao = teclado.next();
32
33       if (opcao.equalsIgnoreCase("A")){
34         p = new Aluno( nome, id);
35       } else {
36         p = new Professor(nome, id);
37       }
38
39       pessoas.add( p );
40
41       System.out.println("Deseja continuar? Digite S ou N");
42       continuar = teclado.next();
43     }while(continuar.equalsIgnoreCase("S"));
44
45     for (Pessoa pessoa:pessoas){
46       mostraId(pessoa);
47     }
48
49
50   }
51
52   // Método novo
53   public static void mostraId( Pessoa pessoa ){
54     System.out.println(  pessoa.getNome() + pessoa.getIdentificacao());
```

```
55   }
56
57 }
```

Já fazia algum tempo que não digitávamos um programa cheio de coisas, não é?

Esse programa permite que você insira quantas pessoas quiser em um `ArrayList`, definindo quais são professores e alunos e depois mostra a identificação de todos.

As linhas de 7 a 11 criam as variáveis que vamos usar, inclusive o `ArrayList` chamado `pessoa`. Observe que ele irá armazenar objetos do tipo `Pessoa` e "aluno é pessoa" e "professor é pessoa".

Das linhas 15 a 43 fica o laço principal que controla a entrada dos dados.

Nas linhas 16 e 17 pedimos o nome da pessoa.

Das linhas 19 a 29 temos um segundo laço que serve para garantir uma entrada segura da identificação. Nós fazemos isso com um bloco `try..catch`, linhas 20 a 28. Se a entrada foi tranquila, a linha 23 faz um `break` e sai do laço[1]. Mas se for digitado algo diferente de um inteiro, o `catch` da linha 24 é disparado. Então a linha 25 mostra a mensagem e a 26 limpa o `scanner`.

Nas linhas 30 a 31 é escolhido o tipo de objeto a ser criado. Em um programa "do mundo real" a coisa não seria assim, mas queremos deixar a criação de alunos e professores aleatória.

As linhas 33 a 37 criam o objeto equivalente à escolha feita. Veja que usamos `Pessoa p = <objeto escolhido>`. Sim, nós podemos fazer isso! No fim das contas `p` vai ser um `Aluno` ou um `Professor`, mas nunca uma `Pessoa`. Por quê? Porque `Pessoa` é uma classe abstrata e classes abstratas não podem ser instanciadas, certo?

A linha 39 adiciona o objeto ao `ArrayList`.

As linhas 41 e 42 cuidam de continuar o laço que é fechado na 43.

Nas linhas de 45 a 47 temos um laço `for` simples que caminha pelo `ArrayList` e chama o método `mostraId` que já conhecemos.

Agora, vejamos, se eu entrar 10 pessoas diferentes, na quinta posição do `ArrayList`, o que será mostrado por essa linha? Resposta: não sabemos, porque será feita uma chamada polimórfica (exatamente lá na linha 54), resolvida em tempo de execução, dependendo dos objetos que eu criar. Bonito, não?

[1] Como a saída do laço é controlada por um `break`, e não pela condição avaliada, na linha 28 temos uma "condição infinita" `while(true)`. Eu podia ter escrito isso no parágrafo, mas fazia tempo que não tínhamos uma nota de rodapé...

Capítulo 24

Interfaces (OMG!)

Se você chegou até aqui e principalmente se entendeu herança, classes abstratas e polimorfismo, pode se considerar um vencedor. Mas não relaxe, ainda temos mais um assunto para abortar antes de poder te dar a faixa marrom: interfaces[1].

Vamos escrever umas coisinhas. Esse programa simula um pilha de coisas em um armazém. Tipicamente, os itens vão se misturando na pilha: ora temos latas de tinta, caixas de livros, ou caixas com sacos de batata-frita. O problema é que cada item desse tem um peso máximo de empilhamento que ele suporta. Imagine o que acontece se as latas de tinta forem colocadas em cima das batatas-fritas? Esse programa empilha as coisas de forma segura, ele verifica se o peso da pilha é suportado.

Vamos lá, primeiro uma coisa chamada `Empilhavel` que vamos explicar depois:

```
1 public interface Empilhavel{
2   public double getPeso();
3
4   public double getPesoSuportado();
5
6   public String getDescricao();
7 }
```

Duas classes para brincarmos:

```
1  public class Lata implements Empilhavel{
2
3    public double getPeso(){
4      return 2.50;
5    }
6
7    public double getPesoSuportado(){
8      return 100.00;
9    }
10
11   public String getDescricao(){
12     return "Lata. Peso: " + getPeso();
13   }
14 }
```

e

```
1  public class BatataFrita implements Empilhavel{
2
3    public double getPeso(){
4      return 0.02;
5    }
6
7    public double getPesoSuportado(){
8      return 2.00;
9    }
10
11   public String getDescricao(){
12     return "Batata Frita. Peso: " + getPeso();
13   }
14 }
```

[1] Caso você esteja pensando que se trata de desenho de telas, não é isso. Para não confundir, as telas da aplicação são chamadas de interface gráfica, camada de apresentação ou simplesmente, apresentação.

E finalmente, a classe de controle:

```
1  public class ControlePilha{
2    public static void main(String[] args) {
3
4      Pilha pilha = new Pilha();
5      Lata l = new Lata();
6      BatataFrita chips = new BatataFrita();
7
8      if (pilha.coloca( chips )){
9        System.out.println("item empilhado com sucesso.");
10     } else {
11       System.out.println("Excedido peso máximo suportado.");
12     }
13
14     if (pilha.coloca( l )){
15       System.out.println("item empilhado com sucesso.");
16     } else {
17       System.out.println("Excedido peso máximo suportado.");
18     }
19
20     pilha.lista();
21   }
22 }
```

O que você deve ver

```
item empilhado com sucesso.
Excedido peso máximo suportado.
Itens da pilha
---------------
Batata Frita. Peso: 0.02
```

Vamos começar as explicações pela classe `Pilha`. Essa classe representa o lugar onde os objetos serão empilhados. Ela tem um `ArrayList`, definido na linha 4.

Nas linhas de 6 a 13 temos o método `coloca`. Ele representa o ato de colocar um objeto na pilha e para isso, o peso da pilha tem que ser comparado com o peso suportado por cada objeto que já está na pilha.

Isso é realizado pelo método `podeEmpilhar`, que está nas linhas de 15 a 33. Como esse não é o foco do capítulo, vou deixar que você compreenda a lógica que está ali.

O problema da nossa pilha é que ela pode receber qualquer coisa que possa ser empilhada. E se você ver ali na linha 21, verá que uma coisa que pode ser empilhada tem `getPesoSuportado()` e na linha 25 verá que ela também tem `getPeso()`.

Então como definir no programa "coisas que podem ser empilhadas"? Uma solução você já conhece: a herança. Eu poderia fazer uma classe mãe que definisse `getPesoSuportado()` e `getPeso()` como métodos abstratos. Resolve? não. Porque, apesar de essa estrutura garantir a presença dos dois métodos, como fazer uma herança onde uma classe filha seja batata-frita, outra seja lata de tinta e outra garrafa de coca-cola? Essas classes são todas coisas que podem ser empilhadas, mas fora isso, não têm mais nada em comum. Por isso a herança não é uma coisa legal.

Lembre-se: a herança define uma estrutura de classes que compartilhas atributos, métodos e que sejam logicamente comuns.

E agora?

Vamos olhar o arquivo `Empilhavel.java`.

Na linha 1 temos:

```
public interface Empilhavel{
```

Viu? Ao invés de `public class`, que define uma classe, temos `public interface`, que define uma interface. Mas o que é uma interface?

Uma interface é um conjunto de declaração de métodos (sem implementação).

Simples assim. Da mesma forma que os métodos abstratos definem uma regra, a interface também:

Quando uma classe implementa uma interface, ele obrigatoriamente tem que implementar todos os métodos definidos em uma interface.

Olhe o arquivo `Lata.java`. Na declaração da classe temos:

```
public class Lata implements Empilhavel{
```

Observe a palavra `implements`, ela está dizendo que essa classe implementa a interface `Empilhavel`. A partir desse momento, uma `Lata` é uma lata, mas também é "uma coisa Empilhavel".

Da mesma maneira a classe `BatataFrita` também é uma coisa `Empilhavel`.

No futuro, se quisermos mais coisas que possam ser empilhadas, independentemente do que sejam, basta fazer a classe implementar a interface.

Para finalizar, vale dizer que a interface faz qualquer classe *ser* o que ela implementa, isto é, *lata é empilhável*. Além disso, se você reparou bem, as chamadas da classe `Pilha` são chamadas polimórficas. Veja a linha 31, por exemplo. O resultado de `getDescricao()` dependerá de qual objeto for passado como parâmetro.

Mas, salvo essas semelhanças, heranças e interfaces são coisas diferentes:

- **Herança** é um relacionamento estrutural, que reune conceitos comuns, com alguns atributos e métodos comuns.
- **Interface** é uma definição de comportamento esperado da classe.

Uma última informação: uma classe pode implementar quantas interfaces quiser.

Boa noite!

Desafio para estudo

Vamos fazer umas brincadeiras.

- na classe de controle, a partir da linha 7, empilhe um nova lata. Veja o resultado.
- crie uma classe `GarrafaCoca.java` que implemente `Empilhavel`. Use o código da classe `Lata` para guiar o que você deverá implementar. Use sua nova classe na pilha.

Capítulo 25

Mostrando mensagens na tela (Doc! Voltamos no tempo?!)

Neste exercício você irá escrever um programa funcional em Java para mostrar uma mensagem importante na tela.

Você já leu essa frase. Foi assim que começamos o primeiro capítulo do primeiro livro [1], e é assim que vamos começar a aprender a criar interfaces gráficas. Até agora o resultado de tudo que fizemos foi apresentado em forma de texto. Contudo, nos próximos capítulos, vamos aprender a criar janelas com mensagens, imagens, campos para inclusão de informações e chegaremos a criar um formulário completo.

E, começando do começo, neste capítulo vamos aprender a criar uma janela e colocar uma mensagem para o usuário nela.

Diga adeus à tela preta! A Era Visual chegou na sua vida!

```
1  import javax.swing.JOptionPane;
2
3  public class Mensagem
4  {
5    public static void main(String[] args)
6    {
7      JOptionPane.showMessageDialog(null, "Ola Mundo");
8    System.exit(0);
9    }
10 }
```

O que você deve ver

Figura 25.1: Resultado da execução do programa Mensagem.java

Legal não é? Você não imaginava que pudesse fazer tanto com tão pouco código, certo? Apesar de simples, temos muito a discutir sobre esse código. A começar pela primeira linha: Estamos importando uma classe que você não conhecia. A `JOptionPane` é uma classe que nos ajuda a criar objetos visuais na tela e faz parte do pacote `Swing`.

O Swing, por sua vez, é um framework [2]que oferece meios de construção de telas usando código Java. Trata-se de uma das duas[3] bibliotecas gráficas oferecidas pelo Java nativamente. Isso quer dizer que estão disponíveis em qualquer versão do Java acima de 1.2. Portanto, não se

[1] "Aprenda Java na Marra", disponível em javanamarra.com.br...
[2] Costumo definir framework como "um conjunto de classes dedicadas a facilitar nossa vida."
[3] A outra é o AWT, a primeira a ser criada e já superada pelo Swing

preocupe com compatiblidade. Esse código vai funcionar em qualquer máquina que rode Java, seja Unix, Windows®, iOS®, etc...

Dentre as classes disponíveis no Swing estão as que criam janelas, formas, imagens, barras de rolagem, campos texto e, também, o `JOptionPane`, que cria caixas de diálogo.

Então, na linha 7 do nosso programa usamos um método **estático** da classe `JOptionPane` para criar uma caixa de diálogo de *mensagem* (você já sabe o que significa "'estático" em nosso mundo).

Esse método recebe como parâmetro a janela à qual essa caixa de diálogo pertence, no caso, nenhuma (null), e um texto qualquer a ser exibido dentro da caixa de diálogo.

Perceba que você não configurou nenhuma imagem em seu código e, mesmo assim, quando a caixa foi criada havia um ícone de informação ao lado da mensagem. Também havia um botão "OK", um botão com um "X" no canto superior direito e um título na caixa de diálogo. Nada disso foi explicitamente programado por você!

Isso acontece por que o Java considera todos esses elementos essenciais para a caixa de diálogo e, como você não os programou, o java tratou de colocar para nós. Vamos aprender, nos próximos capítulos, a dar nossa cara às caixas de diálogo, janelas e demais componentes visuais.

A linha 8 trata de finalizar o programa. Ela é necessária porque na linha 7 criamos um diálogo sem pai (passando `null` no parâmetro correspondente à janela dona da caixa). Ao usar o `JOptionPane` dessa forma deixamos que o Java crie uma janela invisível, sobre a qual não temos controle, para colocar a caixa. E já que não temos controle, a única forma de finalizar o programa é com o comando da linha 8. Relaxe! Vamos esclarecer essa questão de janela pai e filha antes de precisarmos acionar a vara de família! :-)

Desafios para estudo

1. Adicione mais duas caixas de diálogo em seu programa repetindo o comando da linha 7. Use textos diferentes para cada caixa.

2. Observe o comportamento do programa. As caixas foram criadas ao mesmo tempo na tela ou a segunda só apareceu após você clicar no "OK" da primeira?

Capítulo 26

Conversa fiada - Explorando diálogos com `JOptionPane`

Caixas de diálogo são a maneira mais simples de interagir com seu usuário e mantê-lo informado do que se passa no sistema. Essas caixas também são úteis para que o usuário digite informações ou tome decisões sobre o comportamento da aplicação. No exemplo abaixo vamos continuar explorando a classe `JOptionPanel` e estudando o que ela tem a oferecer.

```
1  import javax.swing.JOptionPane;
2  public class Previsao {
3      public static void main(String[] args) {
4
5          String[] signos = {"Aries","Touro","Cancer","Gemeos","Virgem","Leao","
       Capricornio","Aquario","Peixes","Sagitario","Libra"};
6          String[] sorte = {"Sorte", "Azar"};
7          String[] campoDaVida = {"Amor","Trabalho","Dinheiro","Saúde"};
8
9          JOptionPane.showMessageDialog(null, "Bem vindo! Vamos ler sua sorte!", "Guru
        da magia do alem", JOptionPane.QUESTION_MESSAGE  );
10
11         String nome = JOptionPane.showInputDialog(null, "Qual seu nome?", "
       Identificacao", JOptionPane.QUESTION_MESSAGE);
12
13         int outra;
14
15         do {
16             int codigoSigno = JOptionPane.showOptionDialog(null, "Escolha seu signo!
       ", "Signo", JOptionPane.DEFAULT_OPTION, JOptionPane.QUESTION_MESSAGE, null,
       signos, null);
17
18             int previsaoSorte = (int) (Math.random() * 10) % 2;
19             int previsaoCampo = (int) (Math.random() * 10) % 4;
20
21             String mensagem = nome + ", a previsao para " + signos[codigoSigno] + "
       hoje eh " + sorte[previsaoSorte] + " no " + campoDaVida[previsaoCampo];
22
23             JOptionPane.showMessageDialog(null, mensagem ,"Guru da magia do alem",
       JOptionPane.INFORMATION_MESSAGE );
24
25             outra = JOptionPane.showConfirmDialog(null, "Quer fazer outra consulta?"
       );
26         }
27         while (outra == JOptionPane.YES_OPTION);
28
29     System.exit(0);
30     }
31 }
```

O que você deve ver

Embora tenhamos usado sempre a classe `JOptionPane`, você deve ter notado que as chamadas foram bastante diferentes da chamada apresentada no capítulo anterior.

A começar pela linha 9, vemos que há uma chamada ao método que já conhecemos, `showMessageDialog`, que dessa vez recebeu cinco parâmetros. O primeiro deles, a tela pai da

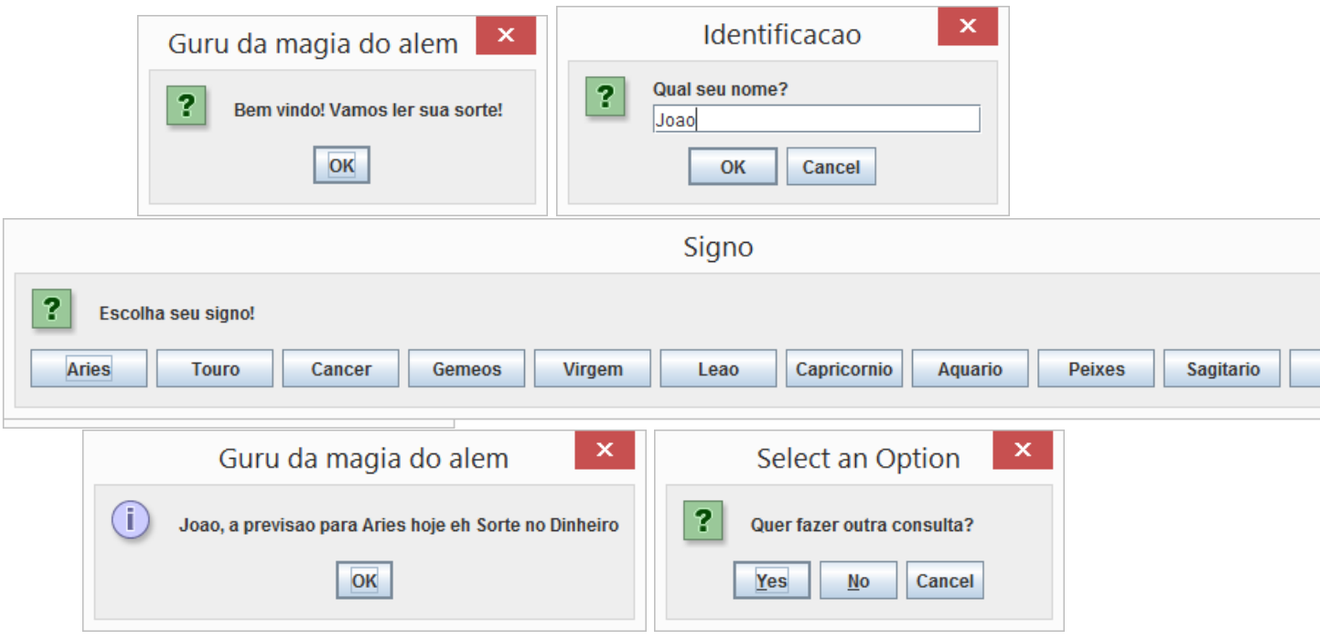

Figura 26.1: Resultado da execução do programa Previsao.java

caixa que decidimos deixar vazio (null). Depois, a mensagem que gostaríamos de exibir para o usuário. Nenhuma novidade até aqui.

O terceiro parâmetro define um título para nossa caixa de diálogo. Ele é uma `String` opcional que fica na barra de título da janela (Antes do X para fechá-la).

O quarto parâmetro é o tipo de ícone a ser exibido do lado esquerdo da caixa de diálogo. São cinco opções disponíveis: `ERROR_MESSAGE` (Erro), `INFORMATION_MESSAGE` (informação), `WARNING_MESSAGE` (aviso), `QUESTION_MESSAGE` (pergunta) e `PLAIN_MESSAGE` (sem ícone). Eu usei a opção de pergunta só para mostrar que podemos usá-lo mesmo que não estejamos perguntando nada (e também para dar um ar de mistério ao programa. Funcionou?!)

Na linha 11 temos algo um pouco mais interessante. Perceba que declaramos uma `String` chamada *nome* e atribuímos a ela o resultado da chamada a um método da `JOptionPane`. Seu instinto de programador já te diz o que acontece aqui: essa função retorna uma `String` após sua execução. Mas não é uma `String` qualquer! Dê uma olhada no nome do método: `showInputDialog`. Sabemos que "input" significa "entrada" em inglês [1] então, pelo nome, percebemos que esse método vai mostrar uma caixa de diálogo que permite ao usuário fazer um *input*, ou seja, informar dados. Aquilo que usuário digitar será o retorno da função e, no nosso caso, será armazenado na variável *nome*.

Os parâmetros recebidos pela `showInputDialog` são os mesmos informados para a `showMessageDialog`: janela pai, mensagem, título e tipo de ícone.

Na linha 16 entramos na primeira instrução do bloco de um laço `do-while`. Essa linha declara uma variável do tipo `int` chamada *codigoSigno* e preenche essa variável com o retorno de um método da classe `JOptionPane` chamado `showOptionDialog`. Ainda não sabemos o que esse método faz, mas temos certeza de que seu retorno é um `int`, não é? [2] Pelo nome do método percebemos que ele exibe uma caixa de diálogo com *opções* para o usuário.

O primeiro parâmetro do método é a janela pai, o segundo a mensagem, o terceiro o título da caixa. O quarto parâmetro seria o tipo de opções que queremos dar ao usuário. Se quiséssemos dar opções "sim" e "não", usaríamos `YES_NO_OPTION`, se nosso desejo fosse permitir "sim", "nao" e "cancelar", `YES_NO_CANCEL_OPTION` e se quiséssemos apenas "ok" e "cancelar" usaríamos `OK_CANCEL_OPTION`. Entretanto, não estamos interessados em nenhuma dessas opções, já que o que queremos é exibir uma lista de opções customizada ao nosso usuário. Assim, usamos `DEFAULT_OPTION`. A seguir informamos o tipo de ícone dentre as opções padrão. Se quiséssemos mostrar uma imagem customizada como ícone poderíamos usar o parâmetro seguinte, mas preferimos deixar nulo (null). O penúltimo parâmetro é muito importante, pois trata-se da lista de opções que nosso

[1] Se não sabia, aprendeu agora!

[2] A essa altura você já deve ter essa sagacidade: mesmo que não conheça o método você é capaz de entender seu retorno se esse retorno estiver sendo atribuído a uma variável, pois o retorno e a variável são, obrigatoriamente, de tipos compatíveis.

usuário poderá escolher. Passamos como parâmetro o vetor de `String`s declarado e inicializado na linha 5. Isso quer dizer que cada elemento do vetor consistirá em um botão clicável na caixa de diálogo e o retorno do método será exatamente o índice do vetor correspondente ao valor escolhido pelo usuário.

Nas linha 18 e 19 sorteamos dois números, de 0 a 1 e de 0 a 3 respectivamente.

A linha 21 monta a sorte do usuário, concatenando seu nome, o signo escolhido, a sorte e o campo da vida sorteados. (perceba que nossas previsões são cuidadosamente calculadas e, portanto, infalivelmente confiáveis!)

Para quem sobreviveu às linhas anteriores, a linha 23 é moleza: exibimos a mensagem montada na linha 21 usando o ícone de informação.

Finalmente, antes de chegar ao fim do bloco `do-while`, usamos um método diferente do `JOptionPane`, o `showConfirmDialog`. Ele exibe uma caixa de diálogo que tem como opções "Sim", "Não" e "Cancelar" e retorna o código da opção selecionada pelo usuário. Guardamos essa resposta na variável `outra` e utilizamos seu valor para avaliar, na linha 27, se devemos repetir o bloco do laço ou se devemos sair do programa na linha 29.

Desafios para estudo

1. Adicione uma caixa de diálogo do tipo "sim" ou "nao" logo após informar a sorte do usuário para perguntar se ele gostou da previsão.
2. Caso ele tenha gostado, exiba uma mensagem dizendo: "Que bom que gostou". Se ele não gostou da previsão, exiba uma mensagem dizendo: "Que pena, mas os astros não mentem!".

Capítulo 27

Painéis e janelas

Quando desenvolvemos interfaces visuais o que fazemos na prática é criar objetos que, de alguma maneira, têm uma manifestação visual na tela. No Java Swing esses objetos precisam estar inseridos em um painel. Pense no painel como um mural de corredor de escola, no qual as pessoas colocam avisos. Cada aviso é um elemento visual que compõe o mural. Da mesma forma, cada elemento visual da nossa interface precisa estar "pendurada" em algum "mural".

Nos capítulos anteriores não nos preocupamos com isso. Isso fez com que, nos nossos exemplos, o Java criasse um painel interno e invisível para pendurar nossas caixas de diálogo. Nesse capítulo vamos tomar as rédeas da situação e criar nossos próprios painéis.

