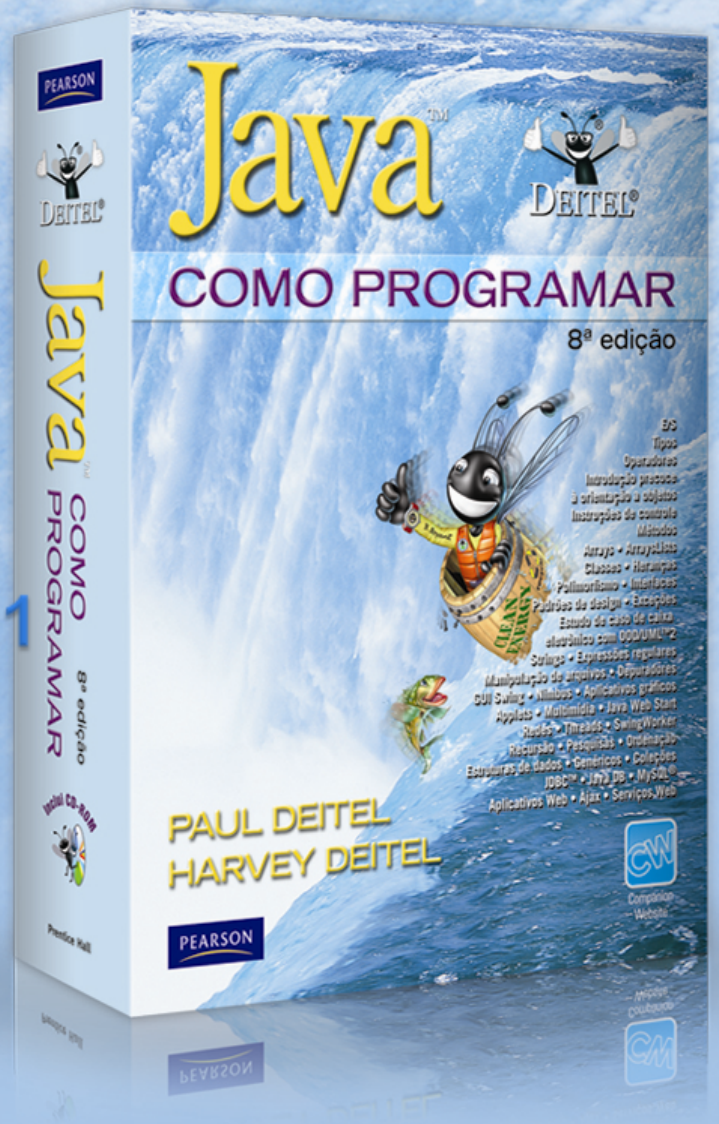
Capítulo 14
Componentes GUI: Parte 1

Java Como Programar, 8/E



OBJETIVOS

Neste capítulo, você aprenderá:

- Os princípios do projeto de interfaces gráficas com o usuário (Graphical User Interfaces — GUIs).
- Como utilizar a nova e elegante interface Nimbus do Java compatível com várias plataformas.
- A construir GUIs e tratar eventos gerados por interações de usuário com GUIs.
- A entender os pacotes que contêm componentes GUI, interfaces e classes de tratamento de evento.
- A criar e manipular botões, rótulos, listas, campos de texto e painéis.
- A tratar eventos de mouse e eventos de teclado.
- A utilizar gerenciadores de layout para organizar componentes GUI.

COMO PROGRAMAR

8ª edição

COMO PROGRAMAR

8ª edição

14.1 Introdução

- Uma **interface gráfica com usuário (*graphical user interface* – GUI)** apresenta um mecanismo amigável ao usuário para interagir com um aplicativo.
- Pronuncia-se “Gui”.
- Fornece a um aplicativo uma “aparência” e “comportamento” distintivos.
- Uma interface com componentes consistentes e intuitivos dá aos usuários um sentido de familiaridade.
- Aprenda os novos aplicativos mais rapidamente e utilize-os mais produtivamente.

Observações sobre a aparência e comportamento 14.1

Interfaces com o usuário consistentes permitem que ele aprenda mais rápido novos aplicativos.

- É construída a partir de **componentes GUI.**
- Às vezes, chamada **controles** ou **widgets** — abreviação de **window gadgets.**
- O usuário interage via mouse, teclado ou outro formulário de entrada, como reconhecimento de voz.
- IDEs
  - Fornecem ferramentas de projeto GUI para especificar o tamanho e a localização exatas de um componente de maneira visual utilizando o mouse.
  - Geram o código GUI para você.
  - Simplificam muito a criação de GUIs, mas cada IDE tem diferentes capacidades e gera códigos distintos.

- Exemplo de uma GUI: aplicativo SwingSet3 (Figura 14.1) download.java.net/javadesktop/swingset3/SwingSet3.jnlp.
- A **barra de título** na parte superior contém o título da janela.
- A **barra de menus** contém **menus** (File e View).
- Na região superior direita da janela está um conjunto de **botões.**
- Em geral, os usuários pressionam os botões para realizar as tarefas.
- Na área GUI Components da janela está uma **caixa de combinação.**
- O usuário pode clicar na seta para baixo no lado direito da caixa para selecionar a partir de uma lista de itens.

Java™ COMO PROGRAMAR

8ª edição

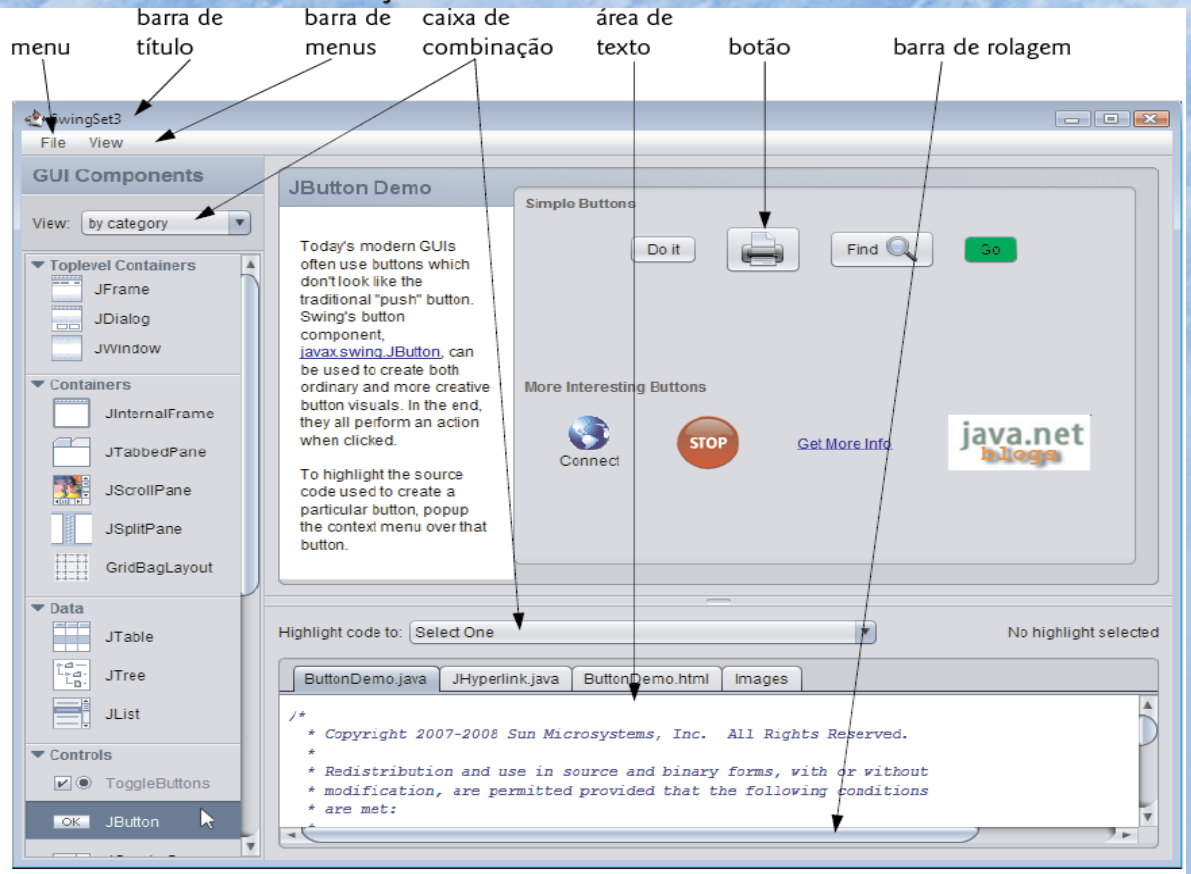


Figura 14.1 | Aplicativo SwingSet3 demonstra muitos dos componentes Swing GUI do Java.

14.2 A nova interface Nimbus do Java

- Java SE 6 update 10.
- Interface nova, elegante e compatível com várias plataformas conhecida como **Nimbus.**
- Configuramos os nossos sistemas para utilizar o Nimbus como a interface padrão.

- Há três modos de utilizar o Nimbus:
  - Configurá-lo como o padrão de todos os aplicativos Java que executam no computador.
  - Configurá-lo como a interface quando se carrega um aplicativo passando um argumento de linha de comando para o comando java.
  - Configurá-lo programaticamente como a interface do seu aplicativo (Seção 25.6).

- Para configurar o Nimbus como o padrão para todos os aplicativos Java:
  - Crie um arquivo de texto chamado `swing.properties` na pasta `lib` tanto na pasta de instalação do JDK como na pasta de instalação do JRE.
  - Insira a seguinte linha do código no arquivo:

```
swing.defaultlaf =
  com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel
```

- Para obter mais informações sobre como localizar essas pastas de instalação, visite `http://java.sun.som/javase/6/webnotes/install/index.html`
- [Nota: Além do JRE autônomo, há um JRE aninhado na pasta de instalação do seu JDK. Se estiver utilizando um IDE que depende do JDK (por exemplo, NetBeans), talvez você também precise inserir o arquivo swing.properties na pasta `lib` da pasta `jre` aninhada.]

- Para selecionar a interface Nimbus individualmente por aplicativo: Coloque o seguinte argumento de linha de comando depois do comando `java` e antes do nome do aplicativo quando for executar o aplicativo:

```
-Dswing.defaultlaf=
  com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel
  NimbusLookAndFeel
```

14.3 Entrada/saída baseada em GUI simples com `JOptionPane`

- A maioria dos aplicativos utiliza janelas ou **caixas de diálogo** (também chamadas de **diálogos**) para interagir com o usuário.
- **JOptionPane** (pacote `javax.swing`) fornece caixas de diálogo pré-construídas tanto para entrada como para saída e são exibidas via métodos `JOptionPane` `static`.
- A Figura 14.2 utiliza dois **diálogos de entrada** para obter inteiros do usuário e um **diálogo de mensagem** para exibir a soma dos inteiros que o usuário insere.


```
1  // Figura 14.2: Addition.java
2  // Programa de adição que utiliza JOptionPane para entrada e saída.
3  import javax.swing.JOptionPane; // programa utiliza JOptionPane
4
5  public class Addition
6  {
7     public static void main( String[] args )
8     {
9        // obtém a entrada de usuário a partir dos diálogos de entrada JOptionPane
10       String firstNumber =
11          JOptionPane.showInputDialog( "Enter first integer" );
12       String secondNumber =
13           JOptionPane.showInputDialog( "Enter second integer" );
14
15       // converte String em valores int para utilização em um cálculo
16       int number1 = Integer.parseInt( firstNumber );
17       int number2 = Integer.parseInt( secondNumber );
18
19       int sum = number1 + number2; // soma os números
20
```

Exibe um diálogo de entrada e retorna um digitado pelo usuário

Figura 14.2 | Programa de adição que utiliza `JOptionPane` para entrada e saída. (Parte 1 de 3.)


```
21          // exibe o resultado em um diálogo de mensagem JOptionPane
22          JOptionPane.showMessageDialog( null, "The sum is " + sum,
23             "Sum of Two Integers", JOptionPane.PLAIN_MESSAGE );
24       } // fim do método main
25    } // fim da classe Addition
```

Exibe um diálogo de mensagem centralizado na tela sem nenhum ícone.

(a) Diálogo de entrada exibido pelas linhas 10–11

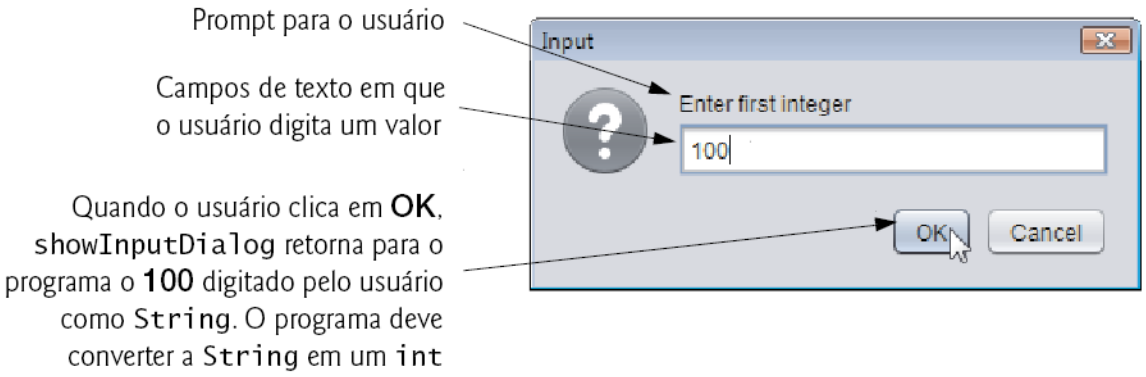
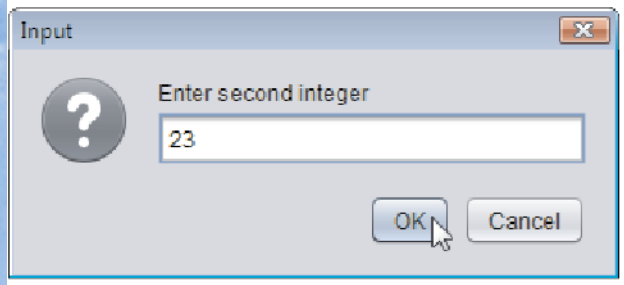
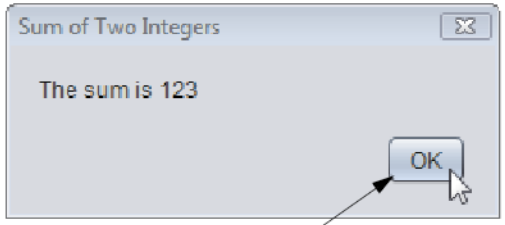


Figura 14.2 | Programa de adição que utiliza `JOptionPane` para entrada e saída. (Parte 2 de 3.)

(b) Diálogo de entrada exibido pelas linhas 12–13



(c) Diálogo de entrada exibido pelas linhas 22–23



Quando o usuário clica em **OK**, a caixa de diálogo da mensagem se fecha (desaparece da tela).

Figura 14.2 | Programa de adição que utiliza `JOptionPane` para entrada e saída. (Parte 3 de 3.)

- O método `JOptionPane` `static` **`showInputDialog`** exibe um diálogo de entrada, utilizando argumentos `String` do método como um prompt.
- O usuário digita caracteres no campo de texto, depois clica em OK ou pressiona a tecla *Enter* para enviar a `String` para o programa.
- Clicar em OK **fecha** (**oculta**) **o diálogo**.
- Pode-se inserir somente `String`s. É comum na maioria dos componentes GUI.
- Se o usuário clicar em Cancel, `retorna null`.
- Diálogos `JOptionPane` são **diálogos** — o usuário não pode interagir com o restante do aplicativo.

Observação sobre a aparência e comportamento 14.2

Em geral, o prompt em um diálogo de entrada emprega ***maiúsculas e minúsculas no estilo de frases*** *— um estilo que emprega a maiúscula inicial apenas na primeira palavra da frase a menos que a palavra seja um nome próprio (por exemplo, Jones).*

Observação sobre a aparência e comportamento 14.3

Não use excessivamente caixas de diálogo modais uma vez que elas podem reduzir a usabilidade dos aplicativos. Utilize uma caixa de diálogo modal somente quando for necessário impedir usuários de interagir com o restante de um aplicativo até que eles descartem o diálogo.

- Convertendo `String`s em valores `int`
- O método `staticparseInt` da classe `Integer` converte seu argumento `String` em um valor `int`.

Diálogos de mensagem

- O método `JOptionPane` `static` **showMessageDialog** exibe um diálogo de mensagem.
- O primeiro argumento ajuda o aplicativo a determinar onde posicionar o diálogo.
- Se for `null`, a caixa de diálogo será exibida no centro da tela.
- O segundo argumento é a mensagem a ser exibida.
- O terceiro argumento é a `String` que deve aparecer na barra de título na parte superior do diálogo.
- O quarto argumento é o tipo de diálogo de mensagem a ser exibido.

Diálogos de mensagem

- Um diálogo **JOption-Pane.PLAIN_MESSAGE** não exibe um ícone à esquerda da mensagem.
- Documentação JOptionPaneonline: java.sun.com/javase/6/docs/api/javax/swing/JOptionPane.html

Observação sobre a aparência e comportamento 14.4

Em geral, a barra de título de uma janela adota o uso de letras ***maiúsculas e minúsculas de título de livro*** *— um estilo que emprega a inicial maiúscula em cada palavra significativa no texto e não termina com pontuação (por exemplo, Uso de Letras Maiúsculas e Minúsculas no Título de um Livro).*

Tipo de diálogo de mensagem	Ícone	Descrição
`ERROR_MESSAGE`		Indica um erro ao usuário.
`INFORMATION_MESSAGE`		Indica uma mensagem informativa ao usuário.
`WARNING_MESSAGE`		Alerta o usuário de um potencial problema.
`QUESTION_MESSAGE`		Propõe uma questão ao usuário. Normalmente, esse diálogo exige uma resposta, como clicar em um botão **Yes** ou **No**.
`PLAIN_MESSAGE`	Nenhum ícone	Um diálogo que contém uma mensagem, mas nenhum ícone.

Figura 14.3 | Constantes `JOptionPane` `static` para diálogo de mensagem.

14.4 Visão geral de componentes Swing

- **Componentes GUI Swing** localizados no pacote **javax.swing.**
- A maioria são componentes Java puros.
- Escritos, manipulados e exibidos completamente no Java.
- Parte das **Java Foundation Classes (JFC)** para desenvolvimento de GUI para múltiplas plataformas.
- Tecnologias desktop JFC e Java: java.sun.com/javase/technologies/desktop/

Componente	Descrição
`JLabel`	Exibe texto não editável ou ícones.
`JTextField`	Permite ao usuário inserir dados do teclado. Também pode ser utilizado para exibir texto editável ou não editável.
`JButton`	Desencadeia um evento quando o usuário clicar nele com o mouse.
`JCheckBox`	Especifica uma opção que pode ser ou não selecionada.
`JComboBox`	Fornece uma lista drop-down de itens a partir da qual o usuário pode fazer uma seleção clicando em um item ou possivelmente digitando na caixa.
`JList`	Fornece uma lista de itens a partir da qual o usuário pode fazer uma seleção clicando em qualquer item na lista. Múltiplos elementos podem ser selecionados.
`JPanel`	Fornece uma área em que os componentes podem ser colocados e organizados. Também pode ser utilizado como uma área de desenho para imagens gráficas.

Figura 14.4 | Alguns componentes GUI básicos.

- O **Abstract Window Toolkit (AWT)** do pacote **java.awt** é outro conjunto de componentes GUI do Java.
- Quando um aplicativo Java com um AWT GUI executa em diferentes plataformas Java, os componentes GUI do aplicativo exibem diferentemente em cada plataforma.
- Juntas, a aparência e a maneira como o usuário interage com o aplicativo são conhecidas como a **aparência e comportamento** desse aplicativo.
- Os componentes GUI Swing permitem especificar uma aparência e comportamento uniforme para o aplicativo em todas as plataformas ou utilizar a aparência e comportamento personalizados de cada plataforma.

Dica de portabilidade 14.1

Os componentes Swing são implementados no Java, desse modo eles são mais portáveis e flexíveis do que os componentes Java GUI originais de pacotes `java.awt`*, que foram baseados nos componentes GUI da plataforma subjacente. Por essa razão, os componentes Swing GUI geralmente são preferidos.*

- A maioria dos componentes Swing não está vinculada a componentes GUI reais da plataforma subjacente.
- Conhecidos como **componentes leves.**
- Os componentes AWT estão vinculados à plataforma local e são chamados **componentes pesados**, porque devem contar com o **sistema de janela** da plataforma local para determinar sua funcionalidade e sua aparência e comportamento.
- Vários componentes Swing são componentes pesados.

Dica de portabilidade 14.2

A aparência e comportamento de uma GUI definida com componentes de GUI peso pesado do pacote `java.awt` *podem variar de uma plataforma para outra. Como os componentes pesados estão vinculados à GUI da plataforma local, a aparência e comportamento variam de plataforma para plataforma.*

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 14.1

Estude os atributos e comportamentos das classes na hierarquia de classe da Figura 14.5. Essas classes declaram os recursos comuns à maioria dos componentes Swing.

- A classe **Component** (pacote `java.awt`) declara muitos dos atributos e comportamentos comuns aos componentes GUI em pacotes `java.awt` e `javax.swing`.
- A maioria dos componentes GUI estende a classe `Component` direta ou indiretamente.
- Documentação on-line:
  java.sun.com/javase/6/docs/api/java/awt/Component.html

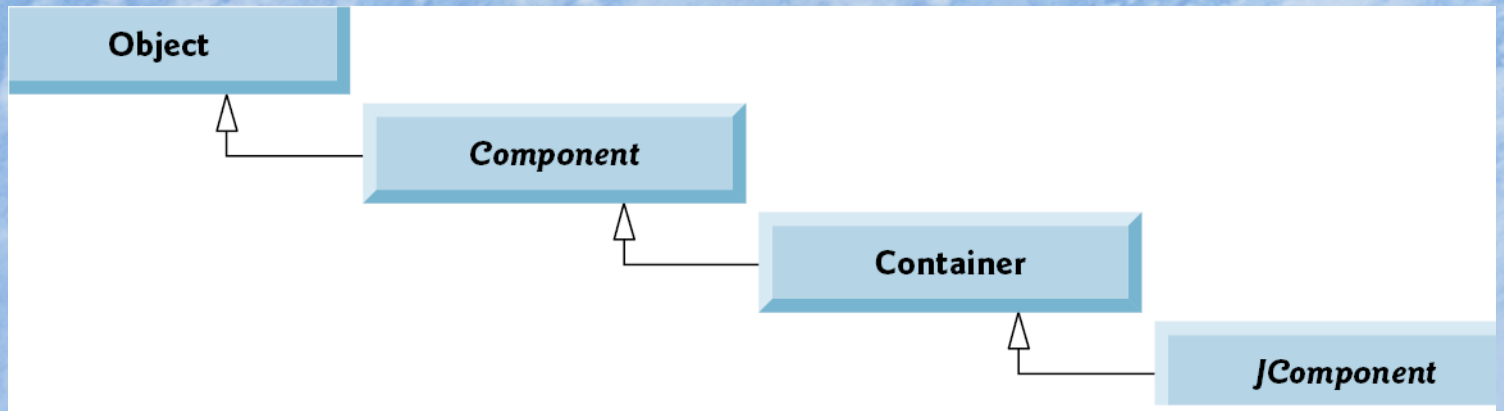


Figura 14.5 | Superclasses comuns de muitos dos componentes do Swing.

