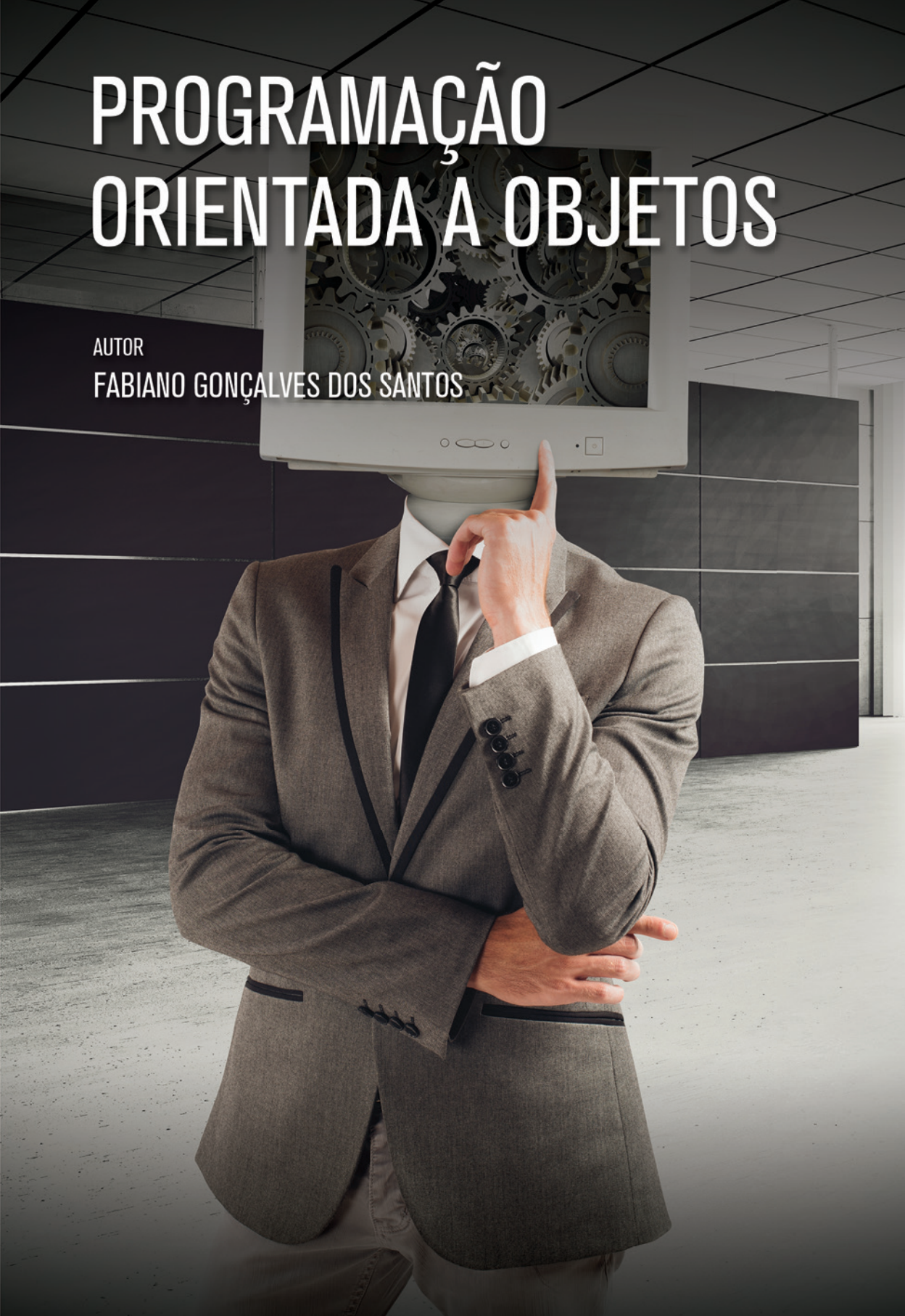
PROGRAMAÇÃO ORIENTADA A OBJETOS

AUTOR
FABIANO GONÇALVES DOS SANTOS



PROGRAMAÇÃO ORIENTADA A OBJETOS

AUTOR
FABIANO GONÇALVES DOS SANTOS

1ª EDIÇÃO
SESES
RIO DE JANEIRO 2017

Conselho editorial ROBERTO PAES E LUCIANA VARGA

Autor do original FABIANO GONÇALVES DOS SANTOS

Projeto editorial ROBERTO PAES

Coordenação de produção LUCIANA VARGA, PAULA R. DE A. MACHADO E ALINE KARINA RABELLO

Projeto gráfico PAULO VITOR BASTOS

Diagramação BFS MEDIA

Revisão linguística BFS MEDIA

Revisão de conteúdo FRANCISCO ALVES CARNEIRO

Imagem de capa ALPHASPIRIT | SHUTTERSTOCK.COM

Dados Internacionais de Catalogação na Publicação (CIP)

S237P SANTOS, FABIANO GONÇALVES DOS
Programação orientada a objetos / Fabiano Gonçalves dos Santos.
Rio de Janeiro: SESES, 2017.
144 P: IL.

ISBN: 978-85-5548-470-4

1. INTERFACES GRÁFICAS. 2. TRATAMENTO DE EVENTOS. 3. COLEÇÕES.
4. PROGRAMAÇÃO. I. SESES. II. ESTÁCIO.

CDD 005.1

Diretoria de Ensino — Fábrica de Conhecimento
Rua do Bispo, 83, bloco F, Campus João Uchôa
Rio Comprido — Rio de Janeiro — RJ — CEP 20261-063

Sumário

Prefácio

Prezados(as) alunos(as),

A orientação a objetos é um tema muito presente nas linguagens de programação modernas. Java, Python, C#, PHP e outras possuem suporte para este paradigma e tornam estas linguagens poderosas para serem aplicadas em vários tipos de plataformas e sistemas operacionais. Java por exemplo pode ser usada em sistemas *desktop*, em dispositivos móveis, programas embarcados e na *web*. E a orientação a objetos permite esta flexibilidade.

Neste livro vamos estudar a linguagem Java de um modo bem aplicado: aprender a criar interfaces gráficas usando uma biblioteca já consagrada pela linguagem: a biblioteca *Swing*. Esta biblioteca faz parte da *Java Foundation Classes* (JFC) a qual é um conjunto de classes que agrupam componentes para criar interfaces gráficas, também chamadas de *Graphical User Interfaces* (GUI) que têm como características principais possuir o mesmo comportamento e aparência entre plataformas e sistemas operacionais diferentes.

Vários programas existentes no mercado e em empresas, inclusive no governo, usam componentes *Swing*. Embora estejamos na era da internet, muitos aplicativos ainda precisam ser desenvolvidos para desktop como é o exemplo de sistemas empresariais como os ERP (*Enterprise Resource Planning*). Além disso, a demanda pela manutenção de sistemas já desenvolvidos também é grande. A *Oracle*, empresa que mantém ativa a tecnologia Java, possui todos os seus programas para desktop desenvolvidos em Java, como muitas empresas usam o banco de dados da *Oracle*, a programação para *desktop* é muito aplicada nessas empresas.

E para terminar, os conhecimentos que você vai adquirir aqui vão servir como base e fundamento para outras tecnologias que você tem interesse em aprender e trabalhar. Apesar da linguagem ser diferente, a estrutura dos mecanismos dos componentes gráficos, eventos e outros recursos são bem parecidos e você não terá dificuldade em comparar o que será visto aqui com outras tecnologias e linguagens.

Bons estudos!

1

Criação de interfaces gráficas usando as JFC/*Swing*

Criação de interfaces gráficas usando as JFC/*Swing*

O primeiro capítulo do nosso livro vai apresentar os elementos básicos e necessários para a criação de interfaces gráficas em Java usando a biblioteca JFC/*Swing*.

A criação de interfaces gráficas é uma área muito importante no desenvolvimento de *software* para usuários em geral. Um *software* com uma aparência adequada e agradável, desenvolvido de acordo com as recomendações estabelecidas nos padrões de interatividade, e apresentação, certamente terá uma aceitabilidade maior pelo usuário.

A tecnologia Java, desde a época da *Sun* (a empresa responsável pela "criação" da linguagem) possui elementos que ajudam na criação de interfaces bonitas e compatíveis com os principais sistemas operacionais usados no mundo como o Windows, Mac OS e Unix de uma maneira geral. Com a evolução da tecnologia, a Sun melhorou as bibliotecas gráficas iniciais e mesmo após a compra pela *Oracle*, percebemos que a preocupação com os elementos gráficos continuou e foram aprimorados.

A biblioteca de componentes gráficos originais do Java é chamada de *Abstract Window Toolkit* (AWT) e continha alguns elementos que encontramos no Microsoft Windows (por exemplo) como janelas, botões, barras de rolagem etc. Porém havia limitações as quais vamos tratar ao longo deste livro.

Nosso foco é estudar a biblioteca JFC/*Swing*. Temos certeza que você vai gostar e usar muito essa biblioteca no desenvolvimento de aplicações *desktop* em Java.

Vamos começar?

OBJETIVOS

Este capítulo tem os seguintes objetivos:

- Apresentar a biblioteca JFC/*Swing*
- Demonstrar o uso de gerenciadores de *layout*:
  - *Flow Layout*
  - *Grid Layout*
  - *Border Layout*
  - *GridBagLayout*

Hierarquia de classes

A linguagem Java é orientada a objetos e como qualquer outra linguagem com esta característica, possui classes predefinidas, as quais possuem uma hierarquia.

A hierarquia é obtida por meio de derivações das classes principais chamada Herança. A herança aproveita a estrutura das classes pais (ou superclasses) e transfere para os filhos (ou subclasses) e estas podem acrescentar novas funcionalidades e características.

Os componentes *Swing* são derivados da hierarquia mostrada na figura 1.1. A superclasse principal desta hierarquia é a classe Object, verifique.

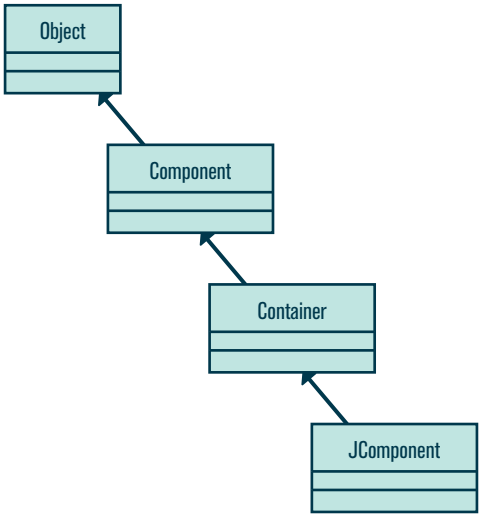


Figura 1.1 – Hierarquia de classes dos componentes *Swing* (ORACLE CORPORATION, 2015).

A classe *Component* deriva de *Object* e possui muitos atributos e métodos que são encontrados nos componentes gráficos. Botões, *checkboxes, labels* e caixas de texto são exemplos de componentes encontrados nesta classe. Como vemos na figura 1.2, encontramos na classe Component vários elementos que conhecemos e de fato a maioria dos elementos gráficos que existem nas aplicações modernas pertencem direta ou indiretamente a esta classe.

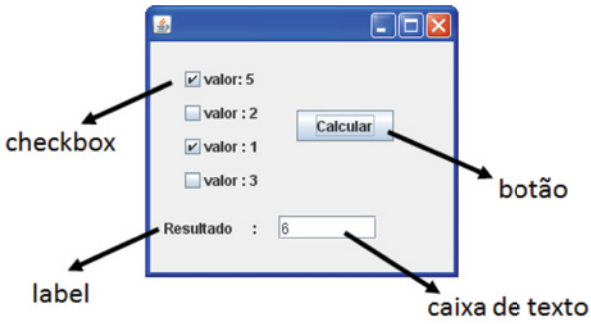


Figura 1.2 – Exemplos de componentes *Swing*.

A classe *Container* deriva de *Component* como vemos na figura 1.1.

Um dos elementos mais usados desta classe é o Frame que é a janela que conhecemos, como mostrada na figura 1.2. Em um Frame é possível inserir vários componentes e montar uma tela para o usuário, ou seja, qualquer objeto da classe *Container* pode ser usado para juntar e organizar outros objetos da classe *Component* para exibir uma *interface* gráfica para o usuário. Um *Container* é como se fosse um quadro de recados: cada recado que penduramos ou criamos seria um objeto da classe *Component*, conforme demonstra a figura 1.3.

Figura 1.3 – Comparação de um *Container* com um quadro de recados.

A classe *JComponent* possui todos os componentes *Swing* de fato, exceto os *Container*. Os objetos desta classe possuem algumas características de acordo com a documentação oficial, entre elas:

APARÊNCIA E COMPORTAMENTO PLUGÁVEIS	Permite que o chamado *look* and *feel* (aparência e comportamento) possam ser personalizados em cada componente.
TRATAMENTO DE TECLAS DE ATALHO	Desta forma o usuário consegue usar o teclado para acessar os componentes sem o uso do *mouse*.
SUPORTE ÀS DICAS DO COMPONENTE	Você já deve ter visto que em alguns programas, quando você deixa o mouse sobre um elemento, aparece uma pequena janela contendo uma breve descrição do componente. Essa descrição é chamada de *tooltip*.
SUPORTE À ACESSIBILIDADE	Para pessoas que possuem algum tipo de deficiência e outras.

CONEXÃO

Quer saber mais sobre estas classes? A melhor forma de realmente entender o seu funcionamento e características é consultar a própria documentação da *Oracle*. Porém são páginas em inglês. Não tenha medo de ler em inglês, se você não está acostumado, conforme ler este tipo de texto você logo se acostumará.

A documentação pode ser encontrada em: <https://docs.oracle.com/javase/7/docs/api/javax/swing/JComponent.html>.

Modelos de desenvolvimento de *interfaces* gráficas

Desenvolvimento do *Swing* para GUI

Atualmente a internet é o grande palco para as aplicações. Lendo algumas revistas e notícias percebemos a quantidade de soluções como *softwares* ERP, agendas, aplicativos e outros que rodam na *web*, bastando que o usuário tenha um navegador.

Porém, o uso em aplicações *desktop* ainda é atual e muito necessário. Nem todas as aplicações devem rodar na internet por vários tipos de questões: segurança, recursos, velocidade são apenas algumas destas justificativas.

A linguagem Java ainda é a mais usada segundo o *ranking* da *Tiobe* e sendo assim, muitas aplicações são desenvolvidas diariamente usando esta linguagem.

Como já dissemos, a linguagem Java usa a biblioteca JFC/*Swing* para permitir a criação de *interfaces* para *desktop*. Para isso, o programador além de conhecer a linguagem Java e orientação a objetos, vai precisar conhecer também os principais componentes que formam a biblioteca *Swing* (botões, caixas de texto, marcadores etc.).

Falando em componentes, toda interface com o usuário considera basicamente três aspectos principais:

ELEMENTOS DE *INTERFACE*	É aquilo que o usuário vê e interage.
LAYOUTS	Definem como os elementos de interface serão organizados na tela e criam o "*look and feel*" final para o usuário
COMPORTAMENTO	São os eventos que ocorrem quando o usuário interage com os elementos de *interface*.

CONCEITO

Look and Feel: Quando você estiver estudando a JFC/*Swing*, você certamente vai encontrar o termo "*look and feel*". Este termo traduzindo diretamente é "aparência e sentimento", mas tem mais relação com a aparência e comportamento de uma aplicação gráfica. Cada plataforma (Windows, Mac Os, Linux, etc) tem uma forma própria de interação com o

usuário e a linguagem Java tenta obedecer esta característica usando as mesmas formas, cores, *layout* e fontes tipográficas parecidas com a plataforma na qual o programa está sendo executado, além de respeitar o comportamento dinâmico das caixas, áreas de texto, botões e outros elementos da plataforma.

A seguir vamos listar os principais componentes usados da biblioteca JFC/*Swing*.

COMPONENTE	DESCRIÇÃO
JLABEL	Serve para colocar um texto em um *container*.
JBUTTON	É o famoso e clássico botão que conhecemos.
JCOLORCHOOSER	Mostra um painel para o usuário escolher uma cor e manipulá-la.
JCHECKBOX	É um componente gráfico que pode somente assumir dois valores: verdadeiro ou falso.
JRADIOBUTTON	É um componente gráfico que é parecido com o *JCheckBox*, porém trabalha em grupo com outros componentes.
JLIST	Mostra uma lista de opções para o usuário com rolagem.
JCOMBOBOX	Mostra uma lista de opções para o usuário por meio de um menu.
JTEXTFIELD	É uma caixa de texto, podendo inserir texto.
JPASSWORDFIELD	É semelhante ao anterior, porém específico para receber senhas.
JTEXTAREA	É outro componente para entrada de texto, porém possibilita várias linhas.
IMAGEICON	É um componente que armazena uma imagem para ser usada em várias situações.
JSCROLLBAR	Mostra uma barra de rolagem para o usuário escolher entre uma faixa de valores.

COMPONENTE	DESCRIÇÃO
JOPTIONPANE	Mostra uma caixa de diálogo padrão com uma mensagem para informação ou entrada de dados.
JFILECHOOSER	Mostra uma janela de diálogo para o usuário escolher um arquivo dentro da estrutura de pastas do sistema operacional.
JPROGRESSBAR	Mostra uma barra de progresso, indicando a porcentagem de conclusão de um determinado processo.
JSLIDER	O slider permite que o usuário selecione graficamente um valor deslizando um ponteiro sobre uma barra graduada.
JSPINNER	É um campo de entrada de linha simples que permite ao usuário selecionar um número ou um valor de objeto de uma sequência ordenada.

Tabela 1.1 – Principais componentes da biblioteca JFC/*Swing*.

? CURIOSIDADE

Na instalação do Java, na versão completa do JDK, existe um diretório com uma aplicação que mostra todos os componentes *Swing*. Procure dentro do diretório \demo\jfc\SwingSet2

Gerenciadores de *layout*

Em aplicações *desktop* é muito comum que o usuário maximize, minimize ou altere o tamanho das janelas do aplicativo que ele está usando. Dependendo do conteúdo da janela, os componentes internos podem se auto organizar ou não. Experimente fazer isso com um programa qualquer, como o Microsoft Word: se você redimensionar a janela você vai perceber que os componentes internos serão cortados ou readequados ao tamanho conforme você diminui o tamanho da tela.

Mas e em Java? Dá para fazer isso? Sim! Por meio dos gerenciadores de *layout*. A vantagem de se usar gerenciadores de *layout* é que evita os programadores terem que se preocupar com a determinação da posição e tamanho dos componentes e assim a sua preocupação terá o foco na aparência e no comportamento de cada componente escolhido.

Os gerenciadores de *layout* são objetos da classe *LayoutManager* (do pacote java.awt), ou mais especificamente, eles implementam a interface *LayoutManager*. Na classe Container existe um método chamado setLayout o qual aceita um objeto que implementa a interface *LayoutManager* como um argumento. Vamos exemplificar isto adiante para maior esclarecimento.

Podemos organizar os componentes em uma janela de 3 formas diferentes:

POSICIONAMENTO ABSOLUTO	Basicamente esta forma é criar a interface gráfica "na mão", ou seja, você vai posicionar os componentes em um *Container* (por exemplo, uma janela) baseado na sua real posição dentro do *Container* em relação ao seu canto superior esquerdo. Além disso, você vai precisar dimensionar o tamanho do componente manualmente. De fato, é a forma de *layout* mais trabalhosa.
GERENCIADOR DE LAYOUT	Esta forma é bem mais prática. Usar um gerenciador de *layout* envolve também perder um pouco do controle, posicionamento e tamanho dos componentes.
PROGRAMAÇÃO VISUAL EM ALGUMA IDE	Existem algumas IDE como o *Netbeans*, que suportam a programação visual de interfaces gráficas. O uso dos recursos de arrastar e soltar os componentes dentro de um *Container* facilita bastante o trabalho do programador, pois ele consegue visualizar rapidamente como ficará o resultado final da interface. Portanto o tamanho dos componentes assim como o seu posicionamento, é feito de forma visual, o que torna esta forma a mais produtiva em relação às apresentadas anteriormente.

Flow *Layout*

Este é sem dúvida o gerenciador de layout mais simples. Os elementos da *interface* são colocados em um fluxo (por isso o nome "*flow*") da esquerda para direita ordenados pela forma como são colocados no *container*. Se houver mais componentes do que a largura que o *container* permitir, os próximos componentes são colocados na linha abaixo ainda da esquerda para a direita.

Melhor mostrar um exemplo, certo? Observe a figura 1.4:

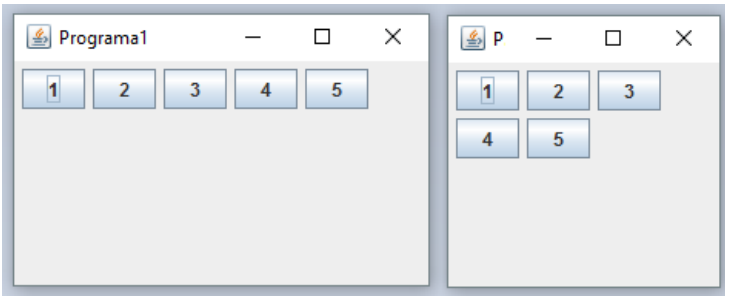


Figura 1.4 – Exemplo de *FlowLayout.*

Criamos um *frame* e adicionamos 5 botões a ele. Os botões quando clicados não vão fazer coisa alguma, estão ali apenas para demonstrar como o *layout* funciona. Neste caso, o container é o próprio frame. Conforme os botões são adicionados no frame, eles se posicionam um ao lado do outro a partir da esquerda. Quando redimensionamos a janela, perceba que os botões são rearranjados automaticamente para obedecer a regra do *layout:* sempre alinhado à esquerda e abaixo, neste caso. Veja o código relativo a este exemplo.

```
1   import java.awt.*;
2   import javax.swing.*;
3
4   public class Programa1{
5       JFrame f;
6
7       Programa1(){
8           f=new JFrame("Programa1");
9
10          JButton b1=new JButton("1");
11          JButton b2=new JButton("2");
12          JButton b3=new JButton("3");
13          JButton b4=new JButton("4");
14          JButton b5=new JButton("5");
15
16          f.add(b1);f.add(b2);f.add(b3);f.add(b4);f.add(b5);
17
18          f.setLayout(new FlowLayout(FlowLayout.LEFT));
19          f.setSize(300,300);
```

```
20          f.setVisible(true);
21      }
22
23      public static void main(String[] args) {
24          new Programa1();
25      }
26  }
```

Listagem 1 – *FlowLayout.*

! ATENÇÃO

Entendendo o código.

LINHA	O QUE FAZ
1 E 2	Importam os pacotes que serão usados no programa
4	Define a classe Programa1
5	Define um objeto f da classe *Frame*. Este será o container do exemplo. Lembre-se: container é o lugar onde vamos "pendurar" nossos componentes.
7	Define o construtor da classe Programa1
8	Instancia o frame f com o título "Programa 1"
10 A 14	Criam e instanciam 5 objetos da classe *JButton*. São os 5 botões presentes no exemplo. Em Java, o botão é um objeto da classe *JButton*. Eles podem ser instanciados de várias formas. A forma que usamos é a mais comum e usada.
16	Veja o que fizemos aqui. Literalmente adicionamos, com o método *add()*, os 5 botões no *frame*. Perceba a ordem na qual foram inseridos. Esta ordem é a que será exibida no *frame*.
18	Nesta linha atribuímos o **FlowLayout** para o frame. Observe que usamos a constante **LEFT** para definir que os componentes serão colocados a partir da esquerda. Poderíamos ter usado também as constantes *CENTER* e *RIGHT*, que posicionam os componentes a partir do centro ou da direita, respectivamente.

19 E 20	Nestas linhas o frame é dimensionado (linha 19) e mostrado na tela (linha 20)
23 A 25	No método principal, criamos um objeto da classe Programa1, o qual ao ser criado, chamará o construtor definido na linha 7.

GridLayout

Com o *GridLayout* é possível arranjar os elementos em um *grid* retangular. Cada componente será mostrado em um retângulo no qual terá a mesma largura e altura dos demais. Os componentes são adicionados, lembre-se do método *add*(), partir do canto esquerdo superior e continua com direção à direita e para a próxima linha.

Assim como o *FlowLayout*, o *GridLayout* possui algumas formas de instanciação. Vamos usar a mais comum. Observe a listagem 2 e a figura 1.5.

```
1       import java.awt.*;
2       import javax.swing.*;
3
4       public class Programa2{
5           JFrame f;
6
7           Programa2(){
8               f=new JFrame("Programa2");
9
10              JButton b1=new JButton("1");
11              JButton b2=new JButton("2");
12              JButton b3=new JButton("3");
13              JButton b4=new JButton("4");
14              JButton b5=new JButton("5");
15              JButton b6=new JButton("6");
16              JButton b7=new JButton("7");
17              JButton b8=new JButton("8");
18              JButton b9=new JButton("9");
```

```
19
20              f.add(b1);f.add(b2);f.add(b3);
21              f.add(b4);f.add(b5);f.add(b6);
22              f.add(b7);f.add(b8);f.add(b9);
23
24              f.setLayout(new GridLayout(3,3));
25              f.setSize(300,300);
26              f.setVisible(true);
27          }
28
29          public static void main(String[] args) {
30              new Programa2();
31          }
32      }
```

Listagem 2 – Programa 2.

! ATENÇÃO

Entendendo o código.

Veja que o Programa 2 é bem parecido com o Programa1. Porém existem pequenas diferenças:

LINHAS 10-18:	Criamos os botões para demonstrar a disposição do *grid*. Instanciamos 9 objetos para poder caber em um *grid* de 3x3.
20-23:	Adicionamos os botões no frame instanciado na linha 8.
24:	Nesta linha instanciamos e atribuímos um *GridLayout* para o frame. Usamos o construtor do *GridLayout* no qual o tamanho do *grid* já é passado como parâmetro. Como já dissemos, o *GridLayout* possui outras formas de ser instanciado e optamos pela mais comum e útil.

As demais linhas do Programa 2 são parecidas com o Programa1.

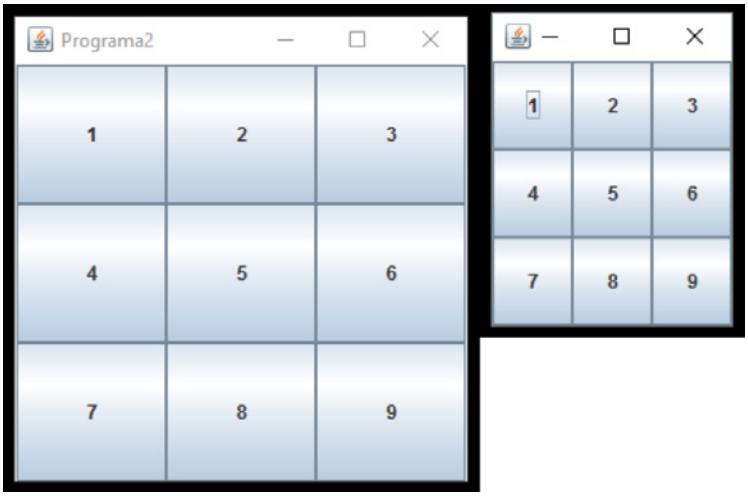


Figura 1.5 – GridLayout.

A figura 1.5 mostra o resultado da execução do programa. O frame da esquerda é o frame original, com tamanho 300 x 300 e o da direita é uma demonstração do que ocorre quando o frame é redimensionado. Observe que como usamos o grid 3 x 3, o redimensionamento não afeta esta distribuição.

Border Layout

Os dois *layouts* apresentados são os mais simples encontrados na biblioteca *Swing*. Podemos também combinar os componentes visuais para criar interfaces mais complexas e mais interessantes. Além disso, também podemos combinar os tipos de *layouts* para criar janelas. Não se esqueça que a biblioteca *Swing* possui outros tipos de gerenciadores de *layout*, os quais podem ser usados.