```
1  import javax.swing.*;
2
3  public class Mesa extends JFrame {
4    public Mesa() {
5      setTitle("Mesa com matrioskas");
6      setSize(800, 600);
7
8      String num = JOptionPane.showInputDialog(this, "Quantas matrioskas quer ver?", "
       Quantidade de bonecas", JOptionPane.QUESTION_MESSAGE);
9      int numero = Integer.parseInt(num);
10
11     Matrioska novamatrioska = new Matrioska(numero);
12     novamatrioska.setVisible(true);
13   }
14
15   public static void main(String args[]) {
16     Mesa f = new Mesa();
17     f.setVisible(true);
18     f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19     f.toBack();
20   }
21 }
```

```
1  import javax.swing.*;
2  import java.awt.*;
3
4  public class Matrioska extends JFrame {
5    private final int larguraMaxima = 800;
6    private final int alturaMaxima = 600;
7
8    public Matrioska(int numero) {
9      setTitle("Matrioska " + numero);
10
11     int altura = alturaMaxima / numero;
12     int largura = larguraMaxima / numero;
13
14     setSize(largura, altura);
15
16     String mensagem = "Oi, eu sou a Matrioska " + numero + " !";
17
18     JLabel texto = new JLabel(mensagem);
19     getContentPane().add(texto);
20
21     JLabel texto2 = new JLabel("teste");
22     getContentPane().add(texto2, BorderLayout.PAGE_END);
23
24
25     texto.setVerticalTextPosition(JLabel.BOTTOM);//this.setLayout(new FlowLayout());
```

```
26
27      if (numero > 1) {
28        Matrioska novaMatrioska = new Matrioska(--numero);
29        novaMatrioska.setVisible(true);
30      }
31    }
32  }
```

O que você deve ver

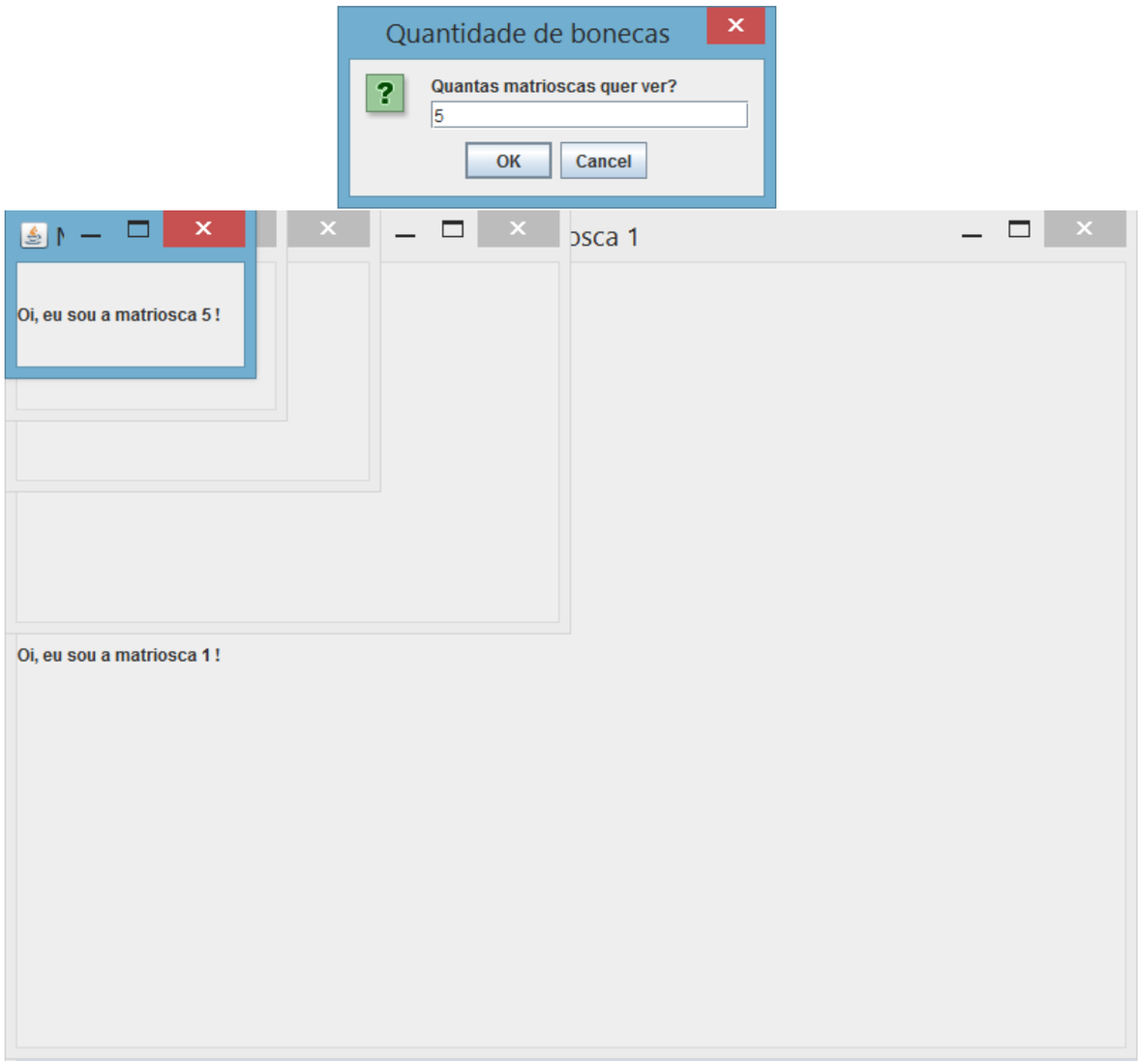

Figura 27.1: Resultado da execução do programa Mesa.java

Esse programa tem um pouco mais de linhas que o exemplo anterior, e conta com duas classes ao invés de uma. Ainda assim, é um programa simples. Aliás, você vai perceber que a programação visual com Java Swing não tem nada de muito diferente do que você já domina em termos de codificação. Tudo se resume à criação de objetos, configuração desses objetos e invocação de métodos.

Esse exemplo cria várias telas de tamanhos diferentes uma em cima da outra, como se fossem Matrioskas [1]. Para isso temos uma classe representando a mesa e outra representando as matrioskas.

O arquivo Mesa.java define uma classe chamada Mesa que estende `JFrame`. Isso faz com que essa classe sejá um `JFrame` e tenha todos os atributos e métodos que um `JFrame` tem, dentre eles, a capacidade de se apresentar visualmente na tela e de servir de contêiner para outros elementos visuais. Assim, sempre que uma classe estender `JFrame` ela sera capaz de conter outros elementos

[1] Matrioska é um brinquedo tradicional da Rússia, constituída por uma série de bonecas, feitas de diversos materiais, que são colocadas umas dentro das outras, da maior até a menor. Não sabia o nome? Tudo bem, eu também não...

visuais como botões, campos de digitação, textos, caixas de diálogo, e quaisquer outros elementos, inclusive outros `Frame`s.

Em seguida, na linha 4 definimos o construtor dessa classe. A linha 5 define o título da janela e a linha 6 define suas dimensões em pixels (no nosso caso, 800 pixels[2] de largura e 600 pixels de altura). Perceba que `setTitle` e `setSize` são métodos herdados da ancestral `JFrame`.

A linha 8 usa um recurso que já conhecemos, o `JOptionPane`, para criar uma caixa de diálogo que pergunta ao usuário quantas Matrioskas ele quer criar. Perceba que, diferentemente dos exemplos dos capítulos anteriores, dessa vez passamos um valor diferente de `null` como primeiro parâmetro do método `showInputDialog`. Você já sabe que esse parâmetro se refere ao elemento contenedor da caixa de diálogo, ou seja, o pai dela. O que usamos dessa vez é o `this`, que voce já sabe o que é. Nesse caso o `this` é a instância de `Mesa` em execução. Isso significa que ela é a "dona" dessa caixa de diálogo. Na prática, a diferença é que quando a caixa de diálogo for fechada o foco volta para a janela `Mesa`.

Na linha 9 apenas convertemos o valor digitado pelo usuário, `String`, para o tipo que queremos, `int`.

Agora de uma olhadinha na linha 11. Trata-se de uma criação de objeto comum, certo? Sim! vamos, então, estudar o código da classe Matrioska.

Na linha 3 logo percebemos a instrução `extends JFrame`. Opa! já sabemos o que isso quer dizer! Essa classe define um objeto visual que pode servir de contêiner para outros objetos visuais. Legal!

No construtor dessa classe, na linha 8, definimos o título da janela, da mesma maneira que fizemos na classe `Mesa`.

Nas linhas 10 e 11 calculamos as dimensões da janela. Queremos imitar um jogo de Matrioskas, então queremos criar janelas de tamanhos diferentes, cada uma menor que a outra. Assim, estamos, aqui, apenas dividindo as dimensões pelo número da Matrioska. Se for a primeira, número 1, seu tamanho será o original. Se for a segunda, terá metade do tamanho original (`alturaMaxima/2` e `larguraMaxima/2`), Se for a terceira terá um terço do tamanho original, e assim por diante. Por isso recebemos o número da boneca no construtor (linha 7).

Na linha 13 usamos os valores de altura e largura calculados para definir o tamanho da janela. Sem mistério aqui... A linha 15 é igualmente simples e faz somente a montagem de um texto usando o número da Matrioska.

Na linha 18 o seu radar de coisas novas apitou! Mas trata-se de algo extremamente simples. Vou te dar apenas a tradução da palavra *Label*, que em português significa *Rótulo* e te pedir que volte a estudar as linhas 18 e 19 por 30 segundos e tente entender o que está acontecendo...

tic-tac... tic-tac...

Conseguiu? Aposto que sim. Você já começou a perceber que os componentes Swing começam com a letra J. Então, `JLabel` é um componente Swing. Eu te contei que Label é Rotulo em inglês, você viu que essa classe recebeu a mensagem que montamos em seu construtor. Em seguida você viu que esse `JLabel` foi adicionado (método `add`) a algum objeto retornado por um método da própria classe Matrioska. Sendo a Matrioska um `JFrame` você logo percebe o que esta acontecendo aqui: Na linha 17 criamos um `JLabel` para exibir um texto e na linha 18 adicionamos esse texto à janela que estamos construindo.

Isso conclui a montagem dessa tela. Ela tem um título (linha 8), dimensões definidas(linha 13) e um componente visual associado: um texto (`JLabel`, criado na linha 17 e adicionado na tela na linha 18).

Como estamos simulando as bonecas uma dentro da outra, precisamos criar quantas bonecas o usuário solicitou. Assim, deixamos que cada boneca crie a seguinte até que a quantidade de bonecas a criar se esgote. Por isso há um comando `if` na linha 20. Se estivermos criando a boneca de número 1 significa que já criamos todas e, portanto, saímos sem fazer nada. Caso contrário, precisamos criar a próxima boneca, que será um numero menor que a atual (por isso o comando `--numero` dentro do construtor da Matrioska.).

[2] um pixel pode ser entendido como o menor ponto que se pode desenhar na tela. uma sequencia de pixels alinhados forma uma reta, por exemplo.

Por último, mas não menos importante, definimos que a Matrioska recém criada deva ser visível, usando o comando da linha 22. Por padrão os `JFrame`s são criados invisíveis, e todo código escrito não teria nenhum efeito visual sem esse comando. Voltando ao `Mesa.java`, esse comando também é executado na linha 12, para fazer a primeira Matrioska criada também ficar visível.

Seguindo as regras do jogo, precisamos de um método `main` em alguma classe, que será eleita a classe controladora do fluxo do programa. Então temos esse método na linha 15. Ele apenas cria uma nova `Mesa`, define que ela sejá visível e faz mais dois ajustezinhos referentes a componentes visuais.

O primeiro, linha 18, ajuste define o comportamento quando o X no canto direito superior da janela for clicado. Nesse caso estamos dizendo para o Java que queremos que o programa inteiro sejá encerrado.

O segundo ajuste, linha 19, define que essa janela fique atrás de todas as demais. Repare que, apesar da criação da `Mesa` ter sido a primeira a ser iniciada, ela é a última a ser concluída, pois em seu construtor ela cria uma `Matrioska`, que por sua vez cria outra, e outra e outra quantas forem as bonecas solicitadas pelo usuário, e só depois de todas as `Matrioska`s criadas é que a criação da `Mesa` é concluída. No Java Swing o comportamento padrão é que os objetos vão se sobrepondo a medida que sua criação é concluída. Então, como queremos a mesa atrás de tudo, temos que forçar esse posicionamento.

Desafios para estudo

1. Adicione mais um JLabel à Matrioska, com linhas semelhantes às 18 e 19, mas use uma mensagem diferente (por exemplo, “Vamos beber vodka!").

2. Observe o resultado. Aconteceu o que você esperava? Acredito que não. Vire a página e vamos aprender mais sobre o posicionamento de elementos em um JFrame.

Capítulo 28

Posicionando elementos visuais

No capítulo anterior, ao implementar o desafio, você tentou colocar dois `JLabel` no mesmo `JFrame` e se deu mal. O resultado foi que o segundo `JLabel` se sobrepôs ao primeiro (percebeu isso?).

Até agora trabalhamos com componentes visuais, mas não nos preocupamos em definir a posição desses componentes na tela. Deixamos o Java desenhar cada elemento onde achasse melhor. Mas, como vimos, nem sempre o Java consegue tomar as melhores decisões sozinho. Ele precisa de alguma dica de como queremos que as coisas sejam desenhadas.

Nesse capítulo vamos aprender como orientar o Java Swing a respeito do posicionamento dos objetos que vamos adicionando aos `JFrame`.

Para tanto, digite o código abaixo, que simula o posicionamento de atletas em diversos esportes.

```
1  import java.awt.*;
2  import javax.swing.*;
3
4  public class EducacaoFisica {
5      public static void main(String args[]) {
6          Formacao volei = new Formacao("Quadra de volei");
7          GridLayout voleiLayout = new GridLayout(0,3);
8          volei.getContentPane().setLayout(voleiLayout);
9          volei.setVisible(true);
10
11         Formacao remo = new Formacao("Barco de remo");
12         BoxLayout remoLayout = new BoxLayout(remo.getContentPane(),BoxLayout.Y_AXIS)
       ;
13         remo.getContentPane().setLayout(remoLayout);
14         remo.setVisible(true);
15
16         Formacao hino = new Formacao("Hora do hino nacional");
17         FlowLayout hinoLayout = new FlowLayout();
18         hino.getContentPane().setLayout(hinoLayout);
19         hino.setVisible(true);
20
21         FormacaoFutsal futsal = new FormacaoFutsal();
22         futsal.setVisible(true);
23     }
24 }
```

```
1  import javax.swing.*;
2
3  public class Formacao extends JFrame {
4      public Formacao(String nome) {
5          setTitle(nome);
6
7          setSize(400,200);
8
9          JLabel texto = new JLabel("Atleta 1");
10         getContentPane().add(texto);
11
12         JLabel texto2 = new JLabel("Atleta 2");
13         getContentPane().add(texto2);
14
15         JLabel texto3 = new JLabel("Atleta 3");
16         getContentPane().add(texto3);
17
18         JLabel texto4 = new JLabel("Atleta 4");
```

```
19          getContentPane().add(texto4);
20
21          JLabel texto5 = new JLabel("Atleta 5");
22          getContentPane().add(texto5);
23
24          JLabel texto6 = new JLabel("Atleta 6");
25          getContentPane().add(texto6);
26      }
27  }
```

