

Você acha que posso ouvir essas coisas o dia todo?

— Lewis Carroll

Mesmo um pequeno evento na vida de uma criança é um evento no mundo da criança e, portanto, um evento do mundo.

— Gaston Bachelard

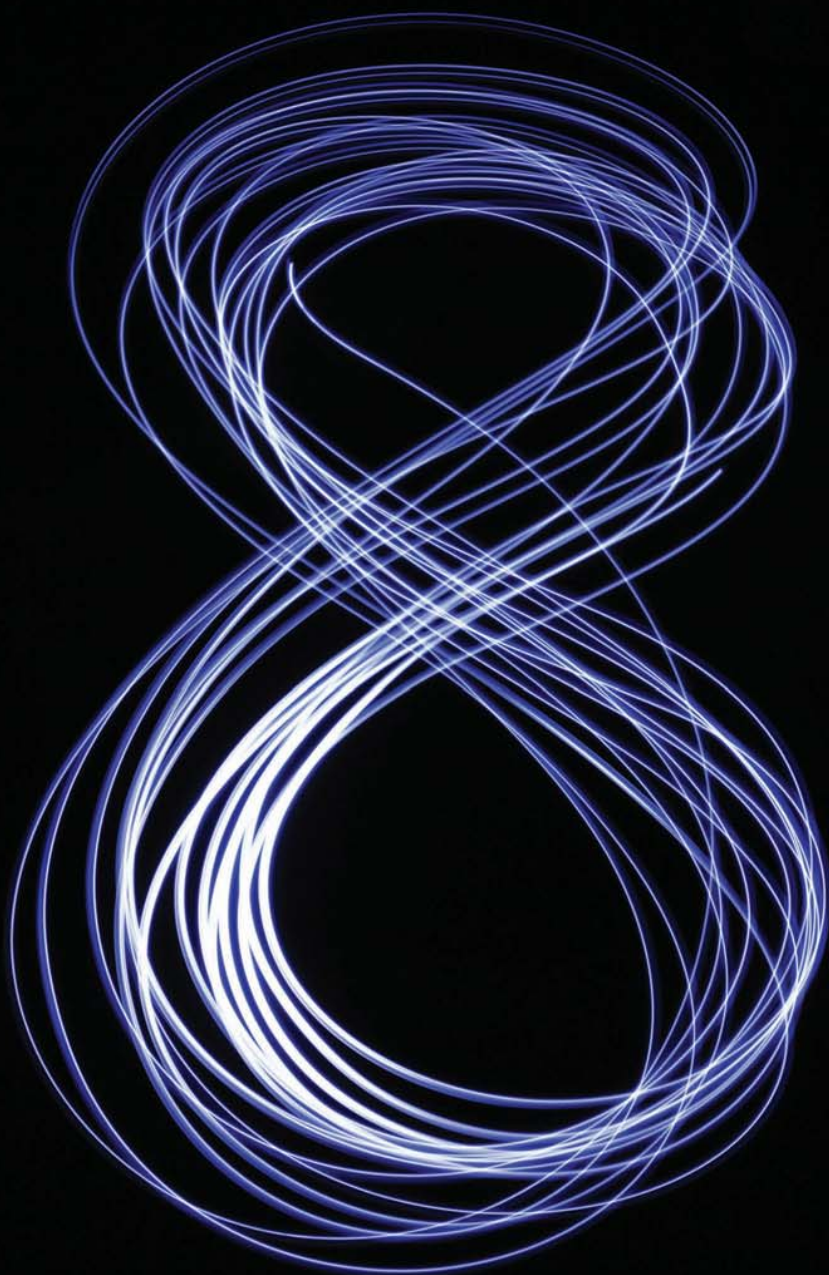
Você paga e faz a sua escolha.

— Punch

Objetivos

Neste capítulo, você irá:

- Entender como usar a aparência e o comportamento Nimbus.
- Construir GUIs e tratar eventos gerados por interações de usuário com GUIs.
- Entender os pacotes que contêm componentes GUI, interfaces e classes de tratamento de evento.
- Criar e manipular botões, rótulos, listas, campos de texto e painéis.
- Lidar com eventos de mouse e eventos de teclado.
- Usar gerenciadores de layout para organizar componentes GUI.



Sumário

- 12.1** Introdução
- 12.2** A nova aparência e comportamento do Java Nimbus
- 12.3** Entrada/saída baseada em GUI simples com JOptionPane
- 12.4** Visão geral de componentes Swing
- 12.5** Exibição de texto e imagens em uma janela
- 12.6** Campos de texto e uma introdução ao tratamento de eventos com classes aninhadas
- 12.7** Tipos comuns de eventos GUI e interfaces ouvintes
- 12.8** Como o tratamento de evento funciona
- 12.9** JButton
- 12.10** Botões que mantêm o estado
 - 12.10.1 JCheckBox
 - 12.10.2 JRadioButton
- 12.11** JComboBox e uso de uma classe interna anônima para tratamento de eventos
- 12.12** JList
- 12.13** Listas de seleção múltipla
- 12.14** Tratamento de evento de mouse
- 12.15** Classes de adaptadores
- 12.16** Subclasse JPanel para desenhar com o mouse
- 12.17** Tratamento de eventos de teclado
- 12.18** Introdução a gerenciadores de layout
 - 12.18.1 FlowLayout
 - 12.18.2 BorderLayout
 - 12.18.3 GridLayout
- 12.19** Utilizando painéis para gerenciar layouts mais complexos
- 12.20** JTextArea
- 12.21** Conclusão

Resumo | Exercícios de revisão | Respostas dos exercícios de revisão | Questões | Fazendo a diferença

12.1 Introdução

Uma **interface gráfica com usuário** (*graphical user interface* — **GUI**) apresenta um mecanismo amigável ao usuário para interagir com um aplicativo. Uma GUI dá ao aplicativo uma “aparência e comportamento” distintos. As GUIs são construídas a partir de **componentes GUI**. Eles às vezes são chamados de *controles* ou *widgets* — abreviação para *window gadgets* (controles de janela). Um componente GUI é um objeto com que o usuário interage por meio do mouse, do teclado ou de outro formulário de entrada, como reconhecimento de voz. Neste capítulo e no Capítulo 22, Componentes GUI: parte 2, você aprenderá sobre muitos dos chamados **componentes GUI Swing** do Java do pacote **javax.swing**. Abrangemos outros componentes GUI conforme necessário ao longo do livro. No Capítulo 25 e na Sala Virtual, você aprenderá sobre o JavaFX — as APIs mais recentes do Java para GUIs, elementos gráficos e multimídia.



Observação sobre a aparência e comportamento 12.1

Fornecer aos diferentes aplicativos componentes de interface com o usuário consistentes e intuitivos permite que os usuários se familiarizem com um novo aplicativo, para que possam aprendê-lo e utilizá-lo mais rápida e produtivamente.

Suporte do IDE para o design de GUI

Muitos IDEs fornecem ferramentas de design de GUI com as quais você pode especificar *tamanho*, *localização* e outros atributos de um componente de uma forma visual usando o mouse, o teclado e arrastar e soltar. Os IDEs geram o código GUI para você. Isso simplifica muito a criação de GUIs, mas cada IDE gera esse código de maneira diferente. Por essa razão, escrevemos o código GUI manualmente, como você verá nos arquivos de código-fonte para os exemplos deste capítulo. Encorajamos você a construir cada GUI visualmente usando seu(s) IDE(s) preferido(s).

GUI de exemplo: o aplicativo de demonstração SwingSet3

Como um exemplo de uma GUI, considere a Figura 12.1, que mostra o aplicativo de demonstração SwingSet3 a partir do download de demos e exemplos do JDK em <http://www.oracle.com/technetwork/java/javase/downloads/index.html>. Esse aplicativo é uma boa maneira de navegar pelos vários componentes GUI fornecidos pelas APIs da GUI Swing do Java. Simplesmente clique em um nome de componente (por exemplo, JFrame, JTabbedPane etc.) na área GUI Components à esquerda da janela para ver uma demonstração do componente GUI à direita da janela. O código-fonte para cada demo é mostrado na área de texto na parte inferior da janela. Rotulamos alguns dos componentes GUI no aplicativo. Na parte superior da janela há uma **barra de título** que contém o título da janela. Abaixo dele há uma **barra de menu** que contém **menus** (File e View). Na região superior direita da janela há um conjunto de **botões** — tipicamente, os usuários clicam em botões para realizar tarefas. Na área GUI Components da janela há uma **caixa de combinação**; o usuário pode clicar na seta para baixo à direita da caixa para selecionar a partir de uma lista de itens. Os menus, botões e caixa de combinação fazem parte da GUI do aplicativo. Eles permitem interagir com o aplicativo.

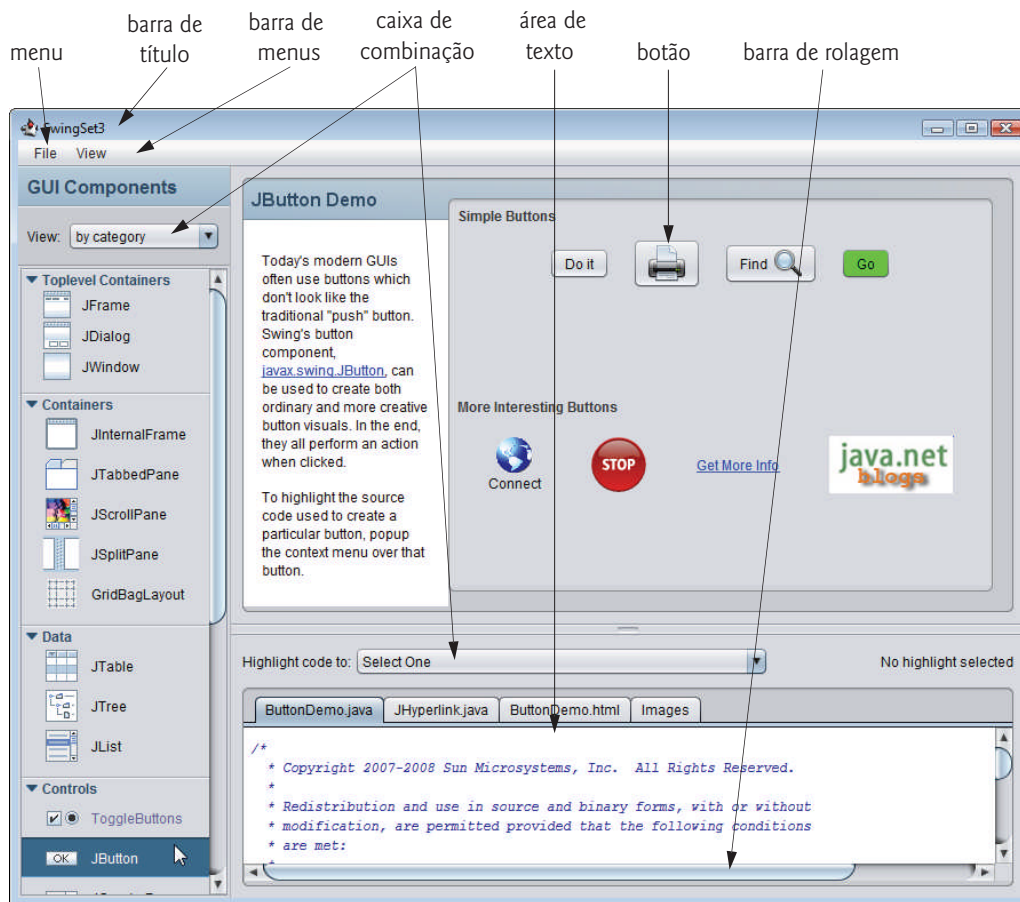


Figura 12.1 | O aplicativo **SwingSet3** demonstra muitos dos componentes GUI Swing do Java.

12.2 A nova aparência e comportamento do Java Nimbus

A aparência de uma GUI consiste nos aspectos visuais, como cores e fontes, e o comportamento, nos componentes que você usa para interagir com a GUI, como botões e menus. Juntos, esses são conhecidos como a aparência e o comportamento da GUI. O Swing tem uma aparência e comportamento multiplataforma conhecidos como **Nimbus**. Para capturas de tela da GUI como a Figura 12.1, configuramos nossos sistemas para usar o Nimbus como a aparência e o comportamento padrão. Há três maneiras de usar o Nimbus:

1. Defini-lo como padrão para todos os aplicativos Java executados no seu computador.
2. Defini-lo como a aparência e o comportamento no momento em que você inicia um aplicativo passando um argumento de linha de comando para o comando `java`.
3. Defini-lo programaticamente como a aparência e o comportamento no seu aplicativo (ver Seção 22.6).

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

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

Além do JRE autônomo, há um JRE aninhado na pasta de instalação do seu JDK. Se estiver utilizando um IDE que depende do JDK, talvez você também precise inserir o arquivo `swing.properties` na pasta `lib` aninhada na pasta `jre`.

Se você preferir selecionar o Nimbus com base em cada aplicativo, insira o seguinte argumento de linha de comando após o comando `java` e antes do nome do aplicativo ao executá-lo:

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

12.3 Entrada/saída baseada em GUI simples com JOptionPane

Os aplicativos nos capítulos 2 a 10 exibem o texto na janela de comando e obtêm a entrada a partir da janela de comando. A maioria dos aplicativos que você usa diariamente utiliza janelas ou **caixas de diálogo** (também chamadas de **diálogos**) para interagir com o usuário. Por exemplo, um programa de e-mail permite digitar e ler mensagens em uma janela que o programa fornece. As caixas de diálogo são janelas em que programas exibem mensagens importantes para o usuário ou obtêm informações do usuário. A classe **JOptionPane** do Java (pacote `javax.swing`) fornece caixas de diálogo pré-construídas tanto para entrada como para saída. Elas são exibidas invocando métodos `JOptionPane` `static`. A Figura 12.2 apresenta um aplicativo de adição simples que 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 12.2: Addition.java
2 // Programa de adição que utiliza JOptionPane para entrada e saída.
3 import javax.swing.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;
20
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    }
25 } // fim da classe Addition

```

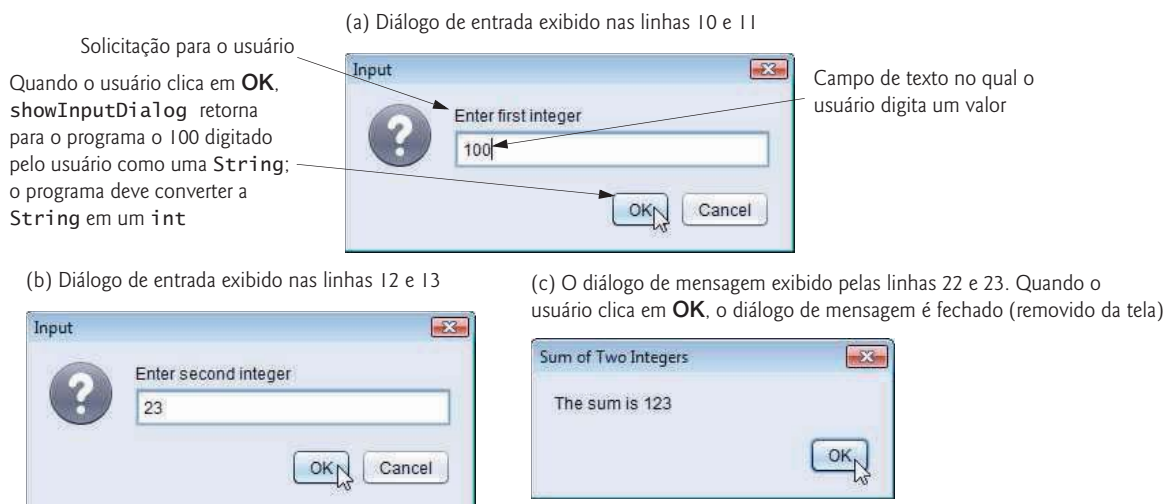


Figura 12.2 | Programa de adição que utiliza `JOptionPane` para entrada e saída.

Diálogos de entrada

A linha 3 importa a classe `JOptionPane`. As linhas 10 e 11 declaram a variável `String` local `firstNumber` e atribuem a ela o resultado da chamada ao método `JOptionPane` static `showInputDialog`. Esse método exibe um diálogo de entrada [ver a captura de tela na Figura 12.2(a)], utilizando o argumento `String` do método ("Enter first integer") como um prompt.



Observação sobre a aparência e comportamento 12.2

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).

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 também **fecha (oculta) o diálogo**. [Observação: se você digitar no campo de texto e não aparecer nada, ative o campo de texto clicando nele com o mouse.] Ao contrário de `Scanner`, que pode ser utilizado para inserir valores de *vários* tipos do usuário no teclado, *um diálogo de entrada pode inserir somente Strings*. Isso é comum na maioria dos componentes GUI. O usuário pode digitar *quaisquer* caracteres no campo de texto da caixa de diálogo de entrada. Nosso programa supõe que o usuário insere um inteiro *válido*. Se o usuário clicar em Cancel, `showInputDialog` retornará `null`. Se o usuário digitar tipos de valor não inteiros ou clicar no botão Cancel na caixa de diálogo de entrada, ocorrerá uma exceção e o programa não funcionará corretamente. As linhas 12 e 13 exibem outro diálogo de entrada que pede que o usuário insira o segundo inteiro. Cada caixa de diálogo `JOptionPane` exibida é uma **caixa de diálogo modal** — enquanto a caixa de diálogo está na tela, o usuário *não pode* interagir com o restante do aplicativo.



Observação sobre a aparência e comportamento 12.3

Não abuse de caixas de diálogo modais, uma vez que elas podem reduzir a usabilidade dos seus aplicativos. Use uma caixa de diálogo modal apenas quando for necessário, para evitar que os usuários interajam com o restante de um aplicativo até que eles fechem a caixa de diálogo.

Convertendo Strings em valores int

Para realizar o cálculo, convertamos as `Strings` que o usuário inseriu em valores `int`. Lembre-se de que o método `parseInt` da classe `Integer` static converte seu argumento `String` em um valor `int` e pode lançar uma `NumberFormatException`. As linhas 16 e 17 atribuem os valores convertidos às variáveis locais `number1` e `number2`, e a linha 19 soma esses valores.

Caixas de diálogo de mensagem

As linhas 22 e 23 utilizam método static `JOptionPane` `showMessageDialog` para exibir um diálogo de mensagem (a última tela da Figura 12.2) que contém a soma. O primeiro argumento ajuda o aplicativo Java a determinar onde *posicionar* a caixa de diálogo. Um diálogo é tipicamente exibido a partir de um aplicativo GUI em uma janela própria. O primeiro argumento referencia essa janela (conhecida como *janela pai*) e faz com que a caixa de diálogo seja exibida no centro em relação ao pai (como foi feito na Seção 12.9). Se o primeiro argumento for `null`, a caixa de diálogo será exibida no *centro* da tela. O segundo argumento é a *mensagem* a exibir — nesse caso, o resultado da concatenação de `String` "The sum is" e do valor de `sum`. O terceiro argumento — "Sum of Two Integers" — é a `String` que deve aparecer na *barra de título* na parte superior da caixa de diálogo. O quarto argumento — `JOptionPane.PLAIN_MESSAGE` — é o *tipo de caixa de diálogo de mensagem a exibir*. O diálogo `PLAIN_MESSAGE` *não* exibe um *ícone* à esquerda da mensagem. A classe `JOptionPane` fornece várias versões sobrecarregadas dos métodos `showInputDialog` e `showMessageDialog`, bem como os métodos que exibem outros tipos de diálogo. Para informações completas, visite <<http://docs.oracle.com/javase/7/docs/api/javax/swing/JOptionPane.html>>.



Observação sobre a aparência e comportamento 12.4

Em geral, a barra de título de uma janela usa 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*).

Constantes de diálogo de mensagem JOptionPane

As constantes que representam os tipos de diálogo de mensagem são mostradas na Figura 12.3. Todos os tipos de diálogo de mensagem exceto `PLAIN_MESSAGE` exibem um ícone à *esquerda* da mensagem. Esses ícones fornecem uma indicação visual da importância da mensagem para o usuário. Um ícone `QUESTION_MESSAGE` é o *ícone padrão* para uma caixa de diálogo de entrada (ver Figura 12.2).





Tipo de diálogo de mensagem	Ícone	Descrição
ERROR_MESSAGE		Indica um erro.
INFORMATION_MESSAGE		Indica uma mensagem informativa.
WARNING_MESSAGE		Alerta de um potencial problema.
QUESTION_MESSAGE		Faz uma pergunta. Normalmente, esse diálogo exige uma resposta, como clicar em um botão Yes ou No.
PLAIN_MESSAGE	Sem ícone	Um diálogo que contém uma mensagem, mas nenhum ícone.

Figura 12.3 | Constantes JOptionPane static para diálogo de mensagem.

12.4 Visão geral de componentes Swing

Embora seja possível realizar entrada e saída usando os diálogos JOptionPane, a maioria dos aplicativos GUI exige interfaces com o usuário mais elaboradas. O restante deste capítulo discute muitos componentes GUI que permitem aos desenvolvedores de aplicações criar GUIs robustas. A Figura 12.4 lista vários componentes básicos da GUI Swing que discutimos.

Componente	Descrição
JLabel	Exibe <i>texto</i> e/ou ícones <i>não editáveis</i> .
TextField	Normalmente <i>recebe entrada</i> do usuário.
Button	Dispara um evento quando o usuário clicar nele com o mouse.
CheckBox	Especifica uma opção que pode <i>ser</i> ou <i>não selecionada</i> .
ComboBox	Uma <i>lista drop-down dos itens</i> a partir dos quais o <i>usuário</i> pode fazer uma <i>seleção</i> .
List	Uma <i>lista dos itens</i> a partir dos quais o usuário pode fazer uma <i>seleção clicando</i> em <i>qualquer um</i> deles. <i>Múltiplos elementos podem</i> ser selecionados.
Panel	Uma área em que os <i>componentes</i> podem ser <i>colocados</i> e <i>organizados</i> .

Figura 12.4 | Alguns componentes GUI Swing básicos.

Swing versus AWT

Há realmente *dois* conjuntos de componentes GUI no Java. Nas versões anteriores do Java, GUIs eram construídas com componentes do **Abstract Window Toolkit (AWT)** no pacote `java.awt`. Eles se pareciam com os componentes GUI nativos da plataforma em que um programa Java executa. Por exemplo, um objeto Button exibido em um programa Java em execução no Microsoft Windows se parece com aqueles em outros aplicativos do *Windows*. No Apple Mac OS X, o Button se parece com aqueles de outros aplicativos do *Mac*. Às vezes, até a maneira como um usuário pode interagir com um componente AWT *difere entre plataformas*. A aparência do componente e a maneira como o usuário interage com ele são conhecidas como sua **aparência e comportamento** (*look-and-feel*).



Observação sobre a aparência e comportamento 12.5

Os componentes GUI Swing permitem especificar uniformemente a aparência e comportamento para o aplicativo em todas as plataformas ou utilizar a aparência e comportamento personalizados de cada plataforma. Um aplicativo pode até mesmo alterá-los durante a execução para permitir aos usuários escolher a aparência e comportamento preferidos.

Componentes GUI leves versus pesados

A maioria dos componentes Swing são **componentes leves** — eles são escritos, manipulados e exibidos completamente no Java. Componentes AWT são **componentes pesados**, porque eles contam com **sistema de janelas** da plataforma local para determinar sua funcionalidade e sua aparência e comportamento. Vários componentes Swing são componentes *pesados*.

Superclasses de componentes GUI leves do Swing

O diagrama de classe UML da Figura 12.5 mostra uma *hierarquia de herança* que contém classes a partir das quais os componentes Swing leves herdam seus atributos e comportamento comuns.

A classe **Component** (pacote `java.awt`) é uma superclasse que declara os recursos comuns dos componentes GUI nos pacotes `java.awt` e `javax.swing`. Qualquer objeto que *é um Container* (pacote `java.awt`) pode ser usado para organizar Components *anexando* os Components ao Container. Containers podem ser colocados em outros Containers para organizar uma GUI.

A classe **JComponent** (pacote `javax.swing`) é uma subclasse de Container. JComponent é a superclasse de todos os componentes *leves* Swing e declara seus atributos e comportamentos comuns. Como JComponent é uma subclasse de Container, todos os componentes Swing leves também são Containers. Alguns recursos comuns suportados por JComponent incluem:

1. Uma **aparência e comportamento plugáveis** para *personalizar* a aparência dos componentes (por exemplo, para uso em plataformas específicas). Você verá um exemplo disso na Seção 22.6.
2. Teclas de atalho (chamadas **mnemônicos**) para acesso direto a componentes GUI pelo teclado. Você verá um exemplo disso na Seção 22.4.
3. Breves descrições do propósito de um componente GUI (chamadas **dicas de ferramenta**) que são exibidas quando o *cursor de mouse é posicionado sobre o componente* por um breve instante. Você verá um exemplo disso na próxima seção.
4. Suporte para a **acessibilidade**, como leitores de tela em braille para deficientes visuais.
5. Suporte para **localização** de interface com o usuário — isto é, personalizar a interface com o usuário para exibir em diferentes linguagens e utilização de convenções culturais locais.

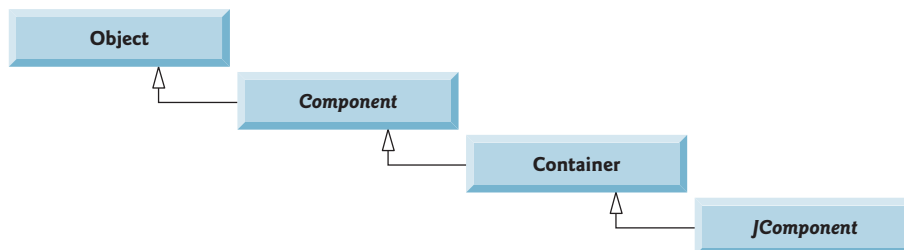


Figura 12.5 | Superclasses comuns dos componentes Swing leves.

12.5 Exibição de texto e imagens em uma janela

Nosso próximo exemplo introduz um framework para construir aplicativos GUI. Vários conceitos nessa estrutura aparecerão em muitos dos nossos aplicativos GUI. Esse é nosso primeiro exemplo em que o aplicativo aparece na própria janela. A maioria das janelas que você criará 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` que fornece os atributos e comportamentos de uma janela — uma *barra de título* no topo e *botões* para *minimizar*, *maximizar* e *fechar* a janela. Visto que, em geral, a GUI de um aplicativo é específica ao aplicativo, a maioria dos nossos exemplos consistirá em *duas* classes — uma subclasse de `JFrame` que ajuda a demonstrar novos conceitos das GUIs e uma classe de aplicativo em que `main` cria e exibe a principal janela do aplicativo.

Rotulando componentes GUI

Uma GUI típica consiste em muitos componentes. Designers de GUI muitas vezes fornecem texto indicando a finalidade de cada um. Esse texto é conhecido como **rótulo** e é criado com um **JLabel** — uma subclasse de `JComponent`. Um `JLabel` exibe texto somente de leitura, uma imagem, ou tanto texto como imagem. Os aplicativos raramente alteram o conteúdo de um rótulo depois de criá-lo.



Observação sobre a aparência e comportamento 12.6

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

O aplicativo das figuras 12.6 e 12.7 demonstra vários recursos `JLabel` e apresenta o framework que utilizamos na maioria de nossos exemplos de GUIs. Não destacamos o código nesse exemplo, visto que a maior parte dele é nova. [Observação: há muito mais recursos para cada componente GUI que podemos abranger em nossos exemplos. Para aprender os detalhes completos de cada componente GUI, visite sua página na documentação on-line. Para a classe `JLabel`, visite docs.oracle.com/javase/7/docs/api/javaw/swing/JLabel.html.]

