UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE EDUCAÇÃO TUTORIAL
CIÊNCIAS DA COMPUTAÇÃO

Componentes da Interface Gráfica da Linguagem Java

Wanderson Rigo

Versão 1.1
Florianópolis, Outubro de 2004

.

Esta versão inicial pode conter erros que passaram despercebidos ao julgamento do autor. Assim sendo, o mesmo pede que lhe sejam reportadas as falhas que por ventura o leitor encontre. Qualquer dúvida ou sugestão deve ser encaminhada por e-mail para `pet@inf.ufsc.br` ou para o autor. A versão atualizada deste documento deve estar disponíveis no endereço `http://monica.inf.ufsc.br`.

Este documento foi construído e formatado com LaTeX 2_ε

.

Sumário

Lista de Figuras

Prefácio

Este material destina-se a usuários da linguagem Java que pretendem incluir interfaces gráficas em suas aplicações, sejam elas autônomas ou applets. Salientamos que é de grande valia ao usuário já estar familiarizado com a Linguagem Java, pois o conteúdo desse material não explana conceitos básicos nem discute a sintaxe da Linguagem. Aconselhamos o leitor, caso julgar necessário, a buscar uma base no material *Introdução à Linguagem Java*, sitiado em `http://monica.inf.ufsc.br`.

Centralizaremos nossos esforços de forma à transmitir de maneira prática e sucinta o conteúdo, de modo que o leitor possa aprender e aplicar os tópicos abordados. Assim sendo, após apresentarmos uma breve explicação de cada componente (botões, janelas, barras de rolagem, etc...) usado para compormos nossas Interfaces, o leitor será convidado a aprofundar seus conhecimentos nos exemplos.

Capítulo 1 traz uma breve introdução do que é e para que servem as Interfaces Gráficas. Também dá bases ao entendimento por parte do leigo usando uma analogia de fácil compreensão.

Capítulo 2 apresenta a hierarquia das classes e pacotes em que o assunto propriamente dito se estrutura. Aqui também diferencia-se os componentes em dois hemisférios: Swing e A.W.T..

Capítulo 3 apresenta os componentes onde toda a Interface Gráfica deve estar galgada, as características de cada um, bem como alguns métodos e constantes para o gerenciamento e distribuição de outros elementos sobre os próprios componentes.

Capítulo 4 neste capítulo explicitamos os componentes que estamos mais habituados a utilizar, pois, de algum modo, realizam ou disparam funções corriqueiras à qualquer programa.

Capítulo 5 mostra os tópicos relacionados aos eventos disparados por muitos dos componentes já vistos nos capítulos precedentes. Também apresentamos métodos para gerir e tratar os eventos.

Após esta breve descrição dos tópicos que serão explicitados, sinta-se à vontade para explorar o conteúdo da maneira que melhor lhe convier.

Wanderson Rigo
wander@inf.ufsc.br

Programa de Educação Tutorial - PET/CCO
pet@inf.ufsc.br

Considerações Gerais

Ressaltamos que iremos explanar aqui somente alguns métodos necessários às funcionalides dos exemplos, cabendo ao leitor procurar na bibliografia indicada, se julgar necessário, maior esclarecimento sobre os demais métodos das classes. Cabe ainda lembrar que os exemplos mostrados aqui são voltados ao propósito maior, ou seja, habilitar o leitor a criar interfaces gráficas. A funcionalidade ou complexidade dos mesmos não é discutida pois estes exemplos só têm caráter ilustrativo e didático.

É bem sabido que existem programas de desenvolvimento que facilitam a tarefa que aqui propomos elucidar. Além de propiciarem retorno visual imediato, eles geram o código automaticamente. Aos olhos do programador pragmático, essas virtudes evocam produtividade e facilidade. Porque escolheríamos o caminho das pedras, que consiste num processo em que temos que pensar, compor nossos códigos e compilar cada vez que mudamos pequenos detalhes? Ora, como programador que se preze, e , com certeza , o leitor há de ser, essa via representa a continuidade do que aprendemos, ou seja, a implementação direta e sob nosso domínio, mantida sob nossa responsabilidade e competência. Além do mais, quem se aventurar pelas linhas dessa material, tranquilamente poderá explorar os dois caminhos de forma sólida e concisa.

Obviamente aqui não focalizaremos a construção de Interfaces no que toca a sua estética, mas sim no como fazer. Também em nenhum momento questionamos a qualidade dos aplicativos de linha de comando, porém, se só olhássemos o lindo azul do céu, como iríamos ver as demais belezas do planeta?

Esperamos que o fascínio de compor janelas elegantes e funcionais o motivem a extrair o máximo de proveito desse material, pois depois de toda a evocação e motivação precedente, estamos aptos a iniciar o estudo.

Capítulo 1

Introdução à Concepção de Interfaces Gráficas

É notório a todos nós que muitos dos programas que conhecemos interagem com os usuários através da troca de informações. O meio pelo qual a parte humana solicita ao programa a execução de tarefas, alguma resposta, ou qualquer comunicação entre as partes é feita pela Interface. Muitas vezes confundida com o programa em sí, a Interface tem peso significativo na aceitação do software, já que, com certeza, clientes são atraídos pela facilidade de manipulação e de aprendizado, telas atraentes e chamativas, bem como pelos componentes auto-explicativos. Essas características mencionadas são obtidas aliando-se criatividade, bom senso, organização lógica, conhecimento técnico, etc..

A interface gráfica com o usuário (GUI - *graphical user interface*) fornece a um programa um conjunto consistente de componentes intuitivos, familiarizando o usuário com as diversas funções e diminuindo o tempo de aprendizado da nova ferramenta [1]. As GUIs são construídas a partir de componetes GUI, que são objetos com o qual o usuário interage atráves dos disposítivos de entrada, ou seja, o mouse, o teclado, a voz, etc.

1.1 Analogia Recorrente

Antes de iniciarmos o conteúdo técnico deste material, vamos compor um cenário familiar ao leitor, de modo que este panorma venha a esclarecer a filosofia de trabalho que utilizaremos logo adiante. Valeremo-nos de uma analogia, que servirá de base ao entendimento dos componentes descritos nesse curso.

Imagine que construir interfaces consiste em colar adesivos em uma tela de vidro. Antes de tudo, é óbvio que devemos possuir uma tela que, como veremos, é representada pelos **conteiners**. Também dispomos de adesivos de diversos tamanhos que podem ser distribuídos e anexados livremente pela superfície do vidro. Tais adesivos elementares são os **painéis**. Além disso, dispomos de adesivos mais elaborados que já estão pré-definidos com figuras de botões, rótulos, etc. Eles podem ser colados diretamente no vidro, ou sobre os outros adesivos rudimentares (painéis), tal qual é a nossa vontade, embora limitando-se à capacidade do espaço físico disponível.

Na figura abaixo, a qual ilustra alguns componentes que serão estudados mais a frente, vemos a concepção real de nossa analogia.

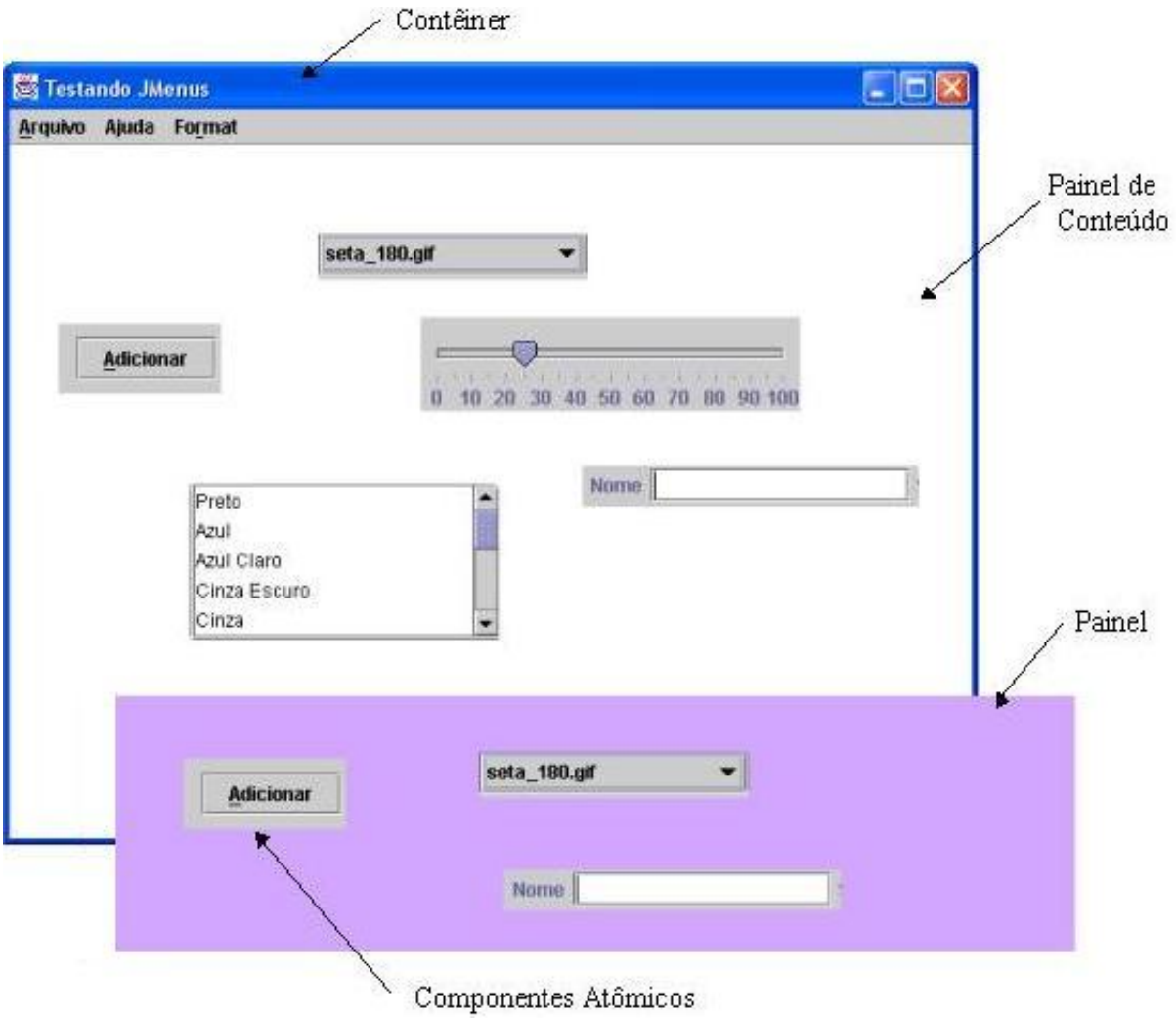


Figura 1.1: Apresentação de alguns Componentes GUI

Caro leitor, você há de convir que parece ser uma tarefa demasiadamente fácil construirmos interfaces que viabilizem a interação Homem x Máquina. Então agora, sem maiores delongas, iremos nos embrenhar pelo mundo da programação utilizando as classes, pacotes e as interfaces necessárias à solução de nosso problema.

Capítulo 2

Criação de Interfaces Gráficas

Em Java, as classes nas quais nos baseamos para criar os componentes, bem como para fornecer-lhes funcionalidade, estão agrupadas em dois grandes pacotes: **java.awt** (pacote do núcleo) e **javax.swing** (pacote de extensão). Os dois pacotes definem componentes com peculiaridades distintas e que serão discutidas logo abaixo.

2.1 Componentes Swing

O pacote javax.swing foi criado em 1997 e inclui os componentes GUI que se tornaram padrão em Java a partir da versão 1.2 da plataforma Java 2. A maioria dos componentes Swing (assim são denominados) são escritos, manipulados e exibidos completamente em Java, sendo conhecidos como componentes Java puros. Isso oferece a eles um maior nível de portabilidade e flexibilidade. Os nomes de tais componentes recebem um "J", como, por exemplo: JLabel, JButton, JFrame, JPanel, etc. Tal peculiaridade se justifica para diferenciar esses componentes dos que serão mencionados logo adiante. São considerados peso-leve e fornecem funcionalidade e aparência uniforme em todas as plataforma, sendo denominada de aparência de metal *(metal look-and-feel)*.

Figura 2.1: Aparência de metal (comum em todas as plataformas)

Entretanto, muitos componentes Swing ainda são considerados peso-pesados. Em particular, as subclasses de java.awt.Window, como **JFrame**, utilizada para exibir janelas e as de java.applet.Applet, como **JApplet** originam componentes que se apoiam no sistema de janelas da plataforma local para determinar sua funcionalidade, aparência e seu comportamento[1].

Figura 2.2: Aparência personalizada com o estilo do Motif

O Swing também fornece flexibilidade para personalizar a aparência e o comportamento dos componentes de acordo com o modo particular de cada plataforma, ou mesmo altera-los enquanto o programa está sendo executado. As opções são a personalização com o estilo do Microsoft Windows, do Apple Macintosh ou do Motif (UNIX).

2.2 Componentes A.W.T.

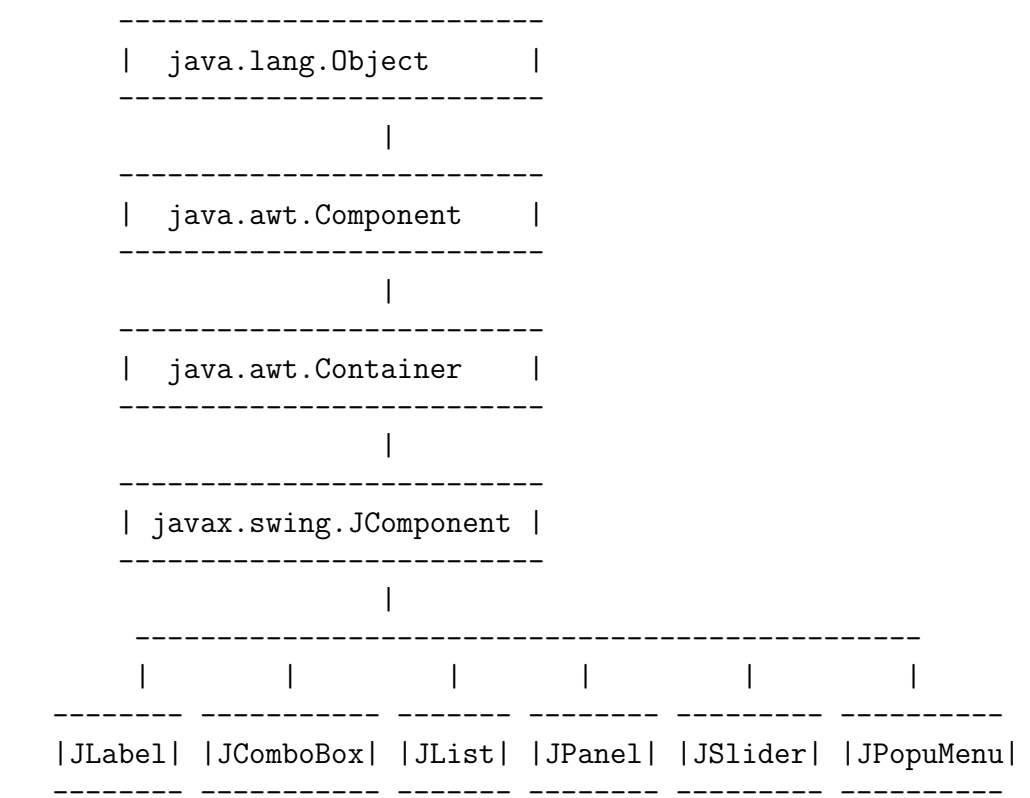
Os componentes GUI oriundos do pacote Abstract Windowing Toolkit (java.awt) tiveram origem na versão 1.0 da plataforma Java 2, e estão diretamente associados com os recursos da interface gráfica da plataforma do usuário. Dessa forma, a aparência dos componentes difere quando o programa é executado no Microsoft Windows e no Apple Macintosh. Podemos dizer que estes componentes considerados peso-pesados "herdam" a aparência definida pela plataforma, pois o A.W.T. foi projetado para que cada máquina virtual Java implemente seu elemento de interface. Isso pode ser desejável, uma vez que permite aos usuários do programa utilizar os componentes GUI com que eles já estão familiarizados, porém o leiaute e o alinhamento dos componentes pode se alterar devido aos tamanhos diferentes dos mesmos em cada plataforma [1].

Se você executar o exemplo implementado pelo código precedente, verá que a aparência dos componentes muda, porém a janela, que é um objeto da classe JFrame (considerado componente peso-pesado), permanece inalterável. Também é possível notar isso comparando as figuras 2.1 e 2.2.

Essa coleção de componentes para construção de Interfaces Gráficas está desatualizada e foi substituida pelo projeto Swing [6]. Em virtude disso, nossa ênfase reside no estudo e uso do pacote em maior evidência.

2.3 Hierarquia das Classes dos Componentes

Mostraremos abaixo a hierarquia de herança das classes que definem atributos e comportamentos que são comuns a maioria dos componentes Swing. Cada classe é exibida com o seu pacote:



As operações comuns à maioria dos componentes GUI, tanto Swing como AWT são definidas na classe **Component**. Isso inclui métodos relativos à posicionamento, personalização, tamanho, visibilidade, pintura, registro de tratadores de eventos, ajuste e retorno de estado dos componentes.

Em aplicativos com JFrames e em applets, anexamos os elementos ao painel de conteúdo, que é um objeto da classe **Container**. Logo, a classe Container dá suporte à adição e posicionamento dos componentes ao painel de conteúdo de um contêiner.

A classe **JComponent**, que define os atributos e comportamentos para suas subclasses, é a superclasse da maioria dos componentes Swing. Com exceção dos conteiners JFrame e JDialog, todos os demais componentes Swing cujo nome comece com "J" descendem da classe **JComponent** [2].

Agora mostraremos o código que implementa a funcionalidade de mudança de aparência dos componentes, sendo que esta, já foi descrita anteriormente:

```
1  // Mudando a aparencia da GUI
2  import java.awt.*;
3  import java.awt.event.*;
4  import javax.swing.*;
5
6  public class TesteLookAndFeel extends JFrame {
7
```

```
8       private String strings[] = { "Metal", "Motif", "Windows" };
9       private UIManager.LookAndFeelInfo aparencia[];
10      private JRadioButton radio[];
11      private ButtonGroup grupo;
12      private JButton botao;
13      private JLabel rotulo;
14      private JComboBox comboBox;
15      private JTextField campo;
16      private JTextArea texto;
17
18      // configura a GUI
19      public TesteLookAndFeel()
20      {
21          super( "Testando a Aparência e Comportamento" );
22
23          Container container = getContentPane();
24
25          // configura painel para a região NORTH de BorderLayout
26          JPanel painelNorte = new JPanel();
27          painelNorte.setLayout( new GridLayout( 2, 2, 5, 5 ));
28
29          // configura o rótulo para o painel NORTH
30          rotulo = new JLabel( "Esta é a aparência Metal" );
31          rotulo.setVerticalTextPosition(SwingConstants.CENTER);
32          container.add( rotulo );
33
34          // configura o botão para o painel NORTH
35          botao = new JButton( "Eu sou um Botão" );
36          painelNorte.add( botao );
37
38          campo = new JTextField( "Qualquer texto" );
39          painelNorte.add( campo );
40
41          // configura caixa de combinação para o painel NORTH
42          comboBox = new JComboBox( strings );
43          painelNorte.add( comboBox );
44
45          // anexa o painelNorte à região NORTH do painel de conteúdo
46          container.add( painelNorte, BorderLayout.NORTH );
47
48          // cria array para os botões de opção
49          radio = new JRadioButton[ 3 ];
50
51          // configura painel para a região SOUTH de BorderLayout
52          JPanel painelSul = new JPanel();
53          painelSul.setLayout( new GridLayout( 1, 3 ) );
54
55          // configura botões de opção para o painelSul
56          radio = new JRadioButton[ 3 ];
57                  radio[ 0 ] = new JRadioButton( "Metal" );
58                  radio[ 1 ] = new JRadioButton( "Motif" );
59                  radio[ 2 ] = new JRadioButton( "Windows" );
60
61          grupo = new ButtonGroup(); //implementa exclusão mútua
62          TratadorDeItens trat = new TratadorDeItens();
63
64          for ( int count = 0; count < radio.length; count++ ) {
65              radio[ count ].addItemListener( trat );
66              grupo.add( radio[ count ] );
```

```
67              painelSul.add( radio[ count ] );
68          }
69
70          // anexa o painelSul à região SOUTH do painel de conteúdo
71          container.add( painelSul, BorderLayout.SOUTH );
72
73          // obtém informções sobre a aparência e
74          // comportamento instalado
75          aparencia = UIManager.getInstalledLookAndFeels();
76
77          setSize( 400, 300 );
78          setVisible( true );
79          radio[ 0 ].setSelected( true );
80      }
81
82      // usa UIManager para mudar a aparência e comportamento da GUI
83      private void mudeTheLookAndFeel( int valor )
84      {
85          // muda aparência e comportamento
86          try {
87              UIManager.setLookAndFeel(
88              aparencia[ valor ].getClassName() );
89              SwingUtilities.updateComponentTreeUI( this );
90          }
91          // processa problemas com a mudança da aparência e
92          // do comportamento
93          catch ( Exception exception ) {
94              exception.printStackTrace();
95          }
96      }
97
98      // executa a aplicação
99      public static void main( String args[] )
100     {
101         TesteLookAndFeel aplicacao = new TesteLookAndFeel();
102         aplicacao.setDefaultCloseOperation(
103         JFrame.EXIT_ON_CLOSE );
104     }
105
106     // classe interna privativa para tratar eventos dos botões de opção
107     private class TratadorDeItens implements ItemListener {
108
109         // processa a seleção de aparência e comportamento
110         // feita pelo usuário
111         public void itemStateChanged( ItemEvent evento )
112         {
113             for ( int count = 0; count < radio.length; count++ )
114             {
115                 if ( radio[ count ].isSelected() ) {
116                     rotulo.setText( "Esta é a aparência " +
117                     strings[ count ] );
118                     comboBox.setSelectedIndex( count );
119                     mudeTheLookAndFeel( count );
120                 }
121             }
122         }
123     }
124 }
```

Capítulo 3

Bases de Estruturação das Interfaces Gráficas

Visto que agora já temos uma idéia espacial concebida, resta-nos analisar a anatomia das interfaces gráficas em Java, a qual baseia-se nos elementos que serão descritos nestas próximas seções.

