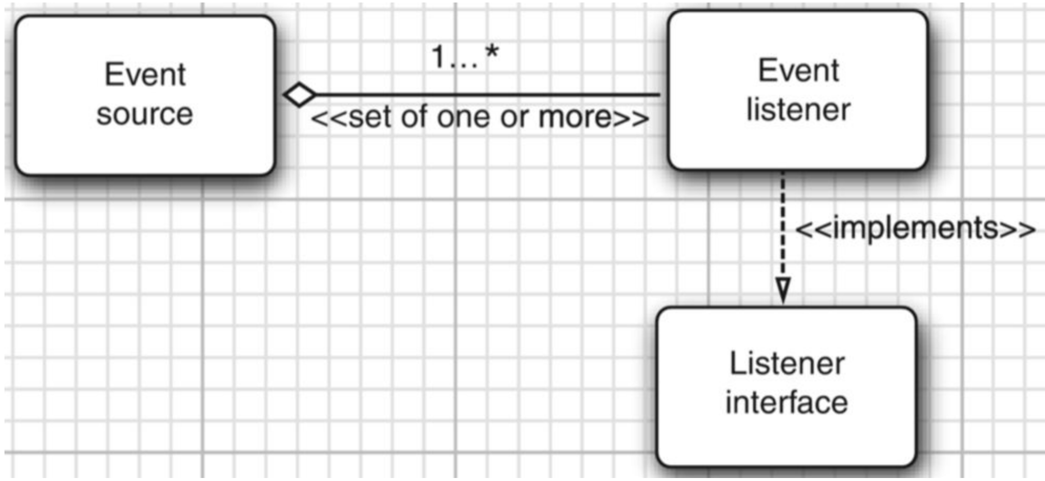
Chapter 7
Programming Graphical User Interface (GUI)

EVENT-DRIVEN PROGRAMMING

1 Basics of Event Handling

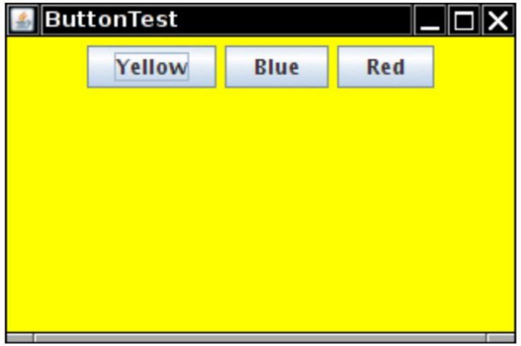
- An event source is an object that
  - register listener objects
  - sends out event objects to all registered listeners when that event occurs.
- The listener objects decide their reaction to the event.



1.1 Basics of Event Handling

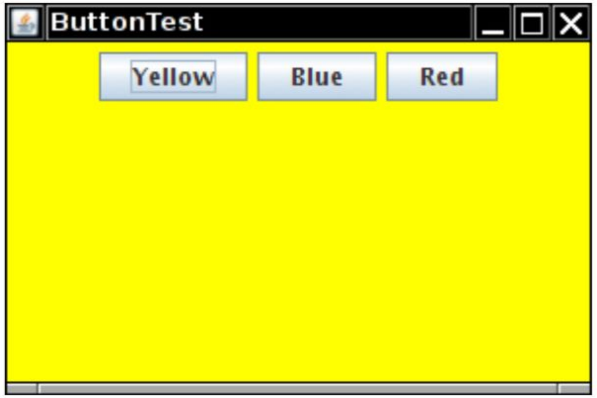
```
import java.awt.*;import java.awt.event.*;import javax.swing.*;

public class ButtonFrame extends JFrame{
  private JPanel buttonPanel;
  private static final int DEFAULT_WIDTH = 300;
  private static final int DEFAULT_HEIGHT = 200;

  public ButtonFrame(){
    setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

    JButton yellowButton = new JButton("Yellow");
    JButton blueButton = new JButton("Blue");
    JButton redButton = new JButton("Red");

    buttonPanel = new JPanel();
    buttonPanel.add(yellowButton); buttonPanel.add(blueButton);
    buttonPanel.add(redButton);
    add(buttonPanel);  // add panel to frame

    // create button actions
    ColorAction yellowAction = new ColorAction(Color.YELLOW);
    ColorAction blueAction = new ColorAction(Color.BLUE);
    ColorAction redAction = new ColorAction(Color.RED);

    // associate actions with buttons
    yellowButton.addActionListener(yellowAction);
    blueButton.addActionListener(blueAction);
    redButton.addActionListener(redAction);
  }
```

```
  /**
   * An action listener that sets the panel's background color.
   */
  private class ColorAction implements ActionListener
  {
    private Color backgroundColor;

    public ColorAction(Color c)
    {
      backgroundColor = c;
    }

    public void actionPerformed(ActionEvent event)
    {
      buttonPanel.setBackground(backgroundColor);
    }
  }
}
```



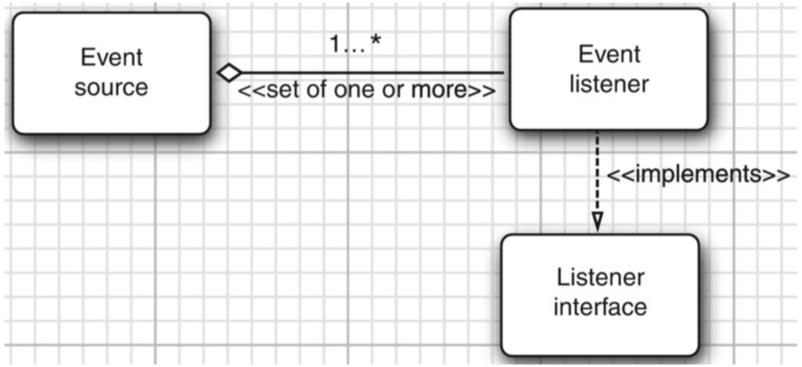
1.2 Specifying Listeners Concisely

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ButtonFrameLambda extends JFrame{
  private JPanel buttonPanel;
  private static final int DEFAULT_WIDTH = 300;
  private static final int DEFAULT_HEIGHT = 200;

  public ButtonFrameLambda(){
    setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

    JButton yellowButton = new JButton("Yellow");
    JButton blueButton = new JButton("Blue");
    JButton redButton = new JButton("Red");

    buttonPanel = new JPanel();
    buttonPanel.add(yellowButton);
    buttonPanel.add(blueButton);
    buttonPanel.add(redButton);

    add(buttonPanel);

    yellowButton.addActionListener(event -> buttonPanel.setBackground(Color.YELLOW));
    blueButton.addActionListener(event -> buttonPanel.setBackground(Color.BLUE));
    redButton.addActionListener(event -> buttonPanel.setBackground(Color.RED));
  }
}
```



1.3 Adapter Classes

- When the user tries to close a window, the JFrame object is the source of a WindowEvent.
- To catch that event, you must have an appropriate listener object and add it to the frame's list of window listeners.



```
WindowListener listener = . . .;
frame.addWindowListener(listener);

public interface WindowListener {
        void windowOpened(WindowEvent e);
        // only need one but override all methods in listener
        void windowClosing(WindowEvent e);
        void windowClosed(WindowEvent e);
        void windowIconified(WindowEvent e);
        void windowDeiconified(WindowEvent e);
        void windowActivated(WindowEvent e);
        void windowDeactivated(WindowEvent e);
}
```

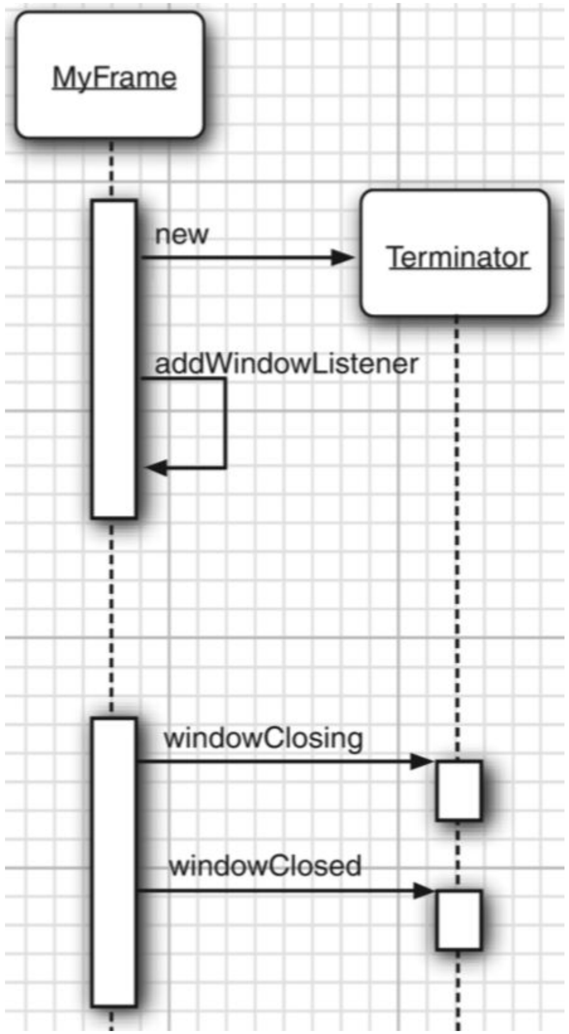
1.3 Adapter Classes

- each interface that have more than one method comes with a companion adapter class that implements all the methods in the interface but does nothing with them.