```

1  // Figura 12.6: LabelFrame.java
2  // JLabels com texto e ícones.
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 final JLabel label1; // JLabel apenas com texto
13     private final JLabel label2; // JLabel construído com texto e ícone
14     private final 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
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     }
42 } // fim da classe LabelFrame

```

Figura 12.6 | JLabels com texto e ícones.

```

1  // Figura 12.7: LabelTest.java
2  // Testando LabelFrame.
3  import javax.swing.JFrame;
4
5  public class LabelTest

```

continua

continuação

```

6  {
7      public static void main(String[] args)
8      {
9          JLabelFrame labelFrame = new JLabelFrame();
10         labelFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11         labelFrame.setSize(260, 180);
12         labelFrame.setVisible(true);
13     }
14 } // fim da classe JLabelTest

```

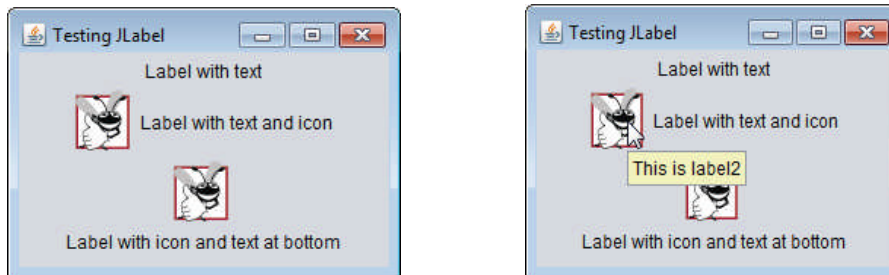


Figura 12.7 | Testando JLabelFrame.

A classe JLabelFrame (Figura 12.6) estende JFrame para herdar os recursos de uma janela. Utilizaremos uma instância da classe JLabelFrame para exibir uma janela contendo três JLabels. As linhas 12 a 14 declaram as três variáveis de instância JLabel que são instanciadas no construtor JLabelFrame (linhas 17 a 41). Em geral, o construtor JFrame da subclasse constrói a GUI que é exibida na janela quando o aplicativo executa. A linha 19 invoca o construtor da superclasse JFrame com o argumento "Testing JLabel". O construtor de JFrame utiliza essa String como o texto na barra de título da janela.

Especificando o layout

Ao construir uma GUI, você deve anexar cada componente GUI a um contêiner, como uma janela criada com um JFrame. Além disso, você normalmente tem de decidir onde *posicionar* cada componente GUI — conhecido como *especificar o layout*. O Java fornece vários **gerenciadores de layout** que podem ajudá-lo a posicionar componentes, como veremos mais adiante neste capítulo e no Capítulo 22.

Muitos IDEs fornecem ferramentas de design de GUI em que você pode especificar visualmente o *tamanho* e *localização* exata de um componente utilizando o mouse; então, o IDE gerará o código GUI para você. Esses IDEs podem simplificar significativamente a criação de GUI.

Para garantir que nossas GUIs podem ser usadas com *qualquer* IDE, *não* utilizamos um IDE para criar o código da GUI. Usamos gerenciadores de layout do Java para *dimensionar* e *posicionar* os componentes. Com o gerenciador de layout **FlowLayout**, os componentes são posicionados em um *contêiner* da esquerda para a direita na ordem em que eles são adicionados. Quando componentes não cabem na linha atual, eles continuam a ser exibidos da esquerda para a direita na linha seguinte. Se o contêiner for *redimensionado*, um FlowLayout *reorganiza* os componentes, possivelmente com mais ou menos linhas com base na nova largura do contêiner. Cada contêiner tem um *layout padrão*, que alteramos de JLabelFrame para um FlowLayout (linha 20). O método **setLayout** é herdado na classe JLabelFrame indiretamente da classe Container. O argumento para o método deve ser um objeto de uma classe que implementa a interface LayoutManager (por exemplo, FlowLayout). A linha 20 cria um novo objeto FlowLayout e passa sua referência como o argumento para setLayout.

Criando e anexando JLabel

Agora que especificamos o layout da janela, podemos começar a criar e anexar componentes GUI à janela. A linha 23 cria um objeto JLabel e passa a "Label with text" para o construtor. O JLabel exibe esse texto na tela. A linha 24 utiliza o método **setToolTipText** (herdado por JLabel de JComponent) para especificar a dica de ferramenta que é exibida quando o usuário posiciona o cursor de mouse sobre o JLabel na GUI. Você pode ver uma dica de ferramenta de exemplo na segunda captura de tela da Figura 12.7. Ao executar esse aplicativo, passe o ponteiro do mouse sobre cada JLabel para ver a dica de ferramenta. A linha 25 (Figura 12.6) anexa label1 ao JLabelFrame passando label1 para o método **add**, que é herdado indiretamente da classe Container.



Erro comum de programação 12.1

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 12.7

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.

A interface Icon e a classe ImageIcon

Os ícones são uma maneira popular de aprimorar a aparência e comportamento de um aplicativo e também são comumente utilizados para indicar funcionalidade. Por exemplo, o mesmo ícone é usado para reproduzir a maioria dos tipos de mídia atuais em dispositivos como leitores de DVD e MP3 players. Vários componentes Swing podem exibir imagens. Um ícone normalmente é especificado com um argumento **Icon** (pacote `javax.swing`) para um construtor ou método **setIcon** do componente. A classe **ImageIcon** suporta vários formatos de imagem, incluindo Graphics Interchange Format (GIF), Portable Network Graphics (PNG) e Joint Photographic Experts Group (JPEG).

A linha 28 declara um **ImageIcon**. O arquivo `bug1.png` contém a imagem para carregar e armazenar no objeto **ImageIcon**. Essa imagem está incluída no diretório desse exemplo. O objeto **ImageIcon** é atribuído à referência `Icon bug`.

Carregando um recurso de imagem

Na linha 28, a expressão `getClass().getResource("bug1.png")` invoca o método **getResource** (herdado indiretamente da classe **Object**) para recuperar uma referência ao objeto **Class** que representa a declaração da classe `LabelFrame`. Essa referência é então utilizada para invocar o método **Class.getResource**, que retorna a localização da imagem como um URL. O construtor **ImageIcon** utiliza o URL para localizar a imagem e, em seguida, carrega essa imagem na memória. Como discutimos no Capítulo 1, a JVM carrega as declarações de classe na memória, utilizando um carregador de classe. 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. Nesse exemplo, o arquivo de imagem é armazenado na mesma localização que o arquivo `LabelFrame.class`. As técnicas descritas aqui permitem que um aplicativo carregue arquivos de imagem a partir de locais que são relativos à localização do arquivo de classe.

Criando e anexando Label2

As linhas 29 e 30 utilizam outro construtor **JLabel** para criar um **JLabel** que exibe o texto "Label with text and icon" e o `Icon bug` criado na linha 28. O último argumento do construtor indica que o conteúdo do rótulo está justificado à esquerda ou alinhado à esquerda (isto é, o ícone e texto estão à esquerda da área do rótulo na tela). A interface **SwingConstants** (pacote `javax.swing`) declara um conjunto de constantes de inteiro comuns (como **SwingConstants.LEFT**, **SwingConstants.CENTER** e **SwingConstants.RIGHT**) que são utilizadas com muitos componentes Swing. Por padrão, o texto aparece à direita da imagem quando um rótulo contém tanto texto como imagem. Observe que os alinhamentos horizontal e vertical de um **JLabel** podem ser configurados com os métodos **setHorizontalAlignment** e **setVerticalAlignment**, respectivamente. A linha 31 especifica o texto de dica de tela para `label2` e a linha 32 adiciona `label2` ao **JFrame**.

Criando e anexando Label3

A classe **JLabel** fornece métodos para alterar a aparência do **JLabel** depois de ele ter sido instanciado. A linha 34 cria um **JLabel** vazio com o construtor sem argumento. A linha 35 utiliza o método **JLabel.setText** para configurar o texto exibido no rótulo. O método **getText** pode ser usado para recuperar o texto atual do **JLabel**. A linha 36 usa o método **JLabel.setIcon** para especificar o `Icon` a exibir. O método **getIcon** pode ser usado para recuperar o `Icon` atual exibido em um rótulo. As linhas 37 e 38 utilizam os métodos **JLabel.setHorizontalTextPosition** e **setVerticalTextPosition** para especificar a posição de texto no rótulo. Nesse caso, o texto será centralizado *horizontalmente* e aparecerá na *parte inferior* do rótulo. Portanto, o `Icon` aparecerá *acima* do texto. As constantes de posição horizontal em **SwingConstants** são **LEFT**, **CENTER** e **RIGHT** (Figura 12.8). As constantes de posição vertical em **SwingConstants** são **TOP**, **CENTER** e **BOTTOM** (Figura 12.8). A linha 39 (Figura 12.6) define o texto das dicas de ferramenta para `label3`. A linha 40 adiciona o `label3` a **JFrame**.

Constante	Descrição	Constante	Descrição
<i>Constantes de posição horizontal</i>		<i>Constantes de posição vertical</i>	
LEFT	Coloca o texto à esquerda	TOP	Coloca o texto na parte superior
CENTER	Coloca o texto no centro	CENTER	Coloca o texto no centro
RIGHT	Coloca o texto à direita	BOTTOM	Coloca o texto na parte inferior

Figura 12.8 | Constantes de posição (membros `static` da Interface **SwingConstants**).

Criando e exibindo uma janela `LabelFrame`

A classe `LabelTest` (Figura 12.7) cria um objeto de classe `LabelFrame` (linha 9) e, em seguida, especifica a operação de fechamento padrão da janela. Por padrão, fechar uma janela simplesmente a *oculta*. Entretanto, quando o usuário fechar a janela `LabelFrame`, queremos que o aplicativo *termine*. A linha 10 invoca o método `setDefaultCloseOperation` de `LabelFrame` (herdado de classe `JFrame`) com a constante `JFrame.EXIT_ON_CLOSE` como o argumento para indicar que o programa deve *terminar* quando a janela for fechada pelo usuário. Essa linha é importante. Sem essa linha o aplicativo *não* terminará quando o usuário fechar a janela. Em seguida, as linhas 11 invocam o método `setSize` de `LabelFrame` para especificar a *largura* e *altura* em *pixels*. Por fim, a linha 12 invoca o método `setVisible` de `LabelFrame` com o argumento `true` para exibir a janela na tela. Tente redimensionar a janela para ver como o `FlowLayout` altera as posições `JLabel` à medida que a largura de janela muda.

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

Normalmente, um usuário interage com uma GUI do aplicativo para indicar as tarefas que o aplicativo deve realizar. Por exemplo, ao escrever um e-mail em um aplicativo de e-mail, clicar no botão `Send` instrui o aplicativo a enviar o e-mail para os endereços de e-mail especificados. 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. Algumas interações de usuário comuns que fazem com que um aplicativo realize uma tarefa incluem *clicar* em um botão, *digitar* em um campo de texto, *selecionar* o item de um menu, *fechar* uma janela e *mover* o mouse. O código que realiza uma tarefa em resposta a um evento é chamado de **rotina de tratamento de evento** e o processo total de responder a eventos é conhecido como **tratamento de evento**.

Vamos considerar dois outros componentes GUI que podem gerar eventos — `JTextFields` e `JPasswordField` (pacote `javax.swing`). A classe `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. Cada um desses componentes é uma área de uma única linha em que o usuário pode inserir texto pelo teclado. Os aplicativos também podem exibir texto em um `JTextField` (ver a saída da Figura 12.10). Um `JPasswordField` mostra que os caracteres estão sendo digitados à medida que o usuário os insere, mas oculta os caracteres reais com um **caractere de eco**, supondo que eles representam uma senha que só deve ser conhecida pelo usuário.

Quando o usuário digita em um `JTextField` ou `JPasswordField`, e depois pressiona `Enter`, um *evento* ocorre. Nosso próximo exemplo demonstra como um programa pode executar uma tarefa *em resposta* a esse evento. Todas as técnicas mostradas aqui são aplicáveis a componentes GUI que geram eventos.

A aplicação das figuras 12.9 e 12.10 utiliza as classes `JTextField` e `JPasswordField` para criar e manipular quatro campos de texto. Quando o usuário digitar em um dos campos de texto, e depois pressionar `Enter`, o aplicativo exibirá uma caixa de diálogo de mensagem que contém o texto que ele digitou. Você pode digitar somente no campo de texto que estiver “em **foco**”. Ao *clicar* em um componente, ele *recebe o foco*. Isso é importante, pois o campo de texto com o foco é o campo que gera um evento quando você pressiona `Enter`. Neste exemplo, quando você pressiona `Enter` no `JPasswordField`, a senha é revelada. Começamos discutindo a configuração da GUI, e depois discutimos o código de tratamento de evento.

```

1 // Figura 12.9: TextFieldFrame.java
2 // JTextField e JPasswordField.
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 final JTextField textField1; // campo de texto com tamanho configurado
14     private final JTextField textField2; // campo de texto com texto
15     private final JTextField textField3; // campo de texto com texto e tamanho
16     private final 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());
23     }

```

continua

```

24 // cria campo de texto com 10 colunas
25 textField1 = new JTextField(10);
26 add(textField1); // adiciona textField1 ao JFrame
27
28 // cria campo de texto com texto padrão
29 textField2 = new JTextField("Enter text here");
30 add(textField2); // adiciona textField2 ao JFrame
31
32 // cria campo de texto com 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 // cria campo de senha com texto padrão
38 passwordField = new JPasswordField("Hidden text");
39 add(passwordField); // adiciona passwordField ao JFrame
40
41 // rotinas de tratamento de evento registradoras
42 TextFieldHandler handler = new TextFieldHandler();
43 textField1.addActionListener(handler);
44 textField2.addActionListener(handler);
45 textField3.addActionListener(handler);
46 passwordField.addActionListener(handler);
47 }
48
49 // classe interna private para tratamento de evento
50 private class TextFieldHandler implements ActionListener
51 {
52     // processa eventos de campo de texto
53     @Override
54     public void actionPerformed(ActionEvent event)
55     {
56         String string = "";
57
58         // usuário pressionou Enter no JTextField textField1
59         if (event.getSource() == textField1)
60             string = String.format("textField1: %s",
61                                     event.getActionCommand());
62
63         // usuário pressionou Enter no JTextField textField2
64         else if (event.getSource() == textField2)
65             string = String.format("textField2: %s",
66                                     event.getActionCommand());
67
68         // usuário pressionou Enter no JTextField textField3
69         else if (event.getSource() == textField3)
70             string = String.format("textField3: %s",
71                                     event.getActionCommand());
72
73         // usuário pressionou Enter no JTextField passwordField
74         else if (event.getSource() == passwordField)
75             string = String.format("passwordField: %s",
76                                     event.getActionCommand());
77
78         // exibe o conteúdo de JTextField
79         JOptionPane.showMessageDialog(null, string);
80     }
81 } // fim da classe TextFieldHandler interna private
82 } // fim da classe TextFieldFrame

```

Figura 12.9 | JTextFields e JPasswordField.

A classe TextFieldFrame estende JFrame e declara três variáveis JTextField e uma variável JPasswordField (linhas 13 a 16). Cada um dos campos de texto correspondentes é instanciado e anexado ao TextFieldFrame no construtor (linhas 19 a 47).

Criando a GUI

A linha 22 define o layout do `TextFieldFrame` como `FlowLayout`. A linha 25 cria `textField1` com 10 colunas de texto. A largura em *pixels* de uma coluna de texto é determinada pela largura média de um caractere na fonte atual do campo de texto. Quando o texto é exibido em um campo de texto e ele for mais largo que o próprio campo de texto, uma parte do texto à direita não é visível. Se você digitar em um campo de texto e o cursor alcançar a borda direita, o texto na borda esquerda é empurrado para fora do lado esquerdo do campo e não mais é visível. Usuários podem utilizar as teclas de seta para a esquerda e para a direita a fim de percorrer todo o texto. A linha 26 adiciona o `textField1` a `JFrame`.

A linha 29 cria `textField2` com o texto inicial "Enter text here" para exibir no campo de texto. A largura do campo é determinada pela largura do texto padrão especificado no construtor. A linha 30 adiciona o `textField2` a `JFrame`.

A linha 33 cria `textField3` e chama o construtor `JTextField` com dois argumentos — o texto padrão "Uneditable text field" a ser exibido e a largura dos campos de texto em número de colunas (21). A linha 34 utiliza o método `setEditable` (herdado por `JTextField` da classe `JTextComponent`) para tornar o campo de texto *não editável* — isto é, o usuário não pode modificar o texto no campo. A linha 35 adiciona o `textField3` a `JFrame`.

A linha 38 cria `passwordField` com o texto "Hidden text" a ser exibido no campo de texto. A largura do campo é determinada pela largura do texto padrão. Ao executar o aplicativo, note que o texto é exibido como uma string de asteriscos. A linha 39 adiciona o `passwordField` a `JFrame`.

Passos necessários para configurar o tratamento de evento para um componente GUI

Esse exemplo deve exibir um diálogo de mensagem que contém o texto de um campo de texto quando o usuário pressionar Enter nesse campo de texto. Antes que um aplicativo possa responder a um evento para um determinado componente GUI, você deve:

1. Criar uma classe que representa a rotina de tratamento de evento e implementa uma interface apropriada — conhecida como **interface ouvinte de evento**.
2. Indicar que um objeto da classe do Passo 1 deve ser notificado quando o evento ocorre — conhecido como **registrar a rotina de tratamento de evento**.

Utilizando uma classe aninhada para implementar uma rotina de tratamento 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 — elas são chamadas de **classes aninhadas**. As classes aninhadas podem ser `static` ou não `static`. As classes não `static` aninhadas são chamadas **classes internas** e são frequentemente utilizadas para implementar *rotina de tratamento de evento*.

Um objeto da classe interna deve ser criado por um objeto da classe de primeiro nível que contém a classe interna. Cada objeto da classe interna tem *implicitamente* uma referência a um objeto da classe de primeiro nível. O objeto da classe interna pode usar essa referência implícita para acessar diretamente todas as variáveis e métodos da classe de primeiro nível. 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. Como veremos no Capítulo 13, "Imagens gráficas e Java 2D", a API dos elementos gráficos Java 2D usa as classes aninhadas `static` extensivamente.

Classe interna `TextFieldHandler`

O tratamento de evento nesse exemplo é realizado por um objeto da *classe interna* `private TextFieldHandler` (linhas 50 a 81). Essa classe é `private` porque será utilizada apenas para criar rotina de tratamento de evento para os campos de texto na classe de primeiro nível `TextFieldFrame`. Tal como acontece com outros membros de classe, as *classes internas* podem ser declaradas `public`, `protected` ou `private`. Como rotinas de tratamento de evento tendem a ser específicas para o aplicativo em que elas são definidas, muitas vezes elas são implementadas como classes internas `private` ou como *classes internas anônimas* (Seção 12.11).

Componentes GUI podem gerar muitos eventos em resposta às interações do usuário. Cada evento é representado por uma classe e pode ser processado apenas pelo tipo de rotina de tratamento de evento apropriado. Normalmente, eventos suportados por um componente estão descritos na documentação da API Java para a classe e superclasses desse componente. Quando o usuário pressiona Enter em um `JTextField` ou `JPasswordField`, ocorre um **ActionEvent** (pacote `java.awt.event`). Um evento assim é processado por um objeto que implementa a interface **ActionListener** (pacote `java.awt.event`). As informações discutidas aqui são disponíveis na documentação de Java API das classes `JTextField` e `ActionEvent`. Visto que `JPasswordField` é uma subclasse de `JTextField`, `JPasswordField` suporta os mesmos eventos.

A fim de se preparar para tratar os eventos nesse exemplo, a classe interna `TextFieldHandler` implementa a interface `ActionListener` e declara o único método nessa interface — `actionPerformed` (linhas 53 a 80). Esse método especifica as tarefas a serem realizadas quando ocorrer um `ActionEvent`. Assim, a classe interna `TextFieldHandler` satisfaz o *Passo 1* listado anteriormente nesta seção. Discutiremos os detalhes do método `actionPerformed` em breve.

Registrando a rotina de tratamento de evento para cada campo de texto

No construtor `TextFieldFrame`, a linha 42 cria um objeto `TextFieldHandler` e o atribui à variável `handler`. O método `actionPerformed` desse objeto será chamado automaticamente quando o usuário pressionar *Enter* em qualquer um dos campos de texto da GUI. Entretanto, antes que isso possa ocorrer, o programa deve registrar esse objeto como a rotina de tratamento de evento de cada campo de texto. As linhas 43 a 46 são as instruções de *registro de evento* que especificam `handler` como a rotina de tratamento de evento para os três `JTextFields` e o `JPasswordField`. O aplicativo chama o método `JTextField` `addActionListener` para registrar a rotina de tratamento de evento para cada componente. Esse método recebe como seu argumento um objeto `ActionListener`, que pode ser um objeto de qualquer classe que implemente `ActionListener`. O objeto `handler` é um `ActionListener`, porque a classe `TextFieldHandler` implementa `ActionListener`. Depois que as linhas 43 a 46 são executadas, o objeto `handler` **ouve eventos**. Agora, quando o usuário pressiona *Enter* em qualquer desses quatro campos de texto, o método `actionPerformed` (linhas 53 a 80) na classe `TextFieldHandler` é chamado para tratar o evento. Se uma rotina de tratamento de evento *não* é registrada para um campo de texto particular, o evento que ocorre quando o usuário pressiona *Enter* nesse campo de texto é **consumido** — isto é, é simplesmente *ignorado* pelo aplicativo.



Observação de engenharia de software 12.1

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



Erro comum de programação 12.2

Se você esquecer de registrar um objeto tratador de eventos para um tipo de evento de um componente GUI particular, eventos desse tipo serão ignorados.

Detalhes do método `actionPerformed` da classe `TextFieldHandler`

Nesse exemplo, usamos um método `actionPerformed` do objeto de tratamento de evento (linhas 53 a 80) para lidar com os eventos gerados pelos quatro campos de texto. Visto que gostaríamos de gerar saída do nome de variável de instância de cada campo de texto para propósitos de demonstração, devemos determinar *qual* campo de texto gerou o evento toda vez que `actionPerformed` for chamado. A **origem do evento** é o componente com o qual o usuário interage. Quando o usuário pressiona *Enter* enquanto um campo de texto ou de senha *tiver o foco*, o sistema cria um objeto `ActionEvent` único que contém informações sobre o evento que acabou de ocorrer, como a origem de evento e o texto no campo de texto. O sistema passa esse objeto `ActionEvent` para o método `actionPerformed` do ouvinte de evento. A linha 56 declara a `String` que será exibida. A variável é inicializada com a **string vazia** — uma `String` que não contém nenhum caractere. O compilador requer que a variável seja inicializada caso nenhuma das ramificações da instrução `if` aninhada nas linhas 59 a 76 seja executada.

Um método `ActionEvent` `getSource` (chamado nas linhas 59, 64, 69 e 74) retorna uma referência à fonte do evento. A condição na linha 59 pergunta, “`textField1` é a origem de evento?” Essa condição compara referências com o operador `==` para determinar se elas referenciam o mesmo objeto. Se *ambas* referenciarem `textField1`, o usuário pressionou *Enter* em `textField1`. Então, as linhas 60 e 61 criam uma `String` contendo a mensagem que a linha 79 exibe em uma caixa de diálogo de mensagem. A linha 61 utiliza o método `getActionCommand` de `ActionEvent` para obter o texto que o usuário digitou no campo de texto que gerou o evento.

Nesse exemplo, exibimos o texto da senha no `JPasswordField` quando o usuário pressiona *Enter* nesse campo. Às vezes é necessário processar programaticamente os caracteres em uma senha. A classe `JPasswordField` do método `getPassword` retorna os caracteres da senha como um array do tipo `char`.

Classe `TextFieldTest`

A classe `TextFieldTest` (Figura 12.10) contém o método `main`, que executa esse aplicativo e exibe um objeto da classe `TextFieldFrame`. Ao executar o aplicativo, mesmo o `JTextField` não editável (`textField3`) pode gerar um `ActionEvent`. Para testar isso, clique no campo de texto para colocá-lo em foco, depois pressione *Enter*. Também, o texto real da senha é exibido quando você pressionar *Enter* no `JPasswordField`. Naturalmente, você em geral não exibiria a senha!