```
1  import java.awt.BorderLayout;
2  import javax.swing.*;
3
4  public class FormacaoFutsal extends JFrame {
5      public FormacaoFutsal(){
6          setTitle("Quadra de futsal");
7
8          setSize(400,200);
9
10         JLabel texto = new JLabel("Goleiro");
11         getContentPane().add(texto, BorderLayout.PAGE_START);
12         texto.setHorizontalAlignment(JLabel.CENTER);
13
14         JLabel texto2 = new JLabel("Ala Esquerda");
15         getContentPane().add(texto2, BorderLayout.LINE_START);
16
17         JLabel texto3 = new JLabel("Zagueiro");
18         getContentPane().add(texto3, BorderLayout.CENTER);
19         texto3.setHorizontalAlignment(JLabel.CENTER);
20
21         JLabel texto4 = new JLabel("Ala direita");
22         getContentPane().add(texto4, BorderLayout.LINE_END);
23
24         JLabel texto5 = new JLabel("Atacante");
25         getContentPane().add(texto5, BorderLayout.PAGE_END);
26         texto5.setHorizontalAlignment(JLabel.CENTER);
27     }
28 }
```

O que você deve ver

Como se sente no papel de treinador? Foi fácil colocar os atletas na posição desejada? Já se sente apto a assumir nossa seleção? Garanto que nosso atual treinador não tem tanto controle sobre seus atletas quanto você teve nesse exercício.

E assim como nos esportes reais, no posicionamento de elementos visuais do Swing você se limita a dizer ao elemento **como** ele deve se posicionar. Quem escolhe onde ficar ainda é o componente, mas obedecendo suas diretrizes. Vamos entender isso com mais detalhes!

Comecemos estudando o código da classe `Formacao`. Na linha 3 percebemos que ela estende `JFrame`, o que faz dela um Frame, ou seja, uma janela visual que pode conter outros elementos visuais.

Nas linhas 5 e 7 definimos o título e o tamanho da janela.

Na linha 9 criamos um `JLabel`, um componente visual capaz de exibir textos. Já conhecemos esse componente do capítulo anterior. Na linha 10 adicionamos esse texto ao nosso `JFrame` por meio de seu painel.

Nós estamos obtendo o painel com o comando `getContentPane`. O que é o painél? Todo `JFrame` tem um `JPane` associado, que é o objeto que, de fato, contém os elementos visuais. Por isso adicionamos os elementos usando `getContentPane()` do `JFrame` e não adicionamos ao `JFrame` diretamente. Na prática, pelo menos por enquanto, isso não muda em nada nossa vida.

Código semelhante ao das linhas 9 e 10 é repetido mais cinco vezes, totalizando seis atletas criados nessa janela.

Perceba que, da forma que está criado, esse `JFrame` se comportaria como o seu programa de matrioskas depois do desafio, ou seja, os textos iriam se sobrepor e você só veria o ultimo atleta adicionado ao `JFrame`.

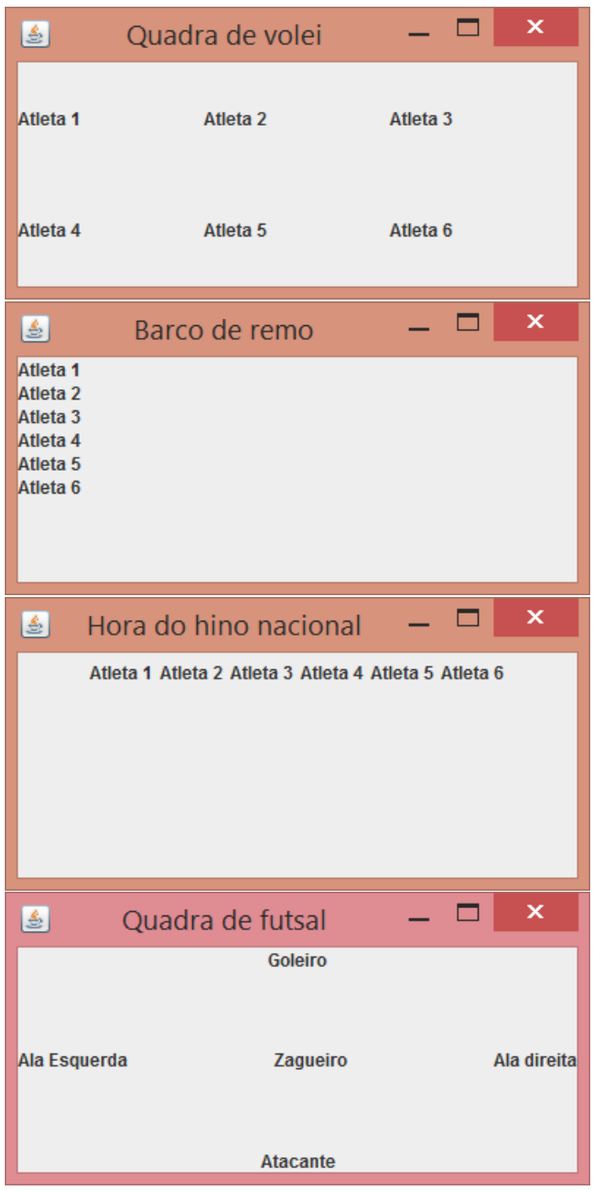

Figura 28.1: Resultado da execução do programa EducacaoFisica.java

Passemos, portanto, a estudar o código da classe `EducacaoFisica`. Ela **não** é um `JFrame`, mas apenas uma classe controle, onde nosso programa começa. Assim sendo, seu método `main` cria na linha 6 um objeto do tipo `Formacao` chamado `volei` e passa como parâmetro o texto "Quadra de volei". Como vimos, esse texto será usado para definir o título da janela.

Agora o show começa!

Na linha 7 criamos um objeto chamado `voleiLayout` do tipo `GridLayout`. Trata-se de uma classe presente no pacote `java.awt` (por isso o include da linha 1) que define um tipo de configuração visual para nosso `JFrame`.

Os layouts são classes que implementam padrões de disposição de objetos visuais em `JFrame`s de acordo com regras internas. Nessa caso, o `GridLayout` transforma nossa janela em um "tabelão"imaginário e coloca cada componente visual dela em uma célula imaginária. No exemplo, quando passamos no construtor da classe os valores `0` e `3` (linha 7: `new GridLayout(0,3);`), estamos dizendo para o `GridLayout`: "Por favor, quando for desenhar a tela, utilize uma tabela imaginária de três colunas e quantas linhas forem necessárias".

Em seguida, na linha 8, configuramos nosso `JFrame volei` com esse layout, já definido para montar *grids* de três colunas.

Quando executamos o comando da linha 9, o Swing não só torna visível a tela, desenhando nossos textos, como o faz obedecendo ao layout que definimos, qual seja, itens dispostos como se estivessem em uma tabela com três colunas.

De maneira muito semelhante criamos a janela que simula um barco de remadores. A

linha 11 cria outro objeto do tipo `Formacao`, dessa vez para dispor os atletas como membros de uma equipe de remo, ou seja, um atrás do outro. Conseguimos isso usando o `BoxLayout`

O `BoxLayout` é uma classe que define uma maneira de desenhar objetos visuais na tela alinhados vertical ou horizontalmente. Para fazer isso usamos a sintaxe da linha 12. Criamos um novo objeto do tipo `BoxLayout` e passamos dois parâmetros: o `Jpane` a ser utilizado, no caso o `Jpane` do objeto recém criado `remo` e o tipo de alinhamento, vertical (`BoxLayout.Y_AXIS`) ou horizontal (`BoxLayout.X_AXIS`).

A linha 13 faz o mesmo que a linha 8, atribui o layout criado ao `JFrame`.

Assim, quando a linha 14 torna nosso barco de remo visível, os elementos estão alinhados verticalmente, como queríamos.

Repare que usamos a mesma classe, `Formacao`, para gerar duas exibições completamente diferentes, apenas alternando entre um layout e outro. Não fizemos nenhuma alteração na classe `Formacao` e mesmo assim fomos capazes de alterar sensivelmente a maneira como ela se apresenta visualmente, definindo seu layout. Na linha 16 definimos mais um exemplo de como alterar a disposição dos elementos visuais usando apenas definições de layouts distintas.

A linha 16 cria mais um objeto do tipo `Formacao`, dessa vez com o título "Hora do hino nacional".

Na linha 17 criamos um layout chamado `FlowLayout`. Como o nome sugere (flow = fluxo), esse layout desenha os objetos na tela um ao lado do outro, horizontalmente, até o limite da janela que os contém. Se os elementos não couberem na tela ele simplesmente quebra linha e segue o fluxo.

Trata-se de um layout mais simples, que não recebe nenhum parâmetro especial. Assim, as linhas 17, 18 e 19 são bastantes simples de se entender: estamos criando o layout, atribuindo ao `JFrame` criado e fazendo esse `JFrame` visível.

(respire antes de irmos para o ultimo layout...)

Na linha 21 criamos um objeto do tipo `FormacaoFutsal`. Na linha 22 chamamos um método desse objeto chamado `setVisible`. Por isso você, ligeiro no raciocínio, já suspeita de que esse objeto seja um `JFrame`. Vamos lá no `FormacaoFutsal.java` conferir!

Linha 4: ... `extends JFrame`. Trata-se de um `JFrame`, ou seja, uma janela que pode conter elementos visuais.

O construtor da classe, que começa na linha 5, não tem nada de diferente do que vimos utilizando. `setTitle` na linha6, `setSize` na linha 8, criação de um `JLabel` na linha 10.

A linha 11 nos traz novidade. Ao adicionar o texto ao painel, passamos um segundo parãmetro que, até então, não utilizávamos. Isso é necessário sempre que desejarmos utilizar o layout `BorderLayout`, que é, por sinal, o layout padrão do `JFrame`.

O `BorderLayout` coloca cada elemento em uma de cinco áreas: Em cima, embaixo, à direita, à esquerda e ao centro.

Assim, na linha 12 colocamos o goleiro no topo da página, usando o parãmetro `BorderLayout.PAGE_START`. Na linha 15 colocamos o ala esquerda no começo da linha, ou seja, alinhado à esquerda, usando `BorderLayout.LINE_START`. O Zagueiro ficou no meio da tela: `BorderLayout.CENTER`. O ala direita ficou à direita da tela, no final da linha, ou seja `BorderLayout.LINE_END`. E por fim, o atacante fica embaixo, no fim da tela: `BorderLayout.PAGE_END`.

Por padrão, o JLabel se alinha o máximo à esquerda que puder. Isso não afeta layouts como o `BorderLayout.LINE_START` e o `BorderLayout.LINE_END`, pois esses posicionamentos não são ambíguos.

Contudo, layouts como o `BorderLayout.PAGE_START` indicam ao elemento visual que ele deve ficar no topo da página, mas não define exatamente em que parte do topo da página ele deve ficar. Essa ambiguidade é resolvida pelo `JLabel` com alinhamento à esquerda.

Assim, para deixar a formação do time do jeito que queremos, precisamos, também, do comando das linhas 12, 19 e 26. Esse comando define o alinhamento horizontal de um `JLabel`. Em nosso caso, alteramos o padrão para `JLabel.CENTER`.

Perceba que o código da `FormacaoFutsal` termina logo após a criação dos `JLabel`s, e o código da `EducacaoFisica` em momento algum define o layout para a formação de futsal. Isso mostra que o `BorderLayout` é mesmo o layout padrão para os `JFrame`, mas para que ele funcione o segundo parâmetro do método `add` deve ser informado.

Desafios para estudo

1. Com o mouse na borda de cada janela, aumente e diminua as dimensões dela. Perceba, em cada layout, como se posicionam os jogadores à medida em que a diminuição da tela os atinge.
2. Altere a classe `EducacaoFisica` na linha 7 para que o `GridLayout` tenha duas colunas. Observe como fica o time de volei.
3. Altere a classe `EducacaoFisica` na linha 12 para que os remadores se alinhem horizontalmente (usando `BoxLayout.X_AXIS`). Observe como se dispõem os remadores.
4. Na classe `FormacaoFutsal` remova os comandos das linhas 12, 19 e 26 e observe como fica o time.

Capítulo 29

Campos Texto e Botões

Muito bem! você aprendeu a posicionar elementos visuais na tela. Mas até agora estávamos trabalhando apenas com textos estáticos, e isso é muito chato. Então, chegou a hora de conhecermos componentes mais interessantes! Vamos começar com campos texto e botões que nos permitirão receber informações digitadas pelo usuário e processá-las adequadamente de acordo com o botão pressionado.

Nesse exemplo, construiremos um conversor da criptografia Zenit Polar. Trata-se de um algoritmo de substituição que consiste em trocar as letras usando a seguinte regra: Se a letra for Z, troca-se por P, se a letra for E, troca-se por O, se a letra for N, troca-se por L, e assim por diante. A palavra "Algoritmo", então, ficaria "Ingetarme". Funciona bem né?

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  public class ZenitPolar extends JFrame implements ActionListener{
6      JLabel textoCripto;
7      JTextField campoTexto;
8
9      public ZenitPolar(){
10         setTitle("Criptografia Zenit Polar");
11         setSize(400,150);
12
13         JLabel texto = new JLabel("Digite o texto a ser criptografado");
14         getContentPane().add(texto);
15
16         campoTexto = new JTextField(20);
17         getContentPane().add(campoTexto);
18
19         JButton botaoCripto = new JButton("Criptografar!");
20         botaoCripto.setActionCommand("CONVERTER");
21         botaoCripto.addActionListener(this);
22         getContentPane().add(botaoCripto);
23
24         JButton botaoLimp = new JButton("Limpar texto");
25         botaoLimp.setActionCommand("LIMPAR");
26         botaoLimp.addActionListener(this);
27         getContentPane().add(botaoLimp);
28
29         JLabel texto2 = new JLabel("O texto criptografado ficou:");
30         getContentPane().add(texto2);
31
32         textoCripto = new JLabel();
33         getContentPane().add(textoCripto);
34
35         GridLayout layout =  new GridLayout(0,2);
36         this.setLayout(layout);
37
38     }
39
40     public void actionPerformed(ActionEvent e) {
41         if(e.getActionCommand().equals("CONVERTER")) {
42       String digitado = campoTexto.getText();
43             String convertido = converteZenitPolar(digitado);
44             textoCripto.setText(convertido);
45         }
46         else if (e.getActionCommand().equals("LIMPAR")) {
47             textoCripto.setText("");
```

```
48              campoTexto.setText("");
49          }
50      }
51
52      private String converteZenitPolar(String aConverter) {
53          String convertido = aConverter.toLowerCase();
54
55          convertido = convertido.replace("z", "P");
56          convertido = convertido.replace("e", "O");
57          convertido = convertido.replace("n", "L");
58          convertido = convertido.replace("i", "A");
59          convertido = convertido.replace("t", "R");
60          convertido = convertido.replace("p", "Z");
61          convertido = convertido.replace("o", "E");
62          convertido = convertido.replace("l", "N");
63          convertido = convertido.replace("a", "I");
64          convertido = convertido.replace("r", "T");
65
66          return convertido.toUpperCase();
67      }
68
69      public static void main(String args[]) {
70      ZenitPolar zp = new ZenitPolar();
71      zp.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
72          zp.setVisible(true);
73    }
74 }
```

O que você deve ver

Figura 29.1: Resultado da execução do programa ZenitPolar.java

Para esse exemplo utilizamos apenas uma classe. Mas não se anime! Há muita informação nova para nós aqui!

Logo de cara vemos que a declaração da classe na linha 5 tem um detalhe diferente do que vinhamos usando para criar telas. Nossa classe agora, além de estender `JFrame`, implementa uma interface chamada `ActionListener`. Você já sabe: se uma classe implementa uma interface ela deve ter código para todos os métodos definidos nessa interface. No nosso caso, fomos forçados a escrever o método `void actionPerformed(ActionEvent e)` da linha 40. Vamos discutir isso daqui a alguns parágrafos, quando discutirmos botões. Por enquanto eu preciso apenas que você se lembre de que estamos trabalhando com uma janela diferente das anteriores.

Seguindo no código, o construtor, declarado na linha 9, tem suas duas primeiras linhas definindo o título da janela e suas dimensões.

Em seguida, as linhas 13 e 14 adicionam um `JLabel` da maneira que já conhecemos.

Na linha 16 instanciamos uma variável chamada `campoTexto` do tipo `JTextField`. A classe `JTextField` corresponde ao elemento visual campo texto. É com ela que podemos criar campos nos quais o usuário pode digitar textos. Quando trabalhavamos com a tela preta ~~horrível~~ do console, usávamos a classe `Scanner` para permitir ao usuário digitar algo, lembra-se? O `JTextField` é a classe correspondente ao `Scanner` no mundo visual, mas a maneira de codificar seu uso é ligeiramente diferente. Vejamos.

A primeira coisa que devemos saber é a quantidade prevista de caracteres que aceitaremos nesse campo. No exemplo da linha 16 esperamos um texto com 20 caracteres. Perceba que não devemos nos preocupar com as dimensões do campo (altura e largura) já que elas serão definidas de acordo com o layout utilizado na tela. O que precisamos definir é a "capacidade"desse campo texto. Essa informação do construtor ajuda o Java a desenhar o campo adequadamente.

Na linha 17 adicionamos esse campo texto recém criado na tela `ZenitPolar` que estamos montando.

Isso é o bastante para produzirmos, na tela, o efeito visual que queremos, ou seja, um campo capaz de receber textos de 20 posições. Contudo, como podemos, no código, acessar esse texto digitado pelo usuário? Você se lembra que na `Scanner` invocávamos um método chamado `next()` que já nos retornava o texto digitado. Aqui a banda toca diferente. Tudo que o usuário digita fica armazenado em um atributo interno da classe `JTextField`. Sempre que quisermos saber o que está escrito no campo texto devemos invocar o método `getText()`, como fizemos na linha 42.

Parabéns, você já domina mais um componente visual. Vamos cair pra dentro do próximo?!

Botões são componentes visuais que recebem cliques e executam tarefas. Podemos associar trechos de código a um botão de modo que esse trecho seja executado sempre que o usuário clicar nesse botão.

Para fazer isso em Java Swing usamos o código que começa na linha 19. Primeiramente instanciamos um objeto do tipo `JButton` e passamos em seu construtor o nome que queremos que decore esse botão. Na linha seguinte usamos o método `setActionCommand` para definir um "comando"associado a esse botão. Esse comando pode ser o texto que acharmos conveniente. Ele deve representar a ação que queremos que o botão execute. Nesse caso escolhemos a palavra "CONVERTE". Essa ação corresponde a um trecho de código e esse trecho de código deve estar escrito no método `actionPerformed`, que fomos obrigados a implementar por que nossa classe implementa a interface `ActionListener`.

Vamos, então, dar uma olhada nesse método `actionPerformed` da linha 40. Repare no comando da linha 41: `if(e.getActionCommand().equals("CONVERTER"))`. O que estamos fazendo aqui é verificando se o valor de `e.getActionCommand()` é igual à String "CONVERTER". Isso será `true` sempre que o botão `botaoCripto` for pressionado.

Vamos com calma aqui...

Sempre que um usuário clicar no botão `botaoCripto` o Java, internamente, vai chamar o método `actionPerformed` da linha 40. Para essa chamada é passado como parâmetro um objeto chamado `e` do tipo `ActionEvent`. Esse `ActionEvent` tem um `ActionCommand` que tem o mesmo valor configurado no `ActionCommand` do botão por meio do método `setActionCommand`. Então, quando fizemos `setActionCommand("CONVERTER")` na linha 20, definimos o valor do `ActionCommand` interno do botão `botaoCripto`. Quando o usuário clica nesse botão o Java executa o `actionPerformed` e o objeto `e` tem em seu `ActionCommand` exatamente o que esperamos, o valor "CONVERTER".

Então, na prática, o que o comando `if` da linha 41 está tentando fazer é verificar se o botão clicado foi o botão `botaoCripto`. Isso é necessário por que o método `actionPerformed` é compartilhado por todos os elementos visuais que respondem a eventos nessa tela. Repare que na linha 25 definimos o `ActionCommand` de outro botão (o `botaoLimp`, criado na linha 24) como "LIMPAR"e na linha 46 testamos se o botão clicado foi o limpar. Logo, o mesmo método (`actionPerformed`) é chamado pelo Java tanto quando acontecer um cliqueo em "Converter"quanto quando acontecer um clique em "Limpar".

Usando esse artifício conseguimos isolar os trechos de código de forma que as linhas 42 a 44 sejam executadas somente quando o usuário clicar em "Converter"e as linhas 47 e 48 apenas quando ele clicar em "Limpar".

E já que estamos aqui, vamos entender o que cada botão faz.

O "converter"recupera o texto digitado pelo usuário na linha 42 e em seguida usa um método chamado `converteZenitPolar` [1] para transformar o texto digitado em texto convertido. Na linha 43 esse texto convertido é usado para definir o texto de um `JLabel` chamado `textoCripto`, criado na linha 32 e adicionado na tela na linha 33.

O "limpar"define o texto do `JLabel` de resultado e do `JTextField` `campoTexto` como texto vazio.

Não sei se você reparou, mas as variáveis `textoCripto` e `campoTexto` foram declaradas nas linhas 6 e 7. Tivemos que declará-las fora de todos os métodos para que elas pudessem ser acessadas

[1] Se você não entende sozinho o código desse método, corra até www.javanamarra.com e compre o primeiro livro da série, Aprendendo Java na Marra.

por mais de um método da classe. Com isso fomos capazes de instanciá-las dentro do construtor e acessá-las dentro do método `actionPerformed`.

O último detalhe que você precisa saber sobre a criação de botões é o comando das linhas 21 e 26. O que estamos fazendo aqui é associando o botão criado a uma instância classe que implemente `ActionListener`. No caso, `this`, ou seja, a própria `ZenitPolar`.

Por fim, não podemos esquecer de adicionar os botões na tela (linhas 22 e 27), atribuir um layout à nossa tela (`GridLayout` de duas colunas na linha 35 e 36), e ajustar uns detalhezinhos (Um texto informativo na linha 29 e um texto vazio, que vai conter o texto criptografado, na linha 32).

Desafios para estudo

1. Adicione Mais dois botões na tela com o texto "Converter para maiúscula"e "Converter para minúscula".
2. Faça os ajustes necessários no método `actionPerformed` para que, quando clicados, os botões recém criados façam a ação correspondente.

Obs: Para converter o conteúdo de uma variável String para maiúsculas, use o método `toUpperCase()` e para converter para minúsculas use `toLowerCase()`. Inspire-se na chamada da linha 53.

Capítulo 30

Radio Button

As coisas estão começando a ficar interessantes não é? A medida que vamos conhecendo novos controles visuais vamos nos tornando capazes de fazer aplicações cada vez mais sofisticadas. Por isso é importante dominar o maior numero de componentes que puder. Nesse capítulo vamos continuar a explorar botões, mas dessa vez vamos estudar um tipo especial, o *Radio Button*.

O *Radio Button* é um componente muito popular e certamente você já o viu em muitas interfaces gráficas de sistemas por aí.

Se você curte ouvir uma boa musica, seja sozinho no carro ou junto com sua galera, legal! Mas isso não tem nada a ver com o Radio que vamos estudar nesse capítulo...

Digite o código, observe as telas resultantes da execução e vamos conversar...

O que você deve ver

Figura 30.1: Resultado da execução do programa Porta.java

No exemplo simulamos o jogo das portas no qual atrás de uma delas está um prêmio. O jogador escolhe apenas uma das portas e então uma outra porta, não premiada, é aberta. Nesse momento o jogador tem a oportunidade de trocar de porta ou continuar com aquela que escolheu no início. Quando a porta do jogador é aberta descobrimos se ele ganhou o prêmio ou não.

Perceba que se trata de uma situação na qual:

1. O usuário deve escolher apenas uma dentre as alternativas disponíveis.
2. A quantidade de alternativas é pequena.
3. As alternativas disponíveis são previamente conhecidas.
4. Podemos desejar impedir que uma ou mais alternativas sejam escolhidas (desabilitar).

Antes de optar pela utilização de um determinado controle visual é preciso entender quais os requisitos envolvidos. Os pontos apresentados acima caracterizam uma situação na qual o *Radio Button* pode ser utilizado.

O *Radio Button* é um componente visual, criado usando a classe `JRadioButton`, utilizado em conjunto com a classe `ButtonGroup`, que nos permite apresentar ao usuário um conjunto de opções dentre as quais ele somente pode escolher uma. Clicando na opção ele a seleciona. Clicando em outra ele seleciona a clicada e, por consequência, remove a seleção da anteriormente selecionada.

Vamos ver como fica o código para criar botões do tipo *Radio* e agrupá-los adequadamente.

A primeira coisa que notamos, na linha 5, é que a classe `Porta` estende `JFrame` e implementa `ActionListener`. Isso nos diz que ela é uma tela e que possui algum elemento que dispara uma ação (por exemplo, um botão).

Nas linhas 6 a 10 declaramos três `JRadioButton` e um `JButton`, que serão usados na tela, além de um `int` que armazenará o numero da porta premiada, a ser sorteado.

Nas linhas 16 e 17 criamos um `JLabel` para exibir uma mensagem ao nosso usuário, orientando que ele escolha uma porta.

Nas linhas 19 a 21 instanciamos os `JRadioButton`s, passando como parâmetro o texto que queremos associar a cada um deles. O botão da linha 19 recebe um segundo parâmetro indicando que ele deve ficar selecionado por padrão, ou seja, quando o programa inicia, ele já fica selecionado, como se o usuário já tivesse clicado nele.

Na linha 23 criamos um `ButtonGroup`. Lembre-se de que por definição um `JRadioButton` funciona em conjunto com outros `JRadioButton`s, formando um grupo dentre o qual apenas um deles poderá ser selecionado. Então, precisamos comunicar ao Java quais são os `JRadioButton`s que fazem parte de cada grupo. Fazemos isso usando a classe `ButtonGroup`.

Então, nas linhas 24 a 26 adicionamos cada um dos botões criados ao grupo. Isso basta para o Java saber que nosso usuário somente poderá selecionar uma das três portas.

Pronto! Nossos `Radio`s estão prontos. Usamos as linhas 28 a 30 para adicioná-los à tela.

Em seguida, nas linhas 32 a 35, criamos um `JButton` comum, da maneira que você já conhece. Repare que esse botão e os `Radio`s foram declarados fora do construtor da classe `Porta` por que queremos acessá-los lá no nosso método `actionPerformed`. Lembra-se de como ele funciona? Na linha 35 falamos para o Java que sempre que esse botão `btnAbrir` for clicado ele deve executar um método chamado `actionPerformed` que é implementado pelo objeto `this`, que no caso é a própria instância da classe `Porta`.

Na linha 37 definimos um layout e na linha 39 sorteamos um número entre 1 e 3 para indicar a porta premiada.

Assim, chega ao fim a execução do construtor e a tela é desenhada para o usuário.

Após o usuário escolher a sua porta e clicar no botão o método `actionPerformed` é disparado. Vamos analisar seu código que começa na linha 42.

Percebemos nas linhas 34 e 69 que esse método `actionPerformed` está preparado para lidar com dois tipos de ações, definidas pelos `ActionCommand`s "ABRIR_SEM_PREMIO"e "ABRIR_-ESCOLHIDA".

Como definimos no construtor (linha 33) que o botão `btnAbrir` tem o `ActionCommand` "ABRIR_SEM_PREMIO", a execução passa para a linha 44, onde se inicia uma lógica simples que vai até a linha 50 e visa determinar o número da porta escolhida pelo jogador. O método `isSelected()` da classe `JRadioButton` retorna `true` caso o botão esteja selecionado e `false` caso o botão não esteja selecionado.

Em seguida, nas linhas 52 a 55, tentamos sortear um número que não seja nem a porta premiada nem a porta escolhida pelo jogador. Esse número é armazenado na variável inteira `abrir` que em seguida é usada nas linhas 56 a 61 para desabilitar o `JRadioButton` correspondente à porta que queremos abrir.