```
public interface WindowListener {
        void windowOpened(WindowEvent e);
        // only need one but override all methods in listener
        void windowClosing(WindowEvent e);
        void windowClosed(WindowEvent e);
        void windowIconified(WindowEvent e);
        void windowDeiconified(WindowEvent e);
        void windowActivated(WindowEvent e);
        void windowDeactivated(WindowEvent e);
}
class WindowAdapter implements WindowListener{
        void windowOpened(WindowEvent e){}
        void windowClosing(WindowEvent e);
        void windowClosed(WindowEvent e);
        void windowIconified(WindowEvent e);
        void windowDeiconified(WindowEvent e);
        void windowActivated(WindowEvent e);
        void windowDeactivated(WindowEvent e);
}
```

1.3 Adapter Classes

```
class Terminator extends WindowAdapter {
        public void windowClosing(WindowEvent e) {
                System.exit(0);
        }
}

// register listener to frame
WindowListener listener = new Terminator();
frame.addWindowListener(listener);
```



1.4 Actions

An action is an object that encapsulates

- A description of the command
- Parameters that are necessary to carry out the command

What you carry out the same action in response to a button, a menu item, or a keystroke:

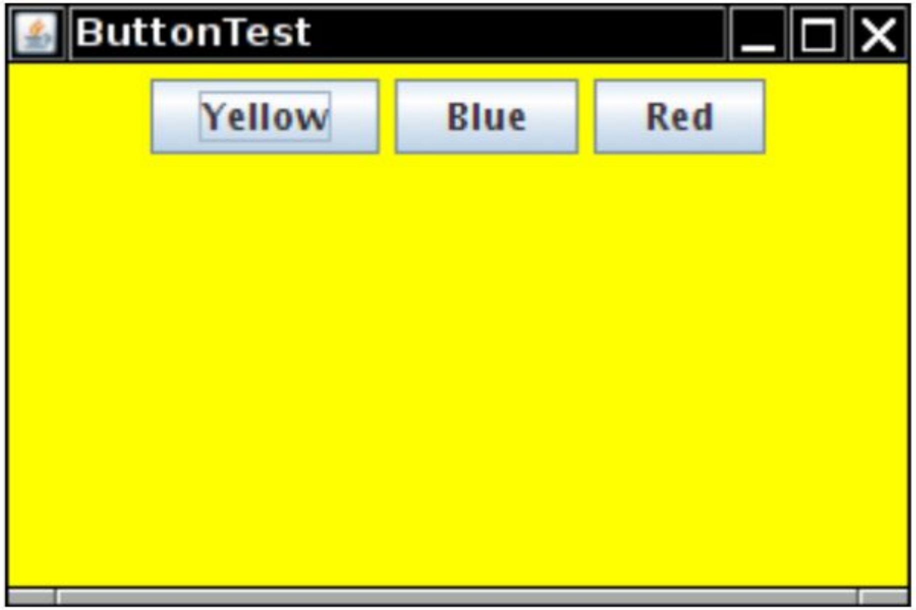
- Implement a class that extends the AbstractAction class.
- Construct an object of the action class.
- Construct a button or menu item from the action object.

  The constructor will read the label text and icon from the action object.

1.4 Actions

- The Action interface has the following methods:
  - void actionPerformed(ActionEvent event)
  - void setEnabled(boolean b)
  - boolean isEnabled()
  - void putValue(String key, Object value)
  - Object getValue(String key)
  - void addPropertyChangeListener(PropertyChangeListener listener)
  - void removePropertyChangeListener(PropertyChangeListener listener)
- Any class implementing this **Action interface** must implement the seven methods we just discussed.
- AbstractAction class that implements all methods except for actionPerformed.

1.4 Actions



1.4 Actions

```
import java.awt.*;
import javax.swing.*;

public class ActionFrame extends JFrame{
   private JPanel buttonPanel;
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 200;

   public ActionFrame(){
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      buttonPanel = new JPanel();

      // define actions
      Action yellowAction = new ColorAction("Yellow",
               new ImageIcon("yellow-ball.gif"), Color.YELLOW);
      Action blueAction = new ColorAction("Blue",
               new ImageIcon("blue-ball.gif"),  Color.BLUE);
      Action redAction = new ColorAction("Red",
               new ImageIcon("red-ball.gif"), Color.RED);

      // add buttons for these actions
      buttonPanel.add(new JButton(yellowAction));
      buttonPanel.add(new JButton(blueAction));
      buttonPanel.add(new JButton(redAction));

      add(buttonPanel);
   }

public class ColorAction extends AbstractAction{
  public ColorAction(String name, Icon icon, Color c){
    putValue(Action.NAME, name);
    putValue(Action.SMALL_ICON, icon);
    putValue(Action.SHORT_DESCRIPTION,
      "Set panel color to " + name.toLowerCase());
    putValue("color", c);
  }
  public void actionPerformed(ActionEvent event){
    Color c = (Color) getValue("color");
    buttonPanel.setBackground(c);
  }  } }
```

SWING

1 Introducing Swing

- GUI programming:
  - Abstract Window Toolkit (AWT)
  - Swing: built on the top of AWT
  - JavaFX
- The resulting program could run on any of these platforms, with the “look-and-feel” of the target platform.
  - Swing has a rich and convenient set of user interface elements.
  - Swing has few dependencies on the underlying platform => less prone to platform-specific bugs.
  - Swing gives a consistent user experience across platforms.

2 Creating a Frame

- A top-level window is called a frame in Java: JFrame
- A window that is not contained inside another window

A frame is hidden when the user closes it, but the program does not terminate.

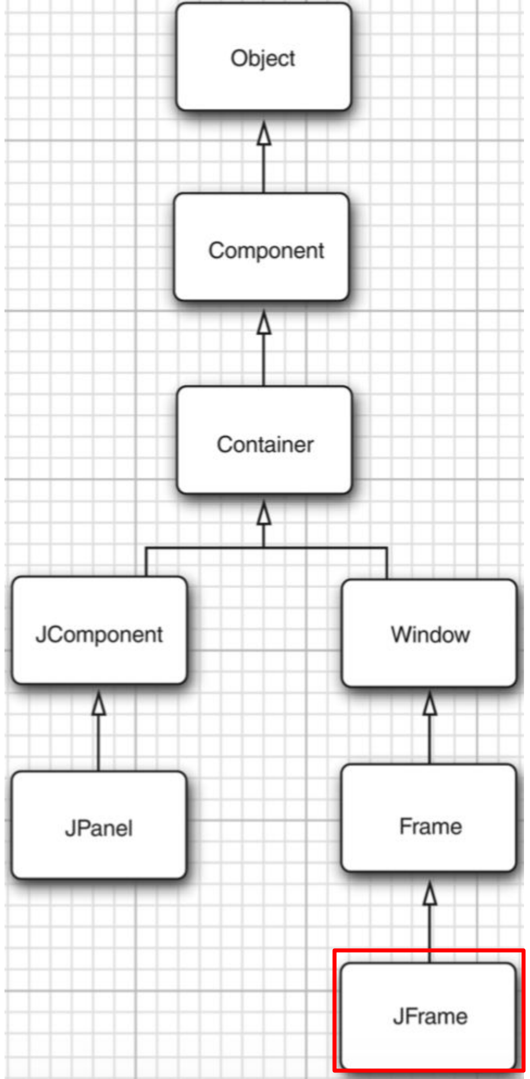
Frames start their life invisible.

```
import java.awt.*;
import javax.swing.*;

class SimpleFrame extends JFrame {
  private static final int DEFAULT_WIDTH = 300;
  private static final int DEFAULT_HEIGHT = 200;

  public SimpleFrame(){
    setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
  }
}

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

3 Positioning a Frame

Most of the methods for working with the size and position of a frame come from the various superclasses of JFrame;

- setLocation, setBounds
- setIconImage
- setTitle
- setResizable



3 Positioning a Frame

3.1 Frame Properties

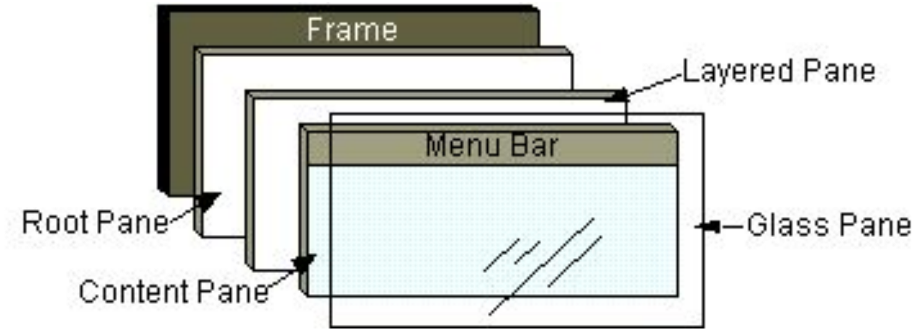
- get the resolution of the user's screen
- write code that resizes the frames accordingly

```
class SizedFrame extends JFrame
{
   public SizedFrame()
   {
      // get screen dimensions

      Toolkit kit = Toolkit.getDefaultToolkit();
      Dimension screenSize = kit.getScreenSize();
      int screenHeight = screenSize.height;
      int screenWidth = screenSize.width;

      // set frame width, height and let platform pick screen location

      setSize(screenWidth / 2, screenHeight / 2);
      setLocationByPlatform(true);
   }
}
```

4 Displaying Information in a Component

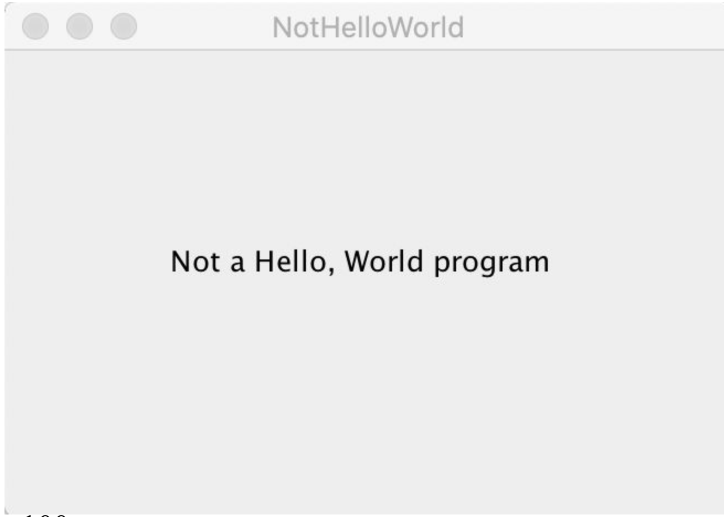
- frames are really designed to be containers for components
- four panes are layered in a JFrame.
  - the root pane: manages other pane
  - glass pane: hidden, by default, like a sheet of glass over all the other parts of the root pane
  - layered pane: serves to position its contents, which consist of the content pane and the optional menu bar.
  - content pane: the container of the root pane's visible components, excluding the menu bar.
  - optional menu bar: the home for the root pane's container's menus.
- designing a frame, you add components into the content pane

```
Container contentPane = frame.getContentPane();
Component c = . . .;
contentPane.add(c);
```