```
1 // Figura 12.10: TextFieldTest.java
2 // Testando TextFieldFrame.
3 import javax.swing.JFrame;
4
5 public class TextFieldTest
6 {
```

continua

continuação

```

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);
12      textFieldFrame.setVisible(true);
13  }
14  } // fim da classe TextFieldTest

```

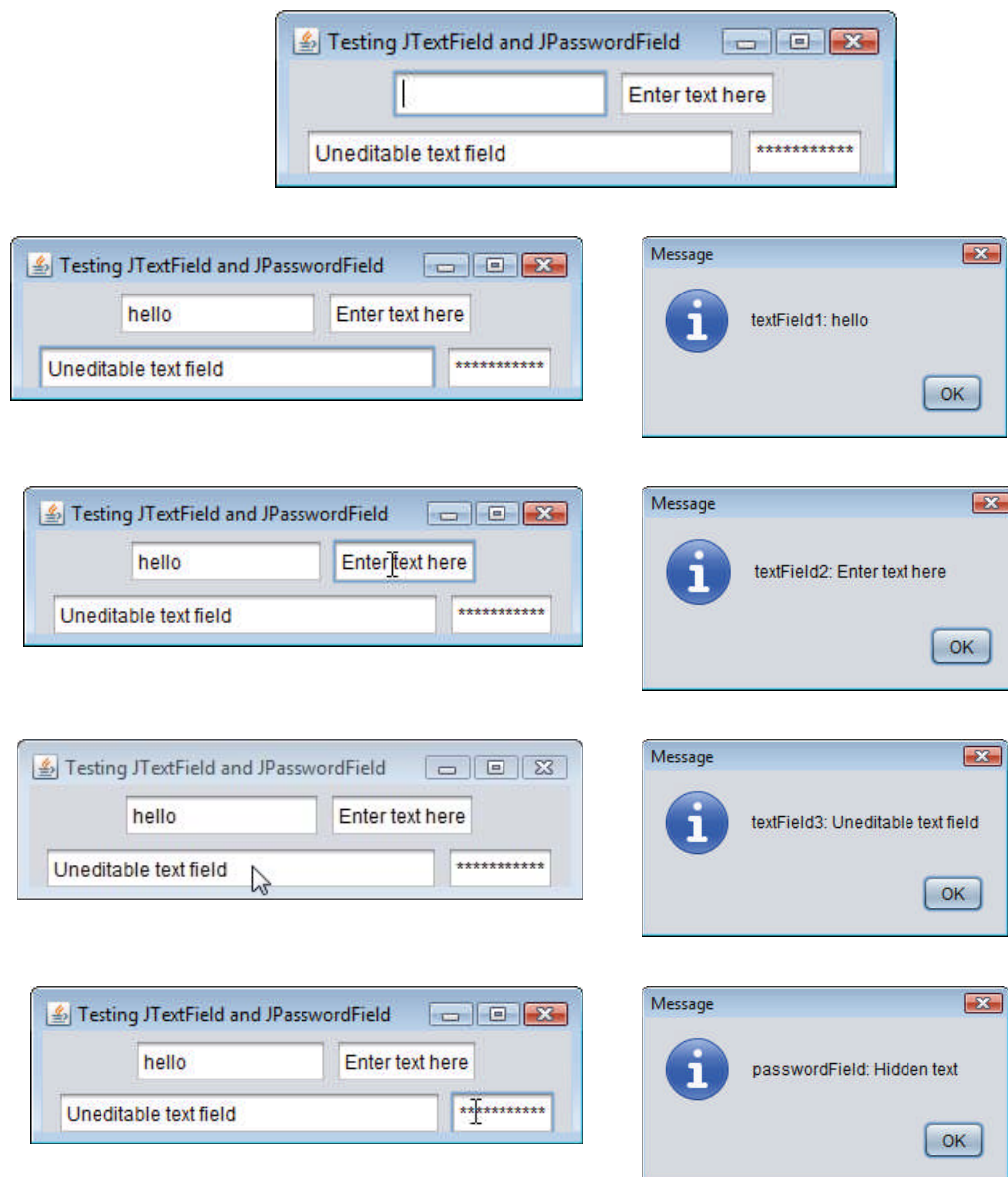


Figura 12.10 | Testando TextFieldFrame.

Esse aplicativo utilizou um único objeto de classe `TextFieldHandler` como o ouvinte, ou *listener*, de eventos para quatro campos de texto. Iniciando na Seção 12.10, você verá que é possível declarar vários objetos ouvintes de evento do mesmo tipo e registrar cada objeto para o evento de um componente GUI separado. Essa técnica permite eliminar a lógica `if...else` utilizada na rotina de tratamento de evento desse exemplo fornecendo rotinas de tratamento de evento separadas para os eventos de cada componente.

Java SE 8: implementando ouvintes de evento com lambdas

Lembre-se de que interfaces como `ActionListener` que têm um único método abstract são interfaces funcionais no Java SE 8. Na Seção 17.9, mostramos uma maneira mais concisa de implementar essas interfaces ouvinte de evento com lambdas Java SE 8.

12.7 Tipos comuns de eventos GUI e interfaces ouvintes

Na Seção 12.6, você aprendeu que as informações sobre o evento que ocorre quando o usuário pressiona *Enter* em um campo de texto são armazenadas em um objeto `ActionEvent`. Muitos tipos diferentes de evento podem ocorrer quando o usuário interage com uma GUI. As informações do evento são armazenadas em um objeto de uma classe que estende `AWTEvent` (do pacote `java.awt`). A Figura 12.11 ilustra uma hierarquia que contém muitas classes de evento do pacote `java.awt.event`. Alguns deles são discutidos neste capítulo e no Capítulo 22. Esses tipos de evento são utilizados tanto com componentes AWT como com Swing. Tipos adicionais de evento que são específicos dos componentes Swing GUI são declarados no pacote `javax.swing.event`.

Vamos resumir as três partes para o mecanismo de tratamento de evento que você viu na Seção 12.6 — a *origem do evento*, o *objeto do evento* e o *ouvinte de eventos*. A origem do evento é o componente GUI com o qual o usuário interage. O objeto do evento encapsula informações sobre o evento que ocorreu, como uma referência à origem do evento e quaisquer informações específicas do evento que podem ser exigidas pelo ouvinte de eventos para tratar o evento. O ouvinte de eventos é um objeto que é notificado pela origem de evento quando um evento ocorre; de fato, ele "ouve" um evento e um de seus métodos executa em resposta ao evento. Um método do ouvinte de eventos recebe um objeto do evento quando o ouvinte de eventos é notificado do evento. O ouvinte de eventos então utiliza o objeto de evento para responder ao evento. O modelo de tratamento de evento descrito aqui é conhecido como **modelo de delegação de evento** — o processamento de um evento é delegado a um objeto particular (o ouvinte de eventos) no aplicativo.

Para cada tipo de objeto de evento, há em geral uma interface ouvinte (ou interface *listener*) de eventos correspondentes. Um ouvinte de evento para um evento GUI é um objeto de uma classe que implementa uma ou mais das interfaces ouvintes de evento dos pacotes `java.awt.event` e `javax.swing.event`. Muitos dos tipos de ouvinte de evento são comuns aos componentes Swing e AWT. Esses tipos são declarados no pacote `java.awt.event` e alguns deles são mostrados na Figura 12.12. Tipos adicionais de ouvinte de evento que são específicos para componentes Swing são declarados no pacote `javax.swing.event`.

Cada interface *listener* de eventos especifica um ou mais métodos de tratamento de evento que *devem* ser declarados na classe que implementa a interface. A partir da Seção 10.9, lembre-se de que qualquer classe que implementa uma interface deve declarar *todos* os métodos *abstract* dessa interface; caso contrário, a classe é *abstract* e não pode ser utilizada para criar objetos.

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. Por exemplo, quando o usuário pressiona a tecla *Enter* em um `TextField`, o método `actionPerformed` do ouvinte registrado é chamado. Na próxima seção, completaremos nossa discussão de como o tratamento de evento funciona no exemplo anterior.

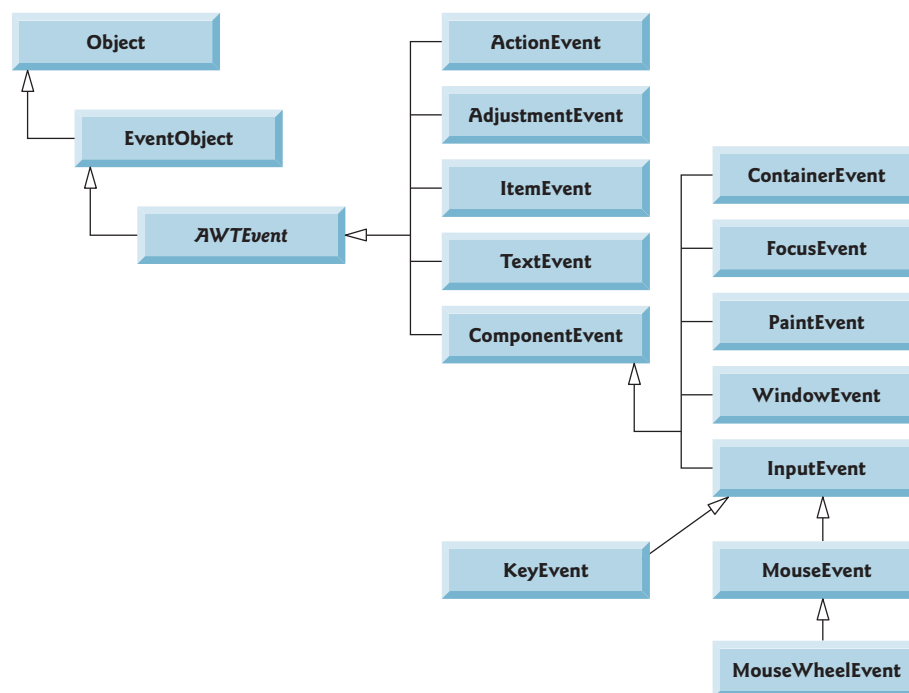


Figura 12.11 | Algumas classes de evento do pacote `java.awt.event`.

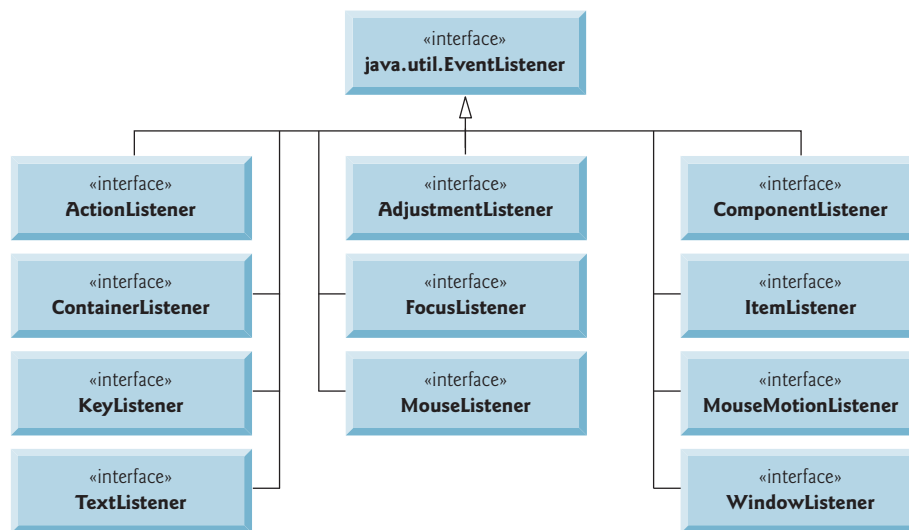


Figura 12.12 | Algumas interfaces listener de eventos comuns do pacote `java.awt.event`.

12.8 Como o tratamento de evento funciona

Ilustraremos como o mecanismo de tratamento de evento funciona, usando `textField1` do exemplo da Figura 12.9. Temos duas perguntas abertas remanescentes da Seção 12.7:

1. Como a rotina de tratamento de evento é registrada?
2. Como o componente GUI sabe chamar `actionPerformed` em vez de algum outro método de tratamento de evento?

A primeira pergunta é respondida pelo registro de evento realizado nas linhas 43 a 46 da Figura 12.9. A Figura 12.13 diagrama a variável `textField1` de `JTextField`, a variável `handler` de `TextFieldHandler` e os objetos que elas referenciam.

Registrando eventos

Cada `JComponent` tem uma variável de instância chamada `listenerList` que referencia um objeto da classe `EventListenerList` (pacote `javax.swing.event`). Cada objeto de uma subclasse `JComponent` mantém referências a seus *ouvintes* (listeners) *registrados* na `listenerList`. Para simplificar, diagramamos `listenerList` como um array abaixo do objeto `JTextField` na Figura 12.13.

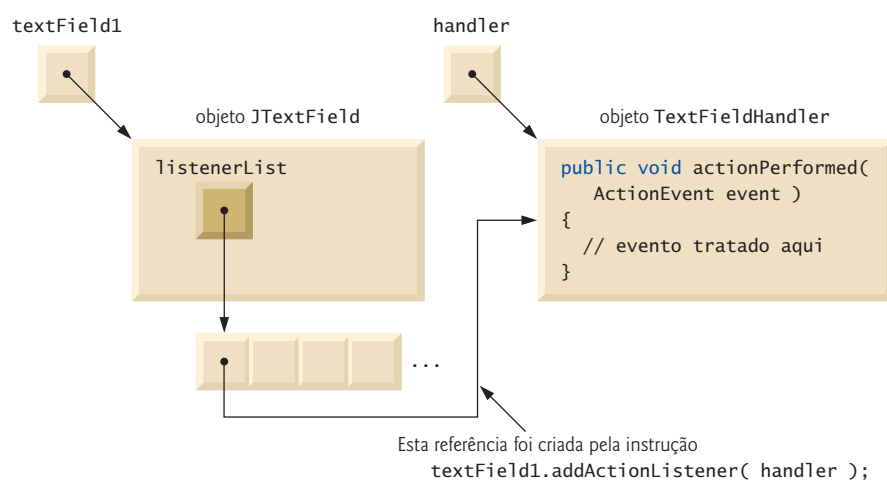


Figura 12.13 | Registro de evento para `JTextField textField1`.

Quando a seguinte instrução (linha 43 da Figura 12.9) é executada

```
textField1.addActionListener(handler);
```

uma nova entrada que contém uma referência ao objeto `TextFieldHandler` é colocada em `listenerList` de `textField1`. Embora não mostrado no diagrama, essa nova entrada também inclui o tipo do listener (nesse caso, `ActionListener`). Utilizando esse mecanismo, cada componente Swing leve mantém sua própria lista de *ouvintes* que foram *registrados* para *tratar* os *eventos* do componente.

Invocação de rotinas de tratamento de evento

O tipo ouvinte de eventos é importante ao responder a segunda pergunta: como o componente GUI sabe chamar `actionPerformed` em vez de outro método? Cada 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. O 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 ouvintes de eventos* correspondentes. Por exemplo, `ActionEvents` são tratados por `ActionListeners`, `MouseEvents` por `MouseListeners`, `MouseMotionListeners` e `KeyEvents` por `KeyListeners`. Quando ocorre um evento, o componente GUI recebe (de 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 *listener*. Para um `ActionEvent`, o evento é despachado para o método `ActionListener` de *cada* `actionPerformed` registrado (o único método na interface `ActionListener`). Para um `MouseEvent`, o evento é despachado para *cada* `MouseListener` ou `MouseMotionListener` registrado, dependendo do evento de mouse que ocorre. O ID de evento do `MouseEvent` determina quais dos vários métodos de tratamento de evento de mouse são chamados. Todas essas decisões são tratadas para você pelos componentes GUI. Tudo que você precisa fazer é registrar uma rotina de tratamento de evento para o tipo particular de evento que seu aplicativo exige e o componente GUI assegurará que o método apropriado da rotina de tratamento de evento é chamado quando o evento ocorre. Discutimos outros tipos de evento e interfaces ouvintes de evento à medida que eles se tornarem necessários com cada novo componente que apresentarmos.



Dica de desempenho 12.1

GUIs sempre devem permanecer responsivas ao usuário. Realizar uma tarefa de longa duração em uma rotina de tratamento de evento evita que o usuário interaja com a GUI até que a tarefa seja concluída. A Seção 23.11 demonstra as técnicas para evitar esses problemas.

12.9 JButton

Um **botão** é um componente em que o usuário clica para acionar uma ação específica. Um aplicativo Java pode utilizar vários tipos de botão, incluindo **botões de comando**, **caixas de seleção**, **botões de alternância** e **botões de opção**. A Figura 12.14 mostra a hierarquia de herança dos botões do Swing que abordaremos neste capítulo. Como você pode ver, todos os tipos de botão são subclasses de `AbstractButton` (pacote `javax.swing`), que declara os recursos comuns de botões Swing. Nesta seção, nos concentramos nos botões que são em geral utilizados para iniciar um comando.

Um *botão de comando* (ver a saída da Figura 12.16) 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**.

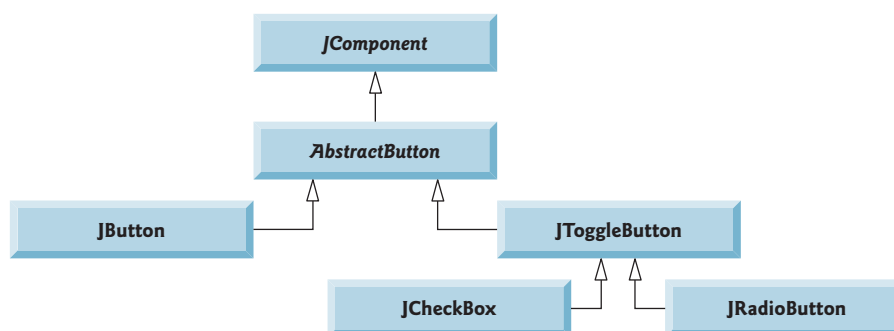


Figura 12.14 | Hierarquia do botão Swing.

**Observação sobre a aparência e comportamento 12.8**

O texto nos botões geralmente usa letras maiúsculas e minúsculas no estilo título de livro.

**Observação sobre a aparência e comportamento 12.9**

Uma GUI pode ter muitos JButtons, mas, em geral, cada rótulo de botão deve ser único na parte da GUI que é atualmente exibida. Ter mais de um JButton com o mesmo rótulo torna os JButtons ambíguos para o usuário.

O aplicativo das figuras 12.15 e 12.16 cria dois JButtons e demonstra que JButtons podem exibir Icons. O tratamento de evento para os botões é realizado por uma única instância da *classe interna* ButtonHandler (Figura 12.15, linhas 39 a 48).

```

1  // Figura 12.15: ButtonFrame.java
2  // Botões de comando e eventos de ação.
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 final JButton plainJButton; // botão apenas com texto
15     private final JButton fancyJButton; // botão com ícones
16
17     // ButtonFrame adiciona JButtons ao JFrame
18     public ButtonFrame()
19     {
20         super("Testing Buttons");
21         setLayout(new FlowLayout());
22
23         plainJButton = new JButton("Plain Button"); // botão com texto
24         add(plainJButton); // adiciona plainJButton ao JFrame
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 de tratamento para tratamento de evento de botão
33         ButtonHandler handler = new ButtonHandler();
34         fancyJButton.addActionListener(handler);
35         plainJButton.addActionListener(handler);
36     }
37
38     // classe interna para tratamento de evento de botão
39     private class ButtonHandler implements ActionListener
40     {
41         // trata evento de botão
42         @Override
43         public void actionPerformed(ActionEvent event)
44         {
45             JOptionPane.showMessageDialog(ButtonFrame.this, String.format(
46                 "You pressed: %s", event.getActionCommand()));
47         }
48     }
49 } // fim da classe ButtonFrame

```

Figura 12.15 | Botões de comando e eventos de ação.

```

1 // Figura 12.16: 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();
10        buttonFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11        buttonFrame.setSize(275, 110);
12        buttonFrame.setVisible(true);
13    }
14 } // fim da classe ButtonTest

```

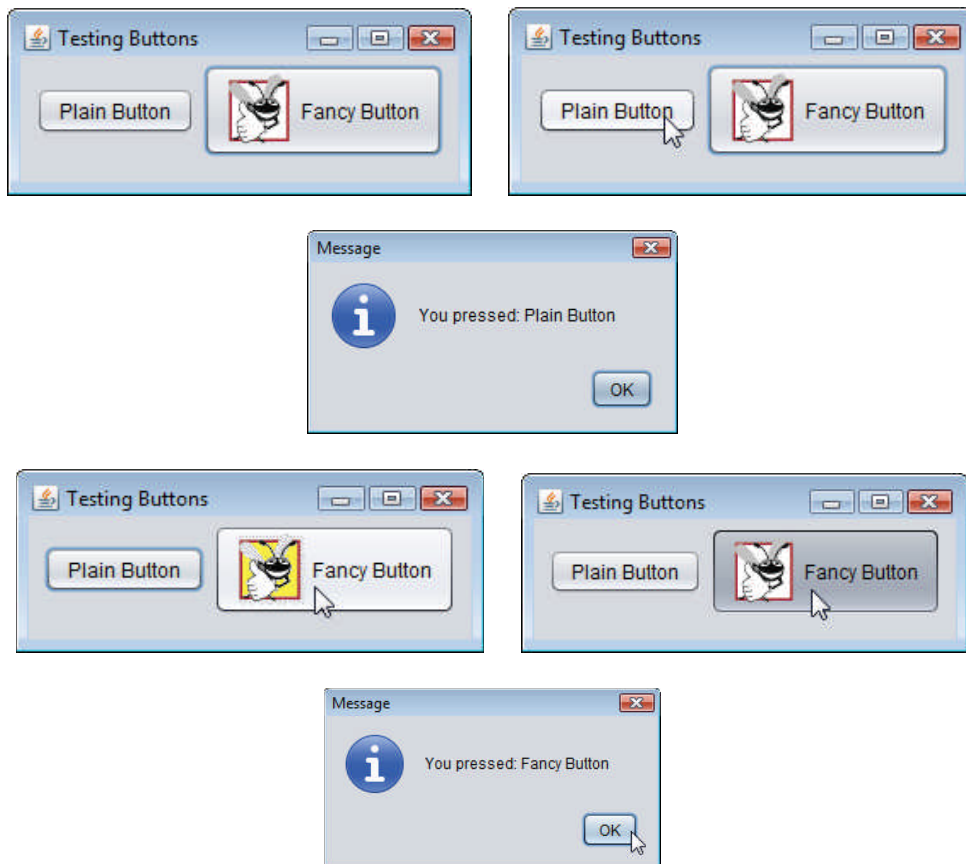


Figura 12.16 | Testando ButtonFrame.

As linhas 14 e 15 declaram as variáveis `JButton plainJButton` e `fancyJButton`. Os objetos correspondentes são instanciados no construtor. A linha 23 cria `plainJButton` com o rótulo de botão "Plain Button". A linha 24 adiciona o `JButton` ao `JFrame`.

Um `JButton` pode exibir um `Icon`. Para fornecer ao usuário um nível extra de interação visual com a GUI, um `JButton` também pode ter um **Icon rollover** — um `Icon` que é exibido quando o usuário posiciona o mouse sobre `JButton`. O ícone em `JButton` altera-se quando o mouse move-se para dentro e para fora da área de `JButton` na tela. As linhas 26 e 27 criam dois objetos `ImageIcon` que representam o `Icon` padrão e o `Icon rollover` para o `JButton` criado na linha 28. As duas declarações supõem que os arquivos de imagem estão armazenados no *mesmo* diretório do aplicativo. As imagens são normalmente inseridas no *mesmo* diretório que o do aplicativo ou em um subdiretório como `images`. Esses arquivos de imagem foram fornecidos para você com o exemplo.

A linha 28 cria `fancyJButton` com o texto "Fancy Button" e o ícone `bug1`. Por padrão, o texto é exibido à *direita* do ícone. A linha 29 utiliza `setRolloverIcon` (herdado da classe `AbstractButton`) para especificar a imagem exibida em `JButton` quando o usuário posiciona o mouse sobre ele. A linha 30 adiciona o `JButton` ao `JFrame`.



Observação sobre a aparência e comportamento 12.10

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 12.11

Ícones de rollover fornecem *feedback visual* indicando que uma ação ocorrerá quando um `JButton` é clicado.

`JButtons`, assim como `TextFields` geram `ActionEvents` que podem ser processados por qualquer objeto `ActionListener`. As linhas 33 a 35 criam um objeto de classe interna `private ButtonHandler` e utilizam `addActionListener` para registrá-lo como a rotina de tratamento de evento para cada `JButton`. A classe `ButtonHandler` (linhas 39 a 48) declara `actionPerformed` para exibir uma caixa de diálogo de mensagem que contém o rótulo do botão que o usuário pressionou. Para um evento `JButton`, o método `getActionCommand` de `ActionEvent` retorna o rótulo no `JButton`.

Acessando a referência `this` em um objeto de uma classe de primeiro nível a partir de uma classe interna

Ao executar esse aplicativo e clicar em um de seus botões, note que o diálogo de mensagem que aparece é centralizado na janela do aplicativo. Isso ocorre porque a chamada para o método `JOptionPane.showMessageDialog` (linhas 45 e 46) usa `ButtonFrame.this`, em vez de `null`, como o primeiro argumento. Quando esse argumento não for `null`, ele representa o componente GUI pai do diálogo de mensagem (nesse caso a janela de aplicativo é o componente pai) e permite ao diálogo estar centralizado sobre esse componente quando o diálogo for exibido. `ButtonFrame.this` representa a referência `this` do objeto de classe de primeiro nível `ButtonFrame`.



Observação de engenharia de software 12.2

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 (`.`) separador, como em `ButtonFrame.this`.

12.10 Botões que mantêm o estado

Os componentes Swing GUI contêm três tipos de botões de estado — `JToggleButton`, `JCheckBox` e `JRadioButton` — que têm valores ativado/desativado ou verdadeiro/falso. As classes `JCheckBox` e `JRadioButton` são subclasses de `JToggleButton` (Figura 12.14). Um `JRadioButton` é diferente de um `JCheckBox` pelo fato de que, normalmente, vários `JRadioButtons` são agrupados e são mutuamente exclusivos — um único no grupo pode ser selecionado a qualquer momento, assim como os botões do rádio de um carro. Primeiro discutiremos a classe `JCheckBox`.

12.10.1 JCheckBox

O aplicativo das figuras 12.17 e 12.18 utiliza dois `JCheckBoxes` para selecionar o estilo desejado de fonte do texto exibido em um `TextField`. Quando selecionado, um aplica o estilo negrito e o outro, o estilo itálico. Se ambos forem selecionados, o estilo é negrito e itálico. Quando o aplicativo é inicialmente executado, nenhuma `JCheckBox` está marcada (isto é, as duas são `false`), então a fonte é simples. A classe `CheckBoxTest` (Figura 12.18) contém o método `main` que executa esse aplicativo.

```

1 // Figura 12.17: CheckBoxFrame.java
2 // JCheckBoxes e eventos de item.
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 {

```

continua

continuação

```

13 private final JTextField textField; // exibe o texto na alteração de fontes
14 private final JCheckBox boldJCheckBox; // para marcar/desmarcar estilo negrito
15 private final JCheckBox italicJCheckBox; // para marcar/desmarcar estilo itálico
16
17 // construtor CheckBoxFrame adiciona JCheckBoxes ao JFrame
18 public CheckBoxFrame()
19 {
20     super("JCheckBox Test");
21     setLayout(new FlowLayout());
22
23     // configura JTextField e sua fonte
24     textField = new JTextField("Watch the font style change", 20);
25     textField.setFont(new Font("Serif", Font.PLAIN, 14));
26     add(textField); // adiciona textField ao JFrame
27
28     boldJCheckBox = new JCheckBox("Bold");
29     italicJCheckBox = new JCheckBox("Italic");
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 }
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     @Override
44     public void itemStateChanged(ItemEvent event)
45     {
46         Font font = null; // armazena a nova Font
47
48         // determina quais CheckBoxes estão marcadas e cria Font
49         if (boldJCheckBox.isSelected() && italicJCheckBox.isSelected())
50             font = new Font("Serif", Font.BOLD + Font.ITALIC, 14);
51         else if (boldJCheckBox.isSelected())
52             font = new Font("Serif", Font.BOLD, 14);
53         else if (italicJCheckBox.isSelected())
54             font = new Font("Serif", Font.ITALIC, 14);
55         else
56             font = new Font("Serif", Font.PLAIN, 14);
57
58         textField.setFont(font);
59     }
60 }
61 } // fim da classe CheckBoxFrame

```

Figura 12.17 | JCheckBoxes e eventos de item.