Ao desabilitar um controle visual ele, geralmente, muda de aparência e não permite mais a interação com o usuário. Note que desabilitar um componente visual é muito diferente de torná-lo invisível usando o método `setVisible()`. Não faça confusão! Usando `setEnabled(false)` o componente continua na tela, mas o usuário não pode mais utilizá-lo.

Na linha 63 usamos o `JOptionPane` para mostrar uma mensagem para nosso jogador informando qual a porta não premiada foi aberta e comunicando que ele tem o direito de trocar de porta se desejar.

As linhas 65 e 66 têm um comando interessante. Perceba que a essa altura do programa precisamos jogar fora o botão que abre uma porta não premiada e criar um botão que abra a porta do usuário. Mas o que fizemos aqui foi reaproveitar o mesmo botão alterando apenas o seu texto de exibição e seu `ActionCommand`. Isso basta para que o usuário perceba que agora o botão tem outra utilidade e para que, quando clicado, de fato ele promova outra ação.

Assim, após o usuário decidir se quer ou não mudar de porta, o clique do botão nos leva

à execução do mesmo `actionPerformed`, mas agora apenas o segundo comando `if` da linha 69 será verdadeiro, pois o `ActionCommand` agora é "ABRIR_ESCOLHIDA", por força da linha 66 executada anteriormente.

Nas linhas 70 a 76 usamos a mesma lógica das linhas 44 a 50 para descobrir qual a porta que o usuário selecionou e nas linhas 78 a 81 exibimos a mensagem adequada de acordo com a comparação entre a porta escolhida e a porta premiada.

Por fim, nas linhas 83 a 85, desabilitamos todos os botões, pois o jogo já acabou.

Desafios para estudo

1. Altere o jogo para que estejam disponíveis cinco portas ao invés de três. Lembre-se de que além de adicionar novos botões você deve alterar a lógica de sorteio da porta premiada e da porta a abrir nas linhas 39 e 54.

Capítulo 31

Combo Box

Quando estudamos o `Radio` aprendemos que ele é um componente visual que se adéqua bem a situações em que o número de opções disponíveis para o usuário é limitado.

No exemplo desse capítulo vamos deixar o usuário escolher um estado brasileiro e o programa informará qual a capital desse estado.

Imagine uma tela com 26 `Radio`s, um para cada estado... Ficaria terrível não é? para essas situações, temos o *combo box*, que você vai aprender agora.

```
1  import java.awt.FlowLayout;
2  import java.awt.event.ActionEvent;
3  import java.awt.event.ActionListener;
4
5  import javax.swing.JComboBox;
6  import javax.swing.JFrame;
7  import javax.swing.JLabel;
8
9
10 public class Capitais extends JFrame implements ActionListener{
11
12   JLabel lblCapital;
13   String[] estados = { "Acre", "Alagoas", "Amapá", "Amazonas", "Bahia", "Ceará", "
       Distrito Federal", "Espítito Santo", "Goiás", "Maranhão", "Mato Grosso", "Mato
       Grosso do Sul", "Minas Gerais", "Pará", "Paraíba", "Paraná", "Pernambuco", "Piauí
       ", "Rio de Janeiro", "Rio Grande do Norte", "Rio Grande do Sul", "Rondônia", "
       Roraima", "Santa Catarina", "São Paulo", "Sergipe", "Tocantins" };
14   String[] capitais = { "Rio Branco", "Maceió", "Macapá", "Manaus", "Salvador", "
       Fortaleza", "Brasília", "Vitoria", "Goiânia", "São Luís", "Cuiabá", "Campo Grande
       ", "Belo Horizonte", "Belém", "João Pessoa", "Curitiba", "Recife", "Teresina", "
       Rio de Janeiro", "Natal", "Porto Alegre", "Porto Velho", "Boa Vista", "
       Florianópolis", "São Paulo", "Aracaju", "Palmas" };
15
16   public Capitais() {
17     setTitle("Capitais do Brasil");
18     setSize(300,150);
19
20     JLabel lblEscolha = new JLabel("Selecione um estado para descobrir sua capital:"
       );
21     getContentPane().add(lblEscolha);
22
23
24         JComboBox cbxEstados = new JComboBox(estados);
25         cbxEstados.addActionListener(this);
26         cbxEstados.setActionCommand("COMBO_ESTADOS");
27
28         getContentPane().add(cbxEstados);
29
30         lblCapital = new JLabel("");
31     getContentPane().add(lblCapital);
32
33
34     setLayout(new FlowLayout());
35
36   }
37
38
39
40   @Override
41   public void actionPerformed(ActionEvent e) {
```

```
42      if (e.getActionCommand().equals("COMBO_ESTADOS")) {
43        JComboBox cb = (JComboBox)e.getSource();
44        String estado = (String)cb.getSelectedItem();
45            int indice = cb.getSelectedIndex();
46            lblCapital.setText("A Capital do estado " + estado + " é: " + capitais[
      indice]);
47      }
48
49    }
50
51    public static void main(String[] args) {
52      Capitais testeCapitais = new Capitais();
53      testeCapitais.setVisible(true);
54    }
55
56  }
```

O que você deve ver

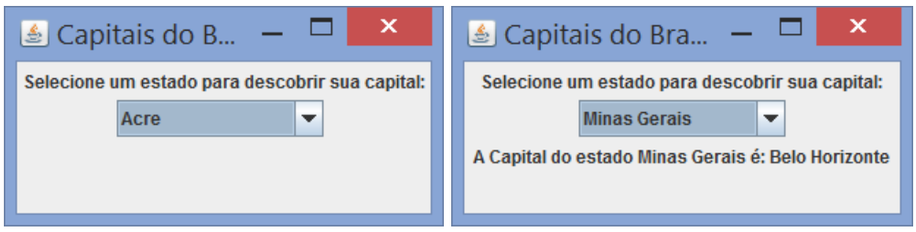

Figura 31.1: Resultado da execução do programa Capitais.java

Nosso programa começa no `main` na linha 34, como você sabe, e a única coisa que esse método faz é criar a tela `Capitais` e deixá-la visível.

O construtor da tela `Capitais` na linha 10 define o título da tela e seu tamanho nas linhas 11 e 12 e em seguida cria um `JLabel` na linha 14 para orientar o usuário. Na linha 15 esse *label* é adicionado à tela.

Na linha 17 criamos nosso `JComboBox`. Você vai perceber que se trata de um componente mais simples de utilizar que o `JRadioButton`, pois tudo que temos a fazer é instanciar um objeto do tipo `JComboBox` passando em seu construtor um vetor com as opções que desejamos oferecer ao nosso usuário. Pronto! Simples assim...

No nosso caso, o vetor que oferecemos ao `JComboBox` é o `estados`, criado na linha 7 e inicializado com a lista de estados brasileiros. Na linha 8 temos um vetor parecido, chamado `capitais`, inicializado com os nomes das capitais, na mesma ordem correspondente ao seu estado no vetor `estados`. Isso é importante, como veremos a seguir.

Na linha 18 adicionamos a própria instância de `Capitais` como `AdctionListener` para o objeto `cbxEstados`. Isso significa que esse objeto, do tipo `JComboBox`, é capaz de disparar eventos e, quando isso ocorrer, o objeto responsável por lidar com esse evento é o `this`, por meio de seu método `actionPerformed`. Até agora você só conhecia a utilização desse mecanismo de *ações e eventos* aplicado a botões, mas agora vê que ele também pode ser usado em `JComboBox`es.

E da mesma forma que fazíamos com botões, definimos o `ActionCommand` do nosso `cbxEstados` na linha 19.

Por fim, não podemos esquecer de adicionar esse `JComboBox` criado à tela, por meio do comando da linha 21.

As linhas 23 e 24 instanciam um `JLabel` sem texto e o adicionam na tela. Esse label será usado para mostrar a capital do estado escolhido.

Após a definição do layout na linha 26 nossa tela está pronta!

O método `actionPerformed`, na linha 29, responsável por executar as ações disparadas por componentes visuais nessa tela, será o responsável por exibir o texto informando a capital correspondente ao estado.

Então, na linha 32 usamos o método `getSelectedItem()` da classe `JComboBox` que nos retorna o valor selecionado pelo usuário. Fazemos isso para descobrir qual o *nome* do estado escolhido.

Na linha 33 usamos o método `getSelectedIndex()` da classe `JComboBox` que nos retorna o índice da escolha do usuário. Se ele escolheu o primeiro item do *combo*, o valor retornado por esse método será `0`. Se ele escolheu o segundo item do *combo*, o valor retornado por esse método será `1`. E assim sucessivamente.

Com o valor desse índice podemos descobrir a capital correspondente ao Estado escolhido apenas acessando a posição do vetor de capitais. Por isso você vê o comando `capitais[indice]` sendo concatenado na string passada como parâmetro para o método `setText()` do nosso *label* de resultado na linha 34.

Desafios para estudo

1. Crie outro `JComboBox` em sua tela para que o usuário faça o caminho inverso, ou seja, ele escolha uma capital e o sistema informe o Estado ao qual ela pertence.

Capítulo 32

Check Box

Você reparou que os últimos dois componentes visuais que estudamos têm uma característica em comum? Ambos dão ao usuário a oportunidade de escolher apenas uma opção dentre as disponíveis. Contudo, há situações em que queremos que o usuário escolha vários itens dentro do conjunto de possibilidades. Nessas situações o `JRadioButton` e o `JComboBox` não serão úteis.

No exemplo desse capítulo temos uma lista de matérias do segundo grau e o usuário deve selecionar aquelas com as quais tem mais afinidade. Baseado na seleção do usuário o programa sugerirá um ou mais cursos superiores adequados ao seu perfil.

Digite o código abaixo, execute, e descubra sua verdadeira vocação!

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  public class TesteVocacional extends JFrame implements ActionListener{
6    JCheckBox fisica;
7    JCheckBox matematica;
8    JCheckBox quimica;
9    JCheckBox portugues;
10   JCheckBox literatura;
11   JCheckBox biologia;
12   JCheckBox historia;
13   JCheckBox geografia;
14   JCheckBox filosofia;
15
16   public TesteVocacional() {
17     setTitle("Teste Vocacional");
18     setSize(450, 200);
19
20     fisica = new JCheckBox("Física");
21     getContentPane().add(fisica);
22
23     matematica = new JCheckBox("Matemática");
24     getContentPane().add(matematica);
25
26     quimica = new JCheckBox("Química");
27     getContentPane().add(quimica);
28
29     portugues = new JCheckBox("Português");
30     getContentPane().add(portugues);
31
32     literatura = new JCheckBox("Literatura");
33     getContentPane().add(literatura);
34
35     biologia = new JCheckBox("Biologia");
36     getContentPane().add(biologia);
37
38     historia = new JCheckBox("Historia");
39     getContentPane().add(historia);
40
41     geografia = new JCheckBox("Geografia");
42     getContentPane().add(geografia);
43
44     filosofia = new JCheckBox("Filosofia");
45     getContentPane().add(filosofia);
46
47     JButton btnVerificar = new JButton("Verificar vocação");
```

```
48      btnVerificar.addActionListener(this);
49      getContentPane().add(btnVerificar);
50
51      setLayout(new FlowLayout());
52    }
53    public void actionPerformed(ActionEvent e) {
54      String vocacoes = "";
55      if(fisica.isSelected() && matematica.isSelected() && quimica.isSelected())
56        vocacoes += "[Engenharia]";
57      if(fisica.isSelected() && matematica.isSelected())
58        vocacoes += "[Arquitetura "+ "[Computação]";
59      if(biologia.isSelected() && quimica.isSelected())
60        vocacoes += "[Biomedicina "+ "[Farmácia]";
61      if(geografia.isSelected() && historia.isSelected())
62        vocacoes += "[Turismo]";
63      if(geografia.isSelected() && historia.isSelected() && matematica.isSelected())
64        vocacoes += "[Contábeis]";
65      if(geografia.isSelected() && historia.isSelected() && matematica.isSelected() &&
       portugues.isSelected())
66        vocacoes += "[Administração]";
67      if(biologia.isSelected() && historia.isSelected())
68        vocacoes += "[Psicologia]";
69      if(portugues.isSelected() && literatura.isSelected())
70        vocacoes += "[Letras]";
71      if(biologia.isSelected() && quimica.isSelected() && portugues.isSelected())
72        vocacoes += "[Medicina]";
73      if(portugues.isSelected() && filosofia.isSelected() && historia.isSelected())
74        vocacoes += "[Direito]";
75      if(fisica.isSelected() && matematica.isSelected() && portugues.isSelected())
76        vocacoes += "[Estatística]";
77
78      if(vocacoes.equals(""))
79        JOptionPane.showMessageDialog(this, "Não foi possivel descobrir sua vocação.
      Escolha mais matérias");
80      else
81        JOptionPane.showMessageDialog(this, "Sua vocação pode ser: " + vocacoes + ".
      Escolha seu futuro com sabedoria.");
82    }
83    public static void main(String[] args) {
84      TesteVocacional testeVocacional = new TesteVocacional();
85      testeVocacional.setVisible(true);
86    }
87  }
```

O que você deve ver

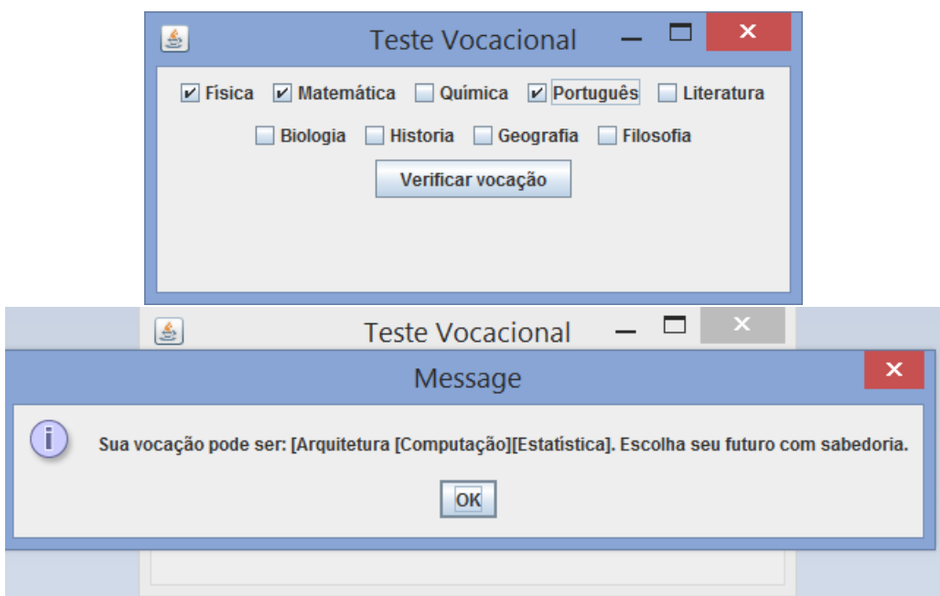

Figura 32.1: Resultado da execução do programa TesteVocacional.java

E aí? gostou do resultado? Vamos ver como nosso teste vocacional funciona...

Mais uma vez nossa classe, definida a partir da linha 5, estende `JFrame` e implementa

`ActionListener`. Você já sabe o que isso significa. (Trata-se de uma tela (`JFrame`) que possui objetos que disparam ações(`ActionListener`).)

Nas linhas 6 a 14 declaramos os objetos do tipo `JCheckBox`. Esse objeto corresponde a uma caixinha selecionável com nome associado, como você viu na tela. Nesse caso temos uma para cada matéria.

No construtor, começando na linha 20 e terminando na linha 45, temos uma sequência de código que, para cada `JCheckBox` declarado cria uma instância com o nome de exibição adequado e em seguida adiciona o `JCheckBox` à tela usando o comando `getContentPane().add()`, como já sabemos fazer.

Já que o usuário pode selecionar quantos e quaisquer itens que quiser, podendo ser inclusive selecionar todos ou nenhum, os `JCheckBox`es são independentes entre si por definição. Isso quer dizer que, como você nota, não há nenhum mecanismo de agrupamento entre eles. Isso contrasta com os componentes que vimos anteriormente. No `JComboBox`, a lista de itens é passada no construtor toda de uma vez. No `JRadioButton` criamos cada botão separadamente mas somos obrigados a agrupá-los usando `ButtonGroup`. No caso dos *checks*, cada `JCheckBox` é um elemento visual independente adicionado diretamente à tela, sem vínculo com os demais `JCheckBox`.

Nas linhas 47 a 49 criamos o botão para verificar a vocação do usuário, colocando nele um rótulo (linha 47), indicando o objeto que cudará de executar uma ação quando ele for clicado (linha 48) e adicionando esse botão à tela (linha 49).

Na linha 51 colocamos o layoutzinho de praxe...

Com a tela pronta, vamos agora ver o que acontece quando o botão é clicado. Isso é definido pela implementação do método `actionPerformed`, como você já sabe.

Repare que o código das linhas 55 a 76 é uma repetição de si mesmo. O que se faz nesse trecho é verificar se o usuário selecionou um determinado subconjunto de matérias que determinam a aptidão a uma carreira específica.

O método `isSelected()` da classe `JCheckBox` retorna `true` se o usuário marcou a caixa correspondente e `false` caso ele não tenha marcado.

Então, por exemplo, se o usuário marcou biologia e química na tela, quando ele aperta o botão a execução do `actionPerformed` passará por todos os comandos `if` até chegar à linha 59, quando `biologia.isSelected()` e `quimica.isSelected()` retornarão `true` e isso fará com que a linha 60 seja executada, montando a string `vocacao` com os textos `[biomedicina] [Farmácia]`.

Na linha 78 verificamos se a seleção feita pelo usuário se encaixou em alguma das hipóteses previstas. Se isso não aconteceu, o valor da variável `vocacoes`, declarada na linha 54, será ainda o valor inicial , e portanto, exibimos ao usuário uma mensagem orientando-o a selecionar mais disciplinas. Do contrário, mostramos ao usuário a sugestão de carreira identificada.

Desafios para estudo

1. Crie um botão "limpar seleção"que fará com que todos os checks fiquem sem seleção. Para fazer isso use o método `setSelected()` dos `JCheckBox`. Lembre-se que você terá de usar aquele artifício do `ActionCommand` para identificar, dentro do método `actionPerformed`, em qual botão o usuário clicou. Se não se lembra de nada disso, você está proibido de passar a página para frente! Volte ao capítulo de botões e revise a parte de `actionPerformed` e `ActionCommand`.

Capítulo 33

Menus

Você já não aguenta mais esse assunto de botões?! Pois bem, nem eu! Vamos escolher algo diferente no menu. Falando em menu, você já conhece o `JMenu`? É um componente visual bastante útil para organizar funcionalidades do seu sistema. A maioria dos programas com interface visual fazem uso desse tipo de componente. Após executar o programa abaixo você vai perceber que já viu ele antes. O código para a criação de menus é um pouco mais complexo do que os componentes que estudamos até agora. Em compensação, você vai notar que ele é bem mais poderoso em termos de organização do conteúdo.

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  public class TextoFormatado  extends JFrame implements ActionListener{
6    JLabel lblNome;
7    public TextoFormatado() {
8      setTitle("Formatação de textos");
9      setSize(300, 200);
10
11     JMenuBar barraMenu = new JMenuBar();
12     JMenu aparencia = new JMenu("Aparência");
13     barraMenu.add(aparencia);
14
15     JMenu estilo = new JMenu("Estilo");
16
17     JMenuItem italico = new JMenuItem("Itálico");
18     italico.setActionCommand("ITALICO");
19     italico.addActionListener(this);
20     estilo.add(italico);
21
22     JMenuItem negrito = new JMenuItem("Negrito");
23     negrito.setActionCommand("NEGRITO");
24     negrito.addActionListener(this);
25     estilo.add(negrito);
26
27     aparencia.add(estilo);
28
29     JMenu cor = new JMenu("Cor");
30
31     JMenuItem vermelho = new JMenuItem("Vermelho");
32     vermelho.setActionCommand("VERMELHO");
33     vermelho.addActionListener(this);
34     cor.add(vermelho);
35
36     JMenuItem azul = new JMenuItem("Azul");
37     azul.setActionCommand("AZUL");
38     azul.addActionListener(this);
39     cor.add(azul);
40
41     aparencia.add(cor);
42
43
44     JMenu fonte = new JMenu("Fonte");
45     barraMenu.add(fonte);
46
47     JMenuItem times = new JMenuItem("Times new Roman");
48     times.setActionCommand("TIMES");
49     times.addActionListener(this);
50     fonte.add(times);
```

```
51
52      JMenuItem courrier = new JMenuItem("Courrier new");
53      courrier.setActionCommand("COURRIER");
54      courrier.addActionListener(this);
55      fonte.add(courrier);
56
57      setJMenuBar(barraMenu);
58
59      String nome = JOptionPane.showInputDialog(this, "Digite seu nome.");
60      lblNome = new JLabel(nome);
61      lblNome.setFont(new Font("Courier New", 0, 40));
62      getContentPane().add(lblNome);
63
64      setLayout(new FlowLayout());
65    }
66    public void actionPerformed(ActionEvent e) {
67      if (e.getActionCommand().equals("ITALICO"))
68        lblNome.setFont(new Font("Courier New", Font.ITALIC, 40));
69
70      if (e.getActionCommand().equals("NEGRITO"))
71        lblNome.setFont(new Font("Courier New", Font.BOLD, 40));
72
73      if (e.getActionCommand().equals("VERMELHO"))
74        lblNome.setForeground(Color.RED);
75
76      if (e.getActionCommand().equals("AZUL"))
77        lblNome.setForeground(Color.BLUE);
78
79      if (e.getActionCommand().equals("TIMES"))
80        lblNome.setFont(new Font("Times New Roman", 0, 40));
81
82      if (e.getActionCommand().equals("COURRIER"))
83        lblNome.setFont(new Font("Courier New", 0, 40));
84
85      if(e.getActionCommand().equals("MINUSCULAS"))
86        lblNome.setText(lblNome.getText().toLowerCase());
87
88      if(e.getActionCommand().equals("MAIUSCULAS"))
89        lblNome.setText(lblNome.getText().toUpperCase());
90    }
91    public static void main(String[] args) {
92      TextoFormatado teste = new TextoFormatado();
93      teste.setVisible(true);
94    }
95  }
```

O que você deve ver

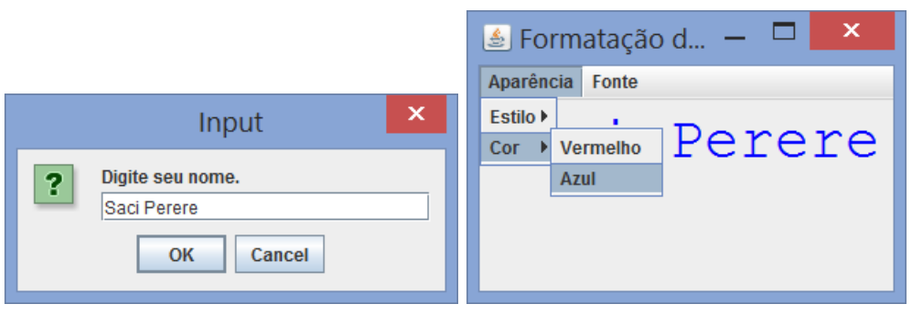

Figura 33.1: Resultado da execução do programa TextoFormatado.java

A primeira coisa que precisamos entender sobre esse tipo de menu é que ele é um componente visual com algumas particularidades. A primeira é que ele tem um lugar pré-determinado para aparecer: o topo da tela. [1]

A segunda coisa é que temos que entender o menu como uma árvore. Uma árvore que cresce de cabeça para baixo. A árvore tem raíz, galhos, os galhos podem ter outros galhos e por fim temos as folhas.