- A classe **Container** (pacote `java.awt`) é uma subclasse de `Component`.
- `Component`s são anexados a `Container`s para que possam ser organizados e exibidos na tela.
- Qualquer objeto que *é um* `Container` pode ser utilizado para organizar outros `Component`s em uma GUI.
- Como um `Container` *é um* `Component`, você pode colocar `Container`s em outros `Container`s para ajudar a organizar uma GUI.
- Documentação on-line:
- java.sun.com/javase/6/docs/api/java/awt/Container.html

- A classe **JComponent** (pacote `java.awt`) é uma subclasse de `Component`.
- `JComponent` é a superclasse de todos os componentes Swing leves, que também são `Container`s.

- Alguns recursos de componentes leves comuns suportados por `JComponent` incluem:
  - **aparência e comportamento plugável**.
  - Teclas de atalho (chamadas **mnemônicas**).
  - Capacidades comuns de tratamento de evento para componentes que iniciam as mesmas ações em um aplicativo.
  - **dicas de ferramenta**.
  - Suporte para acessibilidade.
  - Suporte para **localização**.
- Documentação on-line:
  - `http://java.sun.com/javase/6/docs/api/javax/swing/JOptionPane.html`

- A maioria das janelas que podem conter componentes GUI Swing são instâncias da classe `JFrame` ou uma subclasse de `JFrame`.
- `JFrame` é uma subclasse indireta da classe `java.awt.Window`
- Fornece os atributos básicos e os comportamentos de uma janela:
  - uma barra de título em cima.
  - botões para minimizar, maximizar e fechar a janela.
- A maioria de nossos exemplos consistirá em duas classes:
  - uma subclasse de `JFrame` que demonstra novos conceitos das GUIs.
  - uma classe de aplicativo em que `main` cria e exibe a janela principal do aplicativo.

Observação sobre a aparência e comportamento 14.5

Normalmente, o texto em um `JLabel` *emprega maiúsculas e minúsculas no estilo de frases.*


```
1  // Figura 14.6: LabelFrame.java
2  // Demonstrando a classe JLabel.
3  import java.awt.FlowLayout; // especifica como os componentes são organizados
4  import javax.swing.JFrame; // fornece recursos básicos de janela
5  import javax.swing.JLabel; // exibe texto e imagens
6  import javax.swing.SwingConstants; // constantes comuns utilizadas com Swing
7  import javax.swing.Icon; // interface utilizada para manipular imagens
8  import javax.swing.ImageIcon; // carrega imagens
9
10 public class LabelFrame extends JFrame
11 {
12    private JLabel label1; // JLabel apenas com texto
13    private JLabel label2; // JLabel construído com texto e ícone
14    private JLabel label3; // JLabel com texto e ícone adicionados
15
16    // construtor LabelFrame adiciona JLabels a JFrame
17    public LabelFrame()
18    {
19       super( "Testing JLabel" );
20       setLayout( new FlowLayout() ); // configura o layout de frame
21
```

GUIs personalizadas são construídas normalmente em classes que estendem o `JFrame`

Configura o texto da barra de título do `JFrame` na `String` especificada

Figura 14.6 | `JLabel`s com texto e ícones. (Parte 1 de 2.)


```
22          // Construtor JLabel com um argumento de string
23          label1 = new JLabel( "Label with text" );
24          label1.setToolTipText( "This is label1" );
25          add( label1 ); // adiciona o label1 ao JFrame
26
27          // construtor JLabel com string, Icon e argumentos de alinhamento
28          Icon bug = new ImageIcon( getClass().getResource(  "bug1.png" ) );
29          label2 = new JLabel( "Label with text and icon", bug,
30             SwingConstants.LEFT );
31          label2.setToolTipText( "This is label2" );
32          add( label2 ); // adiciona label2 ao JFrame
33
34          label3 = new JLabel(); // Construtor JLabel sem argumentos
35          label3.setText( "Label with icon and text at bottom" );
36          label3.setIcon( bug ); // adiciona o ícone ao JLabel
37          label3.setHorizontalTextPosition( SwingConstants.CENTER );
38          label3.setVerticalTextPosition( SwingConstants.BOTTOM );
39          label3.setToolTipText( "This is label3" );
40          add( label3 ); // adiciona label3 ao JFrame
41       } // fim do construtor LabelFrame
42    } // fim da classe LabelFrame
```

Cria um `JLabel` com o texto especificado e então configura sua dica de ferramenta

Carrega um ícone da mesma localização da classe `LabelFrame`, então cria um `JLabel` com o texto e um ícone e configura o texto da dica de ferramenta do `JLabel`.

Cria um `JLabel` vazio e então utiliza os métodos *set* para alterar suas características.

Figura 14.6 | `JLabel`s com texto e ícones. (Parte 2 de 2.)

```
1  // Figura 14.7: LabelTest.java
2  // Testando LabelFrame.
3  import javax.swing.JFrame;
4
5  public class LabelTest
6  {
7     public static void main( String[] args )
8     {
9        LabelFrame labelFrame = new LabelFrame(); // cria LabelFrame
10       labelFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       labelFrame.setSize( 260, 180 ); // configura o tamanho do frame
12       labelFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe LabelTest
```

O programa deve terminar quando o usuário clicar no botão close da janela.

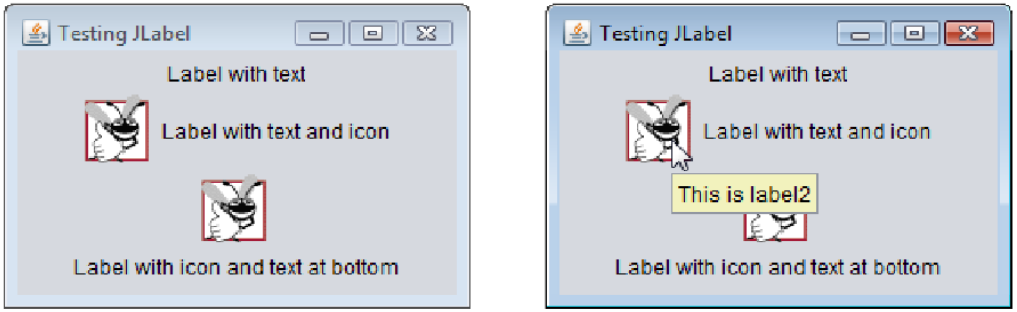


Figura 14.7 | Classe de teste para `LabelFrame`.

- Em uma GUI extensa:
  - Dificulta identificar o objetivo de cada componente.
  - Fornece um texto que declara a finalidade de cada componente.
- Esse texto é conhecido como **rótulo** e é criado com a classe **JLabel** — uma subclasse de `JComponent`.
  - Exibe texto somente de leitura, uma imagem ou tanto texto como imagem.

COMO PROGRAMAR

8ª edição

- O construtor de `JFrame` utiliza seu argumento `String` como o texto na barra de título da janela.
- Deve anexar cada componente GUI a um contêiner, como um `JFrame`.
- Em geral, você deve decidir onde posicionar cada componente GUI.
- Isso é conhecido como especificar o layout dos componentes GUI.
- O Java fornece vários **gerenciadores de layout** que podem ajudá-lo a posicionar os componentes.

- Muitos IDEs fornecem ferramentas de design de GUI nas quais você pode especificar o tamanho e localização exatos de um componente
- O IDE gera o código GUI para você.
- Simplifica bastante a criação de GUIs.
- Para assegurar que os exemplos desse livro podem ser utilizados com qualquer IDE, não utilizamos um IDE para criar o código GUI.
- Utilizamos gerenciadores de layout Java nos nossos exemplos de GUI.

- **FlowLayout**
- Os componentes GUI são colocados em um contêiner da esquerda para a direita na ordem em que são adicionados ao contêiner.
- Quando não houver mais espaço para ajustar componentes da esquerda para a direita, os componentes continuam a aparecer da esquerda para direita na próxima linha.
- Se o contêiner for redimensionado, um `FlowLayout` recorre os componentes para acomodar a nova largura do contêiner, possivelmente com menos ou mais linhas de componentes GUI.
- O método **setLayout** é herdado da classe `Container`. O argumento para o método deve ser um objeto de uma classe que implementa a interface `LayoutManager` (por exemplo, `FlowLayout`).

Erro comum de programação 14.1

Se você não adicionar explicitamente um componente GUI a um contêiner, o componente GUI não será exibido quando o contêiner aparecer na tela.

Observação sobre a aparência e comportamento 14.6

Utilize as dicas de ferramenta para adicionar texto descritivo aos componentes GUI. Esse texto ajuda o usuário a determinar o propósito do componente GUI na interface com o usuário.

- O construtor `JLabel` pode receber uma `String` especificando o texto do rótulo.
- O método **setToolTipText** (herdado por `JLabel` de `JComponent`) especifica a dica de ferramenta que é exibida quando o usuário posiciona o cursor do mouse sobre um `JComponent` (como um `JLabel`).
- Você anexa um componente a um contêiner utilizando o método **add**, que é herdado indiretamente da classe `Container`.

- Os ícones aprimoram a aparência e comportamento de um aplicativo e também são comumente utilizados para indicar funcionalidade.
- Normalmente, um ícone é especificado com um argumento **Icon** para um construtor ou para o método **setIcon** do componente.
- Um `Icon` é um objeto de qualquer classe que implementa a interface `Icon` (pacote `javax.swing`).
- **ImageIcon** (pacote `javax.swing`) suporta vários formatos de imagem, incluindo **Graphics Interchange Format (GIF), Portable Network Graphics (PNG) e Joint Photographic Experts Group (JPEG).**

- `getClass().getResource("bug1.png")`
- Invoca o método **getClass** (herdado indiretamente da classe `Object`) para recuperar uma referência para o objeto `Class` que representa a declaração de classe `LabelFrame`.
- Em seguida, invoca o método `Class` **getResource**, que retorna a localização da imagem como um URL.
- O construtor `ImageIcon` utiliza o URL para localizar a imagem e, então, carrega essa imagem na memória.
- O carregador de classe sabe onde está cada classe que ele carrega no disco. O método `getResource` utiliza o carregador de classe do objeto `Class` para determinar a localização de um recurso, como um arquivo de imagem.

- Um `JLabel` pode exibir um `Icon`.
- O construtor `JLabel` pode receber texto e um `Icon`.
- O último argumento de construtor indica a justificação dos conteúdos do rótulo.
- A interface **SwingConstants** (pacote `javax.swing`) declara um conjunto de constantes de inteiro comuns (como `SwingConstants.LEFT`) que são utilizadas com muitos componentes Swing.
- Por padrão, o texto aparece à direita da imagem quando um rótulo contém tanto texto como imagem.
- Os alinhamentos horizontal e vertical de um `JLabel` podem ser configurados com os métodos **setHorizontalAlignment** e **setVerticalAlignment**, respectivamente.

Constante	Descrição
Constantes de posição horizontal	
`SwingConstants.LEFT`	Coloca o texto à esquerda.
`SwingConstants.CENTER`	Coloca o texto no centro.
`SwingConstants.RIGHT`	Coloca o texto à direita.
Constantes de posição vertical	
`SwingConstants.TOP`	Coloca o texto na parte superior.
`SwingConstants.CENTER`	Coloca o texto no centro.
`SwingConstants.BOTTOM`	Coloca o texto na parte inferior.

Figura 14.8 | Posicionando constantes.

- A classe `JLabel` fornece métodos para alterar a aparência de um rótulo depois de ele ter sido instanciado.
- O método **setText** configura o texto exibido no rótulo.
- O método **getText** recupera o texto atual exibido em um rótulo.
- O método **setIcon** especifica o `Icon` a ser exibido em um rótulo.
- O método **getIcon** recupera o `Icon` atual exibido em um rótulo.
- Os métodos **setHorizontalTextPosition** e **setVerticalTextPosition** especificam a posição do texto no rótulo.

- Por padrão, fechar uma janela simplesmente a oculta.
- Chamar o método **setDefaultCloseOperation** (herdado da classe `JFrame`) com o argumento **JFrame.EXIT_ON_CLOSE** indica que o programa deve terminar quando a janela for fechada pelo usuário.
- O método **setSize** especifica a largura e altura da janela em pixels.
- O método **setVisible** com o argumento `true` exibe a janela na tela.

COMO PROGRAMAR

8ª edição

14.6 Campos de texto e uma introdução ao tratamento de eventos com classes aninhadas

- As GUIs são **baseadas em evento**.
- Quando o usuário interage com um componente GUI, a interação — conhecida como um **evento** — guia o programa para realizar uma tarefa.
- O código que realiza uma tarefa em resposta a um evento é chamado **handler de evento**, e o processo total de responder a eventos é conhecido como **tratamento de evento.**

- **JTextFields e JPasswordFields** (pacote `javax.swing`).
- `JTextField` estende a classe **JTextComponent** (pacote `javax.swing.text`), que fornece muitos recursos comuns aos componentes baseados em texto do Swing.
- A classe `JPasswordField` estende `JTextField` e adiciona métodos que são específicos ao processamento de senhas.
- `JPasswordField` mostra que os caracteres estão sendo digitados à medida que o usuário os insere, mas oculta os caracteres reais com **caracteres eco.**


```
1  // Figura 14.9: TextFieldFrame java
2  // Demonstrando a classe JTextField.
3  import java.awt.FlowLayout;
4  import java.awt.event.ActionListener;
5  import java.awt.event.ActionEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JTextField;
8  import javax.swing.JPasswordField;
9  import javax.swing.JOptionPane;
10
11 public class TextFieldFrame extends JFrame
12 {
13    private JTextField textField1; // campo de texto com tamanho configurado
14    private JTextField textField2; // campo de texto construído com texto
15    private JTextField textField3; // campo de texto com texto e tamanho
16    private JPasswordField passwordField; // campo de senha com texto
17
18    // construtor TextFieldFrame adiciona JTextFields a JFrame
19    public TextFieldFrame()
20    {
21       super( "Testing JTextField and JPasswordField" );
22       setLayout( new FlowLayout() ); // configura o layout de frame
23
```

Figura 14.9 | JTextFields e JPasswordFields. (Parte 1 de 4.)


```
24          // constrói textfield com 10 colunas
25          textField1 = new JTextField( 10 );
26          add( textField1 ); // adiciona textField1 ao JFrame
27
28          // constrói campo de texto com texto padrão
29          textField2 = new JTextField( "Enter text here" );
30          add( textField2 ); // adiciona textField2 ao JFrame
31
32          // constrói textfield com o texto padrão e 21 colunas
33          textField3 = new JTextField( "Uneditable text field", 21 );
34          textField3.setEditable( false ); // desativa a edição
35          add( textField3 ); // adiciona textField3 ao JFrame
36
37          // constrói passwordfield com o texto padrão
38          passwordField = new JPasswordField( "Hidden text" );
39          add( passwordField ); // adiciona passwordField ao JFrame
40
41          // handlers de evento registradores
42          TextFieldHandler handler = new TextFieldHandler();
43          textField1.addActionListener( handler );
44          textField2.addActionListener( handler );
45          textField3.addActionListener( handler );
46          passwordField.addActionListener( handler );
47       } // fim do construtor TextFieldFrame
```

A largura de um `JTextField` é baseada na fonte atual do componente a menos que o gerenciador de layout sobrescreva esse tamanho.

A largura de um `JTextField` é baseada no texto padrão a menos que o gerenciador de layout sobrescreva esse tamanho.

Largura baseada no segundo argumento a menos que o gerenciador de layout sobrescreva esse tamanho.

O texto nesse componente será oculto por asteriscos (*) por padrão.

A classe interna `TextFieldHandler` implementa a interface `ActionListener`, para que ela possa responder a eventos `JTextField`. As linhas 43-46 registram o objeto handler para responder aos eventos de cada componente.

Figura 14.9 | `JTextFields` e `JPasswordFields`. (Parte 2 de 4.)

```
48
49       // classe interna private para tratamento de evento
50       private class TextFieldHandler implements ActionListener
51       {
52          // processa eventos de campo de texto
53          public void actionPerformed( ActionEvent event )
54          {
55             String string = ""; // declara string a ser exibida
56
57             // usuário pressionou Enter no JTextField textField1
58             if (event.getSource() == textField1 )
59                string = String.format( "textField1: %s",
60                   event.getActionCommand() );
61
62             // usuário pressionou Enter no JTextField textField2
63             else if (event.getSource() == textField2 )
64                string = String.format( "textField2: %s",
65                   event.getActionCommand() );
66
67             // usuário pressionou Enter no JTextField textField3
68             else if (event.getSource() == textField3 )
69                string = String.format( "textField3: %s",
70                   event.getActionCommand() );
71
```

Um `TextFieldHandler` é um `ActionListener`.

Chamado quando o usuário pressiona *Enter* em um `JTextField` ou `JPasswordField`.

O `getSource` especifica o componente com o qual o usuário interagiu

Obtém o texto que o usuário inseriu no textfield.

Figura 14.9 | `JTextFields` e `JPasswordFields`. (Parte 3 de 4.)


```
72              // usuário pressionou Enter no JTextField passwordField
73              else if (event.getSource() == passwordField )
74                 string = String.format( "passwordField: %s",
75                    event.getActionCommand() );
76
77              // exibe o conteúdo de JTextField
78              JOptionPane.showMessageDialog( null, string );
79           } // fim do método actionPerformed
80        } // fim da classe TextFieldHandler interna private
81     } // fim da classe TextFieldFrame
```

Figura 14.9 | JTextFields e JPasswordFields. (Parte 4 de 4.)

```
1  // Figura 14.10: TextFieldTest.java
2  // Testando TextFieldFrame.
3  import javax.swing.JFrame;
4
5  public class TextFieldTest
6  {
7     public static void main( String[] args )
8     {
9        TextFieldFrame textFieldFrame = new TextFieldFrame();
10       textFieldFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       textFieldFrame.setSize( 350, 100 ); // configura o tamanho do frame
12       textFieldFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe TextFieldTest
```

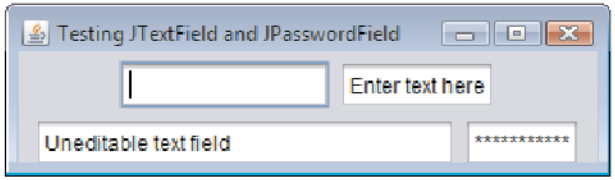


Figura 14.10 | Classe de teste para `TextFieldFrame`. (Parte 1 de 3.)

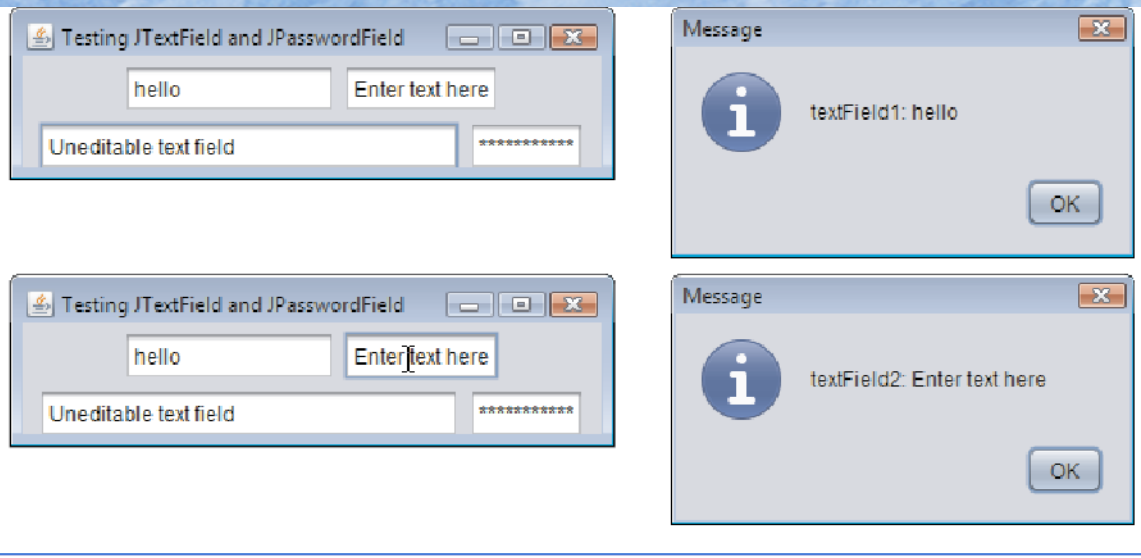


Figura 14.10 | Classe de teste para `TextFieldFrame`. (Parte 2 de 3.)