4 Displaying Information in a Component

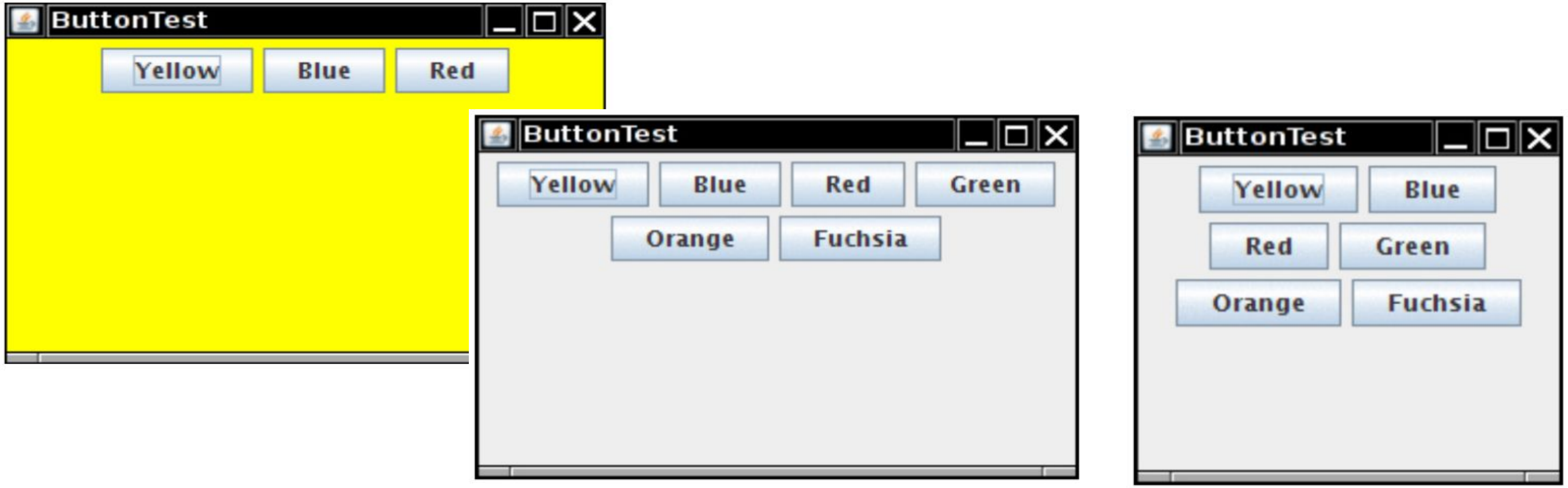
```
public class NotHelloWorld {
        public static void main(String[] args) {
                JFrame frame = new NotHelloWorldFrame();
                frame.setTitle("NotHelloWorld");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);

        }
}
class NotHelloWorldFrame extends JFrame {
        public NotHelloWorldFrame() {
                add(new NotHelloWorldComponent());
                pack();
        }
}
class NotHelloWorldComponent extends JComponent {
        public static final int MESSAGE_X = 75; public static final int MESSAGE_Y = 100;
        private static final int DEFAULT_WIDTH = 300; private static final int DEFAULT_HEIGHT = 200;
        public void paintComponent(Graphics g) {
                g.drawString("Not a Hello, World program", MESSAGE_X, MESSAGE_Y);
        }
        public Dimension getPreferredSize() {
                return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT);
        }
}
```



Not a Hello, World program

5 Introduction to Layout Management

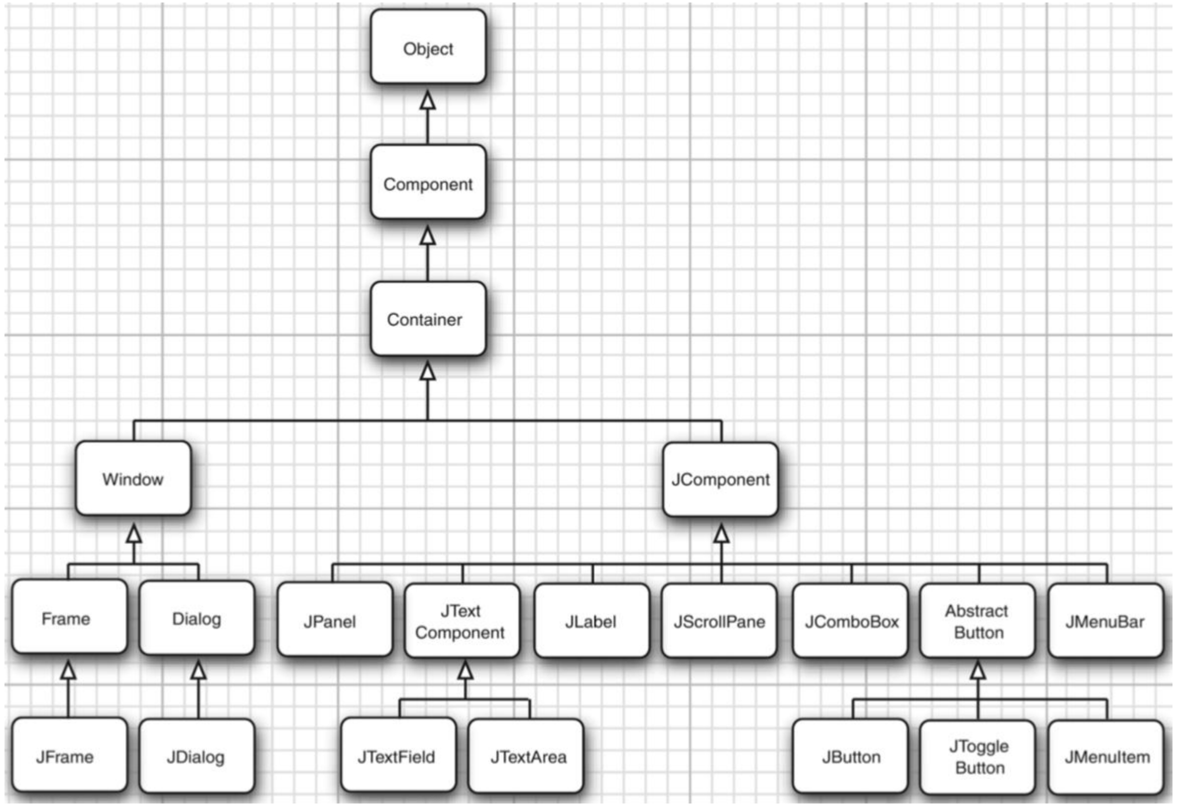
- Components (button, text field, …) are placed inside containers (panels)
- A layout manager determines the positions and sizes of components in a container
- Each container has a default layout manager, but you can always set your own



A panel with three buttons -> A panel with six buttons managed by a flow layout -> Changing the panel size rearranges the buttons automatically.

5 Introduction to Layout Management

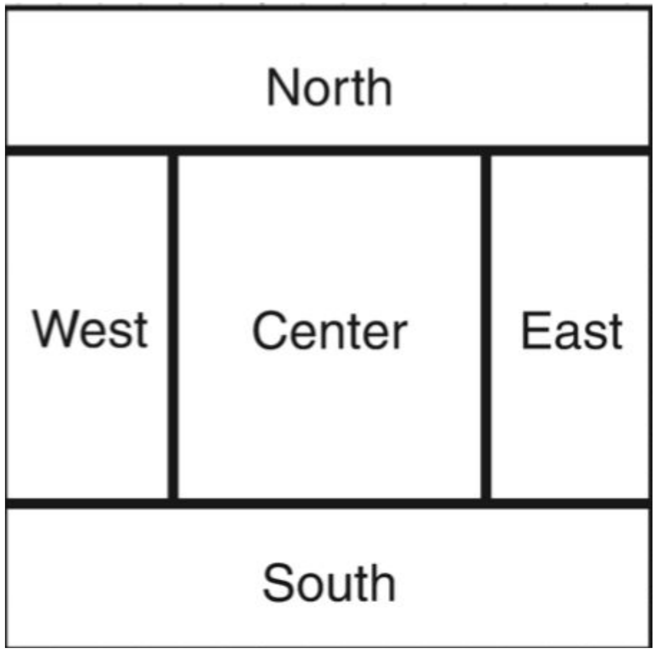
Inheritance hierarchy for the Component class



5 Introduction to Layout Management

5.1 Border Layout

- The border layout manager is the default layout manager of the content pane of every JFrame
- You can choose to place the component in the center, north, south, east, or west of the content pane



5 Introduction to Layout Management

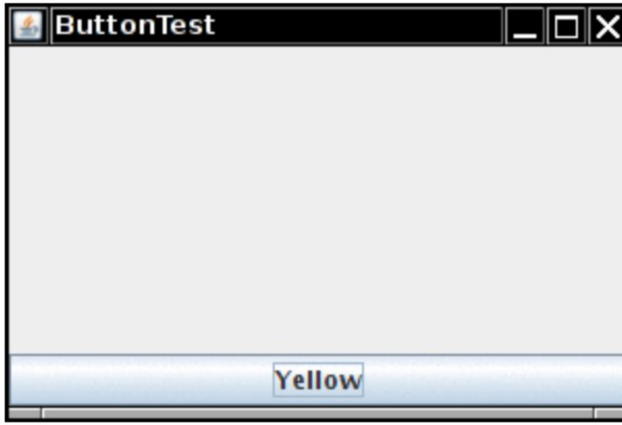
5.1 Border Layout

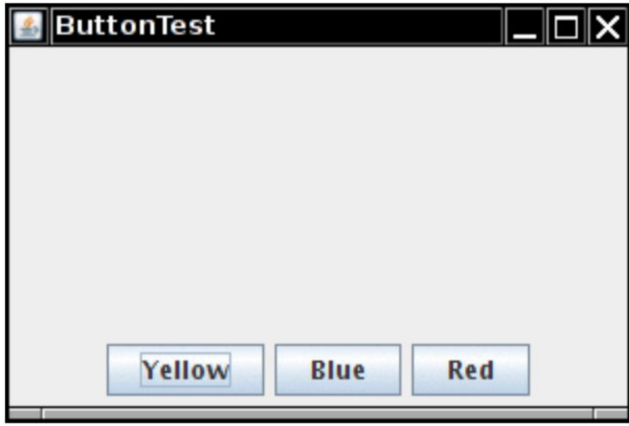
- The border layout grows all components to fill the available space.
- The flow layout leaves each component at its preferred size.

```
(1)    frame.add(yellowButton, BorderLayout.SOUTH);
```

```
(2)
frame.add(yellowButton, BorderLayout.SOUTH);
frame.add(blueButton, BorderLayout.SOUTH);
frame.add(redButton, BorderLayout.SOUTH);
```

```
(3)
JPanel panel = new JPanel(); // flow layout
panel.add(yellowButton);
panel.add(blueButton);
panel.add(redButton);
frame.add(panel, BorderLayout.SOUTH);
```



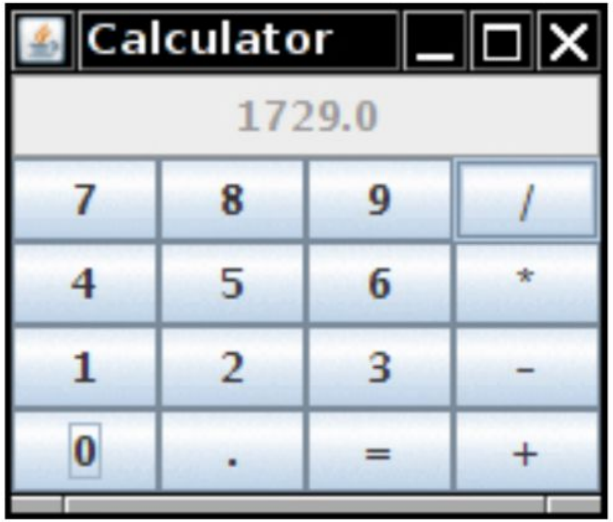


5 Introduction to Layout Management

5.2 Grid Layout

- The grid layout arranges all components in rows and columns like a spreadsheet.
- All components are given the same size.

```
panel.setLayout(new GridLayout(4, 4));
panel.add(new JButton("1"));
panel.add(new JButton("2"));
…
```



5.3 Text Input

5.3.1 Text Fields

- accepts one line of text
- add a text field to a window: add it to a panel or other container.

```
JPanel panel = new JPanel();
JTextField textField = new JTextField("Default input", 20);
panel.add(textField);

textField.setText("Hello!");  // change the text
String text = textField.getText().trim();   // get the text
```

5.3 Text Input

5.3.2 Labels and Labeling Components

Labels are components that hold text.