3.1 Conteiners

Dão suporte a exibição e agrupamento de outros componentes, inclusive outros conteiners. Eles constituem a base onde os outros elementos são anexados. Precisamente, é o local onde podemos montar nossa aplicação.

Como veremos, em praticamente todos os nossos exemplos usamos um objeto da classe **Container** denominado contêiner. A ele atribuímos uma chamada ao método **getContentPane( )**, que devolve uma referência para o painel de conteúdo do aplicativo ou do applet. O painel de conteúdo compreende a área imediatamente inferior a barra de título de uma janela, extendendo-se até os limites da mesma.

A classe **Container** define o método **add(Component)**, para adicionar elementos, e **setLayout (LayoutManager)**, que configura um gerenciador de leiaute para gerir o posicionamento e dimensionamento dos mesmos.

Ressalta-se que a disposição dos elementos adicioandos a um contêiner obedece a ordem em que eles foram anexados e ao gerenciador de leiaute previamnete definido. Se um conteiner não é suficientemente dimensionado para acomodar os componentes anexados a ele, alguns ou todos os elementos GUI simplesmente não serão exibidos [1].

Qualquer programa que ofereça uma interface vai possuir pelo menos um conteiner [5], que pode ser :

- JFrame - janela principal do programa;
- JDialog - janela para diálogos;
- JApplet - janela para Applets.

3.1.1 JFrame

Esta classe define objetos que são frequentemente utilizadas para criar aplicativos baseados em GUI. Eles consistem em uma janela com barra de título e uma borda e fornecem o espaço para a GUI do aplicativo ser construída.

A classe **JFrame** é uma subclasse de **java.awt.Frame**, que por sua vez é subclasse de **java.awt.Window**. Pelo mecanismo de herança, nota-se que **JFrames** são um dos poucos componentes GUI do Swing que não são considerados de peso-leve, pois não são escritos completamente em Java. Sendo assim, quando possível, devemos devolver ao sistema os recursos ocupados pela janela, descartando-a. Frisamos que a janela de um programa Java faz parte do conjunto de componentes GUI da plataforma local e será semelhante às demais janelas, pois serve-se da bibilioteca gráfica do sistema em questão.

Para exibir um titulo na barra de título de uma **JFrame**, devemos chamar o construtor de superclasse de **JFrame** com o argumento **String** desejado, dessa forma:

```
1 super("Título da Barra")
```

A classe JFrame suporta três operações quando o usuário fecha a janela. Por *default*, a janela é removida da tela (ocultada) quando o usuário intervém indicando o seu fechamento. Isso pode ser controlado com o método **setDefaultCloseOperation(int)**, que utiliza como argumento as constantes da interface **WindowConstants** (pacote javax.swing) implementada por JFrame:

DISPOSE_ON_CLOSE: descarta a janela devolvendo os seus recursos ao sistema;

DO_NOTHING_ON_CLOSE: indica que o programador determinará o que fazer quando o usuário designar que a janela deve ser fechada;

HIDE_ON_CLOSE: *(o default)* a janela é ocultada, removida da tela;

EXIT_ON_CLOSE: determinamos que quando fechamos a JFrame, o aplicativo seja finalizado. Essa constante é definida na classe JFrame e foi introduzida na versão 1.3 da Plataforma Java.

A janela só será exibida na tela quando o programa invocar o método **setVisible(boolean)** com um argumento **true**, ou o método **show( )**. O tamanho da janela é configurado com uma chamada ao método **setSize(int x, int y)**, que define nos valores inteiros dos argumentos a largura e a altura da mesma. Se não chamarmos esse método, somente a barra de título será exibida.

Também podemos utilizar o método **pack( )**, que utiliza os *tamanhos preferidos* dos componentes anexados ao painel de conteúdo para determinar o tamanho da janela. Por tamanho preferido, entende-se uma chamada realizada pelos gerenciadores de leiaute ao método **getPreferredSize( )** de cada componente GUI. Esse método indica o melhor tamanho para os componentes. É herdado da classe **java.awt.Component**, de modo que todos

os objetos que derivem-se dessa classe podem responder a essa evocação. Ela devolve um objeto da classe **Dimension** (pacote java.awt).

Podemos fazer uso dos métodos **setMinimumSize(Dimension)** e **setMaximumSize(Dimension)**, que estabelecem os tamanhos extremos dos elementos. O componente não deveria ser maior que o tamanho máximo e nem menor que o mínimo. Entretando, esteja consciente de que certos gerenciadores de leiaute ignoram essa sugestão [2].

Todos os elementos tem um tamanho preferido *default*, como, por exemplo, um objeto **JPanel**, que tem altura e largura de 10 pixels. Se necessitarmos mudar esse tamanho *default*, devemos sobreescrever o método **getPreferredSize( )**, fazendo com que ele retorne um objeto **Dimension** que contenha a nova largura e altura do componente, ou usar o método **setPreferredSize(new Dimension(int x, int y))**.

No que concerce ao posicionamento, por *default*, o canto superior esquerdo da janela é posicionado nas coordenadas (0, 0) da tela, ou seja, no canto superior esquerdo. Podemos alterar essa característica com o método **setLocation(int x, int y)**.

Mais a frente, discutiremos os eventos geradados pela manipulação de janelas e como tratá-los.

JDesktopPane e JInternalFrame

São classes que fornecem suporte à criação de interfaces de multiplos documentos. Uma janela principal (pai) contém e gerencia outras janelas (filhas). A grande utilidade disso é que podemos visualizar vários documentos que estão sendo processados em paralelo ao mesmo tempo, facilitando a edição ou leitura dos mesmos. Veremos logo mais à frente a interface que é implemetada por nosso exemplo,[1] cujo código que obtém as funcionalidades mencionadas anteriormente é o seguinte:

```
1  // Demonstra JDesktopPane e JInternalFrame
2  import java.awt.*;
3  import java.awt.event.*;
4  import javax.swing.*;
5
6  public class TesteJDesktop extends JFrame {
7      private JDesktopPane desktop;
8
9      // configura a GUI
10     public TesteJDesktop()
11     {
12         super( "Testando JInternalFrame contida em" +
13         "uma JDesktopPane" );
14
15         // cria barra de menus
16         JMenuBar barra = new JMenuBar();
17
18         // cria menu "Arquivo"
19         JMenu arquivo = new JMenu( "Arquivo" );
20
21         // cria itens do menu "Arquivo"
```

[1] Julgamos conveniente mostrar a aparência desse componente, pois não o utilizaremos nos próximos exemplos, diferentemente de JFrames, que são a base da maioria deles.

```
22          JMenuItem novo = new JMenuItem( "Novo" );
23          JMenuItem sair = new JMenuItem( "Sair" );
24
25          // anexa os itens ao menu "Arquivo"
26          arquivo.add( novo );
27          arquivo.add( sair );
28
29          // anexa o menu "Arquivo" à barra de menus
30          barra.add( arquivo );
31
32          // anexa a barra de menus à janela do aplicativo
33          setJMenuBar( barra );
34
35          // configura a "desktop"
36          desktop = new JDesktopPane();
37          desktop.setBackground(Color.lightGray);
38          desktop.setToolTipText("Eu sou a JDesktopPane. " +
39          "Você pode utilizar meu menu.");
40          this.getContentPane().add( desktop );
41
42          // configura ouvinte para o item de menu "Novo"
43          novo.addActionListener(
44
45          // classe interna anônima para tratar eventos do
46          // item de menu "Novo"
47          new ActionListener() {
48
49              // exibe nova janela interna
50              public void actionPerformed( ActionEvent evento ) {
51
52                  // cria a janela interna
53                  JInternalFrame frame = new JInternalFrame(
54                  "Janela Interna", true, true, true, true );
55
56                  // obtém painél de conteúdo da janela interna
57                  Container container = frame.getContentPane();
58
59                  JanelaInterna interna = new JanelaInterna();
60
61                  // anexa ao painel de conteúdo da janela interna
62                  // um objeto da classe "JanelaInterna"
63                  container.add( interna, BorderLayout.CENTER );
64
65                  // configura o tamanho da janela interna com o tamanho
66                  // do seu conteúdo
67                  frame.pack();
68
69                  // anexa a janela interna à "Desktop" e a exibe
70                  desktop.add( frame );
71                  frame.setVisible( true );
72              }
73          }
74          );
75
76          // configura ouvinte para o item de menu "Sair"
77          sair.addActionListener(
78
79          // classe interna anônima para tratar eventos do item de menu "Sair"
80          new ActionListener() {
```

```
81
82              // encerra o aplicativo
83              public void actionPerformed( ActionEvent evento ) {
84
85                  System.exit( 0 );
86              }
87          }
88          );
89          // determina o tamanho da janela do aplicativo
90          setSize( 700, 600 );
91          // determina que o conteúdo anexado à janela seja exibido
92          setVisible( true );
93      }
94
95      // executa a aplicação
96      public static void main( String args[] )
97      {
98          TesteJDesktop aplicacao = new TesteJDesktop();
99
100         aplicacao.setDefaultCloseOperation(
101         JFrame.EXIT_ON_CLOSE );
102     }
103 }
104
105 class JanelaInterna extends JPanel{
106     private JTextArea areaTexto;
107
108     public JanelaInterna()
109     {
110         // cria uma área de texto
111         areaTexto = new JTextArea(25,25);
112         // configura mudança automática de linha
113         areaTexto.setLineWrap(true);
114         // determina que as mudança de linha seja definida pelas palavras
115         areaTexto.setWrapStyleWord(true);
116         // configura o texto a ser exibido
117         areaTexto.setText("Este material destina-se a usuários da liguagem " +
118         "Java que pretendem incluir interfaces gráficas em suas aplicações,"+
119         "sejam elas autônomas ou applets. Salientamos que é de grande valia ao " +
120         "usuário se este já estiver familiarizado com a Linguagem Java, pois " +
121         "o conteúdo desse material não explana conceitos básicos, nem discute a " +
122         "sintaxe da Linguagem.");
123
124         // adiciona barras de rolagem se o tamanho da
125         // da àrea de texto for insuficiente para exibir o texto
126         this.add(new JScrollPane(areaTexto));
127     }
128 }
```

Chamamos a atenção para o construtor da JInternalFrame (**new JInternalFrame(String, boolean, boolean, boolean, boolean)**) que nos seus cinco argumentos define, respectivamente:

- o título para a barra de título da janela interna;
- indica se ela pode ser redimensionada pelo usuário;
- indica se ela pode ser fechada pelo usuário;

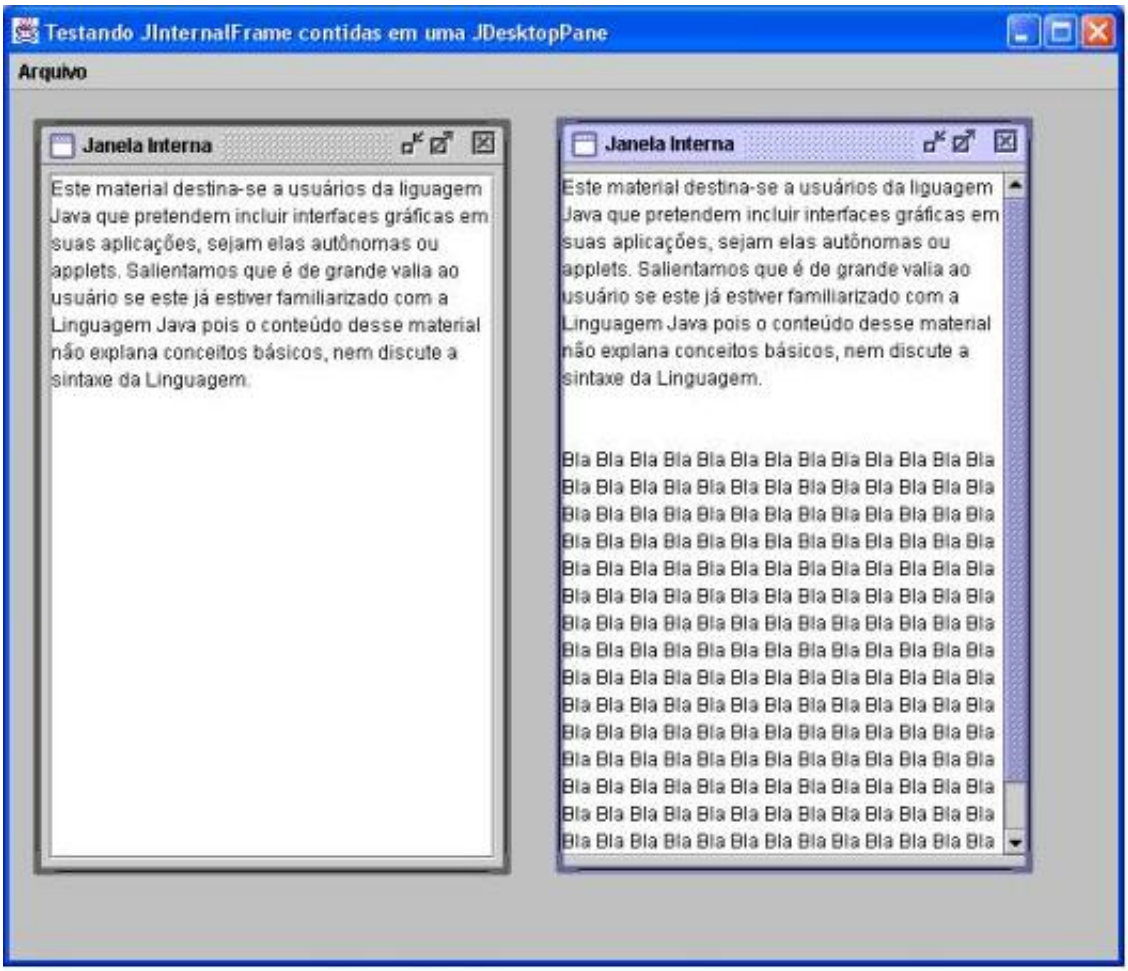


Figura 3.1: Interface do exemplo que usa JDesktopPane e JInternalFrame

- configura se ela poder ser maximizada pelo usuário;
- define se ela pode ser minimizada pelo usuário.

Como ocorre com objetos das classes JFrame e JApplet, um JInternalFrame tem um painel de conteúdo ao qual os componentes GUI podem ser anexados. Sabendo disso, criamos um objeto da classe "JanelaInterna" e o anexamos ao painel de conteúdo da JInternalFrame em nosso exemplo.

3.1.2 JDialog

Usamos a classe **JDialog**, que é susbclasse de **java.awt.Dialog** para criarmos caixas de diálogo elaboradas, embora mais limitados que as originadas por JFrames. Em prol da facilidade, a classe **JOptionPane**, que está definida no pacote de extensão **javax.swing**, oferece caixas de diálogo pré-definidas que permitem aos programas exibir simples mensagens para os usuários. Cada vez que usamos uma **JOptionPane** para implementar um diálogo, na verdade estamos usando uma **JDialog** nos bastidores. A razão é que **JOptionPane** são simplesmente um contêiner que pode automaticamente criar uma **JDialog** e anexa-la ao seu painel de conteúdo [2].

Embora esses diálogos sejam maneiras válidas de receber entrada do usuário e exibir a saída de um programa, suas capacidades são um tanto limitadas - o diálogo pode obter somente um valor por vez e só pode exibir uma mensagem. Mais usual é receber várias entradas de uma só vez, de modo que o usuário possa visualizar todos os campos de dados. A medida que formos avançando no conteúdo, o leitor será capaz de sanar tais deficiências usando novos componentes.

As caixa de diálogo podem ser configuradas como *modais* ou *não-modais*, valendo-se do método **setModal(boolean)**. As modais não permitem que qualquer outra janela do aplicativo seja acessada até que seja tratada a solicitação ou intervenção da caixa de diálogo. O comportamento oposto se observa nas não-modais. Os diálogos exibidos com a classe JOptionPane, por *defaut*, são diálogos modais. Além disso, podemos definir se o tamanho de uma JDialog é redimensionável, com o método **setResizable(boolean)**.

Obviamente, devido a diversidade de funcionalidades e de construtores, para usarmos todas as potencialidades devemos estudar profundamente as classes em questão. Abaixo mostraremos alguns métodos estáticos da classe **JOptionPane.**(todos são precedidos por **JOptionPane.**) e a sintaxe mais comumente utilizada para criarmos caixas de diálogo pré-definidas:

showInputDialog(String) Método usado para solicitar a entrada de algum dado em forma de **String**. Lembre-se que os valores recebidos devem ser atribuídos à variáveis do tipo **String** e convertidos para outros tipos caso desejarmos realizar operações sobre eles.

showMessageDialog(Component, Object, String, int, Icon) Método que exibe uma caixa de diálogo com texto, ícone, posicionamento e título definidos pelo programador.

O propósito do primeiro argumento é especifiar a janela-pai para a caixa de diálogo. Um valor **null** indica que a caixa de diálogo será exibida no centro da tela. No caso de nossa aplicação apresentar várias janelas, podemos especificar nesse argumento a janela-pai, de modo que a caixa de diálogo aparecerá centralizada sobre a janela-pai especifiacada, que necessariamente pode não corresponder ao centro da tela do computador.

O segundo argumento normalmente especifica o *String* a ser mostrado ao usuário. A caixa de diálogo comporta qualquer tamanho de *String*, já que a mesma é dimensionada automaticamente para acomodar o texto. Também é possível exibir longas saidas baseadas em texto,passando como argumento para o método um objeto da classe **JTextArea**.

O terceiro argumento denomina a barra de título. É opcional já que, se forem ignorados o terceiro e o quarto argumento, a caixa de diálogo exibirá uma mensagem do tipo INFORMATION_MESSAGE, com o texto "Message" na barra de título e um ícone de informação à esquerda da mensagem de texto.

O quarto argumento refere-se ao ícone que será exibido e ao tipo de diálogo de mensagem. Podemos fazer uso dos seguintes valores para as constantes [1]:

- JOptionPane.ERROR_MESSAGE Indica mensagem de erro ao usuário;

- JOptionPane.INFORMATION_MESSAGE Exibe uma mensagem com informações que podem ser dispensadas;
- JOptionPane.WARNING_MESSAGE Indica mensagem de advertência sobre algum problema em potencial;
- JOptionPane.QUESTION_MESSAGE Impõe uma mensagem que pergunta algo ao usuário;
- JOptionPane.PLAIN_MESSAGE Exibe um diálogo que simplesmente contém uma mensagem sem nenhum ícone.

No último argumento podemos definir um ícone (classe **Icon**) que será exibido junto da caixa de diálogo. Ele deve residir no mesmo diretório da aplicação ou teremos que especificar o caminho.

showOptionDialog(Component, Object, String, int, int, Icon, Object[ ],Object) Este método apresenta tudo o que foi descrito no método precedente a ainda suporta a criação de outros botões, para opções personalizadas. Como você pode observar, os três primeiros argumentos são os mesmos do método precedente.

O quarto refere-se ao conjunto de botões que aparecem abaixo do diálogo. Escolha um a partir do conjunto de valores padrão:

- DEFAULT_OPTION, YES_NO_OPTION;
- YES_NO_CANCEL_OPTION, OK_CANCEL_OPTION.

O quinto argumento aqui é o mesmo que o o quarto descrito no método precedente, ou seja, determina o tipo de mensagem exibida no diálogo.

O sexto, refere-se ao ícone que será exibido no diálogo.

O argumento seqüente determina que os botões de opção apareçam abaixo do diálogo. Geralmente, especificamos um array de Strings para rotular os botões, sendo que cada elemento do array define um botão.

Cada vez que selecionamos um botão, um valor inteiro que corresponde ao índice do array é retornado pela JOptionPane. Você verá no exemplo[2] logo adiante que podemos atribuir esse valor a uma variável e posteriormente pode-se implementar um processo de decisão que corresponda à escolha feita pelo usuário.