```

1 // Figura 12.18: 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);

```

continua

```

11     checkBoxFrame.setSize(275, 100);
12     checkBoxFrame.setVisible(true);
13 }
14 } // fim da classe CheckBoxTest

```

continuação

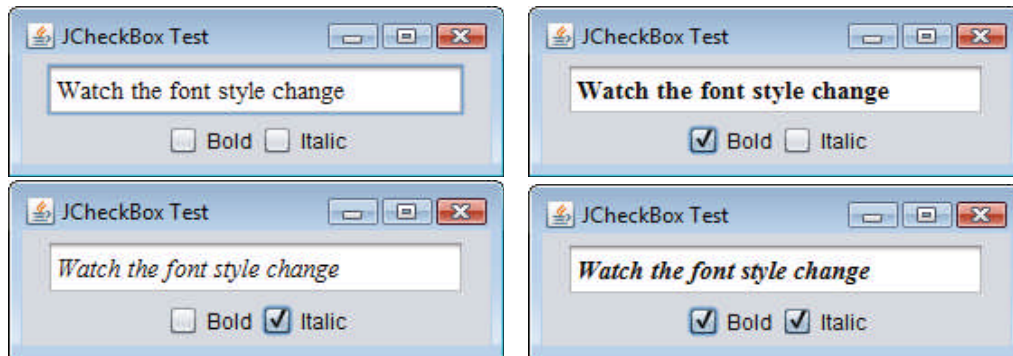


Figura 12.18 | Testando CheckBoxFrame.

Depois de o `JTextField` ser criado e inicializado (Figura 12.17, linha 24), a linha 25 utiliza o método `setFont` (herdado por `JTextField` indiretamente da classe `Component`) para configurar a fonte do `JTextField` como um novo objeto de classe **Font** (pacote `java.awt`). A nova `Font` é inicializada com "Serif" (um nome de fonte para representar uma fonte genérica como Times e suportada em todas as plataformas do Java), estilo `Font.PLAIN` e corpo 14. Em seguida, as linhas 28 e 29 criam dois objetos `JCheckBox`. A `String` passada para o construtor `JCheckBox` é o **rótulo de caixa de seleção** que aparece à direita da `JCheckBox` por padrão.

Quando o usuário clicar em uma `JCheckBox`, um **ItemEvent** ocorre. Esse evento pode ser tratado por um objeto **ItemListener**, que *deve* implementar o método `itemStateChanged`. Nesse exemplo, o tratamento de evento é realizado por uma instância da *classe interna* `private` `CheckBoxHandler` (linhas 40 a 60). As linhas 34 a 36 criam uma instância de classe `CheckBoxHandler` e a registram com o método `addItemListener` como o ouvinte de ambos os objetos `JCheckBox`.

O método `itemStateChanged` de `CheckBoxHandler` (linhas 43 a 59) é chamado quando o usuário clica em `italicJCheckBox` ou `boldJCheckBox`. Nesse exemplo, não determinamos qual `JCheckBox` foi clicada — usamos ambos os estados para determinar a fonte a exibir. A linha 49 usa o método `JCheckBox.isSelected` para determinar se as duas `JCheckBoxes` estão selecionadas. Se estiverem, a linha 50 cria uma fonte em negrito e itálico adicionando as constantes `Font.BOLD` e `Font.ITALIC` ao argumento do estilo de fonte do construtor `Font`. A linha 51 determina se a `boldJCheckBox` está selecionada e, se estiver, a linha 52 cria uma fonte em negrito. A linha 53 determina se a `italicJCheckBox` está selecionada e, se estiver, a linha 54 cria uma fonte em itálico. Se nenhuma das condições anteriores for verdadeira, a linha 56 cria uma fonte simples usando a constante `Font.PLAIN`. Por fim, a linha 58 configura uma nova fonte de `textField`, o que altera a fonte no `JTextField` na tela.

Relacionamento entre uma classe interna e sua classe de primeiro nível

A classe `CheckBoxHandler` utilizou as variáveis `boldJCheckBox` (linhas 49 e 51), `italicJCheckBox` (linhas 49 e 53) e `textField` (linha 58), embora elas *não* sejam declaradas na classe interna. Lembre-se de que uma *classe interna* tem um relacionamento especial com a *classe de nível superior* — ela pode acessar *todas* as variáveis e métodos da classe de nível superior. O método `CheckBoxHandler.itemStateChanged` (linhas 43 a 59) utiliza esse relacionamento para determinar quais `JCheckBoxes` estão marcadas e definir a fonte no `JTextField`. Observe que nenhum código na classe interna `CheckBoxHandler` exige uma referência explícita ao objeto de primeiro nível de classe.

12.10.2 JRadioButton

Botões de opção (declarados com a classe `JRadioButton`) são semelhantes a caixas de seleção pelo fato de ter dois estados — *selecionados* e *não selecionados* (também chamados *desmarcados*). Entretanto, os botões de opção normalmente aparecem como um **grupo** em que apenas *um* botão pode ser selecionado por vez (ver a saída da Figura 12.20). Os botões de opção são utilizados para representar **opções mutuamente exclusivas** (isto é, não é *possível* selecionar múltiplas opções no grupo ao mesmo tempo). O relacionamento lógico entre botões de opção é mantido por um objeto **ButtonGroup** (pacote `javax.swing`), que em si *não* é um componente GUI. Um objeto `ButtonGroup` organiza um grupo de botões e ele mesmo *não* é exibido em uma interface com o usuário. Em seu lugar, os objetos individuais `JRadioButton` do grupo são exibidos na GUI.

O aplicativo das figuras 12.19 e 12.20 é semelhante ao das figuras 12.17 e 12.18. O usuário pode alterar o estilo da fonte do texto de um `TextField`. O aplicativo utiliza botões de opção que permitem que apenas um único estilo de fonte no grupo seja selecionado de cada vez. A classe `RadioButtonTest` (Figura 12.20) contém o método `main`, que executa esse aplicativo.

```

1 // Figura 12.19: 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 final JTextField textField; // utilizado para exibir alterações de fonte
15     private final Font plainFont; // fonte para texto simples
16     private final Font boldFont; // fonte para texto em negrito
17     private final Font italicFont; // fonte para texto em itálico
18     private final Font boldItalicFont; // fonte para texto em negrito e itálico
19     private final JRadioButton plainJRadioButton; // seleciona texto simples
20     private final JRadioButton boldJRadioButton; // seleciona texto em negrito
21     private final JRadioButton italicJRadioButton; // seleciona texto em itálico
22     private final JRadioButton boldItalicJRadioButton; // negrito e itálico
23     private final ButtonGroup radioGroup; // contém botões de opção
24
25     // construtor RadioButtonFrame adiciona JRadioButtons ao JFrame
26     public RadioButtonFrame()
27     {
28         super("RadioButton Test");
29         setLayout(new FlowLayout());
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 texto simples ao grupo
47         radioGroup.add(boldJRadioButton); // adiciona negrito ao grupo
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);
57
58         // registra eventos para JRadioButtons
59         plainJRadioButton.addItemListener(
60             new RadioButtonHandler(plainFont));
61         boldJRadioButton.addItemListener(

```

continua

continuação

```

62         new RadioButtonHandler(boldFont));
63     italicJRadioButton.addItemListener(
64         new RadioButtonHandler(italicFont));
65     boldItalicJRadioButton.addItemListener(
66         new RadioButtonHandler(boldItalicFont));
67 }
68
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;
77     }
78
79     // trata eventos de botão de opção
80     @Override
81     public void itemStateChanged(ItemEvent event)
82     {
83         textField.setFont(font);
84     }
85 }
86 } // fim da classe RadioButtonFrame

```

Figura 12.19 | Criando botões de opção com ButtonGroup e JRadioButton.

```

1 // Figura 12.20: 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);
12        radioButtonFrame.setVisible(true);
13    }
14 } // fim da classe RadioButtonTest

```

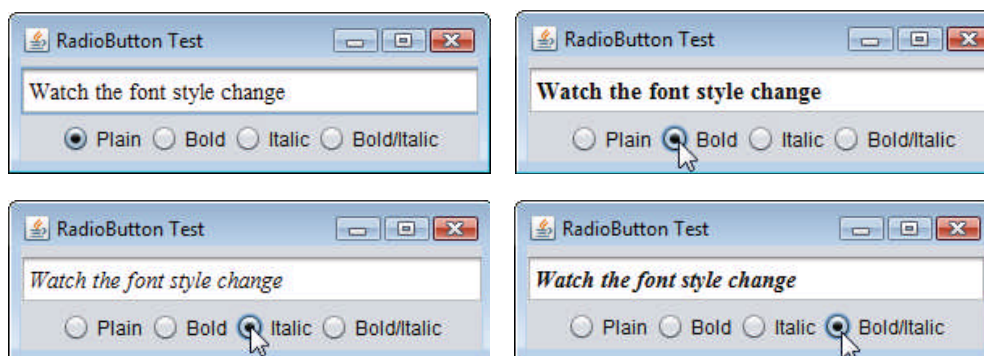


Figura 12.20 | Testando RadioButtonFrame.

As linhas 35 a 42 (Figura 12.19) no construtor criam quatro objetos `JRadioButton` e os adicionam ao `JFrame`. Cada `JRadioButton` é criado com uma chamada de construtor como aquela na linha 35. Esse construtor especifica o rótulo que aparece à direita do `JRadioButton` por padrão e o estado inicial do `JRadioButton`. Um segundo argumento `true` indica que o `JRadioButton` deve aparecer *selecionado* quando for exibido.

A linha 45 instancia objeto `ButtonGroup` `radioGroup`. Esse objeto é a “cola” que forma o relacionamento lógico entre os quatro objetos `JRadioButton` e permite que somente um dos quatro seja selecionado por vez. É possível que nenhum `JRadioButton` em um `ButtonGroup` seja selecionado, mas isso pode ocorrer *somente se nenhum* `JRadioButton` pré-selecionado for adicionado ao `ButtonGroup` e o usuário ainda *não* tiver selecionado um `JRadioButton`. As linhas 46 a 49 utilizam o método `ButtonGroup` **add** para associar cada um dos `JRadioButtons` com `radioGroup`. Se mais de um objeto `JRadioButton` selecionado for adicionado ao grupo, aquele que tiver sido adicionado *primeiro* será selecionado quando a GUI for exibida.

`JRadioButtons`, como `JCheckBoxes`, geram `ItemEvents` quando são *clikados*. As linhas 59 a 66 criam quatro instâncias de classe interna `RadioButtonHandler` (declaradas nas linhas 70 a 85). Nesse exemplo, cada objeto ouvinte de eventos é registrado para tratar o `ItemEvent` gerado quando o usuário clica em um `JRadioButton` particular. Note que todo objeto `RadioButtonHandler` é inicializado com um objeto `Font` particular (criado nas linhas 52 a 55).

A classe `RadioButtonHandler` (linhas 70 a 85) implementa a interface `ItemListener` para que ela possa tratar `ItemEvents` gerados por `JRadioButtons`. O construtor armazena o objeto `Font` que ele recebe como um argumento na variável de instância `font` do objeto ouvinte de eventos (declarada na linha 72). Quando o usuário clica em um `JRadioButton`, `radioGroup` desliga o `JRadioButton` anteriormente selecionado e o método `itemStateChanged` (linhas 80 a 84) configura o `JTextField` como a `Font` armazenada no objeto ouvinte de eventos correspondente do `JRadioButton`. Note que a linha 83 da classe interna `RadioButtonHandler` utiliza a variável de instância `textField` da classe de primeiro nível para configurar a fonte.

12.11 JComboBox e uso de uma classe interna anônima para tratamento de eventos

A caixa de combinação (às vezes chamada **lista drop-down**) permite que o usuário selecione *um* item de uma lista (Figura 12.22). Caixas de combinação são implementadas com a classe `JComboBox` que estende a classe `JComponent`. `JComboBox` é uma classe genérica, como a classe `ArrayList` (Capítulo 7). Ao criar um `JComboBox`, você especifica o tipo dos objetos que ele gerencia — a `JComboBox` então exibe uma representação `String` de cada objeto.

```

1 // Figura 12.21: JComboBoxFrame.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 JComboBoxFrame extends JFrame
13 {
14     private final JComboBox<String> imagesJComboBox; // contém nomes de ícones
15     private final JLabel label; // exibe o ícone selecionado
16
17     private static final String[] names =
18     { "bug1.gif", "bug2.gif", "travelbug.gif", "buganim.gif" };
19     private final 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
25     // construtor JComboBoxFrame adiciona JComboBox ao JFrame
26     public JComboBoxFrame()
27     {
28         super("Testing JComboBox");
29         setLayout(new FlowLayout()); // configura o layout de frame
30
31         imagesJComboBox = new JComboBox<String>(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

```

continua

continuação

```

38         @Override
39         public void itemStateChanged(ItemEvent event)
40         {
41             // determina se o item está selecionado
42             if (event.getStateChange() == ItemEvent.SELECTED)
43                 label.setIcon(Icons[
44                     imagesJComboBox.getSelectedIndex()]);
45         }
46     } // fim da classe interna anônima
47 } // fim da chamada para addItemListener
48
49 add(imagesJComboBox); // adiciona caixa de combinação ao JFrame
50 label = new JLabel(Icons[0]); // exibe primeiro ícone
51 add(label); // adiciona rótulo ao JFrame
52 }
53 } // fim da classe ComboBoxFrame

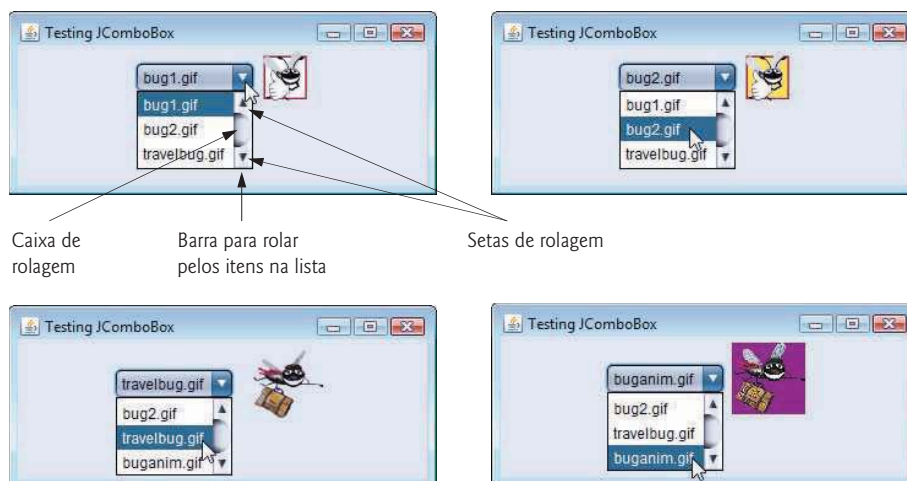
```

Figura 12.21 | JComboBox que exibe uma lista de nomes de imagem.

```

1 // Figura 12.22: 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);
12        comboBoxFrame.setVisible(true);
13    }
14 } // fim da classe ComboBoxTest

```

**Figura 12.22** | Testando ComboBoxFrame.

JComboBoxes geram `ItemEvents` assim como `JCheckBoxes` e `JRadioButtons` fazem. Esse exemplo também demonstra uma forma especial da classe interna que é usada com frequência no tratamento de evento. O aplicativo (figuras 12.21 e 12.22) utiliza uma `JComboBox` para fornecer uma lista de quatro nomes de arquivo de imagem a partir da qual o usuário pode selecionar uma imagem para ser exibida. Quando o usuário seleciona um nome, o aplicativo exibe a imagem correspondente como um `Icon` em um `JLabel`. A classe `ComboBoxTest` (Figura 12.22) contém o método `main` que executa esse aplicativo. As capturas de tela para esse aplicativo mostram a lista `JComboBox` depois que a seleção foi feita para ilustrar qual nome de arquivo de imagem foi selecionado.

As linhas 19 a 23 (Figura 12.21) declaram e inicializam o array `icons` com quatro novos objetos `ImageIcon`. O array `String` `names` (linhas 17 e 18) contém os nomes dos quatro arquivos de imagem que estão armazenados no mesmo diretório do aplicativo.

Na linha 31, o construtor inicializa um objeto `JComboBox`, utilizando `Strings` no array `names` como os elementos na lista. Todo item na lista tem um **índice**. O primeiro item é adicionado no índice 0, o próximo no índice 1 etc. O primeiro item adicionado a uma `JComboBox` aparece como o item atualmente selecionado quando a `JComboBox` é exibida. Outros itens são selecionados clicando no `JComboBox`, então selecionando um item da lista que aparece.

A linha 32 utiliza o método `JComboBox` `setMaximumRowCount` para configurar o número máximo de elementos que é exibido quando o usuário clica em `JComboBox`. Se houver itens adicionais, a `JComboBox` fornece uma **barra de rolagem** (ver a primeira tela) que permite que o usuário role por todos os elementos na lista. O usuário pode clicar nas **setas de rolagem** na parte superior e inferior da barra de rolagem para mover-se para cima e para baixo pela lista um elemento por vez ou então arrastar a **caixa de rolagem** no meio da barra de rolagem para cima e para baixo. Para arrastar a caixa de rolagem, posicione o cursor de mouse sobre ela, segure o botão do mouse e mova-o. Nesse exemplo, a lista drop-down é muito curta para arrastar a caixa de rolagem, assim clique nas setas para cima e para baixo ou use a roda do mouse para rolar pelos quatro itens na lista. A linha 49 anexa a `JComboBox` ao `FlowLayout` (configurado na linha 29) do `ComboBoxFrame`. A linha 50 cria o `JLabel` que exibe `ImageIcons` e inicializa-o com o primeiro `ImageIcon` no array `icons`. A linha 51 anexa o `JLabel` ao `FlowLayout` do `ComboBoxFrame`.



Observação sobre a aparência e comportamento 12.12

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.

Utilizando uma classe interna anônima para tratamento de evento

As linhas 34 a 46 são uma instrução que declara a classe do ouvinte de evento, cria um objeto dessa classe e o registra como o ouvinte `ItemEvent` de `imagesJComboBox`. Esse objeto ouvinte de evento é uma instância de uma **classe interna anônima** — uma classe que é declarada sem um nome e que normalmente aparece dentro de uma declaração de método. *Como com outras classes internas, uma classe interna anônima pode acessar os membros de sua classe de primeiro nível.* Mas uma classe interna anônima tem acesso limitado às variáveis locais do método em que é declarada. Como uma classe interna anônima não tem nomes, deve-se criar um objeto da classe interna anônima no ponto em que a classe é declarada (iniciando na linha 35).



Observação de engenharia de software 12.3

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. A partir do Java SE 8, classes internas anônimas também podem acessar as variáveis locais “efetivamente” `final` de um método — veja no Capítulo 17 informações adicionais.

As linhas 34 a 47 são uma chamada para o método `addItemListener` da `imagesJComboBox`. O argumento para esse método deve ser um objeto que é um `ItemListener` (isto é, qualquer objeto de uma classe que implementa `ItemListener`). As linhas 35 a 46 são uma expressão de criação da instância de classe que declara uma classe interna anônima e cria um objeto dessa classe. Uma referência àquele objeto então é passada como o argumento para `addItemListener`. A sintaxe `ItemListener()` depois `new` inicia a declaração de uma classe interna anônima que implementa a interface `ItemListener`. Isso é semelhante a começar uma declaração de classe com

```
public class MyHandler implements ItemListener
```

A chave de abertura esquerda na linha 36 e a chave de fechamento direita na linha 46 delimitam o corpo da classe interna anônima. As linhas 38 a 45 declaram o método `itemStateChanged` de `ItemListener`. Quando o usuário fizer uma seleção de `imagesJComboBox`, esse método configura `Icon` do `Label`. O `Icon` é selecionado a partir do array `icons` determinando o índice do item selecionado na `JComboBox` com o método `getSelectedIndex` na linha 44. Para cada item selecionado de um `JComboBox`, a seleção de outro item é primeiro removida — então ocorrem dois `ItemEvents` quando um item é selecionado. Pretendemos exibir apenas o ícone do item que o usuário acabou de selecionar. Por essa razão, a linha 42 determina se o método `ItemEvent` `getStateChange` retorna `ItemEvent.SELECTED`. Se retornar, as linhas 43 e 44 configuram o ícone de `Label`.



Observação de engenharia de software 12.4

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

A sintaxe mostrada nas linhas 35 a 46 para criar uma rotina de tratamento de evento com uma classe interna anônima é semelhante ao código que seria gerado por um ambiente de desenvolvimento integrado Java (*integrated development environment* — IDE). Em geral, um IDE permite projetar uma GUI visualmente; então, ele gera o código que implementa a GUI. Simplesmente insira instruções nos métodos de tratamento de evento que declaram a maneira como tratar cada evento.

Java SE 8: implementando classes internas anônimas com lambdas

Na Seção 17.9, mostraremos como usar lambdas Java SE 8 para criar rotinas de tratamento de evento. Como veremos, o compilador converte um lambda em um objeto de uma classe interna anônima.

12.12 JList

Uma lista exibe uma série de itens a partir da qual o usuário pode *selecionar um ou mais deles* (ver a saída da Figura 12.24). Listas são criadas com a classe `JList`, que estende diretamente a classe `JComponent`. A classe `JList` — que como `JComboBox` é uma classe genérica — suporta **listas de seleção única** (que permitem que apenas um item seja selecionado de cada vez) e **listas de seleção múltipla** (que permitem que quaisquer itens sejam selecionados). Nesta seção, discutimos listas de uma única seleção.

O aplicativo das figuras 12.23 e 12.24 cria uma `JList` contendo 13 nomes de cores. Ao clicar em um nome de cor na `JList`, um `ListSelectionEvent` ocorre e o aplicativo muda a cor de fundo da janela de aplicativo para a cor selecionada. A classe `ListTest` (Figura 12.24) contém o método `main` que executa esse aplicativo.

```

1  // Figura 12.23: 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 final JList<String> 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
23     // construtor ListFrame adiciona JScrollPane que contém JList ao JFrame
24     public ListFrame()
25     {
26         super("List Test");
27         setLayout(new FlowLayout());
28
29         colorJList = new JList<String>(colorNames); // lista de 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
38         colorJList.addListSelectionListener(
39             new ListSelectionListener() // classe interna anônima
40             {
41                 // trata eventos de seleção de lista
42                 @Override
43                 public void valueChanged(ListSelectionEvent event)
44             {

```

continua

```

45         getContentPane().setBackground(
46             colors[colorJList.getSelectedIndex()]);
47     }
48 }
49 );
50 }
51 } // fim da classe ListFrame

```

Figura 12.23 | JList que exibe uma lista de cores.

```

1 // Figura 12.24: 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);
12        listFrame.setVisible(true);
13    }
14 } // fim da classe ListTest

```



Figura 12.24 | Selecionando cores de uma JList.

A linha 29 (Figura 12.23) cria o objeto `JList colorJList`. O argumento para o construtor `JList` é o array de `Objects` (nesse caso `Strings`) para exibir na lista. A linha 30 utiliza o método `JList setVisibleRowCount` para determinar o número de itens visíveis na lista.

A linha 33 utiliza o método `JList setSelectionMode` para especificar o **modo de seleção** da lista. A classe `ListSelectionMode` (do pacote `javax.swing`) declara três constantes que especificam o modo de seleção de uma `JList` — `SINGLE_SELECTION` (que permite que apenas um item seja selecionado por vez), `SINGLE_INTERVAL_SELECTION` (para uma lista de seleção múltipla que permite a seleção de vários itens contíguos) e `MULTIPLE_INTERVAL_SELECTION` (para uma lista de seleção múltipla que 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. Nesse caso, um objeto `JScrollPane` é utilizado para fornecer a capacidade de rolagem. A linha 36 adiciona uma nova instância da classe `JScrollPane` ao `JFrame`. O construtor `JScrollPane` recebe como seu argumento o `JComponent` que precisa de funcionalidades de rolagem (nesse caso, `colorJList`). Observe nas capturas de tela que uma barra de rolagem criada pelo `JScrollPane` aparece no lado direito da `JList`. Por padrão, a barra de rolagem só aparece quando o número de itens na `JList` excede o número de itens visíveis.

As linhas 38 a 49 utilizam o método `JList addListSelectionListener` para registrar um objeto que implementa `ListSelectionListener` (pacote `javax.swing.event`) como o listener para os eventos de seleção de `JList`. Mais uma vez, utilizamos uma instância de uma classe interna anônima (linhas 39 a 48) como o listener. Nesse exemplo, quando o usuário faz uma seleção de `colorJList`, o método `valueChanged` (linhas 42 a 47) deve mudar a cor de fundo do `ListFrame` para a cor selecionada. Isso é realizado nas linhas 45 e 46. Observe o uso do método `JFrame getContentPane` na linha 45. Todo `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 no `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`; portanto, para mudar a cor de fundo por trás 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`). Na linha 45, utilizamos depois essa referência para chamar o método `setBackground`, que configura a cor de fundo do painel de conteúdo como um elemento no array `colors`. A cor é selecionada a partir do array utilizando-se o índice do item selecionado. O método `JList.getSelectedIndex` retorna o índice do item selecionado. Como com arrays e `JComboBoxes`, a indexação de `JList` é baseada em zero.

12.13 Listas de seleção múltipla

Uma **lista de seleção múltipla** permite ao usuário selecionar muitos itens de uma `JList` (ver a saída da Figura 12.26). Uma lista `SINGLE_INTERVAL_SELECTION` permite selecionar um intervalo contíguo de itens. Para fazer isso, clique no primeiro item, então pressione e mantenha a tecla *Shift* pressionada ao clicar no último item no intervalo. Uma lista `MULTIPLE_INTERVAL_SELECTION` (o padrão) permite seleção de intervalo contínuo como descrito para uma lista `SINGLE_INTERVAL_SELECTION`. Essa lista também permite que diversos itens sejam selecionados pressionando e segurando a tecla *Ctrl* ao clicar em cada item a selecionar. Para *remover a seleção* de um item, pressione e segure a tecla *Ctrl* ao clicar no item uma segunda vez.

O aplicativo das figuras 12.25 e 12.26 utiliza listas de seleção múltiplas para copiar itens de um `JList` para outro. Uma lista é `MULTIPLE_INTERVAL_SELECTION` e a outra é `SINGLE_INTERVAL_SELECTION`. Ao executar o aplicativo, tente utilizar as técnicas de seleção descritas anteriormente para selecionar itens das duas listas.

```

1 // Figura 12.25: MultipleSelectionFrame.java
2 // JList que permite múltiplas seleções.
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 final JList<String> colorJList; // lista para armazenar nomes de cor
15     private final JList<String> copyJList; // lista para armazenar nomes copiados
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     {
24         super("Multiple Selection Lists");
25         setLayout(new FlowLayout());
26
27         colorJList = new JList<String>(colorNames); // lista dos nomes de 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 >>>");
34         copyJButton.addActionListener(
35             new ActionListener() // classe interna anônima
36             {
37                 // trata evento de botão
38                 @Override
39                 public void actionPerformed(ActionEvent event)
40                 {
41                     // coloca valores selecionados na copyJList
42                     copyJList.setListData(
43                         colorJList.getSelectedValuesList().toArray(
44                             new String[0]));
45                 }
46             }
47     );

```

continua

```

48
49     add(copyJButton); // adiciona botão de cópia ao JFrame
50
51     copyJList = new JList<String>(); // lista para armazenar nomes copiados
52     copyJList.setVisibleRowCount(5); // mostra 5 linhas
53     copyJList.setFixedCellWidth(100); // configura a largura
54     copyJList.setFixedCellHeight(15); // configura a altura
55     copyJList.setSelectionMode(
56         ListSelectionModel.SINGLE_INTERVAL_SELECTION);
57     add(new JScrollPane(copyJList)); // adiciona lista com scrollpane
58 }
59 } // fim da classe MultipleSelectionFrame

```

Figura 12.25 | JList que permite múltiplas seleções.

