Chapter 18

Graphical User Interfaces

CHAPTER GOALS

- To become familiar with common user-interface components, such as text components, radio buttons, check boxes, and menus
- To understand the use of layout managers to arrange user-interface components in a container
- To build programs that handle events from user-interface components
- To learn how to browse the Java documentation

In this chapter, we will delve more deeply into graphical user interface programming. The graphical applications with which you are familiar have many visual gadgets for information entry: text components, buttons, scroll bars, menus, and so on. In this chapter, you will learn how to use the most common user-interface components in the Java Swing user-interface toolkit. Swing has many more components than can be mastered in a first course, and even the basic components have advanced options that can't be covered here. In fact, few programmers try to learn everything about a particular user-interface component. It is more important to understand the concepts and to search the Java documentation for the details. This chapter walks you through one example to show you how the Java documentation is organized and how you can rely on it for your programming.

CHAPTER CONTENTS

18.1 Processing Text Input

We start our discussion of graphical user interfaces with text input. Of course, a graphical application can receive text input by calling the `showInputDialog` method of the `JOptionPane` class, but popping up a separate dialog box for each input is not a natural user interface. Most graphical programs collect text input through **text fields** (see Figure 1). In this section, you will learn how to add text fields to a graphical application, and how to read what the user types into them.

Use `JTextField` components to provide space for user input. Place a `JLabel` next to each text field.

The `JTextField` class provides a text field. When you construct a text field, you need to supply the width—the approximate number of characters that you expect the user to type.

```
final int FIELD_WIDTH = 10;
final JTextField rateField = new JTextField(FIELD_WIDTH);
```

Users can type additional characters, but then a part of the contents of the field becomes invisible.

You will want to label each text field so that the user knows what to type into it. Construct a `JLabel` object for each label:

```
JLabel rateLabel = new JLabel("Interest Rate: ");
```

You want to give the user an opportunity to enter all information into the text fields before processing it. Therefore, you should supply a button that the user can press to indicate that the input is ready for processing.

When that button is clicked, its `actionPerformed` method reads the user input from the text field, using the `getText` method of the `JTextField` class. The `getText` method returns a `String` object. In our sample program, we turn the string into a number, using the `Double.parseDouble` method. After updating the account, we show the balance in another label.

```
class AddInterestListener implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      double rate = Double.parseDouble(rateField.getText());
      double interest = account.getBalance() * rate / 100;
      account.deposit(interest);
      resultLabel.setText("balance: " + account.getBalance());
   }
}
```

The following application is a useful prototype for a graphical user-interface front end for arbitrary calculations. You can easily modify it for your own needs. Place

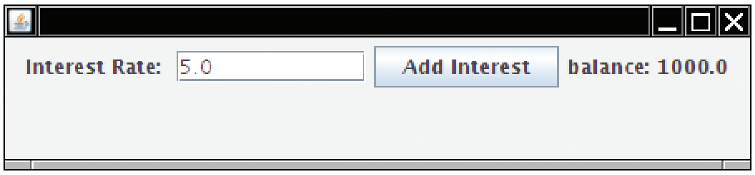


Figure 1 An Application with a Text Field

other input components into the frame. Change the contents of the `actionPerformed` method to carry out other calculations. Display the result in a label.

ch18/textfield/InvestmentViewer3.java

```java
import javax.swing.JFrame;

/**
   This program displays the growth of an investment.
*/
public class InvestmentViewer3
{
   public static void main(String[] args)
   {
      JFrame frame = new InvestmentFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

ch18/textfield/InvestmentFrame.java

```java
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

/**
   A frame that shows the growth of an investment with variable interest.
*/
public class InvestmentFrame extends JFrame
{
   private static final int FRAME_WIDTH = 450;
   private static final int FRAME_HEIGHT = 100;

   private static final double DEFAULT_RATE = 5;
   private static final double INITIAL_BALANCE = 1000;

   private JLabel rateLabel;
   private JTextField rateField;
   private JButton button;
   private JLabel resultLabel;
   private JPanel panel;
   private BankAccount account;

   public InvestmentFrame()
   {
      account = new BankAccount(INITIAL_BALANCE);

      // Use instance variables for components
      resultLabel = new JLabel("balance: " + account.getBalance());

      // Use helper methods
      createTextField();
      createButton();
      createPanel();

      setSize(FRAME_WIDTH, FRAME_HEIGHT);
   }

   private void createTextField()
   {
      rateLabel = new JLabel("Interest Rate: ");

      final int FIELD_WIDTH = 10;
      rateField = new JTextField(FIELD_WIDTH);
      rateField.setText("" + DEFAULT_RATE);
   }

   private void createButton()
   {
      button = new JButton("Add Interest");

      class AddInterestListener implements ActionListener
      {
         public void actionPerformed(ActionEvent event)
         {
            double rate = Double.parseDouble(rateField.getText());
            double interest = account.getBalance() * rate / 100;
            account.deposit(interest);
            resultLabel.setText("balance: " + account.getBalance());
         }
      }

      ActionListener listener = new AddInterestListener();
      button.addActionListener(listener);
   }

   private void createPanel()
   {
      panel = new JPanel();
      panel.add(rateLabel);
      panel.add(rateField);
      panel.add(button);
      panel.add(resultLabel);
      add(panel);
   }
}
```

SELF CHECK

1. What happens if you omit the first `JLabel` object?
2. If a text field holds an integer, what expression do you use to read its contents?

18.2 Text Areas

Use a JTextArea to show multiple lines of text.

In the preceding section, you saw how to construct text fields. A text field holds a single line of text. To display multiple lines of text, use the JTextArea class.

When constructing a text area, you can specify the number of rows and columns:

```
final int ROWS = 10;
final int COLUMNS = 30;
JTextArea textArea = new JTextArea(ROWS, COLUMNS);
```

Use the setText method to set the text of a text field or text area. The append method adds text to the end of a text area. Use newline characters to separate lines, like this:

```
textArea.append(account.getBalance() + "\n");
```

If you want to use a text field or text area for display purposes only, call the setEditable method like this

```
textArea.setEditable(false);
```

Now the user can no longer edit the contents of the field, but your program can still call setText and append to change it.

As shown in Figure 2, the JTextField and JTextArea classes are subclasses of the class JTextComponent. The methods setText and setEditable are declared in the JTextComponent class and inherited by JTextField and JTextArea. However, the append method is declared in the JTextArea class.

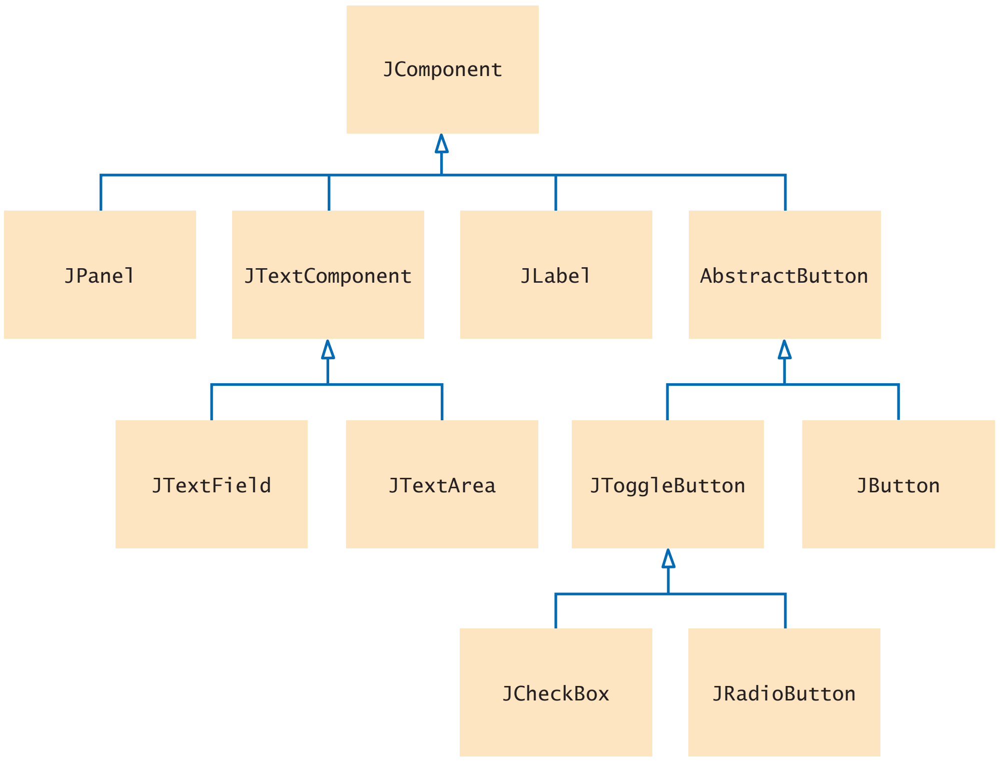
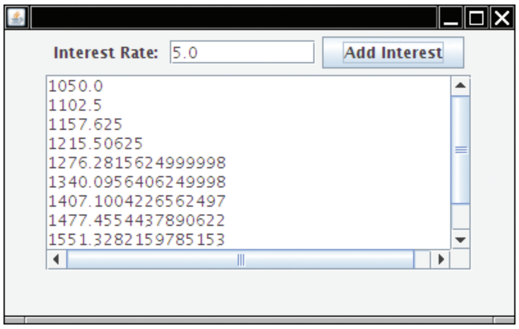


Figure 2 A Part of the Hierarchy of Swing User-Interface Components