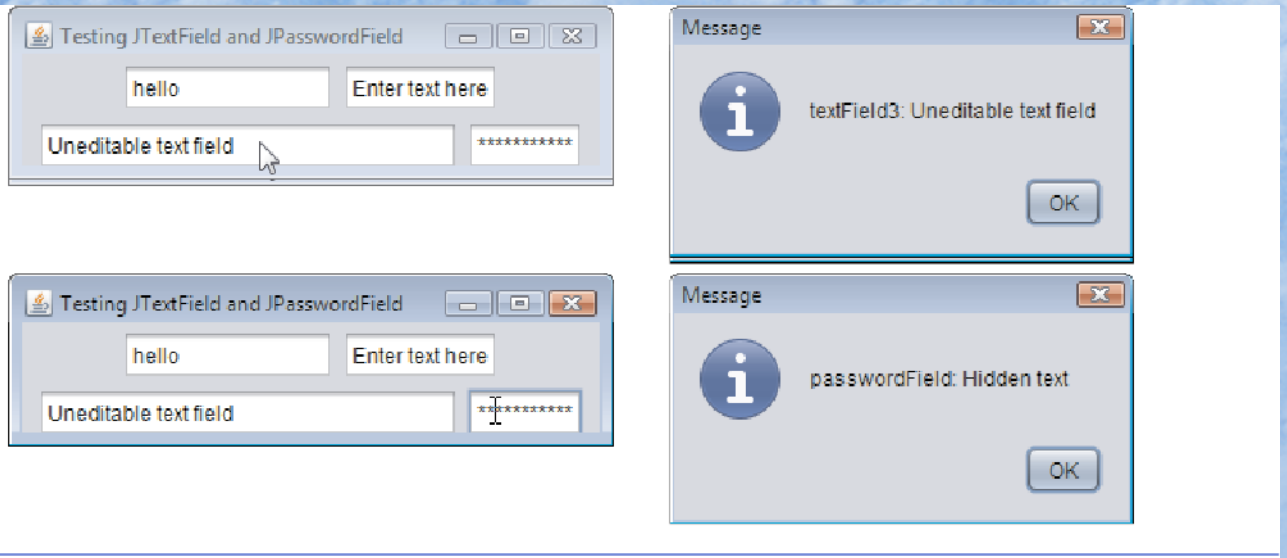


Figura 14.10 | Classe de teste para `TextFieldFrame`. (Parte 3 de 3.)

- Quando o usuário digita dados em um `JTextField` ou `JPasswordField` e, então, pressiona *Enter*, um evento ocorre.
- Você pode digitar somente no campo de texto que estiver "em **foco**".
- Um componente recebe o foco quando o usuário clica no componente.

- Antes que um aplicativo possa responder a um evento de um componente GUI particular, você deve realizar vários passos de codificação:
  - Crie uma classe que represente o handler de evento.
  - Implemente uma interface apropriada, conhecida como **interface ouvinte de eventos**, na classe do *Passo 1*.
  - Indique que um objeto da classe dos Passos 1 e 2 deve ser notificado quando o evento ocorrer. Isso é conhecido como **registrar o handler de evento.**

- Todas as classes discutidas até agora foram chamadas de **classes de primeiro nível** — isto é, elas não foram declaradas dentro de outra classe.
- O Java permite declarar classes dentro de outras classes — são chamadas **classes aninhadas.**
- Podem ser `static` ou não `static`.
- As classes não `static` aninhadas são chamadas `classes internas` e são frequentemente utilizadas para implementar handlers de evento.

Observação de engenharia de software 14.2

Uma classe interna tem permissão de acessar diretamente todas as variáveis e métodos de sua classe superior.

- Antes que um objeto de uma classe interna possa ser criado, deve primeiro haver um objeto da classe de primeiro nível que contém a classe interna.
- Isso é necessário porque um objeto de classe interna tem implicitamente uma referência a um objeto de sua classe de primeiro nível.
- Há também um relacionamento especial entre esses objetos — o objeto da classe interna tem permissão de acessar diretamente todas as variáveis e métodos da classe externa.
- Uma classe aninhada que é `static` não exige um objeto de sua classe de primeiro nível e não tem implicitamente uma referência a um objeto da classe de primeiro nível.

- Classes internas podem ser declaradas `public`, `protected` ou `private`.
- Visto que as rotinas de tratamento de evento tendem a ser específicas do aplicativo em que são definidas, muitas vezes elas são implementadas como classes internas `private`.

- Os componentes GUI podem gerar muitos eventos em resposta a interações de usuário.
- Cada evento é representado por uma classe e pode ser processado apenas pelo tipo de handler de evento apropriado.
- Normalmente, os eventos suportados de um componente são descritos na documentação da Java API para a classe desse componente e suas superclasses.

- Quando o usuário pressiona *Enter* em um `JTextField` ou `JPasswordField`, um **ActionEvent** (pacote `java.awt.event`) ocorre.
- É processado por um objeto que implementa a interface **ActionListener** (pacote `java.awt.event`).
- Para tratar `ActionEvent`s, uma classe deve implementar a interface `ActionListener` e declarar o método `actionPerformed`.
- Esse método especifica as tarefas a serem realizadas quando ocorrer um `ActionEvent`.

Observação de engenharia de software 14.3

O ouvinte de evento para um evento deve implementar a interface apropriada de ouvinte de evento.

Erro comum de programação 14.2

Esquecer de registrar um objeto tratador de evento para um tipo de evento de um componente GUI particular faz com que o tipo seja ignorado.

- Deve-se registrar um objeto como o handler de evento para cada campo de texto.
- **addActionListener** registra um objeto `ActionListener` para tratar `ActionEvent`s.
- Depois de um handler de evento é registrado o objeto que **ouve os eventos.**

- O componente GUI com o qual o usuário interage é a **origem de evento**.
- O método `ActionEvent` `getSource` (herdado da classe `EventObject`) retorna uma referência para a origem de evento.
- O método `ActionEvent` **getActionCommand** obtém o texto que o usuário digitou no campo de texto que gerou o evento.
- O método `JPasswordField` **getPassword** retorna os caracteres da senha como um array de tipo char.

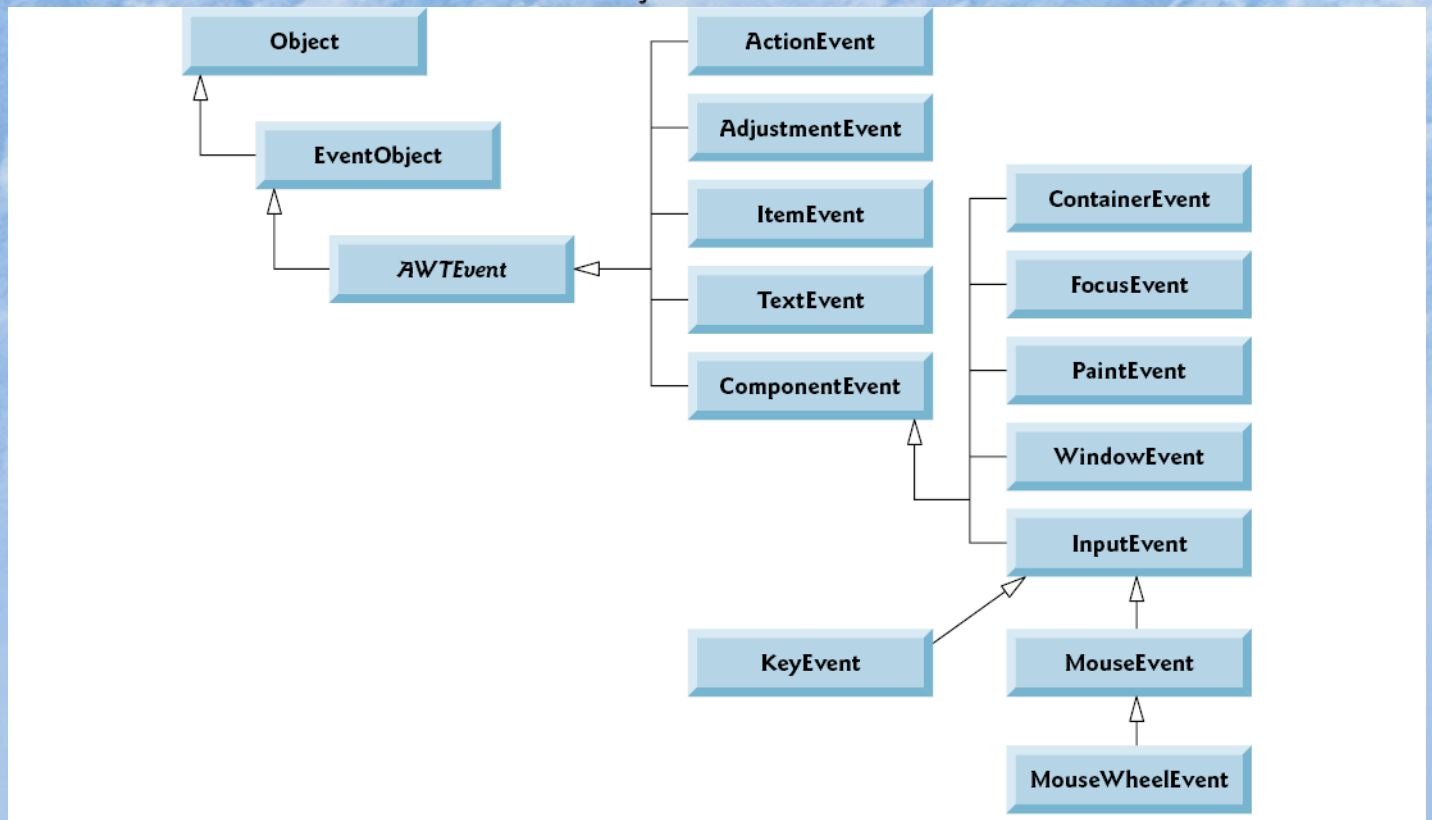


Figura 14.11 | Algumas classes de evento do pacote `java.awt.event`.

14.7 Tipos comuns de eventos GUI e interfaces ouvintes

- A Figura 14.11 ilustra uma hierarquia que contém muitas classes de evento do pacote **java.awt.event.**
- Utilizados em componentes AWT e Swing.
- Tipos adicionais de evento que são específicos dos componentes GUI Swing são declarados no pacote **javax.swing.event.**

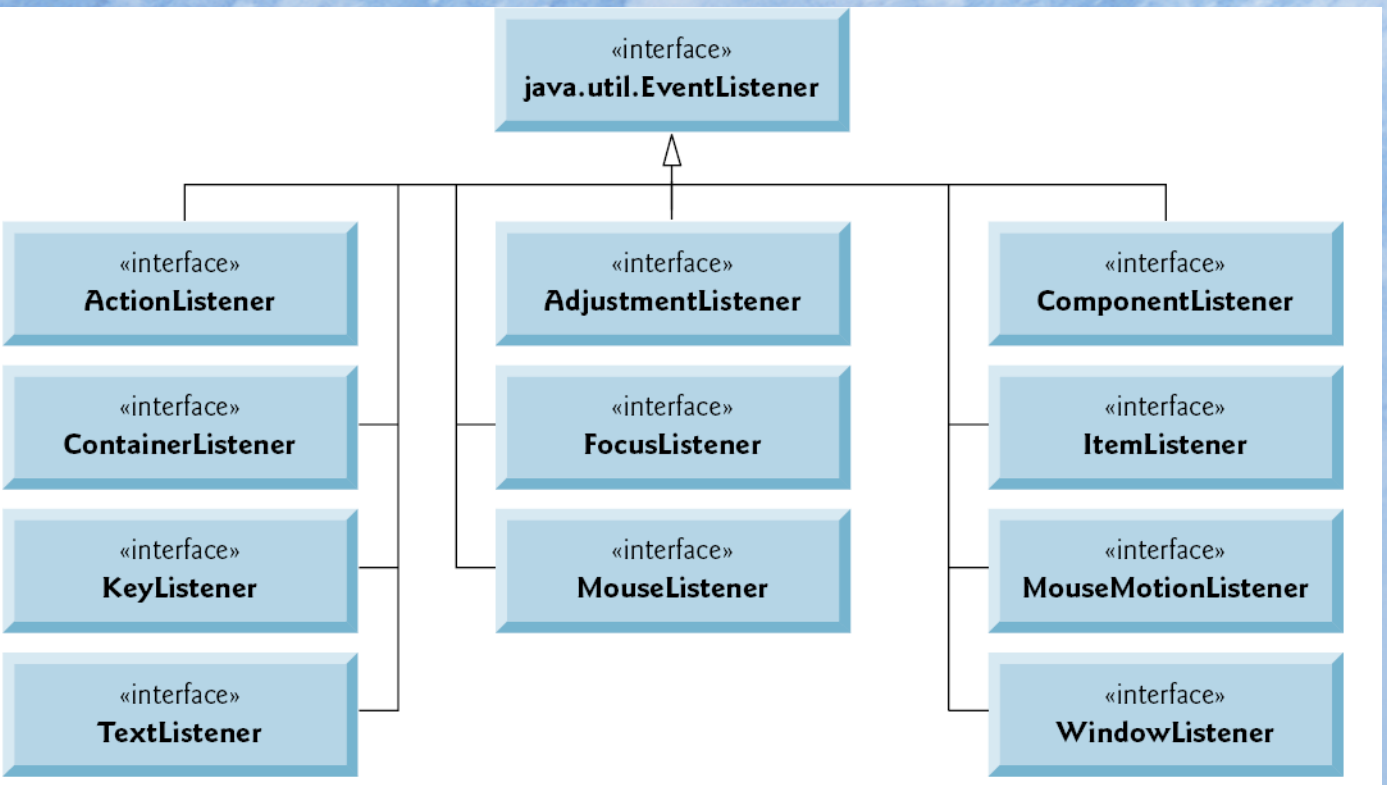


Figura 14.12 | Algumas interfaces listener de eventos comuns do pacote `java.awt.event`.

- **Modelo de evento de delegação** — o processamento de um evento é delegado a um objeto (o ouvinte de eventos) do aplicativo.
- Para cada tipo de objeto de evento, há em geral uma interface ouvinte de eventos correspondente.
- Muitos tipos de ouvinte de evento são comuns aos componentes Swing e AWT.
- Tipos como esses são declarados no pacote `java.awt.event` e alguns deles são mostrados na Figura 14.12.
- Tipos adicionais de ouvinte de evento que são específicos de componentes Swing são declarados no pacote `javax.swing.event`.

- Cada interface ouvinte de eventos especifica um ou mais métodos de tratamento de evento que devem ser declarados na classe que implementa a interface.
- Quando um evento ocorre, o componente GUI com o qual o usuário interagiu notifica seus ouvintes registrados chamando o método de tratamento de evento apropriado de cada ouvinte.

14.8 Como o tratamento de evento funciona

- Como funciona o mecanismo de tratamento de eventos:
- Todo `JComponent` tem uma variável `listenerList` que referencia um **EventListenerList** (pacote `javax.swing.event`).
- Mantém referências a ouvintes registrados na `listenerList`.
- Quando um ouvinte é registrado, uma nova entrada é colocada na `listenerList` do componente.
- Toda entrada também inclui um tipo de ouvinte.

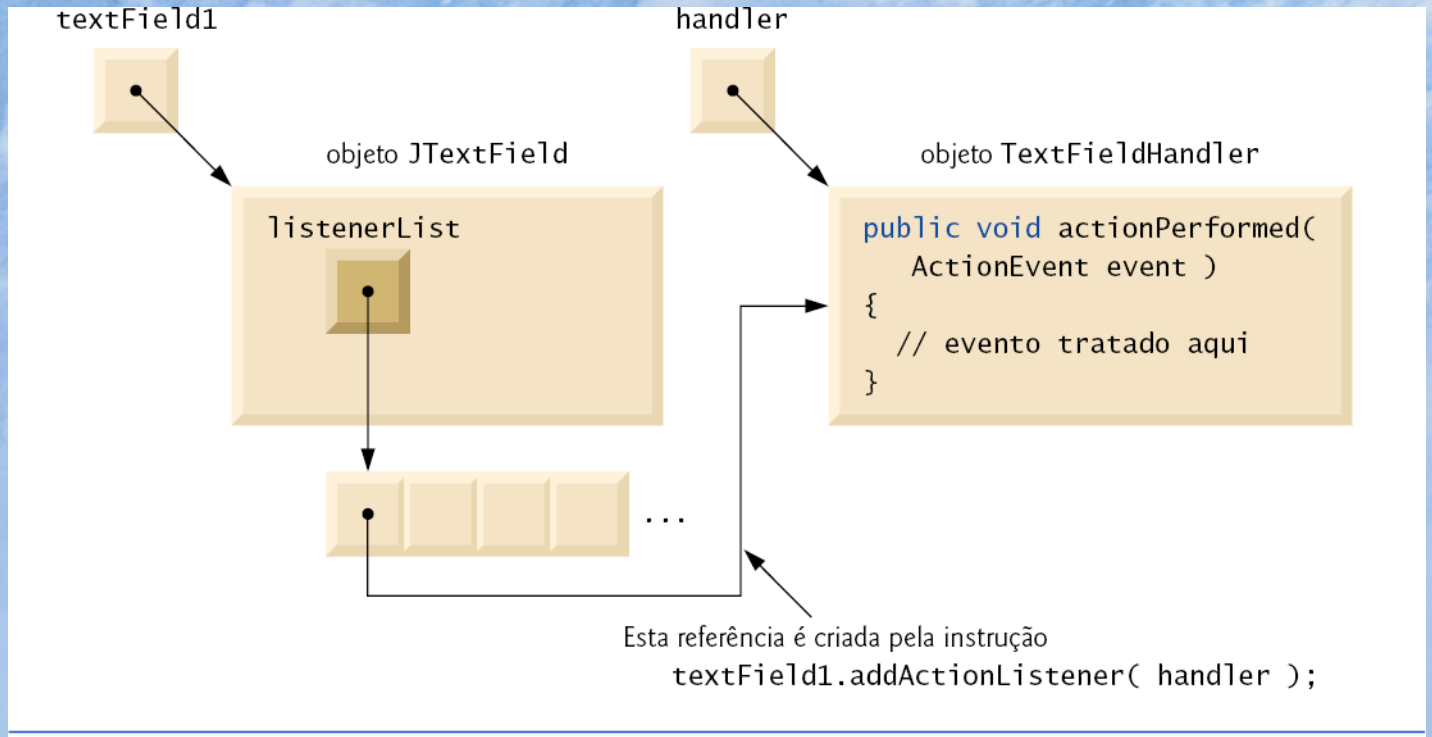


Figura 14.13 | Registro de evento para o `JTextField textField1`.

- Como o componente GUI sabe chamar `actionPerformed` em vez de outro método?
- Todo componente GUI suporta vários tipos de evento, inclusive **eventos de mouse, eventos de teclado** e outros.
- Quando um evento ocorre, o evento é **despachado** apenas para os ouvintes de evento do tipo apropriado.
- Despacho (*dispatching*) é simplesmente o processo pelo qual o componente GUI chama um método de tratamento de evento em cada um de seus ouvintes que são registrados para o tipo de evento que ocorreu.

- Cada tipo de evento tem uma ou mais interfaces ouvinte de eventos correspondentes.
- `ActionEvent`s são tratados por `ActionListener`s.
- **MouseEvent**s são tratados por **MouseListener**s e **MouseMotionListener**s
- **KeyEvent**s são tratados por **KeyListener**s.
- Quando ocorre um evento, o componente GUI recebe (do JVM) um único **ID de evento** para especificar o tipo de evento.
- O componente GUI utiliza o ID de evento para decidir o tipo de ouvinte para o evento que deve ser despachado e decidir qual método chamar em cada objeto ouvinte.

- Para um `ActionEvent`, o evento é despachado para o método `actionPerformed` de todos `ActionListener`'s registrados.
- Para um `Mouse-Event`, o evento é despachado para cada `MouseListener` ou `MouseMotionListener` registrado, dependendo do evento de mouse que ocorre.
- O ID de evento de `MouseEvent` determina quais dos vários métodos de tratamento de evento de mouse serão chamados.

14.9 JButton

- Um **botão** é um componente em que o usuário clica para acionar uma ação específica.
- Vários tipos de botões
  - **botões de comando**
  - **caixas de seleção**
  - **botões de alternação**
  - **botões de opção**
- Os tipos de botão são subclasses de **AbstractButton** (pacote `javax.swing`), que declara os recursos comuns de botões Swing.

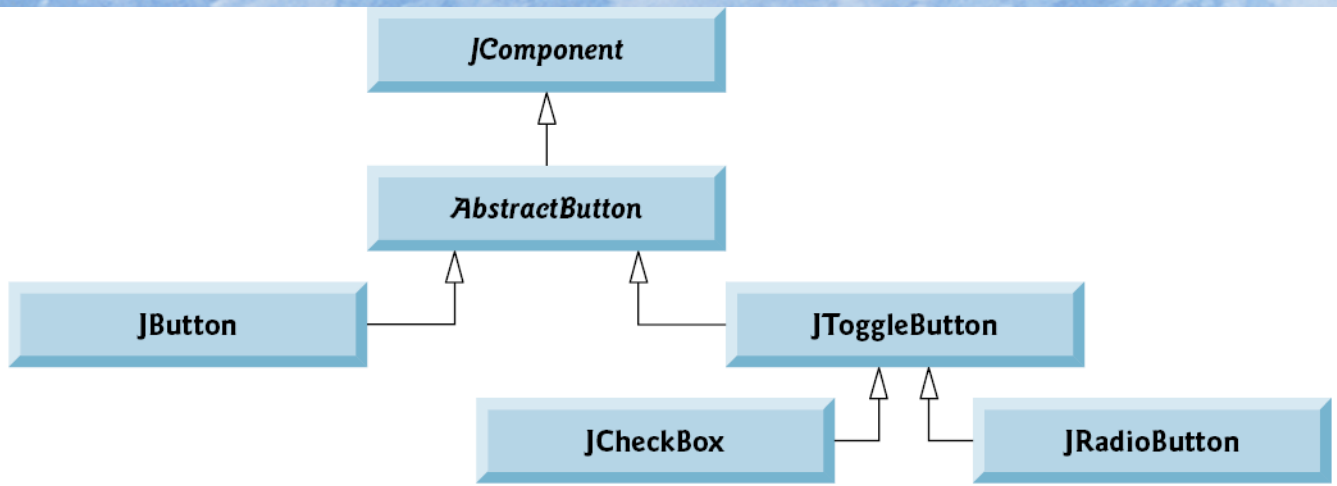


Figura 14.14 | Hierarquia do botão Swing.

Observação sobre a aparência e comportamento 14.7

Em geral, os botões utilizam letras maiúsculas e minúsculas no estilo de título de livro.

- Um botão de comando gera um `ActionEvent` quando o usuário clica nele.
- Os botões de comando são criados com a classe **JButton.**
- O texto na face de um `JButton` é chamado **rótulo de botão.**

Observação sobre a aparência e comportamento 14.8

Ter mais de um `JButton` *com o mesmo rótulo torna os* `JButtons` *ambíguos para o usuário. Forneça um rótulo único para cada botão.*

```
1  // Figura 14.15: ButtonFrame.java
2  // Criando JButtons.
3  import java.awt.FlowLayout;
4  import java.awt.event.ActionListener;
5  import java.awt.event.ActionEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JButton;
8  import javax.swing.Icon;
9  import javax.swing.ImageIcon;
10 import javax.swing.JOptionPane;
11
12 public class ButtonFrame extends JFrame
13 {
14    private JButton plainJButton; // botão apenas com texto
15    private JButton fancyJButton; // botão com ícones
16
17    // ButtonFrame adiciona JButtons ao JFrame
18    public ButtonFrame()
19    {
20       super( "Testing Buttons" );
21       setLayout( new FlowLayout() ); // configura o layout de frame
22
23       plainJButton = new JButton( "Plain Button" ); // botão com texto
24       add( plainJButton ); // adiciona plainJButton ao JFrame
```

Cria um `JButton` com o texto especificado como seu rótulo.