Finalmente, o último argumento define o botão *default* a ser selecionado.

```
1  // Demonstra JOPtionPane
2
3  import java.awt.*;
4  import javax.swing.*;
5
6  public class TesteJOptionPane extends JFrame {
7      String nome;
8      String sobrenome;
9      String todoNome;
10     String stringNumero1;
```

[2] O *feedback* de todos esses métodos pode ser visualizado na execução do código.

```
11      String stringNumero2;
12      int valorInteiroNumero1;
13      int valorInteiroNumero2;
14      int soma, valor;
15      JTextArea areaTexto;
16      JLabel selecao;
17      Icon seta_90 = new ImageIcon( "figuras/seta_90.gif" );
18      final JDialog dialogo;
19      String[] opcoes = {"Sim, plenamente","Não, é muito chato!",
20      "Estou tentando...","Já sei tudo!"}; // titulo dos botões
21
22      public TesteJOptionPane()
23      {
24          setTitle("Testando JOptionPanes e JDialogs");
25          setSize( 500, 300 );
26          setVisible( true );
27
28          // lê o prompt e armazena o string na variável
29          nome = JOptionPane.showInputDialog( "Digite seu nome" );
30
31          // lê o prompt e armazena o string na variável
32          sobrenome = JOptionPane.showInputDialog( "Digite seu sobrenome" );
33
34          // adiciona os strings
35          todoNome = nome +" "+ sobrenome;
36
37          // lê o primeiro número e armazena o string na variável
38          stringNumero1 = JOptionPane.showInputDialog( "Digite " +
39          "um numero inteiro" );
40
41          // lê o segundo número e armazena o string na variável
42          stringNumero2 = JOptionPane.showInputDialog( "Digite " +
43          "outro numero inteiro" );
44
45          // converte os strings para valores inteiros
46          valorInteiroNumero1 = Integer.parseInt( stringNumero1 );
47          valorInteiroNumero2 = Integer.parseInt( stringNumero2 );
48
49          // adiciona os valores inteiros
50          soma = valorInteiroNumero1 + valorInteiroNumero2;
51
52          areaTexto = new JTextArea();
53          areaTexto.setText("Seu Nome\tSeu Sobrenome\n" + nome + "\t"
54          + sobrenome);
55
56          // mostra o resultado das adições no centro da janela
57          // do aplicativo (usa ícone personalizado)
58          JOptionPane.showMessageDialog(this, "Seu nome completo é: "
59          + todoNome, "Nome Completo", JOptionPane.PLAIN_MESSAGE, seta_90  );
60
61          // mostra o resultado das adições em uma JTextArea no
62          // centro da janela do aplicativo
63          JOptionPane.showMessageDialog(this, areaTexto, "Nome Completo",
64          JOptionPane.INFORMATION_MESSAGE);
65
66          // mostra o resultado das adições no centro da tela com título default
67          JOptionPane.showMessageDialog(this, "A soma é: " + soma );
68
69          // demais tipos de mensagens
```

```
70          JOptionPane.showMessageDialog(null, "Qualquer Mensagem de Alerta",
71          "ATENÇÃO!", JOptionPane.WARNING_MESSAGE );
72
73          JOptionPane.showMessageDialog(this, "Qualquer Mensagem Informativa",
74          "Você sabia que...", JOptionPane.INFORMATION_MESSAGE );
75
76          JOptionPane.showMessageDialog(null, "Qualquer Mensagem de Erro",
77          "AVISO DO SISTEMA", JOptionPane.ERROR_MESSAGE );
78
79          JOptionPane.showMessageDialog(this, "Qualquer Mensagem Interrogativa",
80          "Responda!", JOptionPane.QUESTION_MESSAGE );
81
82          // caixa de diálogo com botões de opções personalizadas de escolha
83          // opcoes[0] define o botão selecionado por default
84          int n = JOptionPane.showOptionDialog(
85          this, // o aplicativo é a janela pai
86          "Você está aprendendo com este material?", // texto mostrado ao usuário
87          "Avaliação do trabalho ",      // título da barra de título
88          JOptionPane.DEFAULT_OPTION,    // conjunto de botões
89          JOptionPane.QUESTION_MESSAGE,  // tipo de mensagem exibida
90          null, // indica que não usamos ícone personalizado
91          opcoes,  // cada  botão é um elemento desse array
92          opcoes[0]); // botão default a ser selecionado
93
94          selecao = new JLabel("Você selecionou " + "\"" + opcoes[n] + "\""
95          + " na caixa anterior");
96
97          dialogo = new JDialog( this ,"Sou uma JDialog modal", true);
98          dialogo.setSize(400,200);
99          dialogo.setContentPane(selecao);
100         dialogo.setVisible(true);
101     }
102
103     // inicia a execução do aplicativo Java
104     public static void main( String args[] )
105     {
106         TesteJOptionPane aplicacao = new TesteJOptionPane();
107
108         aplicacao.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
109     }
110 }
```

3.1.3 JApplet

Applet são programas Java que podem ser embutidos em documentos HTML. Quando um navegador carrega uma página da Web que contém um applet, ele baixa esse applet e o executa (o navegador é chamado de contêiner de applets. Os navegadores da World Wide Web que suportam applets esperam que nossos applets tenham certos atributos e comportamentos. A classe **JApplet** (pacote javax.swing.JApplet) fornece todas essas capacidades, bastando ao programador construir classes que extendam-se dela.

Como nosso prósito é compor interfaces, é necessário esclarecer que, com applets continuamos trabalhando da mesma forma, ou seja, devemos anexar nossos componentes ao painel de conteúdo do applet, distribuindo-os com os gerenciadores de leiaute. A peculiaridade é que dimensionamos o tamanho do applet num arquivo de texto plano salvo com extensão HTML, como

mostrado abaixo:

```
1 <HTML>
2 <applet code = "classe do applet.class" width = "100" height = "100">
3 </applet>
4 </HTML>
```

Este arquivo, que deve ser armazenado no mesmo local que o applet reside, indica qual applet deve ser carregado e executado pelo contêiner de applets, bem como o tamanho do mesmo. Note que o nome do arquivo é formado pela classe já compilada.

Outro ponto relevante é que em applets, devemos definir a inicialização dos componentes GUI e anexa-los ao painel de conteúdo no escopo do método **public void init()**, que se assemelha ao construtor de um aplicativo independente. Este método é chamado automaticamente pelo conteiner de applets, o qual carrega o applet, inicializa as variáveis de instância e cria a interface gráfica.

Embora applets sejam muito interessates, por fugir do escopo desta apostila, não iremos explicitar os demais métodos, bem como maiores datalhes sobre a implementação de applets.

3.2 Painéis

São áreas que comportam outros componentes, inclusive outros painéis[3]. Em outras palavras, são elementos que fazem a intermediação entre um contêiner e os demais GUI anexados. São criados com a classe **JPanel**, que é derivada da classe **Container**. As JPanel possibilitam a criação de subconjuntos num contêiner de forma à garantir um maior domínio sobre todas as áreas da interface. Aqui, o jargão "dividir para conquistar" se justifica plenamente.

A classe JPanel não tem painel de conteúdo como applets e JFrames, assim, os elementos devem ser diretamente adicioandos ao objeto painel.

Além de agregar um conjunto de componentes GUI para fins de leiaute, podemos criar áreas dedicadas de desenho e áreas que recebem eventos do mouse. Para isso, devemos implementar subclasses de JPanel e fornecer-lhes tais capacidades sobrescrevendo determinados métodos que não serão mencioandos nesse curso. Cabe lembrar que JPanel não suportam eventos convencionais suportados por outros componentes GUI, como botões, campos de texto e janelas. Apesar disso, JPanel são capazes de reconhecer eventos de nível mais baixo, como eventos de mouse e de teclas.

3.3 Gerenciadores de Leiaute

Organizam os componetes GUI de um contêiner e, para tal, devem ser configurados antes que qualquer membro seja anexado ao painel de conteúdo.

[3]Inúmeros exemplos deste material explicitam o uso de painéis, de modo que nenhum exemplo especial foi criado. Assim sendo, aconselhamos o leitor a verificar nas implementações o uso dos mesmos.

Diríamos que os gerenciadores trabalham como arquitetos, que, após algumas definições do programador, distribuim os elementos no espaço que foi incumbido a eles. Sua utilização poupa o programador da preocupação de posicionar precisamente cada componente.

Embora somente seja permitido apenas um gerenciador de leiaute por contêiner, na elaboração de interfaces complexas, que, com freqüência, consistem em um contêiner onde estão anexados múltiplos painéis com diversos componentes, podemos usar um gerenciador de leiaute por painel, desse modo, distribuindo os elementos de uma forma mais refinada e precisa.

Vejamos cada um deles e suas metodologias:

3.3.1 FlowLayout

E o gerenciador mais elementar. Distribui os componentes pelo contêiner seqüencialmente, da esquerda para a direita e na ordem em que foram adicionados. Seu comportamento assemelha-se a um editor de texto, já que quando se alcança a borda do contêiner, os membros são alocados na próxima linha.

A classe **FlowLayout** permite que os componentes sejam alinhados à esquerda, a direita e centralizados (padrão). Exercite isso em nosso exemplo, clicando nos botões específicos para cada direção de alinhamento. Ressalta-se que este é o gerenciador *default* dos **JPanel**s.

Figura 3.2: Interface do exemplo que usa FlowLayout

```
1  // Demonstra os alinhamentos possíveis do gerenciador FlowLayout
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class TesteFlowLayout extends JFrame {
8      private JButton esquerda, centro, direita;
9      private Container container;
10     private FlowLayout layout;
11     private JLabel pet;
12     private Icon logoPet = new ImageIcon( "figuras/logo.jpg" );
13
14     // configura a GUI e registra ouvintes dos botões
15     public TesteFlowLayout()
```

```
16      {
17          super( "Testando o gerenciador FlowLayout" );
18
19          pet = new JLabel("");
20          pet.setIcon(logoPet);
21
22          // obtém painél de conteúdo
23          container = getContentPane();
24          layout = new FlowLayout();
25
26          // e configura o leiaute
27          container.setLayout( layout );
28          container.add( pet );
29
30          // cria o botão "Esquerda" e registra ouvinte
31          esquerda = new JButton( "Esquerda" );
32          esquerda.addActionListener(
33
34          // classe interna anônima
35          new ActionListener() {
36
37              // processa eventos do botão "Esquerda"
38              public void actionPerformed( ActionEvent evento )
39              {
40                  layout.setAlignment( FlowLayout.LEFT );
41
42                  // realinha os components que foram anexados
43                  layout.layoutContainer( container );
44              }
45          }
46
47          );
48
49          container.add( esquerda );
50
51          // cria o botão "Centro" e registra ouvinte
52          centro = new JButton( "Centro" );
53          centro.addActionListener(
54
55          // classe interna anônima
56          new ActionListener() {
57
58              // processa eventos do botão "Esquerda"
59              public void actionPerformed( ActionEvent evento )
60              {
61                  layout.setAlignment( FlowLayout.CENTER );
62
63                  // realinha os components que foram anexados
64                  layout.layoutContainer( container );
65              }
66          }
67          );
68
69          container.add( centro );
70
71          // cria o botão "Direita" e registra ouvinte
72          direita = new JButton( "Direita" );
73          direita.addActionListener(
74
```

```
75          // classe interna anônima
76          new ActionListener() {
77
78              // processa eventos do botão "Direita"
79              public void actionPerformed( ActionEvent evento )
80              {
81                  layout.setAlignment( FlowLayout.RIGHT );
82
83                  // realinha os components que foram anexados
84                  layout.layoutContainer( container );
85              }
86          }
87          );
88
89          container.add( direita );
90          setSize( 250, 250 );
91          setVisible( true );
92      }
93
94      // executa a aplicação
95      public static void main( String args[] )
96      {
97          TesteFlowLayout aplicacao = new TesteFlowLayout();
98
99          aplicacao.setDefaultCloseOperation(
100         JFrame.EXIT_ON_CLOSE );
101     }
102 }
```

3.3.2 BorderLayout

O painel de conteúdo utiliza como *default* esse gerenciador. Suas virtudes residem na possibilidade de organizar os componentes GUI em cinco regiões: NORTH, SOUTH, EAST, WEST e CENTER. Até cinco componentes podem ser adicionados (em qualquer ordem) a um contêiner ou painel que esteja configurado com esse gerenciador, sendo que cada um deverá ocupar uma região. Acarreta-se que, no caso de mais de um elemento ser adicionado à mesma área, somente o último anexado será visível [1].

Os componentes colocados nas regiões NORTH e SOUTH estendem-se horizontalmente para os lados do contêiner e tem a mesma altura que o componente mais alto anexado em uma dessas regiões.

As regiões EAST e WEST expandem-se verticalmente entre as regiões NORTH e SOUTH e tem a mesma largura que o componente mais largo colocado nessas regiões.

O elemento colocado na região CENTER expande-se para ocupar todo o espaço restante no leiaute. Se a região NORTH ou SOUTH não for ocupada, os membros das regiões EAST, CENTER e WEST expandem-se verticalmente para preencher o espaço restante. Caso a região CENTER não seja ocupada, a área permanecerá vazia, pois os outros componentes não se expandem para preencher o espaço que sobra [1].

Não implementamos nenhum código especial aqui, pois a aplicação desse gerenciador pode ser vista em muitos de nossos exemplos. Cita-se a organização de um painel que constitui uma "carta" do aplicativo que demonstra

o gerenciador CardLayout[4].

Figura 3.3: Interface do exemplo que usa BorderLayout para gerenciar a "Última Carta".

3.3.3 GridLayout

Este é um dos gerenciadores mais interessantes até aqui, pois a área sob sua jurisdição é dividida em linhas e colunas convenientes, formando uma grade, que, à medida que os componentes são anexados, é preenchida da célula superior esquerda em direção à direita. Após preenchida a linha, o processo continua na linha imediatamente inferior. Cada membro em um **GridLayout** tem a mesma largura e comprimento. Podemos ver a aplicação desse gerenciador na figura 3.3, logo acima. Repare como os botões foram organizados. Isso foi conseguido com a seguinte implementação, que é um fragmento do código do exemplo precedente:

```
1 // configura a àrea do painel "AreaDosBotoes"
2 // com quatro colunas e uma linha
3 JPanel AreaDosBotoes = new JPanel();
4 AreaDosBotoes.setLayout( new GridLayout( 4, 1 ) );
```

3.3.4 BoxLayout

Permite que os componentes GUI sejam organizados da esquerda para direita (ao longo do eixo x) ou de cima para baixo (ao longo do eixo y) em um contêiner ou painel. A classe **Box** fornece métodos estáticos para criarmos caixas horizontais ou verticais que podem ser usadas para acomodar botões por exemplo, sendo que o gerenciador *default* do contêiner criado é **BoxLayout**. Também disponibiliza métodos que agregam outras características peculiares ao contêiner, como, por exemplo [1]:

createVerticalStrut(int) :

createHorizontalStrut(int) : Adicionam um suporte vertical ou horizontal ao contêiner. Esse suporte é um componente invisível e tem uma altura fixa em pixels (que é passada no argumento). É utilizado para

[4]Você verá que podemos "empilhar" painéis que são organizados individualmete por gerenciadores diferentes em um monte, que por sua vez é gerido pelo gerenciador CardLayout.

garantir uma quantidade fixa de espaço entre os componentes GUI, caso a janela seja redimensionada.

createVerticalGlue() :

createHorizontalGlue() : Adicionam cola vertical ou horizontal ao contêiner. A cola é um componente invisível, sendo utilizada em componentes GUI de tamanho fixo. Ela mantém uniforme o espaçamento entre membros de um contêiner, normalmente deslocando o espaço extra oriundo do redimensionamento da janela à direita do último componente GUI horizontal, ou abaixo do último componente GUI vertical.

createRigidArea(new Dimension(x, y)) : É um elemento invisível que tem altura e larguras fixas. O argumento para o método é um objeto **Dimension**, que especifica as dimensões da área rígida. Tal área não sofre perturbação quando a janela é redimensionada.

createGlue() : Mantém uniforme o espaçamento entre os membros de um contêiner, se expandindo ou contraindo conforme o tamanho da Box.

Manipule a janela do aplicativo que testa esse gerenciador para ter um *feedback* referente aos métodos acima descritos.

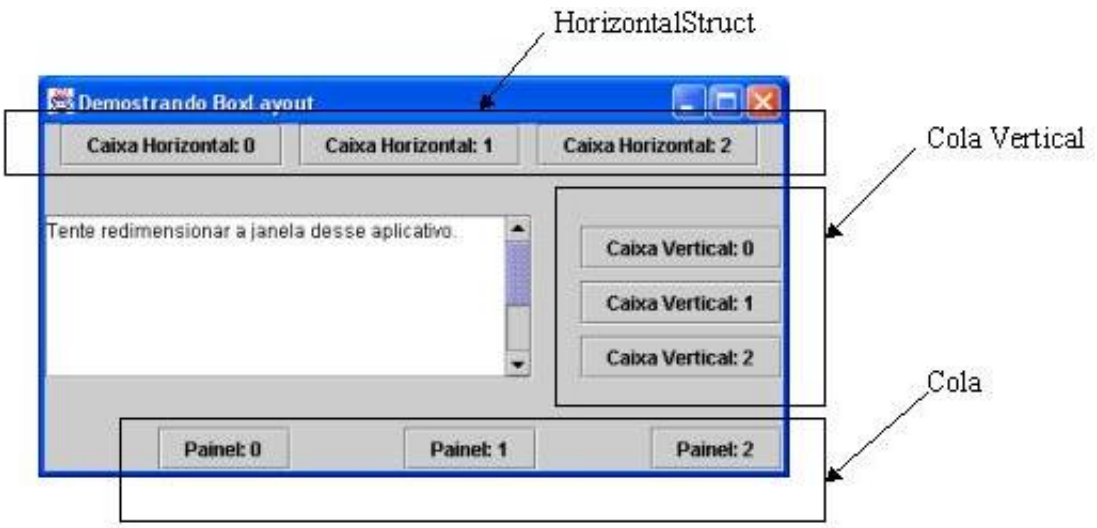


Figura 3.4: Interface do exemplo que usa BoxLayout

```
1  // Demonstra BoxLayout
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class TesteBoxLayout extends JFrame {
8      private JTextArea texto;
9
10     // configura a GUI
11     public TesteBoxLayout()
```

```
12      {
13          // texto da barra de título
14          super( "Demostrando BoxLayout" );
15          final int TAMANHO = 3;
16
17          // obtém painel de conteúdo
18          Container container = getContentPane();
19          // configura seu layout com BorderLayout,
20          // 30 pixels de espaçamento vertical e
21          // horizontal entre os componentes
22          container.setLayout( new BorderLayout(30,30) );
23
24          // cria conteiners Box configurados com o leiaute
25          // default BoxLayout
26          Box boxes[] = new Box[ 2 ];
27
28          // cria área de texto com o tamanho dos argumentos
29          texto = new JTextArea(10,15);
30          // configura mudança automatica de linha
31          texto.setLineWrap(true);
32
33          // retorna um conteiner Box e o configura
34          // como caixa horizontal
35          boxes[ 0 ] = Box.createHorizontalBox();
36          // retorna um conteiner Box e o configura
37          // como caixa vertical
38          boxes[ 1 ] = Box.createVerticalBox();
39
40          for ( int count = 0; count < TAMANHO; count++ ){
41              // cria suporte horizontal e configura em
42              // 10 pixels o espaço entre botões
43              boxes[ 0 ].add(Box.createHorizontalStrut(10));
44              // adiciona botões à boxes[0]
45              boxes[ 0 ].add( new JButton( "Caixa Horizontal: "
46              + count ) );
47          }
48
49          for ( int count = 0; count < TAMANHO; count++ ) {
50              // cria cola vertical, que gerencia a distribuição
51              // de espaços entre botões
52              boxes[ 1 ].add(Box.createVerticalGlue());
53              // adiciona botões à boxes[1]
54              boxes[ 1 ].add( new JButton( "Caixa Vertical: "
55              + count ) );
56          }
57
58          // cria painel
59          JPanel painel = new JPanel();
60          // e o configura na horizontal com o leiaute BoxLayout
61          painel.setLayout(new BoxLayout( painel, BoxLayout.X_AXIS ) );
62
63          for ( int count = 0; count < TAMANHO; count++ ) {
64              // cria cola, que mantém os botões uniformemente
65              // distribuídos no painel caso ele seja redimensionado
66              painel.add(Box.createGlue());
67              // adiciona botões ao painel
68              painel.add( new JButton( "Painel: " + count ) );
69          }
70
```

```
71          // anexa painéis às regiões do conteiner
72          container.add( boxes[ 0 ], BorderLayout.NORTH );
73          container.add( boxes[ 1 ], BorderLayout.EAST );
74          container.add( new JScrollPane(texto), BorderLayout.CENTER );
75          container.add( painel, BorderLayout.SOUTH );
76
77          setSize( 470, 250 );  // dimensiona a janela
78          setVisible( true );   // exibe a janela
79      }
80
81      // executa a aplicação
82      public static void main( String args[] )
83      {
84          TesteBoxLayout aplicacao = new TesteBoxLayout();
85
86          // configura o encerramento da aplicação
87          aplicacao.setDefaultCloseOperation(
88          JFrame.EXIT_ON_CLOSE );
89      }
90  }
```

3.3.5 CardLayout

A serventia desse gerenciador é que ele organiza os componentes como se fossem cartas de um baralho. Qualquer "carta" pode ser exibida na parte superior da pilha, a qualquer momento, valendo-se dos métodos da classe **CardLayout**. Cada "carta" é normalmente um contêiner, como um painel, que pode utilizar qualquer gerenciador de leiaute. No exemplo que refere-se a esse gerenciador, a "Primeira Carta" foi configurada com o gerenciador BorderLayout, assim como a "Terceira Carta", sendo que em a cada região dessa última foi anexado um painel contendo o nome da respectiva região. Já a "Segunda Carta" usa o gerenciador FlowLayout.