[1] Alguns layouts mais complexos podem colocar os menus em local diferente na tela. Mas isso é pouco usado e foge do escopo desse livro.

No mundo dos menus temos a barra de menu (raíz), os menus e submenus (galhos) e os itens de menu (folhas). Assim como nao existe árvore sem esses elementos, também não existe menu sem essas três partes.

A barra de menu apresenta uma sére de opções, que são os menus. Cada menu pode ter itens e submenus. Ao clicar em um item uma ação eventualmente associada será disparada. Ao clicar em um submenu, é exibido para o usuário os itens pertencentes a esse submenu. A hierarquia de submenus pode se dar com quantos níveis desejarmos, mas sempre culmina em pelo menos um item de menu.

Em java, para implementarmos o que foi discutido acima, usamos as classes `JMenuBar`, `JMenu` e `JMenuItem`.

Então, na linha 11 criamos uma barra de menu instanciando um objeto do tipo `JMenuBar`. A esse objeto podemos adicionar menus. Esse objeto é a raiz da nossa árvore. Ele corresponde à barra onde estão os menus "Aparência"e "Fonte".

Na linha 12 tratamos de criar o menu "Aparência"usando a classe `JMenu` e passando em seu construtor o texto que queremos usar como rótulo do menu.

Na linha 13 adicionamos esse `JMenu` recém criado à nossa barra. Essa adição, repare, pode ser feita a qualquer momento. Nesse caso, fizemos antes de terminar de montar o menu `aparencia`. Poderíamos deixar para executar o comando da linha 13 depois de configurar o menu `aparencia` sem problema nenhum. O que importa é que, em algum momento, essa adição seja feita.

Na linha 15 criamos outro objeto, chamado `estilo`, também do tipo `JMenu`. Por enquanto ele está solto. Não está associado a nenhum menu ou barra de menu e nem possui submenus.

Na linha 17 criamos um item de menu, instanciando o objeto `italico` do tipo `JMenuItem`. Essa é a folha da nossa árvore. É o item de menu que, quando clicado, vai disparar uma ação.

Você já sabe como trabalhar com ações em Swing. No objeto que dispara a ação, associamos um comando e falamos qual o objeto que implementa a interface `ActionListener` e possui o método `actionPerformed` onde estará o código da ação.

Então, não tem segredo, a linha 18 define o `ActionCommand` (no caso, "ITALICO"), e o `ActionListener` (o próprio objeto do tipo TextoFormatado, `this`).

Não se espante. Esse artifício envolvendo `ActionListener` pode ser usado com quase todo componente visual que permita interação com o usuário(Campo texto, botões, menus, etc...) e sempre da mesma forma: Define um comando, adiciona o *listener* e implementa ação no método `actionPerformed`.

Na linha 20 adicionamos o item `italico` ao menu `estilo` criado na linha 15 que, por enquanto, ainda solto na tela(ainda não adicionamos esse menu `estilo` à tela nem a um outro menu).

Da linha 22 à 25 criamos outro `JMenuItem` de forma muito parecida com o que fizemos com o `italico`. Dessa vez criamos o `negrito`. O trecho de código é praticamente idêntico ao `italico`. Alteramos apenas o texto de exibição (linha 22) e o `ActionCommand` na linha 23.

Em seguida, finalmente, adicionamos o menu `estilo` a outo menu, o `aparencia`. Olha só o que esta rolando aqui: Na linha 12 criamos um `JMenu`, na linha 15 criamos outro `JMenu` e na linha 27 adicionamos um no outro. É assim que criamos submenus em Java Swing. Bem intuitivo, não é?

Na linha 29 criamos mais um `JMenu` cujo texto é "Cor". A esse menu adicionamos dois `JMenuItem`s que criamos nas linhas 31 a 39 da mesma forma que fizemos com o `italico` e o `negrito`.

Na linha 41 adicionamos esse menu (`cor`) ao mesmo menu `aparencia` ao qual já havíamos adicionado o `estilo`. Então, nao perca a conta, o menu `aparencia` já contém dois outros menus, o `estilo` e agora o `cor`. E cada um desses tem dois `JMenuItem`.

Não se esqueça de que o menu `aparencia`, por sua vez, já foi adicionado à barra de menu na linha 13.

Agora queremos criar o outro menu, que reúne os comandos que batizamos de "Fonte". Então criamos um objeto do tipo `JMenu` na linha 44 e adicionamos esse menu à barra de menus na

linha 45.

Como esse menu não terá submenus, partimos direto para a criação dos `JMenuItem`. Sempre de forma muito parecida com os criados anteriormente: Instanciamos um objeto `JMenuItem` passando o texto no construtor, definimos o `ActionCommand`, o `ActionListener` e adicionamos o item ao menu `fonte`. Fazemos isso da linha 47 a 50 para criar o item `times` e da linha 52 a 55 para criar o item `courrier`.

A linha 57 é fundamental! Não sei se você reparou, mas criamos a barra, um punhado de menus, penduramos os menus uns nos outros e depois na barra, e só. Em momento algum associamos a barra à tela. O método `setJMenuBar()` da classe `JFrame` define a barra de menu da janela. Por isso chamamos esse método passando nossa barra como parâmetro.

Calma! Tá acabando a tela...

Na linha 59 criamos a caixa de diálogo para perguntar o nome do usuário. Na linha 60 criamos o `JLabel` que exibirá o nome do usuário e adicionamos na tela na linha 62, mas não sem antes dar uma formatada no texto. Usando o método `setFont` do `JLabel` somos capazes de mudar a aparência dos textos que exibimos. Nesse caso estamos definindo a fonte "Courrier New"e tamanho 40.

O código do método `actionPerformed` é de uma simplicidade franciscana. Tenho certeza que você consegue entendê-lo sozinho, mas como o nosso esquema é "na marra", vou me esforçar escrevendo mais parágrafos e você se esforce lendo-os!

Cada comando `if` está testando um dos possíveis `ActionCommand`s definidos nos diversos `JMenuItem` criados no construtor. O comando associado a cada `if` executa a formatação de texto correspondente.

Para colocar o texto em itálico ou negrito, usamos o `setFont` passando uma nova font com o parâmetro `Font.ITALIC` ou `Font.BOLD` Ilinhas 68 e 71. Também usamos o método `setFont` para definir a fonte propriamente dita, ou seja, "Times new roman"na linha 80 e "Courrier New"na linha 83.

Quando queremos alterar a cor de um `JLabel` usamos o método `setForeground()`, que recebe como parâmetro o nome da cor desejada. No exemplo usamos `Color.RED` na linha 74 e `Color.BLUE` na linha 77. Você não terá dificuldade em experimentar outras cores básicas se souber seu nome em inglês.

Desafios para estudo

1. Adicione mais dois itens de menu ao menu `Fonte`, para transformar o texto em maiúsculas e minúsculas. Para alterar o texto do label você terá de utilizar métodos que toda `String` tem chamados `toLowerCase()` e `toUpperCase()`. O primeiro método joga todo conteúdo da `String` para caixa baixa e o segundo, o contrário. .

Capítulo 34

HSQLDB e óleo de peixe: pois faz bem lembrar as coisas

Está cansado de ficar imaginando uma forma mirabolante para guardar alguma informação em arquivos? Está tendo dores de cabeça para entender como um arquivo está organizado? Já não aguenta mais ter que tratar registros ruins no arquivo? Seus problemas acabaram! Chegou a hora do Banco de Dados!

Bancos de dados são como criaturas mágicas criadas para organizar a sua vida. Ele pode armazenar de forma organizada todos os dados que você quiser.

Neste primeiro capítulo vamos instalar e configurar um banco de dados, então desta vez não vamos digitar código Java, mas você precisará de uma conexão com a Internet funcionando e vai executar o seguinte procedimento:

1. Abrir um navegador
2. Digitar na barra de endereços: "http://hsqldb.org/"
3. Procurar e clicar o link "download"
4. Procurar "Looking for the latest version?"
5. Baixar o "hsqldb-?.?.?.zip". Quando estive por lá ele se chamava "Download hsqldb-2.3.3.zip (7.8 MB)"
6. Salvar o arquivo onde for conveniente e extrair o conteúdo dele, lembre desse lugar!
7. Você precisará das pastas "lib", "bin" e "data". Faça uma cópia delas dentro da sua pasta de programas fonte.

Agora que você conseguiu uma cópia do gerenciador de banco de dados simples e gratuito, sugiro que você faça uma doação aos autores do programa. Digitar (e criar!) quantidade tão grande de código deve ter dado um bocado de trabalho! Se você for curioso como eu, basta olhar os diversos programas .java na pasta "src/org/hsqldb/" e suas subpastas.

Vamos agora rodar o HSQLDB. Se você seguiu todas as instruções desde o livro 1 , já está familiarizado com o ambiente de construir código, no prompt onde digitamos `javac <programa >`. Pode ser que, se estiver utilizando o sistema operacional Windows, uma janela como a da figura 34.1 lhe pergunte que tipo de acesso um usuário da rede deve ter. Eu deixo apenas "Redes privadas" e clico "Permitir acesso".

Guarde os comandos Java a seguir, pois vamos precisar deles nos próximos capítulos.

Este primeiro comando coloca o servidor de banco de dados para executar.

No Linux:

```
ls -d ./*
cd data
java -classpath ../lib/hsqldb.jar org.hsqldb.server.Server
```

[0]Aprendendo Java na marra

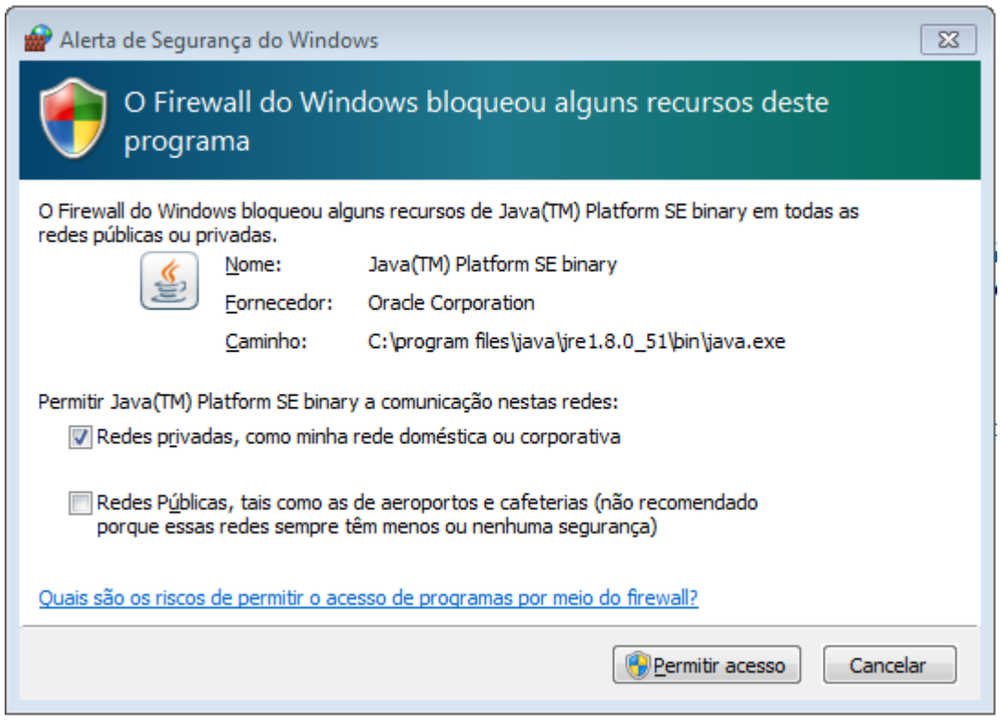

Figura 34.1: Janela do firewall do windows

No Windows:

```
dir /ad
cd data
java -classpath ../lib/hsqldb.jar org.hsqldb.server.Server
```

IMPORTANTE! Se você não fizer a parte do "cd data" sua pasta de programas vai ganhar uns arquivos extras onde seus dados serão gravados. Fique atento! Queremos manter a casa organizada. ;).

Para abrir o "HSQL Database Manager", abra uma segunda janela de comandos, vá para sua pasta de programas e execute:

```
java -cp lib/hsqldb.jar org.hsqldb.util.DatabaseManager
```

Na opção "Type" escolha "HSQL Database Engine Server" e clique "Ok".

Este comando colocou o cliente do banco de dados para executar.

O que você deve ver

```
C:\Users\Roberto\Programas>ls -d */.
bin/.  data/.  lib/.
C:\Users\Roberto\Programas>cd data
C:\Users\Roberto\Programas\data>java -classpath ../lib/hsqldb.jar org.hsqldb.server.
    Server
[Server@24d46ca6]: [Thread[main,5,main]]: checkRunning(false) entered
[Server@24d46ca6]: [Thread[main,5,main]]: checkRunning(false) exited
[Server@24d46ca6]: Startup sequence initiated from main() method
[Server@24d46ca6]: Could not load properties from file
[Server@24d46ca6]: Using cli/default properties only
[Server@24d46ca6]: Initiating startup sequence...
[Server@24d46ca6]: Server socket opened successfully in 23 ms.
[Server@24d46ca6]: Database [index=0, id=0, db=file:test, alias=] opened sucessfully
    in 974 ms.
[Server@24d46ca6]: Startup sequence completed in 1006 ms.
[Server@24d46ca6]: 2015-09-28 15:12:33.714 HSQLDB server 2.3.3 is online on port
    9001
[Server@24d46ca6]: To close normally, connect and execute SHUTDOWN SQL
[Server@24d46ca6]: From command line, use [Ctrl]+[C] to abort abruptly
```

Se você chegou até aqui, parabéns! Você instalou seu primeiro sistema gerenciador de bancos de dados!

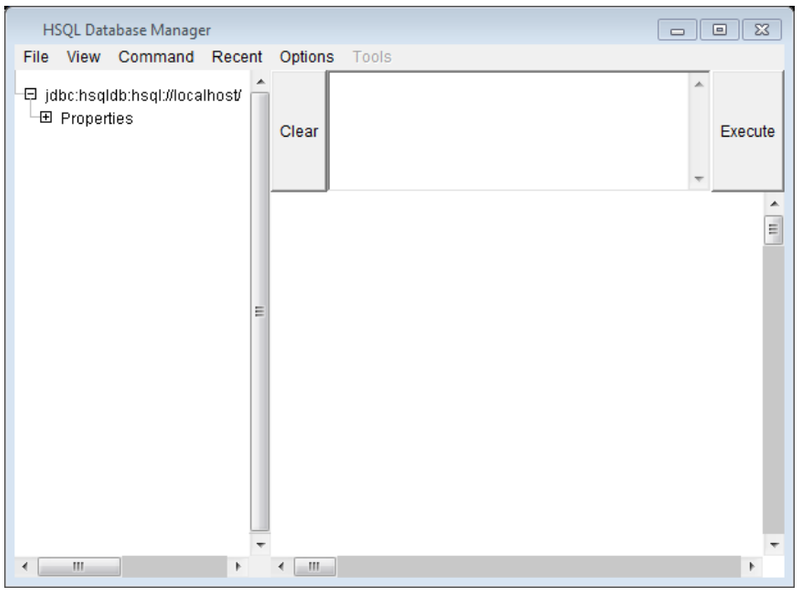

Figura 34.2: Tela de gerência do HSQLDB

Como funciona isso? Bem, se você é um entusiasta da informática, pode gostar da explicação, de outra forma, leia e coloque na sua gaveta de cultura geral. É mais ou menos assim: os computadores modernos são capazes de executar mais de um programa ao mesmo tempo sem que você precise ficar de olho neles. Assim, os programadores conseguem isolar e especializar o que cada programa faz. Esses programas vão se comunicar de alguma forma ou vai ser por troca de mensagens ou através de uma memória compartilhada. No nosso caso, essa comunicação é através de troca de mensagens pela rede, e todo computador moderno assume que você faz parte de alguma rede, nem que seja de um computador apenas falando sozinho. Coisa de louco, não? Os computadores falam consigo mesmos e não tem nenhum problema com isso.[1] Ah, o óleo de peixe esqueci e vou ficar devendo. Nem todo mundo gosta...

Desafios para estudo

1. Você pode tentar instalar outro banco de dados comercial, como o SQL Server http://www.microsoft.com/pt-br/server-cloud/products/sql-server-editions/sql-server-express.aspx

2. Se ainda não estiver satisfeito, pode tentar instalar também este outro banco de dados famoso: http://www.oracle.com/technetwork/database/database-technologies/express-edition/downloads/index.html

Eu realmente não espero que você tente baixar esses bancos. São instalações muito grandes e com muitos recursos que nós não utilizaremos por enquanto. Mas eu tinha que dizer para vocês que essas alternativas existem.

[1] Sua cabeça deve estar girando agora com programas falando sozinhos. Talvez eu seja apenas mais um deles, ha ha ha!

Capítulo 35

Mapeando uma classe em uma tabela

Gostou do último capítulo? Bem, acho melhor você se acostumar, nós ainda vamos precisar dele mais uma vez. Pois você precisará colocar o servidor de banco de dados para executar antes que seu programa consiga falar com ele. Lembre-se de manter a janela do servidor aberta durante todo o tempo em que seu programa precisar dele.

Alguns gerenciadores de banco de dados são menos exibidos e é mais difícil saber se eles estão em execução. E agora, vamos preparar nosso banco de dados para receber nossos preciosos dados.

Você precisa ter concluído o capítulo anterior para se dar bem neste. Se você ainda não fez isso, recomendo que volte e guarde o conhecimento obtido lá com muita atenção pois você pode, e vai, precisar!

Apenas para te ajudar, você deve colocar o banco de dados para executar em uma janela separada com um comando parecido com

```
java -cp lib/hsqldb.jar org.hsqldb.util.DatabaseManager
```

e depois colocar o administrador do banco de dados, o "HSQL Database Manager", em uma segunda janela de comandos:

```
java -classpath ../lib/hsqldb.jar org.hsqldb.server.Server
```

Na opção "Type" escolha "HSQL Database Engine Server" e clique "Ok".

Digite o comando `CREATE TABLE` conforme a figura e clique o botão "Execute" e depois CTRL+R (ou escolha "View" e "Refresh tree"):

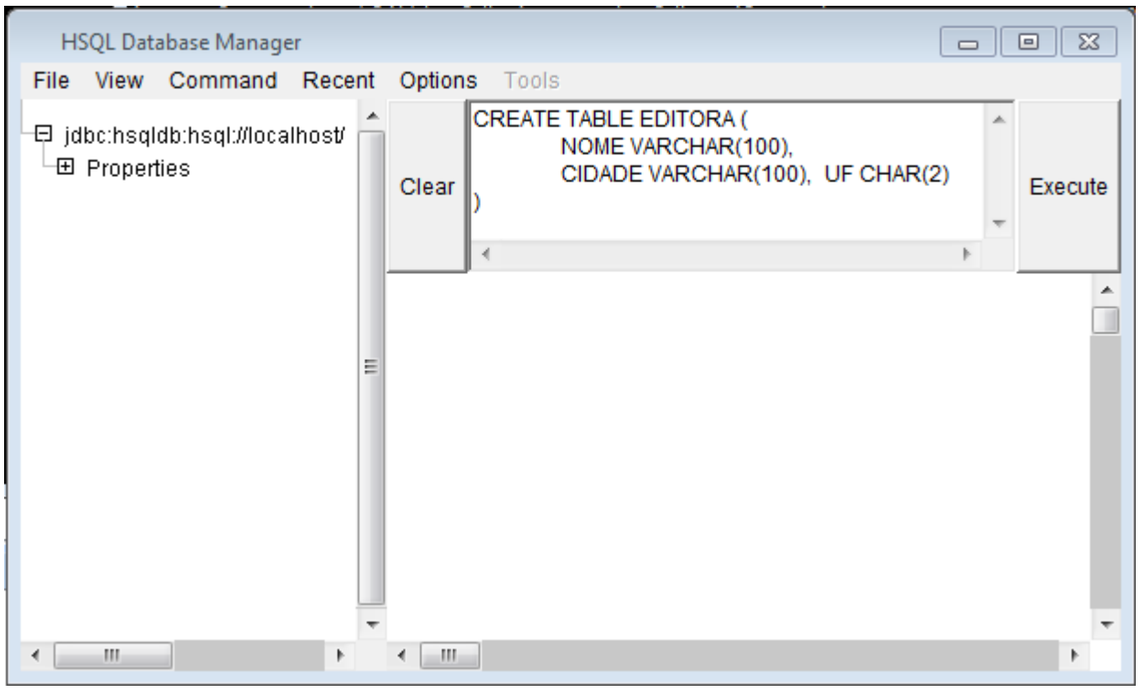

Figura 35.1: Criação de uma tabela

O que você deve ver

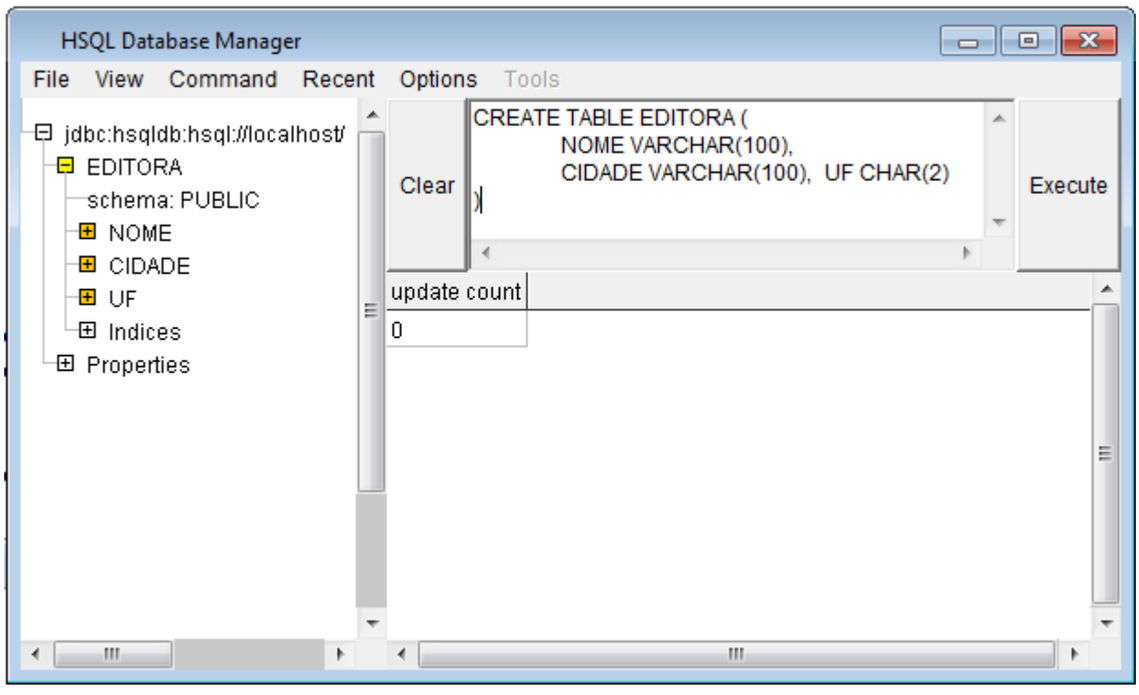

Figura 35.2: Uma tabela criada

Bem, criamos uma tabela. Mas... O que é mesmo uma tabela? Bem, uma tabela é uma estrutura de dados que consegue guardar registros, como aqueles do livro 1[1] que na verdade são classes apenas com tipos primitivos sem métodos. As tabelas são objetos dos bancos de dados onde a informação é armazenada em um formato lógico de uma coluna para cada campo e uma linha para um registro, como um vetor. Lembra do capítulo sobre vetores? Pois bem, as tabelas nos bancos de dados são como os vetores, a diferença é que o valor não vai sumir quando programa terminar. A isso damos o nome de *persistência*.