Figura 14.15 | Botões de comando e eventos de ação. (Parte 1 de 2.)

```
25
26          Icon bug1 = new ImageIcon( getClass().getResource( "bug1.gif" ) );
27          Icon bug2 = new ImageIcon( getClass().getResource( "bug2.gif" ) );
28          fancyJButton = new JButton( "Fancy Button", bug1 ); // configura imagem
29          fancyJButton.setRolloverIcon( bug2 ); // configura imagem de rollover
30          add( fancyJButton ); // adiciona fancyJButton ao JFrame
31
32          // cria novo ButtonHandler para tratamento de evento de botão
33          ButtonHandler handler = new ButtonHandler();
34          fancyJButton.addActionListener( handler );
35          plainJButton.addActionListener( handler );
36       } // fim do construtor ButtonFrame
37
38       // classe interna para tratamento de evento de botão
39       private class ButtonHandler implements ActionListener
40       {
41          // trata evento de botão
42          public void actionPerformed( ActionEvent event )
43          {
44             JOptionPane.showMessageDialog(ButtonFrame.this, String.format(
45                "You pressed: %s", event.getActionCommand() ) );
46          } // fim do método actionPerformed
47       } // fim da classe ButtonHandler private interna
48    } // fim da classe ButtonFrame
```

Carrega duas imagens da mesma localização da classe `ButtonFrame`, então utiliza a primeira como o ícone padrão no `JButton` e a segunda como o ícone de rolagem.

Cria o objeto da classe interna `ButtonHandler` e o registra para que trate os `ActionEvent`s de ambos os `JButton`s.

Os objetos dessa classe podem responder a `ActionEvent`s.

`ButtonFrame.this` é a notação especial que permite à classe interna acessar a referência `this` da classe de primeiro nível `ButtonFrame`.

Figura 14.15 | Botões de comando e eventos de ação. (Parte 2 de 2.)


```
1  // Figura 14.16: ButtonTest.java
2  // Testando ButtonFrame.
3  import javax.swing.JFrame;
4
5  public class ButtonTest
6  {
7     public static void main( String[] args )
8     {
9        ButtonFrame buttonFrame = new ButtonFrame(); // cria ButtonFrame
10       buttonFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       buttonFrame.setSize( 275, 110 ); // configura o tamanho do frame
12       buttonFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe ButtonTest
```

Figura 14.16 | Classe de teste para `ButtonFrame`. (Parte 1 de 2.)

Figura 14.16 | Classe de teste para `ButtonFrame`. (Parte 2 de 2.)

- Um `JButton` pode exibir um `Icon`.
- Um `JButton` também pode ter um **Icon rollover** exibido quando o usuário posiciona o mouse sobre o `JButton`.
- O ícone do `JButton` altera-se quando o mouse move-se para dentro e para fora da área do `JButton` na tela.
- O método `AbstractButton` **setRolloverIcon** especifica a imagem exibida no `JButton` quando o usuário posiciona o mouse sobre ele.

Observação sobre a aparência e comportamento 14.9

Como a classe `AbstractButton` *suporta exibição de texto e imagens em um botão, todas as subclasses de* `AbstractButton` *também suportam exibição de texto e imagens.*

Observação sobre a aparência e comportamento 14.10

Utilizar ícones rollover para `JButton`*s fornece aos usuários um feedback visual que indica que, quando eles clicam no mouse enquanto o cursor está posicionado sobre* `JButton`*, uma ação ocorrerá.*

- `JButton`s, assim como `JTextField`s, geram `ActionEvent`s que podem ser processados por qualquer objeto `ActionListener`.

Observação de engenharia de software 14.4

Quando utilizada em uma classe interna, a palavra-chave `this` *referencia o objeto de classe interna atual sendo manipulado. Um método de classe interna pode utilizar* `this` *do seu objeto de classe externa precedendo* `this` *com o nome de classe externa e um ponto, como em* `ButtonFrame.this`.

14.10 Botões que mantêm o estado

- Três tipos de **botões de estado** — **JToggleButton, JCheckBox** e **JRadioButton** — que têm valores ativados/desativados ou verdadeiro/falso.
- As classes `JCheckBox` e `JRadioButton` são subclasses de `JToggleButton`.
- `JRadioButton`s são agrupados e mutuamente exclusivos — somente um no grupo pode ser selecionado por vez.

14.10.1 JCheckbox

- O método `JTextField` **setFont** (herdado por `JTextField` indiretamente da classe `Component`) configura a fonte do `JTextField` como uma nova **Font** (pacote `java.awt`).
- A `String` passada para o construtor `JCheckBox` é o **rótulo da caixa de seleção** que aparece à direita da `JCheckBox` por padrão.
- Quando o usuário clica em uma `JCheckBox`, um **ItemEvent** ocorre.
- Tratado por um objeto **ItemListener**, que deve implementar o método **itemStateChanged.**
- Um `ItemListener` é registrado com o método **addItemListener.**
- O método `JCheckBox` **isSelected** retorna `true` se uma `JCheckBox` for selecionada.


```
1  // Figura 14.17: CheckBoxFrame.java
2  // Criando botões JCheckBox.
3  import java.awt.FlowLayout;
4  import java.awt.Font;
5  import java.awt.event.ItemListener;
6  import java.awt.event.ItemEvent;
7  import javax.swing.JFrame;
8  import javax.swing.JTextField;
9  import javax.swing.JCheckBox;
10
11 public class CheckBoxFrame extends JFrame
12 {
13    private JTextField textField; // exibe o texto na alteração de fontes
14    private JCheckBox boldJCheckBox; // para selecionar/remover estilo negrito
15    private JCheckBox italicJCheckBox; // para selecionar/remover itálico
16
17    // construtor CheckBoxFrame adiciona JCheckBoxes ao JFrame
18    public CheckBoxFrame()
19    {
20       super( "JCheckBox Test" );
21       setLayout( new FlowLayout() ); // configura o layout de frame
22
23       // configura JTextField e sua fonte
24       textField = new JTextField( "Watch the font style change", 20 );
```

Figura 14.17 | Botões JCheckBox e eventos de item. (Parte 1 de 3.)


```
25          textField.setFont( new Font( "Serif", Font.PLAIN, 14 ) );
26          add( textField ); // adiciona textField ao JFrame
27
28          boldJCheckBox = new JCheckBox( "Bold" ); // cria caixa de seleção para negrito
29          italicJCheckBox = new JCheckBox( "Italic" ); // cria itálico
30          add( boldJCheckBox ); // adiciona caixa de seleção de estilo negrito ao JFrame
31          add( italicJCheckBox ); // adiciona caixa de seleção de itálico ao JFrame
32
33          // listeners registradores para JCheckBoxes
34          CheckBoxHandler handler = new CheckBoxHandler();
35          boldJCheckBox.addItemListener( handler );
36          italicJCheckBox.addItemListener( handler );
37       } // fim do construtor CheckBoxFrame
38
39       // classe interna private para tratamento de evento ItemListener
40       private class CheckBoxHandler implements ItemListener
41       {
42          // responde aos eventos de caixa de seleção
43          public void itemStateChanged( ItemEvent event )
44          {
45             Font font = null; // armazena a nova Font
46
```

setFont pode ser utilizado para alterar a fonte de qualquer componente.

Cria e registra o handler de evento para ambos os JCheckBoxes.

Um objeto dessa classe pode responder a ItemEvents.

Figura 14.17 | Botões JCheckBox e eventos de item. (Parte 2 de 3.)

```
47              // determina que CheckBoxes estão marcadas e cria Font
48              if (boldJCheckBox.isSelected() && italicJCheckBox.isSelected() )
49                 font = new Font( "Serif", Font.BOLD + Font.ITALIC, 14 );
50              else if (boldJCheckBox.isSelected() )
51                 font = new Font( "Serif", Font.BOLD, 14 );
52              else if (italicJCheckBox.isSelected() )
53                 font = new Font( "Serif", Font.ITALIC, 14 );
54              else
55                 font = new Font( "Serif", Font.PLAIN, 14 );
56
57              textField.setFont( font ); // configura a fonte do textField
58           } // fim do método itemStateChanged
59        } // fim da classe CheckBoxHandler interna private
60     } // fim da classe CheckBoxFrame
```

O método JCheckBox `isSelected` retorna `true` se o JCheckBox em que é chamado for verificado.

Figura 14.17 | Botões JCheckBox e eventos de item. (Parte 3 de 3.)


```
1  // Figura 14.18: CheckBoxTest.java
2  // Testando CheckBoxFrame.
3  import javax.swing.JFrame;
4
5  public class CheckBoxTest
6  {
7     public static void main( String[] args )
8     {
9        CheckBoxFrame checkBoxFrame = new CheckBoxFrame();
10       checkBoxFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       checkBoxFrame.setSize( 275, 100 ); // configura o tamanho do frame
12       checkBoxFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe CheckBoxTest
```

Figura 14.18 | Classe de teste para CheckBoxFrame. (Parte 1 de 2.)

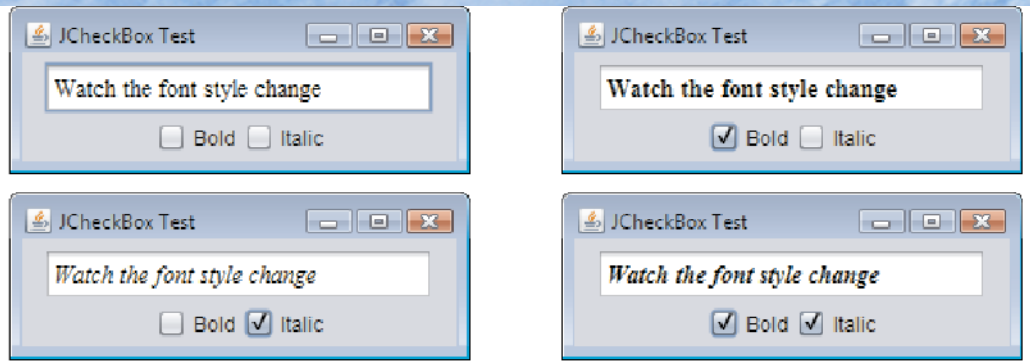


Figura 14.18 | Classe de teste para `CheckBoxFrame`. (Parte 2 de 2.)

14.10.2 JRadioButton

- **Botões de opção** (declarados com a classe `JRadioButton`) são semelhantes a caixas de seleção no sentido de que têm dois estados — selecionado e não selecionado.
- Os botões de opção normalmente aparecem como um **grupo** em que apenas um botão pode ser selecionado por vez.
- Selecionar um botão de opção diferente força a remoção da seleção de todos os outros que estão selecionados.
- Utilizados para representar **opções mutuamente exclusivas.**
- A relação lógica entre os botões de opção é mantida por um objeto **ButtonGroup** (pacote `javax.swing`), que organiza um grupo de botões e não é exibido em uma interface com o usuário.

Erro comum de programação 14.3

Adicionar um objeto `ButtonGroup` *(ou um objeto de qualquer outra classe que não deriva de* `Component`*) a um contêiner resulta em um erro de compilação.*


```
1  // Figura 14.19: RadioButtonFrame.java
2  // Criando botões de opção utilizando ButtonGroup e JRadioButton.
3  import java.awt.FlowLayout;
4  import java.awt.Font;
5  import java.awt.event.ItemListener;
6  import java.awt.event.ItemEvent;
7  import javax.swing.JFrame;
8  import javax.swing.JTextField;
9  import javax.swing.JRadioButton;
10 import javax.swing.ButtonGroup;
11
12 public class RadioButtonFrame extends JFrame
13 {
14    private JTextField textField; // utilizado para exibir alterações de fonte
15    private Font plainFont; // fonte para texto simples
16    private Font boldFont; // fonte para texto em negrito
17    private Font italicFont; // fonte para texto em itálico
18    private Font boldItalicFont; // fonte para texto em negrito e itálico
19    private JRadioButton plainJRadioButton; // seleciona texto simples
20    private JRadioButton boldJRadioButton; // seleciona texto em negrito
21    private JRadioButton italicJRadioButton; // seleciona texto em itálico
22    private JRadioButton boldItalicJRadioButton; // negrito e itálico
23    private ButtonGroup radioGroup; // buttongroup para armazenar botões de opção
24
```

Gerencia o relacionamento entre os botões de opção.

Figura 14.19 | JRadioButtons e ButtonGroups. (parte 1 de 4.)

```
25       // construtor RadioButtonFrame adiciona JRadioButtons ao JFrame
26       public RadioButtonFrame()
27       {
28          super( "RadioButton Test" );
29          setLayout( new FlowLayout() ); // configura o layout de frame
30
31          textField = new JTextField( "Watch the font style change", 25 );
32          add( textField ); // adiciona textField ao JFrame
33
34          // cria botões de opção
35          plainJRadioButton = new JRadioButton( "Plain", true );
36          boldJRadioButton = new JRadioButton( "Bold", false );
37          italicJRadioButton = new JRadioButton( "Italic", false );
38          boldItalicJRadioButton = new JRadioButton( "Bold/Italic", false );
39          add( plainJRadioButton ); // adiciona botão no estilo simples ao JFrame
40          add( boldJRadioButton ); // adiciona botão de negrito ao JFrame
41          add( italicJRadioButton ); // adiciona botão de itálico ao JFrame
42          add( boldItalicJRadioButton ); // adiciona botão de negrito e itálico
43
44          // cria relacionamento lógico entre JRadioButtons
45          radioGroup = new ButtonGroup(); // cria ButtonGroup
46          radioGroup.add( plainJRadioButton ); // adiciona simples ao grupo
47          radioGroup.add( boldJRadioButton ); // adiciona negrito ao grupo
```

Esse será selecionado inicialmente.

Gerencia o relacionamento entre todos os botões de opção que são adicionados ao grupo.

Figura 14.19 | JRadioButtons e ButtonGroups. (Parte 2 de 4.)

```
48          radioGroup.add( italicJRadioButton ); // adiciona itálico ao grupo
49          radioGroup.add( boldItalicJRadioButton ); // adiciona negrito e itálico
50
51          // cria fonte objetos
52          plainFont = new Font( "Serif", Font.PLAIN, 14 );
53          boldFont = new Font( "Serif", Font.BOLD, 14 );
54          italicFont = new Font( "Serif", Font.ITALIC, 14 );
55          boldItalicFont = new Font( "Serif", Font.BOLD + Font.ITALIC, 14 );
56          textField.setFont( plainFont ); // configura a fonte inicial como simples
57
58          // registra eventos para JRadioButtons
59          plainJRadioButton.addItemListener(
60             new RadioButtonHandler( plainFont ) );
61          boldJRadioButton.addItemListener(
62             new RadioButtonHandler( boldFont ) );
63          italicJRadioButton.addItemListener(
64             new RadioButtonHandler( italicFont ) );
65          boldItalicJRadioButton.addItemListener(
66             new RadioButtonHandler( boldItalicFont ) );
67       } // fim do construtor RadioButtonFrame
68
```

Observe que estamos criando um objeto de tratamento de evento separado para cada `JRadioButton`. Isso nos permite especificar a `Font` exata que será utilizada quando uma em particular for selecionada.

Figura 14.19 | JRadioButtons e ButtonGroups. (Parte 3 de 4.)

```
69       // classe interna private para tratar eventos de botão de opção
70       private class RadioButtonHandler implements ItemListener
71       {
72          private Font font; // fonte associada com esse listener
73
74          public RadioButtonHandler( Font f )
75          {
76             font = f; // configura a fonte desse listener
77          } // fim do construtor RadioButtonHandler
78
79          // trata eventos de botão de opção
80          public void itemStateChanged( ItemEvent event )
81          {
82             textField.setFont( font ); // configura fonte de textField
83          } // fim do método itemStateChanged
84       } // fim da classe RadioButtonHandler interna private
85    } // fim da classe RadioButtonFrame
```

Os objetos dessa classe podem responder a `ItemEvents`.

Armazena a `Font` específica para um botão de opção em particular.

Figura 14.19 | `JRadioButtons` e `ButtonGroups`. (Parte 4 de 4.)


```
1  // Figura 14.20: RadioButtonTest.java
2  // Testando RadioButtonFrame.
3  import javax.swing.JFrame;
4
5  public class RadioButtonTest
6  {
7     public static void main( String[] args )
8     {
9        RadioButtonFrame radioButtonFrame = new RadioButtonFrame();
10       radioButtonFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       radioButtonFrame.setSize( 300, 100 ); // configura o tamanho do frame
12       radioButtonFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe RadioButtonTest
```

Figura 14.20 | Classe de teste para `RadioButtonFrame`. (Parte 1 de 2.)

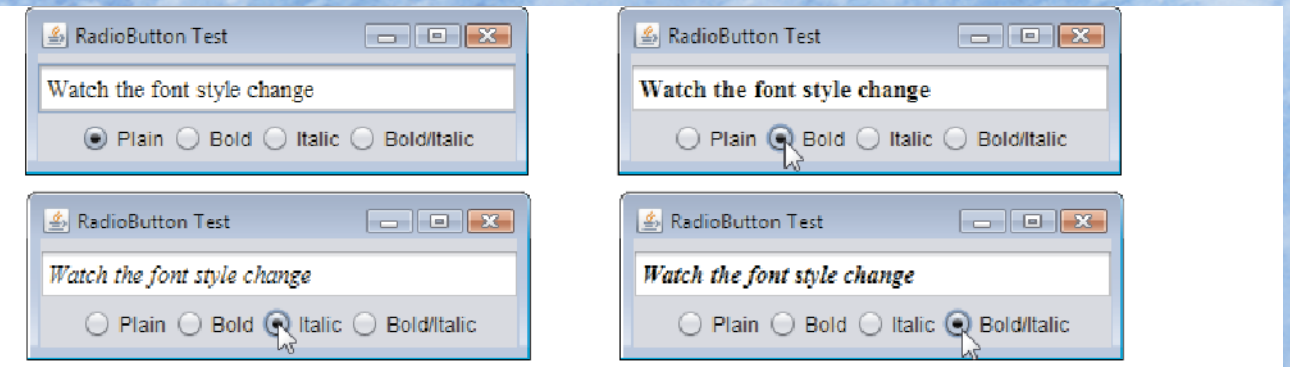


Figura 14.20 | Classe de teste para `RadioButtonFrame`. (Parte 2 de 2.)

- O método `ButtonGroup` **`add`** associa um `JRadioButton` com o grupo.
- Se mais de um objeto `JRadioButton` selecionado for adicionado ao grupo, aquele que tiver sido adicionado primeiro será selecionado quando a GUI for exibida.
- `JRadioButton`s, assim como `JCheckBox`es, geram `ItemEvent`s quando são clicados.

14.11 JComboBox e utilização de uma classe interna anônima para tratamento de eventos

- Uma caixa de combinação (ou **lista drop-down**) permite ao usuário selecionar um item de uma lista.
- Caixas de combinação são implementadas com a classe **JComboBox**, que estende a classe `JComponent`.
- `JComboBox`es geram `ItemEvent`s.


```
1  // Figura 14.21: ComboBoxFrame.java
2  // JComboBox que exibe uma lista de nomes de imagem.
3  import java.awt.FlowLayout;
4  import java.awt.event.ItemListener;
5  import java.awt.event.ItemEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JLabel;
8  import javax.swing.JComboBox;
9  import javax.swing.Icon;
10 import javax.swing.ImageIcon;
11
12 public class ComboBoxFrame extends JFrame
13 {
14    private JComboBox imagesJComboBox; // caixa de combinação para armazenar nomes de ícones
15    private JLabel label; // rótulo para exibir ícone selecionado
16
17    private static final String[] names =
18       { "bug1.gif", "bug2.gif",  "travelbug.gif", "buganim.gif" };
19    private Icon[] icons = {
20       new ImageIcon( getClass().getResource( names[ 0 ] ) ),
21       new ImageIcon( getClass().getResource( names[ 1 ] ) ),
22       new ImageIcon( getClass().getResource( names[ 2 ] ) ),
23       new ImageIcon( getClass().getResource( names[ 3 ] ) ) };
24
```

Figura 14.21 | JComboBox que exibe uma lista de nomes de imagens. (Parte 1 de 3.)

```
25       // construtor ComboBoxFrame adiciona JComboBox ao JFrame
26       public ComboBoxFrame()
27       {
28          super( "Testing JComboBox" );
29          setLayout( new FlowLayout() ); // configura o layout de frame
30
31          imagesJComboBox = new JComboBox( names ); // configura JComboBox
32          imagesJComboBox.setMaximumRowCount( 3 ); // exibe três linhas
33
34          imagesJComboBox.addItemListener(
35             new ItemListener() // classe interna anônima
36             {
37                // trata evento JComboBox
38                public void itemStateChanged( ItemEvent event )
39                {
40                   // determina se o item selecionado
41                   if ( event.getStateChange() == ItemEvent.SELECTED )
42                      label.setIcon( icons[
43                         imagesJComboBox.getSelectedIndex() ] );
44                } // fim do método itemStateChanged
45             } // fim da classe interna anônima
46          ); // fim da chamada para addItemListener
47
```

Utiliza as `String`s nos nomes de array como as opções no `JComboBox`.

As linhas 34-46 criam um objeto de uma classe interna anônima que implementa a interface `ItemListener` e registra esse objeto para que trate os `ItemEvent`s do `JComboBox`.

Figura 14.21 | `JComboBox` que exibe uma lista de nomes de imagens. (Parte 2 de 3.)


```
48          add( imagesJComboBox ); // adiciona combobox ao JFrame
49          label = new JLabel( icons[ 0 ] ); // exibe primeiro ícone
50          add( label ); // adiciona rótulo ao JFrame
51       } // fim do construtor ComboBoxFrame
52    } // fim da classe ComboBoxFrame
```

Figura 14.21 | JComboBox que exibe uma lista de nomes de imagens. (Parte 3 de 3.)


```
1  // Figura 14.22: ComboBoxTest.java
2  // Testando ComboBoxFrame.
3  import javax.swing.JFrame;
4
5  public class ComboBoxTest
6  {
7     public static void main( String[] args )
8     {
9        ComboBoxFrame comboBoxFrame = new ComboBoxFrame();
10       comboBoxFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       comboBoxFrame.setSize( 350, 150 ); // configura o tamanho do frame
12       comboBoxFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe ComboBoxTest
```

Figura 14.22 | Testando ComboBoxFrame. (Parte 1 de 2.)