Figura 3.5: Interface do exemplo que usa CardLayout

```
1  // Demonstra CardLayout
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class TesteCardLayout extends JFrame
8  implements ActionListener {
```

```
9
10      private CardLayout gerenciadorDeCartas;
11      private JPanel monte;
12      private JButton botao[];
13      private String nomes[] = { "Primeira Carta", "Segunda Carta",
14      "Carta Anterior", "Última Carta" };
15
16      // configura a GUI
17      public TesteCardLayout()
18      {
19          // texto da barra de título
20          super( "Testando CardLayout " );
21
22          // obtém painel de conteúdo
23          Container container = getContentPane();
24
25          // cria um JPanel
26          monte = new JPanel();
27          // e o configura com CardLayout
28          gerenciadorDeCartas = new CardLayout();
29          monte.setLayout( gerenciadorDeCartas );
30
31          // configura rótulo e figura para a "carta1"
32          Icon logoPet = new ImageIcon( "figuras/logo.jpg" );
33          JLabel label1 = new JLabel(
34          "Esta é a Primeira Carta",SwingConstants.CENTER);
35          JLabel figura = new JLabel("");
36          figura.setIcon(logoPet);
37          figura.setHorizontalAlignment(SwingConstants.CENTER);
38
39          // cria a "carta1" e a adiciona ao JPanel "monte"
40          JPanel carta1 = new JPanel();
41          carta1.setLayout(new BorderLayout());
42          carta1.add( label1,BorderLayout.NORTH );
43          carta1.add( figura,BorderLayout.CENTER );
44          monte.add( carta1, label1.getText() ); // adicionando ao "monte"
45
46          // configura a "carta2" e a adiciona ao JPanel "monte"
47          JLabel label2 = new JLabel(
48          "Esta é a Segunda Carta", SwingConstants.CENTER );
49          JPanel carta2 = new JPanel();
50          carta2.setLayout(new FlowLayout());
51          carta2.setBackground( Color.orange );
52          carta2.add( label2 );
53          monte.add( carta2, label2.getText() ); // adicionando ao "monte"
54
55          // configura a "carta3" e a adiciona ao JPanel "monte"
56          JLabel label3 = new JLabel( "Esta é a Terceira Carta" );
57          JPanel carta3 = new JPanel();
58          carta3.setLayout( new BorderLayout() );
59          carta3.add( new JButton( "Região Norte" ), BorderLayout.NORTH );
60          carta3.add( new JButton( "Região Oeste" ), BorderLayout.WEST );
61          carta3.add( new JButton( "Região Leste" ), BorderLayout.EAST );
62          carta3.add( new JButton( "Região Sul" ), BorderLayout.SOUTH );
63          carta3.add( label3, BorderLayout.CENTER );
64          monte.add( carta3, label3.getText() ); // adicionando ao "monte"
65
66          // cria e aloca os botões que controlarão o "monte"
67          JPanel AreaDosBotoes = new JPanel();
```

```
68          // configura a àrea do painel "AreaDosBotoes"
69          // com quatro colunas e uma linha
70          AreaDosBotoes.setLayout( new GridLayout( 4, 1 ) );
71          botao = new JButton[ nomes.length ];
72
73          for ( int count = 0; count < botao.length; count++ ) {
74              botao[ count ] = new JButton( nomes[ count ] );
75              // registra a aplicação para tratar os eventos
76              // de precionamento dos botões
77              botao[ count ].addActionListener( this );
78              AreaDosBotoes.add( botao[ count ] );
79          }
80
81          // adiciona o JPanel "monte" e JPanel "botões" ao conteiner
82          container.add( AreaDosBotoes, BorderLayout.WEST );
83          container.add( monte, BorderLayout.CENTER );
84
85          setSize( 490, 200 ); // dimensiona a janela
86          setVisible( true );  // exibe a janela
87      }
88
89      // trata os eventos dos botões fazendo a troca das cartas
90      public void actionPerformed( ActionEvent evento )
91      {
92          // mostra a primeira carta
93          if ( evento.getSource() == botao[ 0 ] )
94              gerenciadorDeCartas.first( monte );
95
96          // mostra a próxima carta
97          else if ( evento.getSource() == botao[ 1 ] )
98              gerenciadorDeCartas.next( monte );
99
100         // mostra a carta anterior
101         else if ( evento.getSource() == botao[ 2 ] )
102             gerenciadorDeCartas.previous( monte );
103
104         // mostra a última carta
105         else if ( evento.getSource() == botao[ 3 ] )
106             gerenciadorDeCartas.last( monte );
107     }
108
109     // executa a aplicação
110     public static void main( String args[] )
111     {
112         TesteCardLayout aplicacao = new TesteCardLayout();
113
114         // configura o encerramento da aplicação
115         aplicacao.setDefaultCloseOperation(
116         JFrame.EXIT_ON_CLOSE );
117     }
118 }
```

3.3.6 GridBagLayout

Finalmente chegamos ao mais complexo e poderoso dos gerenciadores de leiaute predefinidos. Você notará uma grande semelhança entre este gerenciador e o **GridLayout**, já que ambos utilizam uma grade para dispor os

componentes GUI. No entanto, o **GridBagLayout** é muito mais flexível e admite variações no tamanho dos elementos, tanto no número de linhas, como no de colunas, isto é, os componentes podem ocupar múltiplas linhas ou colunas.

Inicialmente, propomos que o leitor esboce um rascunho da GUI em um papel e depois trace linhas e colunas sobre ele, respeitando as extremidades dos componentes que deseja criar, de modo que cada elemento fique incluso em uma ou mais células resultante da intersecção entre linhas e colunas. Posteriormente, deve-se numerar as linhas e colunas, iniciando a contagem pelo zero. Isso é valido para definirmos os "endereços" nos quais os membros serão alocados.

Veja um esboço do que foi anteriormente descrito:

Figura 3.6: Interface do exemplo que usa GridBagLayout

Para utilizarmos esse gerenciador, devemos instanciar um objeto **GridBagConstraints**, que vai fazer o trabalho de distribuir os componentes GUI, baseando-se nas restrições das seguintes variáveis de instância da classe **GridBagConstraints**:

gridx define a coluna em que o canto superior esquerdo do componente será colocado.

gridy define a linha em que o canto superior esquerdo do componente será colocado.

gridwidth determina o número de colunas que o componente ocupa.

gridheight define o número de linhas que o componente ocupa.

fill especifica quanto da área destinada ao componente (o número de linhas e colunas) é ocupada. A essa variável atribui-se uma das seguintes constantes de GridBagConstraints:

- NONE indica que o elemento não crescerá em nenhuma direção. É o valor *default*.
- VERTICAL sinaliza que o elemento crescerá verticalmente.
- HORIZONTAL informa que o elemento crescerá horizontalmente.

- BOTH indica que o elemento crescerá em ambas as direções.

anchor especifica a localização do elemento na área a ele destinada quando este não preencher a área inteira. A essa variável atribui-se uma das seguintes constantes de GridBagConstraints:

- NORTH, NORTHEAST;
- EAST, SOUTHEAST;
- SOUTH, SOUTHWEAST;
- WEST, NORTHWEST;
- CENTER, que é o valor *default*

weightx define se o componente irá ocupar espaço extra horizontal, caso a janela seja redimensionada. O valor zero indica que o elemento não se expande horizontalmente por conta própria. Porém, se um membro da mesma coluna possuir a **weightx** configurada com um valor maior que zero, nosso elemento crescerá horizontalmente na mesma proporção que os outros membros dessa coluna. Isso ocorre porque cada componente deve ser mantido na mesma linha e coluna que foi endereçado originalmente.

weighty define se o componente irá ocupar o espaço extra vertical, oriundo do redimensionamento da janela. O valor zero indica que o elemento não se expande verticalmente por conta própria. Porém, se um membro da mesma linha possuir a **weighty** configurada com um valor maior que zero, nosso elemento crescerá verticalmente na mesma proporção que os outros membros dessa linha. Veja isso na pratica com o exemplo "TesteGridBagLayout".

Para as duas últimas variáveis citadas, infere-se que valores de peso maiores acarretam maior abrangência do espaço adicional a esses componentes em detrimento a outros membros que portem valores inferiores. Se todos os componentes forem configurados com zero, os mesmos aparecerão amontoados no meio da tela quando a janela for redimensionada. Ressalta-se que somente valores positivos são aceitos.

```
1  // Demonstra GridBagLayout
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class TesteGridBagLayout extends JFrame {
8      private Container container;
9      private GridBagLayout layout;
10     private GridBagConstraints restricoes;
11
12     // configura a GUI
13     public TesteGridBagLayout()
14     {
15         super( "Testando GridBagLayout" );
16
```

```
17          // obtém painél de conteúdo
18          container = getContentPane();
19          layout = new GridBagLayout();
20
21          // e o configura com GridBagLayout
22          container.setLayout( layout );
23
24          // criação do objeto que gerencia o posicionamento
25          // dos componentes no conteiner
26          restricoes = new GridBagConstraints();
27
28          // cria os componenetes da GUI
29          JTextArea areaDeTexto = new JTextArea( "Esse GUI não tem" +
30          " funcionalidade nenhuma!");
31          JLabel rotulo = new JLabel( "Redimensione a janela");
32
33          String bandas[] = { "Metallica", "Iron Maiden", "U2" };
34          JComboBox comboBox = new JComboBox( bandas );
35
36          JTextField textField = new JTextField( "Eu sou um JTextField" );
37          JButton botao1 = new JButton( "Abrir" );
38          JButton botao2 = new JButton( "Salvar" );
39          JButton botao3 = new JButton( "Imprimir" );
40
41          /****************ANEXANDO COMPONENTES**********************/
42
43          // areaDeTexto
44          // "weightx" e "weighty" são ambos zero: o valor default
45          // "anchor" para todos os componentes é CENTER: o valor default
46          // preenchimneto é BOTH
47          restricoes.fill = GridBagConstraints.BOTH;
48          adicionarComponente( areaDeTexto, 1, 0, 1, 2 );
49
50          // comboBox
51          // "weightx" e "weighty" são ambos zero: o valor default
52          // preenchimneto é HORIZONTAL
53          adicionarComponente( comboBox, 0, 0, 1, 1 );
54
55                  // botao "Abrir"
56                  // "weightx" e "weighty" são ambos zero: o valor default
57                  // preenchimneto muda de BOTH para HORIZONTAL
58                  restricoes.fill = GridBagConstraints.HORIZONTAL;
59                  adicionarComponente( botao1, 0, 1, 2, 1 );
60
61          // botao "Salvar"
62          restricoes.weightx = 1000;  // pode se extender horizontalmente
63          restricoes.weighty = 1;     // pode se extender verticalmente
64          // preenchimneto muda de HORIZONTAL para BOTH
65          restricoes.fill = GridBagConstraints.BOTH;
66          adicionarComponente( botao2, 1, 1, 1, 1 );
67
68          // botao "Imprimir"
69          // preenchimneto é BOTH
70          restricoes.weightx = 500;  // pode se extender horizontalmente
71          restricoes.weighty = 0.5;  // pode se extender verticalmente
72          adicionarComponente( botao3, 1, 2, 1, 1 );
73
74          // textField
75          // "weightx" e "weighty" são ambos zero: o valor default
```

```
76          // preenchimneto é BOTH
77          adicionarComponente( textField, 3, 0, 3, 1 );
78
79          // rotulo
80          // "weightx" e "weighty" são ambos zero: o valor default
81          // preenchimneto é BOTH
82          adicionarComponente( rotulo, 2, 1, 2, 1 );
83
84          setSize( 450, 150 );
85          setVisible( true );
86      }
87
88      // método que ativa as restrições e distribui os componentes
89      private void adicionarComponente( Component componente,
90      int linha, int coluna, int largura, int altura )
91      {
92          // configura gridx e gridy
93          restricoes.gridx = coluna;
94          restricoes.gridy = linha;
95
96          // configura gridwidth e gridheight
97          restricoes.gridwidth = largura;
98          restricoes.gridheight = altura;
99
100         // configura restricoes e anexa cada componente
101         layout.setConstraints( componente, restricoes );
102         container.add( componente );
103     }
104
105     // executa a aplicação
106     public static void main( String args[] )
107     {
108         TesteGridBagLayout aplicacao = new TesteGridBagLayout();
109
110         aplicacao.setDefaultCloseOperation(
111         JFrame.EXIT_ON_CLOSE );
112     }
113 }
```

Uma outra maneira de gerenciar o leiaute de um contêiner é atribuir ao método **setLayout(LayoutManager)** um argumento **null** e depois ajustar o posicionamento em x, y, bem como a largura e altura de cada componente com o método algumComponente.**setBounds(int, int, int, int)**. Os argumentos obedecem a ordem citada acima. Esta talvez seja a maneira mais árdua de gerenciarmos a disposição dos elementos GUI.

Capítulo 4

Componentes Atômicos

São os botões, scrollbars, labels, sliders, check boxes, etc. Eles não podem conter outros elementos.

4.1 JLabel

São rótulos inertes que geralmente informam ou descrevem a funcionalidade de outros componentes GUI, como por exemplo, campos de texto, ícones, etc. As instruções são mostradas por meio de uma linha de texto somente leitura, uma imagem, ou ambos. Aqui salientamos o uso do método **setToolTipText(String)**, o qual fornece dicas de ferramenta a todos os elementos herdados da classe **JComponent**. Dessa forma, quando o usuário posicionar o cursor do mouse sobre algum componente, ficará ciente da função do mesmo. Veremos isso nos exemplos.

O contrutor mais elaborado é **JLabel(String, Icon, int)**. Os argumentos representam o rótulo a ser exibido, um ícone e o alinhamento, respectivamente. Como vemos, também é possível a exibição de ícones em muito dos componentes Swing. Para **JLabels**, basta especificarmos um arquivo com extensao **png**, **gif** ou **jpg** no segundo argumento do construtor do **JLabel**, ou utilizarmos o método **setIcon(Icon)** Lembramos que o arquivo da imagem algumNome.xxx deve encontrar-se no mesmo diretório do programa, ou especifica-se corretamente a estrutura de diretórios até ele.

As constantes **SwingConstants**, que definem o posicionamento de vários componentes GUI e aqui são apropriadas ao terceiro argumento, determinam a locação do ícone em relação ao texto. São elas:

- SwingConstants.NORTH,
- SwingConstants.SOUTH,
- SwingConstants.EAST,
- SwingConstants.WEST,
- SwingConstants.TOP,
- SwingConstants.BOTTOM,
- SwingConstants.CENTER,

- SwingConstants.HORIZONTAL,
- SwingConstants.VERTICAL,
- SwingConstants.LEADING,
- SwingConstants.TRAILING,
- SwingConstants.NORTH_EAST,
- SwingConstants.NORTH_WEST,
- SwingConstants.SOUTH_WEST,
- SwingConstants.SOUTH_EAST,
- SwingConstants.RIGHT,
- SwingConstants.LEFT

Não iremos detalhar a funcionalidade de cada uma, pois os nomes já são o suficientente auto-elucidativos.

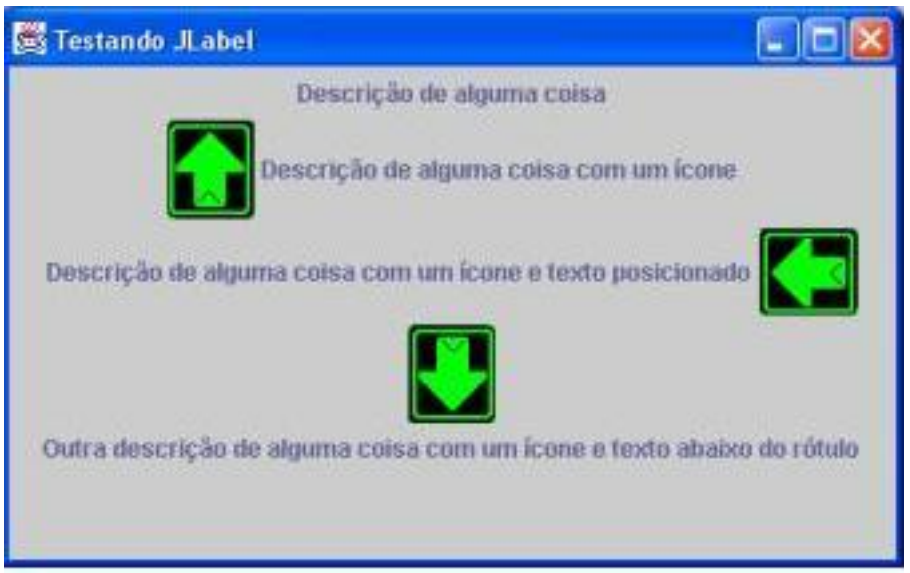


Figura 4.1: Interface do exemplo que usa JLabel

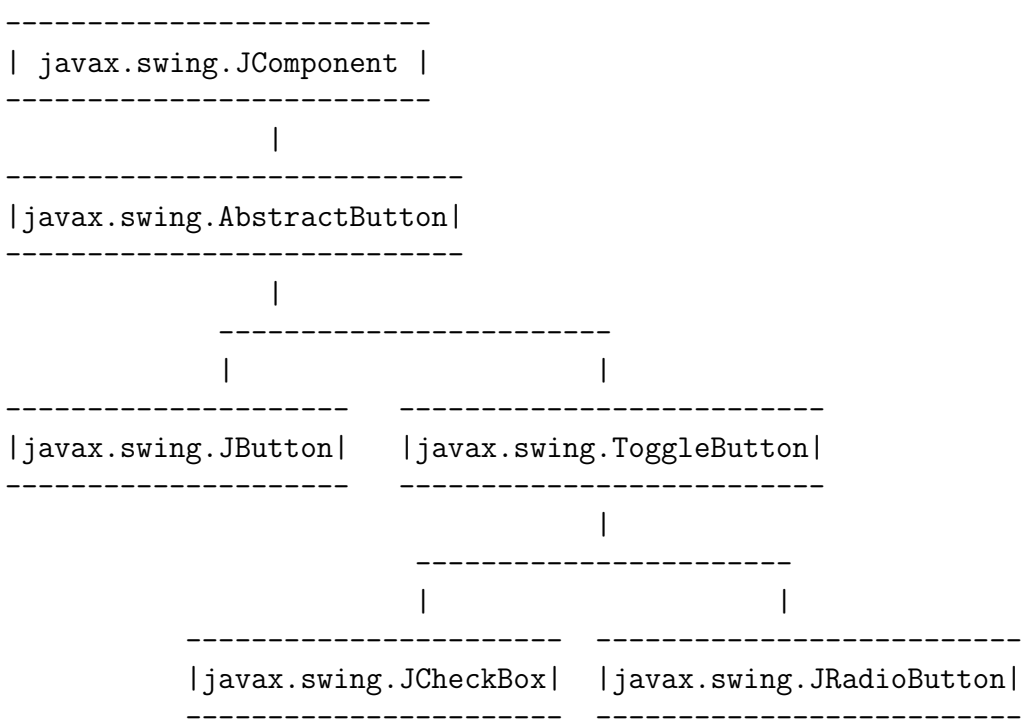
```
1  // Demonstra a classe JLabel.
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class TesteJLabel extends JFrame {
8     private JLabel rótulo1, rótulo2, rótulo3, rótulo4;
9
10    // configura a GUI
11    public TesteJLabel()
12    {
```

```
13        // texto da barra de título
14        super( "Testando JLabel" );
15
16        // obtém painel de conteúdo
17        Container container = getContentPane();
18        // e configura seu layout
19        container.setLayout( new FlowLayout() );
20
21        // construtor JLabel rotulado com o argumento String
22        rótulo1 = new JLabel( "Descrição de alguma coisa" );
23        // o argumento do método é a dica de ferramenta que será exibida
24        rótulo1.setToolTipText( "Dica de algo que isso faz" );
25        // anexa o rótulo1 ao painel de conteúdo
26        rótulo1.setBounds( 50,50, 200, 500);
27        container.add( rótulo1 );
28
29        // construtor JLabel com argumento String, icone e alinhamento
30        Icon seta_90 = new ImageIcon( "figuras/seta_90.gif" );
31        Icon seta_180 = new ImageIcon( "figuras/seta_180.gif" );
32        Icon seta_270 = new ImageIcon( "figuras/seta_270.gif" );
33        rótulo2 = new JLabel( "Descrição de alguma coisa com um ícone",
34           seta_90, SwingConstants.HORIZONTAL );
35        rótulo2.setToolTipText( "Outra dica de algo que isso faz" );
36        container.add( rótulo2 );
37
38        // construtor JLabel sem argumentos
39        rótulo3 = new JLabel();
40        rótulo3.setText( "Descrição de alguma coisa com um ícone" +
41                         " e texto posicionado" );
42        rótulo3.setIcon( seta_180 );
43        // posiciona o texto à esquerda do rótulo
44        rótulo3.setHorizontalTextPosition( SwingConstants.LEFT );
45        // centraliza o texto em relação ao rótulo
46        rótulo3.setVerticalTextPosition( SwingConstants.CENTER );
47        rótulo3.setToolTipText( "Orientação à respeito desse rótulo" );
48        container.add( rótulo3 );
49
50        // construtor JLabel sem argumentos, que posteriormente será
51        //configurado com os métodos "set"
52        rótulo4 = new JLabel();
53        rótulo4.setText( "Outra descrição de alguma coisa com um ícone" +
54                         " e texto abaixo do rótulo" );
55        rótulo4.setIcon( seta_270 );
56        // centraliza o texto em relação ao rótulo
57        rótulo4.setHorizontalTextPosition( SwingConstants.CENTER );
58        // posiciona o texto abaixo do rótulo
59        rótulo4.setVerticalTextPosition( SwingConstants.BOTTOM );
60        rótulo4.setToolTipText( "Orientação à respeito desse rótulo" );
61        container.add( rótulo4 );
62
63        // determina o tamanho da janela do aplicativo
64        setSize( 450, 280 );
65        // determina que o conteúdo anexado à janela seja exibido
66        setVisible( true );
67     }
68
69     // executa a aplicacao
70     public static void main( String args[] )
71     {
```

```
72        TesteJLabel aplicacao = new TesteJLabel();
73        // configura o fechamento da janela
74        aplicacao.setDefaultCloseOperation(
75           JFrame.EXIT_ON_CLOSE );
76     }
77  }
```

4.2 Botões

É um componente que quando clicado dispara uma ação específica. Um programa Java pode utilizar vários tipos de botões, incluindo botões de comando, caixas de marcação, botões de alternância e botões de opção. Para criarmos algum desses tipos de botões, devemos instanciar uma das muitas classes que descendem da classe **AbstractButton**, a qua define muito dos recursos que são comuns aos botões do Swing. Cita-se, por exemplo, a exibição de texto e imagens em um botão, o uso de caracteres mnemônicos, dentre outros. Vejamos a hierarquia de classes, partindo da classe **JComponent**:



4.2.1 JButton

É um dos componentes mais familiares e intuitivos ao usuário. Os botões de comando são criados com a classe **JButton** e seu pressionamento geralmente dispara a ação especificada em seu rótulo, que também suporta a exibição de ícones. Também podemos definir dicas de ferramenta para cada botão, juntamente com mnemônicos, que dão acesso rápido pelo teclado aos comandos definidas nos botões. Para oferecer maior interatividade visual com a GUI, o **JButton** oferece a possibilidade de ícones *rollover*, os quais

mudam de aparência quando o cursor e posicionado sobre eles, dando a entender que o seu pressionamento resulta em uma ação [1]. Deve-se ter a mesma atenção com os arquivos de imagem, de acordo com o que foi mencioando anteriormente. Pressionar um **JButton** gera eventos **ActionEvent** que, juntamente com outros eventos, serão abordados mais a frente.