- have no decorations (for example, no boundaries).
- do not react to user input.
- often use to identify components
  - construct a JLabel component with the correct text.
  - place it close enough to the component you want to identify so that the user can see that the label identifies the correct component.

```
JLabel label = new JLabel("User name: ", SwingConstants.RIGHT);
or
JLabel label = new JLabel("User name: ", JLabel.RIGHT);
```

SwingConstants: specify alignment

5.3 Text Input

5.3.3 Password Fields

- accepts one line of text without showing the contents
- each typed character is represented by an echo character, typically an asterisk (*)

```
JPasswordField passwordField = new JPasswordField();
panel.add(passwordField);
frame.add(panel, BorderLayout.NORTH);
```

5.3 Text Input

5.3.4 Text Areas

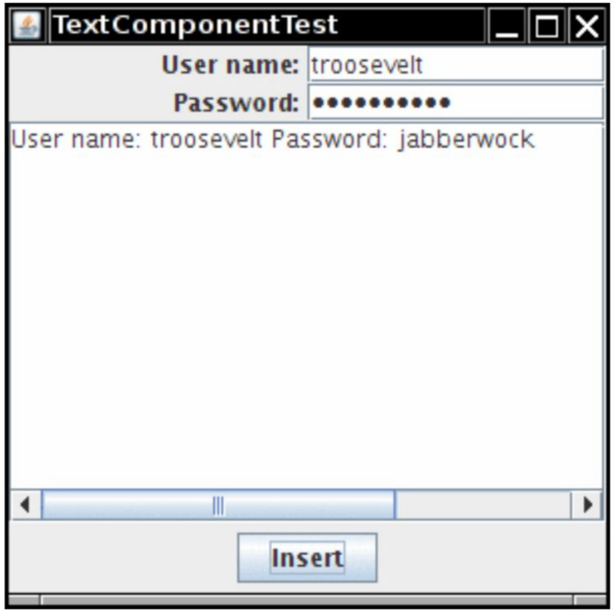
- accepts many lines of text

```
JTextArea textArea = new JTextArea(8, 40) // 8 lines of 40 columns each
panel.add(textArea);
frame.add(panel, BorderLayout.NORTH);
```

12.3.5 Scroll Panes

In Swing, a text area does not have scrollbars.
If want scrollbars, place the text area inside a scroll pane.

```
textArea = new JTextArea(8, 40);
JScrollPane scrollPane = new JScrollPane(textArea);
```



TextComponentTest

User name: troosevelt

Password: ●●●●●●●●●●●

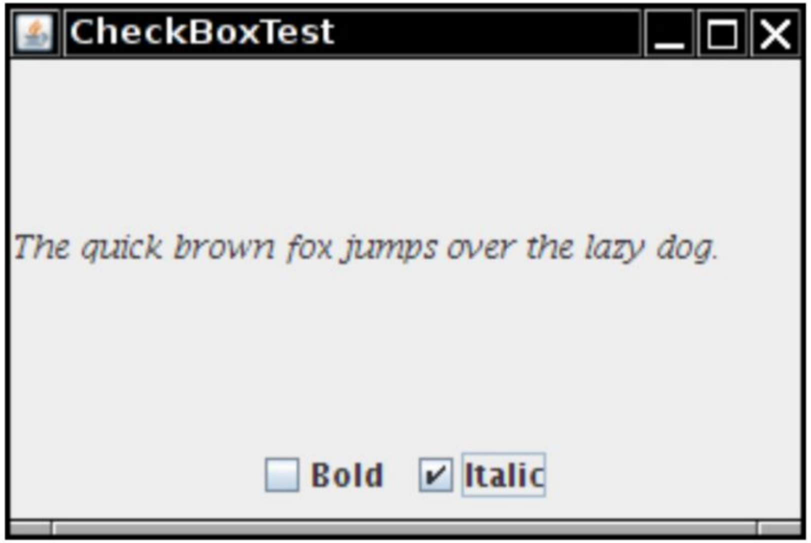
User name: troosevelt Password: jabberwock

```
s TextComponentFrame extends JFrame {
blic static final int TEXTAREA_ROWS = 8;
blic static final int TEXTAREA_COLUMNS = 20;

blic TextComponentFrame() {
        JTextField textField = new JTextField();
        JPasswordField passwordField = new JPasswordField();

        JPanel northPanel = new JPanel();
        northPanel.setLayout(new GridLayout(2, 2));
        northPanel.add(new JLabel("User name: ", SwingConstants.RIGHT));
        northPanel.add(textField);
        northPanel.add(new JLabel("Password: ", SwingConstants.RIGHT));
        northPanel.add(passwordField);
        add(northPanel, BorderLayout.NORTH);

        JTextArea textArea = new JTextArea(TEXTAREA_ROWS, TEXTAREA_COLUMNS);
        JScrollPane scrollPane = new JScrollPane(textArea);
        add(scrollPane, BorderLayout.CENTER);

        // add button to append text into the text area
        JPanel southPanel = new JPanel();
        JButton insertButton = new JButton("Insert");
        southPanel.add(insertButton);
        insertButton.addActionListener(event -> textArea.append(
                        "User name: " + textField.getText() + " Password: "
                                        + new String(passwordField.getPassword()) + "\n"));

        add(southPanel, BorderLayout.SOUTH);

        pack();
}
}
```

5.4 Choice Components

5.4.1 Checkboxes

If you want to collect just a "yes" or "no" input, use a checkbox component. Checkboxes automatically come with labels that identify them



```
public class CheckBoxFrame extends JFrame {
        private JLabel label;
        private JCheckBox bold;
        private JCheckBox italic;
        private static final int FONTSIZE = 24;
        public CheckBoxFrame() {
                // add the sample text label
                label = new JLabel("The quick brown fox jumps over the lazy dog.");
                label.setFont(new Font("Serif", Font.BOLD, FONTSIZE));
                add(label, BorderLayout.CENTER);

                // this listener sets the font attribute of
                // the label to the check box state
                ActionListener listener = event -> {
                        int mode = 0;
                        if (bold.isSelected())
                                mode += Font.BOLD;
                        if (italic.isSelected())
                                mode += Font.ITALIC;
                        label.setFont(new Font("Serif", mode, FONTSIZE));
                };

                // add the check boxes
                JPanel buttonPanel = new JPanel();

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

                buttonPanel.add(bold);

                italic = new JCheckBox("Italic");
                italic.addActionListener(listener);
                italic.setSelected(true);
                buttonPanel.add(italic);

                add(buttonPanel, BorderLayout.SOUTH);
                pack();
        }
}
```

5.4 Choice Components

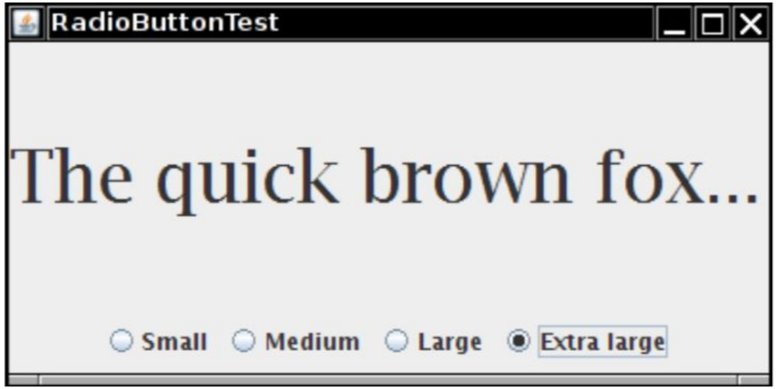
5.4.2 Radio Buttons

- allow the user to check only one of several boxes.
- when another box is checked, the previous box is automatically unchecked

```
ButtonGroup group = new ButtonGroup(); // controls only the behavior of the buttons
JRadioButton smallButton = new JRadioButton("Small", false);
group.add(smallButton);
JRadioButton mediumButton = new JRadioButton("Medium", true);
group.add(mediumButton);
```

5.4 Choice Components

```
public class RadioButtonFrame extends JFrame {
        private JPanel buttonPanel; private ButtonGroup group;
        private JLabel label; private static final int DEFAULT_SIZE = 36;
        public RadioButtonFrame() {
                // add the sample text label
                label = new JLabel("The quick brown fox jumps over the lazy dog.");
                label.setFont(new Font("Serif", Font.PLAIN, DEFAULT_SIZE));
                add(label, BorderLayout.CENTER);

                // add the radio buttons
                buttonPanel = new JPanel();
                group = new ButtonGroup();

                addRadioButton("Small", 8); addRadioButton("Medium", 12); addRadioButton("Large", 18); addRadioButton("Extra large", 36);

                add(buttonPanel, BorderLayout.SOUTH);
                pack();
        }
        public void addRadioButton(String name, int size) {
                boolean selected = size == DEFAULT_SIZE;
                JRadioButton button = new JRadioButton(name, selected);
                group.add(button);
                buttonPanel.add(button);

                // this listener sets the label font size
                ActionListener listener = event -> label.setFont(new Font("Serif", Font.PLAIN, size));
                button.addActionListener(listener);
        }
}
```



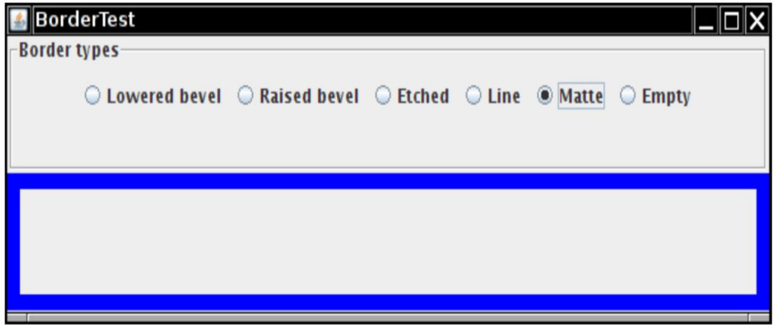
5.4 Choice Components

5.4.3 Borders

Place a border around a panel and fill that panel with other user interface elements, such as radio buttons

1. create a border: ***BorderFactory***
2. add a title to the border by passing it to ***BorderFactory.createTitledBorder***.
3. add the resulting border to the component by calling the ***setBorder*** method of the JComponent class.

