

Глава 13

ЭЛЕМЕНТЫ КОМПОНОВКИ И УПРАВЛЕНИЯ

В первых версиях Java (1.0.x) были созданы элементы управления AWT, такие как метки, кнопки, списки, текстовые поля, предоставляющие пользователю различные способы управления приложением. Эти элементы, наследуемые от абстрактного класса `java.awt.Component` и называемые также компонентами, были частично зависимы от аппаратной платформы и не в полной мере объектно-ориентированы по способу использования. Развитие парадигмы “write once, run everywhere” (“написать однажды, запускать везде”) привело к разработке таких компонентов (библиотеки Swing), которые были не привязаны к конкретной платформе. Эти классы доступны разработчикам в составе как JDK, так и отдельного продукта JFC (Java Foundation Classes). Причем для совместимости со старыми версиями JDK компоненты из AWT остались нетронутыми, хотя компания JavaSoft, отвечающая за выпуск JDK, рекомендует не смешивать в одной и той же программе старые и новые компоненты. Кроме пакета Swing, библиотека JFC содержит большое число компонентов JavaBeans, которые могут использоваться как для ручной, так и для визуальной разработки пользовательских интерфейсов.

Менеджеры размещения

Перед использованием управляющих компонентов (например, кнопок) их надо расположить на форме в нужном порядке. Вместо ручного расположения применяются менеджеры размещения, определяющие способ, который панель использует для задания порядка размещения управляющего элемента на форме. Менеджеры размещения контролируют, как выполняется позиционирование компонентов, добавляемых в окна, а также их упорядочение. Если пользователь изменяет размер окна, менеджер размещения переупорядочивает компоненты в новой области так, чтобы они оставались видимыми и в то же время сохранили свои позиции относительно друг друга.

Менеджер размещения представляет собой один из классов `FlowLayout`, `BorderLayout`, `GridLayout`, `CardLayout`, `BoxLayout`, реализующих интерфейс `LayoutManager`, устанавливающий размещение.

Класс `FlowLayout` – менеджер поточной компоновки. При этом компоненты размещаются от левого верхнего угла окна, слева направо и сверху вниз, как и обычный текст. Этот менеджер используется по умолчанию при добавлении компонентов в апплеты. При использовании библиотеки AWT менеджер `FlowLayout` представляет собой класс, объявленный следующим образом:

```
public class FlowLayout extends Object
    implements LayoutManager, Serializable { }
```

В следующем примере демонстрируются возможности поточной компоновки различных элементов управления.

/ пример #1 : поточная компоновка по центру: FlowLayoutEx.java */*

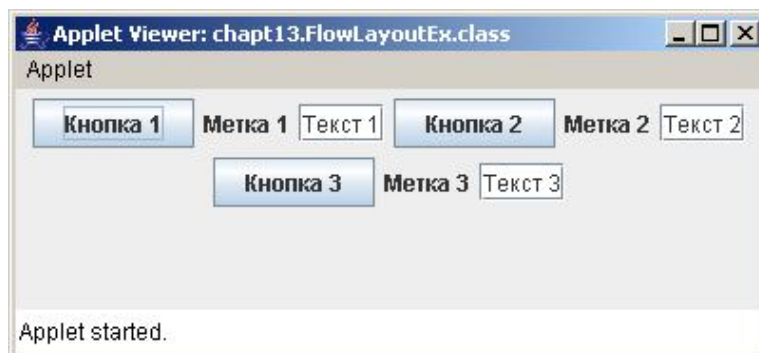
```
package chapt13;
import java.awt.*;
import javax.swing.*;

public class FlowLayoutEx extends JApplet {
    private Component c[] = new Component[9];

    public void init() {
        String[] msg =
            { "Метка 1", "Метка 2", "Метка 3" };
        String[] str =
            { "Кнопка 1", "Кнопка 2", "Кнопка 3" };
        String[] txt = {"Текст 1", "Текст 2", "Текст 3"};
        //установка менеджера размещений
        setLayout(new FlowLayout());
        setBackground(Color.gray);
        setForeground(Color.getHSBColor(1f, 1f, 1f));
        for (int i = 0; i < c.length/3; i++) {
            c[i] = new JButton(str[i]);
            add(c[i]);
            c[i + 3] = new JLabel(msg[i]);
            add(c[i + 3]);
            c[i+6] = new JTextField(txt[i]);
            add(c[i + 6]);
        }
        setSize(450, 150);
    }
}
```

Перегружаемый метод **add(Component ob)**, определенный в классе **java.awt.Container** (подклассе **Component**), добавляет компоненты **JButton**, **JLabel**, **JTextField** к окну и прорисовывает их всякий раз, когда окно отображается на экран.

Метод **setLayout(LayoutManager mgr)** устанавливает менеджер размещения для данного контейнера. Результаты работы апплета приведены на рисунке.



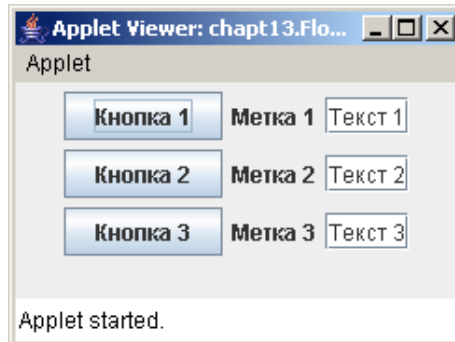


Рис. 13.1. Размещение компонентов **FlowLayout**

Менеджер **GridLayout** разделяет форму на заданное количество рядов и колонок. В отличие от него компоновка **BoxLayout** размещает некоторое количество компонентов по вертикали или горизонтали. На способ расположения компонентов изменение размеров формы не влияет.

```
/* пример # 2 : компоновка в табличном виде: GridLayoutEx.java */
package chapt13;
import javax.swing.*.*;
import java.awt.*.*;
public class GridLayoutEx extends JApplet {
    private Component b[] = new Component[7];
    public void init() {
        setLayout(new GridLayout(2, 4)); /*две строки,
                                         четыре столбца*/
        for (int i = 0; i < b.length; i++)
            add((b[i] = new JButton("(" + i + ")")));
    }
}
```

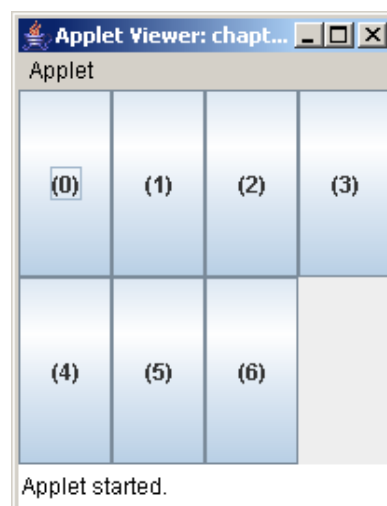


Рис. 13.2. Размещение компонентов **GridLayout**

Менеджер **BorderLayout** позволяет позиционировать элементы и группы из них в областях фиксированного размера, граничащих со сторонами фрейма, которые обозначаются параметрами сторонами света: **NORTH**, **SOUTH**, **EAST**, **WEST**. Остальное пространство обозначается как **CENTER**.

/ пример # 3 : фиксированная компоновка по областям:*

*BorderGridLayoutDemo.java */*

```
package chapt13;
import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.Container;
import javax.swing.JPanel;
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JToggleButton; // «западающая» кнопка

public class BorderGridLayoutDemo extends JFrame {
    public BorderGridLayoutDemo() {
        Container c = getContentPane();
        c.setLayout(new BorderLayout());
        c.add(new JToggleButton("--1--"), BorderLayout.WEST);
        c.add(new JToggleButton("--2--"), BorderLayout.SOUTH);
        c.add(new JToggleButton("--3--"), BorderLayout.EAST);
        JPanel jPanel = new JPanel();
        c.add(jPanel, BorderLayout.NORTH);
        jPanel.setSize(164, 40);
        jPanel.setLayout(new GridLayout(2, 4));
        for (int i = 0; i < 7; i++)
            jPanel.add(new JButton("" + i));
    }
    public static void main(String[] args) {
        BorderGridLayoutDemo fr =
            new BorderGridLayoutDemo();
        fr.setSize(300, 200);
        fr.setTitle("Border & Grid Layouts Example");
        fr.setDefaultCloseOperation(EXIT_ON_CLOSE);
        fr.setVisible(true);
    }
}
```