Figura 4.2: Interface do exemplo que usa JButton

```
1  // Demonstra a classe JButton
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class TesteJButton extends JFrame {
8      private JTextField campo1, campo2;
9      private JLabel nome, sobrenome;
10     private JButton botao1, botao2;
11
12     // configura a GUI
13     public TesteJButton()
14     {
15         super( "Testando JButtons" );
16
17         Container container = getContentPane();
18         container.setLayout( new FlowLayout() );
19
20         nome = new JLabel("Nome");
21         nome.setToolTipText("Escreva seu nome no campo ao lado");
22         container.add( nome );
23
24         // constrói campo de texto com a dimensão do argumento
25         campo1 = new JTextField( 15 );
26         container.add( campo1 );
27
28         sobrenome = new JLabel("Sobrenome");
29         sobrenome.setToolTipText("Escreva seu sobrenome no campo ao lado");
30         container.add( sobrenome );
31
32         // constrói campo de texto com a dimensão do argumento
33         campo2 = new JTextField( 15 );
34         container.add( campo2 );
35
36         // instancia o botão1 com o rótulo "Adicionar"
37         JButton botao1 = new JButton("Adicionar");
38         // configura a tecla "A" como acesso rápido pelo teclado ao comando
39         botao1.setMnemonic(KeyEvent.VK_A);
```

```
40          // configura a dica de ferramenta
41          botao1.setToolTipText("Une o Nome ao Sobrenome");
42          container.add(botao1);
43
44          Icon erase1 = new ImageIcon("figuras/erase1.png");
45          Icon erase2 = new ImageIcon("figuras/erase2.png");
46
47          // instancia o botão2 com o rótulo e um ícone
48          JButton botao2 = new JButton("Apagar", erase1);
49          // configura o botao2 com a capacidade de intuir o pressionamento
50          botao2.setRolloverIcon(erase2);
51          // configura a tecla "P" como acesso rápido pelo teclado ao comando
52          botao2.setMnemonic(KeyEvent.VK_P);
53          // configura a dica de ferramenta
54          botao2.setToolTipText("Limpa os campos Nome e Sobrenome");
55          container.add(botao2);
56
57          // registra tratador de eventos
58          botao1.addActionListener(
59          // cria o objeto que trata o evento de acordo com a definição
60          // de actionPerformed
61          new ActionListener(){
62              public void actionPerformed ( ActionEvent evento ){
63                  JOptionPane.showMessageDialog(null,"Seu nome completo é: " +
64                  campo1.getText() + campo2.getText()); // retorna os textos
65                  // dos campos
66
67              }
68          }
69          );
70
71          // registra tratador de eventos
72          botao2.addActionListener(
73          // cria o objeto que trata o evento de acordo com a definição
74          // de actionPerformed
75          new ActionListener(){
76              public void actionPerformed ( ActionEvent evento ){
77                  campo1.setText(" "); // configura os campos com String vazio
78                  campo2.setText(" ");
79                  repaint();
80              }
81          }
82          );
83
84          setSize( 525, 125 );
85          setVisible( true );
86      }
87
88      // executa a aplicacao
89      public static void main( String args[] )
90      {
91          TesteJButton aplicacao = new TesteJButton();
92
93          aplicacao.setDefaultCloseOperation(
94          JFrame.EXIT_ON_CLOSE );
95      }
96  }
```

4.2.2 JCheckBox

A classe **JCheckBox** dá suporte à criação de botões com caixa de marcação, sendo que qualquer número de itens pode se selecionado. Quando um item é selecioando, um **ItemEvent** é gerado. O mesmo pode ser tratado por um objeto que implemente a interface **ItemListener**. A classe que fornece as funcionalidades para este objeto deve definir o método **itemStateChanged**, mas isso será visto mais tarde no próximo capítulo.

Encaminhe-se para o nosso exemplo que discute as JCheckBox e você verá que ele verifica qual das caixas foi selecionada para, posteriormente, incrementar a variável soma com o valor respectivo de cada caixa de marcação. Usamos o método **isSelected()**, que retorna verdadeiro caso o item esteja selecionado, para tal finalidade.

Figura 4.3: Interface do exemplo que usa JCheckBox

```
1  // Testa botões de caixas de marcação
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class TesteJCheckBox extends JFrame {
8      private JTextField campo;
9      private JCheckBox cinco, sete, treze;
10
11
12     // configura a GUI
13     public TesteJCheckBox()
14     {
15         // texto da barra de título
16         super( "Teste de JCheckBox" );
17
18         // obtém painél de conteúdo
19         Container container = getContentPane();
20         // e configura o leiaute
21         container.setLayout( new FlowLayout() );
22
23         // configura a JTextField e sua fonte
24         campo = new JTextField( "Este campo irá exibir a soma dos " +
25         "valores marcados", 30 );
26         campo.setFont( new Font( "Serif", Font.PLAIN, 14 ) );
27         campo.setEditable(false);
28         container.add( campo ); // anexa ao painél de conteúdo
29
30         // cria as caixas de marcação e define os caracteres mnemônicos
31         cinco = new JCheckBox( "Cinco" );
32         cinco.setMnemonic(KeyEvent.VK_C);
```

```
33          container.add( cinco ); // anexa ao painél de conteúdo
34
35          sete = new JCheckBox( "Sete" );
36          sete.setMnemonic(KeyEvent.VK_S);
37          container.add( sete );  // anexa ao painél de conteúdo
38
39          treze = new JCheckBox( "Treze" );
40          treze.setMnemonic(KeyEvent.VK_T);
41          container.add( treze ); // anexa ao painél de conteúdo
42
43          // registra os ouvintes para as caixas de marcação
44          TratadorCheckBox trat = new TratadorCheckBox();
45          cinco.addItemListener( trat );
46          sete.addItemListener( trat );
47          treze.addItemListener( trat );
48
49          // dimensiona a janela e a exibe
50          setSize( 350, 100 );
51          setVisible( true );
52      }
53
54      // executa a aplicação
55      public static void main( String args[] )
56      {
57          TesteJCheckBox aplicacao = new TesteJCheckBox();
58
59          aplicacao.setDefaultCloseOperation(
60          JFrame.EXIT_ON_CLOSE );
61      }
62
63      // classe interna privativa que trata de eventos ItemListener
64      private class TratadorCheckBox implements ItemListener {
65
66          // responde aos eventos das caixas de marcação
67          public void itemStateChanged( ItemEvent evento )
68          {
69              int soma = 0;
70              // processa evento da caixa de marcação "Cinco"
71              if ( cinco.isSelected() )
72                  soma = soma + 5;
73
74              // processa evento da caixa de marcação "Sete"
75              if ( sete.isSelected() )
76                  soma = soma + 7;
77
78              // processa evento da caixa de marcação "Treze"
79              if ( treze.isSelected() )
80                  soma = soma + 13;
81              // configura texto da JTextField
82              campo.setText("A soma acumulada é: " + soma);
83          }
84      }
85  }
```

4.2.3 JRadioButton

Os botões de opção, que são definidos na classe **JRadioButton**, assemelham-se às caixas de marcação no que concerne aos seus estados (selecionado ou

não selecionado). Entretando, costumeiramente são usados em grupo no qual apenas um botão de opção pode ser marcado, forçando os demais botões ao estado não-selecionado.

Nosso exemplo, que realiza uma função muito elementar, mudar a cor de um JTextField baseado na marcação de um grupo de JRadioButton, requer que somente uma opção seja selecionada dentre as várias oferecidas. Para criarmos o relacionamento lógico que acarreta essa funcionalidade usamos um objeto **ButtonGroup**, do pacote javax.swing, que em si não é um componente GUI. Ele não é exibido na interface gráfica com o usuário, porém sua funcionalidade é destacada no momento em que torna as opções mutuamente exclusivas.

Os métodos aqui utilizados pouco diferem dos da classe anterior, sendo que a única novidade é o método **getSource( )**, que retorna a fonte geradora do evento, ou seja, um dos botões rotulados com o nome das cores Amarelo, Azul ou Vermelho.

Figura 4.4: Interface do exemplo que usa JRadioButton

```
1  // Testa botões de caixas de marcação
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class TesteJRadioButton extends JFrame {
8      private JTextField campo;
9      private JRadioButton amarelo, vermelho, azul;
10     private ButtonGroup onlyOne;
11
12     // configura a GUI
13     public TesteJRadioButton()
14     {
15         // texto da barra de título
16         super( "Teste de JRadioButton" );
17
18         // obtém painél de conteúdo
19         Container container = getContentPane();
20         // e configura o leiaute
21         container.setLayout( new FlowLayout() );
22
23         // configura a JTextField e sua fonte
24         campo = new JTextField( "Este campo irá mudar de cor", 25 );
25         campo.setFont( new Font( "Serif", Font.PLAIN, 14 ) );
26         container.add( campo );
27
28         // cria as caixas de marcação
```

```
29          amarelo = new JRadioButton( "Amarelo" );
30          container.add( amarelo );
31
32          vermelho = new JRadioButton( "Vermelho" );
33          container.add( vermelho );
34
35          azul = new JRadioButton( "Azul" );
36          container.add( azul );
37
38          // cria um botão "virtual" que permite somente
39          // a marcação de uma única caixa
40          onlyOne = new ButtonGroup();
41          onlyOne.add(amarelo);
42          onlyOne.add(vermelho);
43          onlyOne.add(azul);
44
45          // registra os ouvintes para as caixas de marcação
46          TratadorRadioButton trat = new TratadorRadioButton();
47          amarelo.addItemListener( trat );
48          vermelho.addItemListener( trat );
49          azul.addItemListener( trat );
50
51          setSize( 285, 100 );
52          setVisible( true );
53      }
54
55      // executa a aplicação
56      public static void main( String args[] )
57      {
58          TesteJRadioButton aplicação = new TesteJRadioButton();
59
60          aplicação.setDefaultCloseOperation(
61          JFrame.EXIT_ON_CLOSE );
62      }
63
64      // classe interna privativa que trata de eventos ItemListener
65      private class TratadorRadioButton implements ItemListener {
66          private Color cor ;
67
68          // responde aos eventos das caixas de marcação
69          public void itemStateChanged( ItemEvent evento )
70          {
71              // processa evento da caixa de marcação "Vermelho"
72              if ( evento.getSource() == vermelho )
73                  cor = Color.red;
74
75              // processa evento da caixa de marcação "Amarelo"
76              if ( evento.getSource() == amarelo )
77                  cor = Color.yellow;
78
79              // processa evento da caixa de marcação "Azul"
80              if ( evento.getSource() == azul )
81                  cor = Color.blue;
82
83              campo.setBackground(cor);
84          }
85      }
86  }
```

4.3 JTextField

Compreende a área de uma única linha que suporta a inserção ou exibição de texto. Podemos definir se o texto pode ser manipulado com o método **setEditable(boolean)**, utilizando no argumento o valor **true**.

Quando o usuário digita os dados em uma JTexField e pressiona Enter, ocorre um evento de ação. Esse evento é processado pelo ouvinte de evento registrado que pode usar os dados que estão no JTexField no momento em que o evento ocorre[1]. Nosso exemplo implementa diversos campos de texto com um evento associado a cada um deles.

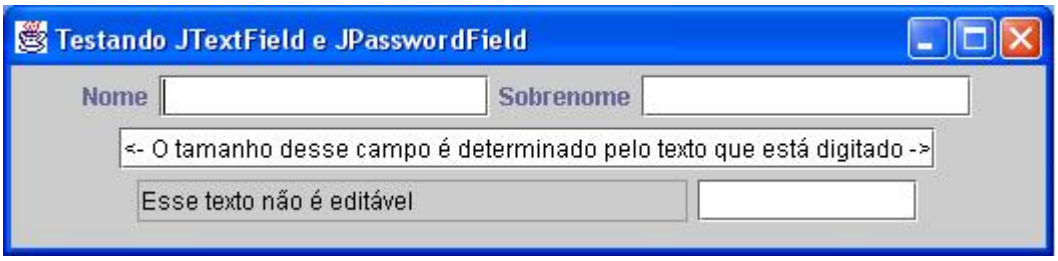


Figura 4.5: Interface do exemplo que usa JTextField

```
1  // Demonstra a classe JTextField.
2  import java.awt.*;
3  import java.awt.event.*;
4  import javax.swing.*;
5
6  public class TesteJTextField extends JFrame {
7      private JTextField campo1, campo2, campo3, campo4;
8      private JPasswordField campoDaSenha;
9      private JLabel nome, sobrenome;
10
11     // configura a GUI
12     public TesteJTextField()
13     {
14         super( "Testando JTextField e JPasswordField" );
15
16         Container container = getContentPane();
17         container.setLayout( new FlowLayout() );
18
19         nome = new JLabel("Nome");
20         nome.setToolTipText("Escreva seu nome no campo ao lado");
21         container.add( nome );
22
23         // constrói campo de texto com a dimensão do argumento
24         campo1 = new JTextField( 15 );
25         container.add( campo1 );
26
27         sobrenome = new JLabel("Sobrenome");
28         sobrenome.setToolTipText("Escreva seu sobrenome no campo ao lado");
29         container.add( sobrenome );
30
```

[1]Julgamos didático a inserção gradual do assunto que veremos mais detalhadamente no próximo cápitulo.

```
31          // constrói campo de texto com a dimensão do argumento
32          campo4 = new JTextField( 15 );
33          container.add( campo4 );
34
35          // constrói campo de texto dimensionado pelo String do argumento
36          campo2 = new JTextField( "<- O tamanho desse campo é determinado" +
37          " pelo texto que está digitado ->" );
38          container.add( campo2 );
39
40          // constrói campo de texto não editável com o String do argumento
41          campo3 = new JTextField( "Esse texto não é editável", 25 );
42          campo3.setEditable( false );
43          container.add( campo3 );
44
45          // constrói campo de texto usado para digitação de senhas com
46          // a dimensão do argumento
47          campoDaSenha = new JPasswordField( 10 );
48          container.add( campoDaSenha );
49
50          // registra tratadores de eventos
51          TratadorTextField trat = new TratadorTextField();
52          campo1.addActionListener( trat );
53          campo2.addActionListener( trat );
54          campo3.addActionListener( trat );
55          campo4.addActionListener( trat );
56          campoDaSenha.addActionListener( trat );
57
58          setSize( 525, 125 );
59          setVisible( true );
60      }
61
62      // executa a aplicacao
63      public static void main( String args[] )
64      {
65          TesteJTextField applicacao = new TesteJTextField();
66
67          applicacao.setDefaultCloseOperation(
68          JFrame.EXIT_ON_CLOSE );
69      }
70
71      // classe privativa interna para tratamento de eventos
72      private class TratadorTextField implements ActionListener {
73
74          // identifica o campo de texto responsável pelo evento e,
75          // em cada caso, o trata
76          public void actionPerformed( ActionEvent evento )
77          {
78              String output = "";
79
80              // usuário pressionou Enter no JTextField campo1
81              if ( evento.getSource() == campo1 )
82                  output = "no campo1: " + evento.getActionCommand();
83
84              // usuário pressionou Enter no JTextField campo2
85              else if ( evento.getSource() == campo2 )
86                  output = "no campo2: " + evento.getActionCommand();
87
88              // usuário pressionou Enter no JTextField campo3
89              else if ( evento.getSource() == campo3 )
```

```
90                  output = "no campo3: " + evento.getActionCommand();
91
92              else if ( evento.getSource() == campo4 )
93                  output = "no campo4: " + evento.getActionCommand();
94
95              // usuário pressionou Enter no JPasswordField
96              else if ( evento.getSource() == campoDaSenha ) {
97
98                  if((new String(campoDaSenha.getPassword())).
99                  equals( new String("Swing"))){
100                     output = "a senha correta, Parabéns!";
101                 }
102                 else output = "uma Senha Inválida!";
103             }
104             JOptionPane.showMessageDialog(null, "Você digitou " + output);
105         }
106     }
107 }
```

4.4 JPasswordField

É uma subclasse de JTextField e acrescenta vários métodos específicos para o processamento de senhas. Sua aparência e comportamento quase nada diferem de uma JTextField, a não ser quando o texto é digitado, pois o mesmo fica ocultado pelos asteriscos. Tal procedimento se justifica para ocultar os caracteres inseridos, dado que esse campo contém uma senha. Sua aparência pode ser vista na região inferior da interface do exemplo que demonstra JTextField.

4.5 JTextArea

É uma área dimensionável que permite que múltiplas linhas de texto sejam editadas com a mesma fonte. Esta classe é herdada de **JTextComponent**, que define métodos comuns para JTextField, JTextArea e outros elementos GUI baseados em texto.

As JTextAreas não têm eventos de ação como os objetos da classe **JTextField**, cujo o pressionamento de *Enter* gera um evento. Então, utiliza-se um outro componente GUI (geralmente um botão) para gerar um evento externo que sinaliza quando o texto de uma JTextArea deve ser processado.

Se desejarmos reconfigurar a fonte de uma JTextArea, devemos criar um novo objeto fonte, como demonstrado nesse exemplo:

```
1 setFont(new Font("Serif", Font.ITALIC, 16));
```

Podemos configurar um texto com **setText(String)** ou acrescentar texto com o método **append (String)**. Para evitar que um longo texto digitado fique incluso em somente uma linha, usamos o método **setLineWrap(boolean)**, que define a quebra da linha quando o texto alcançar a borda da JTextArea. Porém, as palavras podem ficar "quebradas", com caracteres em uma linha e outros na próxima, sem nenhum compromisso com as normas gramaticais. Uma maneira de sanar paliativamente esse problema é invocar o método **setWrapStyleWord(boolean)**, o qual determina que

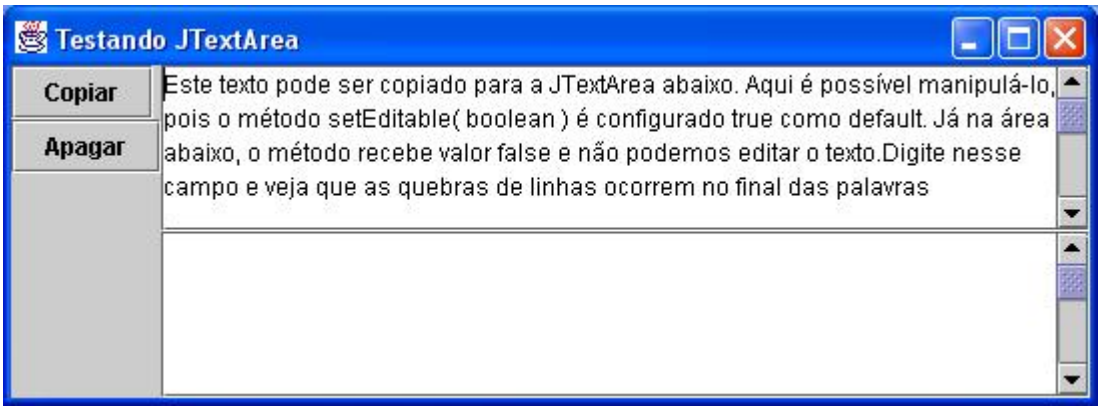


Figura 4.6: Interface do exemplo que usa JTextArea

a mudança de linha seja definida pelas palavras. Em nosso exemplo, usamos esses dois métodos passando no argumento de ambos um valor **true**.