Para persistir uma classe precisamos fazer um mapeamento de seus atributos às colunas de uma tabela que deve estar criada no banco de dados para receber objetos dessa classe.

Dessa forma, vamos lembrar da classe Editora:

[1] Aprendendo Java na marra

```
1  public class Editora
2  {
3    private String nome;
4    private String cidade;
5    private String uf;
6
7    public String toString()
8    {
9      return nome + ":"+ cidade + ", " + uf;
10   }
11
12   public void setNome(String nome)
13   {
14     this.nome = nome;
15   }
16
17   public String getNome()
18   {
19     return this.nome;
20   }
21
22   public void setCidade(String cidade)
23   {
24     this.cidade = cidade;
25   }
26
27   public String getCidade()
28   {
29     return cidade;
30   }
31
32   public void setUf( String uf)
33   {
34     this.uf = uf;
35   }
36
37   public String getUf()
38   {
39     return uf;
40   }
41 }
```

Ela possui três atributos: `String nome`, `String cidade` e `String uf`.

Para armazenar os valores de uma classe em um banco de dados relacional com suporte à linguagem SQL, precisamos respeitar algumas conversões de tipo que vão depender do tipo de banco de dados que estamos utilizando. Você também precisa saber que SQL é um padrão americano ANSI e também é o nome de um produto. Quando falamos do padrão, temos os SQL-86, SQL-89, SQL-92, SQL:1999, SQL:2003 e o SQL:2008. O HSQLDB, por exemplo, suporta todos os tipos definidos no SQL-92 e implementa os tipos numéricos TINYINT, SMALLINT, INTEGER e BIGINT, o tipo booleano BOOLEAN, os tipos que armazenam caracteres CHARL, VARCHARL e CLOB, os tipos para Strings binárias BINARYL, VARBINARYL e BLOB e os tipos DATE, TIME and TIMESTAMP para guardar datas e intervalos de tempos entre outros[2].

Como a classe lida essencialmente com `String`s vamos definir as colunas como sendo do tipo `VARCHAR` e acreditar na nossa intuição de que não vamos precisar guardar uma editora com um nome composto por mais de 100 caracteres, ou uma cidade com um nome maior que 100 caracteres, ou ainda uma sigla de unidade da federação maior que dois caracteres.

A ideia não é dar um curso de linguagem SQL, mas se você conhece um pouco de SQL ou tem um mínimo de noção do inglês, vai ver que digitamos o comando `CREATE TABLE EDITORA ( NOME VARCHAR(100), CIDADE VARCHAR(100), UF VARCHAR(2))` que instrui ao servidor SQL a criar uma tabela com três colunas dos tipos que seguem seus nomes e daquele tamanho. Então usamos `CREATE TABLE`, abrimos um parêntesis, colocamos o nome da primeira coluna, o tipo da primeira coluna e o tamanho dessa coluna entre parêntesis, uma vírgula, e seguimos para a segunda e terceira coluna da tabela encerrando o comando com um parêntesis e clicamos em `Execute`.

[2]Você sempre pode consultar a documentação na internet: `http://hsqldb.org/doc/guide/sqlgeneral-chapt.html#sgc_types_ops`

Desafios para estudo

1. Crie uma tabela chamada Livro conforme a classe criada no início deste livro.

Capítulo 36

Redes sociais de programas: fazendo conexão ao banco de dados

E então? Após tantos capítulos sem programar em Java agora voltamos à ativa. Inicialmente vamos fazer um programa que apenas se conecta no banco de dados de nossa preferência.

Digite, então, sem perda de tempo, o seguinte programa:

```
1  import java.sql.DriverManager;
2  import java.sql.Connection;
3  import java.sql.SQLException;
4
5  public class SGBD {
6    public static Connection getConnection() throws Exception, SQLException {
7      Class.forName("org.hsqldb.jdbc.JDBCDriver" );
8      Connection c = DriverManager.getConnection("jdbc:hsqldb:hsql://localhost/", "SA"
       , "");
9      return c;
10   }
11
12   public static void main(String[] args) {
13     Connection conn;
14     try {
15       conn = getConnection();
16       System.out.println("Conexão realizada com sucesso!");
17       conn.close();
18       System.out.println("Conexão encerrada com sucesso!");
19     } catch (SQLException e) {
20       System.err.println("ERRO:SQLException: não foi possível conectar ao banco de
       dados.");
21       e.printStackTrace();
22       return;
23     } catch (Exception e) {
24       System.err.println("ERRO:Exception: não foi possível carregar o driver JDBC do
       HSQLDB.");
25       e.printStackTrace();
26       return;
27     }
28   }
29 }
```

Na linha 1 começamos algumas importações necessárias do pacote java.sql. Essas classes são `DriverManager`, `Connection` e `SQLException`. A primeira, `DriverManager`, é responsável por localizar o *driver* do banco de dados. Um *driver* é um programa que sabe como o banco de dados se comporta e que também sabe o que um programa Java qualquer pode esperar do banco de dados[1], assim, se precisássemos trocar o banco de dados um dia, bastaria substituir o *driver* por um outro adequando ao novo banco de dados e rodar nosso programa sem maiores modificações[2] e o *driver* se encarregaria de levar nossas demandas ao novo banco de dados da maneira correta.

A classe `Connection` representa uma conexão, ou como nós cavaleiros Jedis do banco de

[1] Essas coisas como *o comportamento do banco de dados* e *o que um programa Java qualquer pode esperar do banco de dados* é só um jeito dramático de dizer que cada programa tem uma *Application Program Interface*, carinhosamente chamada de *API*, ou seja, um padrão de programa de aplicação. Uma vez que sabemos qual é o padrão esperado e o padrão oferecidos sempre poderemos fazer um *driver* capaz de colocar os dois padrões para conversar.

[2] Essa é mais uma daquelas meias verdades. Com o tempo você vai perceber que podemos realizar algumas tarefas especiais a partir de comandos que só um fabricante de banco de dados proporciona, e, nesses casos, trocar o banco de dados se torna uma tarefa bem mais complicada.

dados costumamos dizer, uma sessão com um banco de dados específico. Os comandos SQL e seus resultados são executados e retornados através de uma conexão[3]. Um objeto do tipo `Connection` é bastante poderoso e pode nos trazer informações diversas sobre o banco de dados e seus objetos através do método `getMetaData`. Sem esse método, não poderíamos utilizar ferramentas gráficas bonitinhas e cheias de recursos escritas em Java, como o *SQuirreL SQL Client*[4].

A classe `SQLException` vai nos trazer informações sobre erros de acesso ao banco de dados entre outros. Ela traz coisas como uma descrição e um "SQLstate" entre outras informações. Esse "SQLstate" te ajudará a fazer diagnósticos em relação aos problemas que encontrar enquanto estiver rodando programas que fazem uso de bancos de dados. Consulte o javadoc para mais informações sobre as classes do pacote java.sql[5].

Seguindo, vamos criar um método `getConnection` na nossa classe `SGBD` para que você possa ver as coisas bem divididas, esse método está definido nas linhas 6 a 10.

Na linha 7, temos uma coisa bem incomum: `Class.forName`. A classe `Class` representa todas as classes e interfaces no ambiente Java em execução no momento em que é acionada. Assim, vamos usar o serviço `forName` para carregar o *driver* na memória. Assim, se tivermos dezenas de *drivers* entre as nossas classes o Java vai saber qual utilizar quando executarmos o comando da linha 8. Se estivermos usando um *driver JDBC 4.0* o Java já saberá o que fazer e esta linha se tornará desnecessária. Como o Java vai saber que *driver* carregar você poderá conferir alguns parágrafos adiante.

Na linha 8 fazemos uma chamada ao `DriverManager` instruindo-o a buscar uma conexão através do método `getConnection`. O parâmetro que passamos ao `getConnection` é `"jdbc:hsqldb:hsql://localho` `"SA", ""`. Cada banco de dados terá sua própria string de conexão, então se você não estiver usando o HSQLDB precisará pesquisar na web qual a string para o seu banco, mas não se preocupe, isso é bem fácil de achar. No final, o comando `Driver.getConnection` retornará um objeto `Connection` que nós vamos usar para acessar o nosso banco de dados de fato.

Essa última informação eu preciso que você guarde como se sua vida dependesse disso. Observe a linha 17, onde diz `conn.close()`. O que ela tem de especial? É importantíssimo encerrar a conexão com o banco de dados sempre que não estiver em uso. Quando não fazemos isso deixamos recursos como memória e processos do sistema operacional esperando por comandos para o banco de dados que simplesmente não vão chegar e adivinha o que acontece quando a memória do servidor de banco de dados acaba? O sistema fica lento, os humanos começam a reclamar e o telefone do administrador do banco de dados toca. O seu programa para de funcionar mesmo você não tendo feito nada. E principalmente porque você não fez o que deveria.

As demais linhas você já tem condições de saber o que significam.

O que você deve ver

```
$ java SGBD
ERRO: Exception: não foi possível carregar o driver JDBC do HSQLDB.
java.lang.ClassNotFoundException: org.hsqldb.jdbc.JDBCDriver
        at java.net.URLClassLoader.findClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at java.lang.Class.forName0(Native Method)
        at java.lang.Class.forName(Unknown Source)
        at SGBD.getConnection(SGBD.java:7)
        at SGBD.main(SGBD.java:15)
```

Pensou que tinha terminado, não é? Só que ainda não. Agora estamos utilizando mais de uma classe e elas não estão na mesma pasta. E agora, José?

Existem várias formas de agrupar arquivos executáveis do Java. No momento, você precisa apenas saber o que é o *classpath*. O *classpath* é uma lista de pastas e arquivos, separados por ":", onde o Java vai procurar por classes para executar. Algumas dessas pastas podem estar em arquivos

[3]Se quiser chamar de sessão, pense como um cinema: ao criar uma conexão exibimos o título do filme, desenvolvemos uma história levando e trazendo informações e terminamos no clímax, um resultado importante para o produtor. Câmera, luzes, ação!

[4]http://sourceforge.net/projects/squirrel-sql/

[5]http://docs.oracle.com/javase/7/docs/api/java/sql/package-frame.html

especiais com a extensão `.jar` ou ser pastas contendo uma série de arquivos `.class`. Depois que começamos a utilizar o *classpath*, o Java ganha uns problemas de memória, então, é bom lembrá-lo de que seu programa está na mesma pasta onde você está digitando o comando acrescentando o diretório "." (que é o diretório corrente).

A sua linha de comandos, se estivar usando Mac ou Linux, deverá ser assim:

```
$ java -cp lib/hsqldb.jar:. SGBD
```

Vou ler para você "java", espaço, menos "cp", espaço, "lib", barra, "hsqldb" ponto "jar", dois pontos (um sobre o outro), ponto, espaço, "SGBD".

A sua linha de comandos, se estivar usando o outro sistema operacional, deverá ser assim:

```
$ java -cp lib/hsqldb.jar;. SGBD
```

Qual a diferença? No *unix like* o `classpath` é separado por ":" e no *Windows* é separado por ";".

Caso o seu programa apresente problemas para conectar com o banco de dados, muito provavelmente você estará errando o caminhdo da pasta que contém o `hsqldb.jar`.

Você se lembra da figura 34.2 do capítulo sobre o HSQLDB? Pois então, você deve ter uma janela como essa aberta, senão seu programa não encontrará o HSQLDB e lançará uma `java.sql.SQLTransientConnectionException ... Caused by: java.net.ConnectException: Connection refused: connect`

```
$ java -cp lib/hsqldb.jar:. SGBD
ERRO:SQLException: nâ-'o foi possâ-'vel conectar ao banco de dados.
java.sql.SQLTransientConnectionException: java.net.ConnectException: Connection
    refused: connect
        at org.hsqldb.jdbc.JDBCUtil.sqlException(Unknown Source)
        at org.hsqldb.jdbc.JDBCUtil.sqlException(Unknown Source)
        at org.hsqldb.jdbc.JDBCConnection.<init>(Unknown Source)
        at org.hsqldb.jdbc.JDBCDriver.getConnection(Unknown Source)
        at org.hsqldb.jdbc.JDBCDriver.connect(Unknown Source)
        at java.sql.DriverManager.getConnection(Unknown Source)
        at java.sql.DriverManager.getConnection(Unknown Source)
        at SGBD.getConnection(SGBD.java:8)
        at SGBD.main(SGBD.java:15)
Caused by: org.hsqldb.HsqlException: java.net.ConnectException: Connection refused:
    connect
        at org.hsqldb.ClientConnection.openConnection(Unknown Source)
        at org.hsqldb.ClientConnection.initConnection(Unknown Source)
        at org.hsqldb.ClientConnection.<init>(Unknown Source)
        ... 7 more
Caused by: java.net.ConnectException: Connection refused: connect
        at java.net.DualStackPlainSocketImpl.connect0(Native Method)
        at java.net.DualStackPlainSocketImpl.socketConnect(Unknown Source)
        at java.net.AbstractPlainSocketImpl.doConnect(Unknown Source)
        at java.net.AbstractPlainSocketImpl.connectToAddress(Unknown Source)
        at java.net.AbstractPlainSocketImpl.connect(Unknown Source)
        at java.net.PlainSocketImpl.connect(Unknown Source)
        at java.net.SocksSocketImpl.connect(Unknown Source)
        at java.net.Socket.connect(Unknown Source)
        at java.net.Socket.connect(Unknown Source)
        at java.net.Socket.<init>(Unknown Source)
        at java.net.Socket.<init>(Unknown Source)
        at org.hsqldb.server.HsqlSocketFactory.createSocket(Unknown Source)
        ... 10 more
```

Ao final, se você sobreviveu a maratona desta parte até aqui, você terá carregado um programa que consegue se conectar ao banco de dados, pronto para nos ajudar. Logo você deverá ver:

```
Conexão realizada com sucesso!
```

Desafios para estudo

1. Comente a linha 7 e substitua a linha 8 por:

   `Connection c = DriverManager.getConnection( "jdbc:sqlserver://localhost:1433;user=sa;passwo`

   e tente se conectar ao SQLServer. Lembre-se de substituir a senha pela senha que você definiu na instalação do SQL Server Express e incluir a biblioteca correta no `classpath`.

Capítulo 37

Inserindo dados estilo na marra

Finalmente, depois de tantos capítulos sem código Java vamos matar a saudade de programar e vivenciar um pouco a vida de *hacker*. Respire fundo, peça uma pizza, e sem perda de tempo, digite o programa a seguir. Ele faz uma conexão com o banco de dados, apresenta um menu e grava os dados de uma editora lá de três formas muito peculiares.

```
1  import java.util.Scanner;
2  import java.sql.DriverManager;
3  import java.sql.Connection;
4  import java.sql.SQLException;
5  import java.sql.Statement;
6  import java.sql.PreparedStatement;
7
8  class Editora {
9    String nome;
10   String cidade;
11   String estado;
12 }
13
14 public class Inclui {
15   Connection conexao;
16   public Inclui() throws SQLException, Exception {
17     conexao = getConnection();
18     System.out.println("Conexão realizada com sucesso!");
19     declPreparada = conexao.prepareStatement("INSERT INTO EDITORA (NOME, CIDADE, UF)
       VALUES ( ?, ?, ?)");
20   }
21   public Connection getConnection() throws Exception, SQLException {
22     Class.forName("org.hsqldb.jdbc.JDBCDriver" );
23     Connection novaConexao = DriverManager.getConnection("jdbc:hsqldb:hsql://
       localhost/", "SA", "");
24     return novaConexao;
25   }
26   public void insertEditoraEstatica() throws SQLException {
27     Statement declaracao = conexao.createStatement();
28     declaracao.executeUpdate("INSERT INTO EDITORA (NOME, CIDADE, UF) VALUES ('
       Independente', 'Brasília', 'DF')");
29     System.out.println("Registro estático inserido!");
30     declaracao.close();
31   }
32   public void insertEditoraConcatenando(Editora e) throws SQLException {
33     Statement declaracao = conexao.createStatement();
34     String comandoSQL = "INSERT INTO EDITORA (NOME, CIDADE, UF) VALUES ('"+e.nome+"
       ', '"+e.cidade+"', '"+e.estado+"')";
35     declaracao.executeUpdate(comandoSQL);
36     System.out.println("Registro inserido com o comando "+comandoSQL);
37     declaracao.close();
38   }
39
40   public PreparedStatement declPreparada = null;
41   public void insertEditoraPreparada(Editora e) throws SQLException {
42     declPreparada.setString(1, e.nome);
43     declPreparada.setString(2, e.cidade);
44     declPreparada.setString(3, e.estado);
45     declPreparada.executeUpdate();
46     System.out.println("Registro inserido via declaração preparada!");
47   }
48
49   public Editora getEditora(Scanner s) {
50     Editora e = new Editora();
```

```
51      System.out.println("Digite o nome da editora e tecle enter:");
52      e.nome = s.nextLine();
53      System.out.println("Digite a cidade onde fica a editora e tecle enter:");
54      e.cidade = s.nextLine();
55      System.out.println("Digite a UF do estado onde fica a editora e tecle enter:");
56      e.estado = s.nextLine();
57      return e;
58    }
59    public void close() throws SQLException {
60      declPreparada.close();
61      conexao.close();
62    }
63    public static void main(String[] args) {
64      Scanner s = new Scanner(System.in);
65      try {
66        Inclui i = new Inclui();
67        int item = 0;
68        boolean novamente = true;
69        do {
70          System.out.println("O que você quer fazer? Digite o número correspondente.")
      ;
71          System.out.println("1 - Inserir a editora estática");
72          System.out.println("2 - Inserir uma editora com concatenação");
73          System.out.println("3 - Inserir uma editora com PreparedStatement");
74          System.out.println("4 - Sair");
75          item = s.nextInt(); s.nextLine();
76          switch (item) {
77            case 1: i.insertEditoraEstatica(); break;
78            case 2: i.insertEditoraConcatenando(i.getEditora(s)); break;
79            case 3: i.insertEditoraPreparada(i.getEditora(s)); break;
80            case 4: novamente = false; break;
81            default: System.out.println("\nOpção não prevista!\n"); break;
82          }
83        } while (novamente);
84        i.close();
85      } catch (SQLException e) {
86        System.err.println("ERRO:SQLException: algo não saiu bem ao falar com o banco
       de dados.");
87        e.printStackTrace();
88        return;
89      } catch (Exception e) {
90        System.err.println("ERRO:Exception: não foi possível carregar o driver JDBC do
       HSQLDB, talvez?");
91        e.printStackTrace();
92        return;
93      }
94    }
95  }
```

Como você viu, esse programa é simples. Nós já criamos a tabela `Editora` nos capítulos sobre instalação do banco de dados e também já vimos como fazer a conexão com o banco de dados. Agora é hora de colocar os dados na tabela deste banco. Para isso, nós temos três opções. Podemos colocar os comandos SQL de forma estática, como nas linhas 21 a 24, podemos fazer a coisa do jeito mais fácil, concatenando variáveis, como nas linhas 27 a 32 ou podemos fazer da forma correta[1], como nas linhas 37 a 40. Por quê eu fiz tantas vezes a mesma coisa? Porque os dados são a coisa mais preciosa que uma empresa ou uma pessoa pode ter. E como programadores, nós vamos cuidar desses dados e não podemos fazer isso de qualquer jeito. Você não vai querer o número do seu cartão de crédito por aí ou a lista das contas a receber sendo apagada sem deixar vestígios, quer? Isso vai ficar claro nos *Desafios para estudo*.

Se você não tem lá um espírito muito *hacker* mas tem boa memória, então se lembrará que o método das linhas 15 a 19 providencia uma conexão com o banco de dados. Na linha 21, temos o comando a partir da conexão obtem um objeto da classe `Statement`. É através desse objeto que o nosso banco de dados pode receber *qualquer* comando SQL. Podemos criar tabelas, bancos de dados, apagar, selecionar valores, enfim, podemos qualquer coisa que as permissões do banco de dados permitam. Assim, executamos um comando SQL `INSERT` através da chamada, na linha 22, do método `executeUpdate` do nosso objeto da classe `Statement`, no caso nós chamamos o objeto de `declaracao`[2]. Neste primeiro método, que eu chamei de `insereEditoraEstatica` nós estamos fazendo um comando que não pode ser modificado. O que é meramente ilustrativo.