Um deles é o *BorderLayout*. O *BorderLayout* serve para organizar os componentes em 5 regiões diferentes de um container que correspondem às bordas em um container: Norte (*NORTH*) representa a borda superior, Sul (*SOUTH*) representa a inferior, Leste (*EAST*) representa a borda direita, Oeste (*WEST*) representa a borda esquerda e Central (*CENTER*) para preencher a parte central do container. Veja a figura 1.6. A listagem 3 mostra o código que gerou a figura.

O *BorderLayout* tem 2 construtores principais: um sem argumento e outro no qual podemos colocar espaços entre as bordas.

Figura 1.6 – BorderLayout.

```
1   import java.awt.*;
2   import javax.swing.*;
3   import javax.swing.JFrame;
4   import javax.swing.JButton;
5
6   public class Programa3{
7       JFrame f;
8
9
10      Programa3(){
11          f=new JFrame("Programa3");
12
13          JButton b1=new JButton("Norte");
14          JButton b2=new JButton("Sul");
15          JButton b3=new JButton("Leste");
16          JButton b4=new JButton("Oeste");
17          JButton b5=new JButton("Centro");
18
19          f.add(b1,BorderLayout.NORTH);
20          f.add(b2,BorderLayout.SOUTH);
21          f.add(b3,BorderLayout.EAST);
```

```
22          f.add(b4,BorderLayout.WEST);
23          f.add(b5,BorderLayout.CENTER);
24
25          f.setSize(300,300);
26          f.setVisible(true);
27      }
28
29      public static void main(String[] args) {
30          new Programa3();
31      }
32  }
```

Listagem 3 – *BorderLayout.*

! ATENÇÃO

Entendendo o código.

LINHA	O QUE FAZ
13-18:	Instancia os botões
19-23:	Atribui a cada botão uma borda diferente e adiciona o botão ao *frame*. Veja o uso das constantes *NORTH*, *SOUTH*, *EAST*, *WEST* e *CENTER*
30:	Instancia um novo objeto Programa3 e executa o construtor

GridBagLayout

O *GridBagLayout* (*Grid* = Grade, *Bag* = Sacola) é um dos gerenciadores de *layout* mais flexíveis e também mais complexos. Ele posiciona os componentes em um *grid* de linhas e colunas permitindo que o tamanho das linhas ou das colunas seja diferente. Essencialmente, este tipo de *layout* coloca os componentes em retângulos em um *grid* e usa os tamanhos preferenciais de cada componente para determinar o tamanho de cada célula do *grid*.

Figura 1.7 – *GridBagLayout.*

A figura 1.7 é um exemplo do que pode ser feito com o *GridBagLayout*. É claro que é apenas um exemplo para demonstração e vamos usar componentes simples, mas você pode imaginar que é possível criar *layouts* bem variados com este gerenciador.

O seu funcionamento é bem parecido com os spans existentes em tabelas na linguagem HTML. Observe que na figura da direita temos um grid 3 x 3 (3 linhas por 3 colunas) onde as células são combinadas para formar o efeito desejado como mostrado na figura da esquerda. É interessante que antes de começar a escrever o programa em Java seja feito um protótipo em papel. Você pode pegar um papel quadriculado e desenhar literalmente a interface que deseja para ter melhor resultado quando estiver programando.

Antes de estudar a listagem do programa, precisamos apontar algumas observações quanto a este gerenciador de *layout*.

Para fazer o *layout* neste gerenciador, são aplicados três níveis de controle para formar o desenho visual no *container*:

- Os tamanhos das linhas e colunas e a forma delas se redimensionarem quando o container é modificado;
- Determinar a(s) célula(s) para adicionar o componente;
- Determinar como o componente é redimensionado .

Vamos usar uma classe chamada *GridBagConstraints* que especifica como um componente é inserido em um *GridBagLayout*. Usando esta classe, cada componente terá suas restrições definidas no objeto *GridBagConstraints*. Para atribuir as restrições no objeto *GridBagConstraint*, usamos o método *add* (componente, restrições). Vamos ver um exemplo na próxima listagem.

Neste gerenciador de *layout* existem diversos atributos que são responsáveis por realizar algumas tarefas como por exemplo:

- Controlar linhas e colunas;
- Controlar o tamanho da célula de um componente;
- Posicionar cada componente dentro de uma região.

O controle de linhas e colunas é feito por meio das seguintes restrições:

GRIDX E GRIDY	Determinam a linha e a coluna para posicionar o componente na grade.
GRIDWIDTH E GRIDHEIGHT	Especifica o número de colunas e linhas que o componente terá.
WEIGHTX E WEIGHTY	Definem o peso dos componentes. O peso é uma característica que impacta bastante na aparência dos componentes. Os pesos são usados para determinar como distribuir o espaço entre as colunas (*weightx*) e linhas (*weighty*) e para controlar o comportamento do redimensionamento. Os componentes de pesos maiores ocupam mais espaço adicional que os componentes de peso menor. Quando o valor for 0 (zero) o componente mantém o seu tamanho original. Os valores de peso especificado representam uma proporção de todo o espaço relativo a uma linha ou coluna. Os componentes devem receber valores de peso positivos diferente de zero senão os componentes ficam da mesma forma no meio do container quando o mesmo for redimensionado. Vamos ver isso no exemplo.

Você já deve ter percebido que este gerenciador é mais complexo mesmo, né? Mas é apenas questão de entender como ele funciona, pois, os comandos e demais características não são tão complicados. E é claro, você entenderá melhor quando começar a usá-lo com mais frequência.

Ainda temos algumas observações a fazer sobre este gerenciador:

- O número de linhas e colunas é o maior número de células usadas relativo a cada linha e coluna;
- O tamanho padrão de cada linha e coluna é o tamanho de seu componente mais alto ou mais largo, respectivamente;
- O alongamento da linha e coluna é controlado pelo peso (*weightx* em coluna e *weighty* em linha).

As constantes usadas são:

- *NONE* – componente não crescerá em nenhuma direção;
- *HORIZONTAL* – Indica que o componente cresce horizontalmente;
- *VERTICAL* – Indica que o componente cresce verticalmente;
- *BOTH* – o componente cresce em ambas as direções;
- A variável de instância anchor ("âncora") de *GridBagConstraints* determina a localização do componente, quando o mesmo não preenche toda a área;
- *NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWAST, WEST, NORTHWEAST* ou *CENTER*. O valor default é *CENTER;*
- A variável de instância *fill* de *GridBagConstraints* determina quanto da área alocada é preenchida pelo componente.

Melhor mostrar no código, não é? Vamos lá!

```
1   import java.awt.*;
2   import javax.swing.JFrame;
3   import javax.swing.JTextArea;
4   import javax.swing.JTextField;
5   import javax.swing.JButton;
6
7   public class Programa5 extends JFrame {
8       private GridBagLayout layout;
9       private GridBagConstraints constraints;
10      private JFrame f;
11
12      public Programa5() {
13          f = new JFrame( "Programa 5" );
14
15          layout = new GridBagLayout();
16          f.setLayout( layout );
17          constraints = new GridBagConstraints();
18
19          JTextArea textArea1 = new JTextArea("TextArea1", 5, 10 );
20          JTextArea textArea2 = new JTextArea("TextArea2", 2, 2 );
21          JTextField textField = new JTextField( "TextField" );
22          JButton botao1 = new JButton("Botao 1" );
23          JButton botao2 = new JButton("Botao 2" );
24          JButton botao3 = new JButton("Botao 3" );
25          JButton botao4 = new JButton("Botao 4");
```

```
26
27          constraints.fill = GridBagConstraints.BOTH;
28          adiciona(textArea1, 0, 0, 1, 3 );
29          constraints.fill = GridBagConstraints.HORIZONTAL;
30          adiciona(botao1, 0, 1, 2, 1 );
31          adiciona(botao4, 2, 1, 2, 1 );
32          constraints.weightx = 1000;
33          constraints.weighty = 1;
34          constraints.fill = GridBagConstraints.BOTH;
35          adiciona(botao2, 1, 1, 1, 1 );
36          constraints.weightx = 0;
37          constraints.weighty = 0;
38          adiciona(botao3, 1, 2, 1, 1 );
39          adiciona(textField, 3, 0, 2, 1 );
40          adiciona(textArea2, 3, 2, 1, 1 );
41
42          f.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
43          f.setSize( 300, 150 );
44          f.setVisible( true );
45      }
46
47      private void adiciona( Component component,int row, int column, int width,
int height ){
48          constraints.gridx = column;
49          constraints.gridy = row;
50          constraints.gridwidth = width;
51          constraints.gridheight = height;
52          layout.setConstraints( component, constraints );
53          f.add( component );
54      }
55
56      public static void main( String args[] ) {
57          new Programa5();
58      }
59  }
```

Listagem 4 – *GridBagLayout.*

! ATENÇÃO

Entendendo o código.

LINHA	O QUE ELA FAZ
8-10:	Criação dos objetos da classe: *layout*, constraints e f (observe o tipo de cada um deles).
15-18:	Instanciamos o objeto *layout*, o objeto constraints e atribuímos o *layout* ao *frame.*
19-25:	Nestas linhas instanciamos os componentes gráficos que vamos usar na *interface*. Desta vez vamos usar a textarea e um campo de texto apenas como exemplo. Vamos examinar estes componentes oportunamente. Apenas perceba que eles têm formas de instanciação diferente do *JButton*.
27-40:	Nestas linhas está a parte principal e que faz a "mágica" do *layout*. Veja que criamos um método na classe chamado adiciona o qual faz o trabalho de adicionar um componente ao *container*. Este método foi criado apenas para facilitar o trabalho de inserção dos componentes no *container*. Veja a explicação dele no item referente às linhas 47-54. Observe o uso do objeto constraints e de suas variáveis *fill*, *weightx*, *weighty* nestas linhas.
47-54:	Aqui está a definição do método adiciona. Ele recebe 5 parâmetros: o componente a ser inserido, a linha, a coluna, a largura e a altura de inserção do componente. Dentro do método as restrições do *GridBagLayout* são trabalhadas e usadas para posicionar e definir o comportamento de cada componente. Verifique como definir as *constraints* nas linhas 48 a 54 e como usá-las no *layout* na linha 53.

Combinação de *layouts*

A figura 1.8 mostra uma possibilidade para um *layout* de uma tela típica de cadastro de um sistema. Ela é obtida pela combinação de dois tipos de *layout:* o *GridLayout* e o *BorderLayout*.

A combinação dos *layouts* estudados pode ser feita para gerar vários tipos de telas. Você deve abusar das possibilidades da orientação a objetos para poder criar telas genéricas e a partir de herança, derivar as telas que serão usadas no sistema que você está desenvolvendo. Por enquanto, vamos combinar o *GridLayout* e *BorderLayout* para criar uma tela na qual pode ser bastante reaproveitada.

Observe a listagem 5 e a figura 1.8.

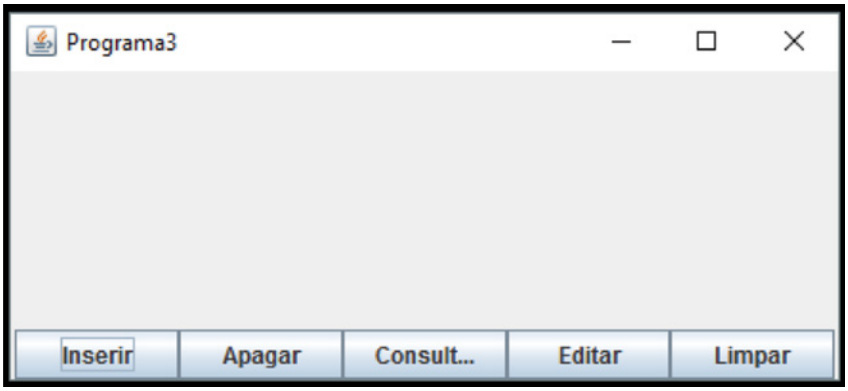


Figura 1.8 – Combinação de *layouts.*

```
1   import java.awt.GridLayout;
2   import java.awt.BorderLayout;
3   import javax.swing.JFrame;
4   import javax.swing.JPanel;
5   import javax.swing.JButton;
6
7   public class Programa3 extends JFrame{
8       JFrame f;
9       JPanel painelBotoes;
10      JButton botoes[];
11
12      Programa3(){
13          f=new JFrame("Programa3");
14
15          botoes = new JButton[5];
16          painelBotoes = new JPanel();
17          painelBotoes.setLayout(new GridLayout(1,botoes.length));
18
19          botoes[0] = new JButton("Inserir");
20          botoes[1]= new JButton("Apagar");
```

```
21          botoes[2]= new JButton("Consultar");
22          botoes[3]= new JButton("Editar");
23          botoes[4]= new JButton("Limpar");
24
25          painelBotoes.add(botoes[0]);
26          painelBotoes.add(botoes[1]);
27          painelBotoes.add(botoes[2]);
28          painelBotoes.add(botoes[3]);
29          painelBotoes.add(botoes[4]);
30
31          f.add(painelBotoes, BorderLayout.SOUTH);
32          f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
33          f.setSize(450,200);
34          f.setVisible(true);
35      }
36
37      public static void main(String[] args) {
38          new Programa3();
39      }
40  }
```

Listagem 5 – *Layouts* combinados.

! ATENÇÃO

Entendendo o código.

Vamos comentar as linhas principais.

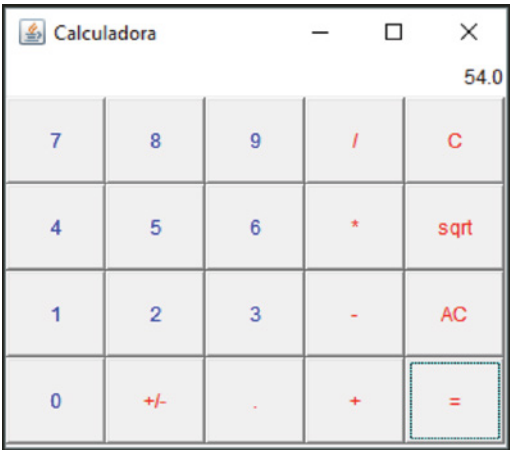
LINHA	O QUE FAZ
1-5	Importações necessárias.
9	Vamos usar um painel para poder posicionar os componentes. O uso de painéis é recomendável porque é uma forma de agrupar os componentes comuns e que possuem o mesmo contexto. No caso vamos criar um painel para os botões.
10	Ao invés de criarmos botão por botão, vamos usar um *array* para poder *indexar* os botões e manipulá-los melhor no código. Isto é uma boa prática em *interfaces* que possuem muitos botões, como em alguns sistemas.

LINHA	O QUE FAZ
12	Método construtor.
15-16	Criação do *array* de objetos da classe *JButton* e do seu painel.
17	Atribuição do *layout grid* para o painel de botões. Perceba que neste caso, o container dos botões é o painel e não o *frame* principal. O *frame* é o container do painel.
19-23	Instanciação dos botões.
25-29	Inserção dos botões no painel.
31	Atribuição do *BorderLayout*. Veja que vamos colocar o painel na borda inferior do *frame* f.
32	Esta linha atribui a saída e finalização correta do programa quando o usuário clica no menu Fechar do *frame*.

ATIVIDADE

Vamos praticar um pouco!

01. Observe a figura a seguir.



Você consegue fazer esta tela em Java? Veja bem: não é preciso deixar a calculadora funcional. Por enquanto o importante é saber fazer igual a figura. Posteriormente a gente deixa ela funcional, que tal?

Você vai precisar, além dos botões de um *JTextField*. Olhe o exemplo do *GridBagLayout* para saber como instanciar um objeto desta classe.

Preste atenção no tipo de *layout* que você vai usar.

REFLEXÃO

Aparentemente pode ser difícil entender tantos componentes novos, suas variáveis de instância e métodos. Porém, a partir do momento que você compreende o seu uso, e com o conhecimento da orientação a objetos, você poderá criar telas reaproveitáveis e poderosas nas suas aplicações para *desktop*. Inclusive para a *web*! Lembre-se que um recurso da linguagem Java é usar os *applets* e estes rodam em páginas na *web*. Que tal criar um *applet* e incorporar um *frame* a ele? Percebeu a relação? Quer dizer, podemos por meio da orientação a objetos e da linguagem Java usar os recursos que estamos aprendendo aqui em aplicações para internet! Fica aí o desafio!

REFERÊNCIAS BIBLIOGRÁFICAS

DEITEL, H. M.; DEITEL, P. J. **Java:** como programar. 6ª. ed. Rio de Janeiro: Bookman, 2005.

HORSTMANN, C. S.; CORNELL, G. **Core Java**. Rio de Janeiro: Pearson Education, 2010.

HUBBARD, J. R. **Programação com Java 2**. Rio de Janeiro: Bookman, 2006.

ORACLE CORPORATION. **The Java Tutorials.** Oracle Java Documentation, 01 mar. 2015. Disponivel em: <https://docs.oracle.com/javase/tutorial>. Acesso em: 1 mar. 2016.

TIOBE SOFTWARE BV. **Tiobe Index**. TIOBE Index for March 2016, 2016. Disponivel em: <http://www.tiobe.com/tiobe_index>. Acesso em: 1 mar. 2016.

2

Criação de *interfaces* gráficas usando as JFC/ *Swing* – Parte 2

Criação de *interfaces* gráficas usando as JFC/*Swing* – Parte 2

No primeiro capítulo conhecemos alguns elementos que organizam as interfaces por meio dos gerenciadores de *layout*. Neste capítulo vamos continuar a abordar alguns elementos visuais que enriquecerão suas telas e *interfaces*.

É importante sempre lembrar a grande qualidade que a linguagem Java oferece em relação à sua capacidade de orientação a objetos e as possibilidades que a herança oferece. Ou seja, tudo que estudamos no capítulo anterior e no próximo pode ser herdado e permitir a criação de *interfaces* padronizadas para os seus sistemas.

É importante também ficar sempre atento às variações que ocorrem nos componentes. Por mais que a biblioteca JFC/*Swing* possua componentes interessantes e práticos de serem usados e reaproveitáveis, a evolução das interfaces não para e novos componentes aparecem e passam a ser requeridos nos programas. Ainda bem que a linguagem Java é flexível a ponto de suportar esta evolução.

Outro ponto a considerar e ser observado é que os componentes que estamos estudando não servem apenas para aplicações *desktop*. Por meio de adaptações e uso intensivo da orientação a objetos podemos aplicar os conhecimentos e técnicas em aplicações para a *web* e até mesmo para outros dispositivos como já ocorre em alguns equipamentos usados no dia a dia como aparelhos de som, som automotivo e outros eletrônicos.

É uma grande possibilidade, basta você se dedicar, estudar e usar a sua criatividade e técnica para poder criar interfaces mais poderosas. Sendo assim, é importante também você procurar por um assunto chamado Engenharia de Usabilidade ou *Interface* Homem Máquina para poder conhecer as contribuições de outras áreas do conhecimento como ergonomia e o quanto elas contribuem para nós, "programadores". Mas isto é assunto para outra disciplina e livro.

Vamos estudar um pouco mais?

OBJETIVOS

Este capítulo tem os seguintes objetivos:

- Explorar a manipulação de aspectos visuais;
- Estudar variações de componentes visuais.

Manipulação de aspectos visuais

O foco principal do capítulo anterior foi abordar as possibilidades de gerenciamento de *layout* oferecidos pela linguagem Java. Conforme vimos nos vários exemplos, usamos alguns componentes como os botões (*JButton*), janelas (*JFrame*), caixas de texto (*JTextField*) e rótulos (*JLabel*). A grande maioria das *interfaces* que encontramos nos programas que usamos não modificam muito o aspecto visual desses componentes, mas existem situações que seria legal modificá-los um pouco.

Observe a figura 2.1. Ela mostra uma das telas de um dos programas mais usados no mundo na área de gestão empresarial. A tela é "carregada" de elementos porque o próprio programa exige tanta informação na tela, porém é agradável de ser usado. Gostaria de destacar os botões "*Ok*", "*Cancel*" e "*You Can Also*": eles não estão na cor padrão dos botões que conhecemos, não é? Além disso, alguns outros componentes no exemplo também sofreram algumas pequenas mudanças. Portanto, como percebemos, as vezes é importante modificar o comportamento padrão dos principais componentes gráficos que usamos.

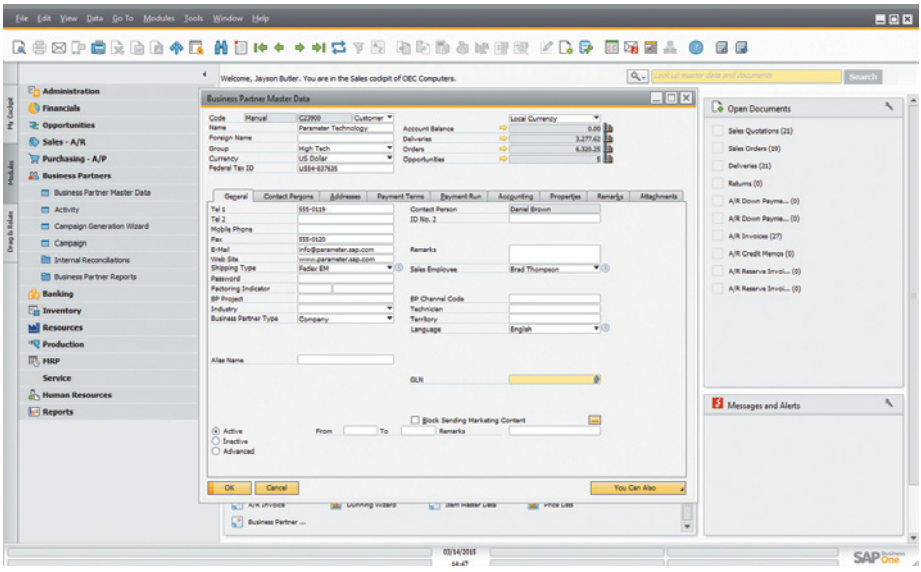


Figura 2.1 – *SAP Business One*. Modificação da cor original do botão. Disponível em: <https://en.wikipedia.org/wiki/File:Screenshot_of_SAP_Business_One_Client_application.png>.

Vamos usar alguns componentes como exemplos. Você vai perceber que a forma que eles trabalham é bem parecida.

Botões (*JButton*)

A classe *JButton* tem algumas formas de ser instanciada.

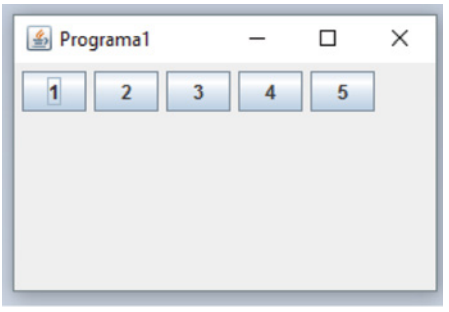
JBUTTON()	Um botão é criado sem texto e sem uma imagem nele (ícone)
JBUTTON(ACTION A)	Cria um botão e o associa a uma ação fornecida
JBUTTON(ICON I)	Cria um botão com uma imagem (ícone)
JBUTTON (STRING S)	Cria um botão com um texto. É a forma mais usada.
JBUTTON (STRING S, ICON I)	Cria um botão com um texto e uma imagem.

Tabela 2.1 – Formas de instanciação de um botão.

No capítulo anterior, no Programa1 criamos um botão, aliás, criamos 5 botões por meio da maneira mais comum: passando um texto como parâmetro para a classe. Observe, lembra do Programa1?

! ATENÇÃO

Lembre-se: Em java, tudo é um objeto. Logo, botões, janelas, *labels* etc., tudo isso são objetos (instâncias) de suas classes!




```
JButton b1=new JButton("1");
JButton b2=new JButton("2");
JButton b3=new JButton("3");
JButton b4=new JButton("4");
JButton b5=new JButton("5");
```

Figura 2.2 – Exemplo de *JButton*.

Como podemos perceber, os botões foram criados com o texto que enviamos como parâmetro para cada uma das classes. Se mandássemos o texto "*Ok*" no JButton b1 do exemplo, o primeiro botão, localizado à esquerda na figura 2.2 teria o texto "*Ok*" dentro dele.

Certo. Este é o padrão. Mas tínhamos comentado que poderíamos mudar o padrão. Como faz isso? API! Nunca se esqueça dela!

CONEXÃO

A API para a classe *JButton* está localizada em: <https://docs.oracle.com/javase/7/docs/api/javax/swing/JButton.html>.

Analisando a API do *JButton*, vamos perceber que temos métodos que são herdados da classe *JComponent*, não esqueça: um *JButton* também é um *JComponent*. E um destes métodos é o *setBackground*. Com ele podemos modificar, como o próprio nome sugere, a cor de fundo de algum objeto da classe *JComponent*.

Veja o uso do método *setBackground* em um exemplo:


```
JButton b1=new JButton("1");
b1.setBackground(Color.RED);
b1.setForeground(Color.YELLOW);
```

Figura 2.3 – Exemplo do método *setBackground*

Como podemos perceber na figura 2.3, usamos a constante pré-definida *Color.RED* para colorir o fundo do botão e a *Color.YELLOW* para colorir o texto.

! ATENÇÃO

Consulte sempre a api para saber quais métodos e propriedades que suas classes possuem. Por meio disso, você poderá alterar vários padrões de aparência e comportamento dos componentes e torná-los mais interessantes.

A api da biblioteca *swing* fica em: <https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html>.

Legendas (*JLabel*)

Esta classe pode exibir texto, uma imagem ou ambos. O conteúdo do *JLabel* é alinhado definindo o alinhamento vertical e horizontal na sua área de exibição. Por padrão, os *labels* são centralizados verticalmente dentro da sua área de exibição. Os *labels* que possuem somente texto ficam alinhados na borda por padrão e as que possuem somente imagem são centralizadas horizontalmente.

Veja a seguir as formas de se instanciar um ***label***.

JLABEL()	Cria uma instância sem imagem e sem texto.
JLABEL (ICON ÍCONE)	Cria um *label* com a imagem enviada por parâmetro.
JLABEL (ICON ÍCONE, INT ALINHAMENTOHOR)	Cria uma instância com a imagem enviada por parâmetro e com o alinhamento especificado.
JLABEL (STRING TEXTO)	Cria um *label* com o texto enviado por parâmetro.
JLABEL (STRING TEXTO,*ICON ÍCONE,INT ALINHAMENTO HOR)*	Cria um label com o texto, imagem e alinhamento enviados por parâmetro.
JLABEL(STRING TEXTO, *INT ALINHAMENTOHOR)*	Cria uma instância com o texto e alinhamento horizontal especificado

Tabela 2.2 – Construtores da classe *JLabel*.

Nosso exemplo vai criar uma instância para cada um dos construtores da tabela *label* na tela. Veja:

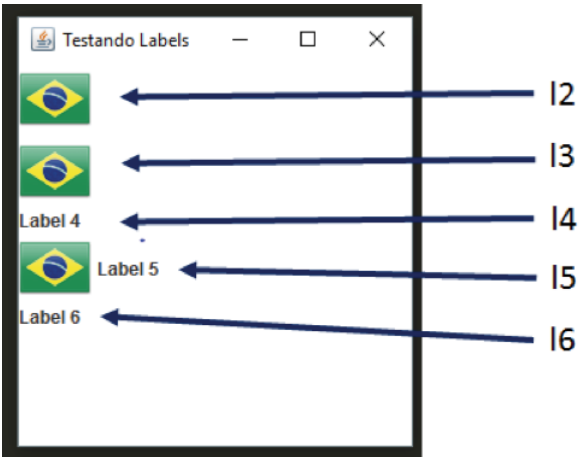


Figura 2.4 – Exemplo com JLabels.