Também vale-se de uma JTextArea como argumento para um diálogo de mensagem, caso seja necessário exibir longas saídas baseadas em texto. Assim, a caixa de mensagem que exibe a JTextArea determina a largura e a altura da area de texto, com base no *String* que ela contém. No construtor **JTextArea (int, int),** podemos definir o tamanho da área de texto passando como argumento, respectivamente, o número de linhas e colunas.

```
1  // Demostra funcionalidades das JTextArea
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class TesteJTextArea extends JFrame {
8      private JTextArea areaTexto1, areaTexto2;
9      private JButton copiar, apagar;
10     private String selecionado;
11
12     // configura a GUI
13     public TesteJTextArea()
14     {
15         // texto da barra de título
16         super( "Testando JTextArea" );
17
18         // cria uma caixa vertical para anexar os botões e os textos
19         Box caixaTextos = Box.createVerticalBox();
20         Box caixaBotoes = Box.createVerticalBox();
21
22         String textoDefault = "Este texto pode ser copiado para a JTextArea " +
23         "abaixo. Aqui é possível manipulá-lo, pois o método " +
24         "setEditable( boolean ) é configurado true como default." +
25         " Já na área abaixo, o método recebe valor false e não "+
26         "podemos editar o texto.Digite nesse campo e veja que as "+
27         "quebras de linhas ocorrem no final das palavras";
28
29         // configura a areaTexto1 com 13 linhas e 15 colunas visíveis
30         areaTexto1 = new JTextArea( textoDefault, 13, 15 );
31         // configura mudança automática de linha
```

```
32          areaTexto1.setLineWrap(true);
33          // determina que as mudança de linha seja definida pelas palavras
34          areaTexto1.setWrapStyleWord(true);
35          //acrescenta barras de rolagem à área de texto
36          caixaTextos.add( new JScrollPane(areaTexto1,
37          JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
38          JScrollPane.HORIZONTAL_SCROLLBAR_NEVER ) );
39
40          // configura o botão "Copiar"
41          copiar = new JButton( "Copiar" );
42          // registra o botão "Copiar" como tratador de eventos
43          copiar.addActionListener(
44          // classe interna anônima que trata os eventos do botão "Copiar"
45          new ActionListener() {
46
47              // exibe o texto selecioando da "areaTexto1" na "areaTexto2"
48              public void actionPerformed( ActionEvent evento )
49              {
50                  selecionado = areaTexto1.getSelectedText();
51
52                  // testa se algo foi selecionado
53                  if(selecionado != null){
54                      areaTexto2.setText(areaTexto1.getSelectedText());
55                                          selecionado = null;
56                  }
57                  else{
58                          JOptionPane.showMessageDialog(null,
59                          "Selecione algum texto!", "Aviso",
60                          JOptionPane.INFORMATION_MESSAGE);
61                  }
62              }
63          }
64
65          );
66
67          // anexa o botão copiar a caixa
68          caixaBotoes.add( copiar );
69
70          // configura o botão "Apagar"
71          apagar = new JButton( "Apagar" );
72          // registra o botão "Apagar" como tratador de eventos
73          apagar.addActionListener(
74
75          // classe interna anônima que trata os eventos do botão "Apagar"
76          new ActionListener() {
77
78              // apaga o texto da "areaTexto2"
79              public void actionPerformed( ActionEvent evento )
80              {
81                  areaTexto2.setText( " " );
82              }
83          }
84          );
85
86          // anexa o botão apagar a caixa
87          caixaBotoes.add( apagar );
88
89          // configura a areaTexto2
90          areaTexto2 = new JTextArea( 13, 15 );
```

```
91          // configura mudança automática de linha
92          areaTexto2.setLineWrap(true);
93          // restringe a manilupação do texto da areaTexto2
94          areaTexto2.setEditable( false );
95          // determina que as mudança de linha seja definida pelas palavras
96          areaTexto2.setWrapStyleWord(true);
97          caixaTextos.add( new JScrollPane( areaTexto2 ) );
98
99          // obtém painel de conteúdo
100         Container container = getContentPane();
101         // anexa e posiciona as caixas de texto no centro do conteiner
102         container.add( caixaTextos,BorderLayout.CENTER );
103         // anexa posiciona a caixa de botoes no lado oeste do conteiner
104         container.add( caixaBotoes,BorderLayout.WEST );
105         setSize( 547, 200 );
106         setVisible( true );
107     }
108
109     // executa a aplicacao
110     public static void main( String args[] )
111     {
112         TesteJTextArea aplicacao = new TesteJTextArea();
113
114         aplicacao.setDefaultCloseOperation(
115         JFrame.EXIT_ON_CLOSE );
116     }
117 }
```

4.6 JScrollPane

Objetos dessa classe fornecem a capacidade de rolagem a componentes da classe **JComponent**, quando estes necessitam de mais espaço para exibir dados.

JScrollpane (Component, int, int) é o construtor mais elaborado e recebe um componente (JTextArea por exemplo) como primeiro argumento, definindo qual será o cliente do JScrollPane, ou seja, para que membro será fornecido as barras de rolagem. Os dois próximos argumentos definem o comportamento da barra vertical e da horizontal, respectivamente. Para isso, podemos fazer uso das constantes definidas na interface **ScrollPaneConstants** que é implementada por **JScrollPane**. Vejamos elas [1]:

JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED

JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED Indicam que as barras de rolagem devem aparecer somente quando necessário.

JScrollPane.VERTICAL_SCROLLBAR_ALWAYS

JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS Indicam que as barras de rolagem devem aparecer sempre.

JScrollPane.VERTICAL_SCROLLBAR_NEVER

JScrollPane.HORIZONTAL_SCROLLBAR_NEVER Indicam que as barras de rolagem nunca devem aparecer.

É possível configurar o comportamento do JScrollPane para um objeto com os métodos **setVerticalScrollBarPolicy(int)** e **setHorizontalScrollBarPolicy(int)**, valendo-se das mesma contantes como argumentos.

Como você já deve ter visto, em muitos exemplos já fizemos uso dessa classe, o que nos exime de implementar um exemplo específico para um componente tão conhecido e sem predicativos merecedores de atenção especial.

4.7 JSlider

É um marcador que desliza entre um intervalo de valores inteiros, podendo selecionar qualquer valor de marca de medida em que o marcador repouse. Uma das inúmeras utilidades desse controle deslizante é restringir os valores de entrada em um aplicativo, evitando que o usuário informe valores que causem erros.

Os JSlider comportam a exibição de marcas de medidas principais, secundárias e rótulos de medida. A aderência às marcas (*snap to ticks)* possibilita ao marcador aderir à marca mais próxima, quando este situar-se entre dois valores.

Este componente responde às interações feitas pelo mouse e pelo teclado (setas, PgDn, PgUp, Home e End). Sua orientação pode ser horizontal, na qual o valor mínimo está situado na extrema esquerda, ou vertical, na qual o valor mínimo está situado na extremidade inferior. As posições de valor mínimo e máximo podem ser invertidas, valendo-se do método **setInvert(boolean)**, com um argumento **true**.

Figura 4.7: Interface do exemplo que usa JSlider

```
1  // Demonstra funcionalidades do JSlider
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.text.DecimalFormat;
6  import javax.swing.*;
7  import javax.swing.event.*;
8
9  public class TesteJSlider extends JFrame {
10     private JSlider slider;
11     private JTextField campo = new JTextField("");
```

```
12      DecimalFormat valor = new DecimalFormat("000");
13
14      // configura a GUI
15      public TesteJSlider()
16      {
17          super( "Testando o JSlider" );
18
19          // configura o JSlider para "trabalhar" com valores entre 0 e 100
20          // o valor inicial é 25
21          slider = new JSlider( SwingConstants.HORIZONTAL, 0, 100, 25 );
22
23          // o intervalo entre as marcas principais é 10
24          slider.setMajorTickSpacing( 10 );
25
26          // o intervalo entre as marcas secundárias é 5
27          slider.setMinorTickSpacing(5);
28
29          // exibe as marcas de medidas
30          slider.setPaintTicks( true );
31
32          // exibe o valor das medidas
33          slider.setPaintLabels( true );
34
35          // configura a fonte a ser exibida no campo
36          campo.setFont(new Font("Monospaced",Font.BOLD,35));
37
38          // dimensiona o campo
39          campo.setSize(100,50);
40
41          // obtém o valor inicial do marcador do JSlider e o exibe num campo
42          campo.setText(valor.format( slider.getValue( ) ));
43
44          // registra o ouvinte de eventos do JSlider
45          slider.addChangeListener(
46
47          // classe interna anônima que trata os eventos do JSlider
48          new ChangeListener() {
49
50              // trata a mudança de valor decorrente do deslize do marcador
51              public void stateChanged( ChangeEvent e )
52              {
53                  campo.setText(valor.format( slider.getValue( ) ));
54              }
55
56          }
57
58          );
59
60          // obtém painel de conteúdo
61          Container container = getContentPane();
62
63          // anexa os componentes ao container
64          container.add( slider, BorderLayout.SOUTH );
65          container.add( campo, BorderLayout.NORTH );
66
67          setSize( 250, 200 );
68          setVisible( true );
69      }
70
```

```
71      // executa a aplicação
72      public static void main( String args[] )
73      {
74          TesteJSlider aplicacao = new TesteJSlider();
75
76          aplicacao.setDefaultCloseOperation(
77          JFrame.EXIT_ON_CLOSE );
78      }
79  }
```

4.8 JComboBox

Assemelha-se a um botão, porém, quando clicado, abre uma lista de possíveis valores ou opções. Mais precisamente é uma caixa de combinação que permite ao usuário fazer uma seleção a partir de uma lista de itens. Atende-se para que a lista da caixa de combinação, quando aberta, não ultrapasse os limites da janela da aplicação.

Também é possível digitar nas linhas de uma caixa de combinação. Elas são implementadas com a classe **JComboBox**, herdada de **JComponent**. Tais caixas de combinação geram ItemEvents, assim como as JCheckBoxes.

Figura 4.8: Interface do exemplo que usa JComboBox

```
1   // Demonstra o uso de uma JComboBox para
2   // selecionar uma figura
3
4   import java.awt.*;
5   import java.awt.event.*;
6   import javax.swing.*;
7
8   public class TesteJComboBox extends JFrame {
9       private JComboBox comboBox, comboBoxEdit;
10      private JLabel rotulo;
11      private JPanel esquerdo, direito;
12      private String nomes[] = {"Wanderson","Leonardo",
13      "Gabriel","Daniel"};
14
15      private String nomesDasFiguras[] = { "figuras/seta_360.gif",
16      "figuras/seta_90.gif", "figuras/seta_180.gif", "figuras/seta_270.gif" };
17
18      private Icon figuras[] = {new ImageIcon(nomesDasFiguras[ 0 ]),
19                                new ImageIcon(nomesDasFiguras[ 1 ]),
20                                new ImageIcon(nomesDasFiguras[ 2 ]),
21                                new ImageIcon(nomesDasFiguras[ 3 ])};
22
23      // configura a GUI
```

```
24      public TesteJComboBox()
25      {
26          super( "Testando uma JComboBox" );
27
28          // obtém painél de conteúdo
29          Container container = getContentPane();
30
31          // e configura seu leiaute
32          container.setLayout( new GridLayout(1,2) );
33
34          // cria a JComboBox
35          comboBox = new JComboBox(nomesDasFiguras );
36
37          // configura a JComboBox para, quando clicada,
38          // exibir 3 linhas
39          comboBox.setMaximumRowCount( 3 );
40
41          // configura a JComboBox para exibir a figura de
42          // indice 2 do array nomeDasFiguras
43          comboBox.setSelectedIndex( 2 );
44
45          comboBoxEdit = new JComboBox( nomes );
46          comboBoxEdit.setEditable(true);
47
48          // registra tratador de eventos
49          comboBox.addItemListener(
50
51          // classe interna anônima para tratar eventos
52          // da JComboBox
53          new ItemListener() {
54
55              // trata os eventos da JComboBox
56              public void itemStateChanged( ItemEvent evento )
57              {
58                  // determina se a caixa de marcação está selecionada
59                  if ( evento.getStateChange() == ItemEvent.SELECTED )
60                      rotulo.setIcon( figuras[comboBox.getSelectedIndex() ] );
61              }
62          }
63          );
64
65          comboBoxEdit.addItemListener(
66
67          // classe interna anônima para tratar eventos da JComboBox
68          new ItemListener() {
69
70              // trata os eventos da JComboBox
71              public void itemStateChanged( ItemEvent evento )
72              {
73                  // determina se a caixa de marcação está selecionada
74                  if ( evento.getStateChange() == ItemEvent.SELECTED )
75                  {
76                      JOptionPane.showMessageDialog(null,
77                      "Você selecionou : " +(comboBoxEdit.getSelectedItem()) );
78                  }
79                  }
80          }
81          );
82
```

```
83          // configura o JLabel para mostrar as figuras
84          rotulo = new JLabel( figuras[ 0 ] );
85
86          // anexando componentes ao painel esquerdo
87          esquerdo = new JPanel();
88          esquerdo.setLayout( new BorderLayout() );
89          esquerdo.add( comboBox, BorderLayout.NORTH );
90          esquerdo.add( rotulo , BorderLayout.CENTER);
91          container.add(esquerdo);
92
93                  // anexando componentes ao painel direito
94          direito = new JPanel();
95          direito.setLayout( new BorderLayout() );
96          direito.add( comboBoxEdit , BorderLayout.NORTH);
97          container.add(direito);
98
99          setSize( 350, 150 );
100         setVisible( true );
101     }
102
103     // executa a  aplicacao
104     public static void main( String args[] )
105     {
106         TesteJComboBox aplicacao = new TesteJComboBox();
107
108         aplicacao.setDefaultCloseOperation(
109         JFrame.EXIT_ON_CLOSE );
110     }
111 }
```

4.9 JList

Exibe em uma coluna uma série de itens que podem ser selecionados. A classe **JList** suporta listas em que o usuário pode selecionar apenas um item e listas de seleção múltipla, permitindo que um número qualquer de itens seja selecionado. Fazemos uso do método **setSelectionMode(ListSelectionMode**) para definir isso.

A classe **ListSelectionMode**, do pacote **javax.swing**, fornece as seguintes constantes que podem ser usadas como argumento do método precedente:

ListSelectionMode.SINGLE_SELECTION configura lista de seleção única;

ListSelectionMode.SINGLE_INTERVAL_SELECTION permite seleção de itens contíguos, ou seja, um logo abaixo do outro;

ListSelectionMode.MULTIPLE_INTERVAL_SELECTION é uma lista de seleção múltipla que não restringe os itens que podem ser selecionados.

Os itens que serão exibidos por uma lista podem ser passados como argumento no momento da inicialização. A classe **JList** fornece construtores que recebem **Vectors** e **arrays** como argumentos. Se você inicializar uma lista com um array ou vetor, o construtor implicitamente cria uma lista modelo *default*. Ela é imutável, ou seja, você não poderá adicionar, remover ou

sobrescrever os itens. Para criar uma lista onde os itens possam ser modificados, devemos configurar o modelo de lista chamando o método **setModel(ListModel)**. Para o mesmo propósito, também é possível instanciar um objeto de uma classe de lista mutável, como **DefaultListMode**, adicionar elementos a ele, para depois passa-lo como argumento do construtor de JList. Vejamos um exemplo:

```
1  modeloLista = new DefaultListModel();
2
3  modeloLista.addElement("Um");
4  modeloLista.addElement("Dois");
5  modeloLista.addElement("Três");
6
7  listaNumeros = new JList(modeloLista);
```

Figura 4.9: Interface do exemplo que usa JList

Atende-se para uma deficiência das JList, pois elas não fornecem barras de rolagem caso haja mais itens na lista que o número de linhas visíveis. Contornamos isso usando um objeto JScrollPane.

Muitas operações de uma lista são gerenciadas por outros objetos. Por exemplo, os itens são gerenciados por um objeto *list model*, a seleção por um *list selection model*. Na maioria das vezes, você não precisa se preocupar com os modelos porque JList os cria se necessário e você interage com eles implicitamente com os métodos convenientes de **JList**.

Em nosso aplicativo de exemplo, fazemos o uso das duas listas, sendo que a de seleção única configura a cor de uma região da janela valendo-se do método **getSelectedIndex()**, que devolve um inteiro referente à posição do item selecionado no array. Já a seleção múltipla permite que seus itens selecionados sejam exibidos numa outra lista abaixo dela. Utilizamos os métodos **setListData(Object**[ ]) e **getSelectedValues( )** para obter essa funcionalidade. Consulte os outros métodos dessa classe para saber que outros tipos de informações podem ser retornadas, tal como valor máximo e mínimo dos índices de uma seleção de itens, dentre outros.

Definimos a largura de uma lista com o método **setFixedCellWidth(int)** e a altura de cada item com **setFixedCellHeight(int)**, que recebem no argumento um inteiro que representa o número de pixels.

Salientamos que uma lista de seleção múltipla não tem um evento específico associado à seleção de múltiplos itens. Assim como para objetos JTextArea, devemos criar outro componente (um botão por exemplo) para gerar um evento externo e processar os itens selecionados.

```
1  // Demonstra funcionalidades da JList
2
3  import java.awt.*;
4  import javax.swing.*;
5  import javax.swing.event.*;
6
7  public class TesteJList extends JFrame {
8      private JList listaDeCores, listaSelecionavel, listaDeTexto;
9      private Container container;
10     private JPanel direita, esquerda;
11
12     private String nomeDasCores[] = { "Preto", "Azul",
13     "Azul Claro", "Cinza Escuro", "Cinza", "Verde",
14     "Cinza Claro", "Magenta", "Laranja", "Rosa",
15     "Vermelho", "Branco", "Amarelo" };
16
17     private Color cores[] = { Color.black, Color.blue,
18     Color.cyan, Color.darkGray, Color.gray, Color.green,
19     Color.lightGray, Color.magenta, Color.orange, Color.pink,
20     Color.red, Color.white, Color.yellow };
21
22     // configura a GUI
23     public TesteJList()
24     {
25         super( "Testando JList" );
26
27         // obtém painel de conteúdo
28         container = getContentPane();
29
30         // e configura o layout
31         container.setLayout( new GridLayout(1,2) );
32
33         esquerda = new JPanel();
34         esquerda.setLayout(new BorderLayout());
35
36         direita = new JPanel();
37         direita.setLayout(new BorderLayout());
38
39         // cria uma lista com itens do array "nomeDasCores"
40         listaSelecionavel = new JList( nomeDasCores );
41
42         // determina o número de itens visíveis na lista
43         listaSelecionavel.setVisibleRowCount( 5 );
44
45         // especifica o modo de seleção na lista
46         listaSelecionavel.setSelectionMode(
47         ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
48
49         // cria uma lista
50         listaDeTexto = new JList( );
```

```
51
52          // determina o número de itens visíveis na lista
53          listaDeTexto.setVisibleRowCount( 5 );
54
55          // configura a largura da lista "listaDeTexto"
56          listaDeTexto.setFixedCellWidth(10);
57
58          // configura a altura da lista "listaDeTexto"
59          listaDeTexto.setFixedCellHeight(10);
60
61          // cria uma lista com itens do array "nomeDasCores"
62          listaDeCores = new JList( nomeDasCores );
63
64          // especifica o modo de seleção na lista
65          listaDeCores.setSelectionMode(
66          ListSelectionModel.SINGLE_SELECTION );
67
68                  // determina o número de itens visíveis na lista
69          listaDeCores.setVisibleRowCount( 5 );
70
71          // adiciona  aos painéis as JList, juntamente com
72          // seus JScrollPane
73          esquerda.add( new JScrollPane(listaDeCores), BorderLayout.NORTH );
74
75          direita.add( new JScrollPane(listaDeTexto), BorderLayout.CENTER );
76          direita.add( new JScrollPane(listaSelecionavel), BorderLayout.NORTH );
77
78          // anexa os painéis ao container
79          container.add(esquerda);
80          container.add(direita);
81
82          // configura tratador de eventos da "listaSelecionavel"
83          listaSelecionavel.addListSelectionListener(
84
85          // classe anônima interna para eventos de
86          // seleção de lista
87          new ListSelectionListener() {
88
89              // trata eventos de seleção de lista
90              public void valueChanged( ListSelectionEvent evento )
91              {
92                  // configura os dados da "listaDeTexto" com os itens
93                  // selecionados da "listaSelecionavel"
94                  listaDeTexto.setListData(
95                  listaSelecionavel.getSelectedValues() );
96              }
97          }
98          );
99
100         // configura tratador de eventos da "listaDeCores"
101         listaDeCores.addListSelectionListener(
102
103         // classe anônima interna para eventos de seleção de lista
104         new ListSelectionListener() {
105
106             // trata eventos de seleção de lista
107             public void valueChanged( ListSelectionEvent evento )
108             {
109                 esquerda.setBackground(
```

```
110                 cores[ listaDeCores.getSelectedIndex() ] );
111             }
112         }
113         );
114
115         setSize( 400, 250 );
116         setVisible( true );
117     }
118
119     // executa a aplicacao
120     public static void main( String args[] )
121     {
122         TesteJList aplicacao = new TesteJList();
123
124         aplicacao.setDefaultCloseOperation(
125         JFrame.EXIT_ON_CLOSE );
126     }
127 }
```

4.10 JPopupMenus

São menus sensíveis ao contexto, ou seja, em virtude da localização do cursor do mouse, um clique no botão direito do mesmo dispara um evento que abre um menu flutuante. Tal menu fornece opções selecionáveis ao determinado componente por sobre o qual o evento de diparo foi gerado.