```
1050.0
1102.5
1157.625
1215.50625
1276.2815624999998
1340.0956406249998
1407.1004226562497
1477.4554437890622
1551.3282159785153
```

Figure 3 The Investment Application with a Text Area

You can add scroll bars to any component with a `JScrollPane`.

To add scroll bars to a text area, use a `JScrollPane`, like this:

```
JTextArea textArea = new JTextArea(ROWS, COLUMNS);
JScrollPane scrollPane = new JScrollPane(textArea);
```

Then add the scroll pane to the panel. Figure 3 shows the result.

The following sample program puts these concepts together. A user can enter numbers into the interest rate text field and then click on the "Add Interest" button). The interest rate is applied, and the updated balance is appended to the text area. The text area has scroll bars and is not editable.

This program is similar to the previous investment viewer program, but it keeps track of all the bank balances, not just the last one.

ch18/textarea/InvestmentFrame.java

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

/**
   A frame that shows the growth of an investment with variable interest.
*/
public class InvestmentFrame extends JFrame
{
   private static final int FRAME_WIDTH = 400;
   private static final int FRAME_HEIGHT = 250;

   private static final int AREA_ROWS = 10;
   private static final int AREA_COLUMNS = 30;
   private static final double DEFAULT_RATE = 5;
   private static final double INITIAL_BALANCE = 1000;

   private JLabel rateLabel;
   private JTextField rateField;
   private JButton button;
   private JTextArea resultArea;
   private JPanel panel;
   private BankAccount account;

   public InvestmentFrame()
   {
      account = new BankAccount(INITIAL_BALANCE);
      resultArea = new JTextArea(AREA_ROWS, AREA_COLUMNS);
      resultArea.setEditable(false);

      // Use helper methods
      createTextField();
      createButton();
      createPanel();

      setSize(FRAME_WIDTH, FRAME_HEIGHT);
   }

   private void createTextField()
   {
      rateLabel = new JLabel("Interest Rate: ");

      final int FIELD_WIDTH = 10;
      rateField = new JTextField(FIELD_WIDTH);
      rateField.setText("" + DEFAULT_RATE);
   }

   private void createButton()
   {
      button = new JButton("Add Interest");

   class AddInterestListener implements ActionListener
      {
         public void actionPerformed(ActionEvent event)
         {
            double rate = Double.parseDouble(rateField.getText());
            double interest = account.getBalance() * rate / 100;
            account.deposit(interest);
            resultArea.append(account.getBalance() + "\n");
         }
      }

      ActionListener listener = new AddInterestListener();
      button.addActionListener(listener);
   }

   private void createPanel()
   {
      panel = new JPanel();
      panel.add(rateLabel);
      panel.add(rateField);
      panel.add(button);
      JScrollPane scrollPane = new JScrollPane(resultArea);
      panel.add(scrollPane);
      add(panel);
   }
}
```

SELF CHECK

3. What is the difference between a text field and a text area?
4. Why did the `InvestmentFrame` program call `resultArea.setEditable(false)`?
5. How would you modify the `InvestmentFrame` program if you didn't want to use scroll bars?

18.3 Layout Management

User-interface components are arranged by placing them inside containers.

Up to now, you have had limited control over the layout of user-interface components. You learned how to add components to a panel. The panel arranged the components from the left to the right. However, in many applications, you need more sophisticated arrangements.

Each container has a layout manager that directs the arrangement of its components.

In Java, you build up user interfaces by adding components into containers such as panels. Each container has its own **layout manager**, which determines how the components are laid out.

Three useful layout managers are the border layout, flow layout, and grid layout.

By default, a `JPanel` uses a **flow layout**. A flow layout simply arranges its components from left to right and starts a new row when there is no more room in the current row.

Another commonly used layout manager is the **border layout**. The border layout groups components into five areas: center, north, west, south, and east (see Figure 4). Not all of the areas need to be occupied.

When adding a component to a container with the border layout, specify the `NORTH`, `EAST`, `SOUTH`, `WEST`, or `CENTER` position.

The border layout is the default layout manager for a frame (or, more technically, the frame's content pane). But you can also use the border layout in a panel:

```
panel.setLayout(new BorderLayout());
```

Now the panel is controlled by a border layout, not the flow layout. When adding a component, you specify the position, like this:

```
panel.add(component, BorderLayout.NORTH);
```

The content pane of a frame has a border layout by default. A panel has a flow layout by default.

The **grid layout** is a third layout that is sometimes useful. The grid layout arranges components in a grid with a fixed number of rows and columns, resizing each of the components so that they all have the same size. Like the border layout, it also expands each component to fill the entire allotted area. (If that is not desirable, you need to place each component inside a panel.) Figure 5 shows a number pad panel that uses a grid layout. To create a grid layout, you supply the number of rows and columns in the constructor, then add the components, row by row, left to right:

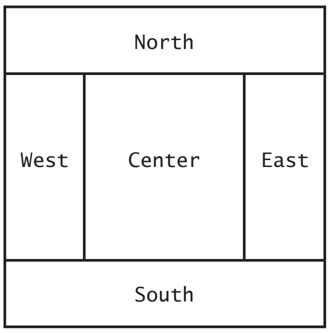
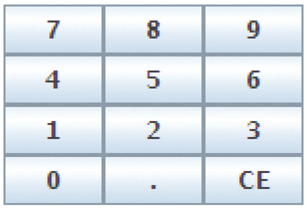


Figure 4
Components Expand to Fill Space in the Border Layout

Figure 5
The Grid Layout



```
JPanel buttonPanel = new JPanel();
buttonPanel.setLayout(new GridLayout(4, 3));
buttonPanel.add(button7);
buttonPanel.add(button8);
buttonPanel.add(button9);
buttonPanel.add(button4);
. . .
```

Sometimes you want to have a tabular arrangement of the components where columns have different sizes or one component spans multiple columns. A more complex layout manager called the *grid bag layout* can handle these situations. The grid bag layout is quite complex to use, however, and we do not cover it in this book; see, for example, Cay S. Horstmann and Gary Cornell, *Core Java 2 Volume 1: Fundamentals,* 8th edition (Prentice Hall, 2008), for more information. Java 6 introduces a group layout that is designed for use by interactive tools—see Productivity Hint 18.1 on page 757.

Fortunately, you can create acceptable-looking layouts in nearly all situations by nesting panels. You give each panel an appropriate layout manager. Panels don't have visible borders, so you can use as many panels as you need to organize your components. Figure 6 shows an example. The keypad buttons are contained in a panel with grid layout. That panel is itself contained in a larger panel with border layout. The text field is in the northern position of the larger panel. The following code produces this arrangement:

```
JPanel keypadPanel = new JPanel();
keypadPanel.setLayout(new BorderLayout());
buttonPanel = new JPanel();
buttonPanel.setLayout(new GridLayout(4, 3));
buttonPanel.add(button7);
buttonPanel.add(button8);
// . . .
keypadPanel.add(buttonPanel, BorderLayout.CENTER);
JTextField display = new JTextField();
keypadPanel.add(display, BorderLayout.NORTH);
```

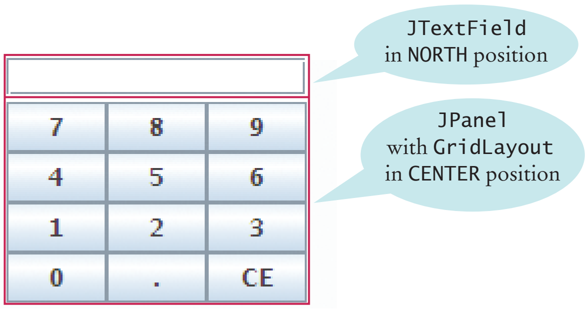

JTextField in NORTH position
JPanel with GridLayout in CENTER position
7
8
9
4
5
6
1
2
3
0
.
CE

Figure 6
Nesting Panels

SELF CHECK

6. How do you add two buttons to the north area of a frame?
7. How can you stack three buttons on top of each other?

18.4 Choices

In the following sections, you will see how to present a finite set of choices to the user. Which Swing component you use depends on whether the choices are mutually exclusive or not, and on the amount of space you have for displaying the choices.

18.4.1 Radio Buttons

For a small set of mutually exclusive choices, use a group of radio buttons or a combo box.

If the choices are mutually exclusive, use a set of **radio buttons**. In a radio button set, only one button can be selected at a time. When the user selects another button in the same set, the previously selected button is automatically turned off. (These buttons are called radio buttons because they work like the station selector buttons on a car radio: If you select a new station, the old station is automatically deselected.) For example, in Figure 7, the font sizes are mutually exclusive. You can select small, medium, or large, but not a combination of them.

Add radio buttons into a ButtonGroup so that only one button in the group is on at any time.

To create a set of radio buttons, first create each button individually, and then add all buttons of the set to a ButtonGroup object:

```
JRadioButton smallButton = new JRadioButton("Small");
JRadioButton mediumButton = new JRadioButton("Medium");
JRadioButton largeButton = new JRadioButton("Large");

ButtonGroup group = new ButtonGroup();
group.add(smallButton);
group.add(mediumButton);
group.add(largeButton);
```

Note that the button group does *not* place the buttons close to each other on the container. The purpose of the button group is simply to find out which buttons to turn off when one of them is turned on. It is still your job to arrange the buttons on the screen.