```

1 // Figura 12.26: 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);
14         multipleSelectionFrame.setVisible(true);
15     }
16 } // fim da classe MultipleSelectionTest

```

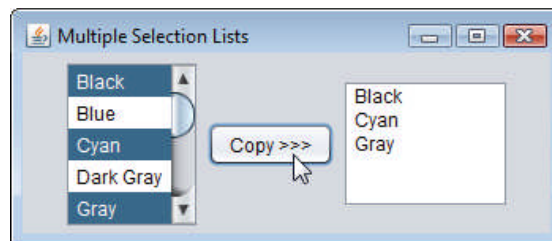


Figura 12.26 | Testando MultipleSelectionFrame.

A linha 27 da Figura 12.25 cria a JList `colorJList` e a inicializa com as Strings no array `colorNames`. A linha 28 configura o número de linhas visíveis em `colorJList` como 5. As linhas 29 e 30 especificam que `colorJList` é uma lista `MULTIPLE_INTERVAL_SELECTION`. A linha 31 adiciona um novo `JScrollPane` contendo `colorJList` ao `JFrame`. As linhas 51 a 57 realizam tarefas semelhantes para `copyJList`, que é declarada como uma lista `SINGLE_INTERVAL_SELECTION`. Se uma JList não contiver itens, ela não será exibida em um `FlowLayout`. Por essa razão, as linhas 53 e 54 utilizam os métodos JList `setFixedCellWidth` e `setFixedCellHeight` para definir a largura da `copyJList` como 100 pixels e a altura de cada item na JList como 15 pixels, respectivamente.

Normalmente, um evento gerado por outro componente GUI (conhecido como um **evento externo**) especifica quando as múltiplas seleções em uma JList devem ser processadas. Nesse exemplo, o usuário clica no JButton chamado `copyJButton` para disparar o evento que copia os itens selecionados em `colorJList` para `copyJList`.

As linhas 34 a 47 declaram, criam e registram um `ActionListener` para o `copyJButton`. Quando o usuário clica em `copyJButton`, o método `actionPerformed` (linhas 38 a 45) utiliza o método JList `setListData` para configurar os itens exibidos em `copyJList`. As linhas 43 e 44 chamam o método `getSelectedValuesList` de `colorJList`, que retorna uma `List<String>` (porque a JList foi criada como uma `JList<String>`) representando os itens selecionados em `colorJList`. Chamamos o método `toArray` de `List<String>` para converter isso em um array de Strings que pode ser passado como o argumento para o método `setListData` de `copyJList`. O método `toArray` de `List` recebe como argumento um array que representa o tipo de array que o método retornará. Você vai aprender mais sobre `List` e `toArray` no Capítulo 16.

Talvez você esteja se perguntando por que `copyJList` pode ser usada na linha 42, embora o aplicativo só crie o objeto ao qual ele se refere depois da linha 49. Lembre-se de que o método `actionPerformed` (linhas 38 a 45) não executa até que o usuário pressione o `copyJButton`, o que só pode ocorrer depois que o construtor foi executado e o aplicativo exibe a GUI. Nesse ponto na execução do aplicativo, `copyJList` já é inicializado com um novo objeto `JList`.

12.14 Tratamento de evento de mouse

Esta seção apresenta as interfaces ouvintes de evento `MouseListener` e `MouseMotionListener` para tratar **eventos de mouse**. Eventos de mouse podem ser processados por qualquer componente GUI que deriva de `java.awt.Component`. Os métodos de interfaces `MouseListener` e `MouseMotionListener` são resumidos na Figura 12.27. O pacote `javax.swing.event` contém a interface `MouseInputListener`, que estende as interfaces `MouseListener` e `MouseMotionListener` para criar uma única interface que contém todos os métodos `MouseListener` e `MouseMotionListener`. Os métodos `MouseListener` e `MouseMotionListener` são chamados quando o mouse interage com um `Component` se objetos listeners de evento apropriados forem registrados para esse `Component`.

Cada um dos métodos de tratamento de evento de mouse recebe como argumento um objeto `MouseEvent` que contém informações sobre o evento de mouse que ocorreu, incluindo as coordenadas x e y da sua localização. Essas coordenadas são medidas do *canto superior esquerdo* do componente GUI em que o evento ocorreu. As coordenadas x iniciam em 0 e *aumentam da esquerda para a direita*. As coordenadas y iniciam em 0 e *aumentam 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.

Métodos de interface `MouseListener` e `MouseMotionListener`.

Métodos da interface `MouseListener`

`public void mousePressed(MouseEvent event)`

Chamado quando um botão do mouse é *pressionado* enquanto o cursor de 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. Sempre precedido por uma chamada a `mousePressed` e `mouseReleased`.

`public void mouseReleased(MouseEvent event)`

Chamado quando um botão do mouse é *liberado depois de ser pressionado*. Sempre precedido por uma chamada a `mousePressed` e uma ou mais chamadas a `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.

Métodos da interface `MouseMotionListener`

`public void mouseDragged(MouseEvent event)`

Chamado quando o botão do mouse é *pressionado* enquanto o cursor de mouse estiver sobre um componente e o mouse é *movido* enquanto o botão do mouse *permanecer pressionado*. Sempre precedido por uma chamada a `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 botões do mouse pressionados) 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 12.27 | Métodos de interface `MouseListener` e `MouseMotionListener`.



Observação de engenharia de software 12.5

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

O Java também fornece a interface `MouseWheelListener` para permitir que aplicativos respondam à *rotação da roda de um mouse*. Essa interface declara o método `mouseWheelMoved`, que recebe um `MouseWheelEvent` como seu argumento. A classe `MouseWheelEvent` (uma subclasse de `MouseEvent`) contém métodos que permitem que a rotina de tratamento de evento obtenha as informações sobre a quantidade de rotação de roda.

Monitorando eventos de mouse em um `JPanel`

O aplicativo `MouseTracker` (figuras 12.28 e 12.29) demonstra os métodos de interface `MouseListener` e `MouseMotionListener`. A classe da rotina de tratamento de evento (linhas 36 a 97 da Figura 12.28) implementa as duas interfaces. Você *deve* declarar todos os sete métodos dessas duas interfaces quando sua classe implementa as duas. Cada evento de mouse nesse exemplo exibe uma `String` no `JLabel` chamada `statusBar`, que é anexada à parte inferior da janela.

```

1  // Figura 12.28: MouseTrackerFrame.java
2  // Tratamento de evento 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 final JPanel mousePanel; // painel em que os eventos de mouse ocorrem
15     private final JLabel statusBar; // exibe informações do evento
16
17     // construtor MouseTrackerFrame configura GUI e
18     // registra rotinas de tratamento de evento de mouse
19     public MouseTrackerFrame()
20     {
21         super("Demonstrating Mouse Events");
22
23         mousePanel = new JPanel();
24         mousePanel.setBackground(Color.WHITE);
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     }
35
36     private class MouseHandler implements MouseListener,
37         MouseMotionListener
38     {
39         // rotinas de tratamento de evento MouseListener
40         // trata evento quando o mouse é liberado imediatamente depois de ter sido pressionado
41         @Override
42         public void mouseClicked(MouseEvent event) {
43             statusBar.setText(String.format("Clicked at [%d, %d]",
44                 event.getX(), event.getY()));
45         }
46
47         // trata evento quando mouse é pressionado
48         @Override
49         public void mousePressed(MouseEvent event) {
50             statusBar.setText(String.format("Pressed at [%d, %d]",
51                 event.getX(), event.getY()));
52         }
53     }

```

continua

continuação

```

54     }
55
56     // trata o evento quando o mouse é liberado
57     @Override
58     public void mouseReleased(MouseEvent event)
59     {
60         statusBar.setText(String.format("Released at [%d, %d]",
61             event.getX(), event.getY()));
62     }
63
64     // trata evento quando mouse entra na área
65     @Override
66     public void mouseEntered(MouseEvent event)
67     {
68         statusBar.setText(String.format("Mouse entered at [%d, %d]",
69             event.getX(), event.getY()));
70         mousePanel.setBackground(Color.GREEN);
71     }
72
73     // trata evento quando mouse sai da área
74     @Override
75     public void mouseExited(MouseEvent event)
76     {
77         statusBar.setText("Mouse outside JPanel");
78         mousePanel.setBackground(Color.WHITE);
79     }
80
81     // rotinas de tratamento de evento MouseMotionListener
82     // trata o evento quando o usuário arrasta o mouse com o botão pressionado
83     @Override
84     public void mouseDragged(MouseEvent event)
85     {
86         statusBar.setText(String.format("Dragged at [%d, %d]",
87             event.getX(), event.getY()));
88     }
89
90     // trata evento quando usuário move o mouse
91     @Override
92     public void mouseMoved(MouseEvent event)
93     {
94         statusBar.setText(String.format("Moved at [%d, %d]",
95             event.getX(), event.getY()));
96     }
97 } // fim da classe interna MouseHandler
98 } // fim da classe MouseTrackerFrame

```

Figura 12.28 | Tratamento de evento de mouse.

```

1 // Figura 12.29: 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);
12        mouseTrackerFrame.setVisible(true);
13    }
14 } // fim da classe MouseTracker

```

continua

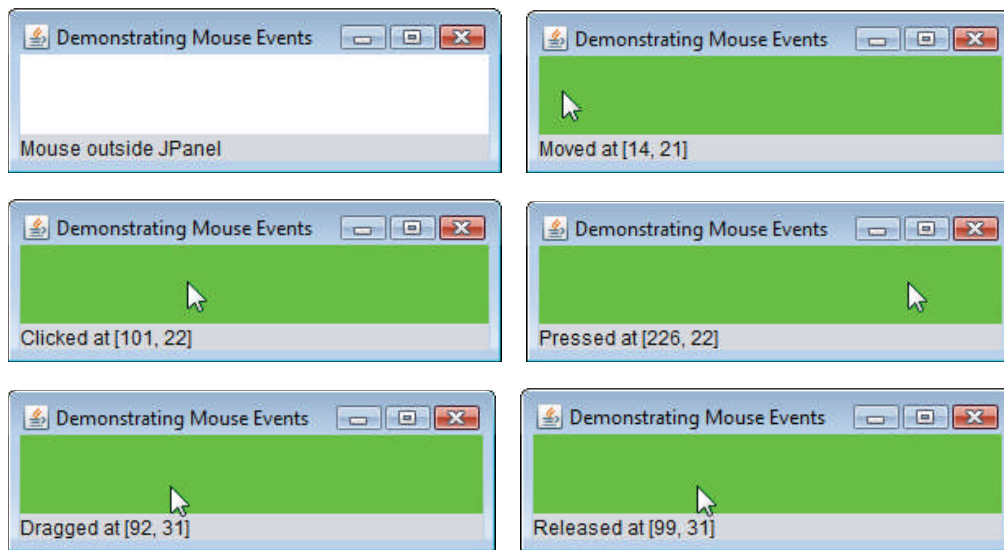


Figura 12.29 | Testando MouseTrackerFrame.

A linha 23 cria `JPanel mousePane1`. Os eventos de mouse desse `JPanel` são monitorados pelo aplicativo. A linha 24 configura a cor de fundo `mousePane1` como branco. Quando o usuário mover o mouse no `mousePane1`, o aplicativo irá mudar a cor de fundo do `mousePane1` para verde. Quando o usuário move o mouse para fora do `mousePane1`, o aplicativo muda a cor de fundo para branco. A linha 25 anexa o `mousePane1` ao `JFrame`. Como vimos, você normalmente tem de especificar o layout dos componentes GUI em um `JFrame`. Nessa seção, introduzimos o gerenciador de layout `FlowLayout`. Aqui usamos o layout padrão do painel de conteúdo de um `JFrame` — **`BorderLayout`**, que organiza as regiões **`NORTH`**, **`SOUTH`**, **`EAST`**, **`WEST`** e **`CENTER`** do componente. **`NORTH`** corresponde ao topo do contêiner. Esse exemplo utiliza as regiões **`CENTER`** e **`SOUTH`**. A linha 25 utiliza uma versão de dois argumentos do método `add` para colocar `mousePane1` na região **`CENTER`**. O **`BorderLayout`** dimensiona o componente no **`CENTER`** automaticamente para utilizar todo o espaço em `JFrame` que não é ocupado pelos componentes nas outras regiões. A Seção 12.18.2 discute **`BorderLayout`** em mais detalhes.

As linhas 27 e 28 no construtor declaram `JLabel statusBar` e o anexam à região **`SOUTH`** de `JFrame`. Esse `JLabel` ocupa a largura do `JFrame`. A altura da região é determinada pelo `JLabel`.

A linha 31 cria uma instância da classe interna `MouseHandler` (linhas 36 a 97) chamada `handler`, que responde aos eventos de mouse. As linhas 32 e 33 registram `handler` como o ouvinte de eventos de mouse do `mousePane1`. Os métodos **`addMouseListener`** e **`addMouseMotionListener`** são herdados indiretamente da classe `Component` e podem ser utilizados para registrar `MouseListeners` e `MouseMotionListeners`, respectivamente. Um objeto `MouseHandler` é um `MouseListener` e é um `MouseMotionListener` porque a classe implementa as duas interfaces. Optamos por implementar ambas as interfaces aqui para demonstrar uma classe que implementa mais de uma interface, mas em vez disso poderíamos ter implementado a interface `MouseListener`.

Quando o mouse entrar e sair da área de `mousePane1`, os métodos `mouseEntered` (linhas 65 a 71) e `mouseExited` (linhas 74 a 79) são chamados, respectivamente. O método `mouseEntered` exibe uma mensagem no `statusBar` que indica que o mouse entrou em `JPanel` e muda a cor de fundo para verde. O método `mouseExited` exibe uma mensagem no `statusBar` que indica que o mouse está fora do `JPanel` (ver a primeira janela de saída de exemplo) e muda a cor de fundo para branco.

Os outros cinco eventos exibem uma sequência na `statusBar` que inclui o evento e as coordenadas em que ele ocorreu. Os métodos `MouseEvent` **`getX`** e **`getY`** retornam as coordenadas *x* e *y*, respectivamente, do mouse no momento em que o evento ocorreu.

12.15 Classes de adaptadores

Muitas interfaces ouvintes de evento, como `MouseListener` e `MouseMotionListener`, contêm múltiplos métodos. Não é sempre desejável declarar cada método em uma interface ouvinte de evento. Por exemplo, um aplicativo pode precisar somente da rotina de tratamento `mouseClicked` de `MouseListener` ou da rotina de tratamento `mouseDragged` de `MouseMotionListener`. A interface `WindowListener` especifica sete métodos de tratamento de evento de janela. Para muitas dessas interfaces ouvintes que têm múltiplos métodos, os pacotes `java.awt.event` e `javax.swing.event` fornecem classes de adaptadores de ouvinte de evento. Uma **classe de adaptadores** implementa uma interface e fornece uma implementação padrão (com o corpo de um método vazio) de cada método na interface. A Figura 12.30 mostra várias classes de adaptadores `java.awt.event` e as interfaces que elas implementam. Você pode estender uma classe de adaptadores para herdar a implementação padrão de cada método e, subsequentemente, sobrescrever somente o(s) método(s) necessário(s) para o tratamento de evento.

Classe de adaptadores de evento em <code>java.awt.event</code>	Implementa a interface
<code>ComponentAdapter</code>	<code>ComponentListener</code>
<code>ContainerAdapter</code>	<code>ContainerListener</code>
<code>FocusAdapter</code>	<code>FocusListener</code>
<code>KeyAdapter</code>	<code>KeyListener</code>
<code>MouseAdapter</code>	<code>MouseListener</code>
<code>MouseMotionAdapter</code>	<code>MouseMotionListener</code>
<code>WindowAdapter</code>	<code>WindowListener</code>

Figura 12.30 | As classes de adaptadores de evento e as interfaces que elas implementam.



Observação de engenharia de software 12.6

Quando uma classe implementar uma interface, a classe tem 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 de adaptadores de evento é um objeto do tipo ouvinte de eventos correspondente (por exemplo, um objeto de uma subclasse de `MouseAdapter` é um `MouseListener`).

Estendendo `MouseAdapter`

O aplicativo das figuras 12.31 e 12.32 demonstra como determinar o número de cliques de mouse (isto é, a contagem de cliques) e como distinguir entre os diferentes botões do mouse. O ouvinte de eventos nesse aplicativo é um objeto da classe interna `MouseClickedHandler` (Figura 12.31, linhas 25 a 46) que estende `MouseAdapter`, então podemos declarar somente o método `mouseClicked` de que precisamos nesse exemplo.

```

1  // Figura 12.31: MouseDetailsFrame.java
2  // Demonstrando cliques de mouse e distinguindo entre botões do mouse.
3  import java.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; // String exibida na statusBar
12     private final JLabel statusBar; // JLabel 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 tratamento de evento
22     }
23
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         @Override
29         public void mouseClicked(MouseEvent event)
30         {
31             int xPos = event.getX(); // obtém a posição x do mouse
32             int yPos = event.getY(); // obtém a posição y do mouse
33         }
34     }
35 }

```

continua

```

34         details = String.format("Clicked %d time(s)",
35                                 event.getClickCount());
36
37         if (event.isMetaDown()) // botão direito do mouse
38             details += " with right mouse button";
39         else if (event.isAltDown()) // botão do meio do mouse
40             details += " with center mouse button";
41         else // botão esquerdo do mouse
42             details += " with left mouse button";
43
44         statusBar.setText(details); // exibe mensagem em statusBar
45     }
46 }
47 } // fim da classe MouseDetailsFrame

```

Figura 12.31 | Demonstrando cliques de mouse e distinguindo entre botões do mouse.

```

1 // Figura 12.32: 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);
12        mouseDetailsFrame.setVisible(true);
13    }
14 } // fim da classe MouseDetails

```

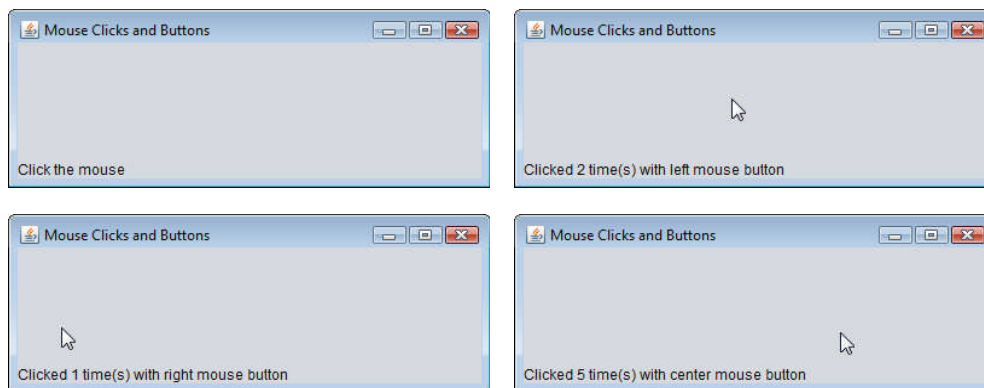


Figura 12.32 | Testando MouseDetailsFrame.



Erro comum de programação 12.3

Se você estender uma classe de adaptadores e digitar incorretamente o nome do método que está sendo sobrescrito, e se o método `@Override` não for declarado, seu método simplesmente se tornará 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 de adaptadores.

Um usuário de um programa Java pode estar em um sistema com um, dois ou três botões do mouse. A classe `MouseEvent` herda vários métodos da classe `InputEvent` que podem distinguir entre o botão do mouse em um mouse de múltiplos botões ou simular um mouse de múltiplos botões com uma combinação de um clique do teclado e um clique de botão do mouse. A Figura 12.33 mostra os métodos `InputEvent` utilizados para distinguir os cliques do botão do mouse. O Java assume que cada mouse contém um botão esquerdo. Portanto, é simples testar um clique com o botão esquerdo do mouse. Entretanto, os usuários com mouse de um ou dois

Método InputEvent	Descrição
<code>isMetaDown()</code>	Retorna <code>true</code> quando o usuário clica no <i>botão direito do mouse</i> 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 <i>Meta</i> no teclado e clicar no botão do mouse.
<code>isAltDown()</code>	Retorna <code>true</code> quando o usuário clica no <i>botão do meio do mouse</i> 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 <i>Alt</i> e clicar no único botão ou no botão esquerdo do mouse, respectivamente.

Figura 12.33 | Métodos InputEvent que ajudam a determinar se o botão direito ou central do mouse foi pressionado.

botões devem utilizar uma combinação de pressionamentos de tecla e cliques do botão do mouse para simular os botões ausentes no 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 com botão único em um mouse de um botão. No caso de um mouse de um botão, um aplicativo Java supõe que o botão direito do mouse é clicado se o usuário mantiver pressionada a tecla *Meta* (às vezes chamada tecla *Command* ou tecla “Apple” em um Mac) e clicar com o botão do mouse.

A linha 21 da Figura 12.31 registra um `MouseListener` para o `MouseDetailsFrame`. O ouvinte de eventos é um objeto da classe `MouseClickedHandler`, que estende `MouseAdapter`. Isso permite declarar apenas o método `mouseClicked` (linhas 28 a 45). Esse método primeiro captura as coordenadas em que o evento ocorreu e as armazena nas variáveis locais `xPos` e `yPos` (linhas 31 e 32). As linhas 34 e 35 criam uma `String` chamada `details` contendo o número de cliques consecutivos de mouse, que é retornado pelo método `MouseEvent` `getClickCount` na linha 35. As linhas 37 a 42 utilizam os métodos `isMetaDown` e `isAltDown` para determinar o botão do mouse em que o usuário clicou e acrescentar uma `String` adequada aos `details` em cada caso. A `String` resultante é exibida em `statusBar`. A classe `MouseDetails` (Figura 12.32) contém o método `main` que executa o aplicativo. Tente clicar repetidamente com cada um dos botões do mouse para ver o incremento de contagem de cliques.

12.16 Subclasse `JPanel` para desenhar com o mouse

A Seção 12.14 mostrou como monitorar eventos de mouse em um `JPanel`. Nesta seção, utilizamos um `JPanel` como uma **área dedicada de desenho** em que o usuário pode desenhar arrastando o mouse. Além disso, esta seção demonstra um ouvinte de eventos que estende uma classe de adaptadores.

Método `paintComponent`

Os componentes Swing leves que estendem a classe `JComponent` (como `JPanel`) contêm o método `paintComponent`, que é chamado quando um componente Swing leve é exibido. Por sobrescrever esse método, você pode especificar como desenhar formas utilizando capacidades gráficas do Java. Ao personalizar um `JPanel` para utilização como uma área dedicada de desenho, a subclasse deve sobrescrever o método `paintComponent` e chamar a versão de superclasse de `paintComponent` como a primeira instrução no corpo do método sobrescrito para assegurar que o componente será exibido corretamente. A razão é que subclasses de `JComponent` suportam **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 for transparente, `paintComponent` não limpará seu fundo quando o programa exibir o componente. Quando um componente for **opaco**, `paintComponent` limpará o fundo do componente antes de o componente ser exibido. A transparência de um componente Swing leve pode ser configurada com o método `setOpaque` (um argumento `false` indica que o componente é transparente).



Dica de prevenção de erro 12.1

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 12.4

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.

Definindo a área personalizada de desenho

O aplicativo Painter das figuras 12.34 e 12.35 demonstra uma subclasse personalizada de `JPanel` que é usada para criar uma área de desenho dedicada. O aplicativo utiliza a rotina de tratamento de evento `mouseDragged` para criar um aplicativo de desenho simples. O usuário pode desenhar figuras arrastando o mouse sobre o `JPanel`. Esse exemplo não usa o método `mouseMoved`, assim nossa classe ouvida de evento (a classe interna anônima nas linhas 20 a 29 da Figura 12.34) estende `MouseMotionAdapter`. Visto que essa classe já declara tanto `mouseMoved` como `mouseDragged`, podemos simplesmente sobrescrever `mouseDragged` para fornecer o tratamento de evento requerido por esse aplicativo.

```

1  // Figura 12.34: PaintPanel.java
2  // Classe de adaptadores utilizada para implementar rotinas de tratamento de evento.
3  import java.awt.Point;
4  import java.awt.Graphics;
5  import java.awt.event.MouseEvent;
6  import java.awt.event.MouseMotionAdapter;
7  import java.util.ArrayList;
8  import javax.swing.JPanel;
9
10 public class PaintPanel extends JPanel
11 {
12     // lista das referências Point
13     private final ArrayList<Point> points = new ArrayList<>();
14
15     // configura GUI e registra rotinas de tratamento de evento de mouse
16     public PaintPanel()
17     {
18         // trata evento de movimento de mouse do frame
19         addMouseListener(
20             new MouseMotionAdapter() // classe interna anônima
21             {
22                 // armazena coordenadas da ação de arrastar e repinta
23                 @Override
24                 public void mouseDragged(MouseEvent event)
25                 {
26                     points.add(event.getPoint());
27                     repaint(); // repinta JFrame
28                 }
29             }
30         );
31     }
32
33     // desenha ovais em um quadro delimitador de 4 x 4 nas localizações especificadas na janela
34     @Override
35     public void paintComponent(Graphics g)
36     {
37         super.paintComponent(g); // limpa a área de desenho
38
39         // desenha todos os pontos
40         for (Point point : points)
41             g.fillOval(point.x, point.y, 4, 4);
42     }
43 } // fim da classe PaintPanel

```

Figura 12.34 | Classe de adaptadores utilizada para implementar rotinas de tratamento de evento.

A classe `PaintPanel` (Figura 12.34) estende `JPanel` para criar a área dedicada de desenho. 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. Nesse exemplo, utilizamos um `ArrayList` de `Points` (linha 13) para armazenar a localização em que cada evento de arrastar com o mouse ocorre. Como você verá, o método `paintComponent` utiliza esses `Points` para desenhar.

As linhas 19 a 30 registram um `MouseListener` para ouvir os eventos de movimento do mouse `PaintPanel`. As linhas 20 a 29 criam um objeto de uma classe interna anônima que estende a classe de adaptadores `MouseMotionAdapter`. Lembre-se de que `MouseMotionAdapter` implementa `MouseListener`, portanto, o objeto *anônimo de classe interna* é um

MouseMotionListener. A classe interna anônima herda as implementações mouseMoved e mouseDragged padrão, assim ela já implementa todos os métodos da interface. Mas as implementações padrão não fazem nada quando são chamadas. Portanto, sobrescrevemos o método mouseDragged nas linhas 23 a 28 para capturar as coordenadas de um evento de arrastar com o mouse e as armazenamos como um objeto Point. A linha 26 invoca o método `getPoint` de MouseEvent para obter o Point onde o evento ocorreu e o armazena na ArrayList. A linha 27 chama o método `repaint` (herdado indiretamente da classe Component) para indicar que o PaintPanel deve ser atualizado na tela o mais rápido possível com uma chamada para o método `paintComponent` de PaintPanel.

O método `paintComponent` (linhas 34 a 42), que recebe um parâmetro Graphics, é chamado automaticamente a qualquer hora que PaintPanel precisar ser exibido na tela — como quando a GUI é inicialmente exibida — ou atualizado na tela — como quando o método `repaint` é chamado ou quando o componente GUI foi *oculto* por outra janela na tela e subsequentemente se torna visível novamente.



Observação sobre a aparência e comportamento 12.13

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

A linha 37 invoca a versão de superclasse de `paintComponent` para limpar o fundo de PaintPanel (JPanels são opacos por padrão). As linhas 40 e 41 desenhavam uma oval no local especificado por cada Point na ArrayList. O método `fillOval` Graphics desenha uma oval sólida. Os 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 coordenadas representam a largura e a altura da área retangular. O método `fillOval` desenha a oval de tal modo que ela toque no meio de cada lado da área retangular. Na linha 41, os dois primeiros argumentos são especificados utilizando as duas variáveis de instância public da classe Point — *x* e *y*. Você aprenderá mais sobre os recursos Graphics no Capítulo 13.



Observação sobre a aparência e comportamento 12.14

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.

Utilizando o JPanel personalizado em um aplicativo

A classe Painter (Figura 12.35) contém o método `main` que executa esse aplicativo. A linha 14 cria um objeto PaintPanel em que o usuário pode arrastar o mouse para desenhar. A linha 15 anexa o PaintPanel ao JFrame.