Em nosso exemplo, dividimos a área do aplicativo em cinco regiões, sendo que cada uma pode ser pintada com uma das cores oferecidas pelas opções do JPopupMenu.

Para compormos os itens do menu, usamos um array de JRadioButtonMenuItem, que nada mais são que os botões descritos em 4.2.3, mas agora com capacidade de serem incluidos em um menu. Cada item é adicionado ao menu com o método **add(JMenuItem)** e registra um tratador de eventos passando como argumento ao método **addActionListener (ActionEvent)** um objeto da classe **TratadorDeItem**. Novamente o relacionamento lógico que "cola" os botões e só permite que um seja selecionado é criado com um objeto **ButtonGroup**.

Criamos um método chamado **verificaEventoDeDisparo(Mouse Event)** que verifica se o evento de disparo ocorreu. Para isso, utilizamos o método **isPopupTrigger( )**, de MouseEvent, que retorna verdadeiro se o evento ocorreu, nesse caso, validando a execução da estrutura condicional posterior. Ela chama o método **show(Component, int, int)** da classe **JPopupMenu**, que em seu primeiro argumento especifica o componente que originou o evento e nos dois argumentos seguintes define as coordenadas x e y relativas ao canto superior esquerdo do elemento de origem sobre o qual o menu deve aparecer. Funcionalmente, esse método exibe o canto superior esquerdo do menu exatamente onde o evento de disparo ocorreu.

Também criamos o método **estou (int x, int y)** que, baseado na posição onde o cursor do mouse estiver, retorna o componente que se encontra nessa respectiva coordenada.

```
1 // Demonstra o uso do JPopupMenu
2
```

Figura 4.10: Interface do exemplo que usa JPopupMenu

```
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class TesteJPopupMenu extends JFrame {
8
9      private JRadioButtonMenuItem itens[];
10     private Color cores[] = { Color.blue, Color.yellow, Color.red,
11     Color.green, Color.orange };
12
13     private JPopupMenu popupMenu;
14     private JPanel norte, sul, leste, oeste, centro;
15     private int x = 0;
16     private int y = 0;
17
18     // configura a GUI
19     public TesteJPopupMenu()
20     {
21         super( "Testando JPopupMenu" );
22
23         TratadorDeItem trat = new TratadorDeItem();
24         String nomeDasCores[] = { "Azul", "Amarelo", "Vermelho",
25         "Verde", "Laranja" };
26
27         // configura o JPopupMenu para selecioanar somente um dos
28         // seus cinco itens
29         ButtonGroup umaCor = new ButtonGroup();
30         popupMenu = new JPopupMenu();
31         itens = new JRadioButtonMenuItem[ 5 ];
32
33         Container container = getContentPane();
34         container.setLayout(new BorderLayout());
35
36         // constrói cada item de menu
37         for ( int count = 0; count < itens.length; count++ ) {
38             itens[ count ] = new JRadioButtonMenuItem( nomeDasCores[ count ] );
39
40             // adiciona os itens ao JPopupMenu e ao botão de seleção única
41             popupMenu.add( itens[ count ] );
42             umaCor.add( itens[ count ] );
43
```

```
44              // registra ouvinte de cada item
45              itens[ count ].addActionListener( trat );
46          }
47          // cria painéis
48          norte = new JPanel();
49          sul = new JPanel();
50          leste = new JPanel();
51          oeste = new JPanel();
52          centro = new JPanel();
53
54          // anexa os painíes em suas respectivas regiões
55          container.add(norte, BorderLayout.NORTH);
56          container.add(sul, BorderLayout.SOUTH);
57          container.add(leste, BorderLayout.EAST);
58          container.add(oeste, BorderLayout.WEST);
59          container.add(centro, BorderLayout.CENTER);
60
61          // define um ouvidor para a janela da aplicação, a qual exibe
62          // um JPopupMenu quando ocorre o evento de acionamento do mesmo(right-click)
63          this.addMouseListener(
64
65          // classe interna anônima  para tratar eventos do mouse (right-click)
66          new MouseAdapter() {
67
68              // trata eventos do pressionamento do mouse
69              public void mousePressed( MouseEvent evento )
70              {
71                  verificaEventoDeDisparo( evento );
72              }
73
74              // trata de eventos de liberação do mouse
75              public void mouseReleased( MouseEvent evento )
76              {
77                  verificaEventoDeDisparo( evento );
78              }
79
80              // determina se o evento deve acionar o JPopupMenu
81              private void verificaEventoDeDisparo( MouseEvent evento )
82              {
83                  x = evento.getX();  // armazena a abcissa
84                  y = evento.getY();  // armazena a ordenada
85
86                  // devolve true se o disparo do JPopupMenu ocorreu
87                  if ( evento.isPopupTrigger() )
88                  {
89                          // exibe o JPopupMenu onde ocorreu o disparo do evento
90                      popupMenu.show( evento.getComponent(),
91                      evento.getX(), evento.getY() );
92                  }
93              }
94          }
95          );
96
97          setSize( 300, 200 );
98          setVisible( true );
99      }
100
101     // executa a  aplicação
102     public static void main( String args[] )
```

```
103     {
104         TesteJPopupMenu aplicacao = new TesteJPopupMenu();
105 
106         aplicacao.setDefaultCloseOperation(
107         JFrame.EXIT_ON_CLOSE );
108     }
109 
110     // retorna o painel sobre o qual o mouse foi clicado
111     public Component estou ( int valorX , int valorY )
112     {
113         return findComponentAt( valorX, valorY);
114     }
115 
116     // classe interna anônima para tratar eventos do mouse (click)
117     private class TratadorDeItem implements ActionListener {
118 
119         // processa a seleção de cada item do JPopupMenu
120         public void actionPerformed( ActionEvent evento )
121         {
122             // determina qual item do menu foi selecionado
123             for ( int i = 0; i < itens.length; i++ )
124             {
125                 if ( evento.getSource() == itens[ i ] ) {
126 
127                     // pinta o componente (painel) sobre o qual
128                     // o mouse foi clicado
129                     estou(x,y).setBackground( cores[ i ] );
130                     repaint();
131                     return;
132                 }
133             }
134         }
135     }
136 }
```

4.11 Menus

Muito familiares a nós, o menus talvez sejam os componentes que mais aparecem nas ferramentas computacionais que utilizamos. Geralmente eles são encontrados no topo da janela da aplicação, de onde dão suporte à organização e agrupamento de funções afins em um mesmo contexto visual, o que facilita muito a localização e entendimento por parte do usuário, já que a estrutura de cada menu está delineada pelas características dos itens.

Os menus, que são instanciados a partir da classe **JMenu**, são anexados a uma barra de menus com o método **add(JMenu)** de **JMenuBar**, sendo que instâncias dessa última classe comportam-se como conteiners para menus. A classe **JMenuBar** fornece os métodos necessários ao gerenciamento da barra onde os menus são anexados. A ordenação dos mesmos depende da ordem em que foram adicioandos, sendo que são "empilhados" horizontalmente da esquerda para a direita. Evidentemente, só podemos anexar menus à janelas da classe **JApplet**, **JDialog**, **JFrame** e **JInternalFrame**, e fazemos isso usando o método **setJMenuBar(JMenuBar)**.

A classe **JMenuItem** capacita a criação de itens de menu que, por sua vez, devem ser anexados a um menu. Podemos usar um item de menu para

executar alguma ação ou para gerir o acionamento de um submenu, o qual fornece mais itens que estão relacionados por alguma característica comum. Veremos isso em nosso exemplo e ainda outras funcionalidades, tais como inserir uma figura, alterar o estilo, a cor da fonte e a própria fonte de um rótulo.

Como você bem sabe, os menus comportam o uso de caracteres mnemônicos e os nossos não poderia ficar para traz. Outra novidade é o uso de objetos **JCheckBoxMenuItem**, que são semelhantes às caixas de marcação vistas em 4.2.2, só que aqui aparecem dentro de um menu, e **JRadioButtonMenuItem**, que são muito parecidos com os botões de rádio descritos em 4.2.3. Aqui eles também encontram-se representando itens de menu de seleção única.

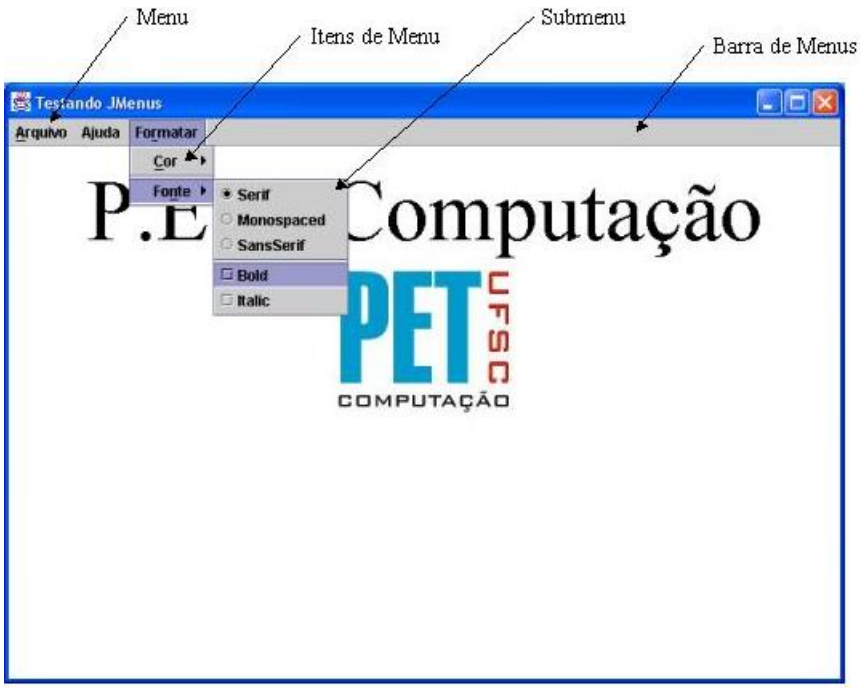


Figura 4.11: Interface do exemplo que usa JMenu

```
1  // Demonstra Jmenu
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class TesteJMenu extends JFrame {
8      private Color colorValues[] =
9                  { Color.black, Color.blue, Color.red, Color.green };
10
11     private JRadioButtonMenuItem itensDasCores[], fontes[];
12     private JCheckBoxMenuItem itensDosEstilos[];
13     private JLabel texto;
14     private ButtonGroup umaUnicaFonte, umaUnicaCor;
15     private int estilo;
16     private Icon carta = new ImageIcon("figuras/carta.gif");
17     private Icon figura1 = new ImageIcon("figuras/logo.jpg");
```

```
18
19      // configura a GUI
20      public TesteJMenu() {
21          // configura título da barra de título
22          super("Testando JMenus");
23
24          // obtém painél de conteudo
25          Container container = getContentPane();
26          container.setLayout(new FlowLayout());
27
28          /*****************MENU ARQUIVO**********************/
29
30          // configura o menu "Arquivo" e seus itens de menu
31          JMenu menuArquivo = new JMenu("Arquivo");
32          menuArquivo.setMnemonic('A');
33
34          JMenuItem itemFigura1 = new JMenuItem("Figura1");
35          itemFigura1.setMnemonic('F');
36          itemFigura1.addActionListener(
37
38          // classe interna anônima para tratar eventos do item de menu "Figura1"
39          new ActionListener() {
40
41                  // insere uma figura na janela do aplicativo quando o
42                  // usuário clica no item "Figura1"
43                  public void actionPerformed(ActionEvent evento) {
44                  inserir();
45                  }
46          });
47
48          menuArquivo.add(itemFigura1);
49
50          // configura o item de menu "Sair"
51          JMenuItem itemSair = new JMenuItem("Sair");
52          itemSair.setMnemonic('S');
53          itemSair.addActionListener(
54
55          // classe interna anônima para tratar eventos do item de menu "itemSair"
56          new ActionListener() {
57
58                  // finaliza o aplicativo quando o usuário clica no
59                  // item "Sair"
60                  public void actionPerformed(ActionEvent evento) {
61                          System.exit(0);
62                  }
63          });
64
65          menuArquivo.add(itemSair);
66
67          /*****************MENU AJUDA**********************/
68
69          // configura o menu "Ajuda" e seus itens de menu
70          JMenu menuAjuda = new JMenu("Ajuda");
71          menuAjuda.setMnemonic('H');
72
73          //configura o item de menu "Universitários"
74          JMenuItem itemUniversitarios = new JMenuItem("Universitários");
75          itemUniversitarios.setMnemonic('U');
76          itemUniversitarios.addActionListener(
```

```
77
78          // classe interna anônima para tratar eventos do item
79          // de menu "Universitários"
80          new ActionListener() {
81
82                  // exibe um diálogo de mensagem quando "Universitários"
83                  // é selecionado
84                  public void actionPerformed(ActionEvent event) {
85                          JOptionPane.showMessageDialog(
86                                  TesteJMenu.this,
87                                  "Você não é um Universitário?\nEntão...",
88                                  "Ajuda",
89                                  JOptionPane.PLAIN_MESSAGE);
90                  }
91          });
92
93          menuAjuda.add(itemUniversitarios);
94
95          // configura o item de menu "Cartas"
96          JMenuItem itemCartas = new JMenuItem("Cartas");
97          itemCartas.setMnemonic('C');
98          itemCartas.addActionListener(
99
100         // classe interna anônima para tratar eventos de item
101         // de menu "Cartas"
102         new ActionListener() {
103
104                 // exibe um diálogo de mensagem quando "Cartas"
105                 // é selecionado
106                 public void actionPerformed(ActionEvent event) {
107                         JOptionPane.showMessageDialog(
108                                 TesteJMenu.this,
109                                 "Não deu Sorte!!!",
110                                 "Cartas",
111                                 JOptionPane.PLAIN_MESSAGE,
112                                 carta);
113                 }
114         });
115
116         menuAjuda.add(itemCartas);
117
118         // cria e anexa a barra de menu à janela TesteJMenu
119         JMenuBar barra = new JMenuBar();
120         setJMenuBar(barra);
121
122         // anexa os menus "Arquivo" e "Ajuda" à barra de menu
123         barra.add(menuArquivo);
124         barra.add(menuAjuda);
125
126         /*****************MENU FORMATAR**********************/
127
128         // cria o menu Formatar, seus submenus e itens de menu
129         JMenu formatMenu = new JMenu("Formatar");
130         formatMenu.setMnemonic('r');
131
132         // cria os nomes do submenu "Cor"
133         String cores[] = { "Preto", "Azul", "Vermelho", "Verde" };
134
135         JMenu menuCor = new JMenu("Cor");
```

```
136         menuCor.setMnemonic('C');
137
138         itensDasCores = new JRadioButtonMenuItem[cores.length];
139         umaUnicaCor = new ButtonGroup();
140         TratadorDeItens trat = new TratadorDeItens();
141
142         // cria itens do menu "Cor" com botões de opção
143         for (int count = 0; count < cores.length; count++) {
144                 itensDasCores[count] = new JRadioButtonMenuItem(cores[count]);
145
146                 menuCor.add(itensDasCores[count]);
147                 umaUnicaCor.add(itensDasCores[count]);
148
149                 itensDasCores[count].addActionListener(trat);
150         }
151
152         // seleciona o primeiro item do menu "Cor"
153         itensDasCores[0].setSelected(true);
154
155         // anexa o menu "menuCor" ao menu "formatMenu"
156         formatMenu.add(menuCor);
157
158         // insere uma barra separadora
159         formatMenu.addSeparator();
160
161         // cria o submenu "Fonte"
162         String nomeDasFontes[] = { "Serif", "Monospaced", "SansSerif" };
163
164         JMenu menuFonte = new JMenu("Fonte");
165         menuFonte.setMnemonic('n');
166
167         fontes = new JRadioButtonMenuItem[3];
168
169         // implementa a exclusão mútua dos itens
170         umaUnicaFonte = new ButtonGroup();
171
172         // cria itens do menu "Fonte" com botões de opção
173         for (int count = 0; count < fontes.length; count++) {
174                 fontes[count] = new JRadioButtonMenuItem(nomeDasFontes[count]);
175
176                 menuFonte.add(fontes[count]);
177                 umaUnicaFonte.add(fontes[count]);
178
179                 // registra o tratador de eventos para os JRadioButtonMenuItens
180                 fontes[count].addActionListener(trat);
181         }
182
183         // seleciona o primeiro item do menu "Fonte"
184         fontes[0].setSelected(true);
185
186         // insere uma barra separadora
187         menuFonte.addSeparator();
188
189         // configura os itens de estilo do menu "Fonte"
190         String estiloNames[] = { "Bold", "Italic" };
191
192         itensDosEstilos = new JCheckBoxMenuItem[estiloNames.length];
193         TratadorDeEstilo estiloHandler = new TratadorDeEstilo();
194
```

```
195         // cria os itens de estilo do menu
196         for (int count = 0; count < estiloNames.length; count++) {
197                 itensDosEstilos[count] = new JCheckBoxMenuItem(estiloNames[count]);
198
199                 menuFonte.add(itensDosEstilos[count]);
200
201                 itensDosEstilos[count].addItemListener(estiloHandler);
202         }
203
204         // anexa o menu "Fonte" ao menu "Formatar"
205         formatMenu.add(menuFonte);
206
207         // anexa o menu "Formatar" à barra de menu
208         barra.add(formatMenu);
209
210         // configura o rótulo para exibir o texto
211         texto = new JLabel("P.E.T. Computação", SwingConstants.CENTER);
212         texto.setForeground(colorValues[0]);
213         texto.setFont(new Font("TimesRoman", Font.PLAIN, 72));
214
215         container.setBackground(Color.white);
216         container.add(texto);
217
218         setSize(700, 500);
219         setVisible(true);
220     }
221
222         // insere o logo do PET na janela do aplicativo
223         public void inserir() {
224                 JLabel labelFigura1 = new JLabel();
225                 labelFigura1.setIcon(figura1);
226                 this.getContentPane().add(labelFigura1);
227                 this.repaint();
228                 this.show();
229         }
230
231         // executa a aplicação
232         public static void main(String args[]) {
233                 TesteJMenu application = new TesteJMenu();
234
235                 application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
236         }
237
238         // classe interna anônima para tratar eventos de ação
239         // dos itens de menu
240         private class TratadorDeItens implements ActionListener {
241
242                 // processa as seleções de cor e fonte
243                 public void actionPerformed(ActionEvent evento) {
244                         // processa seleção de cor
245                         for (int count = 0; count < itensDasCores.length; count++)
246                                 if (itensDasCores[count].isSelected()) {
247                                         texto.setForeground(colorValues[count]);
248                                         break;
249                                 }
250
251                         // processa seleção de fonte
252                         for (int count = 0; count < fontes.length; count++)
253                                 if (evento.getSource() == fontes[count]) {
```

```
254                                     texto.setFont( new Font(fontes[count].getText(),
255                                                    estilo, 72));
256                                        break;
257                                 }
258
259                         repaint();
260                 }
261         }
262
263         // classe interna anônima  que trata eventos dos itens
264         // de menu que usam caixa de marcação
265         private class TratadorDeEstilo implements ItemListener {
266
267                 // processa seleção de estilo das fontes do Label
268                 public void itemStateChanged(ItemEvent item) {
269                         estilo = 0;
270
271                         // verifica se negrito foi selecionado
272                         if (itensDosEstilos[0].isSelected())
273                                 estilo += Font.BOLD;
274
275                         // verifica se itálico foi selecionado
276                         if (itensDosEstilos[1].isSelected())
277                                 estilo += Font.ITALIC;
278
279                         texto.setFont(new Font(texto.getFont().getName(),
280                                                estilo, 72));
281                         repaint();
282                 }
283         }
284 }
```

Capítulo 5

Eventos

Eventos são o resultado da interação do usuário com algum componente GUI. Mover o mouse, clicá-lo, digitar num campo de texto, selecionar um item de menu, fechar uma janela, clicar num botão, etc. são interações que enviam eventos para o programa, normalmente realizando serviços.

Eventos também podem ser gerados em resposta a modificações do ambiente, como por exemplo, quando a janela de um applet é coberta por outra janela. Em outras palavras, define-se eventos GUI como mensagens (chamadas a métodos) que indicam que o usuário do progama interagiu com um dos componentes GUI.

5.1 Tratamento de Eventos

O mecanismo de tratamtendo de eventos compreende três partes: a origem, o objeto e o ouvinte do evento.

5.1.1 A Origem do Evento

É o componente GUI em particular com o qual o usuário interage.

5.1.2 O Objeto Evento

Dada a interação com algum componente, um objeto evento é criado. Ele encapsula as informações sobre o evento que ocorreu, incluindo uma referência para a origem e demais dados necessários para que o ouvinte do evento o trate.