The isSelected method is called to find out whether a button is currently selected or not. For example,

```
if (largeButton.isSelected()) { size = LARGE_SIZE; }
```

Because users will expect one radio button in a radio button group to be selected, call setSelected(true) on the default radio button before making the enclosing frame visible.

You can place a border around a panel to group its contents visually.

If you have multiple button groups, it is a good idea to group them together visually. It is a good idea to use a panel for each set of radio buttons, but the panels themselves are invisible. You can add a *border* to a panel to make it visible. In Figure 7, for example, the panels containing the Size radio buttons and Style check boxes have borders.

There are a large number of border types. We will show only a couple of variations and leave it to the border enthusiasts to look up the others in the Swing

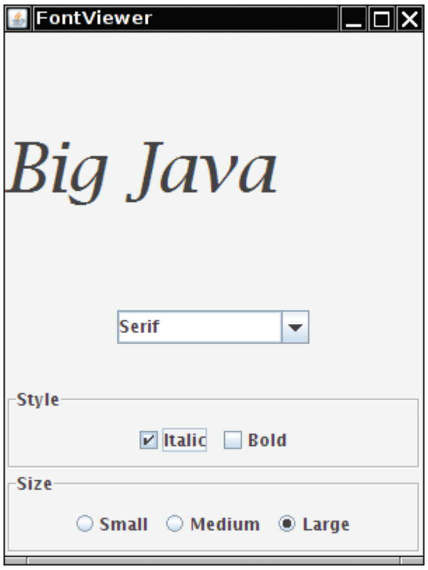


Figure 7
A Combo Box, Check Boxes, and Radio Buttons

documentation. The `EtchedBorder` class yields a border with a three-dimensional, etched effect. You can add a border to any component, but most commonly you apply it to a panel:

```
JPanel panel = new JPanel();
panel.setBorder(new EtchedBorder());
```

If you want to add a title to the border (as in Figure 7), you need to construct a `TitledBorder`. You make a titled border by supplying a basic border and then the title you want. Here is a typical example:

```
panel.setBorder(new TitledBorder(new EtchedBorder(), "Size"));
```

18.4.2 Check Boxes

For a binary choice, use a check box.

A check box is a user-interface component with two states: checked and unchecked. You use a group of check boxes when one selection does not exclude another. For example, the choices for “Bold” and “Italic” in Figure 7 are not exclusive. You can choose either, both, or neither. Therefore, they are implemented as a set of separate check boxes. Radio buttons and check boxes have different visual appearances. Radio buttons are round and have a black dot when selected. Check boxes are square and have a check mark when selected.

You construct a check box by giving the name in the constructor:

```
JCheckBox italicCheckBox = new JCheckBox("Italic");
```

Because check box settings do not exclude each other, you do not place a set of check boxes inside a button group.

As with radio buttons, you use the `isSelected` method to find out whether a check box is currently checked or not.

18.4.3 Combo Boxes

For a large set of choices, use a combo box.

If you have a large number of choices, you don't want to make a set of radio buttons, because that would take up a lot of space. Instead, you can use a **combo box**. This component is called a combo box because it is a combination of a list and a text field. The text field displays the name of the current selection. When you click on the arrow to the right of the text field of a combo box, a list of selections drops down, and you can choose one of the items in the list (see Figure 8).

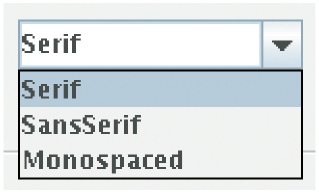


Figure 8
An Open Combo Box

If the combo box is *editable*, you can also type in your own selection. To make a combo box editable, call the `setEditable` method.

You add strings to a combo box with the `addItem` method.

```
JComboBox facenameCombo = new JComboBox();
facenameCombo.addItem("Serif");
facenameCombo.addItem("SansSerif");
. . .
```

You get the item that the user has selected by calling the `getSelectedItem` method. However, because combo boxes can store other objects in addition to strings, the `getSelectedItem` method has return type `Object`. Hence you must cast the returned value back to `String`.

```
String selectedString = (String) facenameCombo.getSelectedItem();
```

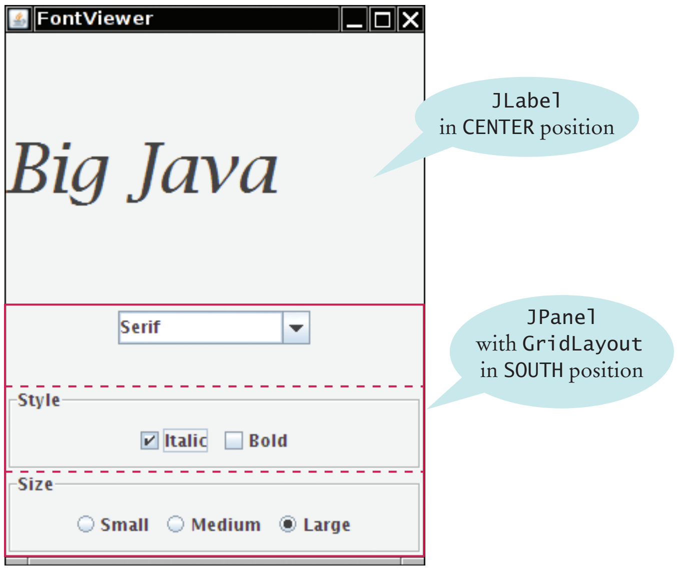

FontViewer
Big Java
JLabel in CENTER position
Serif
Style
Italic
Bold
Size
Small
Medium
Large
JPanel with GridLayout in SOUTH position

Figure 9 The Components of the `FontViewerFrame`

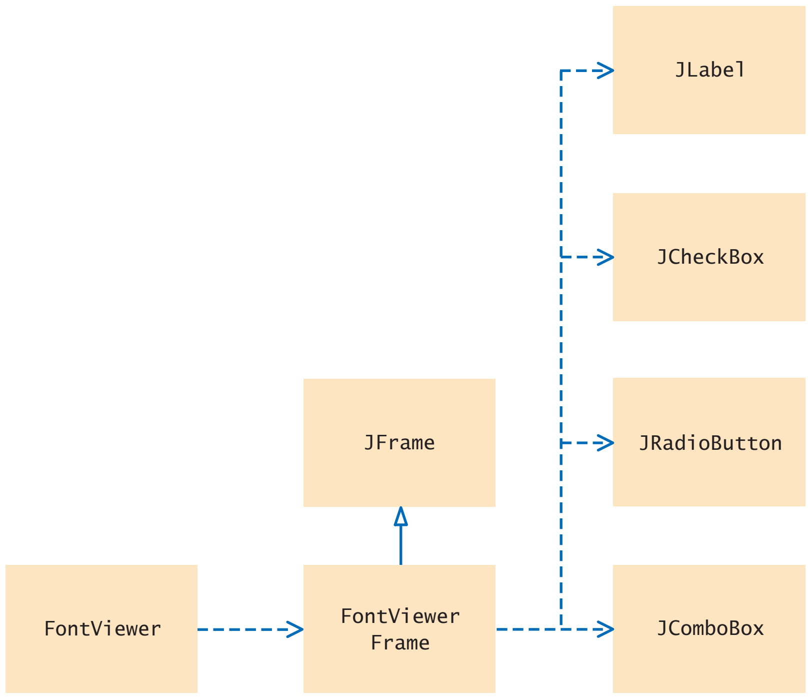


Figure 10 Classes of the Font Viewer Program

Radio buttons, check boxes, and combo boxes generate action events, just as buttons do.

You can select an item for the user with the `setSelectedItem` method.

Radio buttons, check boxes, and combo boxes generate an `ActionEvent` whenever the user selects an item. In the following program, we don't care which component was clicked—all components notify the same listener object. Whenever the user clicks on any one of them, we simply ask each component for its current content, using the `isSelected` and `getSelectedItem` methods. We then redraw the text sample with the new font.