5.4 Choice Components



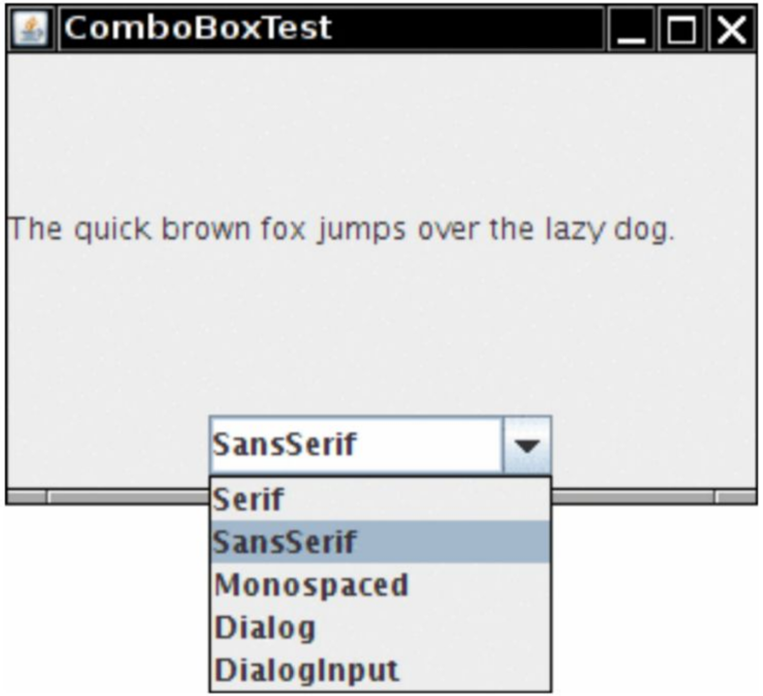
```
public class BorderFrame extends JFrame {
    private JPanel demoPanel;
    private JPanel buttonPanel;
    private ButtonGroup group;
    public BorderFrame() {
            demoPanel = new JPanel();
            buttonPanel = new JPanel();
            group = new ButtonGroup();

            addRadioButton("Lowered bevel", BorderFactory.createLoweredBevelBorder());
            addRadioButton("Raised bevel", BorderFactory.createRaisedBevelBorder());
            addRadioButton("Etched", BorderFactory.createEtchedBorder());
            addRadioButton("Line", BorderFactory.createLineBorder(Color.BLUE));
            addRadioButton("Matte", BorderFactory.createMatteBorder(10, 10, 10, 10, Color.BLUE));
            addRadioButton("Empty", BorderFactory.createEmptyBorder());

            Border etched = BorderFactory.createEtchedBorder();
            Border titled = BorderFactory.createTitledBorder(etched, "Border types");
            buttonPanel.setBorder(titled);

            setLayout(new GridLayout(2, 1));
            add(buttonPanel);
            add(demoPanel);
            pack();
    }

    public void addRadioButton(String buttonName, Border b) {
            JRadioButton button = new JRadioButton(buttonName);
            button.addActionListener(event -> demoPanel.setBorder(b));
            group.add(button);
            buttonPanel.add(button);
    }
}
```

5.4 Choice Components

5.4.4 Combo Boxes

- radio buttons take up too much screen space
- combo boxes: the user can then select item from a list of choices drops down

5.4 Choice Components



The quick brown fox jumps over the lazy dog.

```
public class ComboBoxFrame extends JFrame {
        private JComboBox<String> faceCombo;
        private JLabel label;
        private static final int DEFAULT_SIZE = 24;

        public ComboBoxFrame() {
                // add the sample text label

                label = new JLabel("The quick brown fox jumps over the lazy dog.");
                label.setFont(new Font("Serif", Font.PLAIN, DEFAULT_SIZE));
                add(label, BorderLayout.CENTER);

                // make a combo box and add face names
                faceCombo = new JComboBox<>();
                faceCombo.addItem("Serif");
                faceCombo.addItem("SansSerif");
                faceCombo.addItem("Monospaced");
                faceCombo.addItem("Dialog");
                faceCombo.addItem("DialogInput");

                // the combo box listener changes the label font to the selected face name
                faceCombo.addActionListener(event -> label
                                .setFont(new Font
                                (faceCombo.getItemAt(faceCombo.getSelectedIndex()),
                                Font.PLAIN, DEFAULT_SIZE)));

                // add combo box to a panel at the frame's southern border

                JPanel comboPanel = new JPanel();
                comboPanel.add(faceCombo);
                add(comboPanel, BorderLayout.SOUTH);
                pack();
        }
}
```

5.4 Choice Components

5.4.5 Sliders

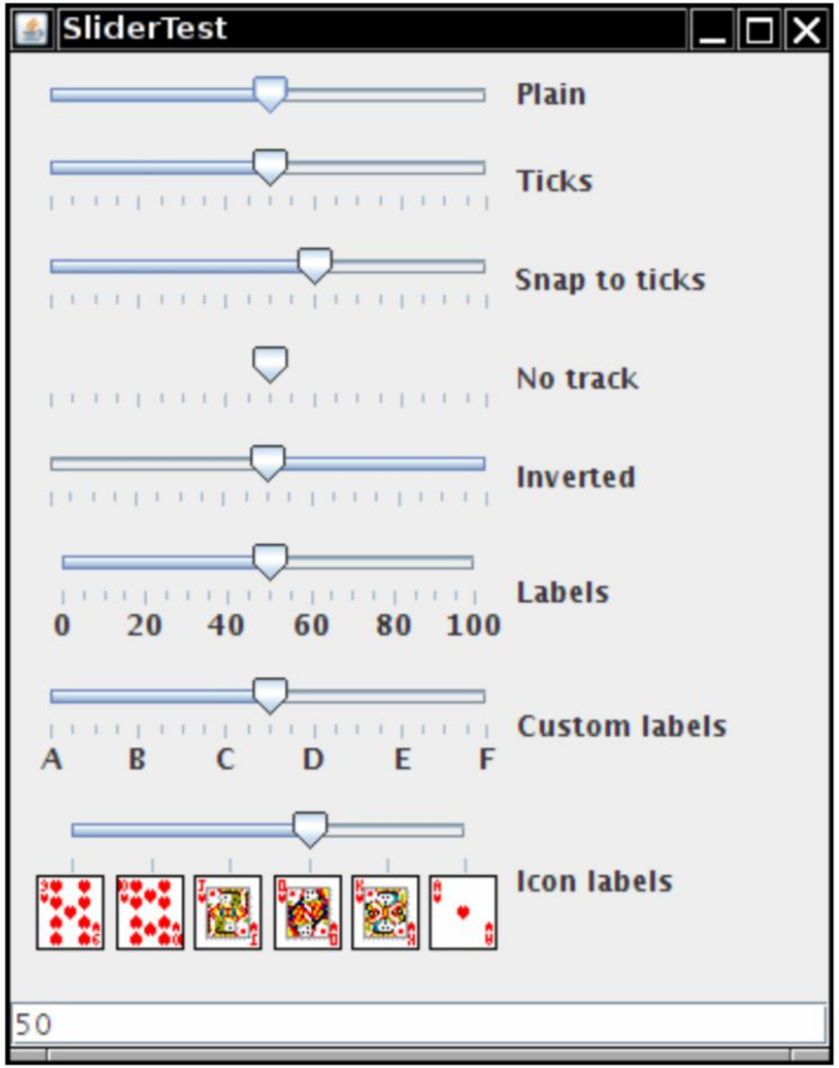
- Combo boxes let users choose from a discrete set of values.
- Sliders offer a choice from a continuum of values - e.g., any number between 1 and 100.
- Construct slider

```
JSlider slider = new JSlider(min, max, initialValue);
```

- retrieve the slider value:

```
ChangeListener listener = event -> {
JSlider slider = (JSlider) event.getSource();
int value = slider.getValue();
...
};
```

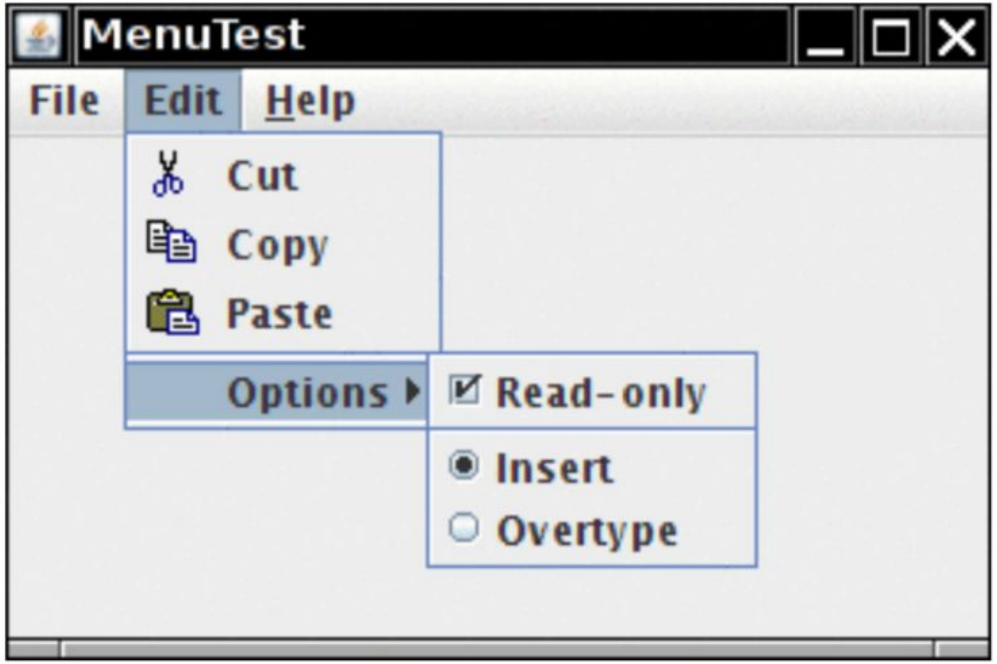
Code: v1ch12.slider



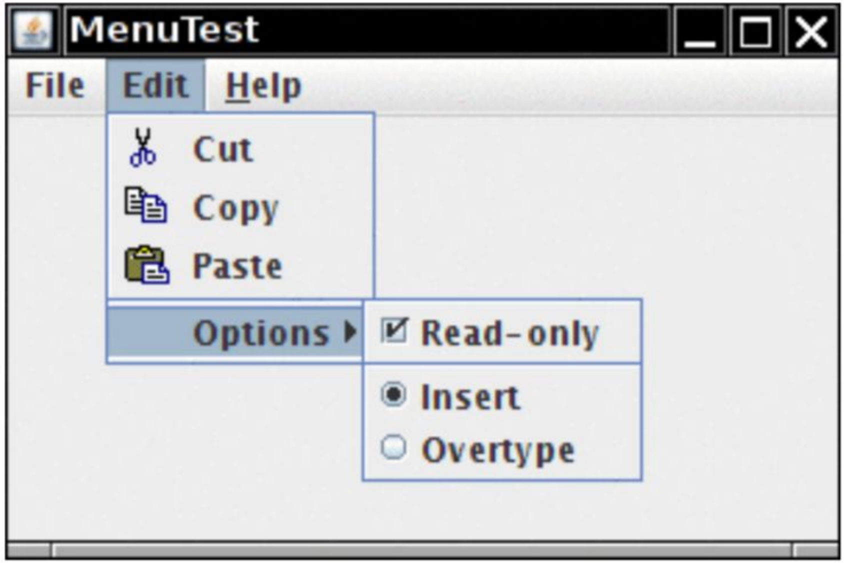
50

5.5 Menus

A menu bar at the top of a window contains the names of the pull-down menus. Clicking on a name opens the menu containing menu items and submenus.