```

1 // Figura 12.35: Painter.java
2 // Testando o PaintPanel.
3 import java.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();
15        application.add(paintPanel, BorderLayout.CENTER);
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);
23        application.setVisible(true);

```

continua

```

24     }
25 } // fim da classe Painter

```

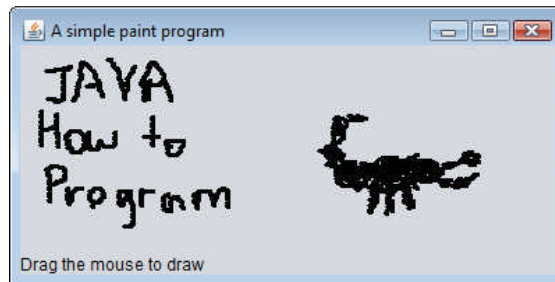


Figura 12.35 | Testando PaintPanel.

12.17 Tratamento de eventos de teclado

Esta seção apresenta a interface `KeyListener` para tratar **eventos de teclado**, que são gerados quando as teclas no teclado são pressionadas e liberadas. Uma classe que implementa `KeyListener` deve fornecer declarações para métodos **`keyPressed`**, **`keyReleased`** e **`keyTyped`**, cada um dos quais 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 **tecla de ação**. (As teclas de ação são qualquer seta, *Home*, *End*, *Page Up*, *Page Down*, qualquer tecla de função etc.) O método `keyReleased` é chamado quando a tecla é liberada após qualquer evento `keyPressed` ou `keyTyped`.

O aplicativo das figuras 12.36 e 12.37 demonstra os métodos `KeyListener`. A classe `KeyDemoFrame` implementa a interface `KeyListener`, então todos os três métodos são definidos no aplicativo. O construtor (Figura 12.36, linhas 17 a 28) registra o aplicativo para tratar seus próprios eventos de teclado utilizando o método **`addKeyListener`** na linha 27. O método `addKeyListener` é declarado na classe `Component`, então cada subclasse de `Component` pode notificar objetos `KeyListener` de eventos de teclado desse `Component`.

```

1  // Figura 12.36: KeyDemoFrame.java
2  // Tratamento de evento de teclado.
3  import java.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 final String line1 = ""; // primeira linha da área de texto
12     private final String line2 = ""; // segunda linha da área de texto
13     private final String line3 = ""; // terceira linha da área de texto
14     private final JTextArea textArea; // área de texto para exibir a 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);
24         textArea.setDisabledTextColor(Color.BLACK);
25         add(textArea); // adiciona área de texto ao JFrame
26
27         addKeyListener(this); // permite que o frame processe os eventos de teclado
28     }
29
30     // trata pressionamento de qualquer tecla

```

continuação

```

31  @Override
32  public void keyPressed(KeyEvent event)
33  {
34      line1 = String.format("Key pressed: %s",
35          KeyEvent.getKeyText(event.getKeyCode())); // mostra a tecla pressionada
36      setLines2and3(event); // configura a saída das linhas dois e três
37  }
38
39  // trata liberação de qualquer tecla
40  @Override
41  public void keyReleased(KeyEvent event)
42  {
43      line1 = String.format("Key released: %s",
44          KeyEvent.getKeyText(event.getKeyCode())); // mostra a tecla liberada
45      setLines2and3(event); // configura a saída das linhas dois e três
46  }
47
48  // trata pressionamento de uma tecla de ação
49  @Override
50  public void keyTyped(KeyEvent event)
51  {
52      line1 = String.format("Key typed: %s", event.getKeyChar());
53      setLines2and3(event); // configura a saída das linhas dois e três
54  }
55
56  // configura segunda e terceira linhas de saída
57  private void setLines2and3(KeyEvent event)
58  {
59      line2 = String.format("This key is %san action key",
60          (event.isActionKey() ? "" : "not "));
61
62      String temp = KeyEvent.getKeyModifiersText(event.getModifiers());
63
64      line3 = String.format("Modifier keys pressed: %s",
65          (temp.equals("") ? "none" : temp)); // modificadores de saída
66
67      textArea.setText(String.format("%s\n%s\n%s\n",
68          line1, line2, line3)); // gera saída de três linhas de texto
69  }
70 } // fim da classe KeyDemoFrame

```

Figura 12.36 | Tratamento de evento de teclado.

Na linha 25, o construtor adiciona `JTextArea textArea` (onde a saída do aplicativo é exibida) ao `JFrame`. Uma `JTextArea` é uma *área multilinha* em que você pode exibir texto. (Discutimos `JTextAreas` em mais detalhes na Seção 12.20.) Note nas capturas de tela que `textArea` ocupa a *janela inteira*. Isso ocorre por causa da `BorderLayout` padrão do `JFrame` (discutida na Seção 12.18.2 e demonstrada na Figura 12.41). Quando um único Component é adicionado a um `BorderLayout`, o Component ocupa o Container *inteiro*. A linha 23 desativa a `JTextArea` para que o usuário não possa digitar nela. Isso faz com que o texto na `JTextArea` torne-se cinza. A linha 24 utiliza o método `setDisabledTextColor` para mudar a cor do texto na `JTextArea` para preto a fim de facilitar a legibilidade.

Os métodos `keyPressed` (linhas 31 a 37) e `keyReleased` (linhas 40 a 46) utilizam o método `KeyEvent` `getKeyCode` para obter o **código de tecla virtual** da tecla pressionada. A classe `KeyEvent` contém constantes de código de tecla virtual que representam cada tecla no teclado. Essas constantes podem ser comparadas com o valor de retorno de `getKeyCode` para testar as teclas individuais no teclado. O valor retornado por `getKeyCode` é passado para o método static `KeyEvent` `getKeyText`, que retorna uma string contendo o nome da tecla que foi pressionada. Para obter uma lista completa das constantes de tecla virtual, consulte na documentação on-line a classe `KeyEvent` (pacote `java.awt.event`). O método `keyTyped` (linhas 49 a 54) utiliza o método `KeyEvent` `getKeyChar` (que retorna um `char`) para obter o valor Unicode do caractere digitado.

Todos os três métodos de tratamento de evento terminam chamando o método `setLines2and3` (linhas 57 a 69) e passando para ele o objeto `KeyEvent`. Esse método utiliza o método `KeyEvent` `isActionKey` (linha 60) para determinar se a tecla no evento é uma tecla de ação. Além disso, o método `InputEvent` `getModifiers` é chamado (linha 62) para determinar se alguma tecla modificadora (como *Shift*, *Alt* e *Ctrl*) foi pressionada quando o evento de teclado ocorreu. O resultado desse método é passado para o método static `KeyEvent` `getKeyModifiersText`, que produz uma `String` contendo os nomes das teclas modificadoras pressionadas.


```

1 // Figura 12.37: 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);
12        keyDemoFrame.setVisible(true);
13    }
14 } // fim da classe KeyDemo

```



Figura 12.37 | Testando KeyDemoFrame.

[*Observação:* se você precisar testar uma tecla específica no teclado, a classe `KeyEvent` fornece uma **tecla constante** para cada uma delas. Essas constantes podem ser utilizadas a partir das rotinas de tratamento de evento de teclado para determinar se uma tecla particular foi pressionada. Além disso, para determinar se as teclas *Alt*, *Ctrl*, *Meta* e *Shift* são pressionadas individualmente, os métodos `InputEvent` **`isAltDown`**, **`isControlDown`**, **`isMetaDown`** e **`isShiftDown`** retornam um `boolean` indicando se a tecla particular foi pressionada durante o evento de teclado.]

12.18 Introdução a gerenciadores de layout

Os **gerenciadores de layout** *organizam* componentes GUI em um contêiner para propósitos de apresentação. Utilize os gerenciadores de layout para obter capacidades básicas de layout, em vez de determinar a posição e o tamanho exatos de cada componente GUI. Essa funcionalidade permite que você se concentre na aparência e comportamento básicos e deixa os gerenciadores de layout processarem a maioria dos detalhes de layout. Todos os gerenciadores de layout implementam a interface **`LayoutManager`** (no pacote `java.awt`). O método `setLayout` da classe `Container` aceita um objeto que implementa a interface `LayoutManager` como um argumento. Há basicamente três maneiras de organizar componentes em uma GUI:

1. *Posicionamento absoluto:* ele fornece o maior nível de controle sobre a aparência de uma GUI. Configurando o layout de um `Container` como `null`, você pode especificar a *posição absoluta de cada componente GUI* em relação ao canto superior esquerdo do `Container` usando métodos `Component` `setSize` e `setLocation` ou `setBounds`. Se fizer isso, você também deve especificar o tamanho de cada componente GUI. A programação de uma GUI com posicionamento absoluto pode ser tediosa, a menos que você tenha um ambiente de desenvolvimento integrado (IDE) que pode gerar o código para você.

2. *Gerenciadores de layout*: usar gerenciadores de layout para posicionar elementos pode ser mais simples e mais rápido do que criar uma GUI com posicionamento absoluto, e torna suas GUIs mais redimensionáveis, mas você perde parte do controle em relação ao tamanho e posicionamento precisos de cada componente.
3. *Programação visual em um IDE*: os IDEs fornecem ferramentas que facilitam a criação de GUIs. Em geral, todo IDE fornece uma **ferramenta de desenho GUI** que permite arrastar e soltar componentes GUI de uma caixa de ferramenta em uma área de desenho. Você então pode posicionar, dimensionar e alinhar componentes GUI como quiser. O IDE gera o código Java que cria a GUI. Além disso, em geral, você pode adicionar o código de tratamento de evento de um componente particular dando um clique duplo no componente. Algumas ferramentas de desenho também permitem utilizar os gerenciadores de layout descritos neste capítulo e no Capítulo 22.



Observação sobre a aparência e comportamento 12.15

A maioria dos IDEs Java fornece ferramentas de design de GUI 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 12.16

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 e tomar cuidado no sentido 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`.

A Figura 12.38 resume os gerenciadores de layout apresentados neste capítulo. Alguns gerenciadores de layout adicionais serão discutidos no Capítulo 22.

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

Figura 12.38 | Os gerenciadores de layout.

12.18.1 `FlowLayout`

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

O aplicativo das figuras 12.39 e 12.40 cria três objetos `JButton` e os adiciona ao aplicativo, utilizando um `FlowLayout`. Os componentes são centralizados por padrão. Quando o usuário clica em **Left**, o alinhamento do `FlowLayout` é alterado para alinhado à esquerda. Quando o usuário clica em **Right**, o alinhamento do `FlowLayout` é alterado para alinhado à direita. Quando o usuário clica em **Center**, o alinhamento do `FlowLayout` é alterado para alinhado no centro. As janelas de saída de exemplo mostram cada alinhamento. O último exemplo de saída mostra o alinhamento centralizado depois que a janela foi redimensionada para uma largura menor de modo que o botão **Right** flua em uma nova linha.

Como visto anteriormente, um layout do contêiner é configurado com o método `setLayout` da classe `Container`. A linha 25 (Figura 12.39) configura o gerenciador de layout como o `FlowLayout` declarado na linha 23. Normalmente, o layout é configurado antes de qualquer componente GUI ser adicionado a um contêiner.



Observação sobre a aparência e comportamento 12.17

Cada contêiner individual só pode ter um gerenciador de layout, mas múltiplos contêineres no mesmo aplicativo podem usar diferentes gerenciadores de layout.

```

1  // Figura 12.39: FlowLayoutFrame.java
2  // FlowLayout permite que os componentes fluam ao longo de múltiplas linhas.
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 final JButton leftJButton; // botão para configurar alinhamento à esquerda
13     private final JButton centerJButton; // botão para configurar alinhamento centralizado
14     private final JButton rightJButton; // botão para configurar alinhamento à direita
15     private final FlowLayout layout; // objeto de layout
16     private final Container container; // contêiner para configurar layout
17
18     // configura GUI e registra listeners de botão
19     public FlowLayoutFrame()
20     {
21         super("FlowLayout Demo");
22
23         layout = new FlowLayout();
24         container = getContentPane(); // obtém contêiner para layout
25         setLayout(layout);
26
27         // configura leftJButton e registra listener
28         leftJButton = new JButton("Left");
29         add(leftJButton); // adiciona o botão Left ao frame
30         leftJButton.addActionListener(
31             new ActionListener() // classe interna anônima
32             {
33                 // processa o evento leftJButton
34                 @Override
35                 public void actionPerformed(ActionEvent event)
36                 {
37                     layout.setAlignment(FlowLayout.LEFT);
38
39                     // realinha os componentes anexados
40                     layout.layoutContainer(container);
41                 }
42             }
43         );
44
45         // configura centerJButton e registra o listener
46         centerJButton = new JButton("Center");
47         add(centerJButton); // adiciona botão Center ao frame
48         centerJButton.addActionListener(
49             new ActionListener() // classe interna anônima
50             {
51                 // processa evento centerJButton
52                 @Override
53                 public void actionPerformed(ActionEvent event)
54                 {
55                     layout.setAlignment(FlowLayout.CENTER);
56
57                     // realinha os componentes anexados
58                     layout.layoutContainer(container);
59                 }
60             }
61         );
62     }
63 }

```

continua

continuação

```

60     }
61 };
62
63 // configura rightJButton e registra o listener
64 rightJButton = new JButton("Right");
65 add(rightJButton); // adiciona botão Right ao frame
66 rightJButton.addActionListener(
67     new ActionListener() // classe interna anônima
68     {
69         // processa evento rightJButton
70         @Override
71         public void actionPerformed(ActionEvent event)
72         {
73             layout.setAlignment(FlowLayout.RIGHT);
74
75             // realinha os componentes anexados
76             layout.layoutContainer(container);
77         }
78     }
79 );
80 } // fim do construtor FlowLayoutFrame
81 } // fim da classe FlowLayoutFrame

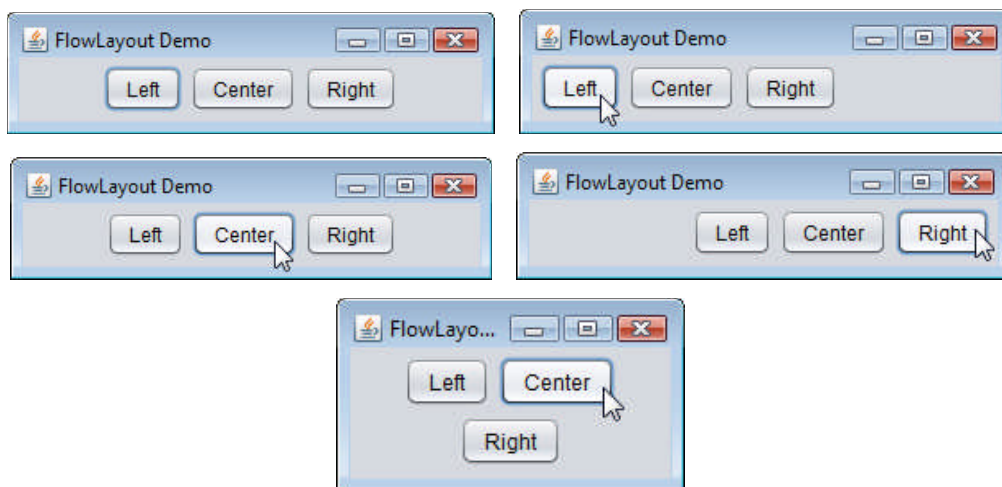
```

Figura 12.39 | FlowLayout permite que os componentes fluam sobre múltiplas linhas.

```

1 // Figura 12.40: 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);
12        flowLayoutFrame.setVisible(true);
13    }
14 } // fim da classe FlowLayoutDemo

```

**Figura 12.40** | Testando FlowLayoutFrame.

A rotina de tratamento de evento de cada botão é especificada com um objeto da classe interna anônima separada (linhas 30 a 43, 48 a 61 e 66 a 79, respectivamente), e o método `actionPerformed` em cada caso executa duas instruções. Por exemplo, a linha 37

na rotina de tratamento de eventos para `leftJButton` utiliza o método `FlowLayout` `setAlignment` para mudar o alinhamento do `FlowLayout` para um `FlowLayout` alinhado à esquerda (`FlowLayout.LEFT`). A linha 40 utiliza o método `layoutContainer` (que é herdado por todos os gerenciadores de layout) da interface `LayoutManager` para especificar que o `JFrame` deve ser reorganizado com base no layout ajustado. Dependendo do botão clicado, o método `actionPerformed` de cada botão configura o alinhamento de `FlowLayout` como `FlowLayout.LEFT` (linha 37), `FlowLayout.CENTER` (linha 55) ou `FlowLayout.RIGHT` (linha 73).

12.18.2 BorderLayout

O gerenciador de layout `BorderLayout` (o gerenciador de layout padrão de um `JFrame`) organiza componentes em cinco regiões: `NORTH`, `SOUTH`, `EAST`, `WEST` e `CENTER`. `NORTH` corresponde à parte superior do contêiner. A classe `BorderLayout` estende `Object` e implementa a interface `LayoutManager2` (uma subinterface de `LayoutManager` que adiciona vários métodos para obter um processamento de layout aprimorado).

Um `BorderLayout` limita um `Container` a conter *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. Os componentes colocados nas regiões `NORTH` e `SOUTH` estendem-se horizontalmente para os lados do contêiner e têm a mesma altura que o componente mais alto colocado nessas regiões. As regiões `EAST` e `WEST` expandem verticalmente entre as regiões `NORTH` e `SOUTH` e são tão largas quanto os componentes colocados nessas regiões. O componente colocado na região `CENTER` *expande para preencher todo o espaço restante no layout* (que é a razão de `JTextArea` na Figura 12.37 ocupar a janela inteira). Se todas as cinco regiões são ocupadas, o espaço do contêiner inteiro é coberto por componentes GUI. Se a região `NORTH` ou `SOUTH` não for ocupada, os componentes GUI nas regiões `EAST`, `CENTER` e `WEST` expandem verticalmente para preencher o espaço restante. Se a região `EAST` ou `WEST` não for ocupada, o componente GUI na região `CENTER` *expande horizontalmente para preencher o espaço restante*. Se a região `CENTER` não for ocupada, a área é deixada *vazia* — os outros componentes GUI *não* se expandem para preencher o espaço restante. O aplicativo das figuras 12.41 e 12.42 demonstra o gerenciador de layout `BorderLayout` utilizando cinco `JButtons`.

```

1 // Figura 12.41: BorderLayoutFrame.java
2 // BorderLayout contendo cinco botões.
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 final 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 final BorderLayout layout;
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);
23         buttons = new JButton[names.length];
24
25         // cria JButtons e registra ouvintes 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         }
31
32         add(buttons[0], BorderLayout.NORTH);
33         add(buttons[1], BorderLayout.SOUTH);
34         add(buttons[2], BorderLayout.EAST);
35         add(buttons[3], BorderLayout.WEST);
36         add(buttons[4], BorderLayout.CENTER);

```

continua

continuação

```

37     }
38
39     // trata os eventos de botão
40     @Override
41     public void actionPerformed(ActionEvent event)
42     {
43         // verifica a origem de evento e o painel de conteúdo de layout de acordo
44         for (JButton button : buttons)
45         {
46             if (event.getSource() == button)
47                 button.setVisible(false); // oculta o botão que foi clicado
48             else
49                 button.setVisible(true); // mostra outros botões
50         }
51
52         layout.layoutContainer(getContentPane()); // define o layout do painel de conteúdo
53     }
54 } // fim da classe BorderLayoutFrame

```

Figura 12.41 | BorderLayout que contém cinco botões.

A linha 21 da Figura 12.41 cria um BorderLayout. Os argumentos de construtor especificam o número de pixels entre componentes que estão organizados horizontalmente (**espaçamento horizontal**) e entre componentes que são organizados verticalmente (**espaçamento vertical**), respectivamente. O padrão tem horizontal e verticalmente um pixel de espaçamento. A linha 22 usa o método `setLayout` para definir o layout do painel de conteúdo como `layout`.

Adicionamos Components a um BorderLayout com outra versão do método `Container add` que aceita dois argumentos — o Component para adicionar e a região em que o Component deva aparecer. Por exemplo, a linha 32 especifica que `buttons[0]` deve aparecer na região NORTH. Os componentes podem ser adicionados em *qualquer* ordem, mas apenas *um* componente deve ser adicionado a cada região.



Observação sobre a aparência e comportamento 12.18

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



Erro comum de programação 12.5

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.

A classe `BorderLayoutFrame` implementa `ActionListener` diretamente nesse exemplo, então `BorderLayoutFrame` tratará os eventos de `JButtons`. Por essa razão, a linha 29 passa a referência `this` para o método `addActionListener` de cada `JButton`. Quando o usuário clica em um `JButton` específico no layout, o método `actionPerformed` (linhas 40 a 53) é executado. A instrução `for` aprimorada nas linhas 44 a 50 utiliza um `if...else` para ocultar o `JButton` particular que gerou o evento. O método `setVisible` (herdado em `JButton` da classe `Component`) é chamado com um argumento `false` (linha 47) para ocultar `JButton`. Se o `JButton` atual no array não é o que gerou o evento, o método `setVisible` é chamado com um argumento `true` (linha 49) para assegurar que o `JButton` é exibido na tela. A linha 52 utiliza o método `LayoutManager layoutContainer` para recalcular o layout do painel de conteúdo. Note nas capturas de tela da Figura 12.42 que certas regiões no BorderLayout alteram a forma quando os `JButtons` são *ocultados* e exibidos em outras regiões. Tente redimensionar a janela do aplicativo para ver como as várias regiões se redimensionam com base na largura e altura da janela. *Para layouts mais complexos, agrupe componentes em `JPanels`, cada um com um gerenciador de layout separado.* Coloque os `JPanels` no `JFrame` utilizando BorderLayout padrão ou algum outro layout.

```

1 // Figura 12.42: BorderLayoutDemo.java
2 // Testando BorderLayoutFrame.
3 import javax.swing.JFrame;
4
5 public class BorderLayoutDemo

```

continua

```

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);
12         borderLayoutFrame.setVisible(true);
13     }
14 } // fim da classe BorderLayoutDemo

```

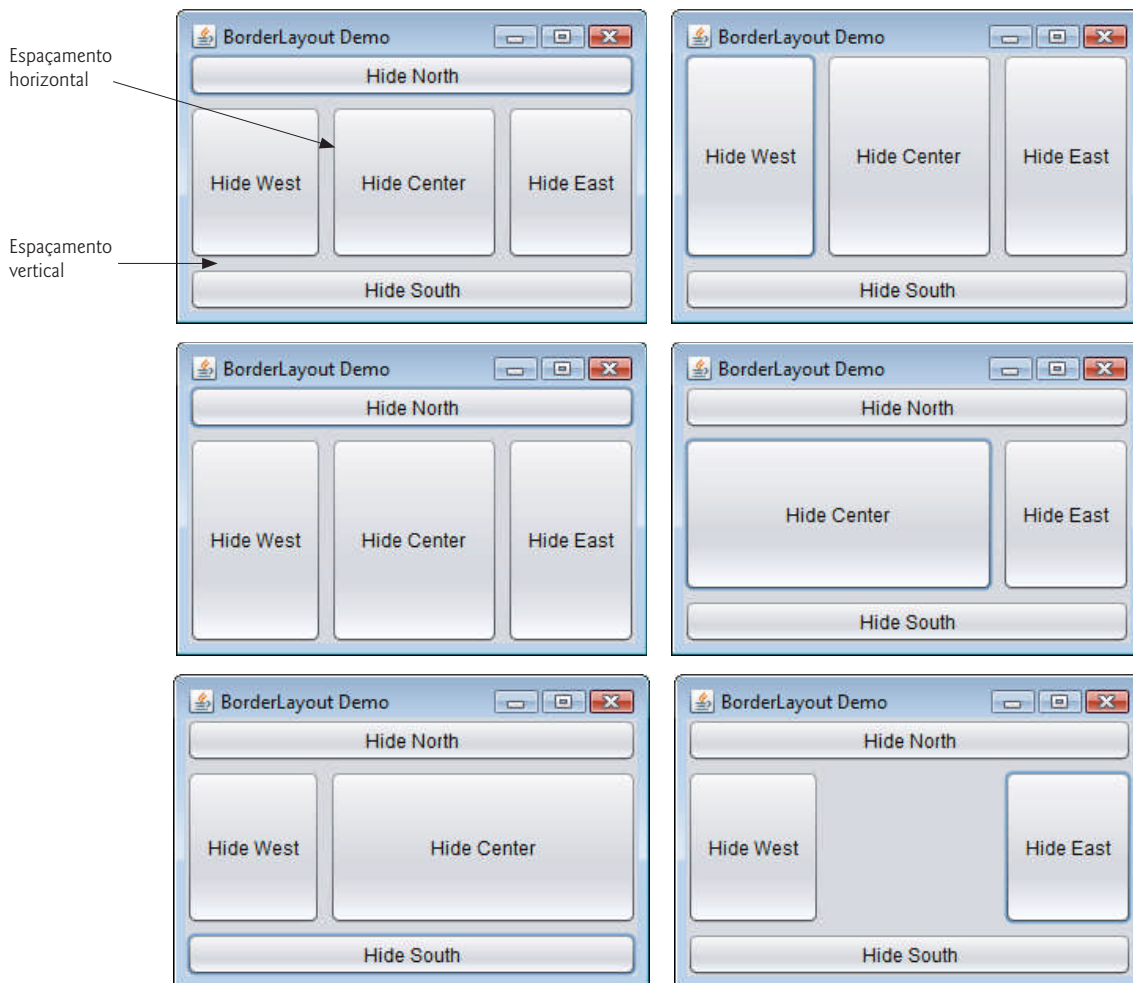


Figura 12.42 | Testando BorderLayoutFrame.

12.18.3 GridLayout

O gerenciador de layout **GridLayout** divide o contêiner em uma *grade* de modo que os componentes podem ser colocados nas *linhas* e *colunas*. A classe `GridLayout` herda diretamente da classe `Object` e implementa a interface `LayoutManager`. Cada `Component` em um `GridLayout` tem a *mesma* largura e altura. Os componentes são adicionados a um `GridLayout` iniciando a célula na 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 aplicativo das figuras 12.43 e 12.44 demonstra o gerenciador de layout `GridLayout` utilizando seis `JButtons`.