Figure 9 shows how the components are arranged in the frame. Figure 10 shows the relationships between the classes used in the font viewer program.

ch18/choice/FontViewer.java

```
import javax.swing.JFrame;

/**
   This program allows the user to view font effects.
*/
public class FontViewer
{
   public static void main(String[] args)
   {
      JFrame frame = new FontViewerFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setTitle("FontViewer");
      frame.setVisible(true);
   }
}
```

ch18/choice/FontViewerFrame.java

```java
import java.awt.BorderLayout;
import java.awt.Font;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.ButtonGroup;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JRadioButton;
import javax.swing.border.EtchedBorder;
import javax.swing.border.TitledBorder;

/**
   This frame contains a text field and a control panel
   to change the font of the text.
*/
public class FontViewerFrame extends JFrame
{
   private static final int FRAME_WIDTH = 300;
   private static final int FRAME_HEIGHT = 400;

   private JLabel sampleField;
   private JCheckBox italicCheckBox;
   private JCheckBox boldCheckBox;
   private JRadioButton smallButton;
   private JRadioButton mediumButton;
   private JRadioButton largeButton;
   private JComboBox facenameCombo;
   private ActionListener listener;

   /**
      Constructs the frame.
   */
   public FontViewerFrame()
   {
      // Construct text sample
      sampleField = new JLabel("Big Java");
      add(sampleField, BorderLayout.CENTER);

      // This listener is shared among all components
      class ChoiceListener implements ActionListener
      {
         public void actionPerformed(ActionEvent event)
         {
            setSampleFont();
         }
      }

      listener = new ChoiceListener();

      createControlPanel();
      setSampleFont();
      setSize(FRAME_WIDTH, FRAME_HEIGHT);
   }

   /**
      Creates the control panel to change the font.
   */
   public void createControlPanel()
   {
      JPanel facenamePanel = createComboBox();
      JPanel sizeGroupPanel = createCheckBoxes();
      JPanel styleGroupPanel = createRadioButtons();

      // Line up component panels

      JPanel controlPanel = new JPanel();
      controlPanel.setLayout(new GridLayout(3, 1));
      controlPanel.add(facenamePanel);
      controlPanel.add(sizeGroupPanel);
      controlPanel.add(styleGroupPanel);

      // Add panels to content pane

      add(controlPanel, BorderLayout.SOUTH);
   }

   /**
      Creates the combo box with the font style choices.
      @return the panel containing the combo box
   */
   public JPanel createComboBox()
   {
      facenameCombo = new JComboBox();
      facenameCombo.addItem("Serif");
      facenameCombo.addItem("SansSerif");
      facenameCombo.addItem("Monospaced");
      facenameCombo.setEditable(true);
      facenameCombo.addActionListener(listener);

      JPanel panel = new JPanel();
      panel.add(facenameCombo);
      return panel;
   }

   /**
      Creates the check boxes for selecting bold and italic styles.
      @return the panel containing the check boxes
   */
   public JPanel createCheckBoxes()
   {
      italicCheckBox = new JCheckBox("Italic");
      italicCheckBox.addActionListener(listener);

      boldCheckBox = new JCheckBox("Bold");
      boldCheckBox.addActionListener(listener);

      JPanel panel = new JPanel();
      panel.add(italicCheckBox);
      panel.add(boldCheckBox);
      panel.setBorder(new TitledBorder(new EtchedBorder(), "Style"));

      return panel;
   }

   /**
      Creates the radio buttons to select the font size.
      @return the panel containing the radio buttons
   */
   public JPanel createRadioButtons()
   {
      smallButton = new JRadioButton("Small");
      smallButton.addActionListener(listener);

      mediumButton = new JRadioButton("Medium");
      mediumButton.addActionListener(listener);

      largeButton = new JRadioButton("Large");
      largeButton.addActionListener(listener);
      largeButton.setSelected(true);

      // Add radio buttons to button group

      ButtonGroup group = new ButtonGroup();
      group.add(smallButton);
      group.add(mediumButton);
      group.add(largeButton);

      JPanel panel = new JPanel();
      panel.add(smallButton);
      panel.add(mediumButton);
      panel.add(largeButton);
      panel.setBorder(new TitledBorder(new EtchedBorder(), "Size"));

      return panel;
   }

   /**
      Gets user choice for font name, style, and size
      and sets the font of the text sample.
   */
   public void setSampleFont()
   {
      // Get font name
      String facename
            = (String) facenameCombo.getSelectedItem();

      // Get font style

      int style = 0;
      if (italicCheckBox.isSelected())
      {
         style = style + Font.ITALIC;
      }
      if (boldCheckBox.isSelected())
      {
         style = style + Font.BOLD;
      }

      // Get font size

      int size = 0;
```

```

        final int SMALL_SIZE = 24;
        final int MEDIUM_SIZE = 36;
        final int LARGE_SIZE = 48;

        if (smallButton.isSelected()) { size = SMALL_SIZE; }
        else if (mediumButton.isSelected()) { size = MEDIUM_SIZE; }
        else if (largeButton.isSelected()) { size = LARGE_SIZE; }

        // Set font of text field

        sampleField.setFont(new Font(facename, style, size));
        sampleField.repaint();
    }
}
```

SELF CHECK

8. What is the advantage of a `JComboBox` over a set of radio buttons? What is the disadvantage?
9. Why do all user-interface components in the `FontViewerFrame` class share the same listener?
10. Why was the combo box placed inside a panel? What would have happened if it had been added directly to the control panel?

How To 18.1 Laying Out a User Interface

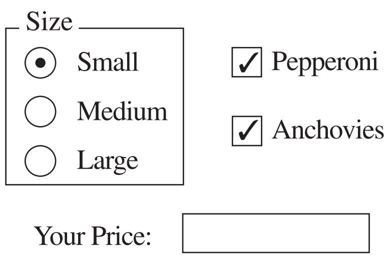
A graphical user interface is made up of components such as buttons and text fields. The Swing library uses containers and layout managers to arrange these components. This How To explains how to group components into containers and how to pick the right layout managers.

Step 1 Make a sketch of your desired component layout.

Draw all the buttons, labels, text fields, and borders on a sheet of paper. Graph paper works best.

Here is an example—a user interface for ordering pizza. The user interface contains

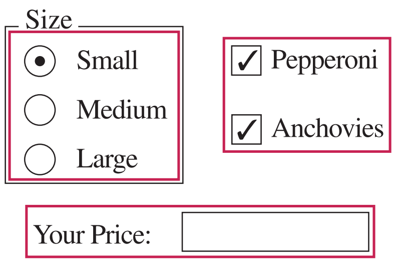
- Three radio buttons
- Two check boxes
- A label: "Your Price:"
- A text field
- A border



Step 2 Find groupings of adjacent components with the same layout.

Usually, the component arrangement is complex enough that you need to use several panels, each with its own layout manager. Start by looking at adjacent components that are arranged top to bottom or left to right. If several components are surrounded by a border, they should be grouped together.

Here are the groupings from the pizza user interface:



Step 3 Identify layouts for each group.

When components are arranged horizontally, choose a flow layout. When components are arranged vertically, use a grid layout with one column.

In the pizza user interface example, you would choose

- A (3, 1) grid layout for the radio buttons
- A (2, 1) grid layout for the check boxes
- A flow layout for the label and text field

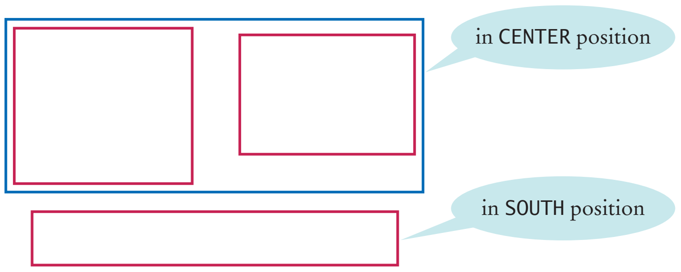
Step 4 Group the groups together.

Look at each group as one blob, and group the blobs together into larger groups, just as you grouped the components in the preceding step. If you note one large blob surrounded by smaller blobs, you can group them together in a border layout.

You may have to repeat the grouping again if you have a very complex user interface. You are done if you have arranged all groups in a single container.

For example, the three component groups of the pizza user interface can be arranged as:

- A group containing the first two component groups, placed in the center of a container with a border layout.
- The third component group, in the southern area of that container.



In this step, you may run into a couple of complications. The group "blobs" tend to vary in size more than the individual components. If you place them inside a grid layout, the grid layout forces them all to be the same size. Also, you occasionally would like a component from one group to line up with a component from another group, but there is no way for you to communicate that intent to the layout managers.

These problems can be overcome by using more sophisticated layout managers or implementing a custom layout manager. However, those techniques are beyond the scope of this book. Sometimes, you may want to start over with Step 1, using a component layout that is easier to manage. Or you can decide to live with minor imperfections of the layout. Don't worry about achieving the perfect layout—after all, you are learning programming, not user-interface design.

Step 5 Write the code to generate the layout.

This step is straightforward but potentially tedious, especially if you have a large number of components.

Start by constructing the components. Then construct a panel for each component group and set its layout manager if it is not a flow layout (the default for panels). Add a border to the panel if required. Finally, add the components to their panels. Continue in this fashion until you reach the outermost containers, which you add to the frame.

Here is an outline of the code required for the pizza user interface.

```
JPanel radioButtonPanel = new JPanel();
radioButtonPanel.setLayout(new GridLayout(3, 1));
radioButton.setBorder(new TitledBorder(new EtchedBorder(), "Size"));
radioButtonPanel.add(smallButton);
radioButtonPanel.add(mediumButton);
radioButtonPanel.add(largeButton);

JPanel checkBoxPanel = new JPanel();
checkBoxPanel.setLayout(new GridLayout(2, 1));
checkBoxPanel.add(pepperoniButton());
checkBoxPanel.add(anchoviesButton());

JPanel pricePanel = new JPanel(); // Uses FlowLayout by default
pricePanel.add(new JLabel("Your Price:"));
pricePanel.add(priceTextField);

JPanel centerPanel = new JPanel(); // Uses FlowLayout
centerPanel.add(radioButtonPanel);
centerPanel.add(checkBoxPanel);

// Frame uses BorderLayout by default
add(centerPanel, BorderLayout.CENTER);
add(pricePanel, BorderLayout.SOUTH);
```

Of course, you also need to add event handlers to the components. See How To 10.1.