5.5 Menus



5.5.1 Menu Building

- create a menu bar:

```
JMenuBar menuBar = new JMenuBar();
```

- add it at the top of the frame

```
frame.setJMenuBar(menuBar);
```

- for each menu, you create a menu object:

```
JMenu editMenu = new JMenu("Edit");
```

- add the top-level menus to the menu bar:

```
menuBar.add(editMenu);
```

- add menu items, separators, and submenus to the menu object:

```
JMenuItem pasteItem = new JMenuItem("Paste");
editMenu.add(pasteItem);
editMenu.addSeparator();
JMenu optionsMenu = . . .; // a submenu
editMenu.add(optionsMenu);
```

- install an action listener for each menu item:

```
ActionListener listener = . . .;
pasteItem.addActionListener(listener);
```

5.5 Menus

5.5.2 Icons in Menu Items

- specify the icon with the JMenuItem(String, Icon) or JMenuItem(Icon) constructor, setIcon

```
JMenuItem cutItem = new JMenuItem("Cut", new ImageIcon("cut.gif"));
```

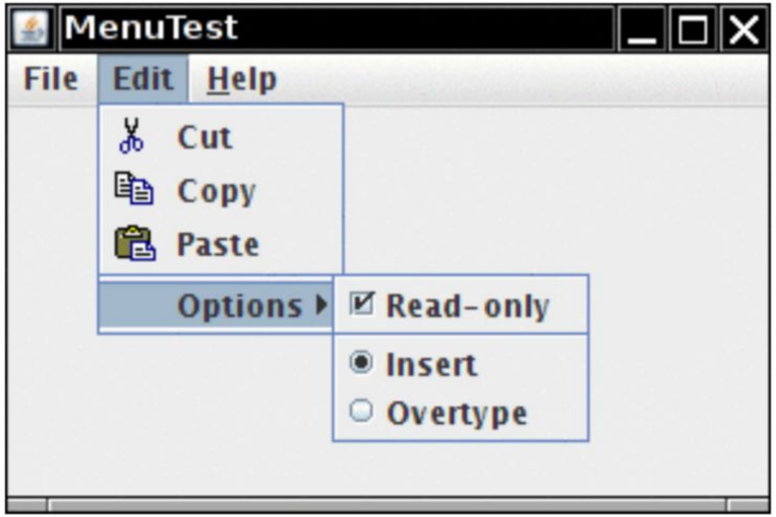
- By default, the menu item text is placed to the right of the icon; change it by setHorizontalTextPosition

```
cutItem.setHorizontalTextPosition(SwingConstants.LEFT);
```

- you can set the icon in the AbstractAction constructor:

```
cutAction = new AbstractAction("Cut", new ImageIcon("cut.gif")) {
        public void actionPerformed(ActionEvent event) {
        ... }
};
```

5.5 Menus



5.5.3 Checkbox and Radio Button Menu Items

Checkbox and radio button menu items display
a checkbox or radio button next to the name

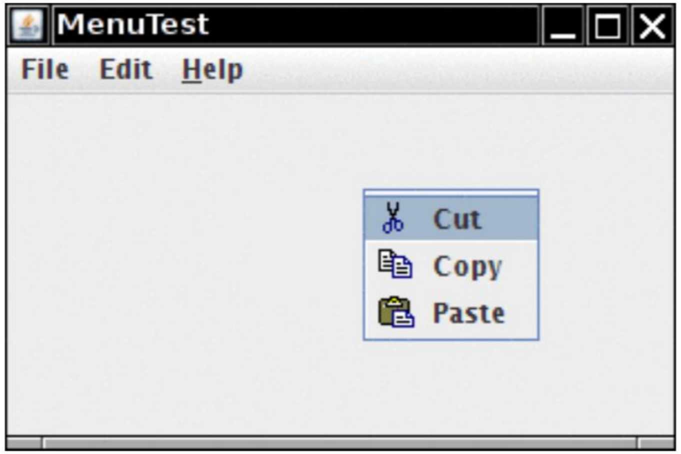
- Create a checkbox menu item

```
JCheckBoxMenuItem readonlyItem = new
JCheckBoxMenuItem("Read-only");
optionsMenu.add(readonlyItem);
```

- Create radio button menu items, add them to a button group

```
ButtonGroup group = new ButtonGroup();

JRadioButtonMenuItem insertItem = new
JRadioButtonMenuItem("Insert");
insertItem.setSelected(true);

JRadioButtonMenuItem overtypeItem = new
JRadioButtonMenuItem("Overtype");

group.add(insertItem);
group.add(overtypeItem);

optionsMenu.add(insertItem);
optionsMenu.add(overtypeItem);
```

5.5 Menus



5.5.4 Pop-Up Menus: a menu that is not attached to a menu bar but floats somewhere

- create a pop-up menu

```
JPopupMenu popup = new JPopupMenu();
```

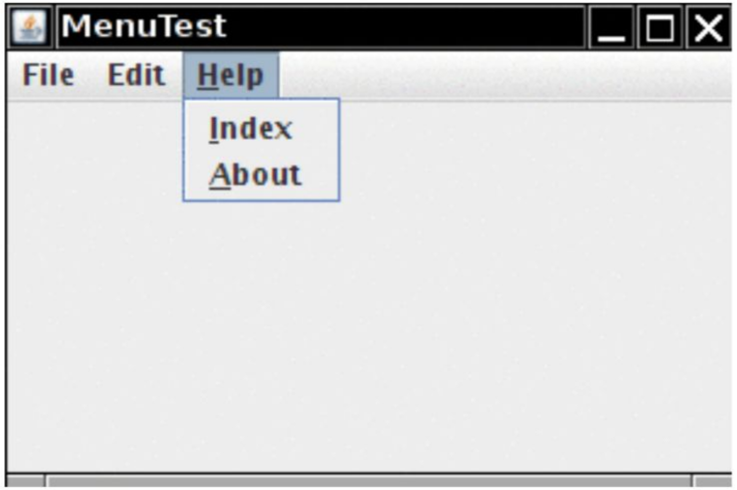
- add your menu items

```
JMenuItem item = new JMenuItem("Cut");
item.addActionListener(listener);
popup.add(item);
```

- Pop up a menu when the user clicks on a component

```
component.setComponentPopupMenu(popup);
```

5.5 Menus



5.5.5 Keyboard Mnemonics and Accelerators:

- Keyboard mnemonics: select a submenu or menu item from the currently open menu
  - specifying a mnemonic letter in the menu item constructor, or setMnemonic

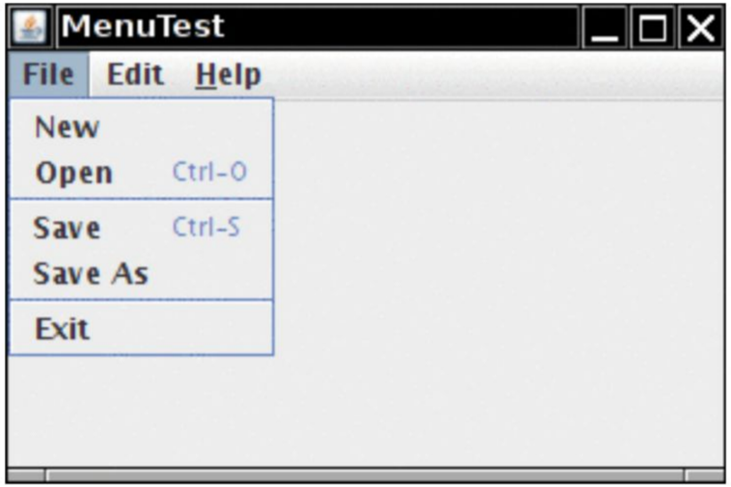
```
JMenuItem aboutItem = new JMenuItem("About", 'A');
```

  - Use setMnemonic()
    ```
    helpMenu.setMnemonic('H');
    ```

  - If you have an Action object, you can add the mnemonic as the value of the Action.MNEMONIC_KEY key:

```
cutAction.putValue(Action.MNEMONIC_KEY, new Integer('A'));
```

5.5 Menus

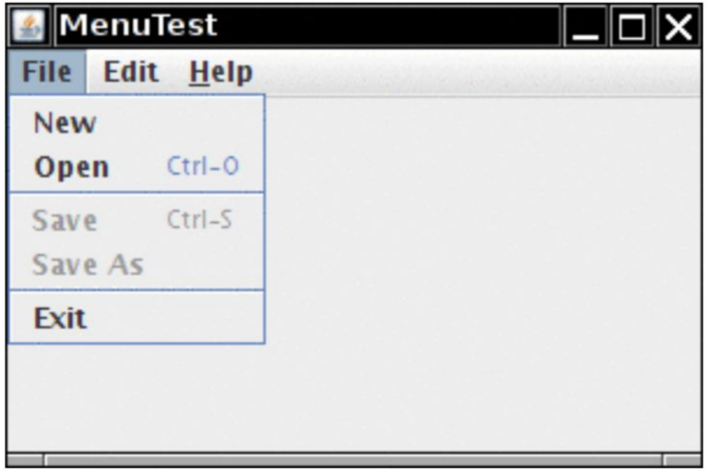


5.5.5 Keyboard Mnemonics and Accelerators

- Accelerators are keyboard shortcuts that let you select menu items without ever opening a menu
- Attaches the accelerator Ctrl+O to the openItem menu item

```
openItem.setAccelerator(KeyStroke.getKeyStroke("ctrl O"));
```