```

1  // Figura 12.43: GridLayoutFrame.java
2  // GridLayout contendo seis botões.
3  import java.awt.GridLayout;
4  import java.awt.Container;

```

continuação

```

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 final 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 final Container container; // contêiner do frame
17     private final GridLayout gridLayout1; // primeiro gridlayout
18     private final GridLayout gridLayout2; // segundo gridlayout
19
20     // construtor sem argumento
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();
27         setLayout(gridLayout1);
28         buttons = new JButton[names.length];
29
30         for (int count = 0; count < names.length; count++)
31         {
32             buttons[count] = new JButton(names[count]);
33             buttons[count].addActionListener(this); // ouvinte registrado
34             add(buttons[count]); // adiciona o botão ao JFrame
35         }
36     }
37
38     // trata eventos de botão alternando entre layouts
39     @Override
40     public void actionPerformed(ActionEvent event)
41     {
42         if (toggle) // define layout com base nos botões de alternção
43             container.setLayout(gridLayout2);
44         else
45             container.setLayout(gridLayout1);
46
47         toggle = !toggle;
48         container.validate(); // refaz o layout do contêiner
49     }
50 } // fim da classe GridLayoutFrame

```

Figura 12.43 | GridLayout que contém seis botões.

```

1  // Figura 12.44: 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);
12         gridLayoutFrame.setVisible(true);

```

continua

```

13     }
14 } // fim da classe GridLayoutDemo

```



Figura 12.44 | Testando GridLayoutFrame.

As linhas 24 e 25 (Figura 12.43) criam dois objetos `GridLayout`. O construtor `GridLayout` utilizado na linha 24 especifica um `GridLayout` com 2 linhas, 3 colunas, 5 pixels de espaçamento horizontal entre os `Components` na grade e 5 pixels de espaçamento vertical entre `Components` na grade. O construtor `GridLayout` utilizado na linha 25 especifica um `GridLayout` com 3 linhas e 2 colunas que utiliza o espaçamento padrão (1 pixel).

Os objetos `JButton` nesse exemplo são inicialmente organizados utilizando-se `gridLayout1` (configura para o painel de conteúdo na linha 27 com o método `setLayout`). O primeiro componente é adicionado à primeira coluna da primeira linha. O próximo componente é adicionado à segunda coluna da primeira linha e assim por diante. Quando um `JButton` é pressionado, o método `actionPerformed` (linhas 39 a 49) é chamado. Toda chamada para `actionPerformed` alterna o layout entre `gridLayout2` e `gridLayout1`, utilizando a variável `boolean toggle` para determinar o próximo layout a ser configurado.

A linha 48 mostra outra maneira de reformatar um contêiner cujo layout foi alterado. O método `Container validate` recalcula o layout do contêiner com base no gerenciador de layout atual para o `Container` e o conjunto atual de componentes GUI exibidos.

12.19 Utilizando painéis para gerenciar layouts mais complexos

GUIs complexas (como a Figura 12.1) frequentemente exigem que cada componente seja colocado em uma localização exata. Elas 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`, assim todo `JPanel` é um `Container`. Portanto, cada `JPanel` pode ter componentes, inclusive outros painéis, anexados a ele com o método `Container add`. O aplicativo das figuras 12.45 e 12.46 demonstra como um `JPanel` pode ser utilizado para criar um layout mais complexo em que vários `JButtons` são colocados na região `SOUTH` de um `BorderLayout`.

```

1  // Figura 12.45: 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 final JPanel buttonJPanel; // painel para armazenar botões
12     private final JButton[] buttons;
13
14     // construtor sem argumento
15     public PanelFrame()
16     {
17         super("Panel Demo");
18         buttons = new JButton[5];
19         buttonJPanel = new JPanel();
20         buttonJPanel.setLayout(new GridLayout(1, buttons.length));
21
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

```

continuação

```

27     }
28
29     add(buttonJPanel, BorderLayout.SOUTH); // adiciona painel ao JFrame
30 }
31 } // fim da classe PanelFrame

```

Figura 12.45 | JPanel com cinco JButtons em um GridLayout anexado à região SOUTH de um BorderLayout.

```

1 // Figura 12.46: 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);
12        panelFrame.setVisible(true);
13    }
14 } // fim da classe PanelDemo

```

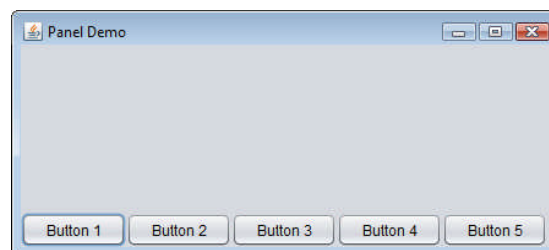


Figura 12.46 | Testando PanelFrame.

Depois de o JPanel `buttonJPanel` ser declarado (linha 11 da Figura 12.45) e criado (linha 19), a linha 20 configura o layout de `buttonJPanel` como um `GridLayout` de uma linha e cinco colunas (há cinco `JButtons` no array `buttons`). As linhas 23 a 27 adicionam o `JButtons` no array ao `JPanel`. A linha 26 adiciona os botões diretamente ao `JPanel` — a classe `JPanel` não tem um painel de conteúdo, ao contrário de um `JFrame`. A linha 29 usa `BorderLayout` padrão do `JFrame` para adicionar `buttonJPanel` à região `SOUTH`. A região `SOUTH` tem a mesma altura que os botões no `buttonJPanel`. Um `JPanel` é dimensionado aos componentes que ele contém. À medida que mais componentes são adicionados, o `JPanel` *cresce* (de acordo com as restrições de seu gerenciador de layout) para acomodar os componentes. Redimensione a janela para ver como o gerenciador de layout afeta o tamanho dos `JButtons`.

12.20 JTextArea

JTextArea fornece uma área para *manipular múltiplas linhas de texto*. Assim como a classe `JTextField`, `JTextArea` é uma subclasse de `JTextComponent` que declara os métodos comuns para `JTextFields`, `JTextAreas` e vários outros componentes GUI baseados em texto.

O aplicativo nas figuras 12.47 e 12.48 demonstra as `JTextAreas`. Uma `JTextArea` exibe texto que o usuário pode selecionar. A outra não é editável pelo usuário e é usada para exibir o texto que o usuário selecionou na primeira `JTextArea`. Ao contrário de `JTextFields`, `JTextAreas` não têm eventos de ação — ao pressionar *Enter* enquanto digita em uma `JTextArea`, o cursor simplesmente move-se para a próxima linha. Como com `JLists` de seleção múltipla (Seção 12.13), um evento externo de outro componente GUI indica quando processar o texto em uma `JTextArea`. Por exemplo, ao digitar uma mensagem de correio eletrônico, você normalmente clica em um botão **Send** para enviar o texto da mensagem para o destinatário. De maneira semelhante, ao editar um documento em um processador de texto, você normalmente salva o arquivo selecionando um item de menu **Save** ou **Save As...** Nesse programa, o botão **Copy >>>** gera o evento externo que copia o texto selecionado em `JTextArea` à esquerda e o exibe em `JTextArea` à direita.


```

1  // Figura 12.47: TextAreaFrame.java
2  // Copiando texto selecionado de uma área de JText 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 final JTextArea textArea1; // exibe a string demo
14     private final JTextArea textArea2; // texto destacado é copiado aqui
15     private final JButton copyButton; // começa a copiar o texto
16
17     // construtor sem argumento
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";
25
26         textArea1 = new JTextArea(demo, 10, 15);
27         box.add(new JScrollPane(textArea1)); // adiciona scrollpane
28
29         copyButton = new JButton("Copy >>>"); // cria botão de cópia
30         box.add(copyButton); // adiciona o botão de cópia à box
31         copyButton.addActionListener(
32             new ActionListener() // classe interna anônima
33             {
34                 // configura texto em textArea2 como texto selecionado de textArea1
35                 @Override
36                 public void actionPerformed(ActionEvent event)
37                 {
38                     textArea2.setText(textArea1.getSelectedText());
39                 }
40             }
41         );
42
43         textArea2 = new JTextArea(10, 15);
44         textArea2.setEditable(false);
45         box.add(new JScrollPane(textArea2)); // adiciona scrollpane
46
47         add(box); // adiciona box ao frame
48     }
49 } // fim da classe TextAreaFrame

```

Figura 12.47 | Copiando texto selecionado de uma JTextArea para outra.

```

1  // Figura 12.48: TextAreaDemo.java
2  // Testando TextAreaFrame.
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);
12         textAreaFrame.setVisible(true);
13     }
14 } // fim da classe TextAreaDemo

```

continua

continuação

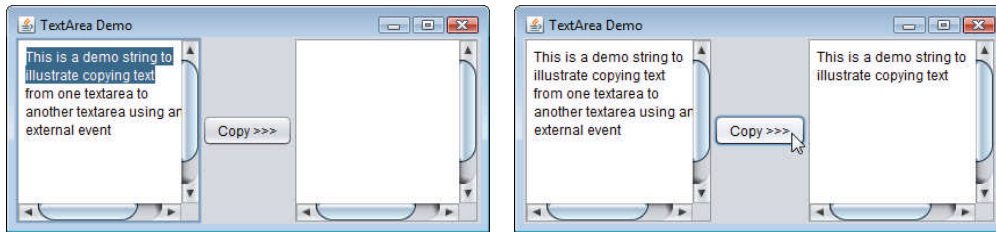


Figura 12.48 | Testando TextAreaFrame.

No construtor (linhas 18 a 48), a linha 21 cria um contêiner **Box** (pacote `javax.swing`) para organizar os componentes GUI. **Box** é uma subclasse de **Container** que usa um gerenciador de layout **BoxLayout** (discutido em detalhes na Seção 22.9) para organizar os componentes GUI horizontal ou verticalmente. O método `static createHorizontalBox` de **Box** cria uma **Box** que organiza componentes da esquerda para a direita na ordem que eles são anexados.

As linhas 26 e 43 criam **JTextAreas** `textArea1` e `textArea2`. A linha 26 utiliza o construtor de três argumentos de **JTextArea**, que aceita uma **String** que representa o texto inicial e dois **ints** para especificar que a **JTextArea** tem 10 linhas e 15 colunas. A linha 43 utiliza o construtor de dois argumentos da **JTextArea**, especificando que a **JTextArea** tem 10 linhas e 15 colunas. A linha 26 especifica que `demo` deve ser exibido como o conteúdo **JTextArea** padrão. Uma **JTextArea** não fornece barras de rolagem se não puder exibir seu conteúdo completo. Assim, a linha 27 cria um objeto **JScrollPane**, inicializa-o com `textArea1` e o atribui ao contêiner **box**. Por padrão, as barras de rolagem horizontais e verticais aparecem conforme necessário em um **JScrollPane**.

As linhas 29 a 41 criam objeto **JButton** `copyJButton` com o rótulo "Copy >>>", adicionam `copyJButton` ao contêiner **box** e registram a rotina de tratamento de evento ao **ActionEvent** de `copyJButton`. Esse botão fornece o evento externo que determina quando o programa deve copiar o texto selecionado na `textArea1` para `textArea2`. Quando o usuário clica em `copyJButton`, a linha 38 em `actionPerformed` indica que o método `getSelectedText` (herdado em **JTextArea** de **JTextComponent**) deve retornar o texto selecionado a partir de `textArea1`. O usuário seleciona texto arrastando o mouse sobre o texto desejado para destacá-lo. O método `setText` altera o texto em `textArea2` para a string retornada por `getSelectedText`.

As linhas 43 a 45 criam `textArea2`, configuram sua propriedade editável como `false` e adicionam essa propriedade ao contêiner **box**. A linha 47 adiciona o **box** a **JFrame**. Lembre-se, a partir do foi falado na Seção 12.18.2, de que o layout padrão de um **JFrame** é uma **BorderLayout** e que o método `add` por padrão anexa seu argumento ao **CENTER** da **BorderLayout**.

Quando o texto alcança a borda direita de uma **JTextArea**, ele pode recorrer para a próxima linha. Isso é referido como **mudança de linha automática**. Por padrão, **JTextArea** *não* muda de linha automaticamente.



Observação sobre a aparência e comportamento 12.19

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

Diretivas de barra de rolagem JScrollPane

Esse exemplo utiliza um **JScrollPane** para fornecer rolagem para uma **JTextArea**. Por padrão, **JScrollPane** só exibe barras de rolagem se elas forem necessárias. Você pode definir as **diretivas da barra de rolagem** horizontal e vertical de um **JScrollPane** quando ele é construído. Se um programa tem uma referência a um **JScrollPane**, ele pode usar os métodos **JScrollPane** `setHorizontalScrollBarPolicy` e `setVerticalScrollBarPolicy` para alterar as diretivas da barra de rolagem a qualquer momento. 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_AS_NEEDED
```

para indicar que *uma barra de rolagem deve aparecer somente se necessário* (os padrões) e as constantes

```
JScrollPane.VERTICAL_SCROLLBAR_NEVER
JScrollPane.HORIZONTAL_SCROLLBAR_NEVER
```

para indicar que *uma barra de rolagem nunca deve aparecer*. Se a diretiva de barra de rolagem horizontal for configurada como **JScrollPane.HORIZONTAL_SCROLLBAR_NEVER**, uma **JTextArea** anexada ao **JScrollPane** mudará automaticamente de linhas.

12.21 Conclusão

Neste capítulo, você aprendeu muitos componentes GUI e como tratar seus eventos. Você também aprendeu sobre as classes aninhadas, classes internas e classes internas anônimas. Você viu o relacionamento especial entre um objeto de classe interna e um objeto de sua classe de primeiro nível. Aprendeu a utilizar diálogos de `JOptionPane` para obter entrada de texto do usuário e a exibir mensagens para ele. Você também aprendeu a criar aplicativos que são executados em suas próprias janelas. Discutimos a classe `JFrame` e componentes que permitem ao usuário interagir com um aplicativo. Também mostramos como exibir texto e imagens para o usuário. Você aprendeu a personalizar `JPanels` para criar áreas de desenho personalizadas, que serão extensamente utilizadas no próximo capítulo. Você viu como organizar componentes em uma janela utilizando gerenciadores de layout e como criar GUIs mais complexas utilizando `JPanels` para organizar componentes. Por fim, aprendeu sobre o componente `JTextArea` em que um usuário pode inserir texto e um aplicativo pode exibir texto. No Capítulo 22, você aprenderá sobre os componentes GUI mais avançados, como controles deslizantes, menus e gerenciadores de layout mais complexos. No próximo capítulo, você aprenderá a adicionar imagens gráficas ao aplicativo GUI. Os recursos gráficos permitem desenhar formas e texto com cores e estilos.

Resumo

Seção 12.1 Introdução

- Uma interface gráfica com usuário (*graphical user interface* — GUI) apresenta um mecanismo amigável ao usuário para interagir com um aplicativo. Uma GUI dá a um aplicativo uma aparência e um comportamento distintos.
- Fornecer a diferentes aplicativos componentes de interface intuitivos e consistentes dá aos usuários uma sensação de familiaridade com um novo aplicativo, assim eles podem entendê-lo mais rapidamente.
- As GUIs são construídas a partir de componentes GUI — às vezes chamados controles ou widgets.

Seção 12.2 A nova aparência e comportamento do Java Nimbus

- A partir da atualização 10 do Java SE 6, o Java é distribuído com uma interface nova, elegante e compatível com várias plataformas conhecida como Nimbus.
- Para definir Nimbus como o padrão para todos os aplicativos Java, crie um arquivo de texto `swing.properties` na pasta `lib` das pastas de instalação do JDK e JRE. Insira a seguinte linha do código no arquivo:

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

- Para selecionar Nimbus em uma base aplicativo por aplicativo, coloque o seguinte argumento de linha de comando após o comando Java e antes do nome do aplicativo quando você o executa:

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

Seção 12.3 Entrada/saída baseada em GUI simples com `JOptionPane`

- A maioria dos aplicativos usa janelas ou caixas de diálogo para interagir com o usuário.
- A classe `JOptionPane` do pacote `javax.swing` fornece caixas de diálogo pré-construídas tanto para entrada como saída. O método `JOptionPane static showInputDialog` exibe um diálogo de entrada.
- Em geral, um prompt utiliza maiúsculas e minúsculas no estilo de frases — empregando a maiúscula inicial apenas na primeira palavra da frase a menos que a palavra seja um nome próprio.
- Um diálogo de entrada só pode inserir `Strings` de entrada. Isso é típico da maioria dos componentes GUI.
- O método `JOptionPane static showMessageDialog` exibe um diálogo de mensagem.

Seção 12.4 Visão geral de componentes Swing

- A maioria dos componentes GUI Swing está localizada no pacote `javax.swing`.
- 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 uniformemente a aparência e comportamento para o aplicativo em todas as plataformas ou utilizar a aparência e comportamento personalizados de cada plataforma.
- Os componentes Swing leves não são amarrados aos componentes GUI reais suportados pela plataforma subjacente em que um aplicativo é executado.
- Vários componentes Swing são componentes pesados que exigem interação direta com o sistema de janela local, que pode restringir sua aparência e funcionalidades.
- A classe `Component` do pacote `java.awt` declara muitos dos atributos e comportamentos comuns para os componentes GUI nos pacotes `java.awt` e `javax.swing`.

- A classe `Container` do pacote `java.awt` é uma subclasse de `Component`. `Components` são anexados a `Containers`; desse modo, os `Components` podem ser organizados e exibidos na tela.
- A classe `JComponent` do pacote `javax.swing` é uma subclasse de `Container`. `JComponent` é a superclasse de todos os componentes `Swing` leves e declara seus atributos e comportamentos comuns.
- Alguns recursos `JComponent` comuns incluem uma aparência e um comportamento plugáveis, teclas de atalho chamadas `mnemônicos`, dicas de ferramentas, suporte para tecnologias auxiliares e suporte para “localização” (tradução) da interface com o usuário.

Seção 12.5 Exibição de texto e imagens em uma janela

- A classe `JFrame` fornece os atributos e comportamentos básicos de uma janela.
- Um `JLabel` exibe somente texto de leitura, uma imagem, ou texto e uma imagem. Normalmente, o texto em um `JLabel` emprega maiúsculas e minúsculas no estilo de frases.
- Cada componente GUI deve ser anexado a um contêiner, como uma janela criada com um `JFrame`.
- Muitos IDEs fornecem ferramentas de design de GUI em que você pode especificar o tamanho e localização exata de um componente utilizando o mouse; então, o IDE gerará o código GUI para você.
- O método `setToolTipText` `JComponent` especifica a dica de ferramenta que é exibida quando o usuário posiciona o cursor do mouse sobre um componente leve.
- O método `Container.add` anexa um componente GUI a um `Container`.
- A classe `ImageIcon` suporta vários formatos de imagem, incluindo GIF, PNG e JPEG.
- O método `getClass` da classe `Object` recupera uma referência ao objeto `Class`, que representa a declaração de classe do objeto em que o método é chamado.
- O método `Class.getResource` retorna a localização de seu argumento como um URL. O método `getResource` utiliza o carregador de classe do objeto `Class` para determinar a localização do recurso.
- Os alinhamentos horizontais e verticais de um `JLabel` podem ser definidos com os métodos `setHorizontalAlignment` e `setVerticalAlignment`, respectivamente.
- Os métodos `setText` e `getText` de `JLabel` definem e exibem o texto em um rótulo.
- Os métodos `setIcon` e `getIcon` `JLabel` definem e obtêm o `Icon` em um rótulo.
- Os métodos `setHorizontalTextPosition` e `setVerticalTextPosition` de `JLabel` especificam a posição do texto no rótulo.
- O método `JFrame.setDefaultCloseOperation` com a constante `JFrame.EXIT_ON_CLOSE` como o argumento indica o que o programa deve terminar quando a janela é fechada pelo usuário.
- O método `Component.setSize` especifica a largura e altura de um componente.
- O método `Component.setVisible` com o argumento `true` exibe um `JFrame` na tela.

Seção 12.6 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, os eventos guiam o programa para realizar tarefas.
- Uma rotina de tratamento de evento realiza uma tarefa em resposta a um evento.
- A classe `JTextField` estende `JTextComponent` do pacote `javax.swing.text`, que fornece recursos comuns de componentes baseados em texto. A classe `JPasswordField` estende `JTextField` e adiciona vários métodos que são específicos ao processamento de senhas.
- Um `JPasswordField` mostra que os caracteres estão sendo digitados à medida que o usuário os insere, mas oculta os caracteres reais com caracteres `eco`.
- Um componente recebe o foco quando o usuário clica no componente.
- O método `JTextComponent.setEditable` pode ser utilizado para tornar um campo de texto não editável.
- Para responder a um evento para um componente GUI específico, você deve criar uma classe que representa a rotina de tratamento de evento e implementar uma interface ouvinte de evento apropriada, e então registrar um objeto da classe de tratamento de evento como a rotina de tratamento de evento.
- As classes não `static` aninhadas são chamadas de classes internas e são frequentemente utilizadas para tratamento de evento.
- Um objeto de uma classe interna não `static` deve ser criado por um objeto da classe de nível superior que contém a classe interna.
- Um objeto de classe interna pode acessar diretamente as variáveis de instância e métodos de sua classe de primeiro nível.
- 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.
- Pressionar *Enter* em um `JTextField` ou `JPasswordField` gera um `ActionEvent` que pode ser manipulado por um `ActionListener` do pacote `java.awt.event`.
- O método `addActionListener` `JTextField` registra uma rotina de tratamento de evento para um `ActionEvent` de um campo de texto.
- O componente GUI com o qual o usuário interage é a origem de evento.
- Um objeto `ActionEvent` contém informações sobre o evento que acabou de ocorrer, como a origem de evento e o texto no campo de texto.

- O método `ActionEvent.getSource` retorna uma referência à origem de evento. O método `ActionEvent.getActionCommand` retorna o texto que o usuário digitou em um campo de texto ou o rótulo em um `JButton`.
- O método `JPasswordField.getPassword` retorna a senha que o usuário digitou.

Seção 12.7 Tipos comuns de eventos GUI e interfaces ouvintes

- Cada tipo de objeto de evento normalmente tem uma interface ouvinte de evento correspondente que especifica um ou mais métodos da rotina de tratamento de evento, que devem ser declarados na classe que implementa a interface.

Seção 12.8 Como o tratamento de evento funciona

- 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.
- Cada componente GUI suporta vários tipos de evento. Quando um evento ocorre, ele é despachado apenas para os ouvintes de evento do tipo apropriado.

Seção 12.9 JButton

- Um botão é um componente em que o usuário clica para acionar uma ação. Todos os tipos de botão são subclasses de `AbstractButton` (pacote `javax.swing`). Rótulos de botão normalmente usam letras maiúsculas.
- Botões de comando são criados com a classe `JButton`.
- Um `JButton` pode exibir um `Icon`. Um `JButton` também pode ter um `Icon` de rollover — um `Icon` que é exibido quando o usuário posiciona o mouse sobre o botão.
- O método `setRolloverIcon` da classe `AbstractButton` especifica a imagem exibida em um botão quando o usuário posiciona o mouse sobre ele.

Seção 12.10 Botões que mantêm o estado

- Existem três tipos Swing de estado de botão — `JToggleButton`, `JCheckBox` e `JRadioButton`.
- As classes `JCheckBox` e `JRadioButton` são subclasses de `JToggleButton`.
- O método `setFont` de `Component` define a fonte do componente como um novo objeto `Font` do pacote `java.awt`.
- Clicar em um `JCheckBox` causa um `ItemEvent` que pode ser tratado por um `ItemListener` que define o método `itemStateChanged`. O método `addItemListener` registra o ouvinte para o `ItemEvent` de um objeto `JCheckBox` ou `JRadioButton`.
- O método `JCheckBox.isSelected` determina se uma `JCheckBox` está selecionada.
- `JRadioButtons` têm dois estados — selecionado e não selecionado. Os botões de rádio normalmente aparecem como um grupo em que um único botão pode ser selecionado de cada vez.
- `JRadioButtons` são usados para representar opções mutuamente exclusivas.
- A relação lógica entre `JRadioButtons` é mantida por um objeto `ButtonGroup`.
- O método `addButtonGroup` associa cada `JRadioButton` a um `ButtonGroup`. Se mais de um objeto `JRadioButton` selecionado for adicionado a um grupo, aquele selecionado que foi adicionado primeiro será selecionado quando a GUI for exibida.
- `JRadioButtons` geram `ItemEvents` quando são clicados.

Seção 12.11 JComboBox e uso de uma classe interna anônima para tratamento de eventos

- Uma `JComboBox` fornece uma lista de itens em que o usuário pode fazer uma única seleção. `JComboBoxes` geram `ItemEvents`.
- Cada item em um `JComboBox` tem um índice. O primeiro item adicionado a uma `JComboBox` aparece como o item atualmente selecionado quando a `JComboBox` é exibida.
- O método `JComboBox.setMaximumRowCount` configura o número máximo de elementos que é exibido quando o usuário clica na `JComboBox`.
- Uma classe interna anônima é uma classe sem um nome e normalmente aparece dentro de uma declaração de método. Um objeto da classe interna anônima deve ser criado quando a classe é declarada.
- O método `JComboBox.getSelectedIndex` retorna o índice do item selecionado.

Seção 12.12 JList

- Uma `JList` exibe uma série de itens da qual o usuário pode selecionar um ou mais itens. A classe `JList` suporta listas de uma única seleção e listas de seleção múltipla.
- Quando o usuário clica em um item em uma `JList`, ocorre um `ListSelectionEvent`. O método `addListSelectionListener` de `JList` registra um `ListSelectionListener` para os eventos de seleção de uma `JList`. Um `ListSelectionListener` do pacote `javax.swing.event` deve implementar o método `valueChanged`.
- O método `setVisibleRowCount` de `JList` especifica o número de itens visíveis na lista.

- O método `setSelectionMode` `JList` especifica o modo de seleção de uma lista.
- Uma `JList` pode ser anexada a um `JScrollPane` para fornecer uma barra de rolagem para a `JList`.
- O método `JFrame` `getContentPane` retorna uma referência ao painel de conteúdo de `JFrame` em que os componentes GUI são exibidos.
- O método `getSelectedIndex` `JList` retorna o índice do item selecionado.

Seção 12.13 Listas de seleção múltipla

- Uma lista de seleção múltipla permite ao usuário selecionar muitos itens de uma `JList`.
- O método `JList` `setFixedCellWidth` configura a largura de uma `JList`. O método `setFixedCellHeight` configura a altura de cada item em uma `JList`.
- Normalmente, um evento externo gerado por outro componente GUI (como um `JButton`) especifica quando as múltiplas seleções em uma `JList` devem ser processadas.
- O método `JList` `setListData` configura os itens exibidos em uma `JList`. O método `JList` `getSelectedValues` retorna um array de `Objects` para representar os itens selecionados em uma `JList`.

Seção 12.14 Tratamento de evento de mouse

- As interfaces ouvintes de eventos `MouseListener` e `MouseMotionListener` são usadas para tratar eventos de mouse. Os eventos de mouse podem ser interrompidos por qualquer componente GUI que estenda `Component`.
- A interface `MouseInputListener` do pacote `javax.swing.event` estende interfaces `MouseListener` e `MouseMotionListener` para criar uma interface simples que contém todos os seus métodos.
- Cada método de tratamento de evento de mouse recebe um objeto `MouseEvent` que contém informações sobre o evento, incluindo as coordenadas x e y em que o evento ocorreu. As coordenadas são medidas a partir do canto superior esquerdo do componente GUI em que o evento ocorreu.
- Os métodos e as constantes de classe `InputEvent` (superclasse de `MouseEvent`) permitem que um aplicativo determine o botão do mouse em que o usuário clicou.
- A interface `MouseWheelListener` permite que os aplicativos respondam a eventos de roda do mouse.