Productivity Hint 18.1

Use a GUI Builder

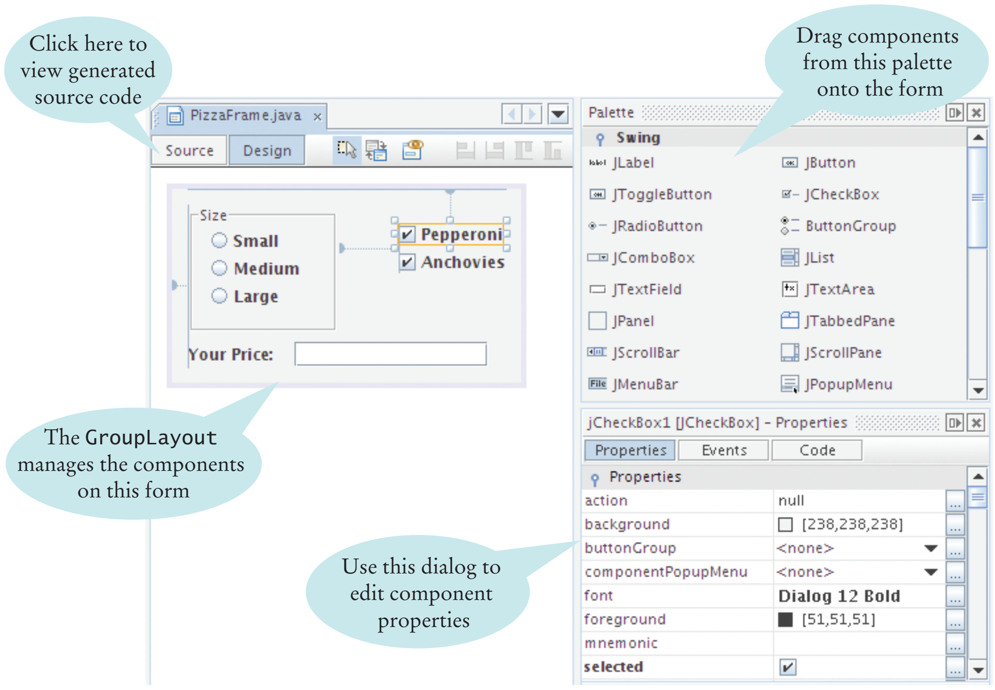
As you have seen, implementing even a simple graphical user interface in Java is quite tedious. You have to write a lot of code for constructing components, using layout managers, and providing event handlers. Most of the code is boring and repetitive.

A GUI builder takes away much of the tedium. Most GUI builders help you in three ways:

- You drag and drop components onto a panel. The GUI builder writes the layout management code for you.
- You customize components with a dialog box, setting properties such as fonts, colors, text, and so on. The GUI builder writes the customization code for you.
- You provide event handlers by picking the event to process and providing just the code snippet for the listener method. The GUI builder writes the boilerplate code for attaching a listener object.

Java 6 introduced `GroupLayout`, a powerful layout manager that was specifically designed to be used by GUI builders. The free NetBeans development environment, available from `http://netbeans.org`, makes use of this layout manager—see Figure 11.

If you need to build a complex user interface, you will find that learning to use a GUI builder is a very worthwhile investment. You will spend less time writing boring code, and you will have more fun designing your user interface and focusing on the functionality of your program.



action	null
background	[238,238,238]
buttonGroup	<none>
componentPopupMenu	<none>
font	Dialog 12 Bold
foreground	[51,51,51]
mnemonic	
selected	✓

Figure 11 A GUI Builder

18.5 Menus

A frame contains a menu bar. The menu bar contains menus. A menu contains submenus and menu items.

Anyone who has ever used a graphical user interface is familiar with pull-down menus (see Figure 12). In Java it is easy to create these menus.

The container for the top-level menu items is called a *menu bar*. A *menu* is a collection of *menu items* and more menus (submenus). You add menu items and submenus with the add method:

```
JMenuItem fileExitItem = new JMenuItem("Exit");
fileMenu.add(fileExitItem);
```

Menu items generate action events.

A menu item has no further submenus. When the user selects a menu item, the menu item sends an action event. Therefore, you want to add a listener to each menu item:

```
fileExitItem.addActionListener(listener);
```

You add action listeners only to menu items, not to menus or the menu bar. When the user clicks on a menu name and a submenu opens, no action event is sent.

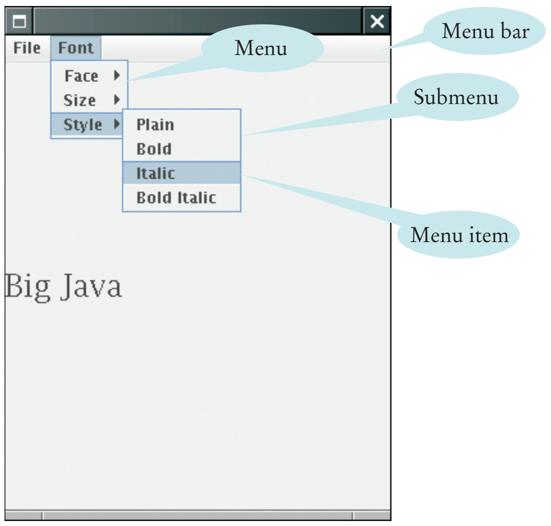


Figure 12 Pull-Down Menus

The following program builds up a small but typical menu and traps the action events from the menu items. To keep the program readable, it is a good idea to use a separate method for each menu or set of related menus. Have a look at the `createFaceItem` method, which creates a menu item to change the font face. The same listener class takes care of three cases, with the `name` parameters varying for each menu item. The same strategy is used for the `createSizeItem` and `createStyleItem` methods.

ch18/menu/FontViewer2.java

```
import javax.swing.JFrame;

/**
   This program uses a menu to display font effects.
*/
public class FontViewer2
{
   public static void main(String[] args)
   {
      JFrame frame = new FontViewer2Frame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

ch18/menu/FontViewer2Frame.java

```
import java.awt.BorderLayout;
import java.awt.Font;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.ButtonGroup;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;
import javax.swing.JRadioButton;
import javax.swing.border.EtchedBorder;
import javax.swing.border.TitledBorder;

/**
   This frame has a menu with commands to change the font
   of a text sample.
*/
public class FontViewer2Frame extends JFrame
{
   private static final int FRAME_WIDTH = 300;
   private static final int FRAME_HEIGHT = 400;

   private JLabel sampleField;
   private String facename;
   private int fontstyle;
   private int fontsize;

   /**
      Constructs the frame.
   */
   public FontViewer2Frame()
   {
      // Construct text sample
      sampleField = new JLabel("Big Java");
      add(sampleField, BorderLayout.CENTER);

      // Construct menu
      JMenuBar menuBar = new JMenuBar();
      setJMenuBar(menuBar);
      menuBar.add(createFileMenu());
      menuBar.add(createFontMenu());

      facename = "Serif";
      fontsize = 24;
      fontstyle = Font.PLAIN;

      setSampleFont();
      setSize(FRAME_WIDTH, FRAME_HEIGHT);
   }

   /**
      Creates the File menu.
      @return the menu
   */
   public JMenu createFileMenu()

   {
      JMenu menu = new JMenu("File");
      menu.add(createFileExitItem());
      return menu;
   }

   /**
      Creates the File->Exit menu item and sets its action listener.
      @return the menu item
   */
   public JMenuItem createFileExitItem()
   {
      JMenuItem item = new JMenuItem("Exit");
      class MenuItemListener implements ActionListener
      {
         public void actionPerformed(ActionEvent event)
         {
            System.exit(0);
         }
      }
      ActionListener listener = new MenuItemListener();
      item.addActionListener(listener);
      return item;
   }

   /**
      Creates the Font submenu.
      @return the menu
   */
   public JMenu createFontMenu()
   {
      JMenu menu = new JMenu("Font");
      menu.add(createFaceMenu());
      menu.add(createSizeMenu());
      menu.add(createStyleMenu());
      return menu;
   }

   /**
      Creates the Face submenu.
      @return the menu
   */
   public JMenu createFaceMenu()
   {
      JMenu menu = new JMenu("Face");
      menu.add(createFaceItem("Serif"));
      menu.add(createFaceItem("SansSerif"));
      menu.add(createFaceItem("Monospaced"));
      return menu;
   }

    /**
      Creates the Size submenu.
      @return the menu
   */
   public JMenu createSizeMenu()
   {
      JMenu menu = new JMenu("Size");
      menu.add(createSizeItem("Smaller", -1));
      menu.add(createSizeItem("Larger", 1));
      return menu;
   }

   /**
      Creates the Style submenu.
      @return the menu
   */
   public JMenu createStyleMenu()
   {
      JMenu menu = new JMenu("Style");
      menu.add(createStyleItem("Plain", Font.PLAIN));
      menu.add(createStyleItem("Bold", Font.BOLD));
      menu.add(createStyleItem("Italic", Font.ITALIC));
      menu.add(createStyleItem("Bold Italic", Font.BOLD
            + Font.ITALIC));
      return menu;
   }

   /**
      Creates a menu item to change the font face and set its action listener.
      @param name the name of the font face
      @return the menu item
   */
   public JMenuItem createFaceItem(final String name)
   {
      JMenuItem item = new JMenuItem(name);
      class MenuItemListener implements ActionListener
      {
         public void actionPerformed(ActionEvent event)
         {
            facename = name;
            setSampleFont();
         }
      }
      ActionListener listener = new MenuItemListener();
      item.addActionListener(listener);
      return item;
   }