5.5 Menus



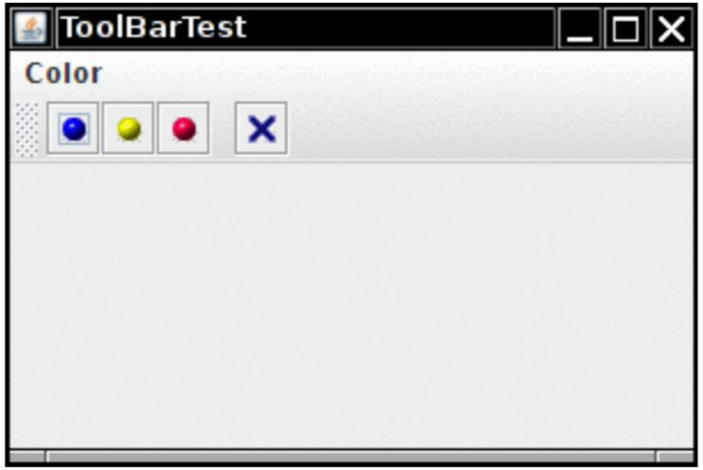
5.5.6 Enabling and Disabling Menu Items

- enable or disable a menu item, use the setEnabled method

```
public void menuSelected(MenuEvent event) {
        saveAction.setEnabled(!readonlyItem.isSelected());
        saveAsAction.setEnabled(!readonlyItem.isSelected());
}
```

Full codes: v1ch12.menu

5.5 Menus



5.5.7 Toolbars

- A toolbar is a button bar that gives quick access to the most commonly used commands in a program, you can move toolbars elsewhere
- create a toolbar, add components or an action object to it:

```
JToolBar bar = new JToolBar();
bar.add(blueButton);
// or add action
// The icon of the action is displayed in the toolbar.
bar.add(blueAction);
```

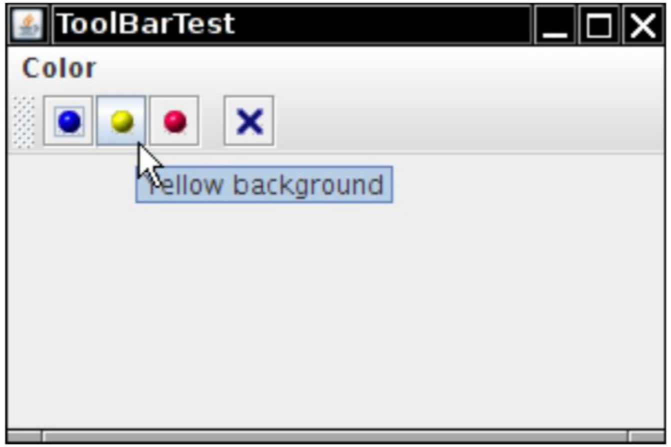
- separate groups of buttons with a separator

```
bar.addSeparator();
```

- add the toolbar to the frame

```
add(bar, BorderLayout.NORTH);
```

5.5 Menus

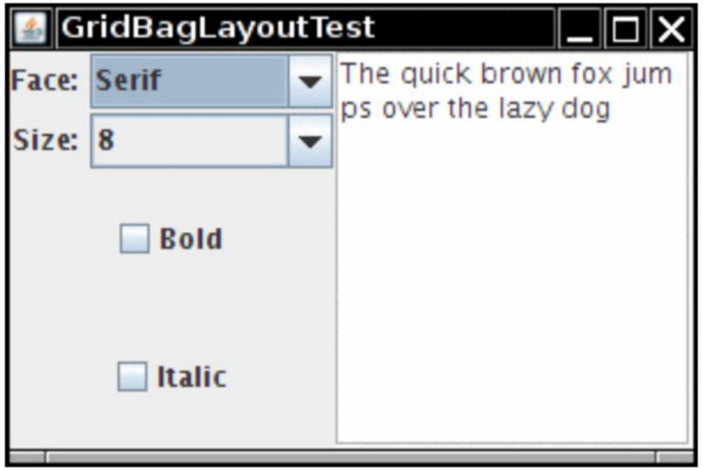


5.5.8 Tooltips

- A tooltip is activated when the cursor rests for a moment over a button.
- The tooltip text is displayed inside a colored rectangle.
- add tooltips to any JComponent simply by calling the setToolTipText method
  `exitButton.setToolTipText("Exit");`
- if using Action objects, associate the tooltip with the SHORT_DESCRIPTION key:
  `exitAction.putValue(Action.SHORT_DESCRIPTION, "Exit");`

Full codes: v1ch12.toolBar

5.6 Sophisticated Layout Management

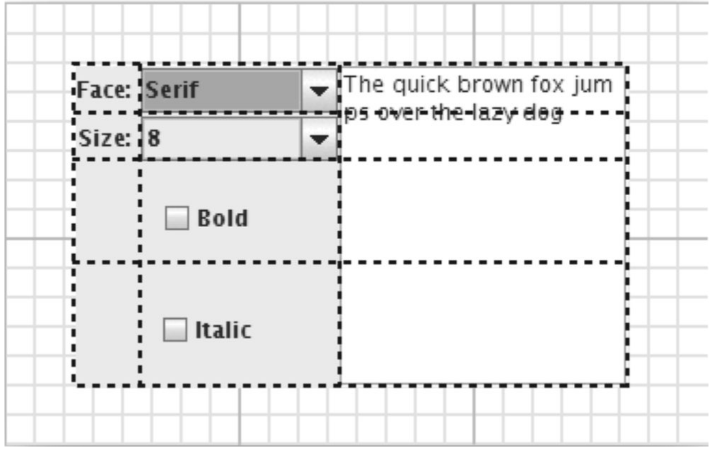


The quick brown fox jum
ps over the lazy dog

5.6.1 The Grid Bag Layout

- The grid bag layout is the mother of all layout managers.
- In a grid bag layout, the rows and columns can have variable sizes.
- You can join adjacent cells to make room for larger components.

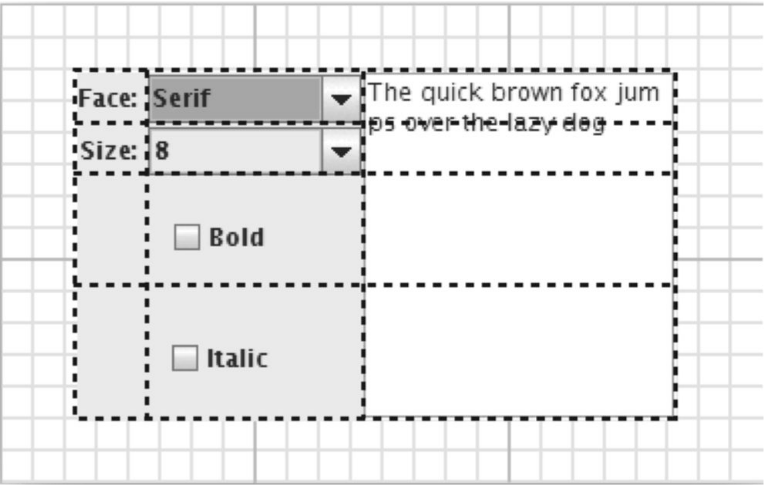
5.6 Sophisticated Layout Management



- Create an object of type GridBagLayout, the layout manager will try to guess it from the information you give it later.
  GridBagLayout layout = new GridBagLayout();

- Set this GridBagLayout object to be the layout manager for the component.
  panel.setLayout(layout);

5.6 Sophisticated Layout Management



- For each component, create an object of type GridBagConstraints.
  - weights are used to determine how to distribute space among columns (weightx) and among rows (weighty)
  - gridx and gridy values specify the column and row positions of the upper left corner of the component to be added.
  - gridwidth and gridheight values determine how many columns and rows the component occupies.

```
GridBagConstraints constraints = new GridBagConstraints();
constraints.weightx = 100;
constraints.weighty = 100;

constraints.gridx = 0;
constraints.gridy = 2;
constraints.gridwidth = 2; // 2 columns
constraints.gridheight = 1; // 1 row
```

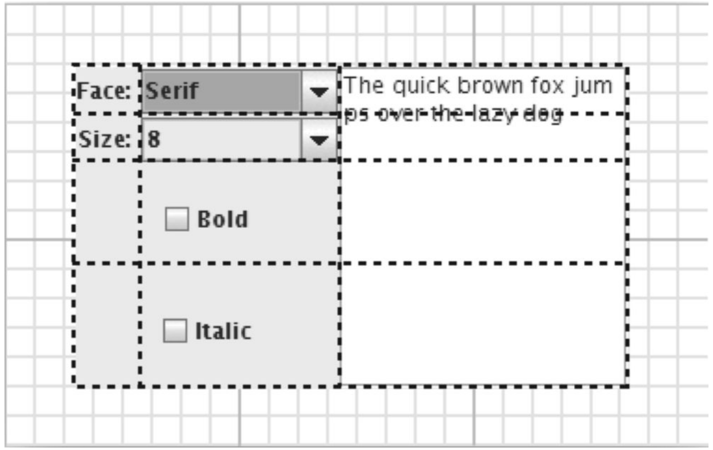
5.6 Sophisticated Layout Management

5.6.1.3 The fill and anchor Parameters

- If you don’t want a component to stretch out and fill the entire area, set the fill constraint.
  - GridBagConstraints.NONE,
  - GridBagConstraints.HORIZONTAL,
  - GridBagConstraints.VERTICAL,
  - GridBagConstraints.BOTH.

- specify where in the area you want it by setting the anchor field
  - GridBagConstraints.CENTER (the default),
  - GridBagConstraints.NORTH,
  - GridBagConstraints.NORTHEAST,
  - GridBagConstraints.EAST

5.6 Sophisticated Layout Management



- Create an object of type GridBagLayout, the layout manager will try to guess it from the information you give it later.
  GridBagLayout layout = new GridBagLayout();

- Set this GridBagLayout object to be the layout manager for the component.
  panel.setLayout(layout);

- For each component, create an object of type GridBagConstraints.
  Set field values of the GridBagConstraints object to specify how the components are laid out within the grid bag.
  GridBagConstraints constraints = new GridBagConstraints();
  constraints.weightx = 100; constraints.weighty = 100;
  constraints.gridx = 0; constraints.gridy = 2;
  constraints.gridwidth = 2; constraints.gridheight = 1;

- add each component with its constraints by using the call
  add(component, constraints);
  panel.add(component, constraints);

5.6 Sophisticated Layout Management

The following recipe makes gridbag layouts relatively trouble free:

1. Sketch out the component layout on a piece of paper.
2. Find a grid such that the small components are each contained in a cell and the larger components span multiple cells.
3. Label the rows and columns of your grid with 0, 1, 2, 3, . . . You can now read off the gridx, gridy, gridwidth, and gridheight values.
4. For each component, ask yourself whether it needs to fill its cell horizontally or vertically. If not, how do you want it aligned? This tells you the fill and anchor parameters.
5. Set all weights to 100. If you want a particular row or column to always stay at its default size, set the weightx or weighty to 0 in all components that belong to that row or column.
6. Write the code. Carefully double-check your settings for the GridBagConstraints.
7. Compile, run, and enjoy.