5.1.3 Ouvinte do Evento

É um objeto de uma classe que implementa uma ou mais das interfaces *listeners* de eventos dos pacotes **java.awt.event** e **javax.swing.event**. Ele é notificado da ocorrência de um evento e usa o objeto evento que recebe para, de acordo com seus métodos de tratamento de eventos, responder ao evento. Para isso o ouvinte deve ser registrado e implementar a interface correspondente ao(s) evento(s) que deseja tratar. Cada fonte de eventos pode ter mais de um ouvinte registrado. Analogmente, um ouvinte pode registrar multiplas fontes de eventos.

Basicamente, quando ocorre um evento (precionar um JButton, por exemplo) , o componente GUI com o qual o usuário interagiu notifica seus ouvintes registrados chamando o método de tratamento de evento (como você verá, é o **ActionPerformed**, nesse caso) apropriado de cada ouvinte. Esse estilo de progamação é conhecido como programação baseada em eventos.

5.2 Tratadores de Eventos ou Ouvintes *(Listeners)*

São objetos de qualquer classe que implemente uma interface específica para o tipo de evento que se deseja tratar. Essa interface é definida para cada classe de eventos. Para a classe de eventos **java.awt.eventFocusEvent** existe a interface **java.awt.eventFocusListener**, por exemplo. Vamos explorar esse assunto nestas próximas seções, descrevendo os métodos definidos por cada interface[1] e em decorrência de quais ações eles são chamados.

5.2.1 ActionListener

A partir dessa interface, instânciamos objetos que "sabem" tratar eventos de ação.

public void actionPerformed(ActionEvent)

Invocado quando clicamos em um botão, pressionamos *Enter* enquanto digitamos em um campo de texto ou selecionamos um item de menu.

5.2.2 FocusListener

Trata de eventos de visibilidade, ou seja, quando o componente fica no foco de ação do teclado (primeiro plano), ganhando ou perdendo abilidade de receber entradas do mesmo. Os métodos recebem como argumento um objeto da classe FocusEvent.

public void focusGained(FocusEvent)

Chamado somente depois que o componente ganha o primeiro plano de ação.

public void focusLost(FocusEvent)

Chamado somente depois que o componente perde o foco de ação.

5.2.3 ItemListener

Compreende eventos relativos a marcação, onde existe a possibilidade do estado selecionado e não-selecionado. Por exemplo, as opções de JCheckBox, JCheckBoxItem e JCombobox.

public itemStateChanged(ItemEvent) Invocado após o componente sofrer um mudança de estado.

[1] Apresentamos somente as interfaces que julgamos de maior interesse.

5.2.4 KeyListener

Aqui apresentaremos a interface *listener* de eventos KeyListener, que trata dos eventos de pressionamento e liberação das teclas. Uma classe que implementa esta interface deve fornecer definição para os métodos:

public void KeyPressed (KeyEvent)

Chamado quando se pressiona qualquer tecla.

public void KeyReleased (KeyEvent)

Chamado quando se libera qualquer tecla.

public void KeyTyped (KeyEvent)

Chamado quando se pressiona uma tecla de ação (setas, Home, End, Page Up, Page Down) ou de função (Num Lock, Caps Lock, Scroll Lock, Pause, Print Screen). Se quiser ver isso na prática, compile nosso exemplo:

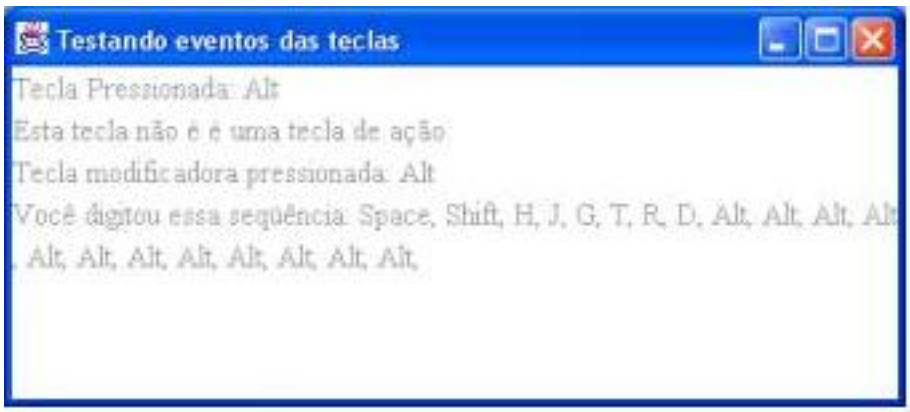


Figura 5.1: Interface do exemplo que demonstra as Atividades do Teclado

```
1  // Demonstra eventos das teclas
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class TesteDasAtividadesDoTeclado extends JFrame
8  implements KeyListener {
9
10     private String linha1 = "";
11     private String linha2 = "";
12     private String linha3 = "";
13     private JTextArea areaTexto;
14     private String teclasDigitadas = "";
15
16     // configura a GUI
17     public TesteDasAtividadesDoTeclado()
18     {
19         //  texto da barra de título
```

```
20          super( "Testando eventos das teclas" );
21
22          // configura a JTextArea
23          areaTexto = new JTextArea( 10, 15 );
24                  areaTexto.setFont(new Font("Serif", Font.TRUETYPE_FONT, 20));
25          areaTexto.setText( "Pressione qualquer tecla..." );
26          areaTexto.setEnabled( false );
27
28          areaTexto.setLineWrap( true );
29          //areaTexto.setBackground(Color.BLUE.brighter());
30          getContentPane().add( areaTexto );
31
32          // registra a janela para processar os eventos de teclas
33          addKeyListener( this );
34
35          setSize( 450, 200 );  // dimensiona a janela
36          setVisible( true );  // exibe a janela
37      }
38
39      // trata o pressionamento de qualquer tecla
40      public void keyPressed( KeyEvent evento )
41      {
42          linha1 = "Tecla Pressionada: " +
43          evento.getKeyText( evento.getKeyCode() );
44          configLinha2e3( evento );
45          teclasDigitadas = teclasDigitadas +
46          evento.getKeyText( evento.getKeyCode()) +", ";
47      }
48
49      // trata a liberação de qualquer tecla
50      public void keyReleased( KeyEvent evento )
51      {
52          linha1 = "Tecla Liberada: " +
53          evento.getKeyText( evento.getKeyCode() );
54          configLinha2e3( evento );
55      }
56
57      //  trata o pressionamento de uma tecla de ação
58      public void keyTyped( KeyEvent evento )
59      {
60          linha1 = "Tecla Acionada: " + evento.getKeyChar();
61          configLinha2e3( evento );
62      }
63
64      // configura a segunda e a terceira linha do output
65      private void configLinha2e3( KeyEvent evento )
66      {
67          linha2 = "Esta tecla " +
68          ( evento.isActionKey() ? "" : "não " ) +
69          "é uma tecla de ação";
70
71          String temp =
72          evento.getKeyModifiersText( evento.getModifiers() );
73
74          linha3 = "Tecla modificadora pressionada: " +
75          ( temp.equals( "" ) ? "nenhuma" : temp );
76
77          areaTexto.setText( linha1 + "\n" + linha2 + "\n" + linha3 +
78          "\n" +"Você digitou essa seqüência: " + teclasDigitadas );
```

```
79      }
80
81      // executa a aplicação
82      public static void main( String args[] )
83      {
84          TesteDasAtividadesDoTeclado aplicacao =
85          new TesteDasAtividadesDoTeclado();
86
87          //  configura o encerramento da aplicação
88          aplicacao.setDefaultCloseOperation(
89          JFrame.EXIT_ON_CLOSE );
90      }
91  }
```

5.2.5 MouseListener

Agora apresentaremos a interface *listener* de eventos MouseListener, que trata dos eventos de pressionamento e liberação dos botões do mouse. Uma classe que implementa esta interface deve fornecer definição para os métodos[1]:

public void mousePressed(MouseEvent)

Chamado quando se pressiona um botão do mouse com o cursor sobre um componente.

public void mouseClicked(MouseEvent)

Chamado quando pressiona-se e libera-se um botão do mouse sobre um componente, sem mover o cursor.

public void mouseReleased(MouseEvent)

Chamado quando se libera um botão do mouse depois de ser pressionado. As chamadas para este método são enviadas para o ouvinte de eventos do componente sobre o qual a operação de arrastar iniciou. Esse evento sempre é precedido por um evento **mousePressed**.

public void mouseEntered(MouseEvent)

Chamado quando o cursor do mouse entra nos limites de um componente.

public void mouseExited(MouseEvent)

Chamado quando o cursor do mouse sai dos limites de um componente.

5.2.6 MouseMotionListener

A interface *listener* de eventos MouseMotionListener trata dos eventos de "arrasto" do mouse. Uma classe que implementa esta **interface** deve fornecer definição para os métodos[1]:

public void mouseDragged(MouseEvent)

Chamado quando se pressiona o botão do mouse com o cursor sobre um componente e se move o mouse. As chamadas para este método são enviadas para o ouvinte de eventos do componente sobre o qual a operação de arrastar iniciou. Esse evento é sempre precedido por uma chamada **mousePressed**.

public void mouseMoved(MouseEvent)

Chamado quando se move o mouse com o cursor sobre um componente.

Os eventos do mouse podem ser capturados por qualquer componente GUI que se derive de java.awt.Component (painéis, botões, etc.), sendo que o componente deve ter um objeto *listener* registrado. Todos esses métodos recebem um objeto **MouseEvent** como argumento, o qual encapsula as informações sobre o evento que ocorreu, incluindo as coordenadas x e y da posição em que o mesmo verificou-se. Consulte nosso próximo exemplo para solidificar seus conhecimentos:

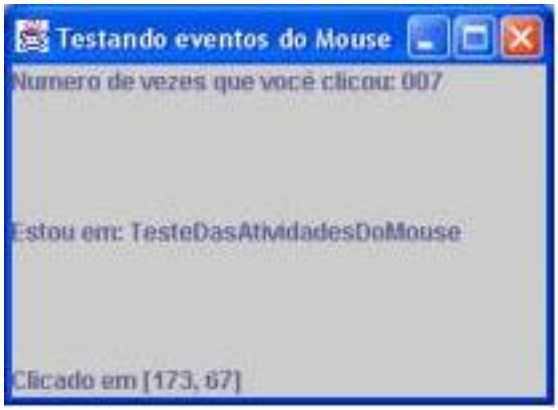


Figura 5.2: Interface do exemplo que demonstra as Atividades do Mouse

```
1  // Demonstra eventos do mouse
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6  import java.text.DecimalFormat;
7
8  public class TesteDasAtividadesDoMouse extends JFrame
9  implements MouseListener, MouseMotionListener {
10
11     private JLabel estado,labelNumeroClicks,estouEm;
12     private int numeroClicks = 0;
13     private JButton limpar;
14
15     // configura a GUI
16     public TesteDasAtividadesDoMouse()
17     {
18         // texto da barra de título
19         super( "Testando eventos do Mouse" );
20
21         estado = new JLabel();
```

```
22          labelNumeroClicks = new JLabel();
23          estouEm = new JLabel();
24          Container container = getContentPane();
25          container.add( labelNumeroClicks, BorderLayout.NORTH );
26          container.add( estado, BorderLayout.SOUTH );
27          container.add( estouEm, BorderLayout.CENTER );
28
29          // a janela do aplicativo espera por seus próprios
30          // eventos do mouse
31          addMouseListener( this );
32          addMouseMotionListener( this );
33
34          setSize( 275, 200 );  // dimensiona a janela
35          setVisible( true );   // exibe a janela
36      }
37
38      // >>> tratador de eventos MouseListener <<<
39
40      // trata evento do mouse quando um botão é liberado
41      // imediatamente após ser pressionado
42      public void mouseClicked( MouseEvent evento )
43      {
44          DecimalFormat valor = new DecimalFormat("000");
45          numeroClicks = numeroClicks + evento.getClickCount();
46          labelNumeroClicks.setText("Numero de vezes que você clicou: " +
47          valor.format(numeroClicks));
48          estado.setText( "Clicado em [" + evento.getX() +
49          ", " + evento.getY() + "]" );
50      }
51
52      // trata evento quando um botão do mouse é pressionado
53      public void mousePressed( MouseEvent evento )
54      {
55          estado.setText( "Pressionado em [" + evento.getX() +
56          ", " + evento.getY() + "]" );
57      }
58
59      // trata evento do mouse quando ele é liberado após
60      // ser arrastado
61      public void mouseReleased( MouseEvent evento )
62      {
63          estado.setText( "Liberado em [" + evento.getX() +
64          ", " + evento.getY() + "]" );
65      }
66
67      // trata evento do mouse quando ele entra na área da janela
68      public void mouseEntered( MouseEvent evento )
69      {
70          estouEm.setText( "Estou em: " +
71          evento.getComponent().getClass().getName());
72          labelNumeroClicks.setText( "Mouse dentro da janela" );
73      }
74
75      // trata evento do mouse quando ele sai da área da janela
76      public void mouseExited( MouseEvent evento )
77      {
78          estado.setText( "Mouse fora da janela" );
79      }
80
```

```
81      // >>> tratadores de eventos MouseMotionListener <<<
82
83      // trata evento quando o usuário arrasta o mouse com
84      // o botão pressionado
85      public void mouseDragged( MouseEvent evento )
86      {
87          estado.setText( "Arrastado em [" + evento.getX() +
88          ", " + evento.getY() + "]" );
89      }
90
91      // trata evento quando o usuário move o mouse
92      public void mouseMoved( MouseEvent evento )
93      {
94          estado.setText( "Movido em [" + evento.getX() +
95          ", " + evento.getY() + "]" );
96      }
97
98      // executa a aplicação
99      public static void main( String args[] )
100     {
101         TesteDasAtividadesDoMouse aplicacao =
102         new TesteDasAtividadesDoMouse();
103
104         // configura o encerramento da aplicação
105         aplicacao.setDefaultCloseOperation(
106         JFrame.EXIT_ON_CLOSE );
107     }
108 }
```

5.2.7 WindowListener

Todas as janelas geram eventos quando o usuário as manipula. Os ouvintes (*listeners*) de eventos são registrados para tratar eventos de janela com o método **addWindowListener(WindowListener)** da classe **Window**.

A interface WindowListener, que é implementada por ouvintes de eventos de janela, fornece sete métodos para tratar esses eventos, Todos os métodos recebem um objeto da classe **WindowEvent**. Vejamos eles:

public void windowActivated(WindowEvent)

Chamado quando o usuário torna uma janela ativa.

public void windowClosed(WindowEvent)

Chamado depois que a janela é fechada.

public void windowClosing (WindowEvent)

Chamado quando o usuário inicia o fechamento da janela.

public void windowDesactivated(WindowEvent)

Chamado quando o usuário torna outra janela a ativa.

public void windowIconified(WindowEvent)

Chamado quando o usuário minimiza a janela.

public void windowDeiconified(WindowEvent)

Chamado quando o usuário restaura uma janela minimiza.

public void windowOpened(WindowEvent)

Chamado quando uma janela é exibida pela primeira vez na tela.

5.3 Classes Adaptadoras

A premissa de que uma classe implementa uma **interface** implica que o programador deverá definir todos os métodos declarados nessa **interface**. Porém, nem sempre é desejável definir todos os métodos. Podemos construir aplicações que utilizem apenas o método tratator de eventos **mouseClicked** da **interface MouseListener**, por exemplo. Para muitas das **interfaces** *listeners* que contém vários métodos, os pacotes **java.awt.event** e **javax.swing.event** fornecem classes adaptadoras de ouvintes de eventos. Essas classes implementam uma interface e fornecem cada método implementado com um o corpo vazio. O programador pode criar uma classe que herde da classe adaptadora todos os métodos com a implementação *default*(corpo vazio) e depois sobrescrever o(s) método(s) necessário(o) para o tratamento de eventos.

Vejamos as classes adaptadoras que implementam as respectivas interfaces:

Classe Adaptadora	**Interface**
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

5.4 Classes Internas Anônimas

Tratadores de eventos podem ser instâncias de qualquer classe. Muitas vezes, se os mesmos possuírem poucas linhas de código, os implementamos usando um classe interna anônima, que é definida dentro de outra classe. Pode parecer confuso, mas elas permitem que a implementação dos tratadores de eventos fique próxima de onde o tratador de eventos é registrado, favorecendo a composição de um código compacto. Um objeto da classe interna pode acessar diretamente todas as variáveis de instância e métodos do

objeto da classe externa que o definiu. Você já deve ter notado que em nossos exemplos esse tipo de implementação é corriqueiro.

Veremos agora como definir uma classe interna anônima e criar um objeto dessa classe, que é passado como argumento do método **addActionListener(ActionListener)**. Usa-se a sintaxe especial de Java:

```
1  // registra tratador de eventos de algumComponente
2  algumComponente.addActionListener(
3
4          // classe interna anônima
5          new ActionListener(){
6
7                  public void actionPerformed(actionEvent evento)
8                  {
9                  ...// aqui vai o código que responde à ação
10                 }
11
12         }  // fim da classe interna anônima
13
14 );  // fim da chamada para addActionListener
```

Utiliza-se o operador **new** para criar o objeto. A sintaxe **ActionListener( )** começa a definição da classe interna anônima que implementa a *interface* ActionListener. Os parentêses depois de **ActionListener** indicam uma chamada ao construtor *default* da classe interna anônima. Isso assemelha-se ao código sequente:

```
1  public class TratadorDeAcao implements ActionListener
2  {
3         public void actionPerformed(ActionEvent evento)
4         {
5         ...// aqui vai o código que responde à ação
6         }
7
8  }
```

Também é possível registrar a classe da aplicação como ouvidora de eventos. Nesse caso, os métodos de tratamento de eventos são declarados no escopo da classe da aplicação ou do applet, que por sua vez, deve implementar uma interface listener de eventos.

5.5 Como implementar um Tratador de Eventos

Podemos definir uma classe que processe eventos de duas formas:

- Implementando uma interface (KeyListener, MouseListener, etc.);
- Extendendo uma classe adaptadora (KeyAdapter, MouseAdapter, etc.).

Na declaração da classe tratadora de eventos, o código que especifica que a classe implementa uma interface listener é o seguinte:

```
public class UmaClasse implements ActionListener
{
    // código que implementa o método listener da interface:
    public void actionPerformed (ActionEvent evento)
    {
        ...// aqui vai o código que responde à ação
    }

}
```

Já o código que representa que uma classe herda de uma outra que implementa uma interface listener é:

```
public class OutraClasse extends WindowAdapter
{
    // código que implementa o método listener da interface:
    public void windowClosing (WindowEvent evento)
    {
        ...// aqui vai o codigo que responde à ação
    }

}
```

O código que registra uma instância da classe tratadora de eventos como ouvidor para um ou mais componentes é:

```
algumComponente.addActionListener(objeto da classe UmaClasse);

// ou

algumComponente.addWindowListener(objeto da classe OutraClasse);
```

Para recapitular tudo o que vimos até aqui, vamos examinar uma situação típica de tratamento de eventos, baseado-se em como os JButtons tratam o evento de pressionar o mouse sobre eles.

Para detectar quando o usuário clica no componente GUI, o programa deve fornecer um objeto que implemente a interface **ActionListener**. Deve-se registrar este objeto como um ouvinte de ação do botão (que é a origem do evento), valendo-se do método **addActionListener(ActionListener)**. Quando o usuário clica no JButton, ele dispara um evento de ação. Isto resulta na invocação do método ouvidor de ação **actionPerformed** (o único da interface ActionListener a ser implementado). O único argumento do método é um objeto ActionEvent que encapsula informações sobre o evento e sua origem.

Considerações Finais

Findado este curso, esperamos ter explicitado de forma agradável os tópicos a que nos propomos elucidar. Sabemos que existem muitos outros componentes que seriam muito bem vindos se alocados nestas páginas, porém o assunto se extenderia demasiadamente.

Fique claro ao leitor que é de interesse do autor incorporar a esse material um tópico referente ao *feedback* visual que um usuário espera, em termos de cores, estruturação e distribuição de componentes. Assim sendo, se é de seu interesse, periodicamente consulte a versão *online* dessa apostila.

Para finalizar, o autor agradece pela atenção dispensada e despedece desejando sucesso ao leitor na aplicação desses conhecimentos.

Referências Bibliográficas

[1] H. M. Deitel, P. J. Deitel *Java Como Programar*. Bookman, Quarta Edição, 2003.

[2] Lisa Friendly, Mary Campione, Kathy Walrath, Alison Huml. *The Java Tutorial*. Sun Microsystems, Terceira Edição, 2003. Disponível para download e online em `http://java.sun.com/docs/books/tutorial/`

[3] Sun Microsystems *Java 2 Platform, Standard Edition, v 1.4.1 API Specification*. Sun Microsystems, 2003. Disponível online e para download em `http://java.sun.com/docs/`

[4] Fábio Mengue *Curso de Java - Módulo II - Swing*. Centro de Computação da Unicamp, 2002. Disponível online em `http://ftp.unicamp.br/pub/apoio/treinamentos/linguagens /curso_java_II.pdf`

[5] Prof. Marcelo Cohen *Interface Gráfica*. Disponível online em `http://www.inf.pucrs.br/~flash/lapro2/lapro2_gui_old.pdf`

[6] Lucas Wanner *Introdução à Linguagem Java*. Versão 1.1, 2002. Disponível online em `http://monica.inf.ufsc.br`