   /**
      Creates a menu item to change the font size
      and set its action listener.
      @param name the name of the menu item
      @param ds the amount by which to change the size
      @return the menu item
   */
   public JMenuItem createSizeItem(String name, final int ds)
   {
      JMenuItem item = new JMenuItem(name);
      class MenuItemListener implements ActionListener
      {
         public void actionPerformed(ActionEvent event)
         {
            fontsize = fontsize + ds;
            setSampleFont();
         }
      }
      ActionListener listener = new MenuItemListener();
      item.addActionListener(listener);
      return item;
   }

   /**
      Creates a menu item to change the font style
      and set its action listener.
      @param name the name of the menu item
      @param style the new font style
      @return the menu item
   */
   public JMenuItem createStyleItem(String name, final int style)
   {
      JMenuItem item = new JMenuItem(name);
      class MenuItemListener implements ActionListener
      {
         public void actionPerformed(ActionEvent event)
         {
            fontstyle = style;
            setSampleFont();
         }
      }
      ActionListener listener = new MenuItemListener();
      item.addActionListener(listener);
      return item;
   }

   /**
      Sets the font of the text sample.
   */
   public void setSampleFont()
   {
      Font f = new Font(facename, fontstyle, fontsize);
      sampleField.setFont(f);
      sampleField.repaint();
   }
}
```

SELF CHECK

11. Why do `JMenu` objects not generate action events?

12. Why is the `name` parameter in the `createFaceItem` method declared as `final`?

How To 18.2 Implementing a Graphical User Interface (GUI)

A GUI program allows users to supply inputs and specify actions. The `textfield/Investment-Viewer3` program has only one input and one action. More sophisticated programs have more interesting user interactions, but the basic principles are the same.

Step 1 Enumerate the actions that your program needs to carry out.

For example, the investment viewer has a single action, to add interest. Other programs may have different actions, perhaps for making deposits, inserting coins, and so on.

Step 2 For each action, enumerate the inputs that you need.

For example, the investment viewer has a single input: the interest rate. Other programs may have different inputs, such as amounts of money, product quantities, and so on.

Step 3 For each action, enumerate the outputs that you need to show.

The investment viewer has a single output: the current balance. Other programs may show different quantities, messages, and so on.

Step 4 Supply the user-interface components.

Use buttons or menus for actions, text components for inputs, choice components to present finite sets of choices, and labels for outputs. Implement your own components to produce graphical output, such as charts or drawings.

Step 5 Use layout managers for layout.

Add the required components to a frame, using the techniques of How To 18.1.

Step 6 Supply event handler classes.

For each button, choice component, or menu item, you need to add an object of a listener class. The listener classes must implement the `ActionListener` interface. Supply a class for each action (or group of related actions), and put the instructions for the action in the `actionPerformed` method.

```
class Button1Listener implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      // button1 action goes here
      . . .
   }
}
```

Remember to declare any local variables accessed by the listener methods as `final`.

Step 7 Make listener objects and attach them to the event sources.

For action events, the event source is a button or other user-interface component, or a timer. You need to add a listener object to each event source, like this:

```
ActionListener listener1 = new Button1Listener();
button1.addActionListener(listener1);
```

18.6 Exploring the Swing Documentation

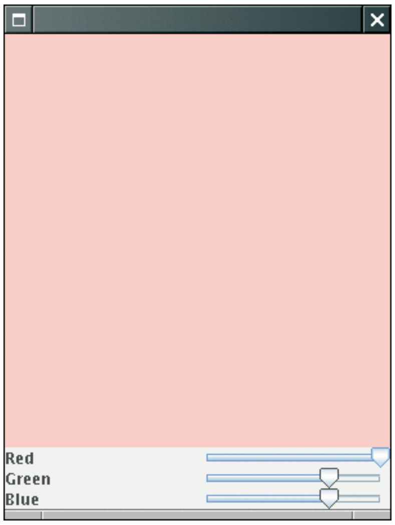
You should learn to navigate the API documentation to find out more about user-interface components.

In the preceding sections, you saw the basic properties of the most common user-interface components. We purposefully omitted many options and variations to simplify the discussion. You can go a long way by using only the simplest properties of these components. If you want to implement a more sophisticated effect, you can look inside the Swing documentation. You will probably find the documentation quite intimidating at first glance, though. The purpose of this section is to show you how you can use the documentation to your advantage without becoming overwhelmed.

As an example, consider a program for mixing colors by specifying the red, green, and blue values. How can you specify the colors? Of course, you could supply three text fields, but sliders would be more convenient for users of your program (see Figure 13).

The Swing user-interface toolkit has a large set of user-interface components. How do you know if there is a slider? You can buy a book that illustrates all Swing components. Or you can run the sample application included in the Java Development Kit that shows off all Swing components (see Figure 14). Or you can look at the names of all of the classes that start with `J` and decide that `JSlider` may be a good candidate.

Figure 13
A Color Viewer



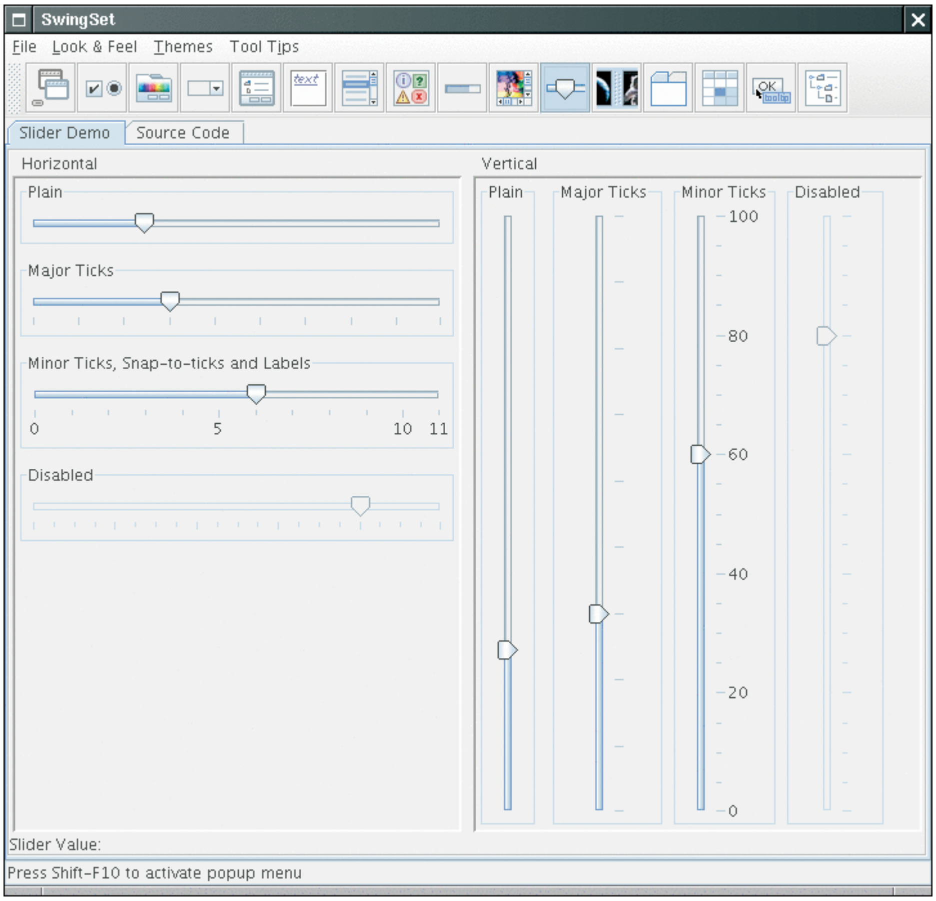


Figure 14 The SwingSet Demo

Next, you need to ask yourself a few questions:

- How do I construct a `JSlider`?
- How can I get notified when the user has moved it?
- How can I tell to which value the user has set it?

When you look at the documentation of the `JSlider` class, you will probably not be happy. There are over 50 methods in the `JSlider` class and over 250 inherited methods, and some of the method descriptions look downright scary, such as the one in Figure 15. Apparently some folks out there are concerned about the `valueIsAdjusting` property, whatever that may be, and the designers of this class felt it necessary to supply a method to tweak that property. Until you too feel that need, your best bet is to ignore this method. As the author of an introductory book, it pains me to tell you to ignore certain facts. But the truth of the matter is that the Java library is so large and complex that nobody understands it in its entirety, not even the designers of Java themselves. You need to develop the ability to separate fundamental concepts from ephemeral minutiae. For example, it is important that you understand the concept of event handling. Once you understand the concept, you can ask the question, “What event does the slider send when the user moves it?” But it is not important that you memorize how to set tick marks or that you know how to implement a slider with a custom look and feel.

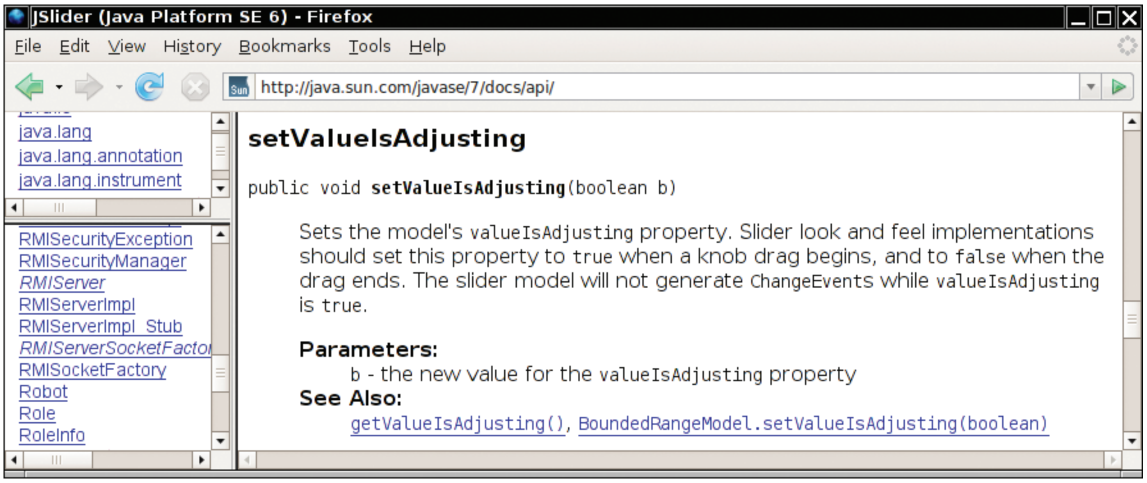
Let us go back to our fundamental questions. In Java 6, there are six constructors for the `JSlider` class. You want to learn about one or two of them. You must strike a balance somewhere between the trivial and the bizarre. Consider

`public JSlider()`
Creates a horizontal slider with the range 0 to 100 and an initial value of 50.

Maybe that is good enough for now, but what if you want another range or initial value? It seems too limited.

On the other side of the spectrum, there is

`public JSlider(BoundedRangeModel brm)`
Creates a horizontal slider using the specified `BoundedRangeModel`.



setValueIsAdjusting

`public void setValueIsAdjusting(boolean b)`

Sets the model's `valueIsAdjusting` property. Slider look and feel implementations should set this property to `true` when a knob drag begins, and to `false` when the drag ends. The slider model will not generate `ChangeEvents` while `valueIsAdjusting` is `true`.

Parameters:
`b` - the new value for the `valueIsAdjusting` property
See Also:
`getValueIsAdjusting()`, `BoundedRangeModel.setValueIsAdjusting(boolean)`

Figure 15 A Mysterious Method Description from the API Documentation

Whoa! What is that? You can click on the `BoundedRangeModel` link to get a long explanation of this class. This appears to be some internal mechanism for the Swing implementors. Let's try to avoid this constructor if we can. Looking further, we find