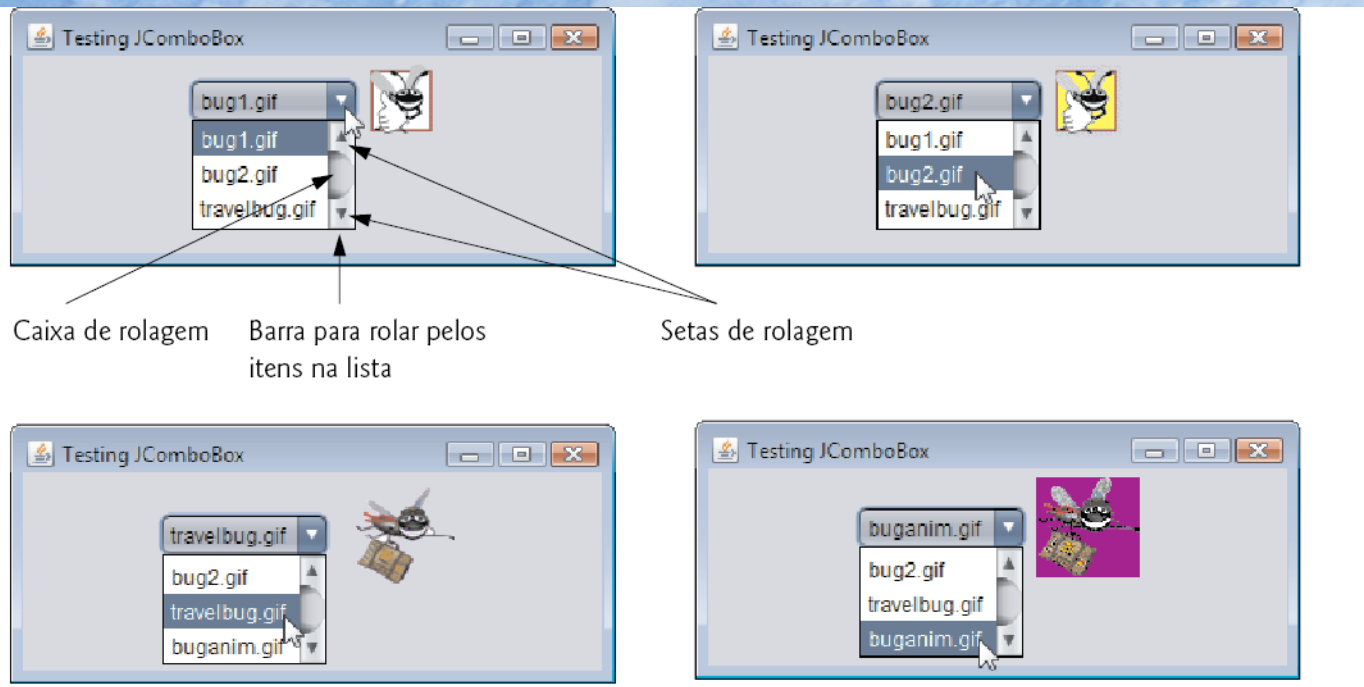


Figura 14.22 | Testando `ComboBoxFrame`. (Parte 2 de 2.)

- O primeiro item adicionado a uma `JComboBox` aparece como o item atualmente selecionado quando a `JComboBox` é exibida.
- Outros itens são selecionados clicando na `JComboBox` e, então, selecionando um item da lista que aparece.
- O método `JComboBox` **setMaximumRowCount** configura o número máximo de elementos que é exibido quando o usuário clica na `JComboBox`.
- Se houver itens adicionais, a `JComboBox` fornece uma **barra de rolagem** que permite que o usuário role por todos os elementos da lista.

Java™ COMO PROGRAMAR 8ª edição

Observação sobre a aparência e comportamento 14.11

Configure a contagem máxima de linha para uma JComboBox *como um número de linhas que impede a lista de expandir para fora dos limites da janela em que ela é utilizada.*

- Uma **classe interna anônima** é uma classe interna que é declarada sem nome e, em geral, aparece dentro de uma declaração de método.
- Como outras classes internas, uma classe interna anônima pode acessar os membros de sua classe de primeiro nível.
- Uma classe interna anônima tem acesso limitado às variáveis locais do método em que ela é declarada.
- Visto que uma classe interna anônima não tem nomes, deve-se criar um objeto da classe interna anônima no ponto em que a classe é declarada.

Observação de engenharia de software 14.5

Uma classe interna anônima declarada em um método pode acessar as variáveis de instância e métodos do objeto de classe de primeiro nível que a declararam, bem como as variáveis locais `final` *do método, mas não pode acessar variáveis locais não* `final` *do método.*

- O método `JComboBox` **getSelectedIndex** retorna o índice do item selecionado.
- Para cada item selecionado de uma `JComboBox`, a seleção de outro item é primeiro removida — então dois `ItemEvent`s ocorrem quando um item é selecionado.
- O método `ItemEvent` **getStateChange** retorna o tipo de alteração de estado. `ItemEvent.SELECTED` indica que um item foi selecionado.

Observação de engenharia de software 14.6

Como qualquer outra classe, quando uma classe interna anônima implementa uma interface, a classe deve implementar cada método na interface.

- Uma lista exibe uma série de itens da qual o usuário pode selecionar um ou mais itens.
- Listas são criadas com a classe `JList`, que estende diretamente a classe `JComponent`.
- Suporta **listas de uma única seleção** (apenas um item selecionado por vez) e **listas de seleção múltipla** (qualquer número de itens selecionado).
- `JList`s geram **ListSelectionEvent**s em listas de uma única seleção.


```
1  // Figura 14.23: ListFrame.java
2  // JList que exibe uma lista de cores.
3  import java.awt.FlowLayout;
4  import java.awt.Color;
5  import javax.swing.JFrame;
6  import javax.swing.JList;
7  import javax.swing.JScrollPane;
8  import javax.swing.event.ListSelectionListener;
9  import javax.swing.event.ListSelectionEvent;
10 import javax.swing.ListSelectionModel;
11
12 public class ListFrame extends JFrame
13 {
14    private JList colorJList; // lista para exibir cores
15    private static final String[] colorNames = { "Black", "Blue", "Cyan",
16       "Dark Gray", "Gray", "Green", "Light Gray", "Magenta",
17       "Orange", "Pink", "Red", "White", "Yellow" };
18    private static final Color[] colors = { Color.BLACK, Color.BLUE,
19       Color.CYAN, Color.DARK_GRAY, Color.GRAY, Color.GREEN,
20       Color.LIGHT_GRAY, Color.MAGENTA, Color.ORANGE, Color.PINK,
21       Color.RED, Color.WHITE, Color.YELLOW };
22
```

Figura 14.23 | `JList` que exibe uma lista de cores. (Parte 1 de 3.)

```
23       // construtor ListFrame adiciona JScrollPane que contém JList ao JFrame
24       public ListFrame()
25       {
26          super( "List Test" );
27          setLayout( new FlowLayout() ); // configura o layout de frame
28
29          colorJList = new JList( colorNames ); // cria com colorNames
30          colorJList.setVisibleRowCount( 5 ); // exibe cinco linhas de uma vez
31
32          // não permite múltiplas seleções
33          colorJList.setSelectionMode( ListSelectionModel.SINGLE_SELECTION );
34
35          // adiciona um JScrollPane que contém JList ao frame
36          add( new JScrollPane( colorJList ) );
37
```

Preenche o `JList` com as `String`s no array `colorNames`.

Permite somente seleções únicas.

Fornece barras de rolagem para o `JList` se necessário.

Figura 14.23 | `JList` que exibe uma lista de cores. (Parte 2 de 3.)

```
38          colorJList.addListSelectionListener(
39             new ListSelectionListener() // classe interna anônima
40             {
41                // trata eventos de seleção de lista
42                public void valueChanged( ListSelectionEvent event )
43                {
44                   getContentPane().setBackground(
45                      colors[colorJList.getSelectedIndex() ] );
46                } // fim do método valueChanged
47             } // fim da classe interna anônima
48          ); // fim da chamada para addListSelectionListener
49       } // fim do construtor ListFrame
50    } // fim da classe ListFrame
```

Escolhe a `Color` apropriada para alterar a cor de fundo da janela.

Figura 14.23 | `JList` que exibe uma lista de cores. (Parte 3 de 3.)

```
1  // Figura 14.24: ListTest.java
2  // Selecionando as cores de uma JList.
3  import javax.swing.JFrame;
4
5  public class ListTest
6  {
7     public static void main( String[] args )
8     {
9        ListFrame listFrame = new ListFrame(); // cria ListFrame
10       listFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       listFrame.setSize( 350, 150 ); // configura o tamanho do frame
12       listFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe ListTest
```

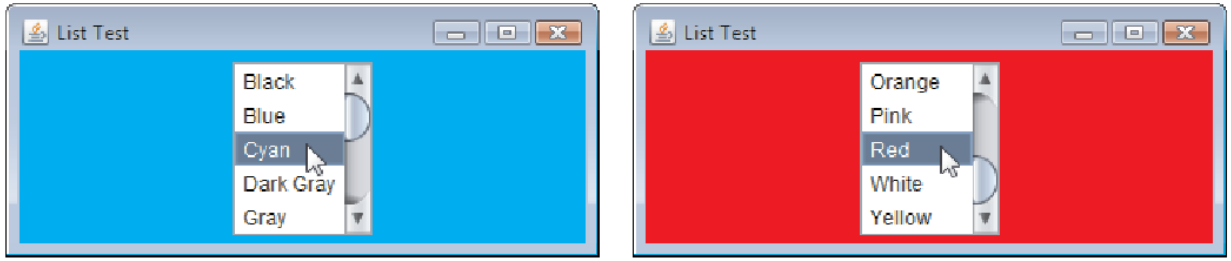


Figura 14.24 | Classe de teste para `ListFrame`.

- **setVisibleRowCount** especifica o número de itens visíveis na lista.
- **setSelectionMode** especifica o **modo de seleção** da lista.
- A classe **ListSelectionModel** (do pacote `javax.swing`) declara as constantes do modo de seleção
- **SINGLE_SELECTION** (apenas um item selecionado por vez).
- **SINGLE_INTERVAL_SELECTION** (permite seleção de vários itens contíguos).
- **MULTIPLE_INTERVAL_SELECTION** (não restringe os itens que podem ser selecionados).

- Ao contrário de uma `JComboBox`, uma `JList` *não* fornece uma barra de rolagem se houver mais itens na lista do que o número de linhas visíveis.
- Um objeto **JScrollPane** é utilizado para fornecer a capacidade de rolagem.
- **addListSelectionListener** registra um **ListSelectionListener** (pacote `javax.swing.event`) como o ouvinte para os eventos de seleção de uma `JList`.

- Cada `JFrame` realmente consiste em três camadas — o fundo, o painel de conteúdo e o painel transparente.
- O painel de conteúdo aparece na frente do fundo e é onde os componentes GUI do `JFrame` são exibidos.
- O painel transparente é utilizado para exibir dicas de ferramenta e outros itens que devem aparecer na frente dos componentes GUI na tela.
- O painel de conteúdo oculta completamente o fundo do `JFrame`.
- Para mudar a cor de fundo detrás dos componentes GUI, você deve mudar a cor de fundo do painel de conteúdo.
- O método `getContentPane` retorna uma referência ao painel de conteúdo do `JFrame` (um objeto da classe `Container`).
- O método `List` **getSelectedIndex** retorna o índice do item selecionado.

14.13 Listas de seleção múltipla

- Uma **lista de seleção múltipla** permite ao usuário selecionar muitos itens de uma `JList`.
- Uma lista `SINGLE_INTERVAL_SELECTION` permite selecionar um intervalo contíguo de itens.
- Para fazer isso, clique no primeiro item e, então, pressione e segure a tecla *Shift* ao clicar no último item do intervalo.
- Uma lista `MULTIPLE_INTERVAL_SELECTION` (o padrão) permite a seleção de intervalo contínuo como descrito para uma lista `SINGLE_INTERVAL_SELECTION` e permite que diversos itens sejam selecionados pressionando e segurando a tecla *Ctrl* ao clicar em cada item a ser selecionado.
- Para remover a seleção de um item, pressione e segure a tecla *Ctrl* ao clicar no item uma segunda vez.

```
1  // Figura 14.25: MultipleSelectionFrame.java
2  // Copiando itens de uma Lista para outra.
3  import java.awt.FlowLayout;
4  import java.awt.event.ActionListener;
5  import java.awt.event.ActionEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JList;
8  import javax.swing.JButton;
9  import javax.swing.JScrollPane;
10 import javax.swing.ListSelectionModel;
11
12 public class MultipleSelectionFrame extends JFrame
13 {
14    private JList colorJList; // lista para armazenar nomes de cor
15    private JList copyJList; // lista para copiar nomes de cor em
16    private JButton copyJButton; // botão para copiar nomes selecionados
17    private static final String[] colorNames = { "Black", "Blue", "Cyan",
18       "Dark Gray", "Gray", "Green", "Light Gray", "Magenta", "Orange",
19       "Pink", "Red", "White", "Yellow" };
20
21    // construtor MultipleSelectionFrame
22    public MultipleSelectionFrame()
23    {
```

Figura 14.25 | `JList` que permite múltiplas seleções. (Parte 1 de 3.)

```
24          super( "Multiple Selection Lists" );
25          setLayout( new FlowLayout() ); // configura o layout de frame
26
27          colorJList = new JList( colorNames ); // armazena nomes de todas as cores
28          colorJList.setVisibleRowCount( 5 ); // mostra cinco linhas
29          colorJList.setSelectionMode(
30             ListSelectionModel.MULTIPLE_INTERVAL_SELECTION );
31          add( new JScrollPane( colorJList ) ); // adiciona lista com scrollpane
32
33          copyJButton = new JButton( "Copy >>>" ); // cria botão de cópia
34          copyJButton.addActionListener(
35
36             new ActionListener() // classe interna anônima
37             {
38                // trata evento de botão
39                public void actionPerformed( ActionEvent event )
40                {
41                   // coloca valores selecionados na copyJList
42                   copyJList.setListData( colorJList.getSelectedValues() );
43                } // fim do método actionPerformed
44             } // fim da classe interna anônima
45          ); // fim da chamada para addActionListener
46
47          add( copyJButton ); // adiciona botão de cópia ao JFrame
```

Permite múltiplas seleções ou intervalos no `JList`.

Figura 14.25 | `JList` que permite múltiplas seleções. (Parte 2 de 3.)


```
48
49          copyJList = new JList(); // cria lista para armazenar nomes de cor copiados
50          copyJList.setVisibleRowCount( 5 ); // mostra 5 linhas
51          copyJList.setFixedCellWidth( 100 ); // configura a largura
52          copyJList.setFixedCellHeight( 15 ); // configura a altura
53          copyJList.setSelectionMode(
54             ListSelectionModel.SINGLE_INTERVAL_SELECTION );
55          add( new JScrollPane( copyJList ) ); // adiciona lista com scrollpane
56       } // fim do construtor MultipleSelectionFrame
57    } // fim da classe MultipleSelectionFrame
```

Permite que somente intervalos únicos (ranges) sejam selecionados.

Figura 14.25 | `JList` que permite múltiplas seleções. (Parte 3 de 3.)

```
1  // Figura 14.26: MultipleSelectionTest.java
2  // Testando MultipleSelectionFrame.
3  import javax.swing.JFrame;
4
5  public class MultipleSelectionTest
6  {
7     public static void main( String[] args )
8     {
9        MultipleSelectionFrame multipleSelectionFrame =
10          new MultipleSelectionFrame();
11       multipleSelectionFrame.setDefaultCloseOperation(
12          JFrame.EXIT_ON_CLOSE );
13       multipleSelectionFrame.setSize( 350, 150 ); // configura o tamanho do frame
14       multipleSelectionFrame.setVisible( true ); // exibe o frame
15    } // fim de main
16 } // fim da classe MultipleSelectionTest
```

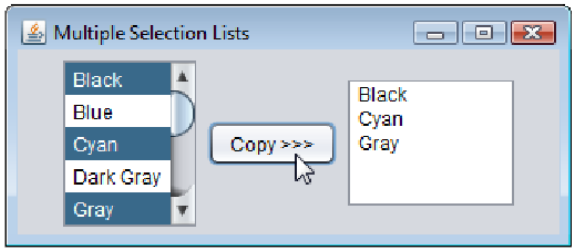


Figura 14.26 | Classe de teste para `MultipleSelectionFrame`.

- Se uma `JList` não contiver itens, ela não será exibida em um `FlowLayout`.
- Utilize os métodos `JList` **setFixedCellWidth** e **setFixedCellHeight** para configurar o item largura e altura.
- Não há nenhum evento para indicar que um usuário fez múltiplas seleções em uma lista de seleção múltipla.
- Um evento gerado por outro componente GUI (conhecido como **evento externo**) especifica quando as múltiplas seleções em uma `JList` devem ser processadas.
- O método **setListData** configura os itens exibidos em uma `Jlist`.
- O método **getSelectedValues** retorna um array de `Objects` para representar os itens selecionados em uma `JList`.

14.14 Tratamento de eventos de mouse

- Interfaces de ouvinte de eventos **MouseListener** e **MouseMotionListener** para tratar **eventos de mouse**.
- Os eventos de mouse podem ser capturados por qualquer componente GUI que deriva de java.awt.Component.
- O pacote `javax.swing.event` contém a interface **MouseInputListener**, que estende as interfaces `MouseListener` e `MouseMotionListener` para criar uma única interface que contenha todos os métodos.
- Os métodos `MouseListener` e `MouseMotionListener` são chamados quando o mouse interage com um `Component` se objetos listeners de eventos apropriados forem registrados para esse `Component`.

Métodos de interface `MouseListener` e `MouseMotionListener`

Métodos de interface `MouseListener`

`public void mousePressed( MouseEvent event )`

Chamado quando um botão do mouse é pressionado enquanto o cursor do mouse estiver sobre um componente.

`public void mouseClicked( MouseEvent event )`

Chamado quando um botão do mouse é pressionado e liberado enquanto o cursor do mouse pairar sobre um componente. Esse evento é sempre precedido por uma chamada para `mousePressed`.

`public void mouseReleased( MouseEvent event )`

Chamado quando um botão do mouse é liberado depois de ser pressionado. Esse evento sempre é precedido por uma chamada para `mousePressed` e uma ou mais chamadas para `mouseDragged`.

`public void mouseEntered( MouseEvent event )`

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

`public void mouseExited( MouseEvent event )`

Chamado quando o cursor do mouse deixa os limites de um componente.

Figura 14.27 | Métodos de interface `MouseListener` e `MouseMotionListener`. (Parte 1 de 2.)

Métodos de interface `MouseListener` e `MouseMotionListener`

Métodos de interface `MouseMotionListener`

`public void mouseDragged( MouseEvent event )`

Chamado quando o botão do mouse é pressionado enquanto o cursor do mouse estiver sobre um componente e o mouse é movido enquanto o botão do mouse permanecer pressionado. Esse evento é sempre precedido por uma chamada para `mousePressed`. Todos os eventos de arrastar são enviados para o componente em que o usuário começou a arrastar o mouse.

`public void mouseMoved( MouseEvent event )`

Chamado quando o mouse é movido (sem pressionamentos de botões do mouse) quando o cursor do mouse está sobre um componente. Todos os eventos de movimento são enviados para o componente sobre o qual o mouse atualmente está posicionado.

Figura 14.27 | Métodos de interface `MouseListener` e `MouseMotionListener`. (Parte 2 de 2.)

- Cada método de tratamento do evento de mouse recebe um objeto **MouseEvent** que contém informações sobre o evento, incluindo as coordenadas *x* e y da localização em que ocorreu o evento.
- As coordenadas são medidas a partir do canto superior esquerdo do componente GUI em que o evento ocorreu.
- A coordenada *x* inicia em 0 e aumenta da esquerda para a direita. A coordenada *y* inicia em 0 e aumenta de cima para baixo.
- Os métodos e as constantes de classe **InputEvent** (superclasse de `MouseEvent`) permitem determinar o botão do mouse em que o usuário clicou.

Observação de engenharia de software 14.7

As chamadas de método para `mouseDragged` *são enviadas para o* `MouseMotionListener` *do* `Component` *em que a operação de arrastar o mouse se iniciou. De maneira semelhante, a chamada de método* `mouseReleased` *no fim de uma operação de arrastar é enviada para o* `MouseListener` *do* `Component` *em que a operação de arrastar iniciou.*

- A interface **MouseWheelListener** permite aos aplicativos responder à rotação da roda de um mouse.
- O método **mouseWheelMoved** recebe um **MouseWheelEvent** como seu argumento.
- A classe `MouseWheelEvent` (uma subclasse de `MouseEvent`) contém métodos que permitem que o handler de evento obtenha as informações sobre a quantidade de rotação da roda.


```
1  // Figura 14.28: MouseTrackerFrame.java
2  // Demonstrando os eventos de mouse.
3  import java.awt.Color;
4  import java.awt.BorderLayout;
5  import java.awt.event.MouseListener;
6  import java.awt.event.MouseMotionListener;
7  import java.awt.event.MouseEvent;
8  import javax.swing.JFrame;
9  import javax.swing.JLabel;
10 import javax.swing.JPanel;
11
12 public class MouseTrackerFrame extends JFrame
13 {
14    private JPanel mousePanel; // painel em que os eventos de mouse ocorrerão
15    private JLabel statusBar; // rótulo que exibe informações de evento
16
17    // construtor MouseTrackerFrame configura GUI e
18    // registra handlers de evento de mouse
19    public MouseTrackerFrame()
20    {
21       super( "Demonstrating Mouse Events" );
22
```

Figura 14.28 | Tratamento de eventos de mouse. (Parte 1 de 4.)

```
23          mousePanel = new JPanel(); // cria painel
24          mousePanel.setBackground( Color.WHITE ); // configura cor de fundo
25          add( mousePanel, BorderLayout.CENTER ); // adiciona painel ao JFrame
26
27          statusBar = new JLabel( "Mouse outside JPanel" );
28          add( statusBar, BorderLayout.SOUTH ); // adiciona rótulo ao JFrame
29
30          // cria e registra listener para mouse e eventos de movimento de mouse
31          MouseHandler handler = new MouseHandler();
32          mousePanel.addMouseListener( handler );
33          mousePanel.addMouseMotionListener( handler );
34       } // construtor de MouseTrackerFrame de fim
35
36       private class MouseHandler implements MouseListener,
37          MouseMotionListener
38       {
39          // Handlers de evento de MouseListener
40          // trata evento quando o mouse é liberado imediatamente depois de ter sido pressionado
41          public void mouseClicked( MouseEvent event )
42          {
43             statusBar.setText( String.format( "Clicked at [%d, %d]",
44                event.getX(),event.getY() ) );
45          } // fim do método mouseClicked
46
```

Objeto que trata os eventos de mouse e os eventos de ação do mouse.