5.6.3 Using No Layout Manager

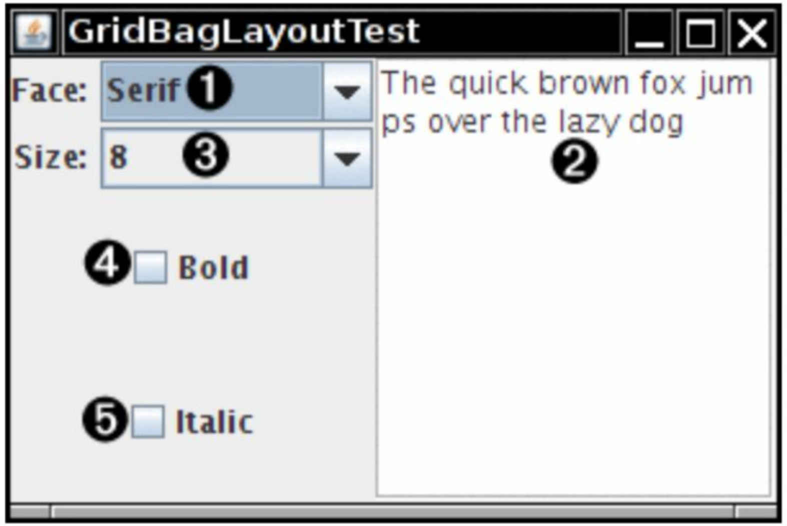
Drop a component at a fixed location: not a great idea for platform-independent applications, but there is nothing wrong with using it for a quick prototype.

1. Set the layout manager to null.
2. Add the component you want to the container.
3. Specify the position and size that you want:

```
frame.setLayout(null);
JButton ok = new JButton("OK");
frame.add(ok);
ok.setBounds(10, 10, 30, 15);
```

5.6.5 Traversal Order

- When a window is first displayed, the first component in the traversal order has the keyboard focus.
- Each time the user presses the Tab key, the next component gains focus.
- The traversal order is straightforward: first, left to right, and then, top to bottom.
- Remove a component from the focus traversal: component.setFocusable(false);



5.7 Dialog Boxes

- Dialog boxes give information to, or get information from, the user.
- Modal dialog boxes: users cannot interact with the remaining windows of the application; use it when you need information from the user before you can proceed with execution
- Modeless dialog boxes: the user can enter information in both the dialog box and the remainder of the application

5.7 Dialog Boxes

5.7.1 Option Dialogs

The JOptionPane has four static methods to show these simple dialogs:

- showMessageDialog: Show a message and wait for the user to click OK
- showConfirmDialog: Show a message and get a confirmation (like OK/Cancel)
- showOptionDialog: Show a message and get a user option from a set of options
- showInputDialog: Show a message and get one line of user input

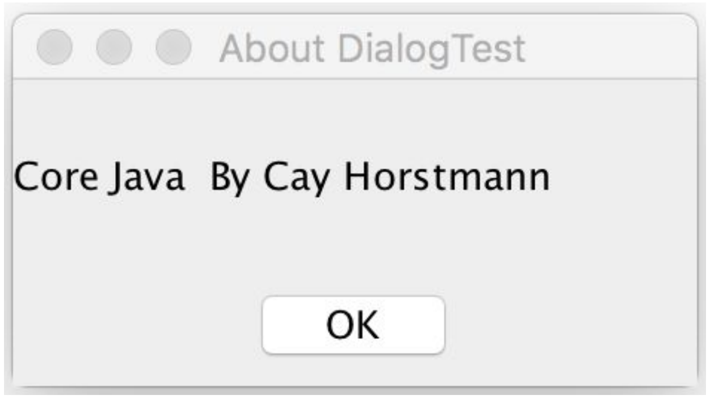
5.7 Dialog Boxes

5.7.1 Option Dialogs

1. Choose the dialog type (message, confirmation, option, or input).
2. Choose the icon (error, information, warning, question, none, or custom).
3. Choose the message (string, icon, custom component, or a stack of them).
4. Choose the option type
   a. For a confirmation dialog: (default, Yes/No, Yes/No/Cancel, or OK/Cancel).
   b. For an option dialog: (strings, icons, or custom components) and the default option.
   c. For an input dialog: choose between a text field and a combo box.
5. Locate the appropriate method to call in the JOptionPane API.

Full codes: v1ch12.optionDialog

5.7 Dialog Boxes



Core Java By Cay Horstmann

5.7.2 Creating Dialogs

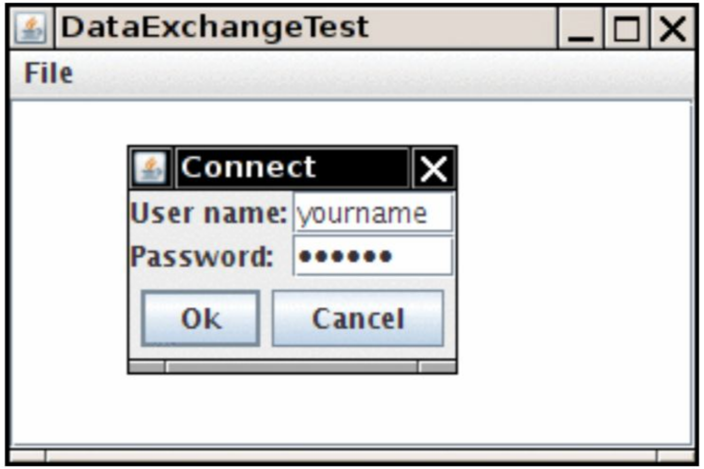
1. In the constructor of your dialog box, call the constructor of the superclass JDialog.
2. Add the user interface components of the dialog box.
3. Add the event handlers.
4. Set the size for the dialog box.

```
public class AboutDialog extends JDialog {
        public AboutDialog(JFrame owner) {
                super(owner, "About DialogTest", true);

                // add label
                add(new JLabel("Core Java\n By Cay Horstmann"), BorderLayout.CENTER);

                // OK button closes the dialog
                JButton ok = new JButton("OK");
                ok.addActionListener(event -> setVisible(false));

                // add OK button to southern border
                JPanel panel = new JPanel();
                panel.add(ok);
                add(panel, BorderLayout.SOUTH);

                pack();
        }
}
```

5.7 Dialog Boxes

5.7.3 Data Exchange

Full codes: v1ch12.dataExchange



```java
public class PasswordChooser extends JPanel {
	private JTextField username;
	private JPasswordField password;
	private JButton okButton;
	private boolean ok;
	private JDialog dialog;

	public PasswordChooser() {
		setLayout(new BorderLayout());

		// construct a panel with user name and password fields
		JPanel panel = new JPanel();
		panel.setLayout(new GridLayout(2, 2));
		panel.add(new JLabel("User name:"));
		panel.add(username = new JTextField(""));
		panel.add(new JLabel("Password:"));
		panel.add(password = new JPasswordField(""));
		add(panel, BorderLayout.CENTER);

		// create Ok and Cancel buttons that terminate the dialog
		okButton = new JButton("Ok");
		okButton.addActionListener(event -> {
			ok = true;
			dialog.setVisible(false);
		});

		JButton cancelButton = new JButton("Cancel");
		cancelButton.addActionListener(event -> dialog.setVisible(false));

		// add buttons to southern border
		JPanel buttonPanel = new JPanel();
		buttonPanel.add(okButton);
		buttonPanel.add(cancelButton);
		add(buttonPanel, BorderLayout.SOUTH);
	}

	public void setUser(User u) {
		username.setText(u.getName());
	}

	public User getUser() {
		return new User(username.getText(), password.getPassword());
	}

	public boolean showDialog(Component parent, String title) {
		ok = false;

		// locate the owner frame
		Frame owner = null;
		if (parent instanceof Frame)
			owner = (Frame) parent;
		else
			owner = (Frame) SwingUtilities.
				getAncestorOfClass(Frame.class, parent);

		// if first time, or if owner has changed, make new dialog
		if (dialog == null || dialog.getOwner() != owner) {
			dialog = new JDialog(owner, true);
			dialog.add(this);
			dialog.getRootPane().setDefaultButton(okButton);
			dialog.pack();
		}

		// set title and show dialog
		dialog.setTitle(title);
		dialog.setVisible(true);
		return ok;
	}
}
```

5.7 Dialog Boxes

```
public class DataExchangeFrame extends JFrame {
        public static final int TEXT_ROWS = 20;
        public static final int TEXT_COLUMNS = 40;
        private PasswordChooser dialog = null;
        private JTextArea textArea;

        public DataExchangeFrame() {
                // construct a File menu
                JMenuBar mbar = new JMenuBar();
                setJMenuBar(mbar);
                JMenu fileMenu = new JMenu("File");
                mbar.add(fileMenu);

                // add Connect and Exit menu items
                JMenuItem connectItem = new JMenuItem("Connect");
                connectItem.addActionListener(new ConnectAction());
                fileMenu.add(connectItem);

                // The Exit item exits the program
                JMenuItem exitItem = new JMenuItem("Exit");
                exitItem.addActionListener(event -> System.exit(0));
                fileMenu.add(exitItem);

                textArea = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
                add(new JScrollPane(textArea), BorderLayout.CENTER);
                pack();
        }
```

```
private class ConnectAction implements ActionListener {
        public void actionPerformed(ActionEvent event) {
                if (dialog == null)
                        dialog = new PasswordChooser();

                // set default values
                dialog.setUser(new User("yourname", null));

                // pop up dialog
                if (dialog.showDialog(DataExchangeFrame.this,
                                        "Connect")) {
                        // if accepted, retrieve user input
                        User u = dialog.getUser();
                        textArea.append("user name = " + u.getName()
                                + ", password = "
                                + (new String(u.getPassword())) + "\n");
                }
        }
}}
```

5.7 Dialog Boxes

5.7.4 File Dialogs

To put up a file dialog box and recover what the user chooses from the box:

1. Make a JFileChooser object, possible to reuse a file chooser dialog with multiple frames.
   ```
   JFileChooser chooser = new JFileChooser();
   ```
2. Set the directory by calling the setCurrentDirectory method.
   ```
   chooser.setCurrentDirectory(new File("."));
   ```
3. If you have a default file name that you expect the user to choose, supply it with the setSelectedFile method:
   ```
   chooser.setSelectedFile(new File(filename));
   ```
4. To enable the user to select multiple files in the dialog, call the setMultiSelectionEnabled method.
   ```
   chooser.setMultiSelectionEnabled(true);
   ```
5. If you want to restrict the display of files in the dialog to those of a particular type, you need to set a file filter.
   ```
   chooser.setFileFilter(new FileNameExtensionFilter("Image files", "gif", "jpg"));
   ```

5.7 Dialog Boxes

5.7.4 File Dialogs

6. By default, a user can select only files with a file chooser. If you want the user to select directories **setFileSelectionMode** with JFileChooser.FILES_ONLY, JFileChooser.DIRECTORIES_ONLY, or JFileChooser. FILES_AND_DIRECTORIES

7. Show the dialog box by calling the showOpenDialog or showSaveDialog method
These calls return only when the user has approved, canceled, or dismissed the file dialog.

8. Get the selected file or files with the getSelectedFile() or getSelectedFiles() method.

```
String filename = chooser.getSelectedFile().getPath();
```