```
public JSlider(int min, int max, int value)
```
Creates a horizontal slider using the specified `min`, `max`, and `value`.

This sounds general enough to be useful and simple enough to be usable. You might want to stash away the fact that you can have vertical sliders as well.

Next, you want to know what events a slider generates. There is no `addActionListener` method. That makes sense. Adjusting a slider seems different from clicking a button, and Swing uses a different event type for these events. There is a method

```
public void addChangeListener(ChangeListener l)
```

Click on the `ChangeListener` link to find out more about this interface. It has a single method

```
void stateChanged(ChangeEvent e)
```

Apparently, that method is called whenever the user moves the slider. What is a `ChangeEvent`? Once again, click on the link, to find out that this event class has *no* methods of its own, but it inherits the `getSource` method from its superclass `EventObject`. The `getSource` method tells us which component generated this event, but we don't need that information—we know that the event came from the slider.

Now let's make a plan: Add a change event listener to each slider. When the slider is changed, the `stateChanged` method is called. Find out the new value of the slider. Recompute the color value and repaint the color panel. That way, the color panel is continually repainted as the user moves one of the sliders.

To compute the color value, you will still need to get the current value of the slider. Look at all the methods that start with `get`. Sure enough, you find

```
public int getValue()
```
Returns the slider's value.

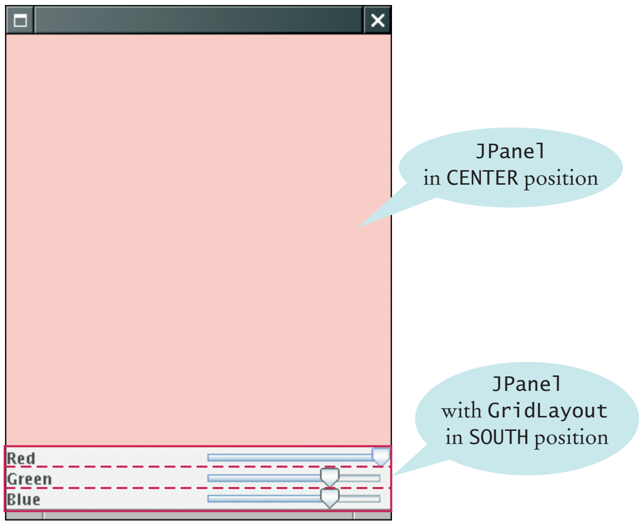
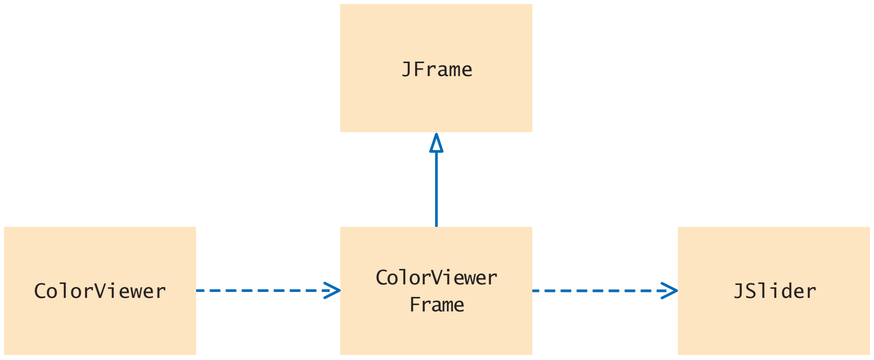

JPanel in CENTER position
JPanel with GridLayout in SOUTH position
Red
Green
Blue

Figure 16
The Components of the `ColorViewerFrame`

Figure 17 Classes of the Color Viewer Program



Now you know everything you need to write the program. The program uses one new Swing component and one event listener of a new type. After having mastered the basics, you may want to explore the capabilities of the component further, for example by adding tick marks—see Exercise P18.17.

Figure 16 shows how the components are arranged in the frame. Figure 17 shows the UML diagram.

ch18/slider/ColorViewer.java

```java
import javax.swing.JFrame;

public class ColorViewer
{
   public static void main(String[] args)
   {
      ColorViewerFrame frame = new ColorViewerFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

ch18/slider/ColorViewerFrame.java

```java
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.GridLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JSlider;
import javax.swing.event.ChangeListener;
import javax.swing.event.ChangeEvent;

public class ColorViewerFrame extends JFrame
{
   private static final int FRAME_WIDTH = 300;
   private static final int FRAME_HEIGHT = 400;

   private JPanel colorPanel;
   private JSlider redSlider;
   private JSlider greenSlider;
   private JSlider blueSlider;

   public ColorViewerFrame()
   {
      colorPanel = new JPanel();

      add(colorPanel, BorderLayout.CENTER);
      createControlPanel();
      setSampleColor();
      setSize(FRAME_WIDTH, FRAME_HEIGHT);
   }

   public void createControlPanel()
   {
      class ColorListener implements ChangeListener
      {
         public void stateChanged(ChangeEvent event)
         {
            setSampleColor();
         }
      }

      ChangeListener listener = new ColorListener();

      redSlider = new JSlider(0, 255, 255);
      redSlider.addChangeListener(listener);

      greenSlider = new JSlider(0, 255, 175);
      greenSlider.addChangeListener(listener);

      blueSlider = new JSlider(0, 255, 175);
      blueSlider.addChangeListener(listener);

      JPanel controlPanel = new JPanel();
      controlPanel.setLayout(new GridLayout(3, 2));

      controlPanel.add(new JLabel("Red"));
      controlPanel.add(redSlider);

      controlPanel.add(new JLabel("Green"));
      controlPanel.add(greenSlider);

      controlPanel.add(new JLabel("Blue"));
      controlPanel.add(blueSlider);

      add(controlPanel, BorderLayout.SOUTH);
   }

   /**
      Reads the slider values and sets the panel to
      the selected color.
   */
   public void setSampleColor()
   {
      // Read slider values

      int red = redSlider.getValue();
      int green = greenSlider.getValue();
      int blue = blueSlider.getValue();

```

```
        // Set panel background to selected color

        colorPanel.setBackground(new Color(red, green, blue));
        colorPanel.repaint();
    }
}
```

SELF CHECK

13. Suppose you want to allow users to pick a color from a color dialog box. Which class would you use? Look in the API documentation.

14. Why does a slider emit change events and not action events?

Summary of Learning Objectives

Use text fields for reading text input.

- Use JTextField components to provide space for user input. Place a JLabel next to each text field.

Use text areas for reading and displaying multi-line text.

- Use a JTextArea to show multiple lines of text.
- You can add scroll bars to any component with a JScrollPane.

Learn how to arrange multiple components in a container.