O código principal para criar os *labels* mostrados na figura 2.4 é o seguinte:

```
Icon brasil=new ImageIcon("brasil.png");
JLabel l1=new JLabel();
JLabel l2=new JLabel(brasil);
JLabel l3=new JLabel(brasil,JLabel.RIGHT);
JLabel l4=new JLabel("Label 4");
JLabel l5=new JLabel("Label 5",brasil,JLabel.RIGHT);
JLabel l6=new JLabel("Label 6", JLabel.CENTER);
```

Tabela 2.3 – Listagem 1: Instanciação de *Jlabels.*

Criamos 6 *labels* chamados l1, l2, l3, l4, l5 e l6. Todos usaram os possíveis construtores que apresentamos. Porém na prática somente o l1 não está presente na tela.

Observe na listagem 1 que na primeira linha criamos um objeto para guardar uma imagem (o Java a chama de ícone, mas pode ser uma imagem de qualquer tamanho). Esta imagem é da classe *ImageIcon* e deve estar localizada dentro do path da aplicação.

Explicando os outros *labels:*

JLABEL L1	Não é mostrado na tela porque seu construtor não recebeu parâmetros. É possível inserir um texto ou imagem posteriormente usando os métodos *setText* e setIcon.
JLABEL L2	É mostrada apenas a imagem da bandeira do Brasil devido ao parâmetro do seu construtor.
JLABEL L3	Este caso é semelhante ao anterior, porém a imagem fica alinhada à direita, dentro do espaço que lhe é concedido. Como neste caso o espaço é bem pequeno, o efeito visual é parecido com o *label* l2.
JLABEL L4	Este é a forma mais usada para os *labels*. Somente mostra o texto que foi enviado por parâmetro.
JLABEL L5:	Este é uma junção do l2 e l3. Usamos uma imagem e um texto no mesmo objeto, com o texto alinhado à direita da figura.
JLABEL L6	Este construtor permite alinhar o texto (assim como a figura do l3) à direita, dentro da área especificada para o *label*.

Assim como os *JButton* que herdam da classe *JComponent*, temos os *JLabel* que também tem essa característica e podem usar métodos para mudar sua fonte, cor, tamanho etc. Observe a figura 2.5. O label agora possui uma cor de fundo (*background color*, amarelo), outra fonte (*foreground color*) com outra cor (*Serif*, azul) e uma borda fina vermelha.

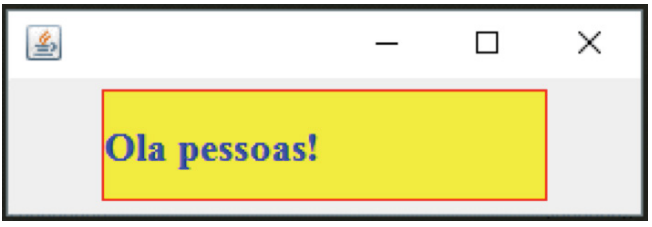


Figura 2.5 – Exemplo de modificação de alguns elementos visuais de um *JLabel.*

O programa para gerar o exemplo da figura 2.6 está mostrado na listagem 2,

```
1   import java.awt.Color;
2   import java.awt.Dimension;
3   import java.awt.FlowLayout;
4   import java.awt.Font;
5
6   import javax.swing.JFrame;
7   import javax.swing.JLabel;
8   import javax.swing.border.LineBorder;
9
10  public class Programa1 {
11      public static void main(String[] args) {
12          JFrame t = new JFrame();
13          t.setBounds(100, 100, 300, 100);
14          JLabel l = new JLabel("Ola pessoas!");
15
16          l.setFont(new Font("Serif", Font.BOLD, 18));
17          l.setOpaque(true);
18          l.setBackground(Color.YELLOW);
19          l.setForeground(Color.BLUE);
20          l.setBorder(new LineBorder(Color.RED));
21          l.setPreferredSize(new Dimension(200,50));
22          t.getContentPane().setLayout(new FlowLayout());
23          t.add(l);
24          t.setVisible(true);
25      }
26  }
```

Listagem 2 – Programa com exemplo de modificação visual em *JLabel.*

! ATENÇÃO

Vamos explicar o programa.

LINHA(S)	O QUE FAZ(EM)
1 A 8:	Importações necessárias das classes usadas no programa
10:	Definição da classe
11:	Definição do programa principal

LINHA(S)	O QUE FAZ(EM)
12:	Instanciação do JFrame, chamado t
13:	Dimensionamento do JFrame t usando o método setBounds
14:	Instanciação do JLabel l com o texto “Ola pessoas!”
16:	Mudança de fonte do Jlabel l. Para isso é necessário instanciar um objeto da classe Font. Fizemos isso com um objeto que possui as propriedades: tipo: Serif, negrito (Font.BOLD) e tamanho 18.
17:	Usamos o método setOpaque para tornar o JLabel l opaco (desta forma é possível aplicar uma cor de fundo)
18:	Definimos a cor de fundo (Background) do JLabel para amarelo
19:	Definimos a cor principal (Foreground) para azul
20:	Criamos uma borda vermelha simples ao redor do JLabel
21:	Nesta linha definimos um tamanho para o JLabel. Isto é necessário porque caso não fosse definido, o JLabel assumiria todo o tamanho do container onde ele está, no caso, o JFrame f.
22-23:	Essas são nossas tradicionais linhas de inserção do componente no JFrame, definição do layout e torná-lo visível

Listas (*JList*)

Você já deve ter visto uma lista em vários *softwares* que você tem contato, por exemplo na lista de contatos do *Facebook*, na lista de fontes de editores de texto, planilhas eletrônicas, enfim, a lista é um componente bastante usado e bem útil.

A lista, *JList* em Java, é um conjunto de itens a partir dos quais o usuário pode selecionar um ou mais itens, ou seja, existem listas de seleção simples ou múltiplas.

Figura 2.6 – Exemplo de *JList* com seleção simples.

O código fonte para gerar o programa da figura 2.6 está mostrado em seguida.

```
1   import java.awt.FlowLayout;
2   import javax.swing.JFrame;
3   import javax.swing.JList;
4   import javax.swing.JScrollPane;
5   import javax.swing.ListSelectionModel;
6
7   public class Lista extends JFrame {
8        private JList lista_de_cores;
9        private   final   String   cores[]   =   {"Preto","Azul","Ciano","Cinza
Escuro","Cinza","Verde","Cinza Claro","Magenta"};
10
11       public Lista(){
12           super("Testando!");
13           setLayout(new FlowLayout());
14
15           lista_de_cores = new JList(cores);
16           lista_de_cores.setVisibleRowCount(5);
17           lista_de_cores.setSelectionMode(ListSelectionMo-
del.SINGLE_SELECTION);
18           add(new JScrollPane(lista_de_cores));
19       }
20
21       public static void main(String args[]){
22           Lista programa = new Lista();
23           programa.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24           programa.setSize(350,150);
25           programa.setVisible(true);
26       }
27  }
```

Listagem 3 – Código fonte da figura 2.6.

Fizemos uma pequena modificação na estrutura do programa apenas para levar você a refletir nas diferentes formas que o Java possui de usar a tecnologia de orientação a objetos.

! ATENÇÃO

Vamos a alguns detalhes da listagem para você entender o que foi feito:

LINHA(S):	O QUE FAZ(EM):
8:	Declaração da lista a ser usada, por meio da criação do objeto lista_de_cores.
9:	Declaração de um *array* contendo os itens da lista. Neste caso, uma lista de cores.
11-18:	Declaração do construtor da classe Lista
12:	Invocação do construtor da superclasse (*JFrame*), passando como parâmetro o valor: "Testando!"
13:	Escolha do gerenciador de *layout* do *Frame.*
15:	Instanciação do objeto lista_de_cores.
16:	Deixando a lista com 5 linhas visíveis para o usuário.
17:	Nesta linha escolhemos o modo de seleção de itens da lista. Usamos a constante *SINGLE_SELECTION* da classe *ListSelectionModel*. Neste caso, a lista só permitirá a seleção de um item por vez. Caso a *JLista* fosse de seleção múltipla, a constante a ser usada seria *MULTIPLE_INTERVAL_SELECTION*
18:	Nesta linha atribuímos uma barra de rolagem vertical para a lista. Isto é necessário em Java e não é automático como em outras linguagens. O programador precisa prever que a lista pode ter mais itens que o seu tamanho e atribuir um *JScrollPane* para o componente necessário.
21-26:	Nestas linhas o método principal é declarado e cria um objeto chamado "programa" para ser usado como *JFrame*. Nestas linhas o *JFrame* é brevemente configurado.

Figura 2.7 – Exemplo de *JList* com múltipla seleção.

A figura 2.7 mostra um *JList* configurado para permitir seleções múltiplas, (neste caso usando a constante *ListSelectionModel.MULTIPLE_INTERVAL_SELECTION*). Além disso, ele foi configurado para mostrar 10 linhas visíveis.

Variações de componentes visuais

Bem, já vimos um pouco sobre os componentes visuais que podemos usar nas telas em geral. É claro que existem muito mais detalhes, métodos e atributos que poderíamos explorar, mas vamos deixar para a sua curiosidade e estudo. É importante que você pratique e explore o que pode ser feito.

Neste tópico vamos colocar um pouco mais de detalhes nas telas de exemplos. Lembre-se que os componentes visuais, por serem subclasses de *JComponent,* possuem muitos métodos em comum com o que já vimos.

Vamos usar como exemplo duas janelas muito encontradas em aplicativos que acessam banco de dados. As telas são utilizadas para executar as operações chamadas *CRUD* (*Create, Read, Update e Delete*, ou, Inserção, Consulta, Atualização e Eliminação respectivamente). Basicamente, estas telas precisam ter componentes visuais para mostrar os dados e fazer operações com eles. Não é nosso objetivo mostrar a conexão com o banco de dados, porém com o que for mostrado dará para você ter uma ideia de como criar este tipo de *interface*.

Para este tipo de desenvolvimento, é recomendado que você use alguma ajuda para criar as telas pois instanciar os componentes, configurá-los e posicioná-los "na mão", ou seja, diretamente pelo código, é uma tarefa árdua e demorada, mesmo usando gerenciadores de *layout*.

CONEXÃO

Procure por algumas ferramentas que ajudarão você no processo de criação de janelas:

- *Netbeans* (<www.netbeans.org>): Sem dúvida, o mais popular que oferece capacidades de criação para arrastar e soltar componentes *Swing;*
- *Window Builder* (<https://eclipse.org/windowbuilder/>) : famoso *plugin* para o Eclipse o qual permite criar interfaces gráficas também com recursos de arrastar e soltar;
- *Form Designer* (<http://www.formdev.com/>): Outra ferramenta gráfica que auxilia o desenvolvedor na criação de *interfaces*.

Vamos usar o *Netbeans* para nos ajudar nessa tarefa.

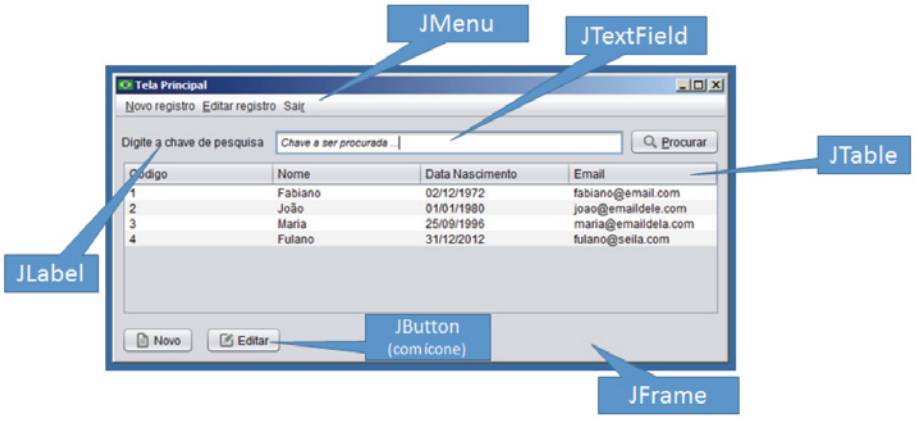


Figura 2.8 – Tela de consulta de dados de uma aplicação exemplo.

A figura 2.8 e a figura 2.9 mostram um exemplo de uma aplicação para cadastro de dados de uma academia de musculação. Lembre-se, é apenas um exemplo, sabemos que poderíamos ter muito mais campos nas telas e a nossa intenção é demonstrar o que pode ser feito e expandido posteriormente.

A figura 2.8 representa uma tela na qual o usuário pode ver os dados cadastrados e alguns dos campos de cadastro. Além disso, ele pode filtrar e acionar os botões para criar um novo registro ou editar outro que esteja selecionado na tabela. O usuário pode também usar os menus para criar novo registro, editar um registro selecionado ou sair da tela. Ou seja, o acionamento dos botões "Novo" ou "Editar" bem como os menus, levam o programa com os dados para a tela da figura 2.9. A

tela da figura 2.8 pode ser usada como uma tela principal de um sistema simples, além disso, ela pode ser uma superclasse para outras telas semelhantes.

Os componentes usados para a criação das telas estão mostrados nas figuras. Cada um dos componentes possui formas de instanciação específicas, porém não muito diferentes do que vimos no *JButton* ou *JLabel*. Veja a forma de instanciação dos componentes a seguir.

```
jScrollPane1 = new javax.swing.JScrollPane();
jTable1 = new javax.swing.JTable();
jTextField1 = new javax.swing.JTextField();
jButton1 = new javax.swing.JButton();
jLabel1 = new javax.swing.JLabel();
jButton2 = new javax.swing.JButton();
jButton3 = new javax.swing.JButton();
jMenuBar1 = new javax.swing.JMenuBar();
jMenu1 = new javax.swing.JMenu();
jMenu2 = new javax.swing.JMenu();
jMenu3 = new javax.swing.JMenu();
```

Portanto, percebemos que a instanciação dos componentes não é complicada. A seguir mostramos como alguns atributos são setados por meio dos métodos de cada um dos componentes.

Sobre o *JTable*

O *JTable* é sem dúvida o mais complexo destes componentes. Para instanciar uma tabela como a da figura 2.8, a qual é bem simples, criamos um *TableModel* como mostrado no extrato do código anterior. O *TableModel* é um objeto que controla o estado e comportamento de um *JTable*. No nosso exemplo, estamos usando valores fixos (por isso temos a *String* mostrada no código a seguir). Em aplicações que se relacionam com banco de dados, o *TableModel* tem que se adequar às consultas e resultados provenientes do banco de dados para poder mostrar os dados corretamente. Veja a configuração da *JTable*:

```
jTable1.setModel(new javax.swing.table.DefaultTableModel(
    new Object [][] {
        {"1", "Fabiano", "02/12/1972", "fabiano@email.com"},
        {"2", "João", "01/01/1980", "joao@emaildele.com"},
        {"3", "Maria", "25/09/1996", "maria@emaildela.com"},
        {"4", "Fulano", "31/12/2012", "fulano@seila.com"}
    },
    new String [] {
        "Código", "Nome", "Data Nascimento", "Email"
    }
));
jScrollPane1.setViewportView(jTable1);
```

Perceba que tivemos que usar outro componente da classe *JScrollPanel* para associar à *JTable* e criar as barras de rolagem vertical quando houver mais dados que o tamanho vertical do componente.

Vamos explorar um pouco mais o *JTable*. Por hora, veja o que é possível fazer com este componente:

Figura 2.9 – Exemplo de *JTable*

Este exemplo mostra uma característica interessante da *JTable*: a possibilidade de transformar um campo boolean em um *checkbox* (veja a coluna Teste). O código fonte que gera o programa da figura 2.9 está mostrado na listagem 4.

```
1   import javax.swing.*;
2   import javax.swing.table.*;
3
4   public class TabelaExemplo extends JFrame {
5
6       private static final long serialVersionUID = 1L;
7       private JTable tabela;
8
9       public TabelaExemplo() {
10          Object[] colunas = {"Tipo", "Empresa", "Mercado", "Preco", "Teste"};
11          Object[][] dados = {
12             {"Servidores", "IBM", new Integer(1000), new Double(80.50), false},
13                {"Software", "MicroSoft", new Integer(2000), new Double(6.25),
true},
14            {"Automoveis", "Toyota", new Integer(3000), new Double(7.35), true},
15              {"Esportes", "Nike", new Integer(4000), new Double(20.00), false}
16          };
17          DefaultTableModel modelo = new DefaultTableModel(dados, colunas);
18          tabela = new JTable(modelo) {
20              private static final long serialVersionUID = 1L;
21              @Override
22              public Class getColumnClass(int coluna) {
23                  switch (coluna) {
24                      case 0:
25                          return String.class;
26                      case 1:
27                          return String.class;
28                      case 2:
29                          return Integer.class;
30                      case 3:
31                          return Double.class;
32                      default:
33                          return Boolean.class;
34                  }
35              }
36          };
37          tabela.setPreferredScrollableViewportSize(tabela.getPreferredSize());
38          JScrollPane scrollPane = new JScrollPane(tabela);
```

```
39          getContentPane().add(scrollPane);
40      }
41
42      public static void main(String[] args) {
43          SwingUtilities.invokeLater(new Runnable() {
44
45              @Override
46              public void run() {
47                  TabelaExemplo frame = new TabelaExemplo();
48                  frame.setDefaultCloseOperation(EXIT_ON_CLOSE);
49                  frame.pack();
50                  frame.setLocation(150, 150);
51                  frame.setVisible(true);
52              }
53          });
54      }
55  }
```

Listagem 4 – Código fonte da figura 2.9.

A classe *JTable* da listagem 4 possui 2 *arrays* que armazenam os títulos das colunas e os dados da tabela. Estes arrays vão formar os parâmetros do construtor do modelo da tabela (veja a linha 17).

Embora já tenhamos falado sobre o modelo anteriormente, o objeto modelo da tabela é usado para gerenciar os dados que vão "popular" a tabela. Este objeto deve implementar a *interface TableModel* (novamente na linha 17). Caso o programador não especifique o objeto da classe *TableModel*, o *JTable* vai criar uma instância da classe *DefaultTableModel*.

De acordo com a documentação oficial da *Oracle*, a classe *JTable* possui os seguintes construtores.

CONSTRUTOR	FUNÇÃO
JTABLE	Instancia uma *JTable* padrão que é inicializada com um modelo de dados e de colunas padrão.
JTABLE (INT,INT)	Instancia uma *JTable* com um número de linhas e colunas respectivamente enviados por parâmetro. As células serão vazias neste caso.

CONSTRUTOR	FUNÇÃO
JTABLE(OBJECT [][], OBJECT[])	Neste construtor é possível passar como parâmetro os dados que estarão na tabela (primeiro parâmetro) e os nomes das colunas (no segundo parâmetro). Observe que o primeiro parâmetro é uma matriz e segundo um vetor.
JTABLE(TABLEMODEL)	Neste caso, a *JTable* é instanciada com um modelo de dados especificado, e um modelo de colunas e de seleção padrão
JTABLE(TABLEMODEL, TABLECOLUMNMODEL)	Neste construtor, a *JTable* é instanciada com um modelo de tabela e de coluna especificado pelos parâmetros.
JTABLE(TABLEMODEL, TABLECOLUMNMODEL, LISTSELECTIONMODEL)	Aqui a *JTable* é inicializada com o modelo de tabelas, modelo de colunas e de seleção passados por parâmetro.
JTABLE(VECTOR,VECTOR)	Nesta forma a *JTable* vai mostrar os valores dos parâmetros como dados e nomes de colunas.

Tabela 2.4 – Construtores da classe *JTable.*

Portanto, observando a tabela 2.4 percebemos que usamos o construtor que recebe um *DataModel* no exemplo da listagem 4. Perceba que na linha 21 existe uma anotação chamada *@Override* (sobrecarga). Esta anotação indica que o método *getColumnClass(int)* presente na classe *JTable* será reescrito pelo nosso programa. Desta forma, quando o Java for instanciar a *JTable* na linha 18, ela receberá como parâmetro o modelo (instanciado na linha 17) e vai aplicar o método. O método vai fazer com que a coluna 5 seja do tipo *Boolean* e na instanciação da *JTable*, o compilador ao invés de mostrar o valor do campo: "*false*" ou "*true*" (definidos nas linhas 12 a 15 para cada linha), mostre um *checkbox*, o qual estará selecionado se o valor for "*true*" ou não selecionado se o valor for "*false*".

Como dissemos, a classe *JTable* dá um pouco mais de trabalho para poder entender e ser usada. Porém há uma grande vantagem nela: se você entender a sua dinâmica e seus modelos, ela pode ser muito flexível a ponto de encontrarmos

derivações da *JTable* na internet até mesmo pagas para poder atender os vários tipos de necessidades de programadores, principalmente aqueles que trabalham com sistemas conectados a banco de dados.

As outras linhas da listagem são mais estruturais e servem para instanciar o *JFrame*, principalmente as linhas dentro do método main. Perceba também que usamos um *JScrollPane* caso o número de linhas, se aumentar, exceda o tamanho da tabela.

Existem mais detalhes sobre a *JTable* que vamos deixar para você estudar e usar.

CONEXÃO

Links sobre *JTable*:

- <https://docs.oracle.com/javase/7/docs/api/javax/swing/JTable.html>. O principal. Aí vamos encontrar toda a documentação oficial da *Oracle* sobre esta classe.
- <https://docs.oracle.com/javase/tutorial/uiswing/components/table.html>. Este *link* faz parte do Java Tutorial. Trata-se de um pequeno tutorial escrito originalmente pela *Sun* o qual foi adaptado pela *Oracle*. Apesar de estar em inglês, vale a pena dar uma olhada.
- <https://www.youtube.com/watch?v=pj9eTCQa2VA>. Para quem quiser assistir um vídeo, o YouTube possui vários sobre a *JTable*. Este aqui é apenas um deles. Pesquise outros também.

Outros componentes

Os campos de texto são mais simples. No exemplo a seguir, a fim de ter um texto dentro da caixa para informar o usuário para digitar um texto a ser procurado, criamos uma formatação com a fonte Arial, tamanho 11, em itálico (o parâmetro 2 é o inteiro equivalente ao itálico). Para colocar um texto padrão na caixa de texto usamos o método *setText*:

```
jTextField1.setFont(new java.awt.Font("Arial", 2, 11));
jTextField1.setText("Chave a ser procurada ...");
```

Já mostramos o funcionamento do botão e a associação de um ícone a ele. A seguir mostramos que podemos atribuir uma tecla de atalho (combinada com a tecla *Control*), chamada mnemônico, por meio do método *setMnemonic*. A letra que é enviada por parâmetro é a tecla de atalho.

```
jButton1.setIcon(new     javax.swing.ImageIcon(getClass().getResource("/examples/
search.png")));
jButton1.setMnemonic('P');
jButton1.setText("Procurar");
jButton2.setIcon(new     javax.swing.ImageIcon(getClass().getResource("/examples/
file-2.png")));
jButton2.setText("Novo");
jButton3.setIcon(new     javax.swing.ImageIcon(getClass().getResource("/examples/
edit.png")));
jButton3.setText("Editar");
```

O menu existente nas *interfaces* feitas com *Swing* é composto de algumas partes: uma barra de menu (*JMenuBar*) e os menus de fato (*JMenu*). No nosso exemplo não temos submenus, mas caso tivéssemos usado, teríamos que instanciar objetos da classe *JMenuItem*. Perceba que instanciamos um objeto da classe *JMenu* e o atribuímos a uma barra de menus (*JMenuBar*). Além disso, podemos usar o método *setMnemonic* nesta classe também:

```
jMenu1.setMnemonic('N');
jMenu1.setText("Novo registro");
jMenuBar1.add(jMenu1);

jMenu2.setMnemonic('E');
jMenu2.setText("Editar registro");
jMenuBar1.add(jMenu2);

jMenu3.setMnemonic('r');
jMenu3.setText("Sair");
jMenuBar1.add(jMenu3);

setJMenuBar(jMenuBar1);
```

? CURIOSIDADE

Embora já citamos, mas você percebeu o quanto usamos métodos parecidos nos componentes da classe *Swing* (*setText, setMnemonic, add*, etc)? A hierarquia da classe *JComponent* é realmente muito importante para o desenvolvimento destas telas.

Vamos agora explorar os componentes usados na tela da figura 2.10. A ideia central desta tela é servir como formulário de inserção de novos dados ou de edição de dados existentes. Portanto, o grande número de *JLabels* é justificado (lembre-se: em um sistema real teríamos muito mais!).

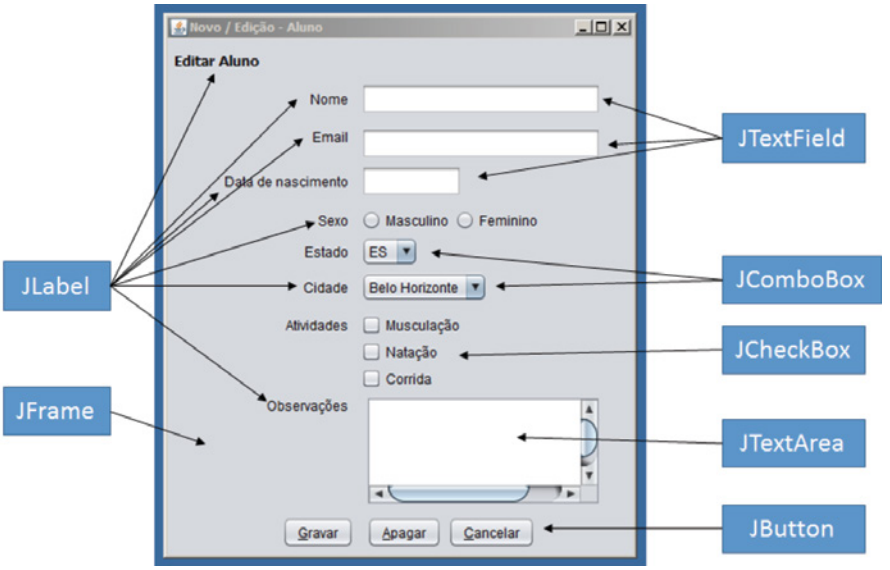


Figura 2.10 – Tela de novo registro ou edição de um registro existente.

Vamos ver como os componentes são instanciados.

```
jLabel1 = new javax.swing.JLabel();
jLabel2 = new javax.swing.JLabel();
jLabel3 = new javax.swing.JLabel();
jRadioButton1 = new javax.swing.JRadioButton();
jRadioButton2 = new javax.swing.JRadioButton();
jTextField1 = new javax.swing.JTextField();
jTextField2 = new javax.swing.JTextField();
```

```
jTextField3 = new javax.swing.JTextField();
jButton1 = new javax.swing.JButton();
jButton2 = new javax.swing.JButton();
jButton3 = new javax.swing.JButton();
jComboBox2 = new javax.swing.JComboBox();
jLabel4 = new javax.swing.JLabel();
jLabel5 = new javax.swing.JLabel();
jLabel6 = new javax.swing.JLabel();
jCheckBox1 = new javax.swing.JCheckBox();
jCheckBox2 = new javax.swing.JCheckBox();
jCheckBox3 = new javax.swing.JCheckBox();
jLabel7 = new javax.swing.JLabel();
jLabel8 = new javax.swing.JLabel();
jScrollPane1 = new javax.swing.JScrollPane();
jTextArea1 = new javax.swing.JTextArea();
jLabel9 = new javax.swing.JLabel();
jComboBox1 = new javax.swing.JComboBox();
```

Novamente, a forma de instanciação é bem simples. O uso dos métodos dos componentes para configurá-los, embora bem semelhante com os métodos usados nos componentes da figura 2.10, possuem algumas pequenas diferenças:

```
Label1.setText("Nome");
jLabel2.setText("Email");
jLabel3.setText("Data de nascimento");
jRadioButton1.setText("Masculino");   jRadioButton2.setText("Feminino");
jButton1.setMnemonic('G');
jButton1.setText("Gravar");
jButton2.setMnemonic('A');
jButton2.setText("Apagar");
jButton3.setMnemonic('C');
jButton3.setText("Cancelar");
jComboBox2.setModel(new  javax.swing.DefaultComboBoxModel  (new  String[]  {"Belo
Horizonte","Rio de Janeiro","São Paulo", "Vitória" }));
jLabel4.setText("Estado");
jLabel5.setText("Cidade");
jLabel6.setText("Sexo");
jCheckBox1.setText("Musculação");
jCheckBox2.setText("Natação");
jCheckBox3.setText("Corrida");
jLabel7.setText("Atividades");
jLabel8.setText("Observações");
```

```
jTextArea1.setColumns(20);
jTextArea1.setRows(5);
jScrollPane1.setViewportView(jTextArea1);
jLabel9.setFont(new java.awt.Font("Tahoma", 1, 12));
jLabel9.setText("Editar Aluno");
jComboBox1.setModel(new  javax.swing.DefaultComboBoxModel(new  String[]  {  "ES",
"MG", "RJ", "SP" }));
```

Assim como o *JTable*, o *JComboBox* também possui um *Model* que define seu estado e comportamento. E da mesma forma que o *JTable*, em uma aplicação conectada a um banco de dados, normalmente o conteúdo do modelo de um *JComboBox* é proveniente do resultado de uma consulta feita pela aplicação.