Um objeto dessa classe *é* um `MouseListener` e *é* um `MouseMotionListener`

Obtém as coordenadas do mouse no momento em que o evento de clique do mouse ocorreu

Figura 14.28 | Tratamento de eventos de mouse. (Parte 2 de 4.)

```
47          // trata evento quando mouse é pressionado
48          public void mousePressed( MouseEvent event )
49          {
50             statusBar.setText( String.format( "Pressed at [%d, %d]",
51                event.getX(),event.getY() ) );
52          } // fim do método mousePressed
53
54          // trata o evento quando o mouse é liberado
55          public void mouseReleased( MouseEvent event )
56          {
57             statusBar.setText( String.format( "Released at [%d, %d]",
58                event.getX(),event.getY() ) );
59          } // fim do método mouseReleased
60
61          // trata evento quando mouse entra na área
62          public void mouseEntered( MouseEvent event )
63          {
64             statusBar.setText( String.format( "Mouse entered at [%d, %d]",
65                event.getX(),event.getY() ) );
66             mousePanel.setBackground( Color.GREEN );
67          } // fim do método mouseEntered
68
```

Obtém as coordenadas do mouse no momento em que o evento de pressionar o mouse ocorreu.

Obtém as coordenadas do mouse no momento em que o evento de liberar o mouse ocorreu.

Obtém as coordenadas do mouse no momento em que o evento de inserir com o mouse ocorreu e altera o fundo para verde.

Figura 14.28 | Tratamento de eventos de mouse. (Parte 3 de 4.)

```
69          // trata evento quando mouse sai da área
70          public void mouseExited( MouseEvent event )
71          {
72             statusBar.setText( "Mouse outside JPanel" );
73             mousePanel.setBackground( Color.WHITE );
74          } // fim do método mouseExited
75
76          // Handlers de evento de MouseMotionListener
77          // trata evento quando usuário arrasta o mouse com o botão pressionado
78          public void mouseDragged( MouseEvent event )
79          {
80             statusBar.setText( String.format( "Dragged at [%d, %d]",
81                event.getX(),event.getY() ) );
82          } // fim do método mouseDragged
83
84          // trata evento quando usuário move o mouse
85          public void mouseMoved( MouseEvent event )
86          {
87             statusBar.setText( String.format( "Moved at [%d, %d]",
88                event.getX(),event.getY() ) );
89          } // fim do método mouseMoved
90       } // fim da classe interna MouseHandler
91    } // fim da classe MouseTrackerFrame
```

Altera o fundo para branco quando o mouse sai da área.

Obtém as coordenadas do mouse no momento em que o evento de arrastar com o mouse ocorreu.

Obtém as coordenadas do mouse no momento em que o evento de mover com o mouse ocorreu.

Figura 14.28 | Tratamento de eventos de mouse. (Parte 4 de 4.)


```
1  // Figura 14.29: MouseTrackerFrame.java
2  // Testando MouseTrackerFrame.
3  import javax.swing.JFrame;
4
5  public class MouseTracker
6  {
7     public static void main( String[] args )
8     {
9        MouseTrackerFrame mouseTrackerFrame = new MouseTrackerFrame();
10       mouseTrackerFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       mouseTrackerFrame.setSize( 300, 100 ); // configura o tamanho do frame
12       mouseTrackerFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe MouseTracker
```

Figura 14.29 | Classe de teste para MouseTrackerFrame. (Parte 1 de 2.)

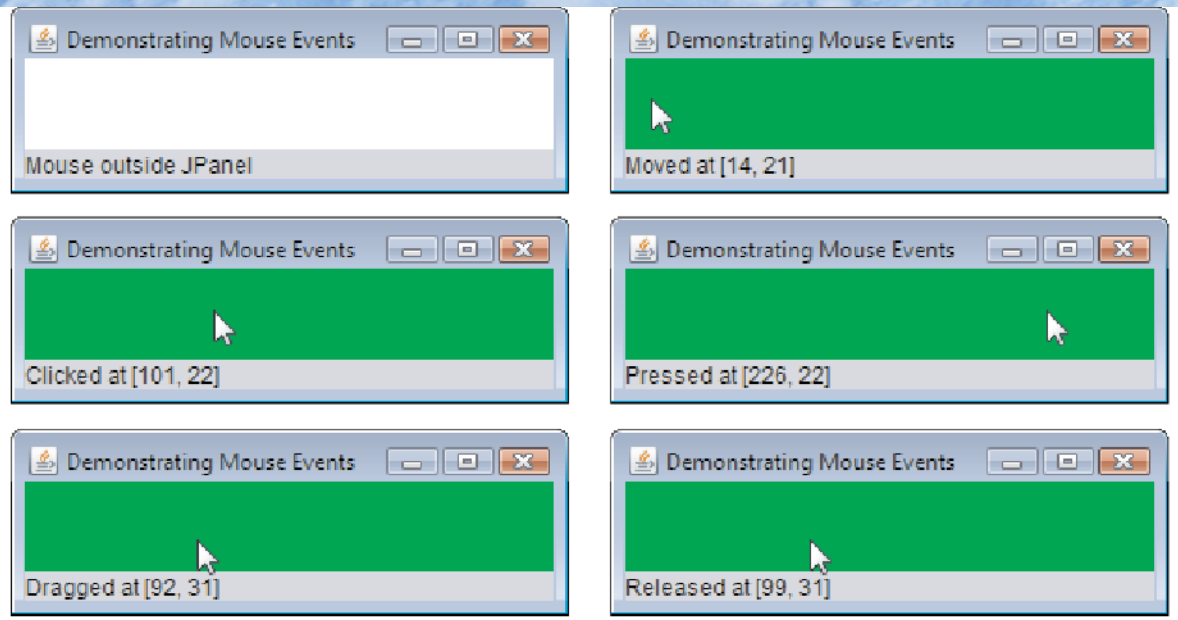


Figura 14.29 | Classe de teste para `MouseTrackerFrame`. (Parte 2 de 2.)

- **BorderLayout** organiza os componentes em cinco regiões: **NORTH**, **SOUTH**, **EAST**, **WEST** e **CENTER**.
- `BorderLayout` dimensiona o componente em `CENTER` para utilizar todo o espaço disponível que não está ocupado.
- Os métodos **addMouseListener** e **addMouseMotionListener** registram `MouseListener`s e `MouseMotionListener`s, respectivamente.
- Os métodos `MouseEvent` **getX** e **getY** retornam as coordenadas *x* e *y* do mouse no momento em que o evento ocorreu.

14.15 Classes adaptadoras

- Muitas interfaces *listener* de eventos contêm múltiplos métodos.
- Uma **classe adaptadora** implementa uma interface e fornece uma implementação padrão (com o corpo de um método vazio) de cada método na interface.
- Você pode estender uma classe adaptadora para herdar a implementação padrão de cada método e sobrescrever somente o(s) método(s) necessário(s) para o tratamento de evento.

Observação de engenharia de software 14.8

Quando uma classe implementar uma interface, a classe terá um relacionamento é um *com essa interface. Todas as subclasses diretas e indiretas dessa classe herdam essa interface. Portanto, um objeto de uma classe que estende uma classe adaptadora de evento é um objeto do tipo ouvinte de evento correspondente (por exemplo, um objeto de uma subclasse de* `MouseAdapter` *é um* `MouseListener`*).*

Classe adaptadora de evento no `java.awt.event`	Implementa a interface
`ComponentAdapter`	`ComponentListener`
`ContainerAdapter`	`ContainerListener`
`FocusAdapter`	`FocusListener`
`KeyAdapter`	`KeyListener`
`MouseAdapter`	`MouseListener`
`MouseMotionAdapter`	`MouseMotionListener`
`WindowAdapter`	`WindowListener`

Figura 14.30 | Classes adaptadoras de evento e as interfaces que elas implementam no pacote `java.awt.event`.


```
1  // Figura 14.31: MouseDetailsFrame.java
2  // Demonstrando cliques de mouse e distinguindo entre botões do mouse.
3  import ons.awt.BorderLayout;
4  import java.awt.event.MouseAdapter;
5  import java.awt.event.MouseEvent;
6  import javax.swing.Jframe;
7  import javax.swing.Jlabel;
8
9  public class MouseDetailsFrame extends Jframe
10 {
11    private String details; // A string que é exibida na barra de status
12    private Jlabel statusBar; // Jlabel que aparece na parte inferior da janela
13
14    // construtor configura String de barra de título e registra o listener de mouse
15    public MouseDetailsFrame()
16    {
17       super( "Mouse clicks and buttons" );
18
19       statusBar = new Jlabel( "Click the mouse" );
20       add( statusBar, BorderLayout.SOUTH );
21       addMouseListener( new MouseClickHandler() ); // adiciona handler
22    } // fim do construtor MouseDetailsFrame
23
```

Figura 14.31 | Clique dos botões esquerdo, central e direito do mouse. (Parte 1 de 2.)

```
24      // classe interna para tratar eventos de mouse
25      private class MouseClickHandler extends MouseAdapter
26      {
27         // trata evento de clique de mouse e determina qual botão foi pressionado
28         public void mouseClicked( MouseEvent event )
29         {
30            int xPos = event.getX(); // obtém a posição x do mouse
31            int yPos = event.getY(); // obtém a posição y do mouse
32
33            details = String.format( "Clicked %d time(s)",
34               event.getClickCount() );
35
36            if (event.isMetaDown() ) // botão direito do mouse
37               details += " with right mouse button";
38            else if (event.isAltDown() ) // botão do meio do mouse
39               details += " with center mouse button";
40            else // botão esquerdo do mouse
41               details += " with left mouse button";
42
43            statusBar.setText( details ); // exibe mensagem em statusBar
44         } // fim do método mouseClicked
45      } // fim da classe interna private MouseClickHandler
46   } // fim da classe MouseDetailsFrame
```

O adaptador permite-nos sobrescrever o único método que utilizamos nesse exemplo.

Retorna o número de cliques do mouse. Se você esperar tempo suficiente entre os cliques, a contagem reinicia em 0.

Ajuda a determinar qual o botão que o usuário pressionou no mouse.

Figura 14.31 | Clique dos botões esquerdo, central e direito do mouse. (Parte 2 de 2.)


```
1  // Figura 14.32: MouseDetails.java
2  // Testando MouseDetailsFrame.
3  import javax.swing.Jframe;
4
5  public class MouseDetails
6  {
7     public static void main( String[] args )
8     {
9        MouseDetailsFrame mouseDetailsFrame = new MouseDetailsFrame();
10       mouseDetailsFrame.setDefaultCloseOperation( Jframe.EXIT_ON_CLOSE );
11       mouseDetailsFrame.setSize( 400, 150 ); // configura o tamanho do frame
12       mouseDetailsFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe MouseDetails
```

Figura 14.32 | Classe de teste para MouseDetailsFrame. (Parte 1 de 2.)

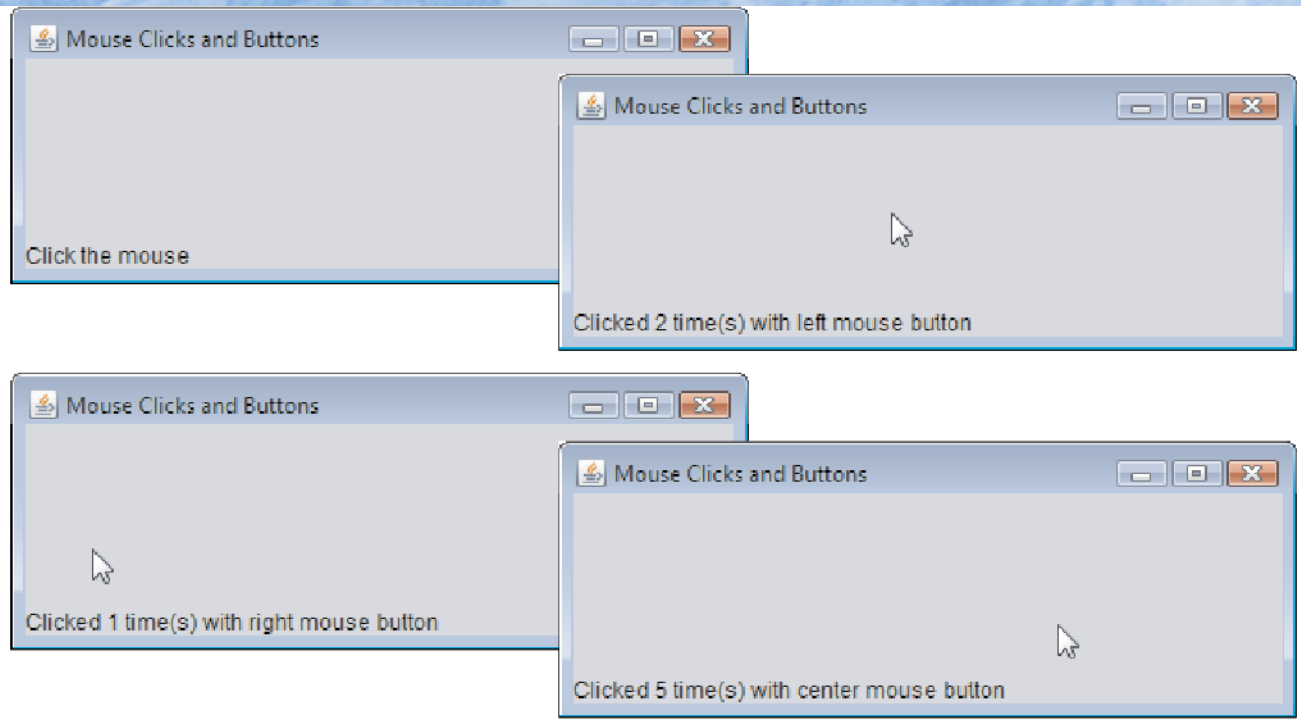


Figura 14.32 | Classe de teste para `MouseDetailsFrame`. (Parte 2 de 2.)

Erro comum de programação 14.4

Se você estender uma classe adaptadora e digitar incorretamente o nome do método que você está sobrescrevendo, o método simplesmente torna-se outro método na classe. Esse é um erro de lógica difícil de ser detectado, visto que o programa chamará a versão vazia do método herdado da classe adaptadora.

- Um mouse pode ter um, dois ou três botões.
- A classe `MouseEvent` herda diversos métodos de `InputEvent` que podem simular um mouse de múltiplos botões com uma combinação de um clique do teclado e um clique de botão do mouse.
- O Java assume que cada mouse contém um botão esquerdo do mouse.

- No caso de um mouse com um ou dois botões, um aplicativo Java assume que o botão do centro do mouse é clicado se o usuário mantém pressionada a tecla *Alt* e clica no botão esquerdo do mouse em um mouse de dois botões ou no botão único do mouse em um mouse de um botão.
- No caso de um mouse com um único botão, um aplicativo Java supõe que o botão direito do mouse é clicado se o usuário mantiver a tecla *Meta* pressionada (às vezes chamada de tecla Command ou tecla "Maçã" em um Mac) e clicar no botão do mouse.

Método `InputEvent`	Descrição
`isMetaDown()`	Retorna `true` quando o usuário clica no botão direito do mouse em um mouse com dois ou três botões. Para simular um clique de botão direito com um mouse de um botão, o usuário pode manter pressionada a tecla *Meta* no teclado e clicar no botão do mouse.
`isAltDown()`	Retorna `true` quando o usuário clica no botão do mouse do meio em um mouse com três botões. Para simular um clique com o botão do meio do mouse em um mouse com um ou dois botões, o usuário pode pressionar a tecla *Alt* e clicar no único botão ou no botão esquerdo do mouse, respectivamente.

Figura 14.33 | Métodos `InputEvent` que ajudam a distinguir entre os cliques do botão esquerdo, do centro e direito do mouse.

- O número de cliques consecutivos de mouse é retornado pelo método `MouseEvent` **getClickCount.**
- Os métodos **isMetaDown** e **isAltDown** determinam em que botão do mouse o usuário clicou.

14.16 Subclasse `JPanel` para desenhar com o mouse

- Utilize um `JPanel` como uma **área dedicada de desenho** em que o usuário pode desenhar arrastando o mouse.
- Os componentes Swing leves que estendem a classe `JComponent` (tal como `JPanel`) contêm o método **paintComponent** chamado quando um componente Swing simples é exibido.
- Sobrescreva esse método para especificar como desenhar.
- Chame a versão de superclasse de `paintComponent` como a primeira instrução no corpo do método sobrescrito para assegurar que o componente será exibido corretamente.

- `JComponent` suporta **transparência.**
- Para exibir um componente corretamente, o programa deve determinar se o componente é transparente.
- O código que determina isso está na implementação `paintComponent` da superclasse `JComponent`.
- Quando um componente é transparente, `paintComponent` não limpará seu fundo.
- Quando um componente é **opaco**, `paintComponent` limpa o fundo do componente.
- A transparência de um componente Swing leve pode ser configurada com o método **setOpaque** (um argumento `false` indica que o componente é transparente).

Java™ COMO PROGRAMAR 8ª edição

Observação sobre a aparência e comportamento 14.12

A maioria dos componentes Swing GUI pode ser transparente ou opaca. Se um componente Swing GUI for opaco, seu fundo será limpo quando seu método `paintComponent` *for chamado. Apenas os componentes opacos podem ser exibidos com uma cor de fundo personalizada. Os objetos* `JPanel` *são opacos por padrão.*

Dica de prevenção de erro 14.1

No método `paintComponent` *de uma subclasse* `JComponent`*, a primeira instrução deve sempre chamar o método da superclasse* `paintComponent` *a fim de assegurar que um objeto da subclasse seja exibido corretamente.*

Erro comum de programação 14.5

Se um método `paintComponent` *sobrescrito não chamar a versão da superclasse, o componente de subclasse pode não ser exibido adequadamente. Se um método* `paintComponent` *sobrescrito chamar a versão da superclasse depois que outro desenho for realizado, o desenho será apagado.*

```
1  // Figura 14.34: PaintPanel.java
2  // Utilizando a classe MouseMotionAdapter.
3  import ons.awt.Point;
4  import ons.awt.Graphics;
5  import java.awt.event.MouseEvent;
6  import java.awt.event.MouseMotionAdapter;
7  import javax.swing.Jpanel;
8
9  public class PaintPanel extends Jpanel
10 {
11    private int pointCount = 0; // número de contagem de pontos
12
13    // array de 10000 referências ons.awt.Point
14    private Point[] points = new Point[ 10000 ];
15
```

Figura 14.34 | Classe adaptadora utilizada para implementar handlers de evento. (Parte 1 de 3.)

```
16      // configura GUI e registra handler de evento de mouse
17      public PaintPanel()
18      {
19         // trata evento de movimento de mouse do frame
20         addMouseMotionListener(
21
22            new MouseMotionAdapter() // classe interna anônima
23            {
24               // armazena coordenadas de arrastar e repinta
25               public void mouseDragged( MouseEvent event )
26               {
27                  if ( pointCount < points.length )
28                  {
29                     points[ pointCount ] = event.getPoint(); // localiza o ponto
30                     pointCount++; // incrementa número de pontos em array
31                     repaint(); // repinta Jframe
32                  } // fim do if
33               } // fim do método mouseDragged
34            } // fim da classe interna anônima
35         ); // fim da chamada para addMouseMotionListener
36      } // fim do construtor PaintPanel
37
```

Armazena pontos a medida que o usuário pressiona o mouse.

Solicita que esse `PaintPanel` seja repintado. Ocasiona uma chamada a `paintComponent`.

Figura 14.34 | Classe adaptadora utilizada para implementar handlers de evento. (Parte 2 de 3.)

```
38       // desenha ovais em um quadro delimitador de 4 por 4 nas localizações especificadas na janela
39       public void paintComponent( Graphics g )
40       {
41          super.paintComponent( g ); // limpa a área de desenho
42
43          // desenha todos os pontos no array
44          for ( int i = 0; i < pointCount; i++ )
45             g.fillOval(points[ i ].x, points[ i ].y, 4, 4 );
46       } // fim do método paintComponent
47    } // fim da classe PaintPanel
```

Desenha um círculo preenchido nas coordenadas especificadas.

Figura 14.34 | Classe adaptadora utilizada para implementar handlers de evento. (Parte 3 de 3.)

- A classe **Point** (pacote `java.awt`) representa uma coordenada *x*-*y*.
- Utilizamos objetos dessa classe para armazenar as coordenadas de cada evento de arrastar com o mouse.
- A classe **Graphics** é utilizada para desenhar.
- O método `MouseEvent` **getPoint** obtém o `Point` em que o evento ocorreu.
- O método **repaint** (herdado indiretamente de `Component`) indica que um `Component` deve ser atualizado na tela o mais rápido possível.

Observação sobre a aparência e comportamento 14.13

Chamar `repaint` *para um componente Swing GUI indica que o componente deve ser atualizado na tela o mais rápido possível. O fundo do componente GUI é limpo somente se o componente for opaco. Para o método* `JComponent` `setOpaque` *pode ser passado um argumento* `boolean` *que indica se o componente é opaco* (`true`) *ou transparente* (`false`).

- O método `Graphics` **fillOval** desenha uma oval sólida.
- Quatro parâmetros do método representam uma área retangular (chamada de quadro delimitador) em que a oval é exibida.
- Os dois primeiros parâmetros são a coordenada *x* superior esquerda e a coordenada *y* superior esquerda da área retangular.
- As duas últimas representam a largura e altura da área retangular.
- O método `fillOval` desenha a oval de tal modo que ela toque no meio de cada lado da área retangular.