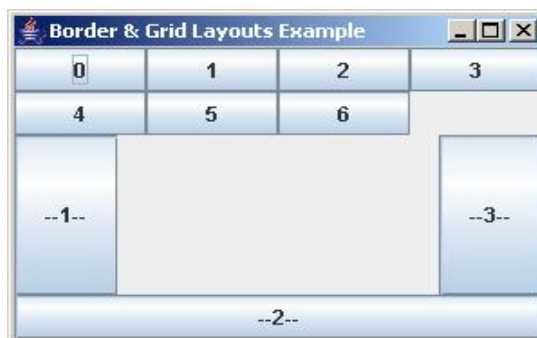


Рис. 13.3. Размещение компонентов **BorderLayout** и **GridLayout**

Компоновка **BoxLayout** позволяет группировать элементы в подобластях фрейма в строки и столбцы. Возможности класса **Box** позволяют размещать компоненты в рамке, ориентированной горизонтально или вертикально.

```

/* пример # 4 : компоновка в группах с ориентацией: BoxLayoutDemo.java */
package chapt13;
import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.Font;
import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JToggleButton;
import javax.swing.border.EtchedBorder;
import javax.swing.border.TitledBorder;

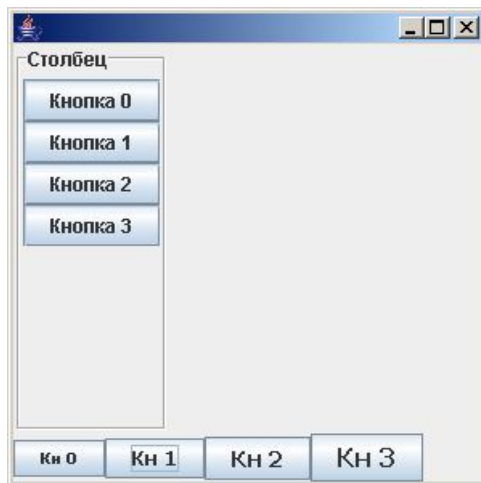
public class BoxLayoutDemo extends JFrame {

    public BoxLayoutDemo() {
        Container c = getContentPane();
        setBounds(20, 80, 300, 300);
        c.setLayout(new BorderLayout());
        Box row = Box.createHorizontalBox();
        for (int i = 0; i < 4; i++) {
            JButton btn = new JButton("Кн " + i);
            btn.setFont(new Font("Tahoma", 1, 10 + i * 2));
            row.add(btn);
        }
        c.add(row, BorderLayout.SOUTH);

        JPanel col = new JPanel();
        col.setLayout(
            new BoxLayout(col, BoxLayout.Y_AXIS));
        col.setBorder(
            new TitledBorder(new EtchedBorder(), "Столбец"));
        for (int i = 0; i < 4; i++) {
            JToggleButton btn =
                new JToggleButton("Кнопка " + i);
            col.add(btn);
        }
        c.add(col, BorderLayout.WEST);
    }

    public static void main(String[] args) {
        BoxLayoutDemo frame = new BoxLayoutDemo();
        frame.setDefaultCloseOperation(EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}

```

Рис. 13.4. Размещение компонентов **BoxLayout** и **Box**

Для того чтобы располагать компоненты в произвольных областях фрейма, следует установить для менеджера размещений значение **null** и воспользоваться методом **setBounds()**.

/ пример # 5 : произвольное размещение: NullLayoutEx.java */*

```
package chapt13;
import java.awt.Container;
import javax.swing.*;

public class NullLayoutEx extends JFrame {
    public NullLayoutEx() {
        Container c = getContentPane();
        //указание размеров фрейма
        setBounds(20, 80, 300, 300);
        c.setLayout(null);
        JButton jb = new JButton("Кнопка");
        //указание координат и размеров кнопки
        jb.setBounds(200, 50, 90, 40);
        c.add(jb);
        JTextArea jta = new JTextArea();
        //указание координат и размеров текстовой области
        jta.setBounds(10, 130, 180, 70);
        jta.setText("Здесь можно вводить текст");
        c.add(jta);
    }

    public static void main(String args[]) {
        NullLayoutEx nl = new NullLayoutEx();
        nl.setDefaultCloseOperation(EXIT_ON_CLOSE);
        nl.setVisible(true);
    }
}
```

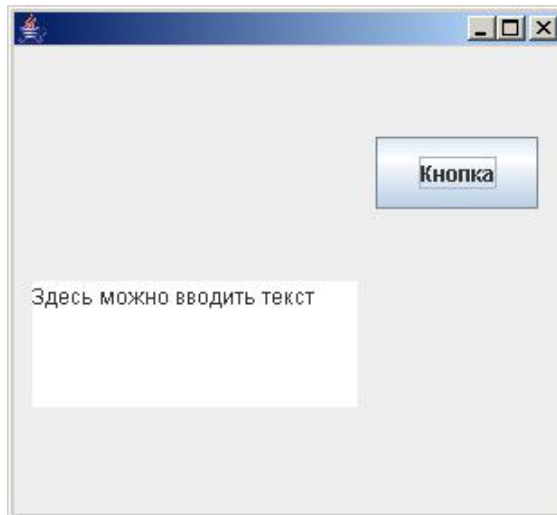


Рис. 13.5. Произвольное размещение компонентов

Элементы управления

Все элементы управления из пакета AWT являются наследниками классов **Component** и **Container**. При использовании пакета Swing компоненты наследуются от класса **JComponent**, производного от класса **Container**.

В качестве примеров можно привести текстовые метки **Label**, **JLabel**, которые создаются с помощью конструкторов, устанавливающих текст метки. Возможность изменения текста метки предоставляет метод **setText(String txt)**. Получить значение текста метки можно методом **getText()**.

Кнопки **Button** и **JButton**, **CheckBox** и **JCheckBox**, **RadioButton** и **JRadioButton**, **JToggleButton** используются для генерации и обработки событий.

Списки **List** и **JList** позволяют выбирать один или несколько элементов из списка.

Полосы прокрутки **ScrollBar** и **JScrollBar** используются для облегчения просмотра.

Однострочная область ввода **TextField** и **JTextField** и многострочная область ввода – **TextArea** и **JTextArea** позволяют редактировать и вводить текст (см. рис. 13.6).

Суперклассом кнопок является класс **AbstractButton**, от которого наследуются два наиболее используемых класса: **JButton** и **JToggleButton**. Первый предназначен для создания обычных кнопок, а второй – для создания «залипающих» кнопок, радиокнопок (класс **JRadioButton**) и отмечаемых кнопок (класс **JCheckBox**). Кроме указанных, от **AbstractButton** наследуется два класса **JCheckBoxMenuItem** и **JRadioButtonMenuItem**, применяемых для организации меню с радиокнопками и отмечаемыми кнопками (см. рис. 13.7).

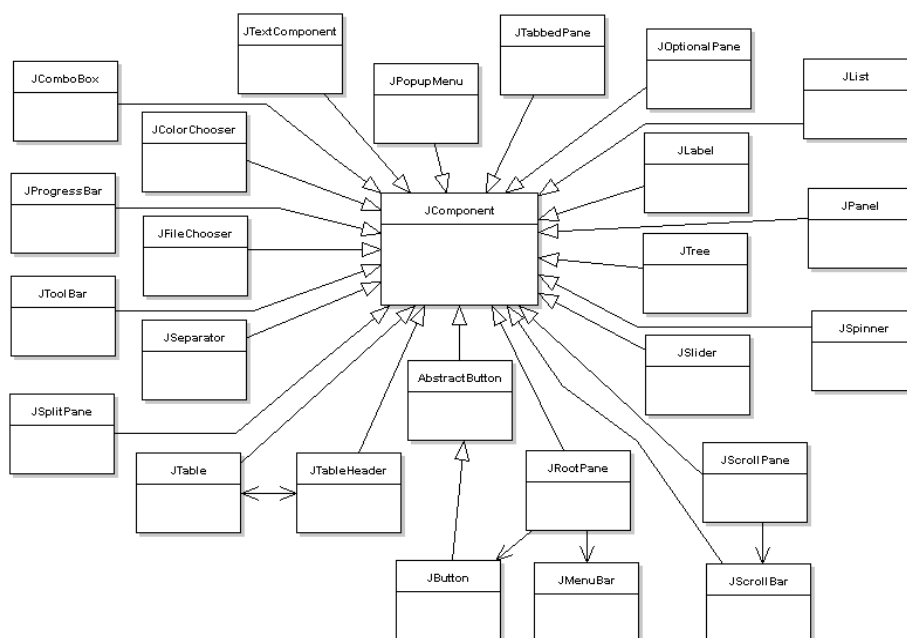


Рис. 13.6. Иерархия наследования компонентов в библиотеке Swing

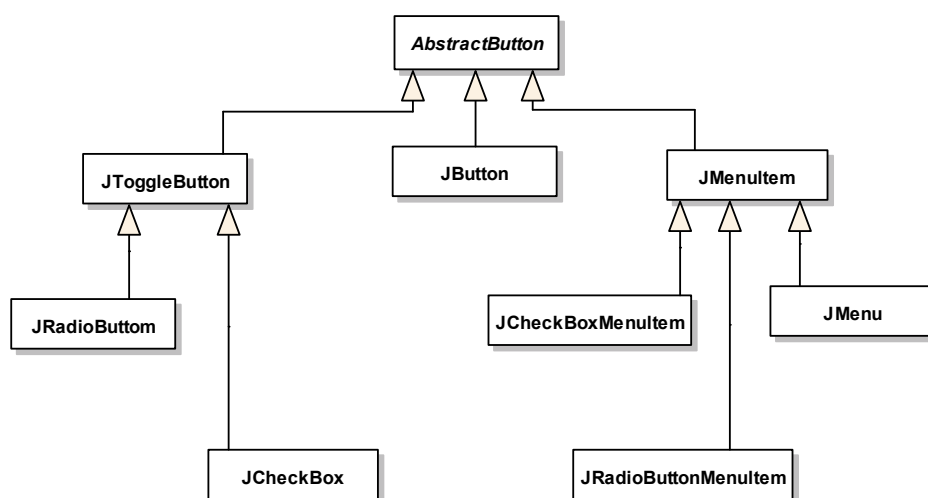


Рис. 13.7. Классы кнопок в Swing

Процесс создания кнопок достаточно прост: вызывается конструктор **JButton** с меткой, которую нужно поместить на кнопке. Класс **JButton** библиотеки Swing для создания обычных кнопок предлагает несколько различных конструкторов: **JButton()**, **JButton(String s)**, **JButton(Icon i)**, **JButton(String s, Icon i)**.

Если используется конструктор без параметров, то получится абсолютно пустая кнопка. Задав текстовую строку, получим кнопку с надписью. Для создания

кнопки с рисунком конструктору передается ссылка на класс пиктограммы. Класс **JButton** содержит несколько десятков методов. **JButton** – это компонент, который автоматически перерисовывается как часть обновления. Это означает, что не нужно явно вызывать перерисовку кнопки, как и любого управляющего элемента; он просто помещается на форму и сам автоматически заботится о своей перерисовке. Чтобы поместить кнопку на форму, достаточно выполнить это в методе **init()**. Каждый раз, когда кнопка нажимается, генерируется action-событие. Оно посылается блокам прослушивания, зарегистрированным для приема события от этого компонента.