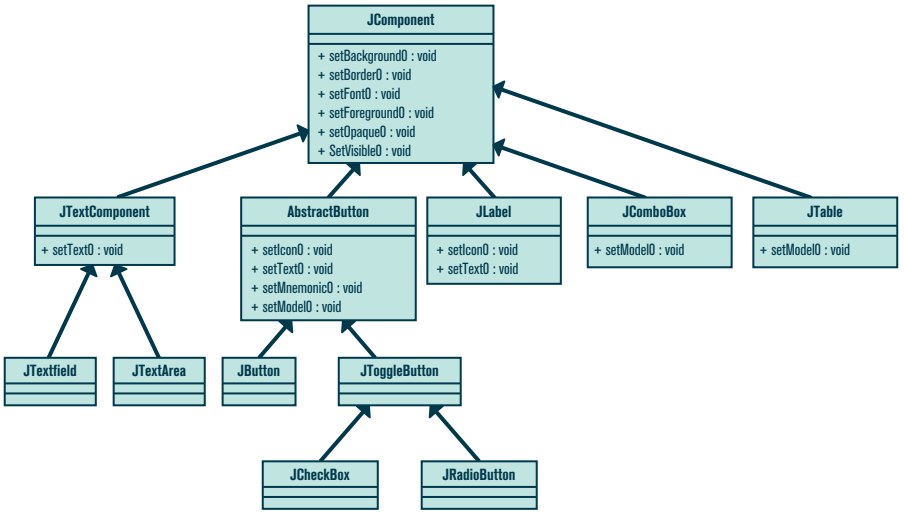


Figura 2.11 – Hierarquia da classe *JComponent* com os componentes usados e seus principais métodos.

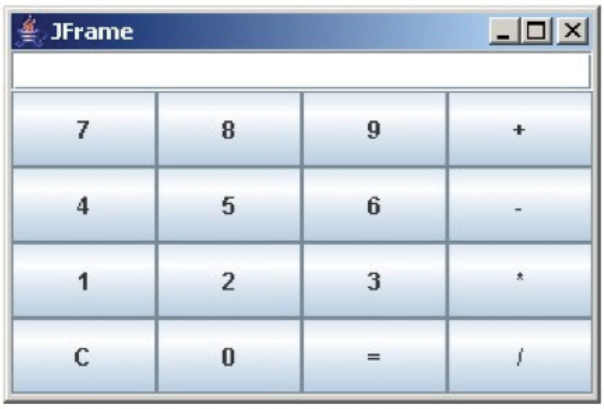
A figura 2.11 mostra um resumo dos componentes e métodos que usamos nos exemplos anteriores. Perceba o quanto os conceitos de herança, os quais você deve ter estudado quando aprendeu orientação a objetos, são importantes. Veja que todos os componentes apresentados na hierarquia podem ter sua cor de fundo mudada, assim como a cor principal, mudar a fonte, entre outros métodos que

não foram mostrados aqui por não fazerem parte do nosso contexto. É interessante observar que o *JCheckBox* e o *JRadioButton*, de acordo com a hierarquia, são "irmãos" (e são tão diferentes!) e se pararmos para analisar, são sobrinhos do *JButton*! Mais uma vez mostramos o quanto é importante você dedicar um tempo à API da JFC *Swing* para descobrir os métodos e atributos das classes e o quanto elas podem ser úteis em suas aplicações.

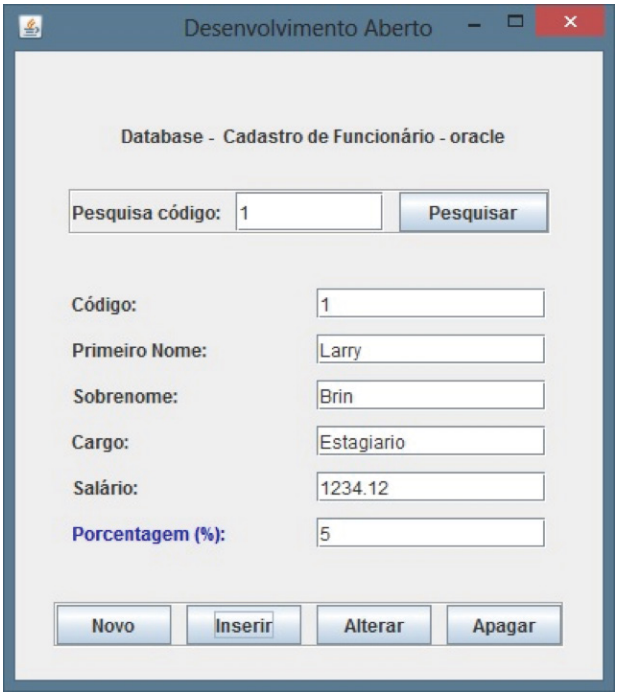
Nos exemplos (figura 2.8 e figura 2.10), foi usado o *Netbeans* para criar e ajudar na diagramação das telas. O gerenciador de *layout* padrão adotado pelo *Netbeans* é o *GroupLayout*. Para você ter uma ideia o quanto o uso de uma IDE pode simplificar e facilitar o trabalho do programador, somente a parte de configuração e posicionamento dos componentes dentro do layout possui 100 linhas aproximadamente para a tela da figura 2.10. Ao todo, esta tela possui 222 linhas de código gerados automaticamente pelo *Netbeans*, retirando todos os comentários. É bastante código para uma tela aparentemente simples. E ainda não incluímos as interatividades da tela (que serão vistas no próximo capítulo).

ATIVIDADES

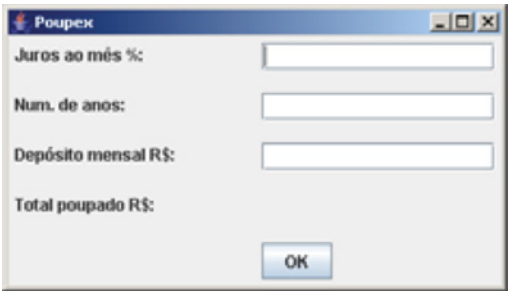
01. Monte a seguinte tela. Não acrescente as funcionalidades ainda.

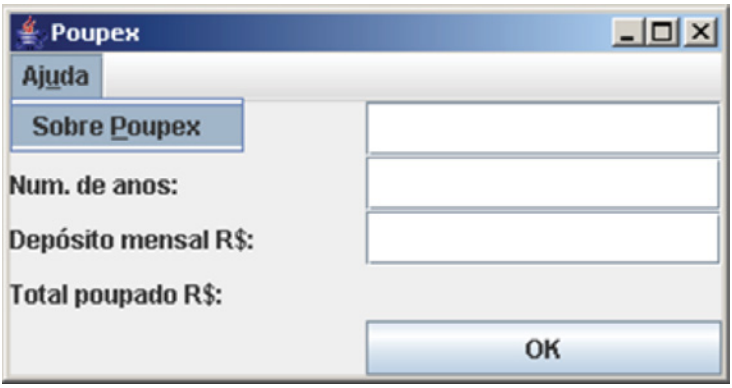


02. Monte a seguinte tela, sem acrescentar funcionalidades.



03. Faça as seguintes telas sem acrescentar funcionalidades.





REFLEXÃO

Durante todo o capítulo percebemos o quanto a orientação a objetos influencia o desenvolvimento de telas e interfaces em Java. Vimos a hierarquia da classe *JComponent* e os principais métodos que usamos nos exemplos. Outros métodos e atributos existem, multiplicando as possibilidades de criação de telas. Como desenvolvedor, você naturalmente tem que ter a curiosidade de explorar os elementos que apresentamos de uma maneira mais detalhada. Procure na internet por componentes derivados da *JTable*. Você vai encontrar componentes gratuitos muito bons e outros pagos que também valem a pena ser explorados.

Considere usar alguma IDE para o desenvolvimento de *interfaces*. Apesar do código fonte gerado por essas ferramentas ser maior, é possível otimizá-los e unir a produtividade dessas ferramentas e sua habilidade natural para criar estruturas mais eficientes. Bom trabalho!

REFERÊNCIAS BIBLIOGRÁFICAS

DEITEL, H. M.; DEITEL, P. J. **Java:** como programar. 6ª. ed. Rio de Janeiro: Bookman, 2005.

HORSTMANN, C. S.; CORNELL, G. **Core Java**. Rio de Janeiro: Pearson Education, 2010.

HUBBARD, J. R. **Programação com Java 2**. Rio de Janeiro: Bookman, 2006.

ORACLE CORPORATION. **The Java Tutorials.** Oracle Java Documentation, 01 mar. 2015. Disponivel em: <https://docs.oracle.com/javase/tutorial>. Acesso em: 1 mar. 2016.

ORACLE CORPORATION. **Java Platform, Standard Edition 7**. API Specification, 2016. Disponivel em: <https://docs.oracle.com/javase/7/docs/api/>. Acesso em: 01 mar. 2016.

3

Tratamento de eventos para *interfaces* gráficas I

Tratamento de eventos para *interfaces* gráficas I

Até agora abordamos vários componentes que podem ser usados em *interfaces* gráficas em Java. Mas sabemos que apenas os componentes não bastam para dar a interatividade prevista para estas telas, é necessário que os componentes executem as ações óbvias para cada um deles e que o programa responda para isso (exemplo: o mais óbvio em um botão é clicá-lo, logo, esperamos que o botão ao ser clicado faça alguma coisa!).

Assim como os componentes, os eventos são objetos de classes que pertencem a uma hierarquia em Java. E mais uma vez apontamos a grande necessidade de você entender e usar corretamente a orientação a objetos. Desta vez vamos usar a hierarquia da classe *AWTEvent*, ou seja, os eventos são pertencentes a uma classe da biblioteca AWT que é a primeira forma usada pela linguagem Java para tratar de interfaces gráficas.

Novamente também apontamos que você não esqueça nunca de consultar a API do Java para poder buscar outros tipos de eventos diferentes dos triviais para seus componentes. O *JButton* por exemplo, pode ter outros eventos diferentes do simples "*click*", sabia?

Por enquanto é isso, vamos começar nosso estudo. Vamos estudar os eventos em duas partes e este capítulo será a primeira delas.

OBJETIVOS

Este capítulo tem os seguintes objetivos:

- Explorar a manipulação de aspectos visuais;
- Estudar variações de componentes visuais.

Tipos comuns de eventos e *listeners*

Um evento é algum tipo de resultado a uma ação. Quando clicamos com o botão direito do *mouse* (ação), temos como resultado o aparecimento de um menu de opções chamado *Dropdown* (reação). Quando apertamos a tecla *Enter* em um campo de texto também ocorre uma reação e assim por diante.

Existe um paradigma chamado Programação Orientada a Eventos. Neste paradigma a ideia central é entender que a aplicação não segue um fluxo tradicional de começo, meio e fim. Em uma *interface* gráfica, como um navegador de internet, o programa fica esperando o que o usuário vai fazer: digitar uma *url*, clicar em um botão, acionar um menu etc. O programa é executado conforme o acionamento de componentes (ação) e a sua reação: os eventos.

Na linguagem Java, as informações sobre qualquer tipo de evento que ocorre são guardadas em um objeto de uma classe que estende a classe *AWTEvent*.

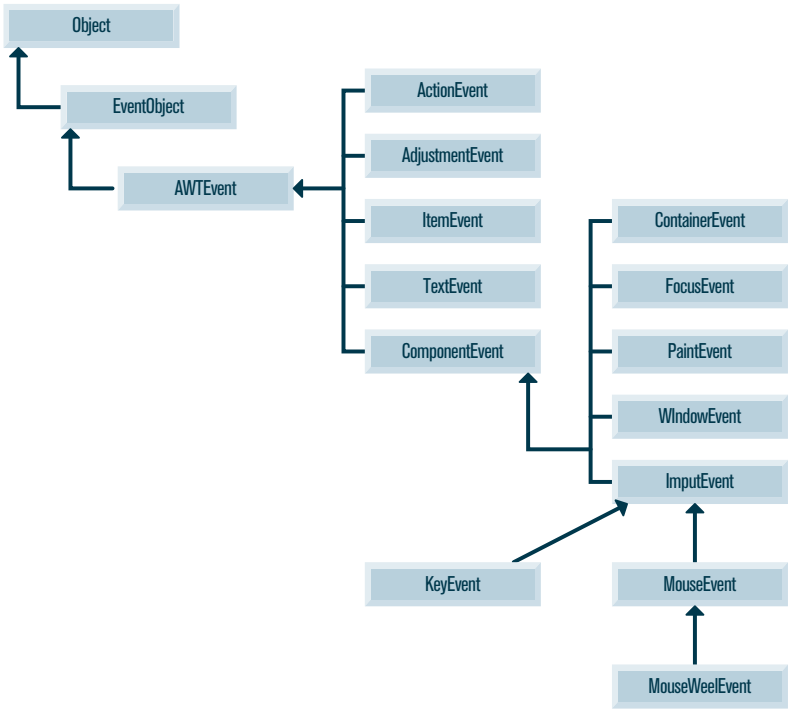


Figura 3.1 – Hierarquia da classe *AWTEvent* (DEITEL & DEITEL, 2005).

Conforme mostra a figura 3.1, os eventos podem ser usados tanto em componentes AWT quanto nos componentes das classes do pacote *javax.swing*. Na verdade, todo objeto pode ser notificado por um evento desde que seja implementada uma *interface* específica e apropriada e então registrar os métodos de tratamento do evento.

Os componentes do pacote *javax.swing* podem gerar vários tipos de eventos, por exemplo:

- Quando o usuário clica em um botão, gera um *ActionListener*;

- Quando fecha um *frame*, gera um *WindowListener*;
- Botão do mouse pressionado gera um *MouseListener*;
- Movimentação do *mouse* gera um *MouseMotionListener.*

Para entender o correto funcionamento dos eventos, precisamos saber que:

- Cada evento é representado por um objeto que contém informações sobre este evento
- A origem do evento (*event source*) é quem gera o evento. Normalmente é algum componente da *interface* gráfica.
- O *listener* responde ao evento e pode ser qualquer classe em Java que implemente a *interface* correta. Perceba que uma única fonte (origem do evento) pode ter múltiplos *listeners* e um único listener pode responder a várias origens.

Basicamente, o que deve ser feito para tratar eventos?

1. Crie uma classe que vai implementar a interface associada ao evento a ser tratado. Observe novamente a grande dependência do conhecimento da orientação a objetos. Implementar uma interface significa codificar todos os métodos que são especificados por ela. Vamos ver isso daqui a pouco com mais detalhes. Esta interface tem o formato componenteListener, por exemplo, MouseListener, ActionListener etc.

2. Crie um objeto desta classe que foi definida no item 1.

3. Registre o objeto criado como o tratador de eventos (handler) de um determinado objeto usando o método apropriado. Este método tem o formado addComponenteListener.

CONCEITO

As ações das respostas realizadas em um evento são chamadas de *handler* (ou manipulador) do evento e o processo geral de responder aos eventos é chamado de tratamento de evento. Para cada tipo de evento, é preciso criar um objeto de escuta, chamado *listener*. Entre os tipos de eventos mais usados temos: uma ação (*ActionEvent*), um clique do *mouse* (*MouseEvent*), quando se aperta uma tecla (*KeyEvent*), e eventos de janela (fechar, maximizar, etc – *WindowEvent*).

Portanto, em uma aplicação ocorre um *loop* invisível para o usuário no qual ele fica esperando que algum evento seja disparado. Quando o evento ocorre, o sistema operacional cria um evento e passa para um tratador de eventos (o *handler*) que foi definido pelo programador. Isto é chamado de *callback*.

Vamos fazer um exemplo para poder explicar melhor. Observe a figura 3.2:

Figura 3.2 – Exemplo de evento usando o *ActionListener*.

Esta figura mostra uma aplicação bem simples. Trata-se de um *Jframe* com um *JButton* o qual ao ser clicado apresenta a caixa de diálogo da direita.

! ATENÇÃO

Veja que o *layout* da janela mudou em relação aos outros capítulos. Isto é proposital. No capítulo 2 o sistema operacional que estava rodando o programa em Java era o Windows 10. A partir deste capítulo vamos usar o *Ubuntu Linux*. Observe que o componente *JFrame* da biblioteca *Swing* é um componente leve e sendo assim, ele toma a forma do ambiente de janelas hospedeiro.

O código fonte referente à figura 3.2 é mostrado a seguir.

```
1   import java.awt.event.ActionListener;
2   import java.awt.event.ActionEvent;
3   import javax.swing.JFrame;
4   import javax.swing.JButton;
5   import javax.swing.JOptionPane;
6   import java.awt.FlowLayout;
7
8   public class Botao1 extends JFrame{
9       private JButton botao1;
```

```
10
11      public Botao1(){
12          super("Testando eventos!");
13          setLayout(new FlowLayout());
14          botao1 = new JButton("Clique-me !");
15          add(botao1);
16
17          //cria novo handler para tratamento de eventos no botao1
18          HandlerBotao handler = new HandlerBotao();
19          botao1.addActionListener(handler);
20      }
21
22      //classe interna para o tratamento de evento do botão
23      private class HandlerBotao implements ActionListener{
24          public void actionPerformed(ActionEvent evento){
25              JOptionPane.showMessageDialog(Botao1.this,String.format("Voce
clicou: %s",evento.getActionCommand()));
26          }
27      }
28
29      public static void main(String[] args){
30          Botao1 programa = new Botao1();
31          programa.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
32          programa.setSize(270,110);
33          programa.setVisible(true);
34      }
35  }
```

Listagem 1 – Exemplo de tratamento de eventos no botão.

Vamos à explicação do código. Devido a existir partes do código que já estudamos, vamos explicar as linhas que são diferentes e estão relacionadas ao tratamento de eventos.

Antes de começar, dê uma olhada na estrutura do programa, ou melhor, da classe Botao1, herdada da *JFrame*. Perceba que na linha 23 existe uma classe (chamada *HandlerBotao*) dentro da Botao1! Isso é possível? Em Java é sim e é chamada de classe interna ou classe aninhada.

CONCEITO

As classes "normais" em Java são chamadas também de classes de primeiro nível, ou seja, são classes que não foram declaradas dentro de outras classes. As classes aninhadas podem ser *static* ou não e quando são *static* são chamadas de classes internas e normalmente são usadas para tratar eventos, como no nosso exemplo. Para um objeto interno ser criado antes é preciso que um objeto da classe externa seja criado.

Ainda, o objeto da classe interna consegue acessar diretamente todas as variáveis de instância e métodos da classe externa.

LINHA(S)	O QUE FAZ(EM)
18:	Na linha 18 criamos um objeto, chamado *handler*, da classe *HandlerBotao.* Veja que este objeto é interno, criado pela classe interna *HandlerBotao*. Este objeto será o nosso *handler* de eventos para o botão. O método *actionPerformed* deste objeto será chamado automaticamente quando o usuário clicar no botão. Mas antes disso acontecer, o programa deve registrar esse objeto como *handler* de evento do botão.
19:	Esta linha é a instrução de registro de evento que especifica *handler* como o *handler* de evento do botão. O programa vai chamar o método *addActionListener* para registrar o *handler* de evento para o botão. O método recebe como parâmetro um objeto *ActionListener* que pode ser um objeto de qualquer classe que implemente *ActionListener*. O *handler* é um *ActionListener* porque a classe *HandlerBotao* implementa *ActionListener*. Depois que esta linha for executada, o handler "ouvirá" eventos. Quando o botão for clicado, o método *actionPerformed* da linha 24 na classe *HandlerBotao* será chamado para tratar o evento. Se um *handler* de evento não é registrado, o evento que ocorre será ignorado pelo programa.
23:	Nesta linha a classe interna é criada. Ela será responsável pelo tratamento do evento acionado pelo botão.
24:	Na linha 24 implementamos o método *actionPerformed* da *interface ActionListener*. Este método vai receber o objeto evento por parâmetro, que carrega as informações sobre o evento que ocorreu, inclusive a informação de quem o disparou. O sistema passa o objeto *ActionEvent* para o método *actionPerformed* do listener de eventos.

LINHA(S)	O QUE FAZ(EM)
25:	Nesta linha, é mostrada uma caixa de diálogo informando o usuário que o clique no botão ocorreu. Este diálogo é para efeitos de demonstração. Em um sistema real, várias tarefas poderiam ser feitas neste espaço. O método *getActionCommand* retorna uma referência à origem do evento e a exibe na caixa de diálogo concatenando com a frase: "Você clicou: ".

Basicamente é isto. Pode ser um pouco complicado no início, mas é um processo mecânico e não varia muito além disso. O que vai variar são as interfaces de listeners que podem ser adicionadas e devem ser implementadas.

Resumindo o que foi visto, temos que para fazer um tratador de eventos precisamos de 3 ações básicas:

1. Especificar uma classe que implemente uma *interface* de *Listener*, por exemplo:

```
public class MinhaClasse implements ActionListener{
```

2. Código que implemente métodos dentro da *interface Listener*:

```
public void actionPerformed(ActionEvent e){
    //implementar o que se deseja para o evento
}
```

3. Executar um código que registre uma instância desta classe como um listener de um ou mais componentes:

```
seuComponente.addActionListener(instância da MinhaClasse)
```

Eventos de Janelas

Uma vez que aprendemos o básico sobre como os eventos funcionam, vamos examinar como podemos aplicar estes conhecimentos nos eventos relacionados com janelas.

Os eventos de janela são tratados por classes que implementam a *interface WindowListener*.

A *interface* mencionada é definida conforme os seguintes métodos os quais tem que ser implementados na classe que implementará a *interface*:

```
public interface WindowListener {
    public void windowClosing(WindowEvent e);
    public void windowClosed(WindowEvent e);
    public void windowOpened(WindowEvent e) ;
    public void windowIconified(Window Event e) ;
    public void windowDeIconified(Window Event e) ;
    public void windowActivated(Window Event e) ;
    public void windowDeactivated(Window Event e) ;
}
```

Como já avisamos anteriormente, existe uma classe chamada *WindowAdapter* que tem uma implementação vazia de cada um desses métodos.

Vamos mostrar um exemplo com um programa bem simples: ele cria um frame no qual responde a alguns eventos entre eles:

- Quando o *frame* é minimizado;
- Quando o *frame* é restaurado;
- Quando o *frame* é aberto.

Para sabermos quais são estes eventos, adivinha onde podemos encontrá-los e sua respectiva documentação? Sim, na API da biblioteca! Neste caso, da biblioteca AWT.

CONEXÃO

A API do pacote *java.awt* está localizada na seguinte URL: <http://docs.oracle.com/javase/6/docs/api/java/awt/package-summary.html>. Nesta URL você encontrará todas as classes do pacote e as *interfaces* dos eventos.

Observando a API, os eventos que vamos usar como exemplo no programa são:

- Quando o *frame* é minimizado, evento *windowIconified*;
- Quando o *frame* é restaurado, evento *windowDeconinified*;
- Quando o *frame* é aberto, evento *windowOpened*.

A execução do programa é apresentada na figura 3.3.

Figura 3.3 – Eventos em janelas.

O programa demonstra o uso de eventos que são executados no *JFrame*. A cada evento que ocorre, invisível ou visível para o usuário, o contador de eventos é atualizado. Ao executar o programa talvez você fique em dúvida quanto ao incremento do contador mostrado na tela. Lembre-se que toda vez que o *JFrame* perder o foco, ele é desativado e um evento é gerado (*Deactivated*) e ao pegar o foco novamente, outro evento é disparado (*Activated*), logo, às vezes no programa o contador pode variar de 2 em 2. Veja o código do programa na listagem 2.

Ao verificar o programa, veremos que ele não é muito diferente do programa da listagem 1. Mas temos que tomar alguns cuidados. Os métodos das linhas 31, 37, 43, 48, 54 e 59 devem ser inseridos no código mesmo que eles não contenham nenhuma implementação. Isso ocorre porque estamos usando a *interface WindowListener* e ao usar uma *interface*, sabemos pelo paradigma da orientação a objetos, temos que implementar os métodos previstos nela.

No nosso exemplo, todos os métodos da *interface* foram implementados incrementando o contador e o mostrando na tela. Caso algum destes métodos não sejam declarados no programa, ocorrerá um erro de compilação alertando o usuário sobre a falta do método na classe.

```
 1 import java.awt.*;
 2 import java.awt.event.*;
 3 import javax.swing.*;
 4 import javax.swing.JOptionPane;
 5
 6 public class EventosJanela extends JFrame{
 7
 8     private JLabel label1;
 9     private JLabel labelContador;
10     private int contador;
11
12     public EventosJanela(){
13         super("Testando os eventos de Janela");
14         setLayout(new FlowLayout());
15         label1 = new JLabel("Eventos na Janela:");
16         add(label1);
17         labelContador = new JLabel("0");
18         add(labelContador);
19
20         HandlerJanela handler = new HandlerJanela();
21         this.addWindowListener(handler);
22     }
23
24     private class HandlerJanela implements WindowListener{
25         public void windowIconified(WindowEvent e) {
26             JOptionPane.showMessageDialog(null,"Minimizando ...");
27             contador++;
28             labelContador.setText(Integer.toString(contador));
29         }
30
31         public void windowOpened(WindowEvent e) {
32             JOptionPane.showMessageDialog(null,"Abrindo ...");
33             contador++;
34             labelContador.setText(Integer.toString(contador));
35           }
36
37           public void windowClosing(WindowEvent e) {
38               JOptionPane.showMessageDialog(null,"Tchau!");
39               contador++;
40               labelContador.setText(Integer.toString(contador));
41         }
42
43         public void windowClosed(WindowEvent e) {
```

```
44              contador++;
45              labelContador.setText(Integer.toString(contador));
46            }
47
48            public void windowDeiconified(WindowEvent e) {
49              JOptionPane.showMessageDialog(null,"Restaurando ...");
50              contador++;
51              labelContador.setText(Integer.toString(contador));
52            }
53
54            public void windowActivated(WindowEvent e) {
55                contador++;
56                labelContador.setText(Integer.toString(contador));
57            }
58
59            public void windowDeactivated(WindowEvent e) {
60                contador++;
61                labelContador.setText(Integer.toString(contador));
62            }
63    }
64
65
66     public static void main(String[] args){
67         EventosJanela programa = new EventosJanela();
68         programa.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
69         programa.setSize(270,110);
70         programa.setVisible(true);
71     }
72 }
```

Listagem 2 – Eventos em janelas.

Eventos de botões e menus

No capítulo 2 vimos que os objetos da classe *JButton*, assim como os da classe *JCheckBox* e *JRadioButton* são subclasses da classe *AbstractButton*, ou seja, existem atributos e métodos presentes nos *JButton* os quais também estão nos *checkboxes e radiobuttons.*

Vamos mostrar um exemplo usando *JCheckBox* a fim de colocar você em contato com outros tipos de componentes e fugir do *JButton*, muito comum neste tipo de assunto.