Observação sobre a aparência e comportamento 14.14

O desenho em qualquer componente GUI é realizado com as coordenadas que são medidas a partir do canto superior esquerdo (0, 0) desse componente GUI, não o canto superior esquerdo da tela.

```
1  // Figura 14.35: Painter.java
2  // Testando o PaintPanel.
3  import ons.awt.BorderLayout;
4  import javax.swing.Jframe;
5  import javax.swing.Jlabel;
6
7  public class Painter
8  {
9     public static void main( String[] args )
10    {
11       // cria o Jframe
12       Jframe application = new Jframe( "A simple paint program" );
13
14       PaintPanel paintPanel = new PaintPanel(); // cria o painel de pintura
15       application.add( paintPanel, BorderLayout.CENTER ); // no centro
16
17       // cria um rótulo e o coloca em SOUTH do BorderLayout
18       application.add( new Jlabel( "Drag the mouse to draw" ),
19          BorderLayout.SOUTH );
20
21       application.setDefaultCloseOperation( Jframe.EXIT_ON_CLOSE );
22       application.setSize( 400, 200 ); // configura o tamanho do frame
23       application.setVisible( true ); // exibe o frame
24    } // fim de main
25 } // fim da classe Painter
```

Cria uma área de desenho dedicada.

Anexa a área de desenho dedicada ao centro da janela.

Figura 14.35 | Classe de teste para `PaintFrame`. (Parte 1 de 2.)

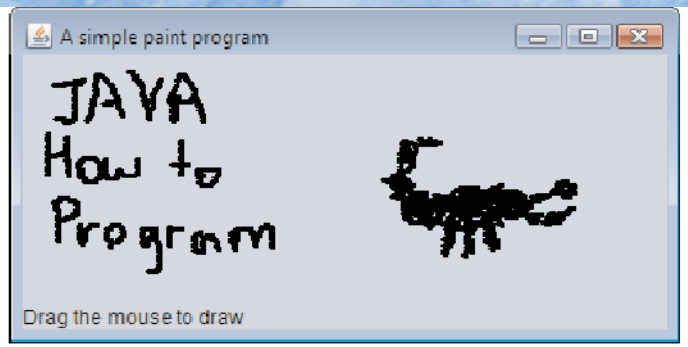


Figura 14.35 | Classe de teste para `PaintFrame`. (Parte 2 de 2.)

14.17 Tratamento de eventos de teclado

- Interface `KeyListener` para tratar **eventos de teclado.**
- Eventos de tecla são gerados quando as teclas do teclado são pressionadas e liberadas.
- Uma `KeyListener` deve definir os métodos **keyPressed**, keyReleased e **keyTyped.** Cada um recebe um `KeyEvent` como seu argumento.
- A classe `KeyEvent` é uma subclasse de `InputEvent`.
- O método `keyPressed` é chamado em resposta ao pressionamento de qualquer tecla.
- O método `keyTyped` é chamado em resposta ao pressionamento de qualquer tecla que não seja uma **action key.**
- O método `keyReleased` é chamado quando a tecla é liberada depois de qualquer evento `keyPressed` ou `keyTyped`.


```
1  // Figura 14.36: KeyDemoFrame.java
2  // Demonstrando os eventos de pressionamento de tecla.
3  import ons.awt.Color;
4  import java.awt.event.KeyListener;
5  import java.awt.event.KeyEvent;
6  import javax.swing.Jframe;
7  import javax.swing.JtextArea;
8
9  public class KeyDemoFrame extends Jframe implements KeyListener
10 {
11    private String line1 = ""; // primeira linha de textarea
12    private String line2 = ""; // segunda linha de textarea
13    private String line3 = ""; // terceira linha de textarea
14    private JtextArea textArea; // textarea a exibir saída
15
16    // construtor KeyDemoFrame
17    public KeyDemoFrame()
18    {
19       super( "Demonstrating Keystroke Events" );
20
21       textArea = new JtextArea( 10, 15 ); // configura JtextArea
22       textArea.setText( "Press any key on the keyboard..." );
23       textArea.setEnabled( false ); // desativa textarea
```

Essa classe pode tratar seus próprios KeyEvents.

Figura 14.36 | Tratamento de eventos de teclado. (Parte 1 de 3.)


```
24          textArea.setDisabledTextColor( Color.BLACK ); // configura a cor do texto
25          add( textArea ); // adiciona textarea ao Jframe
26
27          addKeyListener( this ); // permite que o frame processe os eventos de teclado
28       } // fim do construtor KeyDemoFrame
29
30       // trata pressionamento de qualquer tecla
31       public void keyPressed( KeyEvent event )
32       {
33          line1 = String.format( "Key pressed: %s",
34             KeyEvent.getKeyText( event.getKeyCode() ) ); // mostra a tecla pressionada
35          setLines2and3( event ); // configura a saída das linhas dois e três
36       } // fim do método keyPressed
37
38       // trata liberação de qualquer tecla
39       public void keyReleased( KeyEvent event )
40       {
41          line1 = String.format( "Key released: %s",
42             KeyEvent.getKeyText( event.getKeyCode() ) ); // mostra a tecla liberada
43          setLines2and3( event ); // configura a saída das linhas dois e três
44       } // fim do método keyReleased
45
```

Registra o objeto dessa classe como o handler de evento.

Obtém o texto da tecla pressionada.

Obtém o texto da tecla liberada.

Figura 14.36 | Tratamento de eventos de teclado. (Parte 2 de 3.)

```
46       // trata pressionamento de uma tecla de ação
47       public void keyTyped( KeyEvent event )
48       {
49          line1 = String.format( "Key typed: %s", event.getKeyChar() );
50          setLines2and3( event ); // configura a saída das linhas dois e três
51       } // fim do método keyTyped
52
53       // configura segunda e terceira linhas de saída
54       private void setLines2and3( KeyEvent event )
55       {
56          line2 = String.format( "This key is %san action key",
57             (event.isActionKey() ? "" : "not " ) );
58
59          String temp = KeyEvent.getKeyModifiersText( event.getModifiers() );
60
61          line3 = String.format( "Modifier keys pressed: %s",
62             ( temp.equals( "" ) ? "none" : temp ) ); // modificadores de saída
63
64          textArea.setText( String.format( "%s\n%s\n%s\n",
65             line1, line2, line3 ) ); // gera saída de três linhas de texto
66       } // fim do método setLines2and3
67    } // fim da classe KeyDemoFrame
```

Obtém o texto das teclas modificadoras.

Figura 14.36 | Tratamento de eventos de teclado. (Parte 3 de 3.)


```
1  // Figura 14.37: KeyDemo.java
2  // Testando KeyDemoFrame.
3  import javax.swing.Jframe;
4
5  public class KeyDemo
6  {
7     public static void main( String[] args )
8     {
9        KeyDemoFrame keyDemoFrame = new KeyDemoFrame();
10       keyDemoFrame.setDefaultCloseOperation( Jframe.EXIT_ON_CLOSE );
11       keyDemoFrame.setSize( 350, 100 ); // configura o tamanho do frame
12       keyDemoFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe KeyDemo
```

Figura 14.37 | Classe de teste para KeyDemoFrame. (Parte 1 de 2.)

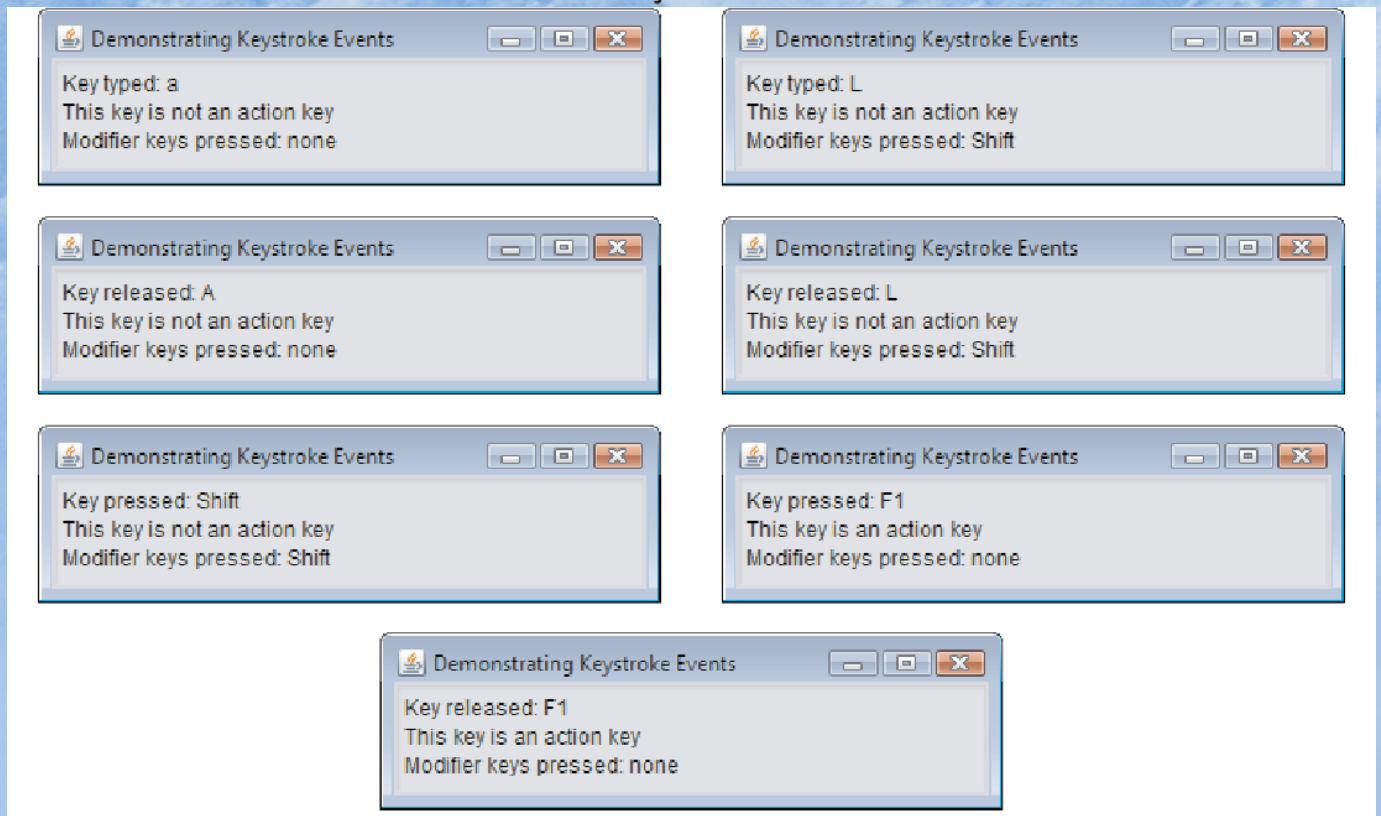


Figura 14.37 | Classe de teste para `KeyDemoFrame`. (Parte 2 de 2.)

- Registra os handlers de evento de teclado com o método **addKeyListener** da classe `Component`.
- O método `KeyEvent` **getKeyCode** obtém o **código de tecla virtual** da tecla pressionada.
- `KeyEvent` contém constantes de código de tecla virtuais que representa cada tecla do teclado.
- O valor retornado por `getKeyCode` pode ser passado ao método `KeyEvent` `static`**getKeyText** para obter uma string que contém o nome da tecla que foi pressionada.
- O método `KeyEvent` **getKeyChar** (que retorna um `char`) obtém o valor Unicode do caractere digitado.
- O método `KeyEvent` **isActionKey** determina se a tecla do evento era uma tecla de ação.

- O método **getModifiers** determina se alguma tecla modificadora (como *Shift*, *Alt* e *Ctrl*) foi pressionada quando o evento de teclado ocorreu.
- O resultado pode ser passado ao método `static` `KeyEvent` **getKeyModifiersText** para obter uma string que contém os nomes das teclas modificadoras pressionadas.
- Os métodos `InputEvent` **isAltDown**, **isControlDown**, **isMetaDown** e **isShiftDown** retornam um `boolean` indicando se a tecla particular foi pressionada durante o evento de teclado.

14.18 Introdução a gerenciadores de layout

- Os **gerenciadores de layout** organizam os componentes GUI em um contêiner para propósitos de apresentação.
- Pode-se utilizá-los para capacidades de layout básicas.
- Permitem que você se concentre na aparência e comportamento básicos — o gerenciador de layout processa os detalhes do layout.
- Os gerenciadores de layout implementam a interface **LayoutManager** (do pacote `java.awt`).
- O método `setLayout` de `Container` aceita um objeto que implementa a interface `LayoutManager` como um argumento.

- Há três maneiras de organizar os componentes em uma GUI:
- Posicionamento absoluto
  - Maior nível de controle.
  - Configure o layout de Container como null.
  - Especifique a posição absoluta de cada componente GUI com relação ao canto superior esquerdo de Container utilizando os métodos Component setSize e setLocation ou setBounds.
  - Deve-se especificar o tamanho de cada componente GUI.

- Gerenciadores de layout.
  - Mais simples e rápidos do que o posicionamento absoluto.
  - Perdem algum controle sobre o tamanho e o posicionamento precisos dos componentes GUI.
- Programação visual em um IDE
  - Utiliza ferramentas que facilitam a criação de GUIs.
  - Permite que você arraste e solte componentes GUI de uma caixa de ferramenta em uma área de desenho.
  - Você então pode posicionar, dimensionar e alinhar componentes GUI como quiser.

Observação sobre a aparência e comportamento 14.15

A maioria dos IDEs Java fornece ferramentas de projeto para projetar visualmente uma GUI; as ferramentas então escrevem o código Java que cria a GUI. Essas ferramentas costumam fornecer maior controle sobre o tamanho, posição e alinhamento de componentes GUI do que os gerenciadores de layouts integrados.

Observação sobre a aparência e comportamento 14.16

É possível configurar o layout de um `Container` *como* `null`*, que indica que nenhum gerenciador de layout deve ser utilizado. Em um* `Container` *sem gerenciador de layout, você deve posicionar e dimensionar os componentes no contêiner dado e tomar o cuidado de que, em eventos de redimensionamento, todos os componentes sejam reposicionados conforme necessário. Os eventos de redimensionamento de um componente podem ser processados por um* `ComponentListener`*.*

Gerenciador de layout	Descrição
`FlowLayout`	Padrão para `javax.swing.Jpanel`. Coloca os componentes sequencialmente (da esquerda para a direita) na ordem em que foram adicionados. Também é possível especificar a ordem dos componentes utilizando o método `Container` add que aceita um `Component` e uma posição de índice do tipo inteiro como argumentos.
`BorderLayout`	Padrão para `JFrames` (e outras janelas). Organiza os componentes em cinco áreas: `NORTH`, `SOUTH`, `EAST`, `WEST` e `CENTER`.
`GridLayout`	Organiza os componentes nas linhas e colunas.

Figura 14.38 | Gerenciadores de layout.

14.18.1 FlowLayout

- `FlowLayout` é o gerenciador de layout mais simples.
- Os componentes GUI são colocados da esquerda para direita na ordem em que são adicionados ao contêiner.
- Quando a borda do contêiner é alcançada, os componentes continuam a ser exibidos na próxima linha.
- `FlowLayout` permite que os componentes GUI sejam alinhados à esquerda, centralizados (o padrão) e alinhados à direita.

Observação sobre a aparência e comportamento 14.17

Cada contêiner individual pode ter apenas um gerenciador de layout, mas vários contêineres no mesmo aplicativo podem utilizar, cada um, gerenciadores de layout diferentes.


```
1  // Figura 14.39: FlowLayoutFrame.java
2  // Demonstrando os alinhamentos FlowLayout.
3  import java.awt.FlowLayout;
4  import java.awt.Container;
5  import java.awt.event.ActionListener;
6  import java.awt.event.ActionEvent;
7  import javax.swing.JFrame;
8  import javax.swing.JButton;
9
10 public class FlowLayoutFrame extends JFrame
11 {
12    private JButton leftJButton; // botão para configurar alinhamento à esquerda
13    private JButton centerJButton; // botão para configurar alinhamento centralizado
14    private JButton rightJButton; // botão para configurar alinhamento à direita
15    private FlowLayout layout; // objeto de layout
16    private Container container; // contêiner para configurar layout
17
18    // configura GUI e registra listeners de botão
19    public FlowLayoutFrame()
20    {
21       super( "FlowLayout Demo" );
22
```

Figura 14.39 | `FlowLayout` permite que os componentes fluam sobre múltiplas linhas. (Parte 1 de 4.)

```
23          layout = new FlowLayout(); // cria FlowLayout
24          container = getContentPane(); // obtém contêiner para layout
25          setLayout( layout ); // configura o layout de frame
26
27          // configura leftJButton e registra listener
28          leftJButton = new JButton( "Left" ); // cria botão Left
29          add( leftJButton ); // adiciona o botão Left ao frame
30          leftJButton.addActionListener(
31
32             new ActionListener() // classe interna anônima
33             {
34                // processa o evento leftJButton
35                public void actionPerformed( ActionEvent event )
36                {
37                   layout.setAlignment( FlowLayout.LEFT );
38
39                   // realinha os componentes anexados
40                   layout.layoutContainer( container );
41                } // fim do método actionPerformed
42             } // fim da classe interna anônima
43          ); // fim da chamada para addActionListener
44
```

Alinha os componentes à esquerda.

Aplica o layout dos componentes do contêiner novamente com base nas alterações de layout.

Figura 14.39 | `FlowLayout` permite que os componentes fluam sobre múltiplas linhas. (Parte 2 de 4.)

```
45          // configura centerJButton e registra o listener
46          centerJButton = new JButton( "Center" ); // cria botão Center
47          add( centerJButton ); // adiciona botão Center ao frame
48          centerJButton.addActionListener(
49
50             new ActionListener() // classe interna anônima
51             {
52                // processa evento centerJButton
53                public void actionPerformed( ActionEvent event )
54                {
55                   layout.setAlignment( FlowLayout.CENTER );
56
57                   // realinha os componentes anexados
58                   layout.layoutContainer( container );
59                } // fim do método actionPerformed
60             } // fim da classe interna anônima
61          ); // fim da chamada para addActionListener
62
63          // configura rightJButton e registra o listener
64          rightJButton = new JButton( "Right" ); // cria botão Right
65          add( rightJButton ); // adiciona botão Right ao frame
```

Alinha os componentes no centro.

Aplica o layout dos componentes do contêiner novamente com base nas alterações de layout.

Figura 14.39 | `FlowLayout` permite que os componentes fluam sobre múltiplas linhas. (Parte 3 de 4.)


```
66          rightJButton.addActionListener(
67
68             new ActionListener() // classe interna anônima
69             {
70                // processo evento rightJButton
71                public void actionPerformed( ActionEvent event )
72                {
73                   layout.setAlignment( FlowLayout.RIGHT );
74
75                   // realinha os componentes anexados
76                   layout.layoutContainer( container );
77                } // fim do método actionPerformed
78             } // fim da classe interna anônima
79          ); // fim da chamada para addActionListener
80       } // fim do construtor FlowLayoutFrame
81    } // FlowLayoutFrame fim da classe
```

Alinha os componentes à direita.

Aplica o layout dos componentes do contêiner novamente baseado nas alterações de layout.

Figura 14.39 | `FlowLayout` permite que os componentes fluam sobre múltiplas linhas. (Parte 4 de 4.)


```
1  // Figura 14.40: FlowLayoutDemo.java
2  // Testando FlowLayoutFrame.
3  import javax.swing.JFrame;
4
5  public class FlowLayoutDemo
6  {
7     public static void main( String[] args )
8     {
9        FlowLayoutFrame flowLayoutFrame = new FlowLayoutFrame();
10       flowLayoutFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       flowLayoutFrame.setSize( 300, 75 ); // configura o tamanho do frame
12       flowLayoutFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe FlowLayoutDemo
```

Figura 14.40 | Classe de teste para `FlowLayoutFrame`. (Parte 1 de 2.)

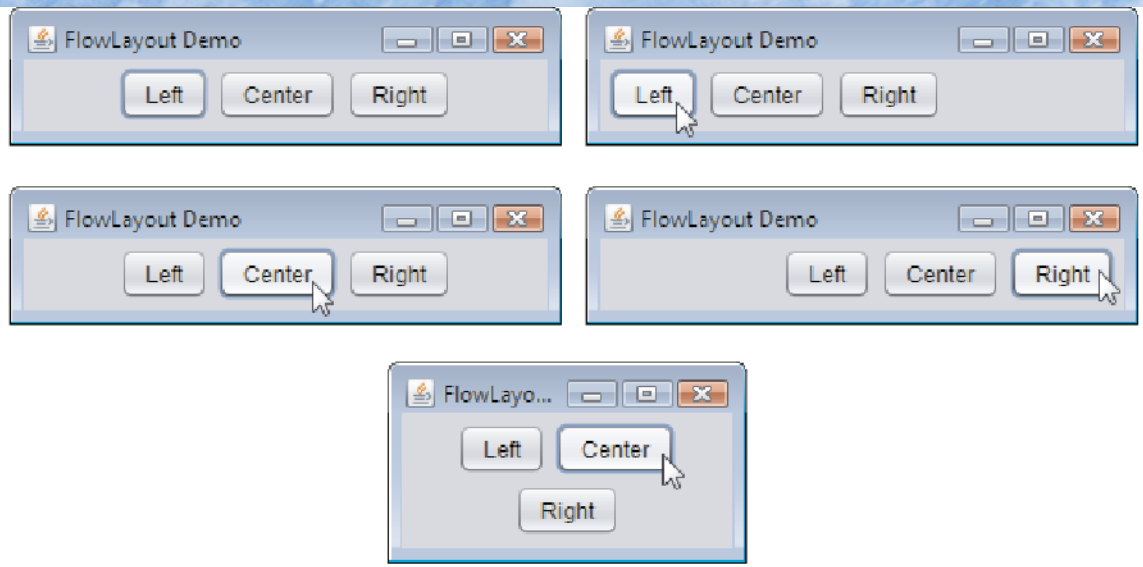


Figura 14.40 | Classe de teste para `FlowLayoutFrame`. (Parte 2 de 2.)

- O método `FlowLayout` **setAlignment** altera o alinhamento do `FlowLayout`.
  - **FlowLayout.LEFT**
  - **FlowLayout.CENTER**
  - **FlowLayout.RIGHT**
- O método de interface `LayoutManager` **layoutContainer** (que é herdado por todos gerenciadores de layout) especifica que o contêiner deve ser reorganizado com base no layout ajustado.