[1] Considerando as nossas habilidades do presente. Existem formas ainda mais sofisticadas de inserir dados em um banco de dados, mas isso é assunto para depois.

[2] Não é lá um nome muito inteligente, eu sei. Declaração é a tradução de `Statement` para a Língua Portuguesa.

Na segunda abordagem, no método `insereEditoraConcatenando`, nós vamos fazer algo um pouco mais divertido, que é compor o comando SQL com algumas variáveis obtidas a partir da classe `Editora`. Dessa forma, na linha 28 estamos criando um comando SQL concatenando os parâmetros no comando. Temos que observar algumas coisas bem importantes aqui. Então, concentre-se! Primeiro: o que marca o início e o fim de uma `String` na linguagem SQL são os apóstrofos. No Java, são as aspas. Sabendo disso, podemos então combinar `String`s no Java e no SQL se tivermos bastante atenção. Lá pelas tantas você tem a `String` Java `... VALUES ('` terminando no apóstrofe, delimitada pelas aspas. Quando terminamos a `String` Java podemos então concatenar o comando SQL com a variável `e.nome`, por exemplo, e depois continuar o comando SQL, que espera o fim da `String` que representa o primeiro campo com um apóstrofe. Por isso é que concatenamos a `String ', '` antes de colocar o próximo valor `e.cidade` e assim sucessivamente até terminar o comando SQL com `')`. Segundo: se você está a vontade com a concatenação de `String`s para fazer o comando SQL, não se anime muito. Essa abordagem é vulnerável a um ataque ao código para obtenção de dados, ou destruição desses, conhecido como *SQL Injection*. No ataque de *SQL Injection* um humano tenta executar comandos SQL que você não colocou no código com o objetivo de desviar o comportamento do seu programa. Pode isso, Armando? Pode. E se pode, será feito. Quando? É uma questão de tempo. Então, não recomendo que você use essa abordagem. Ela funciona, mas é perigosa.

A terceira abordagem é a mais elegante, embora exija um pouco mais de código. Para ela funcionar, precisamos olhar com atenção as linhas 34, 59, 36, 37, 38, 39 e 78. O que essas linhas fazem? Vamos tratá-las a seguir.

Bem, a linha 34 declara um objeto da classe `PreparedStatement`[3]. Um objeto da classe `PreparedStatement` é uma operação de banco de dados pré interpretada e que pode ser reutilizada muitas vezes de maneira rápida e fácil. Os parâmetros utilizados em um `PreparedStatement` são marcados como `"‘?"'`, a `"‘?"'` também é conhecida como parâmetro *IN*. Lembre-se de utilizar sempre tipos compatíveis conforme os valores que você está atribuindo. Um `PreparedStatement` vai fazer coisas interessantes, como ver se o tipo de parâmetro que está sendo passado para as consultas está correto. Quando criamos uma tabela dizemos o tipo de cada coluna, e quando usamos as abordagens anteriores, nem o Java nem o Banco de Dados podem nos ajudar caso tenhamos passado um tipo de parâmetro errado. Outra coisa que o `PreparedStatement` vai fazer é garantir que o valor daquele parâmetro seja inserido completamente na coluna, com isso evitamos o ataque de *SQL Injection*. Outra vantagem é que o `PreparedStatement` é mais rápido que o `Statement`. O banco de dados não precisa reinterpretar o comando cada vez que você o executa. Isso faz muita diferença quando estamos falando de milhares de linhas feitas em sequência. E mais uma vantagem é que parece bem mais fácil ler um `PreparedStatement` do que aquele monte de aspas e apóstrofes, não é mesmo?

A linha 59 só precisa ser executada uma única vez em todo o código, isso porque depois que ela retorna o `PreparedStatement` nós só precisamos atualizar os parâmetros e podemos reutilizá-lo a vontade. E isso é bem legal.

As linhas 36 a 38 farão a substituição dos parâmetros *IN* (as `"‘?"'` presentes na linha 34, lembra?). Aqui você precisa ficar atento. Os parâmetros *IN* começam em 1, e não em 0. Se utilizarmos 0 ganhamos de presente erro `java.sql.SQLException: Invalid column index` e nosso programa não funcionará. Fique atento!

A última linha que precisa da nossa valiosa atencão é a linha 78, onde fechamos o `PreparedStatement` liberando os recursos no banco de dados relacionados a essa consulta específica. A linha 79 também é importante, pois, como dissemos no capítulo anterior, ela libera todos os recursos no banco de dados relacionados a esta sessão do programa.

Cabe uma observação ainda, antes de concluirmos este capítulo, sobre desempenho. Como nosso programa é muito simples, dificilmente perceberemos a diferença entre fechar ou não a conexão com o banco de dados. Você precisa lembrar que quando fazemos um `getConnection()` ou um `prepareStatement` estamos travando recursos no servidor de banco de dados e esses recursos ficam travados durante toda a execução do nosso programa. Enquanto o humano fica lá, pensando no que vai fazer, o recurso está travado. Então, considere com atenção o lugar do código onde a conexão será aberta, onde as declarações de queries serão feitas e onde tudo isso será fechado. Abrir uma conexão gasta tempo, tempo este usado pelo servidor de banco de dados (aquela janela a toa que fica aberta, ou o SQL Express que está executando como serviço) na alocação dos recursos.

As demais linhas você já é grandinho para entender sozinho, então, vamos àquele refresco,

[3] Consulte a documentação do `PreparedStatement` em http://docs.oracle.com/javase/7/docs/api/java/sql/PreparedStatement.html

pois sua cabeça deve estar fervendo! A pizza já chegou?

O que você deve ver

```
$ java -cp lib/hsqldb.jar:. Inclui
Conexão realizada com sucesso!
O que você quer fazer? Digite o número correspondente.
1 - Inserir a editora estática
2 - Inserir uma editora com concatenação
3 - Inserir uma editora com PreparedStatement
4 - Sair
1
Registro estático inserido!
O que você quer fazer? Digite o número correspondente.
1 - Inserir a editora estática
2 - Inserir uma editora com concatenação
3 - Inserir uma editora com PreparedStatement
4 - Sair
2
Digite o nome da editora e tecle enter:
Editora na marra
Digite a cidade onde fica a editora e tecle enter:
Goiania
Digite a UF do estado onde fica a editora e tecle enter:
GO
Registro inserido com o comando INSERT INTO EDITORA (NOME, CIDADE, UF) VALUES ('
    Editora na marra', 'Goiania', 'GO')
O que você quer fazer? Digite o número correspondente.
1 - Inserir a editora estática
2 - Inserir uma editora com concatenação
3 - Inserir uma editora com PreparedStatement
4 - Sair
3
Digite o nome da editora e tecle enter:
Ediprata
Digite a cidade onde fica a editora e tecle enter:
Sao Paulo
Digite a UF do estado onde fica a editora e tecle enter:
SP
Registro inserido via declaração preparada!
O que você quer fazer? Digite o número correspondente.
1 - Inserir a editora estática
2 - Inserir uma editora com concatenação
3 - Inserir uma editora com PreparedStatement
4 - Sair
4
```

Mais uma surpresa! Você também deve carregar o gerenciador de banco de dados e executar um "'SELECT * FROM EDITORA"' para ver os dados que você acabou de inserir. É emocionante, não? Ah, confesse!

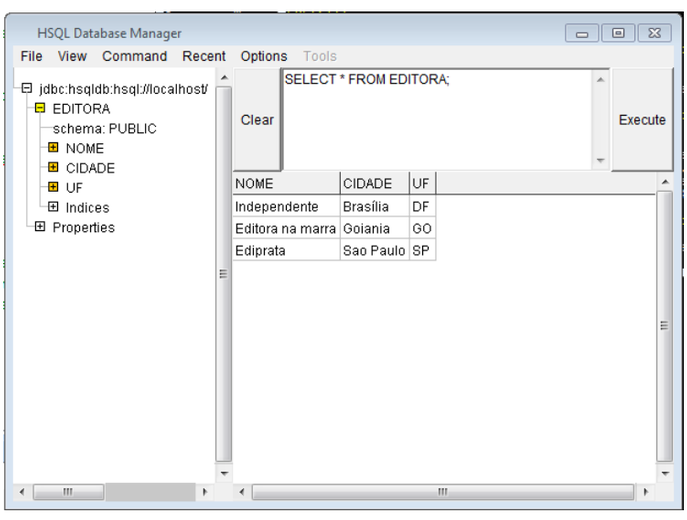

Figura 37.1: Banco de dados após as inserções

Desafios para estudo

1. Vamos testar um ataque conhecido com o *SQL Injection*. Nesse ataque, ao invés de colocar os dados como o sistema espera, nós embutimos um comando SQL no nome do campo. Tente colocar no campo UF exatamente o seguinte código:

   ```
   '); delete from editora where 1=1; --
   ```

   e veja o que acontece com os dados que estão inseridos. Tente fazer o mesmo com um PreparedStament e veja o que acontece.

2. Recrie a tabela Editora, fazendo com que UF tenha 200 caracteres ao invés de 2 e repita o experimento. O que você vê? (Dica: use o comando `DROP TABLE EDITORA`)

3. Que tal adaptar o programa para usar o SQL Server Express?

Capítulo 38

Buscando dados estilo na marra

Animado com o acesso ao banco de dados? O caminho até aqui foi duro, mas ele é compensador. Você já conseguiu colocar linhas na tabela, agora nosso programa agora vai buscar os dados de uma editora lá no banco, usando um campo à escolha entre os campos da tabela, conforme a nossa opção através de um menu. Se você for bem empolgado, vai perceber que motores famosos de busca na internet começaram assim (ou de uma maneira bem parecida...).

Sem mais apresentações, vamos ao programa.

```
1  import java.util.Scanner;
2  import java.sql.DriverManager;
3  import java.sql.Connection;
4  import java.sql.SQLException;
5  import java.sql.PreparedStatement;
6  import java.sql.ResultSet;
7
8  class Editora {
9    String nome = null;
10   String cidade = null;
11   String estado = null;
12 }
13
14 public class Busca {
15   private conexao = null;
16   private PreparedStatement pNome;
17   private PreparedStatement pCidade;
18   private PreparedStatement pEstado;
19   private ResultSet rs = null;
20
21   public Busca() throws Exception, SQLException {
22     conexao = getConnection();
23     System.out.println("Conexão realizada com sucesso!");
24     pNome = conexao.prepareStatement("SELECT * FROM EDITORA WHERE NOME = ?");
25     pCidade = conexao.prepareStatement("SELECT * FROM EDITORA WHERE CIDADE = ?");
26     pEstado = conexao.prepareStatement("SELECT * FROM EDITORA WHERE UF = ?");
27   }
28   public Connection getConnection() throws Exception, SQLException {
29     Class.forName("org.hsqldb.jdbc.JDBCDriver" );
30     Connection novaConexao = DriverManager.getConnection("jdbc:hsqldb:hsql://
       localhost/", "SA", "");
31     return novaConexao;
32   }
33   public Editora findEditora(Editora e) throws SQLException {
34     PreparedStatement pComum = null;
35     if(e == null)
36       return null;
37     if(e.nome != null) {
38       pNome.setString(1, e.nome);
39       pComum = pNome;
40     } else if (e.cidade != null){
41       pCidade.setString(1, e.cidade);
42       pComum = pCidade;
43     } else if (e.estado != null){
44       pEstado.setString(1, e.estado);
45       pComum = pEstado;
46     }
47     if(pComum == null) {
48       return null;
49     } else {
50       rs = pComum.executeQuery();
```

```
51      if ( rs.next() ) {
52        return retryEditora();
53      }
54    }
55    return null;
56  }
57  private Editora retryEditora() throws SQLException {
58    Editora ne = new Editora();
59    ne.nome = rs.getString("NOME");
60    ne.cidade = rs.getString(2);
61    ne.estado = rs.getString("UF");
62    return ne;
63  }
64  public Editora nextEditora() throws SQLException {
65    if( rs != null ) {
66      if ( rs.next() ) {
67        return retryEditora();
68      } else {
69        rs.close();
70        rs = null;
71      }
72    }
73    return null;
74  }
75  public void showEditora(Editora e) {
76    System.out.println("Editora:"+e.nome);
77    System.out.println("Cidade :"+e.cidade);
78    System.out.println("Estado :"+e.estado+"\n");
79  }
80  public boolean hasResultado() {
81    if(rs != null)
82      return true;
83    return false;
84  }
85  public void close() throws SQLException {
86    if(rs != null)
87      rs.close();
88    pNome.close();
89    pCidade.close();
90    pEstado.close();
91    conexao.close();
92  }
93  public static void main(String[] args) {
94    Scanner s = new Scanner(System.in);
95    Editora busca;
96    Editora resultado;
97    String chave;
98
99    try {
100     Busca b = new Busca();
101     int item = 0;
102     boolean novamente = true;
103     resultado = null;
104     do {
105       System.out.println("O que você quer fazer? Digite o número correspondente.")
    ;
106       System.out.println("1 - Buscar editora pelo nome");
107       System.out.println("2 - Buscar editora pela cidade");
108       System.out.println("3 - Buscar editora pelo UF");
109       if( b.hasResultado() )
110         System.out.println("4 - Próximo resultado");
111
112       System.out.println("9 - Sair");
113       busca = new Editora();
114       item = s.nextInt(); s.nextLine();
115       if(item <= 3) {
116         switch (item) {
117           case 1: System.out.println("Digite o nome da editora que está buscando e
     tecle enter."); break;
118           case 2: System.out.println("Digite o nome da cidade onde está a editora
    que está buscando e tecle enter."); break;
119           case 3: System.out.println("Digite o nome da UF onde fica a editora que
    está buscando e tecle enter."); break;
120         }
121         chave = s.nextLine();
122         switch (item) {
123           case 1: busca.nome  =chave; resultado = b.findEditora(busca); break;
124           case 2: busca.cidade=chave; resultado = b.findEditora(busca); break;
```

```
125             case 3: busca.estado=chave; resultado = b.findEditora(busca); break;
126           }
127         }
128         switch(item) {
129           case 1:
130           case 2:
131           case 3: break;
132           case 4: resultado = b.nextEditora(); break;
133           case 9: novamente = false; break;
134           default: System.out.println("\nOpção não prevista!\n"); break;
135         }
136         if( resultado != null ) {
137           b.showEditora(resultado);
138           resultado = null;
139         } else {
140           if(item != 9)
141             System.out.println("Nenhum resultado para mostrar.");
142         }
143       } while (novamente);
144       b.close();
145     } catch (SQLException e) {
146       System.err.println("ERRO:SQLException: algo não saiu bem ao falar com o banco
    de dados.");
147       e.printStackTrace();
148       return;
149     } catch (Exception e) {
150       System.err.println("ERRO:Exception: não foi possível carregar o driver JDBC do
     HSQLDB, talvez?");
151       e.printStackTrace();
152       return;
153     }
154   }
155 }
```

Nosso segundo programa vai recuperar os dados que já foram incluídos na tabela previamente. Nas linhas 1 a 6 vamos importar as classes de apoio que nos ajudam com entrada de dados (`Scanner`, é sua amiga já, não?), localizam o *driver* do banco de dados, conectam nosso programa ao banco, nos avisam se algo sair errado (`SQLException`), nos ajudam a fazer um código seguro (`PreparedStatement`) e conjunto de resultados (`ResultSet`).

Este programa parece assustador e grande. Mas você não deve temê-lo. Vou prosseguir nos comentários antes de falar mais do `ResultSet`, que é a novidade da vez.

Definimos uma classe `Editora` como um registro (linhas 8 a 12), a la primeiro livro, para simplificar o código. Deveríamos ter declarado os atributos como private e feito os `get`*ers* e `set`*ers* apropriados, só achei que você iria gostar de digitar um pouco menos.

Vamos definir uma classe chamada `Busca` que guarda uma conexão, um objeto `PreparedStatement` para cada coluna da tabela que queremos buscar e um `ResultSet`. Um `ResultSet` é uma tabela de dados resultante de um subconjunto do banco de dados normalmente gerado a partir de uma consulta a esse banco.

Na linha 50 o `ResultSet` virá como resultado da chamada ao método `executeQuery` do `PreparedStatement`. *`Statement` e `ResultSet` classes com sabedoria e responsabilidade utilizadas devem ser; para você elas poder grande traz!*[1] Para você que fugiu ou tem pavor das escolas *nerds*, explico. A classe `ResultSet` deve ser utilizada de maneira adequada pois seu potencial de danos ao desempenho de um programa é grande.

Devemos fazer a consulta ao banco de dados de forma mais completa possível para que não precisemos tratar, no programa Java, questões que o banco de dados pode tratar para nós. É mais ou menos assim: peça todas as linhas que você precisar ao banco de dados de uma vez, um único `Statement.executeQuery` retornando um `ResultSet`. O `ResultSet` foi feito para conter várias linhas e isso é bem melhor que pedir uma linha de cada vez, chamando `executeQuery` muitas vezes.[2]. Se você gosta de exemplos mais domésticos, se uma linha do banco de dados for igual a um ovo, e precisamos de 12 ovos, pedir uma única linha do banco de dados através de um `executeQuery`

[1] Isso deve te lembrar pelo menos uns dois heróis! Use a força com sabedoria e grandes poderes trazem grandes responsabilidades.

[2] Isso acontece porque o *driver* do banco de dados vai precisar iniciar várias transferências de dados através da rede ao invés de fazer apenas uma. Para cada transferência é preciso um intervalo de tempo, chamado latência, que é pago uma vez para cada `executeQuery`, além do tempo de transferência e outras burocracias (ou *overhead*, para eu me sentir mais inteligente).

é como mandar alguém 12 vezes à feira. É melhor mandar buscar um `ResultSet` (ou uma dúzia!) de ovos.

As linhas 21 a 27 inicializam o banco de dados e os objetos que utilizaremos para buscar a `Editora` a partir da tabela. Para isso empregamos o comando `SELECT`. A sintaxe básica de um comando `SELECT` é:

```
SELECT COLUNA1[,COLUNA2][,COLUNA3][...]  FROM [TABELA] WHERE [EXPRESSÃO];
```

Podemos substituir as `COLUNA1[,COLUNA2][,COLUNA3][...]` por `*` (asterisco) para buscar todas as colunas da tabela.

As linhas 28 a 32 você já viu antes. Não é impressão sua.

O método `findEditora` recebe um objeto do tipo `Editora` com o parâmetro da busca. Aqui nas linhas 33 a 60 testamos os parâmetros, afinal, o usuário do seu programa, seja ele uma pessoa ou programador, pode ter um probleminha de cabeça e esquecer das coisas... ... ... O que eu dizia mesmo?

Bem, nas linhas 33 a 60 nós instanciamos na variável `pComum` o `PreparedStatement` adequado à busca. Se nenhuma das propriedades está definida, nossa consulta[3] retornará nulo, senão estamos prontos para executar consulta.

A linha 50 é quem faz o programa ir, de fato, ao banco de dados executar a consulta selecionada e devidamente parametrizada. Esse parâmetro, da mesma forma que você viu no capítulo anterior, vai substituir a `?` no `PreparedStatement` e isso vai acontecer em uma das linhas 38, 41 ou 44. Aqui usamos o método `setString` pois o valor da coluna na tabela é do tipo `VARCHAR` e na classe `Editora` é do tipo `String`. Se na tabela tivéssemos uma coluna do tipo `INTEGER` utilizaríamos o método setInt. Essa interrogação também é chamada de *IN parameter* na documentação do Java[4].

A linha 51 trás uma chamada `next()` que retorna um `boolean`. Se ele for verdadeiro, então, a consulta tem uma linha para processarmos e as colunas selecionadas estarão disponíveis através do nome da coluna ou do número de ordem dessa coluna, a partir de 1. Além do nome, ou do número de ordem dessa coluna, precisamos utilizar um método `get` adequado para recuperar o valor da tabela. Como todas as nossas colunas são do tipo `String` usamos o método `getString`. Além do `getString()` temos os métodos `getInt, getDate, getDouble` entre outros.

O método `nextEditora` da nossa classe `Busca` se parece bastante com as linhas 51 a 57 e fica como desafio deixar este programa mais enxuto.

A linha 70 aparece um método chamado `close()`. Este método sinaliza para o *driver* que ele pode descartar qualquer informação temporária, agilizando o uso de recursos do gerenciador de banco de dados[5]. Se você por acaso esquecer, o Java te ajuda fechando o `ResultSet` automaticamente quando o `Statement` que o gerou é fechado, reexecutado ou usado para recuperar o próximo resultado em uma sequência de múltiplos resultados.[6]

Os métodos das linhas 76 em diante não devem ser nenhum segredo para você, que chegou até aqui *na marra*! Apenas vou dizer que no método `main` eu crio um objeto da classe `Editora` para receber os valores digitados e um para armazenar o último resultado retornado pelo banco de dados. Assim podemos exibí-lo e alterá-lo a qualquer tempo.

Um último ponto de atenção. Veja que na linha 145 quando nosso usuário não quiser mais buscar por editoras nós fazemos uma chamada ao nosso método `close()` da nossa classe `Busca` que por sua vez chama os métodos `close()` necessários liberando todos os `Statement`s, `ResultSet`s e `Connection`s. Afinal, somos educados e devolvemos tudo o que pegamos e fechamos tudo o que abrimos, *não é*?

O que você deve ver

```
$ java -cp lib/hsqldb.jar:. Busca
```

[3] Ou *query*, se você quiser dominar o jargão anglo-saxão e expor seu lado internacional.

[4] Consulte `https://docs.oracle.com/javase/7/docs/api/java/sql/PreparedStatement.html` para mais detalhes

[5] Imagine um milhão de consultas do seu programa em execução, pode ser um bocado de memória!