A figura 3.4 mostra o nosso exemplo. O exemplo consiste em um *JFrame* com uma caixa de texto (*JTextField*) e dois *checkboxes* (*JCheckBox*). A caixa de texto é instanciada com uma frase escrita em uma fonte sem serifa (*Sans Serif*). Se o usuário clicar em alguma caixa de texto, um evento é disparado para o tratador de eventos correspondente o qual vai aplicar a mudança da fonte dentro da caixa de texto. Como podemos ver, as mudanças somente são em tornar o texto em negrito ou itálico. É claro que podemos aumentar as funcionalidades do programa, mas a intenção aqui é demonstrar como ocorrem os eventos de botão, lembrando que *JCheckBox* também é um *AbstractButton*.

Figura 3.4 – Eventos em *checkboxes*.

Observando a listagem 3 percebemos que a parte que faz a "mágica" do programa (mudar o estilo da fonte da caixa de texto) usa métodos específicos da classe *JCheckBox*, os quais são bastante amigáveis:

- O método *isSelected()* das linhas 38 e 42, informa se o *checkbox* está selecionado ou não. Ele retorna um valor booleano como resposta.
- A formatação da fonte é dada pela instanciação de um objeto da classe *Font* (linhas 19 e 44). Observe os parâmetros que existem para o construtor da classe. Usamos as constantes da classe *Font* (*Font.PLAIN, Font.BOLD e Font.ITALIC*) para configurar o estilo da fonte. Para conhecer as outras constantes que existem nesta classe para poder formatar as fontes nas suas aplicações enquanto o programa estiver executando, consulte a API da classe *Font*. Disponível em: <https://docs.oracle.com/javase/7/docs/api/java/awt/Font.html>.

```
1 import java.awt.FlowLayout;
2 import java.awt.Font;
3 import java.awt.event.ItemListener;
4 import java.awt.event.ItemEvent;
5 import javax.swing.JFrame;
```

```
 6 import javax.swing.JTextField;
 7 import javax.swing.JCheckBox;
 8
 9 public class EventosCheck extends JFrame{
10     private JTextField caixaDeTexto;
11     private JCheckBox negrito;
12     private JCheckBox italico;
13
14     public EventosCheck(){
15         super("Testando CheckBoxes");
16         setLayout(new FlowLayout());
17
18         caixaDeTexto = new JTextField("Vamos mudar o estilo da fonte",20);
19         caixaDeTexto.setFont(new Font("Sans Serif",Font.PLAIN,14));
20         add(caixaDeTexto);
21
22         negrito = new JCheckBox("Negrito");
23         italico = new JCheckBox("Italico");
24         add(negrito);
25         add(italico);
26
27         HandlerCheckBox handler = new HandlerCheckBox();
28         negrito.addItemListener(handler);
29         italico.addItemListener(handler);
30     }
31
32     private class HandlerCheckBox implements ItemListener{
33         private int iNegrito = Font.PLAIN;
34         private int iItalico = Font.PLAIN;
35
36         public void itemStateChanged(ItemEvent evento){
37             if (evento.getSource()==negrito){
38                 iNegrito = negrito.isSelected()?Font.BOLD:Font.PLAIN;
39             }
40
41             if(evento.getSource()==italico){
42                 iItalico = italico.isSelected()?Font.ITALIC:Font.PLAIN;
43             }
44            caixaDeTexto.setFont(new Font("Sans Serif",iNegrito+iItalico,14));
45         }
46     }
47
```

```
48      public static void main(String[] args){
49          EventosCheck programa = new EventosCheck();
50          programa.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
51          programa.setSize(320,100);
52          programa.setVisible(true);
53      }
54  }
```

Listagem 3 – Eventos em *checkboxes*.

Vamos, agora, examinar a forma como uma IDE nos ajuda a criar os eventos dos componentes que estudamos. No exemplo a seguir, vamos usar o *Netbeans* e usar um *JMenu* para exemplificar os eventos.

O exemplo consiste de uma tela principal contendo um menu e um item de menu. Ao clicar no item de menu, um novo frame é instanciado e mostrado na tela. Este por sua vez, contém um label com uma imagem e um botão. Quando o usuário clica no botão é mostrado um diálogo para o usuário perguntando se ele deseja terminar a aplicação. Bem simples. Veja as figuras do programa na figura 3.5.

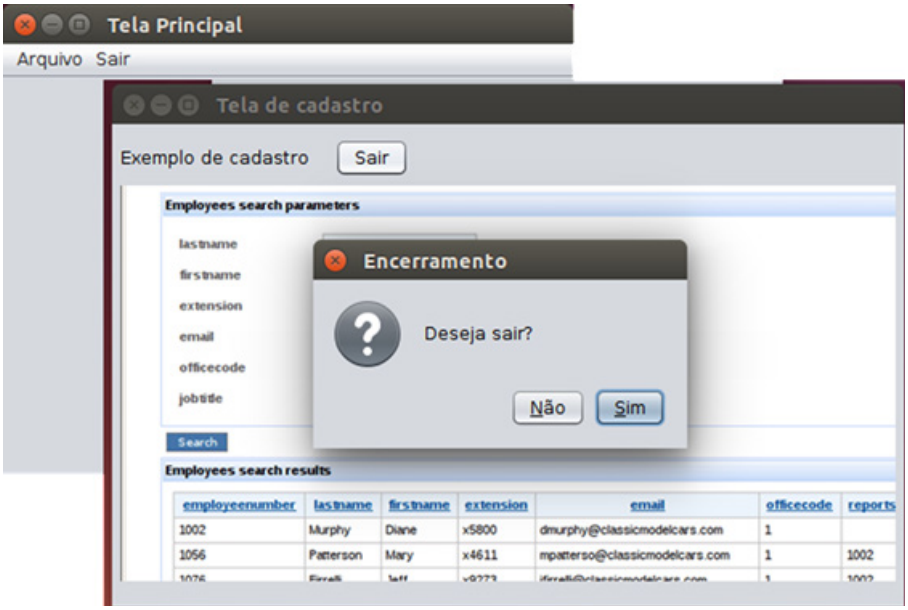


Figura 3.5 – Exemplo de evento de menu chamando *JFrame*.

Este exemplo é simples, porém mais complicado que os demais. Em primeiro lugar são usados 2 *JFrames* para compor o programa. Logo, são 2 arquivos separados.

A listagem do *frame* principal está mostrada em seguida. Gostaríamos de destacar a simplicidade da tela comparada com a grande quantidade de código fonte gerada pela IDE. É bastante para pouca coisa, não é? A Listagem 4 não está completa. Algumas linhas foram excluídas para fins de simplificação e mostrar para você a parte mais significativa e relacionada com o nosso contexto.

Um detalhe que tem que ser observado é que apesar do componente ser um item de menu, o clique do *mouse* sobre ele vai resultar em um evento *ActionPerformed*, o qual foi tratado nas linhas 48 e 49.

Nestas linhas é criada uma instância do outro *frame* e acionado o método *setVisible* dela. Este método, com o argumento *true*, é responsável por exibir o frame na tela.

Mas perceba como o *Netbeans* faz para criar o *handler* do evento na linha 22. Ele não usa uma classe aninhada como nossos exemplos. Ele usa um tipo de classe chamada classe interna anônima.

Esta classe é uma forma especial de classe interna a qual é declarada sem nome e em geral aparece dentro da declaração de um método. Assim como ocorre em outras classes internas, uma classe interna anônima pode acessar os membros de sua classe de primeiro nível. Porém ela tem acesso mais limitado às variáveis locais do método em que a classe interna anônima é declarada.

Devido a classe interna anônima não ter nome, um objeto desta classe deve ser declarado no ponto em que a classe é declarada, que é o que ocorre na linha 22 da Listagem 4.

! ATENÇÃO

Uma classe interna anônima pode ser declarada da seguinte forma.

```
public class Classe{
    public Classe(){     //construtor
        JButton botao1 = new JButton();
        //criação da classe anônima interna
        botao1.addListener( new ActionListener(){
```

```
            public void actionPerformed(ActionEvent evento){
                ...
            }
        }
}
```

Como podemos observar, a classe anônima possui a implementação do método *actionPerformed* na linha 23. Dentro deste método é feita uma chamada de um método da classe externa chamado *jMenuItem1ActionPerformed*, o qual é definido na linha 47.

Neste método ocorre a instanciação do outro frame e sua chamada nas linhas 48 e 49 fazendo com que o frame do cadastro seja mostrado na tela.

Resumindo, quando usamos o *Netbeans* para criar as *interfaces* e aproveitar os seus recursos de arrastar e soltar componentes, não nos livramos de ter que escrever o que cada evento deve fazer. Porém precisamos entender a forma que o *Netbeans* faz para gerenciar os eventos. Basicamente o que ele faz é:

• Criar uma classe anônima para cada evento, dentro do componente e o associar à chamada do método do *Listener* referente ao evento.

• Implementa o(s) método(s) da *interface* usada (como estamos usando neste caso a *ActionListener*, só temos 1 método para implementar). Neste método, ele faz uma chamada para um método da classe externa.

• Na classe externa o próprio *Netbeans* cria um método para responder ao evento.

```
 1
 2 import javax.swing.JOptionPane;
 3
 4 public class FramePrincipal extends javax.swing.JFrame {
 5     public FramePrincipal() {
 6         initComponents();
 7     }
 8
 9     @SuppressWarnings("unchecked")
10     private void initComponents() {
11
12         jMenuBar1 = new javax.swing.JMenuBar();
13         menuArquivo = new javax.swing.JMenu();
14         jMenuItem1 = new javax.swing.JMenuItem();
```

```
15
16          setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
17          setTitle("Tela Principal");
18
19          menuArquivo.setText("Arquivo");
20
21          jMenuItem1.setText("Cadastrar");
22          jMenuItem1.addActionListener(new java.awt.event.ActionListener() {
23              public void actionPerformed(java.awt.event.ActionEvent evt) {
24                  jMenuItem1ActionPerformed(evt);
25              }
26          });
27          menuArquivo.add(jMenuItem1);
28
29          jMenuBar1.add(menuArquivo);
30
31          setJMenuBar(jMenuBar1);
32
33      //commandos relacionados com layout
34      ...
43
44          pack();
45      }
46
47     private void jMenuItem1ActionPerformed(java.awt.event.ActionEvent evt) {//
GEN-FIRST:event_jMenuItem1ActionPerformed
48          FrameCadastro cadastro = new FrameCadastro();
49          cadastro.setVisible(true);
50
51      }
52
53      public static void main(String args[]) {
54          try {
55
56      //commandos relacionados com log
57      ...
58
71          java.awt.EventQueue.invokeLater(new Runnable() {
72              public void run() {
73                  FramePrincipal programa = new FramePrincipal();
74                  programa.pack();
75                  programa.setVisible(true);
```

```
76             }
77         });
78     }
79
80     private javax.swing.JMenuBar jMenuBar1;
81     private javax.swing.JMenuItem jMenuItem1;
82     private javax.swing.JMenu menuArquivo;
83 }
```

Listagem 4 – Código-fonte do *frame* principal da figura 3.5.

Vamos agora dar uma olhada no código do *frame* que é chamado pelo *frame* principal. Este *frame* tem um *label* com uma figura apenas para mostrar que seria um *frame* para termos um cadastro de um sistema por exemplo. Como já destacamos que o *Netbeans* auxilia bastante na praticidade ao criar interfaces gráficas, ele tem um problema de gerar muitas linhas de código. Já pensou se fôssemos mostrar uma tela de cadastro real aqui? Para simplificar, colocamos a figura então.

O botão presente na tela serve para chamar um método que encerra a aplicação.

```
 1 import javax.swing.JOptionPane;
 2
 3 public class FrameCadastro extends javax.swing.JFrame {
 4     public FrameCadastro() {
 5         initComponents();
 6     }
 7
 8     @SuppressWarnings("unchecked")
 9     private void initComponents() {
10         jLabel1 = new javax.swing.JLabel();
11         jLabel2 = new javax.swing.JLabel();
12         jButton1 = new javax.swing.JButton();
13
14         setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
15         setTitle("Tela de cadastro");
16
17                 jLabel1.setIcon(new  javax.swing.ImageIcon(getClass().getRe-
source("/crud.png")));
18         jLabel1.setText("Exemplo");
19
20         jLabel2.setText("Exemplo de cadastro");
21
```

```
22          jButton1.setText("Sair");
23          jButton1.addActionListener(new java.awt.event.ActionListener() {
24              public void actionPerformed(java.awt.event.ActionEvent evt) {
25                  jButton1ActionPerformed(evt);
26              }
27          });
28
29      //commandos para gerenciamento do layout e posicionamento dos componentes
30
31
54          );
55
56          pack();
57      }
58
59      private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
60          int botao = JOptionPane.YES_NO_OPTION;
61            int resposta = JOptionPane.showConfirmDialog(this, "Deseja sair?",
"Encerramento", botao);
62          if(resposta == 0) {
63              System.exit(0);
64          }
65      }
66
67      public static void main(String args[]) {
68          try {
69
70      //commandos para look and feel e log
71
75          } catch (ClassNotFoundException ex) {
76
77      //tratamento de exceções do frame
78
83          }
84
85          java.awt.EventQueue.invokeLater(new Runnable() {
86              public void run() {
87                  new FrameCadastro().setVisible(true);
88              }
89          });
90      }
```

```
91
92      private javax.swing.JButton jButton1;
93      private javax.swing.JLabel jLabel1;
94      private javax.swing.JLabel jLabel2;
95  }
```

Listagem 5 – Codigo-fonte do *frame* chamado pelo *frame* principal

Observando a listagem 5 percebe-se que novamente temos o uso de classes internas anônimas para o tratamento de eventos. Isto ocorre na linha 23 e possui o mesmo mecanismo que foi descrito na listagem anterior. O *Netbeans* possui este comportamento por padrão e se não for configurado, usará classes internas anônimas para o tratamento de eventos.

Perceba também que por questões de praticidade e objetividade suprimimos algumas linhas da listagem.

Na linha 61 usamos uma classe do pacote *javax.swing.JOptionPane* o qual recebe do usuário a confirmação de uma pergunta. No caso, o usuário precisa responder se deseja ou não encerrar a aplicação e dependendo da resposta, a aplicação é encerrada.

Para finalizar, observe a figura 3.6.

Figura 3.6 – Exemplo com *ItemListener.*

A figura anterior mostra um exemplo contendo um evento que até agora não exploramos: a mudança de estado do *checkbox* presente no item de menu. O programa também é bem simples e executa uma mudança no *label* do item de menu toda vez que o usuário clica no *checkbox* presente na tela.

O código fonte para o exemplo é mostrado na listagem 6.

```
 1 import java.awt.event.ItemEvent;
 2 import java.awt.event.ItemListener;
 3 import java.awt.event.KeyEvent;
 5 import javax.swing.AbstractButton;
 7 import javax.swing.JCheckBoxMenuItem;
 8 import javax.swing.JFrame;
 9 import javax.swing.JMenu;
10 import javax.swing.JMenuBar;
11 import javax.swing.JMenuItem;
12
13 public class TesteItemListener {
14
15   public static void main(final String args[]) {
16     JFrame frame = new JFrame("Exemplo de ItemListener");
17     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18     JMenuBar barraMenu = new JMenuBar();
19
20     JMenu menuArquivo = new JMenu("Arquivo");
21     barraMenu.add(menuArquivo);
22
23     JMenuItem menuItemNovo = new JMenuItem("Novo");
24     menuArquivo.add(menuItemNovo);
25
26     JCheckBoxMenuItem menuItemSexo = new JCheckBoxMenuItem("Sexo");
27     menuArquivo.add(menuItemSexo);
28
29     ItemListener itemListener = new ItemListener() {
30       public void itemStateChanged(ItemEvent evento) {
31         AbstractButton botao = (AbstractButton)evento.getSource();
32         int status = evento.getStateChange();
33         String novoLabel;
34         if (status == ItemEvent.SELECTED)
35           novoLabel = "Menina";
36         else
37           novoLabel = "Menino";
38         botao.setText(novoLabel);
```

```
39        }
40      };
41
42      menuItemSexo.addItemListener(itemListener);
43      frame.setJMenuBar(barraMenu);
44      frame.setSize(350, 250);
45      frame.setVisible(true);
46    }
47 }
```

Listagem 6 – Exemplo com *ItemListener.*

Neste exemplo a *interface ItemListener* foi usada. Perceba que assim como a *ActionListener,* ela só possui 1 método a ser implementado: o *itemStateChanged*, que trata da mudança do estado do *checkbox*. O estado é guardado em um atributo o qual é avaliado pelo método *getStateChange* (linha 32). Se o retorno deste método for igual a *ItemEvent.SELECTED* (uma constante da classe *ItemEvent*), significa que o *checkbox* está selecionado. Veja a linha 34.

Finalizando

Vimos neste capítulo alguns eventos relacionados com janelas, botões e menus. O que podemos resumir sobre eventos até aqui?

Basicamente o Java e sua biblioteca *Swing* usa os eventos contidos na hierarquia da classe *AWTEvent*.

Vimos também que para cada tipo de evento precisamos implementar métodos de *interfaces* específicas as quais fazem parte de um conjunto de classes *Listeners*.

No caso do clique do *mouse*, que é o evento mais utilizado nas aplicações em geral, o *listener* apropriado é a classe *ActionListener* a qual está relacionada com a classe de evento *ActionEvent*. Perceba que todos os exemplos que usamos e vamos usar nos próximos capítulos implementaram esta *interface*. Lembre-se também que todas as vezes que mencionamos "implementar uma *interface*" significa que todos os métodos que a *interface* possui deverão ser escritos nas nossas aplicações, mesmo se estiverem vazios. Vamos estudar mais para frente outro conceito que vai evitar que implementemos todos os métodos.

Além do *ActionListener*, a *interface ItemListener* também só possui um método como vimos no último exemplo.

Veremos no próximo capítulo outros componentes e seus respectivos métodos.

ATIVIDADE

01. Uma vez que agora conhecemos os eventos da classe *JButton* e já vimos rapidamente como escrever em *labels*, que tal você desenvolver uma calculadora bem simples em Java? Não é para fazer nada muito complicado ou cheio de recursos. Os botões serão os números, os operadores básicos (adição, subtração, multiplicação e divisão), o "visor" de resultados pode ser um *label* e não se esqueça de colocar um botão para fazer o cálculo e outro para apagar a operação atual.

Não é difícil, concentre-se, lembre-se do que estudamos e faça com calma. Você consegue!

REFLEXÃO

Ao terminar este capítulo podemos, novamente, pensar a respeito do uso de aplicativos que nos auxiliam a programar como é o caso do *Netbeans*. Será que compensa usar um programa que gera tantas linhas de códigos? Veja que quando não o usamos, nossos programas ficam significativamente menores. É de fato um assunto a ser pensado.

Da mesma forma que devemos pensar a respeito da forma como vamos tratar os eventos: por meio de classes internas ou classes internas anônimas. O que será mais eficiente e produtivo? E mais, o que será mais legível para quem estiver programando? Isto também tem que ser considerado: a manutenção do programa.

Todos esses fatores não são decididos individualmente. São fatores que dentro de uma equipe de desenvolvimento são considerados o tempo todo.

REFERÊNCIAS BIBLIOGRÁFICAS

DEITEL, H. M.; DEITEL, P. J. **Java:** como programar. 6ª. ed. Rio de Janeiro: Bookman, 2005.

HORSTMANN, C. S.; CORNELL, G. **Core Java.** Rio de Janeiro: Pearson Education, 2010.

HUBBARD, J. R. **Programação com Java 2**. Rio de Janeiro: Bookman, 2006.

ORACLE CORPORATION. **The Java Tutorials.** Oracle Java Documentation, 01 mar. 2015. Disponível em: <https://docs.oracle.com/javase/tutorial>. Acesso em: 1 mar. 2016.

ORACLE CORPORATION. **Java Platform, Standard Edition 7.** API Specification, 2016. Disponível em: <https://docs.oracle.com/javase/7/docs/api/>. Acesso em: 01 mar. 2016.

4

Tratamento de eventos para *interfaces* gráficas II

Tratamento de eventos para *interfaces* gráficas II

Vimos no capítulo 3 alguns exemplos envolvendo eventos relacionados a componentes como botões e janelas. Neste capítulo vamos continuar nossos estudos neste assunto, porém com componentes diferentes, que poderão deixar suas aplicações ainda mais interessantes e interativas.

Neste capítulo vamos estudar um pouco sobre validação de textos. Você deve navegar diariamente na internet e frequentemente se depara com cadastros e outras telas nas quais você tem que digitar um texto e este ser validado. Em telas de *login* isto ocorre na maioria das vezes. O sistema tem que validar sua entrada para poder permitir o *login* na aplicação. Em programas para *desktop* o processo é o mesmo, e a linguagem Java oferece um ótimo suporte para este processo.

OBJETIVOS

Neste capítulo vamos estudar:

- Eventos de texto e validações;
- Manipulação de eventos de listas;
- Manipulação de eventos de caixas de seleção;
- Modelos de eventos em tabelas e teclas de atalho.

Eventos de Textos

Quando mencionamos eventos de texto em Java estamos nos referindo indiretamente a duas classes principais: a *TextEvent* e a *TextListener* (na verdade esta é uma *interface* e como tal, quando usada, devemos implementar seu único método).

A estrutura hierárquica da classe *TextEvent* é a seguinte:

```
java.lang.Object
java.util.EventObject
java.awt.AWTEvent
java.awt.event.TextEvent
```

Logo, é mais uma classe que herda da família do pacote AWT. E esta informação é fundamental para entender os eventos relacionados com a classe *JTextFieldComponent*.

Esta classe possui somente 1 construtor e somente 1 método:

```
-   TextEvent(Object origem, int id)
-   paramString(), o qual retorna uma string que identifica o evento
```

A *interface TextListener* contém somente um método o qual deve ser implementado toda vez que for usada:

```
-   textValueChanged(TextEvent e): o qual é invocado quando o valor do texto foi
modificado. Perceba que o método recebe como parâmetro um objeto TextEvent
```

Vamos parar e pensar em quais tipos de eventos podemos ter em uma caixa de texto. Sim, há muitos, mas qual é o mais usado?

Sem dúvida, o evento que mais é usado em uma caixa de texto é quando o usuário digita alguma coisa dentro dela. A digitação muda o estado do componente, deixando-o alterado. Da mesma forma, para digitar alguma coisa, é necessário que o usuário aperte e solte uma tecla do teclado. E essas informações são importantes.

Novamente destacamos a importância de se conhecer melhor a API dos componentes que estamos trabalhando nas nossas *interfaces*. Segundo a documentação do *JTextField* (Oracle Corporation, 2016), nos componentes baseados na classe *JTextComponent*, as mudanças que ocorrem dentro da caixa de texto são informadas para o sistema operacional por meio de eventos no Documento, captados pelos *DocumentListeners*. Vamos explicar melhor. Veja a figura 4.1 e os passos que ocorrem na captura de um evento.

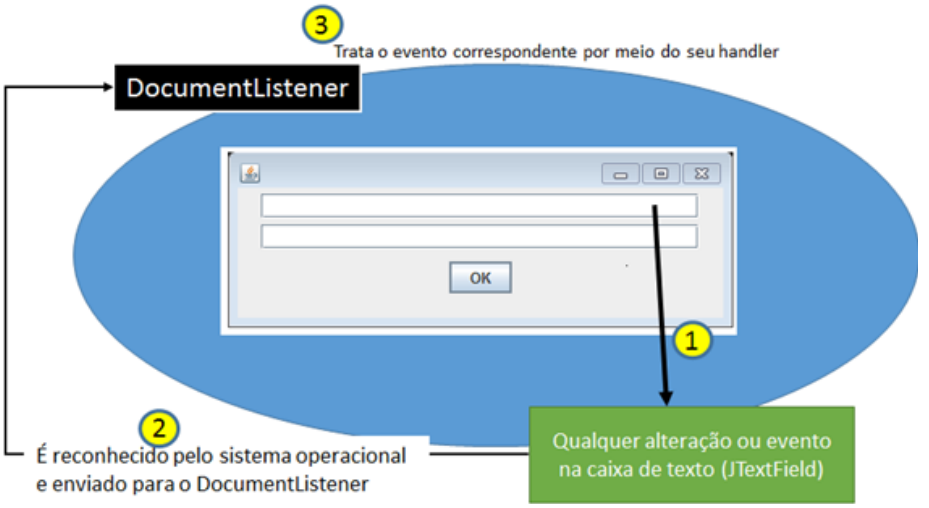


Figura 4.1 – Fases da captura de um evento em uma caixa de texto (*JTextField*).

Vamos exemplificar com um programa que possui a seguinte aparência:

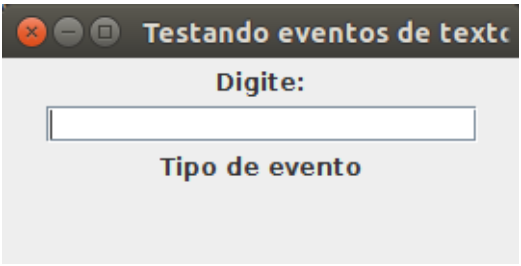


Figura 4.2 – Programa para teste de eventos de texto.

O programa consiste em um *JTextField* no qual o usuário vai digitar o texto que desejar. Também possui 2 *JLabels*, um apenas informativo e outro vai ser atualizado conforme os eventos acontecerão na caixa de texto.

O código fonte para o programa da figura 4.2 está mostrado na listagem 1:

```
1 import java.awt.FlowLayout;
2 import javax.swing.JFrame;
3 import javax.swing.JLabel;
4 import javax.swing.JTextField;
5 import javax.swing.event.DocumentEvent;
6 import javax.swing.event.DocumentListener;
```

```
 7
 8 class TestaTexto extends JFrame {
 9     private JLabel labelEvento, label1;
10     static private JTextField texto;
11
12     public TestaTexto(){
13         super("Testando eventos de texto");
14         setLayout(new FlowLayout());
15         label1 = new JLabel("Digite:");
16         add(label1);
17         texto = new JTextField(20);
18         add(texto);
19         labelEvento = new JLabel("Tipo de evento");
20         add(labelEvento);
21
22         TextoHandler handler = new TextoHandler();
23         texto.getDocument().addDocumentListener(handler);
24     }
25
26     private class TextoHandler implements DocumentListener {
27         @Override
28         public void insertUpdate(DocumentEvent e) {
29             labelEvento.setText("insertUpdate");
30         }
31
32         @Override
33         public void removeUpdate(DocumentEvent e) {
34             labelEvento.setText("removeUpdate");
35         }
36
37         @Override
38         public void changedUpdate(DocumentEvent e) {
39             System.out.println("changedUpdate");
 40             }
41     }
42
43     public static void main(String args[]) {
44         TestaTexto programa = new TestaTexto();
45         programa.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
46         programa.setSize(270,110);
47         programa.setVisible(true);
48   }
49 }
```

Listagem 1 – Código fonte do programa que testa eventos de texto.

Ao executar o programa, assim que o usuário digitar alguma coisa na caixa de texto, um evento *insertUpdate* será disparado e capturado pelo *handler TextoHandler*. Esta classe interna implementa a *interface DocumentListener* a qual possui os seguintes métodos:

INSERTUPDATE (DOCUMENTEVENT E)	Este evento ocorre toda vez que um novo caractere é inserid.
REMOVEUPDATE (DOCUMENTEVENT E)	Ocorre quando um caractere é removido.
CHANGEDUPDATE (DOCUMENTEVENT E)	Este evento ocorre quando houve alguma mudança nos atributos do componente.

A execução do programa gera as seguintes mensagens no *labelEvento*:

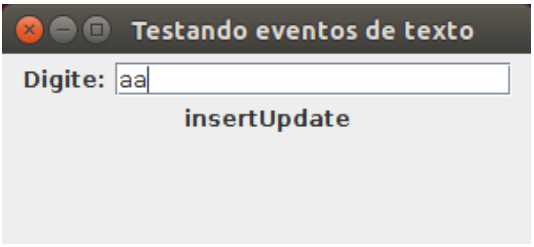


Figura 4.3 – Usuário digitou algum caractere na caixa de texto, o *label* mostra o evento *insertUpdate*.