14.18.2 BorderLayout

- `BorderLayout`
- O gerenciador de layout padrão de `Jframe` organiza os componentes em cinco regiões: `NORTH`, `SOUTH`, `EAST`, `WEST` e `CENTER`.
- `NORTH` corresponde à parte superior do contêiner.
- `BorderLayout` implementa a interface **LayoutManager2** (uma subinterface de `LayoutManager` que adiciona vários métodos para o processamento de layout aprimorado).
- `BorderLayout` limita um `Container` a no máximo cinco componentes — um em cada região.
- O componente colocado em cada região pode ser um contêiner ao qual os outros componentes são anexados.

```
1  // Figura 14.41: BorderLayoutFrame.java
2  // Demonstrando BorderLayout.
3  import java.awt.BorderLayout;
4  import java.awt.event.ActionListener;
5  import java.awt.event.ActionEvent;
6  import javax.swing.JFrame;
7  import javax.swing.JButton;
8
9  public class BorderLayoutFrame extends JFrame implements ActionListener
10 {
11    private JButton[] buttons; // array de botões para ocultar partes
12    private static final String[] names = { "Hide North", "Hide South",
13       "Hide East", "Hide West", "Hide Center" };
14    private BorderLayout layout; // objeto borderlayout
15
16    // configura GUI e tratamento de evento
17    public BorderLayoutFrame()
18    {
19       super( "BorderLayout Demo" );
20
21       layout = new BorderLayout( 5, 5 ); // espaços de 5 pixels
22       setLayout( layout ); // configura o layout de frame
23       buttons = new JButton[ names.length ]; // configura o tamanho do array
```

BorderLayout personalizado com espaçamento horizontal e vertical.

Figura 14.41 | BorderLayout que contém cinco botões. (Parte 1 de 3.)

```
24
25          // cria JButtons e registra listeners para eles
26          for ( int count = 0; count < names.length; count++ )
27          {
28             buttons[ count ] = new JButton( names[ count ] );
29             buttons[ count ].addActionListener( this );
30          } // for final
31
32          add( buttons[ 0 ], BorderLayout.NORTH ); // adiciona botão para o norte
33          add( buttons[ 1 ], BorderLayout.SOUTH ); // adiciona botão para o sul
34          add( buttons[ 2 ], BorderLayout.EAST ); // adiciona botão para o leste
35          add( buttons[ 3 ], BorderLayout.WEST ); // adiciona botão para o oeste
36          add( buttons[ 4 ], BorderLayout.CENTER ); // adiciona botão para o centro
37       } // fim do construtor BorderLayoutFrame
38
```

Esse BorderLayoutFrame trata o ActionEvent de cada JButton.

Figura 14.41 | BorderLayout que contém cinco botões. (Parte 2 de 3.)


```
39       // trata os eventos de botão
40       public void actionPerformed( ActionEvent event )
41       {
42          // verifica a origem de evento e o painel de conteúdo de layout correspondentemente
43          for ( JButton button : buttons )
44          {
45             if ( event.getSource() == button )
46                button.setVisible( false ); // oculta o botão clicado
47             else
48                button.setVisible( true ); // mostra outros botões
49          } // for final
50
51          layout.layoutContainer( getContentPane() ); // faz o layout do painel de conteúdo
52       } // fim do método actionPerformed
53    } // fim da classe BorderLayoutFrame
```

Oculta o botão.

Mostra o botão.

Reaplica o layout do contêiner.

Figura 14.41 | `BorderLayout` que contém cinco botões. (Parte 3 de 3.)


```
1  // Figura 14.42: BorderLayoutDemo.java
2  // Testando BorderLayoutFrame.
3  import javax.swing.JFrame;
4
5  public class BorderLayoutDemo
6  {
7     public static void main( String[] args )
8     {
9        BorderLayoutFrame borderLayoutFrame = new BorderLayoutFrame();
10       borderLayoutFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       borderLayoutFrame.setSize( 300, 200 ); // configura o tamanho do frame
12       borderLayoutFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe BorderLayoutDemo
```

Figura 14.42 | Classe de teste para `BorderLayoutFrame`. (Parte 1 de 3.)

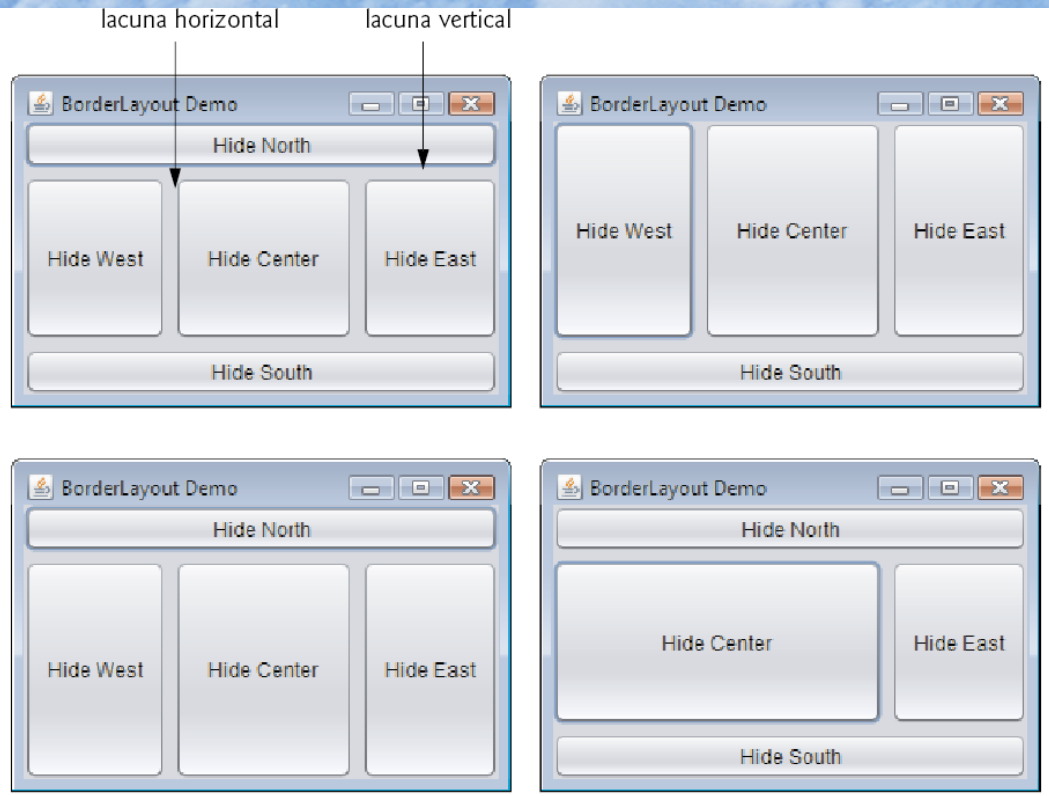


Figura 14.42 | Classe de teste para `BorderLayoutFrame`. (Parte 2 de 3.)

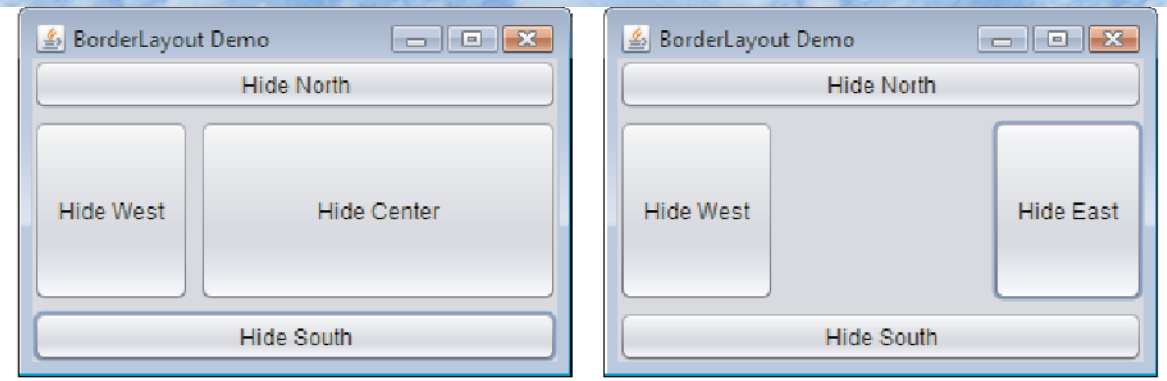


Figura 14.42 | Classe de teste para `BorderLayoutFrame`. (Parte 3 de 3.)

- O construtor BorderLayout especifica o número de pixels entre componentes que estão organizados horizontalmente (espaçamento horizontal) e entre componentes que são organizados verticalmente (espaçamento vertical), respectivamente.
- O padrão é um pixel de espaçamento horizontal e vertical.

Observação sobre a aparência e comportamento 14.18

Se nenhuma região for especificada ao se adicionar um `Component` *a um* `BorderLayout`*, o gerenciador de layout assume que o* `Component` *deve ser adicionado à região* `BorderLayout.CENTER`.

Erro comum de programação 14.6

Quando mais de um componente for adicionado a uma região em um `BorderLayout`*, somente o último componente adicionado a essa região será exibido. Não há nenhum erro que indica esse problema.*

14.18.3 GridLayout

- **GridLayout** divide um contêiner em uma grade de linhas e colunas.
- Implementa a interface `LayoutManager`.
- Todo `Component` têm a mesma largura e altura.
- Os componentes são adicionados iniciando na célula da parte superior esquerda da grade e prosseguindo da esquerda para a direita até a linha estar cheia. Então o processo continua da esquerda para a direita na próxima linha da grade e assim por diante.
- O método `Container` **validate** recalcula o layout do contêiner com base no gerenciador de layout atual e no conjunto atual de componentes GUI exibidos.

```
1  // Figura 14.43: GridLayoutFrame.java
2  // Demonstrando GridLayout.
3  import java.awt.GridLayout;
4  import java.awt.Container;
5  import java.awt.event.ActionListener;
6  import java.awt.event.ActionEvent;
7  import javax.swing.JFrame;
8  import javax.swing.JButton;
9
10 public class GridLayoutFrame extends JFrame implements ActionListener
11 {
12    private JButton[] buttons; // array de botões
13    private static final String[] names =
14       { "one", "two", "three", "four", "five", "six" };
15    private boolean toggle = true; // alterna entre dois layouts
16    private Container container; // contêiner do frame
17    private GridLayout gridLayout1; // primeiro gridlayout
18    private GridLayout gridLayout2; // segundo gridlayout
19
```

Figura 14.43 | GridLayout que contém seis botões. (Parte 1 de 3.)

```
20       // construtor sem argumentos
21       public GridLayoutFrame()
22       {
23          super( "GridLayout Demo" );
24          gridLayout1 = new GridLayout( 2, 3, 5, 5 ); // 2 por 3; lacunas de 5
25          gridLayout2 = new GridLayout( 3, 2 ); // 3 por 2; nenhuma lacuna
26          container = getContentPane(); // obtém painel de conteúdo
27          setLayout( gridLayout1 ); // configura o layout JFrame
28          buttons = new JButton[ names.length ]; // cria array de JButtons
29
30          for ( int count = 0; count < names.length; count++ )
31          {
32             buttons[ count ] = new JButton( names[ count ] );
33             buttons[ count ].addActionListener( this ); // registra listener
34             add( buttons[ count ] ); // adiciona o botão ao JFrame
35          } // for final
36       } // fim do construtor GridLayoutFrame
37
```

GridLayouts personalizados: um com 2 linhas, 3 colunas e espaçamento de 5 pixels entre os componentes e o outro com 3 linhas, duas colunas e o espaçamento padrão.

Figura 14.43 | GridLayout que contém seis botões. (Parte 2 de 3.)


```
38       // trata eventos de botão alternando entre layouts
39       public void actionPerformed( ActionEvent event )
40       {
41          if ( toggle )
42             container.setLayout( gridLayout2 ); // configura layout como segundo
43          else
44             container.setLayout( gridLayout1 ); // configura layout como primeiro
45
46          toggle = !toggle; // alterna para valor oposto
47          container.validate(); // refaz o layout do contêiner
48       } // fim do método actionPerformed
49    } // fim da classe GridLayoutFrame
```

Altera o layout

Reaplica o layout do contêiner.

Figura 14.43 | `GridLayout` que contém seis botões. (Parte 3 de 3.)

```
1  // Figura 14.44: GridLayoutDemo.java
2  // Testando GridLayoutFrame.
3  import javax.swing.JFrame;
4
5  public class GridLayoutDemo
6  {
7     public static void main( String[] args )
8     {
9        GridLayoutFrame gridLayoutFrame = new GridLayoutFrame();
10       gridLayoutFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       gridLayoutFrame.setSize( 300, 200 ); // configura o tamanho do frame
12       gridLayoutFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe GridLayoutDemo
```

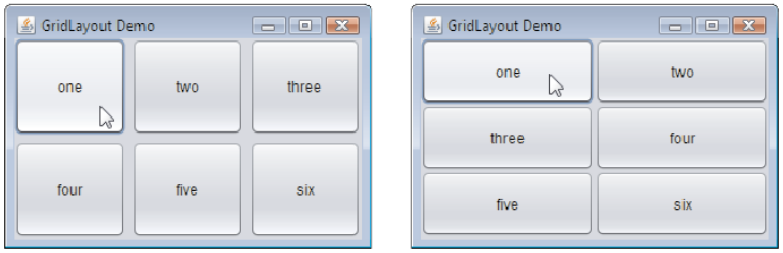


Figura 14.44 | Classe de teste para `GridLayoutFrame`.

14.19 Utilizando painéis para gerenciar layouts mais complexos

- GUIs complexas requerem que cada componente seja colocado em um local exato.
- Frequentemente, consistem em múltiplos painéis, com os componentes de cada painel organizados em um layout específico.
- A classe `JPanel` estende `JComponent` e `JComponent` estende a classe `Container`, portanto todo `JPanel` é um `Container`.
- Todo `JPanel` pode ter componentes, incluindo outros painéis, anexados a ele com o método `Container` `add`.
- `JPanel` pode ser utilizado para criar um layout mais complexo no qual vários componentes estão em uma área específica de outro contêiner.


```
1  // Figura 14.45: PanelFrame.java
2  // Utilizando um JPanel para ajudar a fazer o layout dos componentes.
3  import java.awt.GridLayout;
4  import java.awt.BorderLayout;
5  import javax.swing.JFrame;
6  import javax.swing.JPanel;
7  import javax.swing.JButton;
8
9  public class PanelFrame extends JFrame
10 {
11    private JPanel buttonJPanel; // painel para armazenar botões
12    private JButton[] buttons; // array de botões
13
14    // construtor sem argumentos
15    public PanelFrame()
16    {
17       super( "Panel Demo" );
18       buttons = new JButton[ 5 ]; // cria botões de array
19       buttonJPanel = new JPanel(); // configura painel
20       buttonJPanel.setLayout( new GridLayout( 1, buttons.length ) );
21
```

Figura 14.45 | `JPanel` com cinco `JButtons` anexados à região SOUTH de um `BorderLayout`. (Parte 1 de 2.)

```
22          // cria e adiciona botões
23          for ( int count = 0; count < buttons.length; count++ )
24          {
25             buttons[ count ] = new JButton( "Button " + ( count + 1 ) );
26             buttonJPanel.add( buttons[ count ] ); // adiciona botão ao painel
27          } // for final
28
29          add( buttonJPanel, BorderLayout.SOUTH ); // adiciona painel ao JFrame
30       } // fim do construtor PanelFrame
31    } // fim da classe PanelFrame
```

Coloca o `JPanel` com 5 botões na região South do `BorderLayout` da janela.

Figura 14.45 | `JPanel` com cinco `JButtons` anexados à região SOUTH de um `BorderLayout`. (Parte 2 de 2.)

```
1  // Figura 14.46: PanelDemo.java
2  // Testando PanelFrame.
3  import javax.swing.JFrame;
4
5  public class PanelDemo extends JFrame
6  {
7     public static void main( String[] args )
8     {
9        PanelFrame panelFrame = new PanelFrame();
10       panelFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       panelFrame.setSize( 450, 200 ); // configura o tamanho do frame
12       panelFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe PanelDemo
```

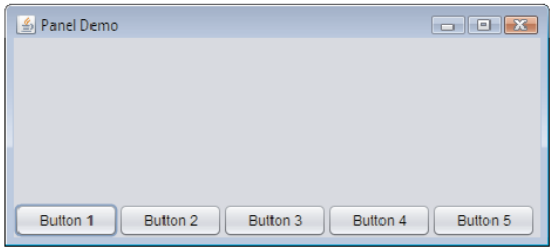


Figura 14.46 | Classe de teste para `PanelFrame`.

14.20 JTextArea

- Uma **JTextArea** fornece uma área para manipular múltiplas linhas de texto.
- `JTextArea` é uma subclasse de `JTextComponent`, que declara métodos comuns para `JTextField`s, `JTextArea`s e vários outros componentes GUI baseados em texto.


```
1  // Figura 14.47: TextAreaFrame.java
2  // Copiando texto selecionado de uma textarea para outra.
3  import java.awt.event.ActionListener;
4  import java.awt.event.ActionEvent;
5  import javax.swing.Box;
6  import javax.swing.JFrame;
7  import javax.swing.JTextArea;
8  import javax.swing.JButton;
9  import javax.swing.JScrollPane;
10
11 public class TextAreaFrame extends JFrame
12 {
13    private JTextArea textArea1; // exibe a string demo
14    private JTextArea textArea2; // texto destacado é copiado aqui
15    private JButton copyJButton; // começa a copiar o texto
16
17    // construtor sem argumentos
18    public TextAreaFrame()
19    {
20       super( "TextArea Demo" );
21       Box box = Box.createHorizontalBox(); // cria box
22       String demo = "This is a demo string to\n" +
23          "illustrate copying text\nfrom one textarea to \n" +
24          "another textarea using an\nexternal event\n";
```

Contêiner que arranja os componentes horizontalmente.

Figura 14.47 | Copiando texto selecionado de uma `JTextArea` para outra. (Parte 1 de 2.)

```
25
26          textArea1 = new JTextArea( demo, 10, 15 ); // cria textarea1
27          box.add( new JScrollPane( textArea1 ) ); // adiciona scrollpane
28
29          copyJButton = new JButton( "Copy >>>" ); // cria botão de cópia
30          box.add( copyJButton ); // adiciona o botão de cópia à box
31          copyJButton.addActionListener(
32
33             new ActionListener() // classe interna anônima
34             {
35                // configura texto em textArea2 como texto selecionado de textArea1
36                public void actionPerformed( ActionEvent event )
37                {
38                   textArea2.setText( textArea1.getSelectedText() );
39                } // fim do método actionPerformed
40             } // fim da classe interna anônima
41          ); // fim da chamada para addActionListener
42
43          textArea2 = new JTextArea( 10, 15 ); // cria segunda textarea
44          textArea2.setEditable( false ); // desativa a edição
45          box.add( new JScrollPane( textArea2 ) ); // adiciona scrollpane
46
47          add( box ); // adiciona box ao frame
48       } // fim do construtor TextAreaFrame
49    } // fim da classe TextAreaFrame
```

Copia o texto selecionado para `textArea2`.

Figura 14.47 | Copiando texto selecionado de uma `JTextArea` para outra. (Parte 2 de 2.)

```
1  // Figura 14.48: TextAreaDemo.java
2  // Copiando texto selecionado de uma textarea para outra.
3  import javax.swing.JFrame;
4
5  public class TextAreaDemo
6  {
7     public static void main( String[] args )
8     {
9        TextAreaFrame textAreaFrame = new TextAreaFrame();
10       textAreaFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11       textAreaFrame.setSize( 425, 200 ); // configura o tamanho do frame
12       textAreaFrame.setVisible( true ); // exibe o frame
13    } // fim de main
14 } // fim da classe TextAreaDemo
```

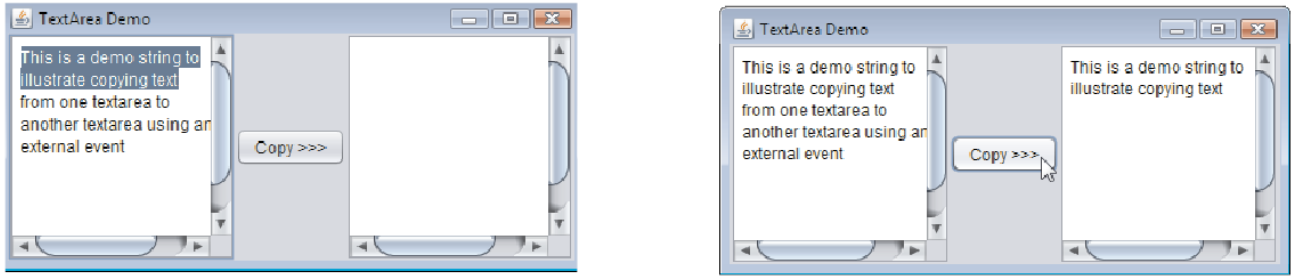


Figura 14.48 | Classe de teste para `TextAreaFrame`.

- Uma **JTextArea** fornece uma área para manipular múltiplas linhas de texto.
- `JTextArea` é uma subclasse de `JTextComponent`.

Observação sobre a aparência e comportamento 14.19

Para fornecer a funcionalidade de mudança de linha automática para uma `JTextArea`*, invoque o método* `JTextArea` **`setLineWrap`** *com um argumento* `true`.

- **Box** é uma subclasse de `Container` que utiliza um **BoxLayout** para organizar os componentes GUI horizontal ou verticalmente.
- O método `static Box` **createHorizontalBox** cria uma `Box` que organiza os componentes da esquerda para a direita na ordem que eles são anexados.
- O método `JTextArea` **getSelectedText** (herdado de `JTextComponent`) retorna o texto selecionado de uma `JTextArea`.
- O método `JTextArea setText` altera o texto de uma `JTextArea`.
- Quando o texto alcançar o canto direito de uma `JTextArea`, o texto pode recorrer para a próxima linha.
- Por padrão, `JTextArea` não muda de linha automaticamente.

- Você pode configurar as **diretivas de barra de rolagem** horizontal e vertical de um `JScrollPane` quando ele é construído.
- Você também pode utilizar os métodos `JScrollPane` **setHorizontalScrollBarPolicy** e **setVerticalScrollBarPolicy** para alterar as diretivas de barra de rolagem.

- A classe `JScrollPane` declara as constantes
  - `JScrollPane.VERTICAL_SCROLLBAR_ALWAYS`
    `JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS`
  - para indicar que uma barra de rolagem sempre deve aparecer e as constantes
    - `JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED`
      `JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS`
  - para indicar que uma barra de rolagem deve aparecer somente se necessário (os padrões) e as constantes
    - `JScrollPane.VERTICAL_SCROLLBAR_NEVER`
      `JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS`
  - para indicar que uma barra de rolagem nunca deve aparecer.
- Se a diretiva for configurada como `HORIZONTAL_SCROLLBAR_NEVER`, uma `JTextArea` anexada ao `JScrollPane` mudará automaticamente de linhas.