// пример #6 : кнопка и ее методы: *VisualEx.java*

```
package chapt13;
import javax.swing.JPanel;
import javax.swing.JApplet;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JTextField;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class VisualEx extends JApplet {
    private JPanel jContentPane = null;
    private JButton yesBtn = null;
    private JButton noBtn = null;
    private JLabel label = null;
    private JTextField textField = null;

    public void init() {
        setSize(180, 160);
        setContentPane(getJContentPane());
        setBackground(java.awt.Color.white);
    }

    private JPanel getJContentPane() {
        if (jContentPane == null) {
            label = new JLabel();
            label.setText("");
            jContentPane = new JPanel();
            jContentPane.setLayout(new FlowLayout());
            jContentPane.add(getYesBtn(), null);
            jContentPane.add(getNoBtn(), null);
            jContentPane.add(label, null);
            jContentPane.add(getTextField(), null);
        }
        return jContentPane;
    }

    private JButton getYesBtn() {
        if (yesBtn == null) {
            yesBtn = new JButton();
            yesBtn.setText("казнить");
            yesBtn.addActionListener(new ActionListener() {
```

```

        public void actionPerformed(ActionEvent e) {
            label.setText("Казнить,");
            textField.setText("нельзя помиловать");
        }
    });
}
return yesBtn;
}
private JButton getNoBtn() {
    if (noBtn == null) {
        noBtn = new JButton();
        noBtn.setText("помиловать");
        noBtn.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                label.setText("Казнить нельзя,");
                textField.setText("помиловать");
            }
        });
    }
    return noBtn;
}
private JTextField getTextField() {
    if (textField == null) {
        textField = new JTextField();
        textField.setColumns(12);
        textField.setHorizontalAlignment(JTextField.CENTER);
        textField.setEditable(false);
    }
    return textField;
}
}

```

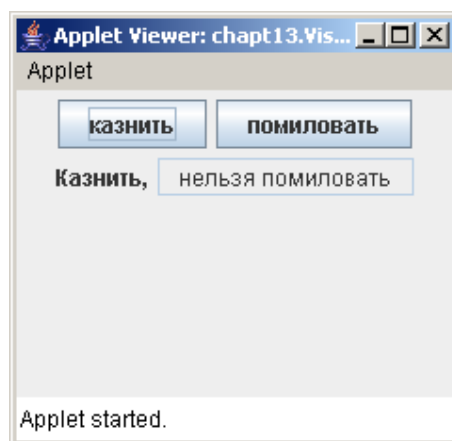


Рис. 13.8. Апплет с кнопками, меткой и текстовым полем

Метод `getSource()` возвращает ссылку на объект, явившийся источником события, который преобразуется в объект `JButton`. Метод `getText()` в виде строки извлекает текст, который изображен на кнопке, и помещает его с помощью метода `setText()` объекта `JLabel` в объект `lbl`. При этом определяется, какая из кнопок была нажата.

Для отображения результата нажатия кнопки использован компонент `JTextField`, представляющий собой поле, где может быть размещен и изменен текст. Хотя есть несколько способов создания `JTextField`, самым простым является сообщение конструктору нужной ширины текстового поля. Как только `JTextField` помещается на форму, можно изменять содержимое, используя метод `setText()`. Реализацию действий, ассоциированных с нажатием кнопки, лучше производить в потоке во избежание “зависания”

Класс `JComboBox` применяется для создания раскрывающегося списка альтернативных вариантов, из которых пользователем производится выбор. Таким образом, данный элемент управления имеет форму меню. В неактивном состоянии компонент типа `JComboBox` занимает столько места, чтобы показывать только текущий выбранный элемент. Для определения выбранного элемента можно вызвать метод `getSelectedItem()` или `getSelectedIndex()`. Чтобы сделать элемент редактируемым, следует использовать метод `setEditable(boolean editable)`. Существуют методы по вставке и удалению элементов списка во время выполнения программы `insertItemAt(int pos)` и `removeItemAt(int pos)`.

// пример #7: простой выпадающий список: ComboBoxEx.java

```
package chapt13;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;

public class ComboBoxEx extends JApplet {
    private final int N = 3;
    private JTextField textField = new JTextField(2);
    private JComboBox comboBox = new JComboBox();
    private Map<String, Integer> exams =
        new HashMap<String, Integer>(N);
    private class ComboListener implements ItemListener {
        // реакция на изменение текущего значения ComboBox
        public void itemStateChanged(ItemEvent ev) {
            String name = (String) ev.getItem();
            textField.setText(exams.get(name).toString());
        }
    }
    public void init() {
        exams.put("Программирование", 4);
        exams.put("Алгебра", 7);
        exams.put("Топология", 8);
        // добавление элементов в ComboBox
        Iterator i = exams.entrySet().iterator();
```

```

        while (i.hasNext())
            comboBox.addItem(
                ((Map.Entry) i.next()).getKey());
        comboBox.addItemListener(new ComboListener());
        textField.setText(exams.get(
            comboBox.getSelectedItem()).toString());

        Container c = getContentPane();
        c.setLayout(new FlowLayout());
        c.add(comboBox);
        c.add(textField);
    }
}

```

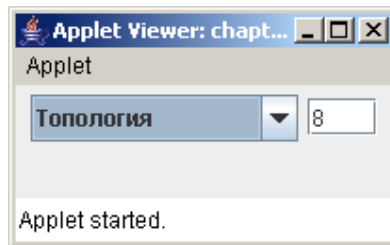


Рис. 13.9. Выпадающий список

При выборе элемента списка генерируется событие **ItemEvent** и посылается всем блокам прослушивания, зарегистрированным для приема уведомлений о событиях данного компонента. Каждый блок прослушивания реализует интерфейс **ItemListener**. Этот интерфейс определяет метод **itemStateChanged()**. Объект **ItemEvent** передается этому методу в качестве аргумента. Приведенная программа позволяет выбрать из списка число, возводит его в квадрат и выводит в объект **JTextField**.

В следующем примере приведен способ по организации многоязычного меню. Здесь также используются возможности класса **ResourceBundle** по извлечению информации из файла свойств (properties). При разработке больших приложений не рекомендуется помещать в код текстовые сообщения, так как при их изменении программисту потребуется корректировать и затем перекомпилировать большую часть приложения.

/ пример # 8 : регистрация, генерация и обработка ActionEvent:*

*ButtonActionDemo.java, Messages.java */*

```

package chapt13;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import java.io.UnsupportedEncodingException;
import java.util.Locale;
import javax.swing.JButton;
import javax.swing.JComboBox;

```

```

import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class ButtonActionDemo extends JFrame {
    private static final String EN_LANGUAGE = "English";
    private static final String RU_LANGUAGE = "Русский";
    private JPanel jContentPane = null;
    private JComboBox languageChooser = null;
    private JButton yesBtn = null;
    private JButton noBtn = null;
    private JLabel jLabel = null;

    public ButtonActionDemo() {
        initialize();
    }
    // ActionListener для кнопки 'Yes'
    private class YesButtonListener
        implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            jLabel.setText(getString("BUTTON_YES_MESSAGE"));
        }
    }
    // ActionListener для кнопки 'No'
    private class NoButtonListener
        implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            jLabel.setText(getString("BUTTON_NO_MESSAGE"));
        }
    }
    // ItemListener для combobox
    private class LanguageChooserItemListener
        implements ItemListener {
        public void itemStateChanged(ItemEvent e) {
            if (((String) e.getItem()).equals(EN_LANGUAGE)) {
                Locale.setDefault(Locale.ENGLISH);
            } else {
                Locale.setDefault(new Locale("RU"));
            }
            yesBtn.setText(getString("BUTTON_YES"));
            noBtn.setText(getString("BUTTON_NO"));
        }
    }
    private void initialize() {
        setSize(230, 200);
        setContentPane(getJContentPane());
        setTitle("JFrame");
        setVisible(true);
    }
}

```

```
private JPanel getJContentPane() {
    if (jContentPane == null) {
        jLabel = new JLabel();
        jLabel.setText("JLabel");
        jContentPane = new JPanel();
        jContentPane.setLayout(new FlowLayout());
    }
    languageChooser = new JComboBox();
    languageChooser.addItem(EN_LANGUAGE);
    languageChooser.addItem(RU_LANGUAGE);
    languageChooser.addItemListener(
        new LanguageChooserItemListener());

    yesBtn = new JButton(getString("BUTTON_YES"));
    yesBtn.addActionListener(
        new YesButtonListener());

    noBtn = new JButton(getString("BUTTON_NO"));
    noBtn.addActionListener(
        new NoButtonListener());

    jContentPane.add(languageChooser);
    jContentPane.add(yesBtn);
    jContentPane.add(noBtn);
    jContentPane.add(jLabel);

    return jContentPane;
}

public static void main(String[] args) {
    Locale.setDefault(Locale.ENGLISH);
    ButtonActionDemo ob = new ButtonActionDemo();
    ob.setDefaultCloseOperation(EXIT_ON_CLOSE);
}

private String getString(String property) {
    String text = "";
    try {
        text = new String(
Messages.getString(property).getBytes(
        "ISO-8859-1"), "CP1251");
    } catch (UnsupportedEncodingException ex) {
        ex.printStackTrace();
    }
    return text;
}

}

package chapt13;
import java.util.MissingResourceException;
import java.util.ResourceBundle;
```

```
public class Messages {
    private static final String BUNDLE_NAME =
        "chapt13.messages";
    private static final ResourceBundle RESOURCE_BUNDLE =
        ResourceBundle.getBundle(BUNDLE_NAME);

    public static String getString(String key) {
        try {
            return RESOURCE_BUNDLE.getString(key);
        } catch (MissingResourceException e) {
            return '!' + key + '!';
        }
    }
}
```

Файлы ресурсов **messages.properties** и **messages_ru.properties**, из которых извлекаются сообщения на английском и русском языках соответственно, выглядят следующим образом:

BUTTON_YES=yes

BUTTON_NO=no

BUTTON_YES_MESSAGE=Button <yes> is pressed

BUTTON_NO_MESSAGE=Button <no> is pressed

BUTTON_YES=да

BUTTON_NO=нет

BUTTON_YES_MESSAGE=Нажата кнопка <да>

BUTTON_NO_MESSAGE=Нажата кнопка <нет>

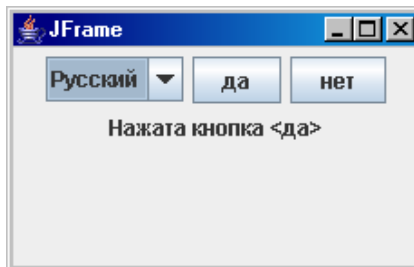


Рис. 13.10. Результат нажатия кнопки «да» отображен в метке

Команда, ассоциированная с кнопкой, возвращается вызовом метода **getActionCommand()** класса **ActionEvent**, экземпляр которого содержит всю информацию о событии и его источнике.