Seção 12.15 Classes de adaptadores

- Uma classe de adaptadores implementa uma interface e fornece implementações padrão dos seus métodos. Ao estender uma classe de adaptadores, é possível sobrescrever apenas o(s) método(s) de que você precisa.
- O método `MouseEvent` `getClickCount` retorna o número de cliques consecutivos de botão do mouse. Os métodos `isMetaDown` e `isAltDown` determinam qual botão foi clicado.

Seção 12.16 Subclasse `JPanel` para desenhar com o mouse

- O método `paintComponent` `JComponent` é chamado quando um componente Swing leve é exibido. Sobrescreva esse método para especificar como desenhar formas usando as capacidades gráficas do Java.
- Ao sobrescrever `paintComponent`, chame a versão da superclasse como a primeira instrução no corpo.
- As subclasses de `JComponent` suportam transparência. Quando um componente é opaco, `paintComponent` limpa o fundo antes de o componente ser exibido.
- A transparência de um componente Swing leve pode ser configurada com o método `setOpaque` (um argumento `false` indica que o componente é transparente).
- A classe `Point` do pacote `java.awt` representa uma coordenada x - y .
- A classe `Graphics` é usada para desenhar.
- O método `MouseEvent` `getPoint` obtém o `Point` em que ocorreu um evento de mouse.
- O método `repaint`, herdado indiretamente de classe `Component`, indica que um componente deve ser atualizado na tela o mais rápido possível.
- O método `paintComponent` recebe um parâmetro `Graphics` e é chamado automaticamente sempre que um componente leve precisar ser exibido na tela.
- O método `fillOval` `Graphics` desenha uma oval sólida. Os dois primeiros argumentos são a coordenada x e y do canto superior esquerdo da caixa delimitadora, e os dois últimos são a largura e a altura da caixa delimitadora.

Seção 12.17 Tratamento de eventos de teclado

- A interface `KeyListener` é usada para tratar eventos de teclado que são gerados quando as teclas do teclado são pressionadas e liberadas. O método `addKeyListener` da classe `Component` registra um `KeyListener`.
- O método `getKeyCode` `KeyEvent` obtém o código das teclas virtuais da tecla pressionada. A classe `KeyEvent` contém constantes de código de tecla virtual que representam cada tecla no teclado.

- O método `getKeyText` `KeyEvent` retorna uma string contendo o nome da tecla pressionada.
- O método `KeyEvent` `getKeyChar` obtém o valor Unicode do caractere digitado.
- O método `isActionKey` `KeyEvent` determina se a tecla em um evento era uma tecla de ação.
- O método `InputEvent` `getModifiers` determina se alguma tecla modificadora (como *Shift*, *Alt* e *Ctrl*) foi pressionada quando o evento de teclado ocorreu.
- O método `getKeyModifiersText` `KeyEvent` retorna uma string contendo as teclas modificadoras pressionadas.

Seção 12.18 Introdução a gerenciadores de layout

- Os gerenciadores de layout organizam componentes GUI em um contêiner para propósitos de apresentação.
- Todos os gerenciadores de layout implementam a interface `LayoutManager` do pacote `java.awt`.
- O método `Container` `setLayout` especifica o layout de um contêiner.
- `FlowLayout` posiciona os componentes da esquerda para a direita na ordem em que eles são adicionados ao contêiner. Quando a borda do contêiner é alcançada, os componentes continuam sendo exibidos na próxima linha. `FlowLayout` permite que componentes GUI sejam alinhados à esquerda, centralizados (o padrão) e alinhados à direita.
- O método `setAlignment` `FlowLayout` muda o alinhamento para um `FlowLayout`.
- `BorderLayout`, o padrão para um `JFrame`, organiza os componentes em cinco regiões: NORTH, SOUTH, EAST, WEST e CENTER. NORTH corresponde à parte superior do contêiner.
- Um `BorderLayout` limita um `Container` a conter no máximo cinco componentes — um em cada região.
- `GridLayout` divide um contêiner em uma grade de linhas e colunas.
- O método `Container` `validate` recalcula o layout de um contêiner com base no gerenciador de layout atual para o `Container` e para o conjunto atual de componentes GUI exibidos.

Seção 12.19 Utilizando painéis para gerenciar layouts mais complexos

- GUIs complexas geralmente consistem em múltiplos painéis com diferentes layouts. Cada `JPanel` pode ter componentes, inclusive outros painéis, anexados a ele com o método `Container` `add`.

Seção 12.20 JTextArea

- Uma `JTextArea` — uma subclasse de `JTextComponent` — pode conter múltiplas linhas de texto.
- A classe `Box` é uma subclasse de `Container` que utiliza um gerenciador de layout `BoxLayout` para organizar os componentes GUI horizontal ou verticalmente.
- O método `Box` `static` `createHorizontalBox` cria um `Box` que organiza componentes da esquerda para a direita na ordem em que eles são anexados.
- O método `getSelectedText` retorna o texto selecionado a partir de uma `JTextArea`.
- Você pode definir as diretivas da barra de rolagem horizontal e vertical de um `JScrollPane` quando ele é construído. Os métodos `setHorizontalScrollBarPolicy` e `setVerticalScrollBarPolicy` `JScrollPane` podem ser usados para alterar as diretrizes da barra de rolagem a qualquer momento.

Exercícios de revisão

12.1 Preencha as lacunas em cada uma das seguintes afirmações:

- a) O método _____ é chamado quando o mouse é movido sem pressionamento de botões e um ouvinte de eventos é registrado para tratar o evento.
- b) O texto que não pode ser modificado pelo usuário é chamado de texto _____.
- c) Um(a) _____ organiza os componentes GUI em um `Container`.
- d) O método `add` para anexar componentes GUI é um método da classe _____.
- e) GUI é um acrônimo de _____.
- f) O método _____ é utilizado para especificar o gerenciador de layout para um contêiner.
- g) Uma chamada de método `mouseDragged` é precedida por uma chamada de método _____ e seguida por uma chamada de método _____.
- h) A classe _____ contém métodos que exibem diálogos de mensagem e diálogos de entrada.
- i) Um diálogo de entrada capaz de receber entrada do usuário é exibido com o método _____ da classe _____.
- j) Um diálogo capaz de exibir uma mensagem para o usuário é exibido com o método _____ da classe _____.
- k) Tanto `JTextField`s como `JTextAreas` estendem diretamente a classe _____.

- 12.2** Determine se cada sentença é *verdadeira* ou *falsa*. Se *falsa*, explique por quê.
- BorderLayout é o gerenciador de layout padrão do painel de conteúdo de um JFrame.
 - Quando o cursor do mouse é movido nos limites de um componente GUI, o método `mouseOver` é chamado.
 - Um JPanel não pode ser adicionado a outro JPanel.
 - Em um BorderLayout, dois botões adicionados à região NORTH serão colocados lado a lado.
 - No máximo cinco componentes podem ser adicionados a um BorderLayout.
 - As classes internas não têm permissão de acessar os membros da classe que os envolve.
 - Um texto da JTextArea é sempre de leitura (*read-only*).
 - A classe JTextArea é uma subclasse direta da classe Component.
- 12.3** Localize o(s) erro(s) em cada uma das seguintes instruções e explique como corrigi-lo(s).
- `buttonName = JButton("Caption");`
 - `JLabel aLabel, JLabel;`
 - `txtField = new JTextField(50, "Default Text");`
 - `setLayout(new BorderLayout());`
`button1 = new JButton("North Star");`
`button2 = new JButton("South Pole");`
`add(button1);`
`add(button2);`

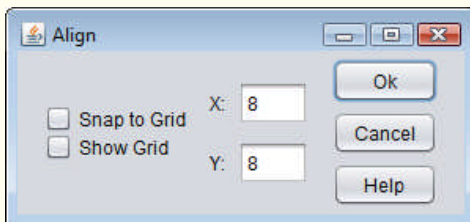
Respostas dos exercícios de revisão

- 12.1** a) `mouseMoved`. b) não editável (de leitura). c) gerenciador de layout. d) `Container`. e) interface gráfica com o usuário. f) `setLayout`. g) `mousePressed`, `mouseReleased`. h) `JOptionPane`. i) `showInputDialog`, `JOptionPane`. j) `showMessageDialog`, `JOptionPane`. k) `JTextComponent`.
- 12.2**
- Verdadeiro.
 - Falso. O método `mouseEntered` é chamado.
 - Falso. Um JPanel pode ser adicionado a outro JPanel, porque JPanel é uma subclasse indireta de Component. Assim, um JPanel é um Component. Qualquer Component pode ser adicionado a um Container.
 - Falso. Apenas o último botão adicionado será exibido. Lembre-se de que apenas um componente deve ser adicionado a cada região em um BorderLayout.
 - Verdadeiro. [Observação: painéis contendo múltiplos componentes podem ser adicionados a cada região.]
 - Falso. As classes internas têm acesso a todos os membros da declaração de classe que os envolve.
 - Falso. JTextAreas são editáveis por padrão.
 - Falso. JTextArea deriva da classe JTextComponent.
- 12.3**
- `new` é necessário para criar um objeto.
 - `JLabel` é um nome de classe e não pode ser utilizado como um nome de variável.
 - Os argumentos passados para o construtor estão invertidos. A `String` deve ser passada primeiro.
 - `BorderLayout` foi configurado e os componentes que estão sendo adicionados sem especificar a região são ambos adicionados à região centro. As instruções `add` adequadas podem ser
`add(button1, BorderLayout.NORTH);`
`add(button2, BorderLayout.SOUTH);`

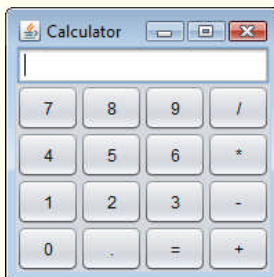
Questões

- 12.4** Preencha as lacunas em cada uma das seguintes afirmações:
- A classe `JTextField` estende diretamente a classe _____.
 - O método `Container` _____ anexa um componente GUI a um contêiner.
 - O método _____ é chamado quando um botão de mouse é liberado (sem mover o mouse).
 - A classe _____ é utilizada para criar um grupo de `JRadioButtons`.
- 12.5** Determine se cada sentença é *verdadeira* ou *falsa*. Se *falsa*, explique por quê.
- Apenas um gerenciador de layout pode ser utilizado por `Container`.
 - Os componentes GUI podem ser adicionados a um `Container` em qualquer ordem em um `BorderLayout`.
 - `JRadioButtons` fornecem uma série de opções mutuamente exclusivas (isto é, apenas um pode ser `true` por vez).
 - O método `Graphics.setFont` é utilizado para configurar a fonte para campos de texto.
 - Uma `JList` exibe uma barra de rolagem se houver mais itens na lista do que podem ser exibidos.
 - Um objeto `Mouse` tem um método chamado `mouseDragged`.

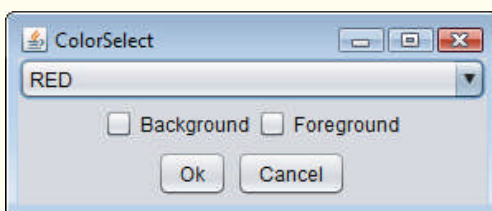
- 12.6** Determine se cada sentença é *verdadeira* ou *falsa*. Se *falsa*, explique por quê.
- Um JPanel é um JComponent.
 - Um JPanel é um Component.
 - Um JLabel é um Container.
 - Um JList é um JPanel.
 - Um AbstractButton é um JButton.
 - Um JTextField é um Object.
 - ButtonGroup é uma subclasse de JComponent.
- 12.7** Localize qualquer erro em cada uma das seguintes linhas de código e explique como corrigi-los.
- `import javax.swing.JFrame`
 - `panelObject.GridLayout(8, 8);`
 - `container.setLayout(new FlowLayout(FlowLayout.DEFAULT));`
 - `container.add(eastButton, EAST); face`
- 12.8** Crie a seguinte GUI. Você não precisa fornecer funcionalidades.



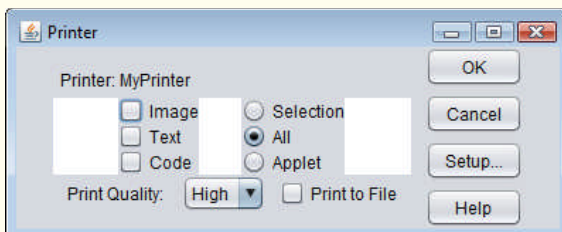
- 12.9** Crie a seguinte GUI. Você não precisa fornecer funcionalidades.



- 12.10** Crie a seguinte GUI. Você não precisa fornecer funcionalidades.



- 12.11** Crie a seguinte GUI. Você não precisa fornecer funcionalidades.



- 12.12** (*Conversão de temperatura*) Escreva um aplicativo de conversão de temperatura que converte de Fahrenheit em Celsius. A temperatura em Fahrenheit deve ser inserida pelo teclado (por um JTextField). Um JLabel deve ser utilizado para exibir a temperatura convertida. Utilize a seguinte fórmula para a conversão:

$$\text{Celsius} = \frac{5}{9} \times (\text{Fahrenheit} - 32)$$

- 12.13 (Modificação de conversão de temperatura)** Aprimore o aplicativo de conversão de temperatura da Questão 12.12 adicionando a escala de temperatura Kelvin. O aplicativo também deve permitir ao usuário fazer conversões entre quaisquer duas escalas. Utilize a seguinte fórmula para a conversão entre Kelvin e Celsius (além da fórmula na Questão 12.12) :

$$\text{Kelvin} = \text{Celsius} + 273,15$$

- 12.14 (Adivinhe o número)** Escreva um aplicativo que execute “adivinhe o número” como mostrado a seguir: Seu aplicativo escolhe o número a ser adivinhado selecionando um inteiro aleatoriamente no intervalo 1–1000. O aplicativo então exibe o seguinte em um rótulo:

I have a number between 1 and 1000. Can you guess my number?
Please enter your first guess.

Um `JTextField` deve ser utilizado para entrar a suposição. Conforme cada suposição é inserida, a cor de fundo deve mudar para vermelho ou azul. Vermelho indica que o usuário está ficando mais “quente”, e azul, “mais frio”. Um `JLabel` deve exibir “Too High” ou “Too Low” para ajudar a encontrar a resposta correta. Quando o usuário obtém a resposta correta, “Correct!” deve ser exibido, e o `JTextField` usado para entrada deve ser alterado para não ser editável. Um `JButton` deve ser fornecido para permitir ao usuário jogar de novo. Quando o `JButton` for clicado, um novo número aleatório deverá ser gerado e a entrada `JTextField` deve ser alterada para o estado editável.

- 12.15 (Exibindo eventos)** Frequentemente, é útil exibir os eventos que ocorrem durante a execução de um aplicativo. Isso pode ajudá-lo a entender quando os eventos ocorrem e como eles são gerados. Escreva um aplicativo que permita ao usuário gerar e processar cada evento discutido neste capítulo. O aplicativo deve fornecer os métodos das interfaces `ActionListener`, `ItemListener`, `ListSelectionListener`, `MouseListener`, `MouseMotionListener` e `KeyListener` para exibir as mensagens quando os eventos ocorrem. Utilize o método `toString` para converter os objetos de evento recebidos em cada rotina de tratamento de evento para `Strings` que possam ser exibidas. O método `toString` cria uma `String` contendo todas as informações no objeto de evento.

- 12.16 (Jogo de dados baseado em GUI)** Modifique o aplicativo da Seção 6.10 para fornecer uma GUI que permite ao usuário clicar em um `JButton` para lançar os dados. O aplicativo também deve exibir quatro `JLabels` e quatro `JTextFields`, com um `JLabel` para cada `JTextField`. Os `JTextFields` devem ser utilizados para exibir os valores de cada dado e a soma dos dados depois de cada lançamento. O ponto deve ser exibido no quarto `JTextField` quando o usuário não ganhar ou perder no primeiro lançamento e deve continuar a ser exibido até que o jogo seja perdido.

(Opcional) Exercício de estudo de caso de GUI e imagens gráficas: expandindo a interface

- 12.17 (Aplicativo de desenho interativo)** Neste exercício, você implementará um aplicativo GUI que usa a hierarquia `MyShape` do Exercício de estudo de caso de GUIs e imagens gráficas 10.2 para criar um aplicativo de desenho interativo. Você criará duas classes para a GUI e fornecerá uma classe de teste que carrega o aplicativo. As classes da hierarquia `MyShape` não exigem nenhuma alteração adicional.

A primeira classe a ser criada é uma subclasse de `JPanel` chamada `DrawPanel`, que representa a área em que o usuário desenha as formas. A classe `DrawPanel` deve ter as seguintes variáveis de instância:

- Um array `shapes` do tipo `MyShape` que armazenará todas as formas que o usuário desenha.
- Um inteiro `shapeCount` que conta o número de formas no array.
- Um inteiro `shapeType` que determina o tipo de forma a ser desenhada.
- Um `MyShape` `currentShape` que representa a forma atual que o usuário está desenhando.
- Um `Color` `currentColor` que representa a cor atual de desenho.
- Um booleano `filledShape` que determina se deve-se desenha ou não uma forma preenchida.
- Um `JLabel` `statusLabel` que representa a barra de status. A barra de status exibirá as coordenadas da posição atual do mouse.

A classe `DrawPanel` também deve declarar os seguintes métodos:

- Método `paintComponent` sobrescrito que desenha as formas no array. Utilize a variável de instância `shapeCount` para determinar quantas formas desenha. O método `paintComponent` também deve chamar método `draw` de `currentShape`, desde que `currentShape` não seja `null`.
- Configure os métodos para o `shapeType`, `currentColor` e `filledShape`.
- O método `clearLastShape` deve eliminar a última forma desenhada decrementando a variável de instância `shapeCount`. Assegure que `shapeCount` nunca é menor que zero.
- O método `clearDrawing` deve remover todas as formas no desenho atual configurando `shapeCount` como zero.

Os métodos `clearLastShape` e `clearDrawing` devem chamar o método `repaint` (herdado de `JPanel`) para atualizar o desenho no `DrawPanel` indicando que o sistema deve chamar o método `paintComponent`.

A classe `DrawPanel` também deve fornecer tratamento de evento para permitir ao usuário desenha com o mouse. Crie uma única classe interna que tanto estende `MouseAdapter` como implementa `MouseMotionListener` para tratar todos os eventos de mouse em uma classe.

Na classe interna, sobrescreva o método `mousePressed` para que ele atribua a `shapeType` uma nova forma do tipo especificado por `currentShape` e inicialize ambos os pontos como a posição do mouse. Em seguida, sobrescreva o método `mouseReleased` para terminar de desenha a forma atual e colocá-la no array. Configure o segundo ponto de `currentShape` como a posição atual do mouse e adicione `currentShape` ao array. A variável de instância `shapeCount` determina o índice de inserção. Configure `currentShape` como `null` e chame o método `repaint` para atualizar o desenho com a nova forma.

Sobrescreva o método `mouseMoved` para configurar o texto do `statusLabel` de modo que ele exiba as coordenadas de mouse — isso atualizará o rótulo com as coordenadas toda vez que o usuário mover (mas não arrastar) o mouse dentro do `DrawPanel`. Em seguida, sobrescreva o método `mouseDragged` de modo que ele configure o segundo ponto do `currentShape` como a posição de mouse atual e chame o método `repaint`. Isso permitirá ao usuário ver a forma ao arrastar o mouse. Além disso, atualize o `JLabel` em `mouseDragged` com a posição atual do mouse.

Crie um construtor para `DrawPanel` que tem um único parâmetro `JLabel`. No construtor, inicialize `statusLabel` com o valor passado para o parâmetro. Também inicialize o array `shapes` com 100 entradas, `shapeCount` como 0, `shapeType` como o valor que representa uma linha, `currentShape` como `null` e `currentColor` como `Color.BLACK`. O construtor então deve configurar a cor de fundo do `DrawPanel` como `Color.WHITE` e registrar o `MouseListener` e `MouseMotionListener` para que o `JPanel` trate adequadamente os eventos de mouse.

Em seguida, crie uma subclasse `JFrame` chamada `DrawFrame` que forneça uma GUI para permitir ao usuário controlar vários aspectos de desenho. Para o layout do `DrawFrame`, recomendamos um `BorderLayout`, com os componentes na região `NORTH`, o principal painel de desenho na região `CENTER` e uma barra de status na região `SOUTH`, como na Figura 12.49. No painel superior, crie os componentes listados a seguir. A rotina de tratamento de evento de cada componente deve chamar o método adequado na classe `DrawPanel`.

- Um botão para desfazer a última forma desenhada.
- Um botão para eliminar todas as formas do desenho.
- Uma caixa de combinação para selecionar a partir das 13 cores predefinidas.
- Uma caixa de combinação para selecionar a forma a desenhar.
- Uma caixa de seleção que especifica se uma forma deve ou não ter preenchimento.

Declare e crie os componentes de interface no construtor de `DrawFrame`. Você precisará criar a barra de status `JLabel` antes de criar o `DrawPanel`, para que possa passar o `JLabel` como um argumento para o construtor do `DrawPanel`. Por fim, crie uma classe de teste que inicialize e exiba o `DrawFrame` para executar o aplicativo.

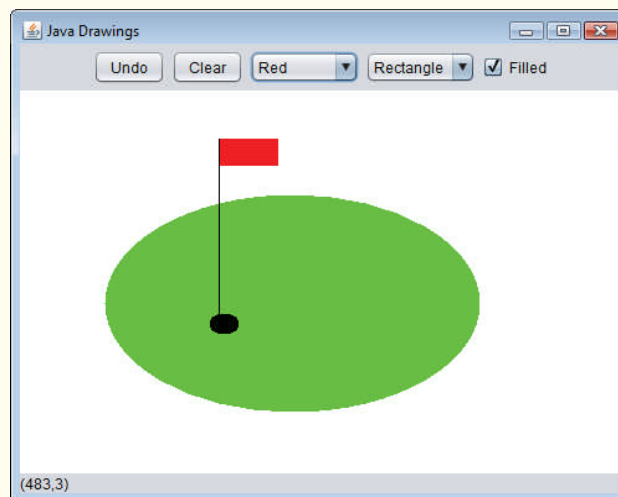


Figura 12.49 | Interface para desenhar formas.

12.18 (Versão baseada em GUI do estudo de caso ATM) Reimplemente o estudo de caso opcional ATM dos capítulos 33 e 34 (em inglês, na Sala Virtual do livro) como um aplicativo baseado em GUI. Use componentes GUI para criar a interface ATM com o usuário mostrada na Figura 33.1. Para o terminal de saque e depósito use `JButtons` rotulados **RemoveCash** e **Insert Envelope**. Isso irá permitir que o aplicativo receba eventos que indicam quando o usuário pega o dinheiro e insere um envelope de depósito, respectivamente.

Fazendo a diferença

12.19 (Ecofont) Ecofont (www.ecofont.eu/ecofont_en.html) — desenvolvida pela Spranq (uma empresa com sede na Holanda) — é uma fonte gratuita de computador de código aberto projetada para reduzir em até 20% a quantidade de tinta usada para impressão, reduzindo assim também o número de cartuchos de tinta usados e o impacto ambiental dos processos de produção e remessa (usando menos energia, menos combustível para o transporte etc.). A fonte, baseada em Verdana sem serifa, tem pequenos “furos” circulares nas letras que não são visíveis em tamanhos menores — como a fonte de 9 ou 10 pontos frequentemente utilizada. Baixe a Ecofont, então instale o arquivo de fonte `Spranq_eco_sans_regular.ttf` usando as instruções do site da Ecofont. Em seguida, desenvolva um programa baseado em GUI que permite inserir uma string de texto a ser exibida na Ecofont. Crie botões **Increase Font Size** e **Decrease Font Size** que permitem aumentar ou reduzir a fonte em um ponto de cada vez. Comece com um tamanho de fonte padrão de 9 pontos. À medida que aumenta o tamanho da fonte, você será capaz de ver os furos nas letras mais claramente. À medida que reduz o tamanho da fonte, os furos serão menos visíveis. Qual é o menor tamanho da fonte em que você começa a perceber os furos?

12.20 (Professor de digitação: aprimorando uma habilidade crucial na era da informática) Digitar rápida e corretamente é uma habilidade essencial para trabalhar de forma eficaz com computadores e a internet. Neste exercício, você construirá um aplicativo GUI que pode ajudar os usuários a aprender a digitar corretamente sem olhar para o teclado. O aplicativo deve exibir um *teclado virtual* (Figura 12.50) e permitir que o usuário veja o que ele está digitando na tela sem olhar para o *teclado real*. Use `JButtons` para representar as teclas. À medida que o usuário pressiona cada tecla, o aplicativo destaca o `JButton` correspondente na GUI e adiciona o caractere a uma `JTextArea` que mostra o que o usuário digitou até agora. [Dica: para destacar um `JButton`, use o método `setBackground` para mudar a cor de fundo. Quando a tecla é liberada, redefina a cor original do fundo. Você pode obter a cor original de fundo do `JButton` com o método `getBackground` antes de mudar a cor.]

Você pode testar seu programa digitando um pangrama — uma frase que contém todas as letras do alfabeto pelo menos uma vez — como “*The quick brown fox jumps over a lazy dog*” ou, em português, “Um pequeno jabuti xereta viu dez cegonhas felizes”. Você pode encontrar outros pangramas na web.

Para tornar o programa mais interessante, monitore a precisão do usuário. Você pode fazer com que o usuário digite frases específicas que você pré-armazenou no seu programa e que você exibe na tela acima do teclado virtual. Pode-se monitorar quantos pressionamentos de tecla o usuário digita corretamente e quantos são digitados incorretamente. Pode-se também monitorar com quais teclas o usuário tem dificuldade e exibir um relatório mostrando essas teclas.

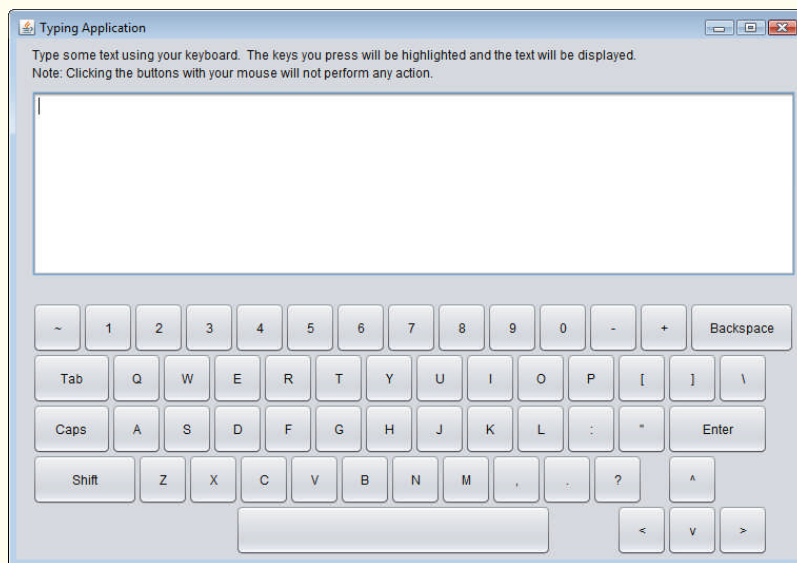


Figura 12.50 | Professor de digitação.