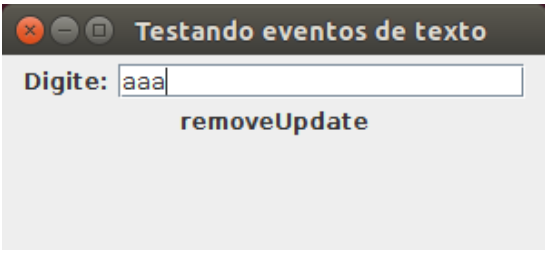


Figura 4.4 – Usuário apagou algum caractere na caixa de texto, o *label* mostra o evento *removeUpdate*.

Observando a listagem 1 percebemos que a estrutura do programa não é muito diferente do que já estudamos no capítulo anterior. A principal diferença é a

interface que foi implementada e a necessidade de inserir código em todos os seus métodos.

Outra diferença é que neste caso o componente no qual o evento é gerado não é exatamente o componente que estamos estudando: estamos estudando o *JTextField* e quem gera o evento é o *Document*. Isto ocorre porque o *JTextField* não possui um *handler* de eventos direto. Outra possibilidade que poderíamos ter é usar um componente *TextField*, do pacote *java.awt* e usar os *handlers* de evento dele. Isto poderia ser feito devido à simplicidade deste componente específico.

Nas linhas 38 a 40 do código, optamos por mostrar no console quando um evento *changedUpdate* for disparado, mas isto não vai ocorrer no nosso programa, pois este evento não será emitido.

Poderíamos colocar um *ActionEvent* associado ao *JTextField*. Este evento responderia a uma ação muito comum em componentes de texto: o pressionamento da tecla *Enter* quando o usuário termina de digitar algo na caixa de texto. Sendo assim, que mudanças devem ser feitas no código fonte da Listagem 1? Pense um pouco, reanalise o que foi estudado e a listagem antes de prosseguir com a leitura.

EXEMPLO

1. Em primeiro lugar precisamos inserir um objeto que vai capturar o evento da tecla *Enter* (*action*). No código já temos uma linha (23) que captura o evento no *Document*, mas como faríamos para o evento do componente? É simples. A resposta pode ser dada fazendo outra pergunta: como fazíamos anteriormente nos outros exemplos? Não usamos o método *addActionListener* e passamos o *handler* como parâmetro? Foi isso! Então vamos adicionar mais uma linha e fazer com que o componente responda a este evento! Assim:

```
texto.addActionComand(handler);
```

2. Uma vez que configuramos o componente para tratar outro evento, qual o próximo passo natural? Implementar a sua interface não é? Precisamos criar outra classe interna para isso? Não necessariamente. Em Java é comum implementar mais de uma interface na mesma classe. Desta forma, nós poderíamos aproveitar o objeto já criado para tratar os eventos do componente (o *handler*, da linha 22). A linha 26 passaria a ser assim:

```
    private class TextoHandler implements DocumentListener, ActionListener {
```

3. O próximo passo é implementar todos os métodos da nova interface adicionada:

```
public void actionPerformed(ActionEvent e) {
    labelEvento.setText("action");
}
```

Com estas pequenas modificações, a execução do programa vai gerar as seguintes telas mostradas na figura 4.5:

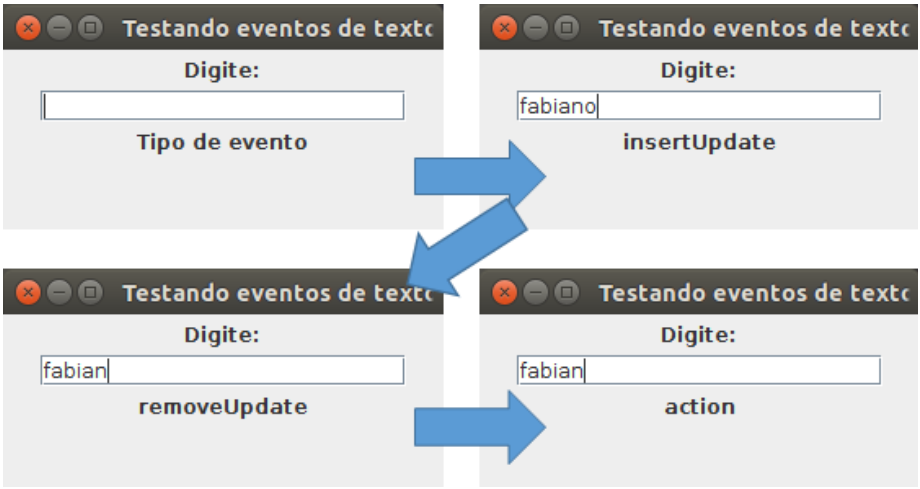


Figura 4.5 – Execução do programa com o evento *action*.

Bem, o mecanismo principal para tratar eventos de texto foi apresentado. Agora é sua vez de acrescentar mais eventos e usar sua criatividade para poder incrementar suas *interfaces*. Vamos estudar outro componente muito presente nas *interfaces:* as listas.

Eventos de listas de seleção

Sabemos que uma lista (*JList* do pacote *javax.swing*) é um componente capaz de apresentar dados agrupados em uma ou mais colunas os quais podem ser selecionados, ou não. As listas podem possuir vários itens e quando isso ocorre, posicionamos elas em *Scroll Panels* (painéis de rolagem).

Vamos mostrar por meio de um exemplo como podemos estudar o tratamento de eventos em listas de seleção. Observe as figuras a seguir.

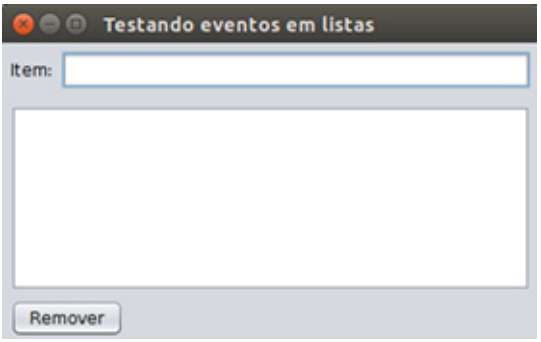


Figura 4.6 – Mostra o nosso exemplo de demonstração dos eventos em listas de seleção. Trata-se de um *frame* com um *label*, uma caixa de texto, uma lista e um botão.

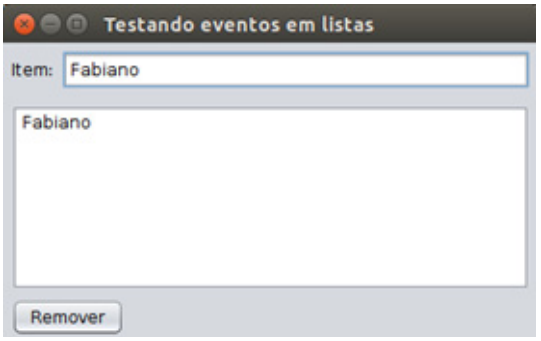


Figura 4.7 – Temos o evento action da caixa de texto sendo disparado. O usuário digita algum item na caixa de texto e o que for digitado é inserido na lista após o pressionamento da tecla *Enter.*

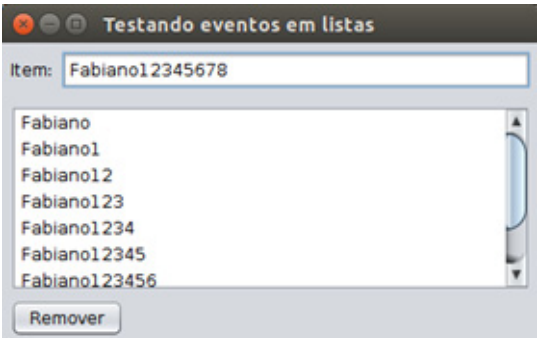


Figura 4.8 – Mostra a lista com vários itens inseridos. Perceba que aparece uma barra de rolagem vertical automaticamente quando a quantidade de itens ultrapassa o tamanho da lista. Perceba que na lista temos o item "Fabiano123" (ele será usado daqui a pouco).

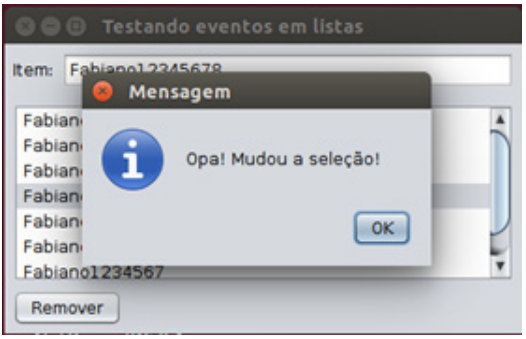


Figura 4.9 – Temos um evento genuíno da lista sendo disparado. Trata-se do evento *valueChanged*. Este evento ocorre quando algum item na lista é selecionado e é "escutado" pelo *ListSelectionListener*. Na verdade, é o único *listener* relacionado com as listas. Pensando bem, não dá para ter outros eventos além de selecionar algum item na lista, não é?

Figura 4.10 – Mostra o resultado do evento *action* aplicado ao botão "Remover" do *frame*. Após selecionado um item, quando o botão é acionado, o item é removido da lista.

O código fonte que gera as figuras anteriores está mostrado na listagem 2:

```
1   import javax.swing.DefaultListModel;
 2 import javax.swing.JOptionPane;
 3 import java.awt.event.ActionEvent;
 4 import javax.swing.event.ListSelectionEvent;
 5 import java.awt.event.ActionListener;
 6 import javax.swing.JFrame;
 7 import javax.swing.JLabel;
 8 import javax.swing.JScrollPane;
 9 import javax.swing.JButton;
10 import javax.swing.JTextField;
```

```
11 import javax.swing.JList;
12 import javax.swing.ListSelectionModel;
13 import java.awt.FlowLayout;
14 import javax.swing.event.ListSelectionListener;
15
16
17 public class FrameLista extends JFrame {
18     private final DefaultListModel modelo;
19     private JButton jButton1;
20     private JLabel jLabel1;
21     private JList jList1;
22     private JTextField jTextField1;
23
24     public FrameLista() {
25         jLabel1 = new JLabel("Item:");
26         jTextField1 = new JTextField(15);
27         modelo = new DefaultListModel();
28         jList1 = new JList(modelo);
29         jButton1 = new JButton("Remover");
30         setTitle("Testando eventos em listas");
31
32         jTextField1.addActionListener(new ActionListener() {
33             public void actionPerformed(ActionEvent evt) {
34                 jTextField1ActionPerformed(evt);
35             }
36         });
37
38         jList1.setModel(jList1.getModel());
39         jList1.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
40         jList1.addListSelectionListener(new ListSelectionListener() {
41             public void valueChanged(ListSelectionEvent evt) {
42                 jList1ValueChanged(evt);
43             }
44         });
45
46         jButton1.addActionListener(new ActionListener() {
47             public void actionPerformed(ActionEvent evt) {
48                 jButton1ActionPerformed(evt);
49             }
50         });
51
52         setLayout(new FlowLayout());
```

```
53          add(jLabel1);
54          add(jTextField1);
55          add(new JScrollPane(jList1));
56          add(jButton1);
57      }
58
61      private void jTextField1ActionPerformed(ActionEvent evt) {
62          modelo.addElement(jTextField1.getText());
63          jList1.setModel(modelo);
64      }
65
66      private void jButton1ActionPerformed(ActionEvent evt) {
67          int indice = jList1.getSelectedIndex();
68          modelo.remove(indice);
69      }
70
71      private void jList1ValueChanged(ListSelectionEvent evt) {
72          JOptionPane.showMessageDialog(this, "Opa! Mudou a seleção!");
73      }
74
75      public static void main(String args[]) {
76          FrameLista programa = new FrameLista();
77          programa.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
78          programa.setSize(320,100);
79          programa.setVisible(true);
80      }
81  }
```

Listagem 2 – Eventos em listas de seleção.

Em relação aos eventos demonstrados no exemplo, o primeiro tipo de evento que temos contato está mostrado entre as linhas 32 a 35. Trata-se do *action*, monitorado pelo *ActionListener*. Este evento, como vimos, vai responder à ação do acionamento da tecla *Enter* quando o foco estiver na caixa de texto. Deste modo, o que estiver escrito na caixa, será transferido para a lista (veja o tratamento do evento nas linhas 61 a 64).

Outro evento ocorre entre as linhas 39 a 44. Este evento responde à ação (evento *valueChanged*) quando o usuário ou o próprio programa seleciona outro elemento da lista. A reação está tratada entre as linhas 71 a 73 e para simplificar mostramos uma caixa de diálogo para demonstrar que houve uma mudança no item selecionado.

O último evento que foi usado neste *frame* refere-se ao acionamento do botão "Remover". Quando acionado, o programa pega o item que está selecionado na lista e o remove dela nas linhas 66 a 69 (veja o método remove, aplicado ao objeto modelo: modelo.remove(índice)). O programa sabe o item a ser removido por meio do seu índice.

Eventos de caixas de seleção

Em Java é possível usar alguns componentes relacionados com "caixas de seleção" ou selection boxes. Descrevemos a seguir 3 dos mais importantes componentes desta categoria, mais frequentemente usados: *JComboBox, JRadioButton e JCheckBox.*

O *JComboBox* certamente é um componente que você vai usar bastante e vamos nos concentrar nele neste tópico. Um *JComboBox* permite que um usuário escolha um item dentre uma lista por meio de duas formas: digitando o item escolhido ou clicando em um menu dropdown do próprio componente. Certamente você já viu ou ouviu a palavra combo em alguns bares e lanchonetes, não é? Combo vem de combination, ou seja, combinação. Um combo box é uma caixa que combina uma caixa de texto e uma lista!

O legal do *JComboBox* é que ele pode armazenar um grande número de elementos em um pequeno espaço na *interface*. E isso é importante, principalmente em telas que possuem muitos elementos. Os *JComboBox* são preferíveis às listas porque elas ocupam muito espaço na tela.

E sobre os eventos? Que eventos um *JComboBox* poderia responder? Analisando a API do Java, temos que o *JComboBox* faz parte da seguinte hierarquia:

```
java.lang.Object
java.awt.Component
java.awt.Container
javax.swing.JComponent
javax.swing.JComboBox
```

Ou seja, os *JComboBoxes* também vão responder a eventos como *ItemEvents* assim como *JCheckboxes* e *JRadioButtons*. Vamos mostrar um exemplo para apresentar como os eventos podem ser tratados em um *JComboBox*.

O programa exemplo é simples e vai gerar as seguintes telas, mostradas na figura 4.11:

Figura 4.11 – Eventos em caixas de seleção.

O programa possui uma lista no *JComboBox* com alguns nomes de arquivos de figuras representando as bandeiras respectivas.

O código fonte para o programa está mostrado a seguir, na listagem 3.

```
1   import java.awt.FlowLayout;
2   import java.awt.event.ItemListener;
3   import java.awt.event.ItemEvent;
4   import javax.swing.JFrame;
5   import javax.swing.JLabel;
6   import javax.swing.JComboBox;
7   import javax.swing.Icon;
8   import javax.swing.ImageIcon;
9
10  public class FrameComboBox extends JFrame {
11     private final JComboBox<String> comboImagens;
12     private JLabel label;
13
14      private static final String[] nomes = {"brasil.png", "espanha.png","eua.
png","franca.png","italia.png","reinounido.png","ue.png"};
15     private final Icon[] figuras = {
16        new ImageIcon(getClass().getResource(nomes[0])),
17        new ImageIcon(getClass().getResource(nomes[1])),
18        new ImageIcon(getClass().getResource(nomes[2])),
```

```
19          new ImageIcon(getClass().getResource(nomes[3])),
20          new ImageIcon(getClass().getResource(nomes[4])),
21          new ImageIcon(getClass().getResource(nomes[5]))};
22
23      public FrameComboBox(){
24          super("Testando os eventos no JComboBox");
25          setLayout(new FlowLayout());
26
27          comboImagens = new JComboBox<String>(nomes);
28          comboImagens.setMaximumRowCount(3);
29
30          Handler handler = new Handler();
31
32          comboImagens.addItemListener(handler);
33          add(comboImagens);
34          label = new JLabel(figuras[0]);
35          add(label);
36      }
37
38      private class Handler implements ItemListener{
39          @Override
40          public void itemStateChanged(ItemEvent evento) {
41              if (evento.getStateChange() == ItemEvent.SELECTED)
42                  label.setIcon(figuras[comboImagens.getSelectedIndex()]);
43          }
44      }
45      public static void main(String[] args) {
46          FrameComboBox programa = new FrameComboBox();
47          programa.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
48          programa.setSize(350, 150);
49          programa.setVisible(true);
50      }
51  }
```

Listagem 3 – Código fonte dos eventos em *JComboBox*.

Ao observar o código fonte anterior, percebemos que ele também não possui uma estrutura diferente dos demais exemplos que mostramos. O que varia é o evento que é acionado e a ação respondida. No caso, como já havíamos comentado, o método *itemStateChanged*, linha 40, trata o evento nas linhas 41 e 42, dentro da classe interna *Handler*, linhas 38 a 44.

Poderíamos ter colocado outros eventos, mas este em especial já é suficiente para mostrar o que pode ser feito com os *JComboBoxes*.

No programa é possível escolher o país por meio do mouse ou simplesmente digitando o seu nome dentro da caixa de texto do *JComboBox*. Como a lista é pequena e existem apenas 2 países com a mesma letra inicial (Espanha e Estados Unidos), só o ato do usuário digitar a primeira letra, o evento já é disparado e a respectiva figura é mostrada no *label*. Lembre-se que podemos criar *labels* somente com figuras, não é? Vimos isto no Capítulo 2. Neste exemplo fizemos uso desta forma de instanciação.

O programa é interessante para nos ajudar a entender como podemos montar uma estrutura para manipular imagens.

Neste caso foi feito o seguinte:

- Foi criado um vetor, chamado nomes (linha 14), contendo os arquivos das imagens. Estas imagens devem, obrigatoriamente, fazer parte do *build path* do programa;
- Os elementos do vetor nomes são apenas *strings* para o Java. O compilador ainda não sabe que se tratam de imagens, sendo assim, objetos do tipo *Icon* são criados nas linhas 15-21. Os objetos são criados de acordo com os seus nomes de arquivos e também armazenados em um vetor chamado "figuras" (nome sugestivo, né?);
- Na linha 23 temos o construtor da classe, da maneira tradicional como temos trabalhado nos exemplos;
- Nas linhas 27 e 28 temos a criação de um *JComboBox* que, assim como o *JList*, obedece a um modelo. Neste caso, criamos um *JComboBox* cujos itens são objetos da classe *String* (os nomes dos arquivos). Logo em seguida, na linha 28 é configurado que o *JComboBox* terá 3 linhas visualizadas na tela.

CONCEITO

Build path: como o nome diz, é o caminho de construção de um programa em Java. No caso, as figuras que pertencem a uma aplicação devem estar dentro deste caminho. O *build path* pode ser configurado facilmente em programas como o *Netbeans*, Eclipse e outros. Sem o uso destas ferramentas, as figuras podem ser guardadas no mesmo diretório dos programas fonte.

Simples, não é? Talvez se nós não tivéssemos apresentado o código fonte do programa você teria dificuldades para estruturá-lo. Esperamos que não, mas como esta pode ser a primeira vez que você entrou em contato com este assunto, a dificuldade em criar um programa desta forma é mais do que natural. Mais uma vez ressaltamos a grande importância do estudo da orientação a objetos para o pleno entendimento destes exemplos.

Vamos agora tratar de eventos em um componente bem importante e muito usado em aplicações, principalmente naquelas que acessam bancos de dados.

Eventos em tabelas e teclas de atalho

Como já vimos anteriormente, o uso de tabelas em java, especialmente no pacote *Swing*, é feito pela classe *JTable*.

Lembre-se que as tabelas em Java apenas mostram dados. Elas não contêm *buffer* ou cache algum dos dados contidos nelas. Isto é importante você saber e lembrar sempre. Ou seja, uma *JTable* não serve para fazer persistência de dados. Assim que o programa for finalizado, os dados que a *JTable* possuíam são perdidos.

AUTOR

A persistência de dados é o conceito dado para quando se deseja gravar os dados atuais em algum dispositivo, na maioria das vezes em um banco de dados. Então a persistência consiste em tirar uma fotografia dos dados atuais, gravá-los no banco de dados e posteriormente recuperá-los (recuperar a fotografia) e apresentar estes dados em algum componente: em um *JTable*, por exemplo.

Os dados da *JTable* são obtidos por meio de um modelo, que é implementado pela classe *AbstractTableModel*. Já falamos sobre isso rapidamente no Capítulo 2, porém agora é hora de vermos com um pouco mais de detalhes. Logo, o "*look and feel*" é implementado na *JTable* e os dados são guardados em um objeto da classe *AbtractTableModel*.

Entretanto, não é obrigatório instanciar um objeto da classe *AbstractTableModel*. É possível instanciar uma *JTable* por meio de um vetor de textos e uma matriz de objetos. Neste caso, o vetor conterá os títulos das colunas e a matriz conterá o "recheio" da tabela. Desta forma, os dados serão armazenados em um modelo padrão o qual é criado automaticamente pelo Java.

Toda instância de *JTable* usa um modelo de tabela para gerenciar os dados contidos na tabela, certo? Já tratamos disso anteriormente. Um objeto de modelo de tabela deve implementar a interface *TableModel*. Como já foi dito também, se o programador não providenciar um objeto desta classe, a *JTable* automaticamente cria uma instância da classe *DefaultTableModel*. A figura 4.12 mostra o relacionamento entre estes conceitos citados.

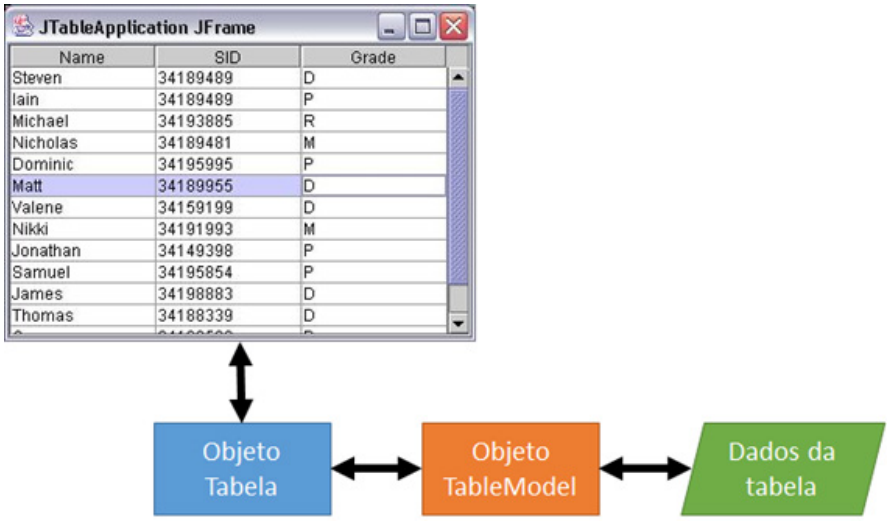


Figura 4.12 – Relacionamento entre a *JTable*, *TableModel* e seus dados. Disponível em: <www.guj.com.br>.

Vamos fazer um exemplo para poder ilustrar esses conceitos. O exemplo é bem simples mas é importante você notar o quanto ele pode te inspirar a criar aplicações conectadas com banco de dados.

Figura 4.13 – Exemplo de aplicação com *JTable*.

A figura 4.13 mostra a tela principal do nosso exemplo. Trata-se de um *JFrame* contendo 3 *JLabels, 3 JTextFields, 3 JButtons* e a *JTable*. A ideia é usar as caixas de texto para ler os dados digitados pelo usuário e por meio dos botões controlar o acesso aos dados para inserir e manipular as informações da tabela. Veja as figuras a seguir e suas explicações.

Figura 4.14 – O usuário digita os dados nas caixas de texto e após o acionamento do botão, os dados vão para a tabela.

Figura 4.15 – Esta é o estado da tabela após algumas inserções. Veja que aparece uma barra de rolagem quando a tabela fica maior que o frame.

Figura 4.16 – Quando o usuário clica em alguma linha da tabela, os dados da linha selecionada são copiados para as respectivas caixas de texto.

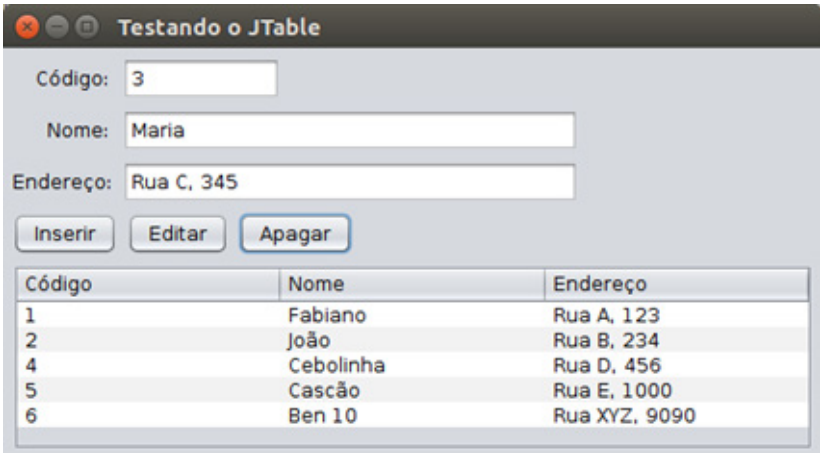


Figura 4.17 – Uma vez que os dados selecionados foram para as caixas de texto, o usuário pode apagar a linha selecionada como ocorre nesta figura. Poderíamos ter caprichado um pouco mais e fazer com que os dados da caixa de texto sejam apagados também, mas não fizemos isso propositalmente para você verificar o estado do frame após o acionamento do botão Apagar.

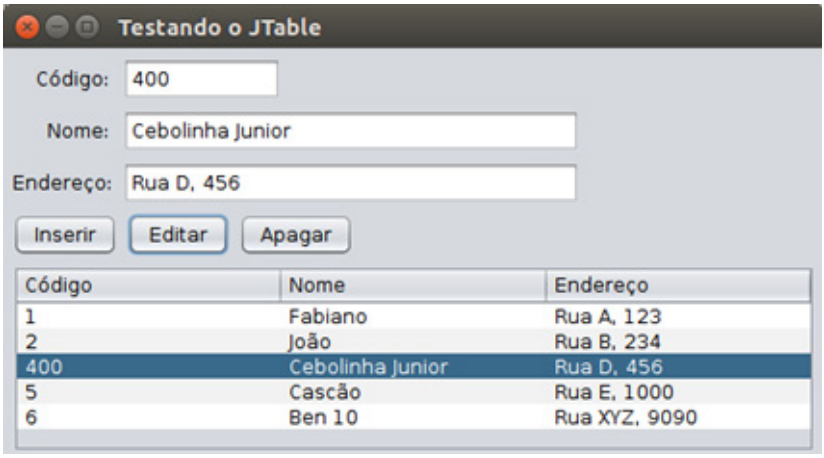


Figura 4.18 – Ao clicar em uma linha, o usuário também pode editar os dados e após estarem prontos, clicar no botão Editar para que eles sejam atualizados na tabela, como vemos nesta figura.