В следующем примере рассмотрен вариант объединения нескольких радиокнопок (**JRadioButton**) в группу и отслеживание изменения их состояния.

/ пример # 9 : отслеживание изменения состояния флажка:*

*RadioBtnGroupEx.java */*

```
package chapt13;
import java.awt.*;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import javax.swing.*;
```

```

public class RadioBtnGroupEx extends JApplet {
    private ButtonGroup btnGroup = new ButtonGroup();
    private JLabel label = null;
    private class RadioItemListener
        implements ItemListener {
        public void itemStateChanged(ItemEvent e) {
            boolean selected =
                (e.getStateChange() == ItemEvent.SELECTED);
            AbstractButton button =
                (AbstractButton) e.getItemSelectable();
            if (selected)
                label.setText("Selected Button: "
                    + button.getActionCommand());
            System.out.println(
                "ITEM Choice Selected: " + selected +
                ", Selection: " + button.getActionCommand());
        }
    }
    public void init() {
        Container c = getContentPane();
        c.setLayout(new FlowLayout());
        JRadioButton red = new JRadioButton("Red");
        red.setSelected(true);
        btnGroup.add(red);
        c.add(red);
        label = new JLabel("Selected Button: Red");
        JRadioButton green = new JRadioButton("Green");
        btnGroup.add(green);
        c.add(green);
        JRadioButton blue = new JRadioButton("Blue");
        btnGroup.add(blue);
        c.add(blue);
        ItemListener itemListener = new RadioItemListener();
        red.addItemListener(itemListener);
        green.addItemListener(itemListener);
        blue.addItemListener(itemListener);
        c.add(label);
    }
}

```

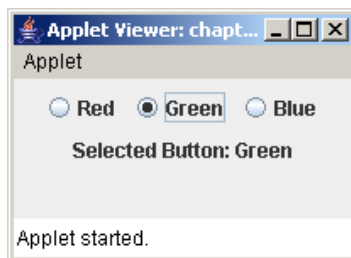


Рис. 13.11. Элемент управления JRadioButton

Ниже приведен простейший вариант использования объекта класса **JSlider**, представляющий собой ползунковый регулятор, позволяющий выбрать значение из интервала возможных значений.

/ пример # 10 : использование ползункового регулятора: SliderEx.java */*

```
package chapt13;
import java.awt.*;
import javax.swing.*;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

public class SliderEx extends JApplet {
    private JLabel sliderValue =
        new JLabel("Slider Value: 25");

    private class SliderListener
        implements ChangeListener {
        public void stateChanged(ChangeEvent e) {
            sliderValue.setText("Slider Value: " +
                ((JSlider) (e.getSource())).getValue());
        }
    }

    public void init() {
        Container c = getContentPane();
        c.setLayout(new FlowLayout());

        JSlider slider =
            new JSlider(JSlider.HORIZONTAL, 0, 50, 25);
        //установка видимости меток и делений
        slider.setPaintLabels(true);
        slider.setPaintTicks(true);
        //установка расстояний между делениями
        slider.setMinorTickSpacing(2);
        slider.setMajorTickSpacing(10);
        slider.setLabelTable(
            slider.createStandardLabels(10));
        slider.addChangeListener(new SliderListener());
        c.add(slider);
        c.add(sliderValue);
    }
}
```

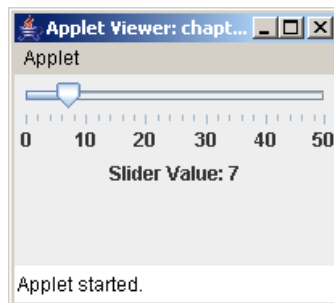


Рис. 13.12. Элемент управления **JSlider**

Блок прослушивания событий в ответ на генерацию объекта события **ItemEvent** вызвал метод **itemStateChanged(ItemEvent e)**, извлекающий из объекта класса **ItemEvent** константу состояния, в данном случае **SELECTED**, а в ответ на генерацию объекта события **ChangeEvent** вызывается метод **stateChanged(ChangeEvent e)**.

Для добавления в приложение различного вида всплывающих меню и диалоговых окон следует использовать обширные возможности класса **JOptionPane**. Эти возможности реализуются статическими методами класса вида **showИмяDialog(параметры)**. Наиболее используемыми являются методы **showConfirmDialog()**, **showMessageDialog()**, **showInputDialog()** и **showOptionDialog()**.

Для подтверждения/отказа выполняемого в родительском окне действия применяется метод **showConfirmDialog()**.

```
/* пример # 11 : диалог Да/Нет: DemoConfirm.java */
package chapt13;
import javax.swing.*;

public class DemoConfirm {
    public static void main(String[] args) {
        int result = JOptionPane.showConfirmDialog(
            null,
            "Хотите продолжить?",
            "Chooser",
            JOptionPane.YES_NO_OPTION);
        if (result == 0)
            System.out.println("You chose Yes");
        else
            System.out.println("You chose No");
    }
}
```

В качестве первого параметра метода указывается окно, к которому относится сообщение, но так как в данном случае здесь и далее используется консоль, то он равен **null**.

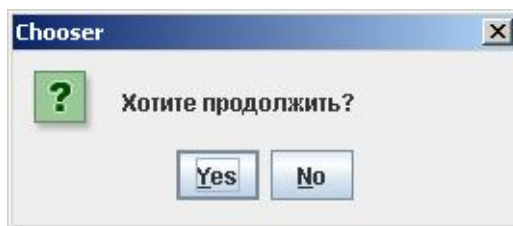


Рис. 13.13. Диалог выбора

Для получения вариаций указанного диалога можно использовать следующие константы: **YES_NO_CANCEL_OPTION**, **OK_CANCEL_OPTION**, **DEFAULT_OPTION** и некоторые другие.

Для показа сообщений (информационных, предупреждающих, вопросительных и т.д.) применяется метод **showMessageDialog()**.

```

/* пример # 12 : сообщение: DemoMessage.java */
package chapt13;
import javax.swing.*;

public class DemoMessage {
    public static void main(String[] args) {
        JOptionPane.showMessageDialog(
            null,
            "Файл может быть удален!",
            "Внимание!",
            JOptionPane.WARNING_MESSAGE);
        // ERROR_MESSAGE – сообщение об ошибке
        // INFORMATION_MESSAGE - информационное сообщение
        // WARNING_MESSAGE - уведомление
        // QUESTION_MESSAGE - вопрос
        // PLAIN_MESSAGE - без иконки
    }
}

```

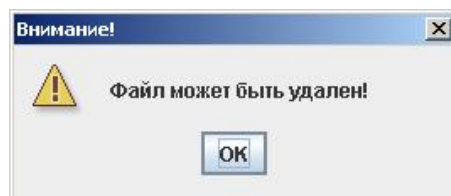


Рис. 13.14. Вывод сообщения

Если необходимо обязать пользователя приложения ввести какую-либо информацию на определенном этапе работы, то следует использовать возможности метода `showInputDialog()`. Причем простейший запрос создать очень легко.

```

/* пример # 13 : запрос на ввод: DemoInput.java */
package chapt13;
import javax.swing.*;

public class DemoInput {
    public static void main(String[] args) {
        String str =
            JOptionPane.showInputDialog(
                "Please input a value");

        if (str != null)
            System.out.println("You input : " + str);
    }
}

```



Рис. 13.15. Простейший запрос на ввод строки

Следующий пример предоставляет возможность выбора из заранее определенного списка значений.

/ пример # 14 : формирование запроса на выбор из списка:*

*DemoInputWithOptions.java */*

```
package chapt13;
import javax.swing.*;

public class DemoInputWithOptions {
    public static void main(String[] args) {
        Object[] possibleValues =
            { "легкий", "средний", "трудный" };
        Object selectedValue =
            JOptionPane.showInputDialog(
                null,
                "Выберите уровень",
                "Input",
                JOptionPane.INFORMATION_MESSAGE,
                null,
                possibleValues,
                possibleValues[0]);
        // possibleValues[1] - элемент для фокуса
        // второй null – иконка по умолчанию
        if (selectedValue != null)
            System.out.println("You input : "
                               + selectedValue);
    }
}
```

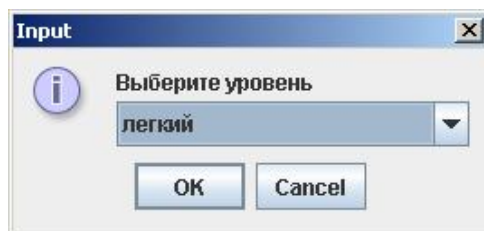


Рис. 13.16. Запрос на выбор из списка

Другим способом создания диалога с пользователем является применение возможностей класса **JDialog**. В этом случае можно создавать диалоги с произвольным набором компонентов.