[6] Não vale para linhas dos tipos Blob, Clob e NClob que precisam ter seu método `free()` invocado.

```
Conexãorealizada com sucesso!
O que você quer fazer? Digite o número correspondente.
1 - Buscar editora pelo nome
2 - Buscar editora pela cidade
3 - Buscar editora pelo UF
9 - Sair
1
Digite o nome da editora que Está buscando e tecle enter.
Editora na marra
Editora:Editora na marra
Cidade :Goiania
Estado :GO

O que você quer fazer? Digite o número correspondente.
1 - Buscar editora pelo nome
2 - Buscar editora pela cidade
3 - Buscar editora pelo UF
4 - Próximo resultado
9 - Sair
2
Digite o nome da cidade onde Está a editora que Está buscando e tecle enter.
Brasilia
Editora:Pequi
Cidade :Brasilia
Estado :DF

O que você quer fazer? Digite o número correspondente.
1 - Buscar editora pelo nome
2 - Buscar editora pela cidade
3 - Buscar editora pelo UF
4 - Próximo resultado
9 - Sair
3
Digite o nome da UF onde fica a editora que Está buscando e tecle enter.
SP
Editora:Ediprata
Cidade :Sao Paulo
Estado :SP

O que você quer fazer? Digite o número correspondente.
1 - Buscar editora pelo nome
2 - Buscar editora pela cidade
3 - Buscar editora pelo UF
4 - Próximo resultado
9 - Sair
4
Nenhum resultado para mostrar.
O que você quer fazer? Digite o número correspondente.
1 - Buscar editora pelo nome
2 - Buscar editora pela cidade
3 - Buscar editora pelo UF
9 - Sair
1
Digite o nome da editora que Está buscando e tecle enter.
Xispa
Nenhum resultado para mostrar.
O que você quer fazer? Digite o número correspondente.
1 - Buscar editora pelo nome
2 - Buscar editora pela cidade
3 - Buscar editora pelo UF
9 - Sair
9
```

Desafios para estudo

1. Ficar escolhendo em qual coluna procurar parece muito coisa de computador, e, como não somos computadores, faça o programa procurar a informação entrada nas três colunas sozinho. Você consegue?

2. Implemente os getters e setters da classe Editora e coloque seus atributos como privados. Crie um construtor que receba nome, cidade e estado de uma vez. Seu programa ficará mais ou menos legível no método `nextEditora()`?

3. Que tal buscar por cidade e uf? Se quisermos achar "'Taguatinga - TO"' ao invés de "'Taguatinga - DF"'? Tente diversificar suas opções pondo uma opção a mais "'4 - Buscar por cidade

e UF"'. Acrescente um `PreparedStatement` que busque pelos parâmetros Cidade e Estado e modifique o método `findEditora()`. Não esqueça de alterar os arredores da linha 133...

4. As linhas 51 a 57 e de 62 a 73 são muito parecidas. Podemos fazer uma chamada a `nextEditora()`?

Capítulo 39

Mudando as coisas sem fazer força

Agora vamos modificar uma linha do banco de dados com nosso programa. O que fazemos, na verdade, é substituir o valor de uma coluna na tabela por outro. Isso é o alterar. Para simplificar as coisas, e fazer você digitar um pouco menos, vamos usar chaves naturais[1] e reaproveitar bastante código. O programa deste capítulo busca uma linha que contem exatamente a editora que queremos modificar e a substitui por uma outra que digitamos, inteira. Se quisermos alterar, realmente, apenas um dos campos, devemos modificar o comando SQL adequadamente, mas, ei, estou adiantando as coisas, vá digitar o programa agora mesmo!

Antes de sair desenfreado digitando coisas, eu disse que ia simplificar, então, vou quebrar o seu galho. O programa que altera utiliza os dois programas feitos nos últimos dois capítulos, então, tenha certeza de que eles estão na mesma pasta do programa a seguir, senão, não vai funcionar. Também comentei o código, logo, basta você copiar o método `main()` do programa de busca e alterar de acordo com os comentários. Se for um ás da digitação, esqueça o que eu disse e copie tudo logo, vai funcionar do mesmo jeito.

```
1  import java.util.Scanner;
2  import java.sql.DriverManager;
3  import java.sql.Connection;
4  import java.sql.SQLException;
5  import java.sql.PreparedStatement;
6  import java.sql.ResultSet;
7
8  class Editora {
9    String nome = null;
10   String cidade = null;
11   String estado = null;
12 }
13
14 // NOVIDADADES COMEÇAM AQUI
15 public class Altera {
16   Connection conexao = null;
17   public PreparedStatement pAltera;
18   public Altera() throws Exception, SQLException {
19     conexao = getConnection();
20     System.out.println("Conexão realizada com sucesso!");
21     pAltera = conexao.prepareStatement("UPDATE EDITORA SET NOME  =?, CIDADE=?, UF=?
       WHERE NOME=? AND CIDADE=? AND UF=?");
22   }
23   public Connection getConnection() throws Exception, SQLException {
24     Class.forName("org.hsqldb.jdbc.JDBCDriver" );
25     Connection novaConexao = DriverManager.getConnection("jdbc:hsqldb:hsql://
       localhost/", "SA", "");
26     return novaConexao;
27   }
28   public void updateEditora(Editora original, Editora novo) throws SQLException {
29     PreparedStatement pComum = null;
30     pAltera.setString(1, novo.nome);
31     pAltera.setString(2, novo.cidade);
32     pAltera.setString(3, novo.estado);
33     pAltera.setString(4, original.nome);
34     pAltera.setString(5, original.cidade);
35     pAltera.setString(6, original.estado);
36     pAltera.executeUpdate();
```

[1] Uma chave natural usa o valor de um campo que faz parte da representação real do problema para diferenciar registros únicos em uma classe ou entidade, como CPF ou número do seguro social. Uma chave artificial muito comum é usar um número sequencial e sequencias distintas para cada tabela, classe ou entidade.

```
37      conexao.commit();
38      System.out.println("Registro alterado!");
39    }
40    public void close() throws SQLException {
41      pAltera.close();
42      conexao.close();
43    }
44  // E TERMINAM AQUI. AS LINHAS ABAIXO EU COPIEI DO PROGRAMA INCLUI.JAVA MUDANDO AS
      LINHAS MARCADAS COM "AQUI"
45    public static void main(String[] args) {
46      Scanner s = new Scanner(System.in);
47      Editora busca;
48      Editora resultado;
49      Editora ultimoResultado = null; // AQUI
50      Editora novosValores = null; // AQUI
51      String chave;
52
53      try {
54        Busca b = new Busca();
55        Altera a = new Altera();  // AQUI
56        Inclui i = new Inclui(); // AQUI
57        int item = 0;
58        boolean novamente = true;
59        resultado = null;
60        do {
61          System.out.println("O que você quer fazer? Digite o número correspondente.")
      ;
62          System.out.println("1 - Buscar editora pelo nome");
63          System.out.println("2 - Buscar editora pela cidade");
64          System.out.println("3 - Buscar editora pelo UF");
65          if( b.hasResultado() ) { // AQUI
66            System.out.println("4 - Próximo resultado"); // AQUI
67            System.out.println("5 - Alterar editora do último resultado"); // AQUI
68          } // AQUI
69
70          System.out.println("9 - Sair");
71          busca = new Editora();
72          item = s.nextInt(); s.nextLine();
73          if(item <= 3) {
74            switch (item) {
75              case 1: System.out.println("Digite o nome da editora que está buscando e
       tecle enter."); break;
76              case 2: System.out.println("Digite o nome da cidade onde está a editora
      que está buscando e tecle enter."); break;
77              case 3: System.out.println("Digite o nome da UF onde fica a editora que
      está buscando e tecle enter."); break;
78            }
79            chave = s.nextLine();
80            switch (item) {
81              case 1: busca.nome  =chave; resultado = b.findEditora(busca); break;
82              case 2: busca.cidade=chave; resultado = b.findEditora(busca); break;
83              case 3: busca.estado=chave; resultado = b.findEditora(busca); break;
84            }
85          }
86          switch(item) {
87            case 1:
88            case 2:
89            case 3: break;
90            case 4: resultado = b.nextEditora(); break;
91            case 5: novosValores = i.getEditora(s); b.showEditora(novosValores); a.
      updateEditora(ultimoResultado, novosValores); break; // AQUI
92            case 9: novamente = false; break;
93            default: System.out.println("\nOpção não prevista!\n"); break;
94          }
95          if( resultado != null ) {
96            b.showEditora(resultado);
97            ultimoResultado = resultado; // AQUI
98            resultado = null;
99          } else {
100           if(item != 9)
101             System.out.println("Nenhum resultado para mostrar.");
102         }
103       } while (novamente);
104       b.close();
105     } catch (SQLException e) {
106       System.err.println("ERRO:SQLException: algo não saiu bem ao falar com o banco
      de dados.");
107       e.printStackTrace();
```

```
108         return;
109       } catch (Exception e) {
110         System.err.println("ERRO: Exception: não foi possível carregar o driver JDBC do
          HSQLDB, talvez?");
111         e.printStackTrace();
112         return;
113       }
114     }
115 }
```

O programa que faz a alteração de dados da tabela é um bocado mais simles que o programa que faz consultas, isso porque já conhecemos tudo até a linha 17, pelo menos.

A primeira diferença que vamos observar é o comando utilizado no `PreparedStatement` e a quantidade de `?` (parâmetros de entrada) que ele possui. O comando `UPDATE` atualiza linhas da tabela no banco de dados e sua sintaxe, de forma simplificada, é:

```
UPDATE TABELA SET COLUNA = VALOR [, COLUNA2 = VALOR2][, COLUNA3 = VALOR3][...]
WHERE EXPRESSÃO
```

Quando o banco de dados recebe esse comando ele vai varrer todas as linhas do banco baseado na `EXPRESSÃO` e sempre que esta for verdadeiro ele vai substituir o conteúdo das colunas `COLUNA, COLUNA2, COLUNA3, ...` indicadas pelos valores passados como parâmetro `VALOR, VALOR2, VALOR3, ....` FIQUE ATENTO: se a condição for verdadeira para mais de uma linha da tabela todas as linhas onde isso ocorrer ficarão com os campos atualizados iguais! Se atualizarmos todas as colunas ficaremos com linhas duplicadas, o que pode não ser desejável. Portanto, atenção à cláusula `WHERE`. Se estiver mal escrita ela poderá afetar mais linhas que o desejado. Teste sempre e com o conjunto de dados certo! E isso você vai conseguir através de uma boa modelagem de dados. Utilizar as chaves artificiais também pode ser uma boa maneira de alterar apenas um registro utilizando-as na cláusula `WHERE`.

Observe o método `updateEditora`. Os parâmetros estão numerados de forma a corresponderem às ?s na ordem em que aparecem na linha 21. Não é possível utilizar, por exemplo, os nomes das colunas no lugar dos números.

Antes de terminar o método `updateEditora` enviamos o comando SQL para execução no banco de dados invocando o método `executeUpdate` do nosso `PreparedStatement pAltera`. Este método pode lançar uma exceção `SQLException` que deve ser tratada adequadamente, ou então, propagada para o código que chamou o método `updateEditora` que deve tratá-la com um `try/catch` ou propagá-la declarando `throws SQLException` e assim sucessivamente.

As linhas 44 em diante implementamos um menu que busca uma `Editora` e carrega uma nova usando os programas dos capítulo anteriores. Para simplificar a sua vida no código te faço digitar todo o registro novamente, afinal, você já deve saber que não tem almoço grátis.[2].

O que você deve ver

```
$ java -cp lib/hsqldb.jar:. Altera
Conexão realizada com sucesso!
Conexão realizada com sucesso!
Conexão realizada com sucesso!
O que você quer fazer? Digite o número correspondente.
1 - Buscar editora pelo nome
2 - Buscar editora pela cidade
3 - Buscar editora pelo UF
9 - Sair
3
Digite o nome da UF onde fica a editora que está buscando e tecle enter.
DF
Editora:Independente
Cidade :Brasilia
Estado :DF

O que você quer fazer? Digite o número correspondente.
1 - Buscar editora pelo nome
```

[2]Falar em lanche grátis, eu deveria comentar com você algo sobre os métodos `commit` e o conceito de transações, mas isso vai ficar para depois. Por enquanto, existe uma coisa chamada *autocommit* que vai garantir que tudo terminará bem.

```
2 - Buscar editora pela cidade
3 - Buscar editora pelo UF
4 - Próximo resultado
5 - Alterar editora do último resultado
9 - Sair
5
Digite o nome da editora e tecle enter:
Independente mesmo
Digite a cidade onde fica a editora e tecle enter:
Engenho das Lages
Digite a UF do estado onde fica a editora e tecle enter:
DF
Editora:Independente mesmo
Cidade :Engenho das Lages
Estado :DF

Registro alterado!
Nenhum resultado para mostrar.
O que você quer fazer? Digite o número correspondente.
1 - Buscar editora pelo nome
2 - Buscar editora pela cidade
3 - Buscar editora pelo UF
4 - Próximo resultado
5 - Alterar editora do último resultado
9 - Sair
9
```

Não é exatemante uma surpresa, mas imagino que você vai querer carregar o gerenciador de banco de dados e executar um "'SELECT * FROM EDITORA"' para ver os dados que você acabou de alterar. Agora foi emocionante, não foi? Vamos, você não é durão assim!

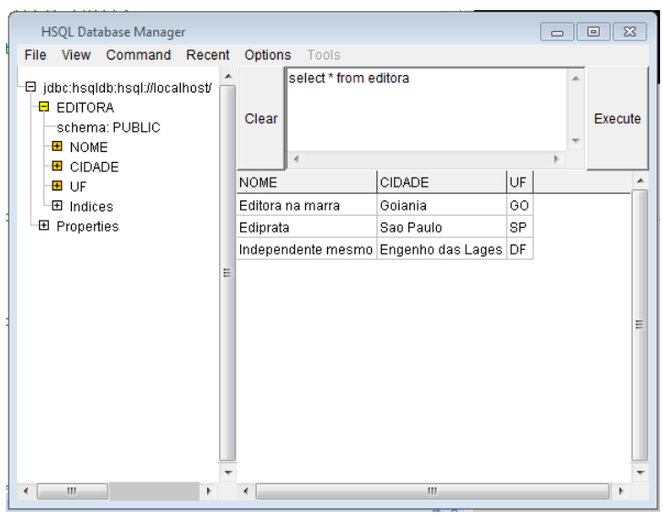

Figura 39.1: Banco de dados após as alterações

Desafios para estudo

1. Volte no programa que insere e modifique o método `getEditora` para que se a editora `s` passada não for nula que o método exiba os valores atuais e pergunte qual campo o usuário quer alterar.

2. Você observou que o código do método `updateEditora` executa o método `executeUpdate` do nosso objeto da classe `PreparedStatement`, o `pAltera`. Na declaração do método ele lança um `SQLException`. Qual seria um lugar melhor para tratar, por exemplo, o caso de o banco de dados ter sido desligado? Implemente um tratamento de exceção melhor.

Capítulo 40

Sumindo com o dado que estava... cadê?

Bem, chegamos à última operação de um CRUD. CRUD? Hum... Você precisa saber o que é um CRUD já! CRUD é a sigra em Inglês para *Create*, *Read*, *Update* e *Delete*, ou, no velho e bom Português: Criar (incluir), Ler (buscar), Atualizar (alterar) e Apagar (excluir)[1]. Ultimamente ando bonzinho. Use o programa que altera como base, e modifique-o conforme as indicações. Ou digite tudo. Dedos à obra!

```
1  import java.util.Scanner;
2  import java.sql.DriverManager;
3  import java.sql.Connection;
4  import java.sql.SQLException;
5  import java.sql.PreparedStatement;
6  import java.sql.ResultSet;
7
8  class Editora {
9    String nome = null;
10   String cidade = null;
11   String estado = null;
12 }
13
14 public class Exclui { // AQUI
15   Connection conexao = null;
16   public PreparedStatement pExclui; // AQUI
17   public Exclui() throws Exception, SQLException {
18     conexao = getConnection();
19     System.out.println("Conexão realizada com sucesso!");
20     pExclui = conexao.prepareStatement("DELETE FROM EDITORA WHERE NOME=? AND CIDADE
       =? AND UF=?"); // AQUI
21   }
22   public Connection getConnection() throws Exception, SQLException {
23     Class.forName("org.hsqldb.jdbc.JDBCDriver" );
24     Connection novaConexao = DriverManager.getConnection("jdbc:hsqldb:hsql://
       localhost/", "SA", "");
25     return novaConexao;
26   }
27   public void deleteEditora(Editora alvo) throws SQLException { // AQUI (O MÉTODO
       INTEIRO)
28     PreparedStatement pComum = null;
29     pExclui.setString(1, alvo.nome);
30     pExclui.setString(2, alvo.cidade);
31     pExclui.setString(3, alvo.estado);
32     pExclui.executeUpdate();
33     conexao.commit();
34     System.out.println("Registro excluído!");
35   }
36   public void close() throws SQLException { // AQUI
37     pExclui.close();
38     conexao.close();
39   }
40
41   public static void main(String[] args) {
42     Scanner s = new Scanner(System.in);
43     Editora busca;
44     Editora resultado;
45     Editora ultimoResultado = null;
46     Editora novosValores = null;
47     String chave;
48
```

[1]Porque *deletar* é para os fracos.

```
49      try {
50        Busca b = new Busca();
51        Exclui e = new Exclui();    // AQUI
52        Inclui i = new Inclui();
53        int item = 0;
54        boolean novamente = true;
55        resultado = null;
56        do {
57          System.out.println("O que você quer fazer? Digite o número correspondente.")
    ;
58          System.out.println("1 - Buscar editora pelo nome");
59          System.out.println("2 - Buscar editora pela cidade");
60          System.out.println("3 - Buscar editora pelo UF");
61          if( b.hasResultado() ) { // AQUI (ESTE BLOCO)
62            System.out.println("4 - Próximo resultado");
63          }
64          if (ultimoResultado != null) // AQUI
65            System.out.println("5 - Exclui editora do último resultado"); // AQUI
66
67          System.out.println("9 - Sair");
68          busca = new Editora();
69          item = s.nextInt(); s.nextLine();
70          if(item <= 3) {
71            switch (item) {
72              case 1: System.out.println("Digite o nome da editora que está buscando e
     tecle enter."); break;
73              case 2: System.out.println("Digite o nome da cidade onde está a editora
    que está buscando e tecle enter."); break;
74              case 3: System.out.println("Digite o nome da UF onde fica a editora que
    está buscando e tecle enter."); break;
75            }
76            chave = s.nextLine();
77            switch (item) {
78              case 1: busca.nome  =chave; resultado = b.findEditora(busca); break;
79              case 2: busca.cidade=chave; resultado = b.findEditora(busca); break;
80              case 3: busca.estado=chave; resultado = b.findEditora(busca); break;
81            }
82          }
83          switch(item) {
84            case 1:
85            case 2:
86            case 3: break;
87            case 4: resultado = b.nextEditora(); break;
88            case 5: e.deleteEditora(ultimoResultado); ultimoResultado = null; break;
    // AQUI
89            case 9: novamente = false; break;
90            default: System.out.println("\nOpção não prevista!\n"); break;
91          }
92          if( resultado != null ) {
93            b.showEditora(resultado);
94            ultimoResultado = resultado;
95            resultado = null;
96          } else {
97            if(item != 9)
98              System.out.println("Nenhum resultado para mostrar.");
99          }
100       } while (novamente);
101       b.close();
102     } catch (SQLException e) {
103       System.err.println("ERRO:SQLException: algo não saiu bem ao falar com o banco
    de dados.");
104       e.printStackTrace();
105       return;
106     } catch (Exception e) {
107       System.err.println("ERRO:Exception: não foi possível carregar o driver JDBC do
     HSQLDB, talvez?");
108       e.printStackTrace();
109       return;
110     }
111   }
112 }
```

Não temos muito o que explicar aqui. Temos? Acho que devo a você alguns comentários sobre a linha 20, onde temos um novo comando SQL:

```
DELETE FROM TABELA WHERE EXPRESSAO
```

O comando `delete` vai eliminar todas as linhas da tabela `TABELA` onde a expressão `EXPRESSAO` dada for verdadeira. No nosso caso, estamos procurando por pares `COLUNA=VALOR` para as colunas `NOME`, `CIDADE` e `UF`. Temos *IN parameters* da mesma maneira que os `PreparedStatement`s dos capítulos anteriores, então, imagino que você já está mestre nisso.

Este programa também precisa do código que implementa a classe `Busca` já implementada. Buscamos uma `Editora` e a armazenamos no atributo `ultimoResultado` e de lá chamamos o método que faz a exclusão.

O que você deve ver

```
Conexãorealizada com sucesso!
Conexão realizada com sucesso!
Conexão realizada com sucesso!
O que você quer fazer? Digite o número correspondente.
1 - Buscar editora pelo nome
2 - Buscar editora pela cidade
3 - Buscar editora pelo UF
9 - Sair
3
Digite o nome da UF onde fica a editora que está buscando e tecle enter.
DF
Editora:Independente mesmo
Cidade :Engenho das Lages
Estado :DF

O que você quer fazer? Digite o número correspondente.
1 - Buscar editora pelo nome
2 - Buscar editora pela cidade
3 - Buscar editora pelo UF
4 - Conexão resultado
5 - Exclui editora do último resultado
9 - Sair
5
Registro excluído!
Nenhum resultado para mostrar.
O que você quer fazer? Digite o número correspondente.
1 - Buscar editora pelo nome
2 - Buscar editora pela cidade
3 - Buscar editora pelo UF
4 - Conexão resultado
9 - Sair
9
```

Agora não é mesmo surpresa, carregue o gerenciador de banco de dados e execute um "'SELECT * FROM EDITORA"' para ver que alguns dados realmente desapareceram. Vamos fazer alguns segundos de silêncio por esses dados que foram parar no céu dos bits.

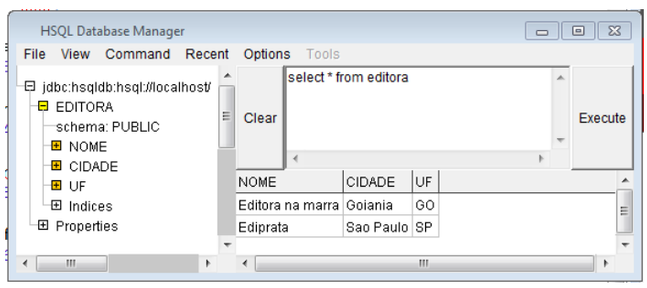

Figura 40.1: Banco de dados após a exclusão

Desafios para estudo

1. Volte no programa que insere, use a opção de inserir um registro constante e faça algumas cópias dele. Carregue o HSQL Database Manager e veja como está a tabela. Execute novamente o programa de Exclusão, reexecute o SELECT no HSQL Database Manager e veja o que aconteceu.

2. Ative as classes `Inclui` e `Altera`, acrescente as opções correspondentes no menu e voilà! Temos um CRUD completo em modo texto!