Este exemplo foi feito no *Netbeans* para melhor posicionamento dos componentes no *frame*, portanto, o código vai ficar um pouco grande, observe.

```
 1
 2 import javax.swing.table.DefaultTableModel;
 3
 4 public class TesteJTable extends javax.swing.JFrame {
 5     DefaultTableModel modelo;
 6
 7     public TesteJTable() {
 8         initComponents();
 9         modelo = (DefaultTableModel) tabClientes.getModel();
10     }
11
12     @SuppressWarnings("unchecked")
13     private void initComponents() {
14         jLabel1 = new javax.swing.JLabel();
15         jLabel2 = new javax.swing.JLabel();
16         jLabel3 = new javax.swing.JLabel();
17         tfCodigo = new javax.swing.JTextField();
18         tfNome = new javax.swing.JTextField();
19         tfEndereco = new javax.swing.JTextField();
20         bInserir = new javax.swing.JButton();
21         bEditar = new javax.swing.JButton();
22         bApagar = new javax.swing.JButton();
23         jScrollPane1 = new javax.swing.JScrollPane();
24         tabClientes = new javax.swing.JTable();
25
26         setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
27         setTitle("Testando o JTable");
28
29         jLabel1.setText("Código:");
30         jLabel2.setText("Nome:");
31         jLabel3.setText("Endereço:");
32
33         bInserir.setText("Inserir");
34         bInserir.addActionListener(new java.awt.event.ActionListener() {
35             public void actionPerformed(java.awt.event.ActionEvent evt) {
36                 bInserirActionPerformed(evt);
37             }
38         });
39
40         bEditar.setText("Editar");
```

```
41          bEditar.addActionListener(new java.awt.event.ActionListener() {
42              public void actionPerformed(java.awt.event.ActionEvent evt) {
43                  bEditarActionPerformed(evt);
44              }
45          });
46
47          bApagar.setText("Apagar");
48          bApagar.addActionListener(new java.awt.event.ActionListener() {
49              public void actionPerformed(java.awt.event.ActionEvent evt) {
50                  bApagarActionPerformed(evt);
51              }
52          });
53
54          tabClientes.setModel(new javax.swing.table.DefaultTableModel(
55              new String [] {
56                  "Código", "Nome", "Endereço"
57              }
58          ));
59          tabClientes.addMouseListener(new java.awt.event.MouseAdapter() {
60              public void mouseClicked(java.awt.event.MouseEvent evt) {
61                  tabClientesMouseClicked(evt);
62              }
63          });
64          jScrollPane1.setViewportView(tabClientes);
65
66                      javax.swing.GroupLayout   layout   =   new   javax.swing.
GroupLayout(getContentPane());
67          getContentPane().setLayout(layout);
    //COMANDOS PARA GERENCIAMENTO DO LAYOUT AUTOMÁTICO DO NETBEANS
124         pack();
125     }
126
127     private void bInserirActionPerformed(java.awt.event.ActionEvent evt) {
128          modelo.insertRow(modelo.getRowCount(), new Object[]{tfCodigo.getTex-
t(),tfNome.getText(),tfEndereco.getText()});
129     }
130
131     private void bEditarActionPerformed(java.awt.event.ActionEvent evt) {
132           modelo.setValueAt(tfCodigo.getText(), tabClientes.getSelectedRow(),
0);
133         modelo.setValueAt(tfNome.getText(), tabClientes.getSelectedRow(), 1);
```

```
134         modelo.setValueAt(tfEndereco.getText(), tabClientes.getSelectedRow(),
2);
135     }
136
137     private void tabClientesMouseClicked(java.awt.event.MouseEvent evt) {
138                 tfCodigo.setText(String.valueOf(modelo.getValueAt(tabClientes.
getSelectedRow(),0)));
139                  tfNome.setText(String.valueOf(modelo.getValueAt(tabClientes.
getSelectedRow(),1)));
140              tfEndereco.setText(String.valueOf(modelo.getValueAt(tabClientes.
getSelectedRow(),2)));
141     }
142
143     private void bApagarActionPerformed(java.awt.event.ActionEvent evt) {
144         modelo.removeRow(tabClientes.getSelectedRow());
145     }
146
147     public static void main(String args[]) {
148         try {
    //COMANDOS AUTOMÁTICOS DE INICIALIZAÇÃO DO JFRAME
165         java.awt.EventQueue.invokeLater(new Runnable() {
166             public void run() {
167                 new TesteJTable().setVisible(true);
168             }
169         });
170     }
171
172     private javax.swing.JButton bApagar;
173     private javax.swing.JButton bEditar;
174     private javax.swing.JButton bInserir;
175     private javax.swing.JLabel jLabel1;
176     private javax.swing.JLabel jLabel2;
177     private javax.swing.JLabel jLabel3;
178     private javax.swing.JScrollPane jScrollPane1;
179     private javax.swing.JTable tabClientes;
180     private javax.swing.JTextField tfCodigo;
181     private javax.swing.JTextField tfEndereco;
182     private javax.swing.JTextField tfNome;
183 }
```

Listagem 4 – Código-fonte do exemplo de *JTable*.

Ao analisar o código gerado pelo *Netbeans* vemos que a quantidade de linhas é bem grande para um problema relativamente simples de ser resolvido. O que temos que considerar é que o posicionamento dos componentes usando o arrastar e soltar gera essa quantidade de linhas.

A *JTable* usada no exemplo usa o *DefaultTableModel* como modelo, ou seja, usa um vetor contendo *Strings* nos títulos das colunas e uma matriz de *Object[]* nas linhas. Inicialmente o modelo é setado com 3 colunas e nenhuma linha.

! ATENÇÃO

Métodos e atributos usados

No exemplo foram usados alguns métodos e atributos para manipular o model da JTable

INSERTROW()	Este método tem como função inserir uma linha ao modelo em uma determinada posição do modelo. Possui como parâmetro a posição e o que vai ser inserido. Veja que usamos o método *getRowCount()* do objeto modelo. Este método retorna a quantidade de linhas existentes no modelo. Como a linha inicial é 0, a linha a ser inserida é igual à quantidade de linhas do modelo. Por exemplo, se temos uma linha apenas no modelo, esta linha terá índice 0. Logo se quisermos inserir uma nova linha, ele deverá ser inserida na posição 1, que é igual à quantidade de linhas atual.
SETVALUEAT()	Este método é responsável por inserir um valor em uma determinada posição. Ele possui 3 parâmetros: o valor a ser inserido, a linha a ser usada, e a coluna. O método *getText()* do *textField* retorna o texto presente no momento no objeto. O método *getSelectedRow()* do modelo , usado no segundo parâmetro, retorna a linha atualmente selecionada do modelo (a seleção foi feita pelo clique do mouse, lembra?). O último parâmetro configura a coluna a ser usada para a inserção do valor: 0, 1 ou 2 (lembre-se que o índice sempre começa com 0). Logo, o método *setVaueAt()* é usado no botão Editar.
REMOVEROW()	Como é de se esperar, este método é usado no botão Apagar e funciona como o "inverso" do método *insertRow()*. Ele é mais simples, pois só apresenta um parâmetro: a linha atualmente selecionada do modelo.

Quanto aos eventos, percebemos que não são tão diferentes do que já estudamos anteriormente. O que é diferente neste caso, são os métodos e atributos que pertencem à *JTable* e a forma como eles são usados.

Basicamente, temos os eventos action gerados pelos botões e mostrados nas linhas 34 a 40 (botão Inserir), 41 a 45 (botão Editar) e 48 a 52 (botão Apagar). Sabemos que o Netbeans implementa as *interfaces* de eventos por meio de classes internas anônimas (linhas 34, 41 e 48). Porém veja só o que o *Netbeans* faz para o evento de *click* do *mouse* na linha 59: ele usa uma classe *Adapter*, ou também chamado de adaptador de evento.

Adaptadores de eventos

Algumas *interfaces* de listeners contém mais do que um método. Por exemplo, a *interface MouseListener* contém 5 métodos: *mousePressed, mouseReleased, mouseEntered, mouseExited e mouseClicked*. Como estamos falando de uma *interface*, então mesmo se a sua aplicação só necessite tratar o clique do *mouse*, é obrigatório que você implemente todos os outros quatro métodos. Neste caso, é possível deixar o corpo dos outros métodos em branco como no exemplo a seguir.

Um exemplo que implementa uma interface de listener diretamente.

```
public class MinhaClasse implements MouseListener {
    ...
        objeto.addMouseListener(this);
    ...
    /* Definição de método vazia. */
    public void mousePressed(MouseEvent e) {
    }

    /* Definição de método vazia. */
    public void mouseReleased(MouseEvent e) {
    }

    /* Definição de método vazia. */
    public void mouseEntered(MouseEvent e) {
    }

    /* Definição de método vazia. */
    public void mouseExited(MouseEvent e) {
    }
```

```
    public void mouseClicked(MouseEvent e) {
        ...//A implementação do evento do listener vai aqui...
    }
}
```

Como podemos perceber, este conjunto de métodos vazios prejudica a leitura do código e sua manutenção. Para evitar a implementação de métodos vazios, a API geralmente inclui uma classe **adaptadora** para cada *interface* que possui mais de um método.

Para usar um adaptador, você cria uma subclasse e sobrescreve somente os métodos de interesse ao invés de implementar todos os métodos da *interface*. O código a seguir mostra um exemplo da modificação do código anterior para estender *MouseAdapter*. Estendendo *MouseAdapter*, ele herda as definições vazias de todos os cinco métodos que *MouseListener* contém.

```
/*
 * Um exemplo de classe adaptadora ao invés de implementar uma interface de listener
 */
public class MinhaClasse extends MouseAdapter {
    ...
        objeto.addMouseListener(this);
    ...
    public void mouseClicked(MouseEvent e) {
        ...//Implementação vai aqui...
    }
}
```

Teclas de atalho

É possível, e bem prático, usar teclas de atalho no Java *Swing* para deixar o acesso às ações que se deseja mais rápido. Como as ações são geralmente acionadas clicando em botões e menus, é muito natural então usar as teclas de atalho a um botão específico ou a um menu.

Em geral, existem dois tipos de tecla de atalho em Java:

MNEMONIC	É um único caractere (normalmente uma letra do alfabeto de A a Z) pressionado depois da tecla *Alt*. Isto acionará um *ActionListener* associado com o menu ou botão. Por exemplo: *Alt + F, Alt +* O etc. O menu associado é exibido se o mnemônico é usado. A letra *mnemônico* é sublinhada na legenda do menu ou botão.

ACELERADOR	É uma combinação de teclas (*Ctrl* + letra ou tecla de função), que quando pressionado, irá acionar um *ActionListener* associado com o menu ou botão. Por exemplo: F5, *Ctrl* + O, *Ctrl* + F5 etc. O menu associado não será exibido se o acelerador é usado. A combinação de teclas é exibida ao lado do item de menu.

A figura 4.19 mostra um menu típico, com ambos os atalhos *mnemônicos* e acelerador:

Figura 4.19 – Teclas de atalho.

O código fonte do exemplo é mostrado na listagem 5:

```
 1 import java.awt.event.ActionEvent;
 2 import java.awt.event.ActionListener;
 3 import java.awt.event.KeyEvent;
 4 import javax.swing.AbstractAction;
 5 import javax.swing.Action;
 6 import javax.swing.JFrame;
 7 import javax.swing.JMenu;
 8 import javax.swing.JMenuBar;
 9 import javax.swing.JMenuItem;
10 import javax.swing.KeyStroke;
11 import javax.swing.SwingUtilities;
12
13 public class TesteAtalho extends JFrame {
14     public TesteAtalho() {
15         super("Testando Atalhos");
16         menu();
17         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18         setSize(300,100);
19         setLocationRelativeTo(null);
20     }
21
22     private void menu() {
```

```
 23          JMenuBar barra = new JMenuBar();
 24          JMenu menuArquivo = new JMenu("Arquivo");
 25          menuArquivo.setMnemonic(KeyEvent.VK_A);
 26
 27          JMenuItem menuAbrir = new JMenuItem("Abrir");
 28          menuAbrir.setMnemonic(KeyEvent.VK_B);
 29
 30               KeyStroke  atalhoAbrir  =  KeyStroke.getKeyStroke(KeyEvent.VK_B,
KeyEvent.CTRL_DOWN_MASK);
 31          menuAbrir.setAccelerator(atalhoAbrir);
 32
 33          menuAbrir.addActionListener(new ActionListener() {
 34              @Override
 35              public void actionPerformed(ActionEvent evt) {
 36                  System.out.println("Abrindo arquivo...");
 37              }
 38          });
 39
 40          JMenuItem menuSalvar = new JMenuItem();
 41          Action salvar = new AbstractAction("Salvar") {
 42              @Override
 43              public void actionPerformed(ActionEvent e) {
 44                  System.out.println("Salvando...");
 45              }
 46          };
 47          salvar.putValue(Action.MNEMONIC_KEY, KeyEvent.VK_S);
 48          salvar.putValue(Action.ACCELERATOR_KEY,
 49               KeyStroke.getKeyStroke(KeyEvent.VK_S, KeyEvent.CTRL_DOWN_MASK));
 50
 51          menuSalvar.setAction(salvar);
 52
 53          JMenuItem menuSair = new JMenuItem("Sair");
 54          menuSair.setMnemonic('R');
 55          menuArquivo.add(menuAbrir);
 56          menuArquivo.add(menuSalvar);
 57          menuArquivo.add(menuSair);
 58          barra.add(menuArquivo);
 59          setJMenuBar(barra);
 60      }
 61
 62      public static void main(String[] args) {
 63          SwingUtilities.invokeLater(new Runnable() {
 64              @Override
```

```
65              public void run() {
66                  new TesteAtalho().setVisible(true);
67              }
68          });
69      }
70  }
```

Listagem 5 – Teclas de atalho

A atribuição das teclas de atalho serão explicadas a seguir.

- Na linha 25 é atribuído o *mnemônico* A ao menu Abrir. Sendo assim, o usuário, ao pressionar Alt+A irá abrir o menu. O mesmo é feito na linha 47 para o menu Salvar.
- Na linha 28 ocorre a atribuição do mnemônico para o item de menu Abrir
- Nas linhas 30 e 31 é feita a atribuição do acelerador Ctrl+B para o menu Abrir. Logo, este item de menu terá um acelerador e um mnemônico simultaneamente. Nas linhas 48 e 49 é feito o acelerador para o menu Salvar. Observe também que na linha 41 temos um objeto chamado salvar, da classe Action.

CONCEITO

Segundo a documentação da *Oracle*, a *interface Action* é uma extensão da *interface ActionListener*. Ela é usada quando vários controles podem (ou não) acessar as mesmas funcionalidades. Por exemplo, se você tiver dois ou mais componentes que fazem a mesma função, uma *Action* pode ser usada para implementar esta função específica. Um objeto *Action* é um *listener* para um evento action que fornece não apenas a manipulação de ação e evento, mas também a manipulação centralizada do estado de acionamento dos componentes que possuem os eventos *action* tais como botões da barra de ferramentas, itens de menu, botões comuns e campos de texto . O estado que um *Action* pode manipular inclui texto, ícone, *mnemônico*, kp ativado e selecionado.

Você pode estar com a seguinte dúvida: "Tá, e daí? Qual a diferença entre um *AbstractAction* e um *ActionListener*"?

A resposta não é tão complicada: um *ActionListener* é uma *interface* que responde a uma ação ocorrida no sistema, certo?

Já a *AbstractAction*, é a ação de fato. Ela possui dados como o nome da ação, a tecla de atalho, o ícone e o *ActionListener* que vai ser ativado caso esta *Action* seja utilizada.

Para exemplificar, vamos supor que no seu sistema possua um comando chamado "Limpar" o qual estará disponível em uma barra de menu, no menu de contexto (quando o usuário clica com o botão direito) e também em um botão na barra de ferramentas. Podemos criar um *Listener* para cada um desses eventos ou criar uma *Action*, chamada *LimparAction* por exemplo, e atribui-la a todos estes componentes. Desta forma, os componentes terão o mesmo *label*, ícone (quando for o caso), tecla de atalho e outros atributos que sejam comuns. Todos os componentes acionarão o mesmo *listener*.

Veja o exemplo a seguir na listagem 6. Foi criada uma *Action* chamada *LimparAction* na qual temos o texto Limpar aplicado a um botão e uma tecla de atalho atribuída. Desta forma, todos os componentes que forem associados a esta *Action* terão estes elementos já configurados e também atribuídos. Para que isso ocorra, é necessário fazer a associação pelo construtor (e não mais pelo método *addActionListener,* preste atenção nisso, veja na listagem um exemplo).

```
 1 public class LimparAction extends AbstractAction {
 2     public LimparAction() {
 3         super("Limpa o conteúdo da tela");
 4
 5         putValue(SHORT_DESCRIPTION, "Limpar");
 6         putValue(MNEMONIC_KEY, KeyEvent.VK_L);
 7     }
 8     public void actionPerformed(ActionEvent e) {
 9        System.out.println("Limpando ...");
10     }
11 }

   //exemplo de instanciação
   JButton botão = new JButton(new LimparAction)
```

Listagem 6 – Exemplo de *AbstractAction.*

A *Action* mostrada na Listagem 6 pode ser atribuída a outros menus ou botões presentes na *interface* que você está criando. Mais simples, né? Porém preste atenção: caso sua aplicação precise de uma ação que vai ocorrer especificamente sob uma determinada circunstância, aí é melhor usar o *Listener* da maneira "tradicional", como vimos até agora.

Beleza? Tudo certo até aqui? Vamos fazer uma atividade para treinar um pouco?

ATIVIDADE

Acredite, se você partir para a área de desenvolvimento em Java, ainda mais em programação para *Desktop*, você vai usar muito as *JTable*. A grande maioria das aplicações faz conexão com um banco de dados e basicamente as operações para fazer a conexão e apresentar os dados são:

1. Conectar com o banco de dados;
2. Conectar com a tabela;
3. Fazer a consulta em SQL usando o comando *Select* com os devidos filtros;
4. Criar o *TableModel* baseado na consulta;
5. Apresentar os dados na *JTable*.

Vamos supor que os itens 1 e 2 já estejam feitos e a consulta do item 3 retornou 25 registros. A tabela contém os campos Codigo, Nome e Endereço. O desafio aqui é criar um *JFrame* simples e uma *JTable* e fazer a simulação da leitura de 25 registros. Veja bem: não é para acessar nenhum banco de dados, a ideia da atividade é fazer você criar uma *JTable* e ler 25 linhas por meio de um loop com dados fictícios. Apenas os nomes das colunas devem ser respeitados.

REFLEXÃO

Com o estudo mais detalhado da *JTable* podemos expandir as capacidades de nossas aplicações que acessam bancos de dados. A internet possui muitas empresas que comercializam componentes muito interessantes baseados na *JTable*. Vale a pena dar uma pesquisada e se basear nas ideias principais vendidas por estas empresas para criar as suas próprias *JTable*. Lembre-se sempre que podemos criar um componente usando as relações de herança disponíveis na linguagem Java. Bons estudos e sucesso!

REFERÊNCIAS BIBLIOGRÁFICAS

DEITEL, H. M.; DEITEL, P. J. **Java:** como programar. 6ª. ed. Rio de Janeiro: Bookman, 2005.

HORSTMANN, C. S.; CORNELL, G. **Core Java.** Rio de Janeiro: Pearson Education, 2010.

HUBBARD, J. R. **Programação com Java 2.** Rio de Janeiro: Bookman, 2006.

ORACLE CORPORATION. **The Java Tutorials.** Oracle Java Documentation, 01 mar. 2015. Disponível em: <https://docs.oracle.com/javase/tutorial>. Acesso em: 1 mar. 2016.

ORACLE CORPORATION. **Java Platform, Standard Edition 7.** API Specification, 2016. Disponível em: <https://docs.oracle.com/javase/7/docs/api/>. Acesso em: 01 mar. 2016.

ORACLE CORPORATION. J**ava Platform, Standard Edition 7.** Class JTextField. 01 mar 2015. Disponível em: <https://docs.oracle.com/javase/7/docs/api/javax/swing/JTextField.html>. Acesso em: 1 mar 2016.

5

Coleções

Coleções

Já falamos ao longo deste livro o quanto a linguagem Java é poderosa e importante. Pode acreditar que muito do que você precisa em termos de recursos para programar aplicativos interessantes você encontrará nativamente na linguagem ou dentro da sua biblioteca de classes.

Um assunto que estudamos na faculdade, abordado em livros e treinamentos é a manipulação de vetores e matrizes em Java. Sim, é um assunto importantíssimo para qualquer programador e dominar este assunto é sem dúvida fundamental. Mas você vai perceber ao longo dos seus dias de trabalho como programador(a) que existem situações em que estas estruturas de dados deixam a desejar em tarefas simples. E é aí que aparecem as coleções em Java, chamadas de *Collections*. Vamos apresentar as principais neste capítulo e te dar uma base para poder explorar as outras por sua própria conta.

Além disso vamos estudar outros assuntos tão importantes quanto as *Collections*: as classes "envelopadas" (*wrapper*) e os genéricos. Estes conceitos quando combinados formam estruturas de dados muito poderosas e úteis. Esperamos que você acompanhe e entenda os exemplos porque certamente farão diferença nos seus programas.

Bom estudo! Vamos lá!

OBJETIVOS

Ao final deste capítulo você estará apto(a) a:

- Criar programas contendo coleções em Java;
- Usar as *wrapped* classes e aplicá-las nos seus projetos;
- Criar classes genéricas usando os conceitos dos *Generics* em Java.

Introdução

Usar *arrays* em Java é muito importante. Sem dúvida são estruturas de dados que mesmo sendo adequadas em muitas situações, apresentam inúmeras restrições quanto a sua manipulação, dentre as quais podemos mencionar:

- Não dá para redimensionar um *array* em Java;
- Se você não souber um índice de um determinado valor, não dá para buscá-lo;

• Não dá para saber quantas "casas" do *array* foram preenchidas a não ser que a gente faça algum método para calcular isso;

• Toda vez que eu quiser saber quantas posições estão preenchidas em um *array* eu vou ter que percorrê-lo?

Percebeu? Esses são apenas alguns dos problemas que o programador encontra ao usar *arrays*. A linguagem Java resolve estes problemas rapidamente por meio de um conjunto de estruturas de dados já definidas chamadas *Collections*.

CONEXÃO

Não deixe de consultar a api da linguagem java para entender um pouco mais sobre as *collections*. Disponível em: <https://docs.oracle.com/javase/7/docs/technotes/guides/collections/overview.html>.

O *framework* das *Collections* em Java é formado ao redor de algumas interfaces padrão (lembre-se que vimos sobre elas no Capítulo 3). Muitas destas *interfaces* podem ser usadas na forma como são apresentadas ou serem modificadas de acordo com a sua necessidade.

Portanto, um *framework* para *Collections* é uma arquitetura unificada para representar e manipular coleções. Esta arquitetura contém:

INTERFACES	Estes tipos de dados abstratos representam as coleções. As *interfaces* permitem que as coleções sejam manipuladas independentemente dos seus detalhes de representação. Já vimos que as *interfaces* podem formar hierarquias.
IMPLEMENTAÇÕES (CLASSES)	São as implementações concretas das *interfaces* das coleções. Na verdade, elas formam as estruturas de dados reusáveis.
ALGORITMOS	São os métodos que realizam a parte computacional como por exemplo a procura, ordenação nos objetos da coleção. Os algoritmos são polimórficos, ou seja, os métodos podem ser usados em vários tipos de implementações diferentes da *interface* de coleção apropriada.

Vamos ver quais são as interfaces que podemos encontrar nas *Collections* em Java:

Interfaces

O *framework* das *Collections* define várias *interfaces.* A seguir uma descrição sucinta de cada uma delas.

COLLECTION	Permite que você trabalhe com grupos de objetos. É a *interface* que está no topo da hierarquia.
LIST	Estende a *interface Collection*. Uma instância de *List* armazena uma coleção de elementos ordenada.
SET	Estende a *interface Collection*. Guarda conjuntos os quais devem conter elementos únicos.
SORTEDSET	Estende a *Set* para lidar com conjuntos ordenados.
MAP	É uma estrutura que mapeia chaves únicas a valores.
MAP.ENTRY	Descreve um elemento (ou seja, um par chave/valor) dentro de um *Map*. Na verdade, *Map.Entry* é uma classe interna (*inner class*) de *Map*.
SORTEDMAP	Estende *Map* de forma que as chaves são mantidas na ordem crescente.
ENUMERATION	É uma *interface* legada que define os métodos pelos quais você pode enumerar (ou seja, obtido um por vez) os elementos dentro de uma coleção de objetos. Esta *interface* foi substituída pela Iterator.

Assim como existem as *interfaces*, temos as classes que compõem o *framework* das coleções em Java. As classes que implementam as *interfaces* de coleção têm nomes típicos no formato <estilo-da-implementação>*Interface*. Veja a tabela 5.1 para entender melhor a nomenclatura.

INTERFACE	TABELA *HASH*	*ARRAY* REDIMENSIONÁVEL	ÁRVORE BALANCEADA	LISTA LIGADA	TABELA *HASH* COM LISTA LIGADA
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Tabela 5.1 – *Collections: Interfaces* e suas implementações.

CONEXÃO

Tenho certeza que você pode estar confuso(a) com tantos termos. Não se preocupe, estas estruturas de dados (*hash*, árvore, lista ligada e outras) serão estudadas em uma disciplina específica na qual todas elas serão vistas com mais detalhes. Por hora, a página da wikipedia sobre estruturas de dados pode ser consultada para você ir se acostumando com elas: <https://pt.wikipedia.org/wiki/Estrutura_de_dados>.

A tabela 5.1 mostra na primeira coluna as principais estruturas de dados de acordo com a *interface* correspondente em Java e nas colunas adjacentes temos a classe que pode ser usada para implementar a estrutura que queremos. Então veja só: se queremos usar uma lista de valores (um *array*) o qual seja redimensionável (entre outras características), podemos usar um *ArrayList*. Se queremos usar outra estrutura de dados chamada lista duplamente ligada (*DEQUE*), temos que procurar uma classe correspondente na tabela.

Veja a figura 5.1 a seguir para entender a hierarquia da *interface Collection*.

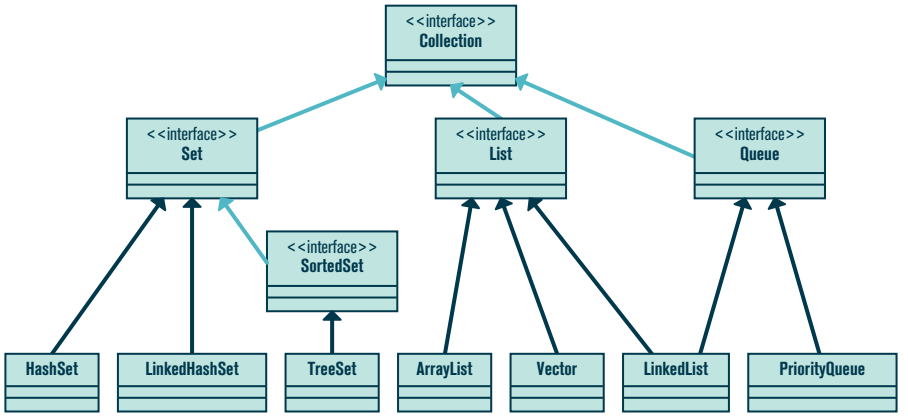


Figura 5.1 – Hierarquia do *framework Collection.*

Para sermos mais práticos e diretos, e para você entender melhor, vamos usar uma classe que é super útil e muito usada em vários tipos de programas, como por exemplo, aqueles que interagem com bancos de dados: a ArrayList.

ArrayList

Em primeiro lugar, gosto de dizer uma frase interessante: "uma coisa é uma coisa, outra coisa é outra coisa! ". Ou seja, um *array* é um *array*. E um *ArrayList* é um *ArrayList*! Melhorando: um *ArrayList* não é um *array*! Antes de começar a explicar sobre *ArrayList*, já tenha isto em mente.

Outro detalhe: o *ArrayList* é uma implementação de *List* (lembre-se da tabela 5.1). Portanto, quando tratamos um *ArrayList*, estamos tratando de um *List*.

A *ArrayList* implementa todas as operações de lista opcionais, e permite todos os elementos, incluindo o *null*. Além de implementar a *interface List*, essa classe fornece métodos para manipular o tamanho do *array* que é usado internamente para armazenar a lista. Vamos à prática!

Para criar um *ArrayList*:

```
ArrayList minhalista = new ArrayList();
```

Como você pode perceber, não usamos os "[]" típicos da criação de um *array*. Ou seja, não definimos o tamanho dela. Logo, qual é a capacidade dela? A resposta é: o quanto de memória RAM separada para o seu programa permitir.

Outro detalhe diferente do *array* é que não especificamos o tipo da lista. Podemos inserir **objetos** nela e isto é diferente do *array* tradicional que você conhece. Ou seja, vamos trabalhar com instâncias da classe *Object*. É importante lembrar que toda Collection trabalha da maneira mais genérica possível.