// пример # 15 : произвольный диалог: MyDialog.java

```
package chapt13;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class MyDialog extends JDialog{
    private final JFrame parent;
    private JCheckBox cb = new JCheckBox();
    private JButton ok = new JButton("Ok");
```

```

    public MyDialog(final JFrame parent, String name) {
        super(parent, name, true);
        this.parent = parent;

        Container c = getContentPane();
        c.setLayout(new FlowLayout());
        c.add(new JLabel("Exit ?"));
        ok.addActionListener(
            new ActionListener() {
                public void actionPerformed(
                    ActionEvent e) {
                    dispose();
                    if (cb.isSelected())
                        parent.dispose();
                }
            });
        c.add(cb);
        c.add(ok);

        setSize(200, 100);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }
}

public class DemoJDialog extends JFrame {
    private JButton jButton = new JButton("Dialog");

    DemoJDialog() {
        super("My DialogFrame");
        Container c = getContentPane();
        c.setLayout(new FlowLayout());
        jButton.addActionListener(
            new ActionListener() {
                public void actionPerformed(
                    ActionEvent e) {
                    JDialog jDialog =
                        new MyDialog(DemoJDialog.this, "MyDialog");
                }
            });
        c.add(jButton);
    }

    public static void main(String[] args) {
        DemoJDialog f = new DemoJDialog();
        f.setDefaultCloseOperation(EXIT_ON_CLOSE);
        f.setSize(200, 120);
        f.setVisible(true);
    }
}

```



Рис. 13.17. Произвольный диалог

Для создания пользовательского меню следует воспользоваться возможностями классов **JMenu**, **JMenuBar** и **JMenuItem**.

/ пример # 16 : создание меню: SimpleMenu.java */*

```
package chapt13;
import javax.swing.*.*;
import java.awt.event.*;

public class SimpleMenu extends JApplet {
    private JMenu menu;
    private JMenuItem item1, item2;

    private class MenuItemListener
        implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JMenuItem jmi =
                (JMenuItem) e.getSource();
            if (jmi.equals(item2))
                System.exit(0);
            else
                showStatus("My Simple Menu");
        }
    }

    public void init() {
        JMenuBar menubar = new JMenuBar();
        setJMenuBar(menubar);
        menu = new JMenu("Main");

        item1 = new JMenuItem("About");
        item2 = new JMenuItem("Exit");
        item1.addActionListener(
            new MenuItemListener());
        item2.addActionListener(
            new MenuItemListener());

        menu.add(item1);
        menu.add(item2);
        menubar.add(menu);
    }
}
```

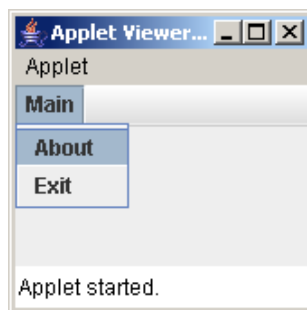


Рис. 13.18. Простейшее меню

Класс **JTextArea** позволяет вводить и отображать многострочную информацию. Такая возможность полезна при обработке текстов, двумерных массивов. Пользователь может сам вводить в этот объект информацию, разделяя строки нажатием клавиши <Enter>. Этот объект не имеет полосы прокрутки (ее можно добавить), поэтому если введено строк больше, чем объявлено, но лишний текст будет утрачен. Класс **JTextArea** является подклассом **JTextComponent**, следовательно, программисту доступны все его возможности. В частности, есть возможность обработки части текста, выделенной пользователем, а также определение количества строк и столбцов (см. рис. 13.19).

В примере решается простая задачи загрузки из файла в объект **JTextArea** двумерного массива и его примитивная обработка. При загрузке матрицы из файла для поиска соответствующего дискового файла использованы возможности класса **JFileChooser**.

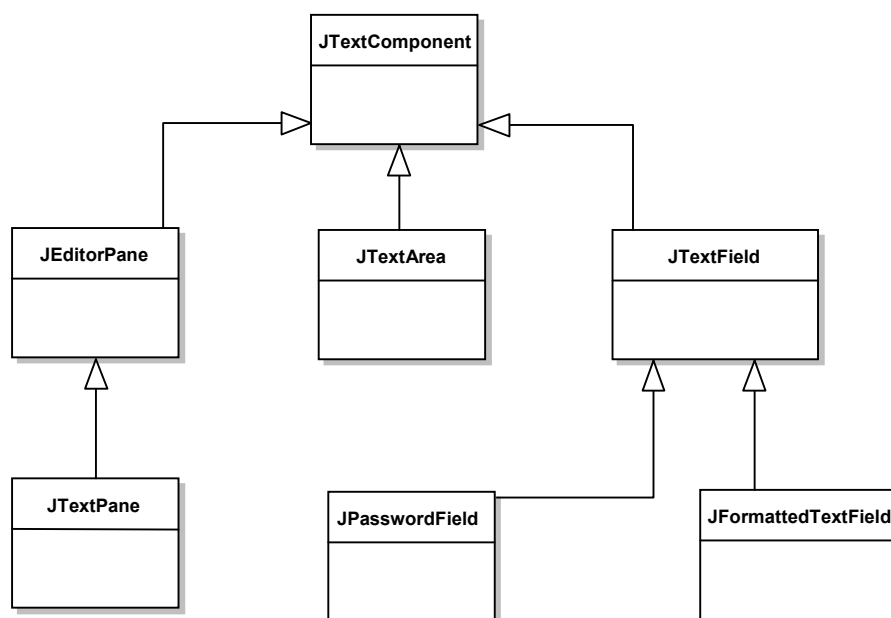


Рис. 13.19. Классы текстовых компонентов

```

/* пример # 17 : вывод двумерного массива в текстовое поле: ArraysWork.java */
package chapt13;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.swing.*;
import javax.swing.text.BadLocationException;
import javax.swing.plaf.metal.MetalBorders.TextFieldBorder;

public class ArraysWork extends JFrame {
    private JFileChooser fileChooser =
        new JFileChooser("D:\\");
    private JButton openBtn = new JButton("Open");
    private JButton resultBtn = new JButton("Result");
    private JButton clearBtn = new JButton("Clear");
    private JTextArea textArea = new JTextArea();
    private ArrayList<int[]> arrays =
        new ArrayList<int[]>();

    private ArraysWork() {
        Container contentPane = getContentPane();
        contentPane.setLayout(null);
        JPanel col = new JPanel();
        col.setBounds(10, 10, 90, 110);
        col.setLayout(new GridLayout(3, 1, 0, 8));
        openBtn.addActionListener(
            new ButtonListener());
        resultBtn.addActionListener(
            new ButtonListener());
        clearBtn.addActionListener(
            new ButtonListener());

        col.add(openBtn);
        col.add(resultBtn);
        col.add(clearBtn);
        contentPane.add(col, BorderLayout.EAST);
        textArea.setBounds(130, 10, 110, 110);
        textArea.setBorder(new TextFieldBorder());
        textArea.setFont(
            new Font("Courier New", Font.PLAIN, 12));
        contentPane.add(textArea);
    }

    private class ButtonListener
        implements ActionListener {
        public void actionPerformed(ActionEvent event)
        {
            if (event.getSource() == clearBtn)
                textArea.setText("");
            else if (event.getSource() == resultBtn)
                viewArrays(true);
        }
    }
}

```



```

        else if (event.getSource() == openBtn)
            readArraysFromFile();
    }
}

public static void main(String[] args) {
    ArraysWork frame = new ArraysWork();

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setBounds(200, 100, 280, 155);
    frame.setVisible(true);
}

private void readArraysFromTextArea() {
    arrays.clear();
    // обработка строк текстового поля
    for (int i = 0; i < textArea.getLineCount(); i++)
        try {
            int startOffset = textArea.getLineStartOffset(i);
            int endOffset = textArea.getLineEndOffset(i);
            StringTokenizer str =
                new StringTokenizer(textArea.getText(
                    startOffset, endOffset - startOffset), " \n");
            arrays.add(new int[str.countTokens()]);
            int k = 0;
            while (str.hasMoreTokens())
                arrays.get(i)[k++] =
                    new Integer(str.nextToken()).intValue();
        } catch (BadLocationException e) {
            e.printStackTrace();
        } catch (NumberFormatException e) {
            JOptionPane.showMessageDialog(
                null, "Enter only numbers!",
                "Fatal error!", JOptionPane.ERROR_MESSAGE);
        }
}

private void readArraysFromFile() {
    String filePath = null;
    int rVal =
        fileChooser.showOpenDialog(fileChooser);
    fileChooser.setVisible(true);
    if (JFileChooser.APPROVE_OPTION == rVal)
        filePath = fileChooser
            .getSelectedFile().getPath();
    if (filePath == null)
        return;
    try {
        Scanner sc = new Scanner(
            new File(filePath));
        int i = 0;
    }
}

```

```

        while (sc.hasNextLine()) {
            StringTokenizer str =
                new StringTokenizer(sc.nextLine());
            arrays.add(new int[str.countTokens()]);
            int k = 0;
            while (str.hasMoreTokens())
                arrays.get(i)[k++] =
                    new Integer(str.nextToken()).intValue();
            i++;
        }
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
    viewArrays(false);
}

private void viewArrays(boolean result) {
    if (result) {
        readArraysFromTextArea();
        handlingOfArrays();
    }
    textArea.setText("");
    for (int i = 0; i < arrays.size(); i++) {
        Formatter fmt = new Formatter();
        for (int j = 0; j < arrays.get(i).length; j++)
            fmt.format("%4d", arrays.get(i)[j]);
        textArea.append(fmt.toString() + '\n');
    }
}

void handlingOfArrays() {
    for (int i = 0; i < arrays.size(); i++)
        try {
            arrays.get(i)[0] = -arrays.get(i)[0];
        } catch (ArrayIndexOutOfBoundsException e) {
            throw new AssertionError("не матрица!");
        }
}
}

```

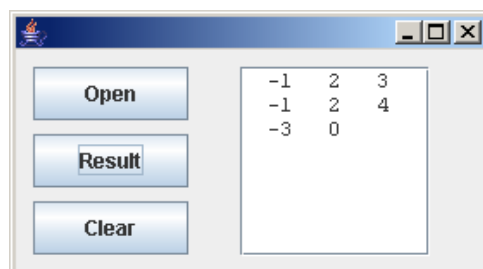


Рис. 13.20. Текстовое поле

Добавить полосу прокрутки можно с помощью следующего кода:

```
JScrollPane sp = new JScrollPane(textarea);
contentPane.add(sp, BorderLayout.CENTER);
```

Компонент **JTable** предназначен для отображения таблицы. Таблицы применяются для отображения связанной информации, например: ФИО, Дата рождения, Адрес, Образование и т. д. Класс **JTable** объявляет конструктор, добавляющий двумерную модель объектов в модель. В примере рассмотрен процесс создания, заполнения и редактирования простой таблицы из двух столбцов.