- User-interface components are arranged by placing them inside containers. Containers can be placed inside larger containers.
- Each container has a layout manager that directs the arrangement of its components.
- When adding a component to a container with the border layout, specify the NORTH, EAST, SOUTH, WEST, or CENTER position.
- The content pane of a frame has a border layout by default. A panel has a flow layout by default.

Select among the Swing components for presenting choices to the user.

- For a small set of mutually exclusive choices, use a group of radio buttons or a combo box.
- Add radio buttons into a ButtonGroup so that only one button in the group is on at any time.
- You can place a border around a panel to group its contents visually.
- For a binary choice, use a check box.
- For a large set of choices, use a combo box.
- Radio buttons, check boxes, and combo boxes generate action events, just as buttons do.

Implement menus in a Swing program.

- A frame contains a menu bar. The menu bar contains menus. A menu contains submenus and menu items.
- Menu items generate action events.
- You should learn to navigate the API documentation to find out more about user-interface components.

Classes, Objects, and Methods Introduced in this Chapter

```
java.awt.BorderLayout
   CENTER
   EAST
   NORTH
   SOUTH
   WEST
java.awt.Container
   setLayout
java.awt.FlowLayout
java.awt.Font
java.awt.GridLayout
javax.swing.AbstractButton
   isSelected
   setSelected
javax.swing.ButtonGroup
   add
javax.swing.ImageIcon
javax.swing.JCheckBox
javax.swing.JComboBox
   addItem
   getSelectedItem
   isEditable
   setEditable
javax.swing.JComponent
   setBorder
   setFont
javax.swing.JFrame
   setJMenuBar
javax.swing.JMenu
   add
javax.swing.JMenuBar
   add
javax.swing.JMenuItem
javax.swing.JRadioButton
javax.swing.JScrollPane
javax.swing.JSlider
   addChangeListener
   getValue
javax.swing.JTextArea
   append
javax.swing.JTextField
javax.swing.border.EtchedBorder
javax.swing.border.TitledBorder
javax.swing.event.ChangeEvent
javax.swing.event.ChangeListener
   stateChanged
javax.swing.text.JTextComponent
   getText
   isEditable
   setEditable
   setText
```

Media Resources

www.wiley.com/college/horstmann

- Lab Exercises
- ⊕ Practice Quiz
- ⊕ Code Completion Exercises

Review Exercises

★G **R18.1** What is the difference between a label, a text field, and a text area?

★★G **R18.2** Name a method that is declared in `JTextArea`, a method that `JTextArea` inherits from `JTextComponent`, and a method that `JTextArea` inherits from `JComponent`.

★G **R18.3** Can you use a flow layout for the components in a frame? If yes, how?

★G **R18.4** What is the advantage of a layout manager over telling the container “place this component at position (x, y)”?

★★G **R18.5** What happens when you place a single button into the `CENTER` area of a container that uses a border layout? Try it out, by writing a small sample program, if you aren’t sure of the answer.

★★G **R18.6** What happens if you place multiple buttons directly into the `SOUTH` area, without using a panel? Try it out, by writing a small sample program, if you aren’t sure of the answer.

★★G **R18.7** What happens when you add a button to a container that uses a border layout and omit the position? Try it out and explain.

★★G **R18.8** What happens when you try to add a button to another button? Try it out and explain.

★★G **R18.9** The `ColorViewerFrame` uses a grid layout manager. Explain a drawback of the grid that is apparent from Figure 16 on page 767. What could you do to overcome this drawback?

★★★G **R18.10** What is the difference between the grid layout and the grid bag layout?

★★★G **R18.11** Can you add icons to check boxes, radio buttons, and combo boxes? Browse the Java documentation to find out. Then write a small test program to verify your findings.

★G **R18.12** What is the difference between radio buttons and check boxes?

★G **R18.13** Why do you need a button group for radio buttons but not for check boxes?

★G **R18.14** What is the difference between a menu bar, a menu, and a menu item?

★G **R18.15** When browsing through the Java documentation for more information about sliders, we ignored the `JSlider` constructor with no parameters. Why? Would it have worked in our sample program?

★G **R18.16** How do you construct a vertical slider? Consult the Swing documentation for an answer.

★★G **R18.17** Why doesn’t a `JComboBox` send out change events?

★★★G **R18.18** What component would you use to show a set of choices, just as in a combo box, but so that several items are visible at the same time? Run the Swing demo application or look at a book with Swing example programs to find the answer.

★★G **R18.19** How many Swing user-interface components are there? Look at the Java documentation to get an approximate answer.

★★G **R18.20** How many methods does the `JProgressBar` component have? Be sure to count inherited methods. Look at the Java documentation.

Programming Exercises

★G **P18.1** Write a graphical application front end for a bank account class. Supply text fields and buttons for depositing and withdrawing money, and for displaying the current balance in a label.

★G **P18.2** Write a graphical application front end for an `Earthquake` class. Supply a text field and button for entering the strength of the earthquake. Display the earthquake description in a label.

★G **P18.3** Write a graphical application front end for a `DataSet` class. Supply text fields and buttons for adding floating-point values, and display the current minimum, maximum, and average in a label.

★G **P18.4** Write an application with three labeled text fields, one each for the initial amount of a savings account, the annual interest rate, and the number of years. Add a button "Calculate" and a read-only text area to display the result, namely, the balance of the savings account after the end of each year.

★★G **P18.5** In the application from Exercise P18.4, replace the text area with a bar chart that shows the balance after the end of each year.

★★★G **P18.6** Write a program that contains a text field, a button "Add Value", and a component that draws a bar chart of the numbers that a user typed into the text field.

★★★G **P18.7** Write a program that draws a clock face with a time that the user enters in two text fields (one for the hours, one for the minutes).

Hint: You need to determine the angles of the hour hand and the minute hand. The angle of the minute hand is easy: The minute hand travels 360 degrees in 60 minutes. The angle of the hour hand is harder; it travels 360 degrees in 12×60 *minutes*.

★G **P18.8** Write an application with three buttons labeled "Red", "Green", and "Blue" that changes the background color of a panel in the center of the frame to red, green, or blue.

★★G **P18.9** Add icons to the buttons of Exercise P18.8.

★★G **P18.10** Write a calculator application. Use a grid layout to arrange buttons for the digits and for the $+ - \times \div$ operations. Add a text field to display the result.

★G **P18.11** Write an application with three radio buttons labeled "Red", "Green", and "Blue" that changes the background color of a panel in the center of the frame to red, green, or blue.

★G **P18.12** Write an application with three check boxes labeled “Red”, “Green”, and “Blue” that adds a red, green, or blue component to the background color of a panel in the center of the frame. This application can display a total of eight color combinations.

★G **P18.13** Write an application with a combo box containing three items labeled “Red”, “Green”, and “Blue” that changes the background color of a panel in the center of the frame to red, green, or blue.

★G **P18.14** Write an application with a Color menu and menu items labeled “Red”, “Green”, and “Blue” that changes the background color of a panel in the center of the frame to red, green, or blue.

★G **P18.15** Write a program that displays a number of rectangles at random positions. Supply buttons “Fewer” and “More” that generate fewer or more random rectangles. Each time the user clicks on “Fewer”, the count should be halved. Each time the user clicks on “More”, the count should be doubled.

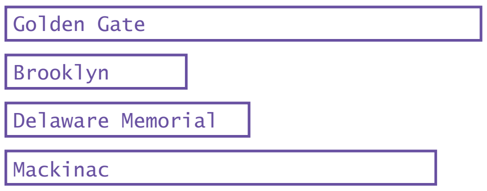
★★G **P18.16** Modify the program of Exercise P18.15 to replace the buttons with a slider to generate fewer or more random rectangles.

★★G **P18.17** In the slider test program, add a set of tick marks to each slider that show the exact slider position.

★★★G **P18.18** Enhance the font viewer program to allow the user to select different fonts. Research the API documentation to find out how to find the available fonts on the user’s system.

Programming Projects

Project 18.1 Write a program that lets users design charts such as the following:



Use appropriate components to ask for the length, label, and color, then apply them when the user clicks an “Add Item” button. Allow the user to switch between bar charts and pie charts.

Project 18.2 Write a program that displays a scrolling message in a panel. Use a timer for the scrolling effect. In the timer’s action listener, move the starting position of the message and repaint. When the message has left the window, reset the starting position to the other corner. Provide a user interface to customize the message text, font, foreground and background colors, and the scrolling speed and direction.

Answers to Self-Check Questions

1. Then the text field is not labeled, and the user will not know its purpose.
2. `Integer.parseInt(textField.getText())`
3. A text field holds a single line of text; a text area holds multiple lines.
4. The text area is intended to display the program output. It does not collect user input.
5. Don't construct a `JScrollPane` but add the `resultArea` object directly to the frame.
6. First add them to a panel, then add the panel to the north end of a frame.
7. Place them inside a panel with a `GridLayout` that has three rows and one column.
8. If you have many options, a set of radio buttons takes up a large area. A combo box can show many options without using up much space. But the user cannot see the options as easily.
9. When any of the component settings is changed, the program simply queries all of them and updates the label.
10. To keep it from growing too large. It would have grown to the same width and height as the two panels below it.
11. When you open a menu, you have not yet made a selection. Only `JMenuItem` objects correspond to selections.
12. The parameter variable is accessed in a method of an inner class.
13. `JColorChooser`.
14. Action events describe one-time changes, such as button clicks. Change events describe continuous changes.