Também podemos abstrair a criação da lista a partir da interface *List*:

```
List minhalista = new ArrayList();
```

Para inserir nomes em uma lista:

```
minhalista.add("Fabiano");
minhalista.add("Vinicius");
```

Como podemos perceber, usamos o método *add()* para inserir elementos em um *ArrayList*. O método *add()* sempre insere os elementos no final. Existe outro método que permite inserir o item em qualquer lugar da lista.

Lembra da ContaBancaria que usamos como exemplo lá no Capítulo 3? Podemos criar uma lista de contas assim:

```
ContaBancaria cb1 = new ContaBancaria();
Cb1.depositar(500);

ContaBancaria cb2 = new ContaBancaria();
Cb1.depositar(50);

ContaBancaria cb3 = new ContaBancaria();
Cb1.depositar(550);

List listaContas = new ArrayList();

listaContas.add(cb1);
listaContas.add(cb2);
listaContas.add(cb3);
```

Veja o que podemos fazer com uma lista. O programa a seguir cria um ArrayList e faz várias operações que vão te ajudar a compreender melhor o que pode ser feito com este tipo de estrutura.

```
 1 import java.util.*;
 2
 3 public class Teste {
 4    public static void main(String args[]) {
 5       // criando a lista, chamamos de al (array list)
 6       ArrayList al = new ArrayList();
 7       System.out.println("Tamanho inicial: " + al.size());
 8
 9       // inserindo elementos na lista
10       al.add("C");
11       al.add("A");
12       al.add("E");
13       al.add("B");
14       al.add("D");
15       al.add("F");
16       al.add(1, "A2");      //observe bem esta forma de inserir
17       System.out.println("Tamanho apos insercoes: " + al.size());
18
19       // mostrando a lista
20       System.out.println("Conteudo da lista: " + al);
21
22       // Removendo elementos
23       al.remove("F");
24       al.remove(2);
25
26       if(al.contains("A")){
27             System.out.println("O elemento 'A' esta na lista");
28       }
29       else {
30             System.out.println("O elemento 'A' NAO esta na lista");
31       }
32
33       System.out.println("Tamanho apos remocoes: " + al.size());
34       System.out.println("Conteudo: " + al);
35    }
36 }
```

O resultado da execução do programa é mostrado a seguir.

```
Tamanho inicial: 0
Tamanho apos insercoes: 7
Conteudo da lista: [C, A2, A,
O elemento 'A' NAO esta na li
Tamanho apos remocoes: 5
Conteudo: [C, A2, E, B, D]
```

O programa mostra alguns métodos importantes sobre o ArrayList:

ADD()	Como já falamos, este método serve para inserir elementos no final da lista (linhas 10 a 15) ou em uma determinada posição, passando esta como parâmetro (linha 16).
SIZE()	Mostra o número de elementos da lista.
REMOVE()	Este método remove um elemento da lista que é passado por parâmetro (linha 23) ou remove o item na posição desejada (linha 24).
CONTAINS()	Este método retorna *true* se o elemento passado por parâmetro está na lista ou *false* caso contrário.

Conhecemos os principais métodos da *ArrayList*. Vamos agora aplicar os conceitos que aprendemos na classe ContaBancaria. Veja a listagem do programa.

```
 1 import java.util.*;
 2 public class ContaCorrente {
 3       private int numero;
 4       private double saldo;
 5       private static int totalContas;
 6
 7       public ContaCorrente(int n, double s){
 8             numero = n;
 9             saldo = s;
10             totalContas++;
11       }
12
```

```
13        public int getNumero(){ return numero; }
14        public double getSaldo(){ return saldo; }
15
16        public void depositar(double valor){
17              saldo += valor;
18        }
19
20        public void sacar(double valor){
21              if(saldo >= valor){
22                    saldo -= valor;
23              } else {
24                    System.out.println("Saldo insuficiente.\n");
25              }
26        }
27
28        public String toString(){
29              String retorno = "Conta "+getNumero()+" - Saldo: "+getSaldo();
30              return retorno;
31        }
32
33        public static void main(String[] args){
34              ContaCorrente.totalContas = 0;
35
36              ContaCorrente conta1 = new ContaCorrente(1,100.0);
37              ContaCorrente conta2 = new ContaCorrente(2,200.0);
38              ContaCorrente conta3 = new ContaCorrente(3,300.0);
39
40              List contas = new ArrayList();
41              contas.add(conta1);
42              contas.add(conta2);
43              contas.add(conta3);
44
45              conta1.depositar(10);
46              conta2.depositar(22);
47              conta3.depositar(33);
48
49              for (int i=0; i<contas.size(); i++){
50                    System.out.println(contas.get(i));
51              }
52        }
53  }
```

Vamos ao programa:

Nas linhas 2 a 31 temos o código que define a conta corrente. É uma classe bem simples contendo variáveis de instância e uma variável de classe.

Entre as linhas 33 e 52 temos a definição do método principal, onde teremos a execução dos testes com o *ArrayList* criado na linha 40.

Usamos os métodos *add() e size()* e apresentamos o *get()* na linha 50. Este método retorna o objeto existente em determinada posição da lista. Como fizemos um *loop* que percorre a lista nas linhas 49 a 51, o objeto de cada posição será impresso. Como temos um método *toString()* na definição da ContaCorrente, o *loop* usará o *toString()* para imprimir as informações na tela conforme o resultado a seguir.

```
Conta 1 - Saldo: 110.0
Conta 2 - Saldo: 222.0
Conta 3 - Saldo: 333.0
```

Mas tem um detalhe: se o *loop* fosse substituído pelo código a seguir, nada iria ser mostrado na tela, pois o objeto da posição "i" (que é uma conta corrente) por si só não foi programado para mostrar informação alguma.

```
for (int i=0; i<contas.size(); i++){
    contas.get(i);
}
```

E se quiséssemos que o saldo seja mostrado em cada repetição do *loop*? Por exemplo, algo como:

```
    contas.get(i).getSaldo();
```

É possível? Não. E é aí que você tem que entender como o *framework* trabalha. O *get* vai retornar sempre uma instância da classe *Object*. Para que o saldo seja recuperado, é necessário fazer um *cast* (conversão) para ContaCorrente da seguinte forma:

```
for (int i = 0; i < contas.size(); i++) {
  ContaCorrente conta = (ContaCorrente) contas.get(i);
  System.out.println(conta.getSaldo());
}
```

! ATENÇÃO

Não se usa o "*for*" para percorrer este tipo de lista. Neste caso, é recomendado usar um iterator ou o "*for* extendido".

Os exemplos mostraram a mecânica principal de trabalho com os *ArrayLists*. A seguir vamos apresentar outros métodos que podem te ajudar na manipulação dos itens deste tipo de estrutura:

Você deve ter percebido que os métodos mostrados na relação são bem úteis e que se usássemos arrays tradicionais seria um pouco mais complicado implementá-los, não é? Logo, para que reinventar a roda? Já existe muita coisa pronta! Use as coleções quando possível ao invés dos *arrays* e aproveite estes métodos. Existem outros métodos que vamos deixar por sua conta pesquisar e verificar como funcionam. Como sempre, lembre-se da API!

DDALL(COLLECTION C)	Insere todos os elementos da coleção c no fim da lista, na ordem que eles são retornados pelo iterador específico da coleção.
ADDALL(INT I, COLLECTION C)	Veja que é o mesmo método que o anterior, porém neste caso, a coleção c será inserida na posição i.
CLEAR()	Remove todos os elementos da lista.
CLONE()	Como o nome sugere, "clona" a lista (faz uma cópia).
ENSURECAPACITY(INT MINCAPACITY)	Aumenta a capacidade da lista, se necessário, para garantir que ela consiga guardar ao menos o número de elementos passado no parâmetro.
INDEXOF(OBJECT O)	Retorna o índice na lista da primeira ocorrência do elemento especificado no parâmetro, ou retorna –1 se a lista não possui o elemento.
LASTINDEXOF(OBJECT O)	Retorna o índice na lista da última ocorrência do elemento especificado, ou –1 se a lista não possui o elemento.

REMOVERANGE(INT INICIO, INT FIM)	Remove todos os elementos da lista que estiverem entre os índices especificados nos parâmetros.
SET(INT INDICE, OBJECT ELEMENTO)	Substitui o elemento na posição especificada no parâmetro pelo elemento do segundo parâmetro.
TOARRAY()	Retorna um *array* de todos os elementos na lista.

Classes Wrapper

Vimos no final do tópico anterior um recurso no qual o *loop* percorria uma lista, na verdade um *ArrayList*, e para cada posição era executado um o método chamado *get()*, que retornava o objeto da posição atual do *loop*.

Porém caso fosse necessário chamar um método da classe do objeto, não seria possível. Para resolver o problema, usamos um *cast* para a classe necessária e deu tudo certo.

A linguagem Java oferece um recurso chamado *wrapper* ou classes empacotadoras (*wrapper classes*), ou invólucro segundo alguns autores, que possuem métodos que fazem conversões em variáveis primitivas e também encapsulam tipos primitivos para serem usados como objetos, ou seja, "embrulham" o conteúdo em outro formato.

Na programação orientada a objetos, uma classe *wrapper* é uma classe que encapsula os tipos, de modo que esses tipos podem ser usados para criar instâncias e métodos de objetos em outra classe que precisa desses tipos. Assim, uma classe *wrapper* é uma classe que encapsula, oculta ou envolve os tipos de dados dos oito tipos de dados primitivos da linguagem (veja o Capítulo 1), de modo que estes podem ser usados para criar objetos instanciados com métodos em outra classe ou em outras classes.

Vamos lembrar de outro exemplo para ajudar no seu entendimento. Lembra no Capítulo 1 quando usamos alguns métodos para converter tipos de dados ao fazer a leitura do teclado, como por exemplo no código a seguir.

```
1  import javax.swing.*;
2
3  public class Teste {
4      public static void main(String args[]) {
5          String entrada = JOptionPane.showInputDialog("Quantos anos voce tem?");
6          int anos = Integer.parseInt(entrada);
7          System.out.println("Voce   tem   "   +365*anos+"   dias   de   ida-
de aproximadamente.");
8      }
9  }
```

Olhe a linha 6. Aí está um exemplo de uma classe *wrapper*! Portanto, temos uma classe wrapper para cada tipo primitivo da linguagem identificado pelo mesmo nome do tipo que possui com a primeira letra maiúscula. Veja a figura 5.3 para entender como funciona a hierarquia das classes *wrapper*.

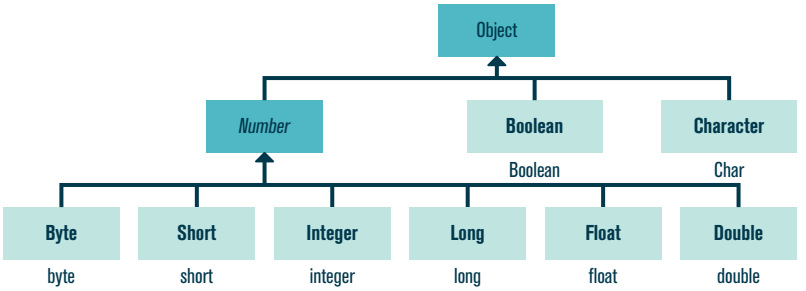


Figura 5.2 – Hierarquia das classes *wrapper*.

O uso das classes *wrapper* é bem abrangente, mas no nosso caso percebemos sua aplicação quando lidamos com objetos de coleção (*Collection*) do Java.

Agora cabe outra reflexão a respeito de usar recursos naturais da linguagem ou da API. No tópico anterior fizemos isso com os *arrays e Collections*. Ou seja, é melhor usar um *array* ou uma classe *Collection*? O mesmo raciocínio pode ser aplicado aqui. No caso de um número inteiro, é melhor usar o tipo int ou a classe *Integer*, uma vez que esta, por ser uma classe, possui métodos específicos para trabalhar com números inteiros e facilitar a nossa vida de programador. A resposta é que cada programa tem uma necessidade diferente e o uso vai depender do caso. Apenas tenha em mente que estamos falando de tipos primitivos e objetos, ou seja, são estruturas e conceitos diferentes.

Temos que lembrar que as classes *wrapper* de tipos primitivos "apenas" encapsulam (ou embrulham) os tipos primitivos e os armazenam. Mas como já

dissemos, se elas são classes então possuem métodos e estes podem ser usados a nosso favor. Veja o caso da classe *Integer* e seus métodos no *link*: <https://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html>. Você vai encontrar métodos para converter *Integer* para outros formatos, obter o complemento de dois do número (o complemento de dois é usado na representação binária de um número), além de outros métodos úteis.

Vamos exemplificar com alguns pedaços de código para você entender melhor. Observe com atenção:

```
String numero1 = “1234”;
String numero2 = “123.45”;

Olha o que podemos fazer:
Float fnum = new Float(numero2);
ou
Float fum = new Float(“123.45”);

Integer inum = new Integer(numero1);
ou
Integer inum = new Integer(“1234”);
```

Para fazer a conversão de tipos primitivos para classes *wrappers* podemos usar os seguintes métodos:

XXXVALUE()	Usado quando precisa realizar uma conversão do valor de um objeto wrapper para um objeto primitivo. É claro que o “xxx” na frente do “*Value*” varia de acordo com o tipo, por exemplo *doubleValue()*, *intValue()* etc.
PARSEXXX(STRING S)	Já conhecemos este método não é? É usado para converter um objeto *String* para um tipo primitivo e retorna um primitivo nomeado. O “xxx” deve ser substituído pelo tipo desejado.
VALUEOF(STRING S)	Usado para converter um objeto *String* para um objeto *wrapper*. O método retorna um objeto *wrapper* recém criado do tipo que chamou o método.
TOSTRING()	Também já conhecido. Ele retorna a representação de um objeto em formato *String* (tipo primitivo encapsulado).

Vamos mostrar a aplicação destes métodos com alguns exemplos.

```
Criando um objeto wrapper
Integer dias = new Integer(365);

Convertendo para um tipo primitivo:
int dias_i = dias.intValue();

Conversão de uma String para o tipo primitivo
double valor = Double.parseDouble("123.45");
System.out.println("Valor = "+valor);

Exemplo do método valueOf com a classe String
Integer valor = new Integer(25);
String vString = String.valueOf(valor);

String valor = new String("123.45");
Double vDouble = Double.valueOf(valor);

System.out.println("Valor string: "+vString);
System.out.println("Valor double: "+vDouble);
```

Agora que aprendemos sobre as classes *wrapper*, vamos começar a relacionar os assuntos deste capítulo.

Na primeira parte do capítulo estudamos sobre as coleções. Vimos que não podemos colocar um inteiro (*int*) ou qualquer outro tipo primitivo dentro de um *ArrayList* (ou outro membro das coleções em Java). As coleções só podem guardar objetos, então temos que dar um jeito de encapsular os tipos primitivos na sua classe *wrapper* correspondente (no caso do *int*, usamos a classe *Integer*).

Quando removemos um objeto de uma coleção, estamos recuperando um *Integer* não é? E para obter o seu int correspondente, vamos usar o método *intValue()* que vimos neste tópico. Este mecanismo tem um nome na orientação a objetos em geral chamado *box e unbox*:

BOX	Quando colocamos o tipo primitivo na sua classe *wrapper* correspondente
UNBOX	No processo contrário, quando retiramos o valor da classe e obtemos o valor primitivo.

Na verdade, este trabalho de fazer o *box* e o *unbox* é meio chato e custoso, além de "poluir" o seu código. Será que não existe uma forma mais prática de automatizar isso? Uma das formas de fazer esta automação necessária é usar outro conceito muito útil em Java chamado *Generics* (tipos genéricos).

Generics

Agora vamos ter que voltar às coleções. Não falamos muito da classe *List*, pois usamos uma implementação específica chamada *ArrayList*. Mas se você entendeu o conceito das listas, você deve ter entendido que podemos inserir qualquer objeto nelas, certo? Logo, podemos misturar objetos em uma lista, assim:

Criamos uma lista:

```
List minhaLista = new ArrayList();

Criamos dois objetos de classes diferentes:
Integer valorInteiro = new Integer("123");
Double valorDouble = new Double("123.45");

Criamos até uma ContaCorrente:
ContaCorrente conta1 = new ContaCorrente(1,100.00);
```

```
Vamos misturar tudo na minhaLista:minhaLista.add("Uma string qualquer");
minhaLista.add(valorInteiro);
minhaLista.add(valorDouble);
minhaLista.add(conta1);
```

Mas isso é possível? Claro que é! Lembre-se: são objetos!

Se fosse possível desenhar esta lista, o resultado seria como mostrado na figura 5.4. Não se preocupe com alguns símbolos presentes na figura. Eles serão apresentados melhor para você na disciplina de Estrutura de Dados. Por hora perceba que a lista conterá objetos de classes diferentes.

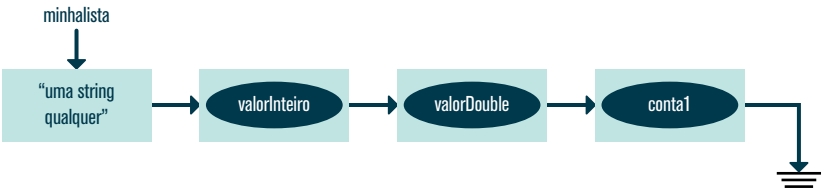


Figura 5.3 – ArrayList minhaLista.

Temos um problema. E se precisarmos recuperar os valores desta lista? Vimos o método *get()* e aprendemos que usamos o *cast* para recuperar os valores. Só que o problema agora é que temos uma lista com objetos de tipos diferentes e isso não vai dar certo. Outro detalhe: não é nem um pouco recomendável, nem útil, que uma lista misture tipos. Para poder "fechar" a lista e permitir apenas uso de um determinado tipo, usamos os *Generics*.

Vamos resolver este problema no exemplo a seguir com as classes das contas bancárias. Uma agência é um conjunto de contas não é? Logo, podemos implementar uma agência como uma lista de contas. Vamos lá:

```
List<ContaCorrente> contas = new ArrayList<>();
contas.add(conta1);
contas.add(conta2);
contas.add(conta3);
```

Temos algo diferente na criação da lista. Usamos o operador diamante "<>" para restringir a criação da lista contas para somente objetos do tipo ContaCorrente. Estamos aqui fazendo o uso dos *Generics*.

Quando usamos esta forma não precisamos mais fazer o *cast* no *loop* que vimos anteriormente. Sendo assim, é possível percorrer a lista da seguinte forma:

```
for(int i = 0; i < contas.size(); i++) {
    ContaCorrente conta = contas.get(i);
    System.out.println(conta.getSaldo());
}
```

Voltando ao exemplo das agências, vamos supor que precisamos listar todas as contas de uma agência. Teríamos que ter algo parecido com isso:

```
class Agencia {
  public ArrayList<ContaCorrente> listaContas() {
    ArrayList<ContaCorrente> contas = new ArrayList<>();

    // implementação do método

    return contas;
  }
}
```

O código está correto sintática e semanticamente, mas há um detalhe: ele só vai retornar a lista na forma de um *ArrayList*. Conforme você for ficando mais experiente e o seu programa necessitar, pode ser que o retorno seja em outro formato como *LinkedList*, por exemplo. Aí não dará certo. Neste caso, é melhor usar a interface mais genérica possível, a *List*. Assim:

```
class Agencia {
    public List<ContaCorrente> listaContas() {
        ArrayList<ContaCorrente> contas = new ArrayList<>();

        // implementação do método

        return contas;
    }
}
```

O exemplo anterior contém o uso da *interface List* no retorno do método. Da mesma forma que fizemos no retorno, é uma boa ideia fazer na assinatura do método e em todos os lugares possíveis. Veja o exemplo da passagem de parâmetro:

```
class Agencia {
    public void verificaContas(List<ContaCorrente> contas) {
        // implementação do método
    }
}
```

É importante, também, declarar atributos como *List* ao invés de fixar em uma implementação específica. Deste modo vamos obter um baixo acoplamento e assim podemos trocar a implementação pois estamos usando uma *interface*.

Métodos genéricos

É possível escrever uma única declaração de método genérico que pode ser chamado com argumentos de tipos diferentes. Isso é bem legal e útil! Com base nos tipos dos argumentos passados para o método genérico, o compilador trata cada chamada de método de forma adequada. Veja como fazer isso.

1. Todas as declarações de métodos genéricos usam o operador diamante que precede o do método de tipo de retorno (<E> no próximo exemplo);

2. Cada seção do tipo de parâmetro contém um ou mais parâmetros de tipo separados por vírgulas. Um tipo de parâmetro, também conhecido como uma variável de tipo, é um identificador que especifica um nome de tipo genérico;

3. Os parâmetros de tipo podem ser usados para declarar o tipo de retorno e atuam como espaços reservados para os tipos dos argumentos passados para o método genérico, que são conhecidos como argumentos de tipo reais;

4. O corpo de um método genérico é declarado como o de qualquer outro método. Importante: os parâmetros de tipo pode representar somente tipos de referência, ou seja, objetos e não tipos primitivos (como *int, double* e *char*).

Vamos entender melhor o que definimos com um exemplo usando *Generics*.

O exemplo é muito interessante porque mostra como podemos imprimir um *array* de tipos diferentes usando um único método genérico. Veja o destaque no código na linha 5 para saber como fazemos isso.

```
 1 import javax.swing.*;
 2
 3 public class Teste {
 4    // criando um metodo genérico mostraArray
 5    public static <E> void mostraArray(E[] arrayEntrada) {
 6       // Mostra os elementos do array
 7       for(E elemento : arrayEntrada) {
 8          System.out.printf("%s ", elemento);
 9       }
10       System.out.println();
11    }
```

```
12
13    public static void main(String args[]) {
14       // Cria um arrays de Integer, Double e Character
15       Integer[] intArray = { 1, 2, 3, 4, 5 };
16       Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
17       Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };
18
19       System.out.println("integerArray:");
20       mostraArray(intArray);   // passando um array de Integer
21
22       System.out.println("\ndoubleArray:");
23       mostraArray(doubleArray);   // passando um array de Double
24
25       System.out.println("\ncharacterArray:");
26       mostraArray(charArray);   // passando um array de Char
27    }
28 }
29
```

Veja que nas linhas 7 a 9 usamos o for extendido, que é a forma mais adequada, junto com o Iterator, para percorrer as coleções.

O resultado da execução é mostrado a seguir.

```
integerArray:
1 2 3 4 5

doubleArray:
1.1 2.2 3.3 4.4

characterArray:
H E L L O
```

Classes parametrizadas

Podemos declarar uma classe como classe genérica da mesma forma que criamos uma classe normal, exceto que o nome da classe é seguido por uma seção tipo de parâmetro.

Tal como acontece com métodos genéricos, a seção tipo de parâmetro de uma classe genérica pode ter um ou mais parâmetros de tipo separados por vírgulas. Essas classes são conhecidas como classes parametrizadas ou tipos parametrizados porque eles aceitam um ou mais parâmetros.

Veja o exemplo a seguir. Observe o parâmetro <T> na declaração da classe (linha 3). E depois veja como usamos os tipos para passar os parâmetros para a classe (linhas 15 e 16).

```
 1 import javax.swing.*;
 2
 3 public class Teste<T> {
 4    private T t;
 5
 6    public void set(T t) {
 7       this.t = t;
 8    }
 9
10    public T get() {
11       return t;
12    }
13
14    public static void main(String[] args) {
15       Teste<Integer> integerBox = new Teste<Integer>();
16       Teste<String> stringBox = new Teste<String>();
17
18       integerBox.set(new Integer(10));
19       stringBox.set(new String("Hello World"));
20
21       System.out.printf("Valor inteiro :%d\n\n", integerBox.get());
22       System.out.printf("Valor String :%s\n", stringBox.get());
23    }
24 }
```

Existem muitas aplicações para os *Generics* em Java. Na verdade, é uma boa prática usá-los. Quando você estiver estudando Estrutura de Dados você vai usar bastante este conceito e sempre que ler ou ouvir falar do assunto chamado "Padrões de Projeto" ou "*Design Patterns*" tenha certeza que terá algum relacionamento com isso.

ATIVIDADES

Vamos resolver alguns testes sobre os assuntos deste capítulo. Os dois primeiros testes deste capítulo são questões reais encontradas em concursos públicos. É interessante saber que estas provas, muitas vezes, contemplam conteúdos técnicos específicos.

01. Ano: 2015 - Banca: FGV - Órgão: TCE-SE - Prova: Analista de Tecnologia da Informação-Desenvolvimento.

Um programador Java precisa utilizar em seu aplicativo uma tabela dinâmica de inteiros, cujo tamanho pode aumentar ao longo da execução. Para isso, ele decide importar a classe *java.util.ArrayList* e a declaração da referência à tabela deverá ser:

a) *ArrayList<int> tabela;*
b) *ArrayList<Integer> tabela;*
c) *ArrayList<int>[] tabela;*
d) *ArrayList<int> tabela[];*
e) *ArrayList<Integer> tabela[].*

02. Ano: 2015 Banca: FGV Órgão: PGE-RO Prova: Analista da Procuradoria - Analista de Sistemas (Desenvolvimento).

Um programador Java precisa utilizar, em seu código, um arranjo dinâmico de números inteiros. A declaração correta para esse arranjo é:

a) *ArrayList<int> arranjo;*
b) *ArrayList<Int> arranjo;*
c) *ArrayList<Integer> arranjo[];*
d) *ArrayList<Int> arranjo[];*
e) *ArrayList<Integer> arranjo.*

03. Escreva um método genérico para trocar as posições de dois elementos de um *array*.

REFLEXÃO

Este foi apenas o último capítulo de um livro preparado cuidadosamente para você. Porém ele representa o início, ou a continuação, do seu aprendizado na área de programação de computadores. Não se prenda apenas a este material e às aulas. Pesquise, e muito. Você tem uma grande biblioteca que é a internet para consultar, conversar, participar de grupos e agregar conhecimento de várias fontes como textos, vídeos, vídeo aulas e outros. Aproveite isto. Não se prenda somente na linguagem Java. Usamos o Java aqui como ferramenta e não como finalidade. Os mecanismos que você estudou aqui são válidos para outras linguagens com algumas variações é claro, mas lembre-se, é como aprender outro idioma. Selecione as suas fontes de aprendizado. Se você está estudando Java, vá direto ao local onde ela é documentada (no site da *Oracle*) e de lá selecione outros locais confiáveis. No mais, só desejamos sucesso e felicidades a você!

REFERÊNCIAS BIBLIOGRÁFICAS

CORNELL, G.; HORSTMANN, C. Core **Java 2:** Recursos Avançados. São Paulo: Makron Books, 2001.

CORNELL, G.; HORSTMANN, C. S. **Core Java** - Vol. 1 - Fundamentos. 8ª. ed. São Paulo: Pearson Education, 2010.

DEITEL, H. M.; DEITEL, P. J. **Java:** como programar. 8ª. ed. Rio de Janeiro: Pearson, 2010.

ECKEL, B. **Thinking in Java.** 4ª. ed. New Jersey: Prentice Hall, 2006.

FLANAGAN, D. Java: **O guia essencial.** 5ª. ed. Rio de Janeiro: Bookman, 2006.

HUBBARD, J. R. **Programação com Java.** 2ª. ed. Rio de Janeiro: Bookman, 2006.

SANTOS, F. G. D. **Linguagem de programação**. Rio de Janeiro: SESE, 2015.

SIERRA, K.; BATES, B. **Use a cabeça:** Java. 2ª. ed. Rio de Janeiro: Alta Books, 2007.

ZEROTURNAROUND. **Java Tools and Technologies Landscape for 2014**, 2014. Disponível em: <https://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-for-2014/6/>. Acesso em: 27 jul. 2016.

GABARITO

Capítulo 5

01. B

02. E

03. Sugestão:

```
public final class Teste {
    public static <T> void troca(T[] a, int i, int j) {
        T temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }
}
```

ANOTAÇÕES

ANOTAÇÕES