/ пример # 18: простая таблица: JTableDemo.java */*

```
package chapt13;
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayList;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.table.AbstractTableModel;

public class JTableDemo extends JFrame {

    static ArrayList<Object[]> list =
        new ArrayList<Object[]>();
    JTable table;
    DataModel dataModel = new DataModel();

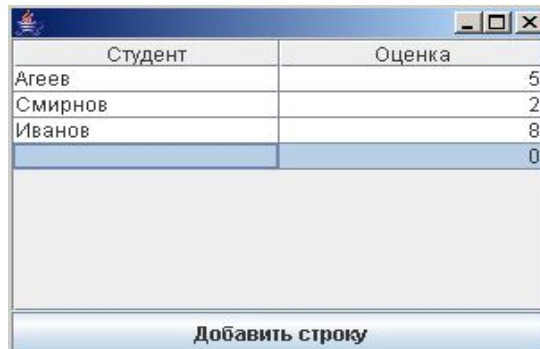
    JTableDemo() {
        dataModel.addRow("Агеев", 5);
        dataModel.addRow("Смирнов", 2);
        table = new JTable(dataModel);
        getContentPane().add(new JScrollPane(table));
        JButton insertBtn = new JButton("Добавить строку");
        insertBtn.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                dataModel.addRow("", 0);
            }
        });
        getContentPane().add(insertBtn, BorderLayout.SOUTH);
    }

    public static void main(String[] args) {
        JTableDemo frame = new JTableDemo();
        frame.setBounds(100, 100, 300, 300);
        frame.setDefaultCloseOperation(EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

```
class DataModel extends AbstractTableModel {

    private final String[] HEADERS =
        {"Студент", "Оценка"};
    private final int COLS = 2;
    private int rows = 0;

    public String getColumnName(int col) {
        return HEADERS[col];
    }
    public int getRowCount() {
        return rows;
    }
    public int getColumnCount() {
        return COLS;
    }
    public Object getValueAt(int row, int col) {
        return JTableDemo.list.get(row)[col];
    }
    public void setValueAt(Object val, int row, int col) {
        if (col == 1){
            int mark = new Integer(val.toString());
            if (mark > 10 || mark < 0){
                JOptionPane.showMessageDialog(null, "Введите число от 0 до
                10", "Ошибка ввода!", JOptionPane.ERROR_MESSAGE);
                return;
            }
            JTableDemo.list.get(row)[col] = val.toString();
            fireTableDataChanged(); //обновление таблицы после изменений
        }
        public boolean isCellEditable(int row, int col) {
            return true;
        }
        public void addRow(String name, int mark) {
            JTableDemo.list.add(new String[COLS]);
            setValueAt(name, rows, 0);
            setValueAt(mark, rows, 1);
            rows++;
        }
        public Class getColumnClass(int col) {
            switch (col) {
                case 0: return String.class;
                case 1: return Integer.class;
            }
            return null;
        }
    }
}
```



Студент	Оценка
Агеев	5
Смирнов	2
Иванов	8
	0

Добавить строку

Рис. 13.21. Объект `JTable`

При попытке добавления в поле «Оценка» значений, выходящих за границы диапазона, генерируется исключительная ситуация, а при попытке внесения нецифровых символов поле подсвечивается красным цветом и добавление не производится.

Компонент **`JSplitPane`** показывает панель, разделенную на две области либо по вертикали, либо по горизонтали. Разграничительную полосу между ними можно перетаскивать, распределяя долю видимой площади для каждой области. Следующий пример показывает простейшее использование контейнерного класса **`JSplitPane`**. В верхней части содержится текстовое окно, а в нижней – текстовое поле.

/ пример # 19 : простое окно с разграничительной полосой: DemoJSplit.java */*

```
package chapt13;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JSplitPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class DemoJSplit extends JFrame {
    private JSplitPane tabs;
    private JTextArea area;
    private JPanel tab;

    public DemoJSplit() {
        tabs = new JSplitPane(JSplitPane.VERTICAL_SPLIT);
        area =
            new JTextArea("Текстовое окно в верхней области");
        //установка текстового окна в верхнюю область
        tabs.setTopComponent(area);
        tab = new JPanel();
        tab.add(new JTextField("Текстовое поле в нижней области"));
        //установка текстового поля в нижнюю область
        tabs.setBottomComponent(tab);
        tabs.setDividerLocation(130);
        setContentPane(tabs); //установка в окно фрейма компоненты tabs
    }
}
```

```

public static void main(String[] args) {
    DemoJSplit dspl = new DemoJSplit();
    dspl.setBounds(200, 200, 250, 200);
    dspl.setVisible(true);
}
}

```

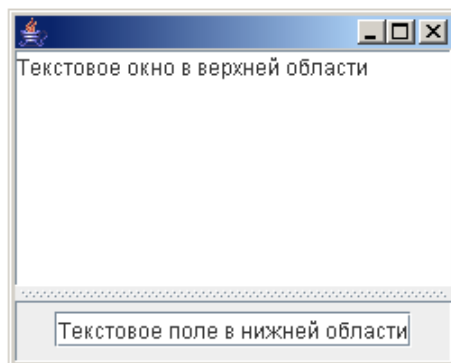


Рис. 13.22. Объект JSplitPane

Контейнерный класс **JTabbedPane** представляет собой окно с закладками, с помощью которых можно выбирать желаемый компонент. Закладки могут иметь всплывающие подсказки, а также содержать как текст, так и значки.

В следующем примере показано простое приложение, использующее окно с закладками.

/ пример # 20 : простое окно с двумя закладками: DemoJTabbed.java */*

```

package chapt13;
import java.awt.Container;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTabbedPane;
import javax.swing.JToggleButton;

public class DemoJTabbed extends JFrame {
    JTabbedPane tabs;
    JPanel pan1, pan2;
    public DemoJTabbed() {
        Container c = getContentPane();
        tabs = new JTabbedPane();
        pan1 = new JPanel();
        pan1.add(new JToggleButton("Button"));
        tabs.addTab("One", pan1); //добавление первой закладки
        pan2 = new JPanel();
        pan2.add(new JCheckBox("CheckBox"));
        tabs.addTab("Two", pan2); //добавление второй закладки
        c.add(tabs);
    }
}

```

```

    public static void main(String[] args) {
        DemoJTabbed dt = new DemoJTabbed();
        dt.setSize(250, 150);
        dt.setLocation(200, 200);
        dt.setVisible(true);
    }
}

```

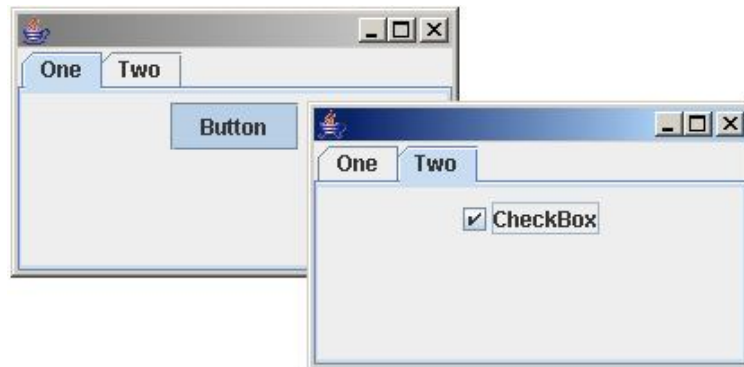


Рис. 13.23. Окно с закладками

В Java 6 стало больше графических возможностей по взаимодействию с рабочим столом Windows и интернетом. Класс `java.awt.SystemTray` обеспечивает прямой доступ к рабочему столу пользователя: позволяет добавлять иконки, советы для кнопок и спадающие меню к системным областям состояниям Windows.

/ пример # 21 : демонстрация SystemTray: DemoTray.java */*

```

package chapt13;
import java.awt.*;

public class DemoTray {
    public static void main(String[] args) {
        if(SystemTray.isSupported()) {
            SystemTray tray = SystemTray.getSystemTray();
            Image image =
                Toolkit.getDefaultToolkit().getImage("icon");
            TrayIcon icon = new TrayIcon(image, "Demo Tray");
            try {
                tray.add(icon);
            } catch (AWTException e) {
                System.err.println(e);
            }
        } else {
            System.err.println("Without system tray!");
        }
    }
}

```

Возможности класса `java.awt.Desktop` позволяют запустить браузер с помощью одной строки:

```
/* пример # 22 : запуск браузера: DesktopTest.java */
package chapt13;
import org.jdesktop.jdic.desktop.*;
import java.net.*;

public class DesktopTest {
    public static void main(String[] args) throws Exception {
        Desktop.browse(new URL("http://java.sun.com"));
    }
}
```

При необходимости можно легко использовать веб-браузер в приложении:

```
/* пример # 23 : использование web- браузера: BrowserTest.java */
package chapt13;
import org.jdesktop.jdic.browser.WebBrowser;
import java.net.URL;
import javax.swing.JFrame;

public class BrowserTest {
    public static void main(String[] args) throws Exception {
        WebBrowser browser = new WebBrowser();
        browser.setURL(new URL("http://www.netbeans.org"));
        JFrame frame = new JFrame("Использование Browser");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(browser);
        frame.pack();
        frame.setSize(640, 480);
        frame.setVisible(true);
    }
}
```

Визуальные компоненты JavaBeans

Визуальные компоненты являются классами языка Java, объекты которых отображаются визуально в проектируемом приложении с помощью средств визуальной разработки. Визуальные компоненты являются удобным средством при создании пользовательских интерфейсов. Обычно часть визуальной разработки приложений состоит в перетаскивании компонентов на форму и в определении их свойств, событий и обработчиков событий. При этом компонент представляет собой объект класса, для которого, кроме данных и методов, дополнительно установлены свойства и события класса. Свойства и события устанавливаются через имена методов на основе соглашения об именовании методов.

JavaBeans (бин) – многократно используемый программный компонент, которым можно манипулировать визуально при создании приложения. Бин реализуется в одном или нескольких взаимосвязанных классах. Основной класс бина должен иметь конструктор по умолчанию, который вызывается визуальной средой при создании экземпляра бина.

Каждый бин должен поддерживать следующие возможности:

- интроспекцию, позволяющую средам разработки анализировать, из чего состоит и как работает данный бин;
- поддержку событий (**events**);
- поддержку свойств (**properties**);
- сохраняемость (**persistence**).

Каждый бин должен быть сериализуемым. Визуальная среда при сохранении скомпилированного приложения сохраняет настройки компонента, сделанные пользователем в процессе разработки приложения путем сериализации бина. При повторной загрузке приложения эти настройки восстанавливаются. Для этого среда разработки десериализует бины из файла.

Свойства бинов

Каждый бин имеет свойства (**properties**), которые определяют, как он будет работать и как выглядеть. Эти свойства являются **private** или **protected** полями класса, которые доступны через специальные методы **getИмя()** и **setИмя()** (getters и setters), называемые также аксессорами. Так, утверждение “данный бин имеет свойство **name** типа **String**” означает, что у этого бина

```
//есть поле
    private String name;
//есть get-метод
    public String getName() {
        return name;
    }
//есть set-метод
    public void setString(String name) {
        this.name = name;
    }
```

Пусть рассматривается, к примеру, компонент **JLabel**. Во-первых, **JLabel** удовлетворяет интерфейсу **Serializable**, во-вторых, имеет конструктор по умолчанию **public JLabel()** и, в-третьих, имеет ряд методов аксессоров, например **public String getText()**, **public void setText(String text)**. Исходя из этого, можно сделать выводы, что **JLabel** является бином, имеющим свойство **text**. Значение этого свойства отображается как заголовок метки. Кроме того, **JLabel** имеет и другие свойства.

Для свойства типа **boolean** в бинах вместо **get**-методов может быть использован **is**-метод. Например, **JLabel** имеет **boolean**-свойство **enabled**, унаследованное от класса **Component**. Для доступа к этому свойству имеются методы

```
public boolean isEnabled()
public void setEnabled(boolean b)
```

Правила построения методов доступа к атрибутам (аксессоров):

```
public void setИмяСвойства (ТипСвойства value);
public ТипСвойства getИмяСвойства ();
public boolean isИмяСвойства ().
```

Свойства бинтов могут быть как базовых типов, так и объектными ссылками. Свойства могут быть индексированными, если атрибут бина массив. Для индексированных свойств выработаны следующие правила. Они должны быть описаны как поля-массивы, например

```
private String[] messages;
```

и должны быть объявлены следующие методы:

```
public ТипСвойства getИмяСвойства(int index);
public void setИмяСвойства(int index, ТипСвойства value);
public ТипСвойства [] getИмяСвойства();
public void setИмяСвойства(ТипСвойства [] value).
```

Так, для приведенного выше примера должны быть методы

```
public String getMessages(int index);
public void setMessages(int index, String message);
public String[] getMessages();
public void setMessages(String[] messages).
```

Кроме аксессоров, бин может иметь любое количество других методов.

Интроеспекция бинтов при помощи Reflection API

Под интроеспекцией понимается процесс анализа bean-компонента для установления его возможностей, свойств и методов. Для интроеспекции можно воспользоваться классами и методами из библиотеки Reflection API. При использовании бина визуальная среда должна знать полное имя класса бина. По строковому имени класса статический метод **forName(String className)** класса **java.lang.Class** возвращает объект класса **Class**, соответствующий данному бину. Далее с помощью метода класса **Class** **getField()**, **getMethods()**, **getConstructors()** можно получить необходимую информацию о свойствах и событиях класса.

В частности, можно получить список всех **public**-методов данного класса. Исследуя их имена, можно выделить из них аксессоры и определить какие атрибуты (свойства) есть у данного бина и какого они типа. Все остальные методы, не распознанные как аксессоры, являются bean-методами.

В результате соответствующая визуальная разработки может построить диалог, в котором будет предоставлена возможность задавать значения этих атрибутов. Наличие конструктора по умолчанию позволяет построить объект bean-класса, **set**-методы позволяют установить в этом объекте значения атрибутов, введенные пользователем, а благодаря сериализации объект с заданными атрибутами можно сохранить в файл и восстановить значение объекта при следующем сеансе работы с данной визуальной средой. Более того, можно изобразить на экране внешний вид бина (если это визуальный бин) в процессе разработки и менять этот вид в соответствии с задаваемыми пользователем значениями атрибутов.

События

Еще одним важным аспектом технологии JavaBeans является возможность бинтов взаимодействовать с другими объектами, в частности, с другими бинами. JavaBeans реализует такое взаимодействие путем генерации и прослушивания событий.

В приложении к бинам взаимодействие объектов с бином через событийную модель выглядит так. Объект, который интересуется тем, что может произойти во внешнем по отношению к нему бине, может зарегистрировать себя как слушателя (**Listener**) этого бина. В результате при возникновении соответствующего события в бине будет вызван определенный метод данного объекта, которому в качестве параметра будет передан объект-событие (**event**). Причем если зарегистрировалось несколько слушателей, то эти методы будут последовательно вызваны для каждого слушателя.

Такой механизм взаимодействия является очень гибким, поскольку два объекта – бин и его слушатель – связаны только посредством данного метода и параметра-события.

Одним из способов экспорта событий является использование связанных свойств. Когда значение связанного свойства меняется, генерируется событие и передается всем зарегистрированным слушателям посредством вызова метода **propertyChange()**.

Создание и использование связанного свойства

Разберемся практически, как создавать и использовать связанные свойства. Начнем с события, которое должно быть сгенерировано при изменении связанного свойства. Это событие класса **java.beans.PropertyChangeEvent** (см. документацию).

Далее можно действовать по следующей инструкции.

1. Для регистрации/дерегистрации слушателя необходимо в бине реализовать два метода:

```
addPropertyChangeListener(PropertyChangeListener p) и removePropertyChangeListener(PropertyChangeListener p);
```

2. Чтобы не реализовывать их вручную, лучше воспользоваться существующим классом **java.beans.PropertyChangeSupport** (см. документацию);

3. В **set**-методе связанного свойства необходимо добавить вызов метода **firePropertyChange()** класса **java.beans.PropertyChangeSupport**;

4. В классе-слушателе реализовать интерфейс **PropertyChangeListener**, т.е. в заголовке класса записать “**implements PropertyChangeListener**”, а в теле класса реализовать метод **public void propertyChange(PropertyChangeEvent evt);**

5. Создать объект-слушатель и зарегистрировать его как слушателя нашего бина при помощи метода **addPropertyChangeListener()**, который был реализован в п.1. Лучше всего это сделать сразу после порождения объекта-слушателя, например:

```
MyListener obj = new MyListener();  
myBean.addPropertyChangeListener(obj);
```

где **myBean** – создаваемый бин (имеется в виду объект, а не класс).

Пункт 4-й должен быть реализован для каждого класса-слушателя, а п.5 – для каждого порожденного объекта-слушателя.

Следует разобрать подробнее пункты 2 и 3.

Сейчас необходимо реализовать генерацию событий. Бин должен генерировать событие **PropertyChangeEvent** при изменении связанного свойства (п.3). Кроме того, согласно правилам событийной модели Java он

должен обеспечивать регистрацию/дерегистрацию слушателей при помощи соответствующих методов **add...Listener/remove...Listener** (п.2).

Т.е. нужно обеспечить наличие в бине некоторого списка слушателей, а также методы **addPropertyChangeListener()** и **removePropertyChangeListener()**.

К счастью, не требуется программировать все это. Соответствующий инструментарий уже подготовлен в пакете **java.beans** – это класс **java.beans.PropertyChangeSupport**. Он обеспечивает регистрацию слушателей и методы **firePropertyChange()**, которые можно использовать в тех местах, где требуется сгенерировать событие, т.е. в **set**-методах, которые изменяют значение связанных атрибутов.

Предложенный механизм будет рассмотрен в следующем примере.

Пусть имеется некоторый бин **SomeBean** с одним свойством **someProperty**:

```
/* пример # 24 : простой bean-класс : SomeBean.java */
package chapt13;
public class SomeBean{
    private String someProperty = null;
    public SomeBean() {
    }
    public String getSomeProperty() {
        return someProperty;
    }
    public void setSomeProperty(String value) {
        someProperty = value;
    }
}
```

Переделаем его так, чтобы свойство **someProperty** стало связанным:

```
/* пример # 25 : bean-класс со связанным свойством: SomeBean.java */
import java.beans.*;
public class SomeBean{
    private String someProperty = null;
    private PropertyChangeSupport pcs;
    public SomeBean() {
        pcs = new PropertyChangeSupport(this);
    }
    public void addPropertyChangeListener
        (PropertyChangeListener pcl) {
        pcs.addPropertyChangeListener(pcl);
    }
    public void removePropertyChangeListener
        (PropertyChangeListener pcl) {
        pcs.removePropertyChangeListener(pcl);
    }
    public String getSomeProperty() {
        return someProperty;
    }
}
```

```

        public void setSomeProperty(String value) {
            pcs.firePropertyChange("someProperty",
                someProperty, value);
            someProperty = value;
        }
    }

```

Здесь реализованы пункты 1, 2 и 3 приведенной инструкции. Остальные пункты относятся к использованию связанного свойства, и для их демонстрации потребуется более реальный пример.

Для обеспечения механизма генерации событий в классе **SomeBean** создан объект класса **PropertyChangeSupport** (поле **pcs**). И все действия по регистрации/дерегистрации слушателей по собственно генерации событий “переадресуются” этому объекту, который за нас выполняет всю эту рутинную работу.

Так, например, метод **addPropertyChangeListener(PropertyChangeListener pcl)** созданного класса просто обращается к одноименному методу класса **PropertyChangeSupport**. В методе **setSomeProperty()** перед собственно изменением значения свойства **someProperty** генерируется событие **PropertyChangeEvent**. Для этого вызывается метод **firePropertyChange()**, который обеспечивает все необходимые для такой генерации действия.

Как видно из кода примера, результат не очень громоздкий, несмотря на то, что наш бин реализует достаточно сложное поведение.

Ограниченные свойства (contrained properties)

Кроме понятия связанных свойств, в JavaBeans есть понятие ограниченных свойств (contrained properties). Ограниченные свойства введены для того, чтобы была возможность запретить изменение свойства бина, если это необходимо. Т.е. бин будет как бы спрашивать разрешения у зарегистрированных слушателей на изменение данного свойства. В случае если слушатель не разрешает ему менять свойство, он генерирует исключение **PropertyVetoException**. Соответственно **set**-метод для ограниченного свойства должен иметь в своем описании **throws PropertyVetoException**, что заставляет перехватывать это исключение в точке вызова данного **set**-метода. В результате прикладная программа, использующая этот бин, будет извещена, что ограниченное свойство не было изменено.

В остальном ограниченные свойства очень похожи на связанные свойства. Как и все свойства, они имеют **get**- и **set**-методы. Но для них **set**-методы могут генерировать исключение **PropertyVetoException** и имеют вид **public void <PropertyName>(ТипСвойства param) throws PropertyVetoException**.

Второе отличие заключается в именах методов для регистрации/дерегистрации слушателей. Вместо методов

```

addPropertyChangeListener() и
removePropertyChangeListener()

```

для ограниченных свойств применяются методы

```

addVetoableChangeListener(VetoableChangeListener v) и

```

removeVetoableChangeListener(VetoableChangeListener v). Здесь **VetoableChangeListener** – интерфейс с одним методом **void vetoableChange(PropertyChangeEvent evt) throws PropertyVetoException()**.

По аналогии со вспомогательным классом **PropertyChangeSupport**, который используется при реализации связанных свойств, для ограниченных свойств в пакете **java.beans** есть вспомогательный класс **VetoableChangeSupport**. В нем реализованы алгоритмы, необходимые для поддержки событий ограниченных свойств.

В качестве примера вспомним класс **SomeBean**, рассмотренный ранее. Его свойство **someProperty()** реализовано как связанное. Переделаем пример и реализуем это свойство как ограниченное.

/ пример # 26 : bean-класс с ограниченным свойством : SomeBean.java */*

```
import java.beans.*;

public class SomeBean {
    private String someProperty = null;
    private VetoableChangeSupport vcs;
    public SomeBean() {
        vcs = new VetoableChangeSupport(this);
    }
    public void addVetoableChangeListener
        (VetoableChangeListener pcl) {
        vcs.addVetoableChangeListener(pcl);
    }
    public void removeVetoableChangeListener
        (VetoableChangeListener pcl) {
        vcs.removePropertyChangeListener(pcl);
    }

    public String getSomeProperty() {
        return someProperty;
    }
    public void setSomeProperty(String value)
        throws
        PropertyVetoException {
        vcs.fireVetoableChange("someProperty",
            someProperty, value);
        someProperty = value;
    }
}
```

Как видно, принципиально ничего не изменилось. Только вместо **PropertyChangeSupport** использован **VetoableChangeSupport** и в описании **set**-метода добавлено **throws PropertyVetoException**. Теперь **someProperty** является ограниченным свойством, и зарегистрировавшийся слушатель может запретить его изменение.

Рассмотренные возможности организации связи бина с другими компонентами не являются единственно возможными. Бин, как и любой класс,

может быть источником событий и/или слушателем. И эти события могут быть не связаны с изменением свойств бина.

В таких случаях обычно используют существующие события типа **ActionEvent**, хотя можно построить и свои события.

Задания к главе 13

Вариант А

1. Создать апплет. Поместить на него текстовое поле **JTextField**, кнопку **Button** и метку **JLabel**. В метке отображать все введенные символы, разделяя их пробелами.
2. Поместить в апплет две панели **JPanel** и кнопку. Первая панель содержит поле ввода и метку “Поле ввода”; вторая – поле вывода и метку “Поле вывода”. Для размещения в окне двух панелей и кнопки “Скопировать” использовать менеджер размещения **BorderLayout**.
3. Изменить задачу 2 так, чтобы при нажатии на кнопку “Скопировать” текст из поля ввода переносился в поле вывода, а поле ввода очищалось.
4. Задача 2 модифицируется так, что при копировании поля ввода нужно, кроме собственно копирования, организовать занесение строки из поля ввода во внутренний список. При решении использовать коллекцию, в частности **ArrayList**.
5. К условию задачи 2 добавляется еще одна кнопка с надписью “Печать”. При нажатии на данную кнопку весь сохраненный список должен быть выведен в консоль. При решении использовать коллекцию, в частности **TreeSet**.
6. Написать программу для построения таблицы значений функции $y = a\sqrt{x} \cdot \cos(ax)$. Использовать метку **JLabel**, содержащую текст “Функция: $y = a\sqrt{x} \cdot \cos(ax)$ ”; панель, включающую три текстовых поля **JTextField**, содержащих значения параметра, шага (например, 0.1) и количества точек. Начальное значение $x=0$. С каждым текстовым полем связана метка, содержащая его название. В приложении должно находиться текстовое поле со скроллингом, содержащее полученную таблицу.
7. Создать форму с набором кнопок так, чтобы надпись на первой кнопке при ее нажатии передавалась на следующую, и т.д.
8. Создать форму с выпадающим списком так, чтобы при выборе элемента списка на экране появлялись GIF-изображения,двигающиеся в случайно выбранном направлении по апплету.
9. В апплете изобразить прямоугольник (окружность, эллипс, линию). Направление движения объекта по экрану изменяется на противоположное щелчком по клавише мыши. При этом каждый второй щелчок меняет цвет фона.
10. Создать фрейм с изображением окружности. Длина дуги окружности изменяется нажатием клавиш от 1 до 9.

11. Создать фрейм с кнопками. Кнопки “вверх”, “вниз”, “вправо”, “влево” двигают в соответствующем направлении линию. При достижении границ фрейма линия появляется с противоположной стороны.
12. Создать фрейм и разместить на нем окружность (одну или несколько). Объект должен “убегать” от указателя мыши. При приближении на некоторое расстояние объект появляется в другом месте фрейма.
13. Создать фрейм/апплет с изображением графического объекта. Объект на экране движется к указателю мыши, когда последний находится в границах фрейма/апплета.
14. Изменить задачу 12 так, чтобы количество объектов зависело от размеров апплета и изменялось при “перетягивании” границы в любом направлении.
15. Промоделировать в апплете вращение спутника вокруг планеты по эллиптической орбите. Когда спутник скрывается за планетой, то он не виден.
16. Промоделировать в апплете аналоговые часы (со стрелками) с кнопками для увеличения/уменьшения времени на час/минуту.

Вариант В

Для заданий варианта В главы 4 создать графический интерфейс для занесения информации при инициализации объекта класса, для выполнения действий, предусмотренных заданием, и для отправки сообщений другому пользователю системы.

Тестовые задания к главе 13

Вопрос 13.1.

Какой менеджер размещения использует таблицу с ячейками равного размера?

- 1) FlowLayout;
- 2) GridLayout;
- 3) BorderLayout;
- 4) CardLayout.

Вопрос 13.2.

Дан код:

```
import java.awt.*;
public class Quest2 extends Frame{
    Quest2() {
        Button yes = new Button("YES");
        Button no = new Button("NO");
        add(yes);
        add(no);
        setSize(100, 100);
        setVisible(true);
    }
    public static void main(String[] args){
        Quest2 q = new Quest2();
    } }
```


В результате будет выведено:

- 1) две кнопки, занимающие весь фрейм, YES – слева и NO – справа;
- 2) одна кнопка YES, занимающая целый фрейм;
- 3) одна кнопка NO, занимающая целый фрейм;
- 4) две кнопки наверху фрейма – YES и NO.

Вопрос 13.3.

Какое выравнивание устанавливается по умолчанию для менеджера размещения **FlowLayout**?

- 1) `FlowLayout.RIGHT`;
- 2) `FlowLayout.LEFT`;
- 3) `FlowLayout.CENTER`;
- 4) `FlowLayout.LEADING`;
- 5) указывается явно.

Вопрос 13.4.

Сколько кнопок будет размещено в приведенном ниже апплете?

```
import java.awt.*;
public class Quest4 extends java.applet.Applet{
    Button b = new Button("YES");
    public void init(){
        add(b);
        add(b);
        add(new Button("NO"));
        add(new Button("NO"));
    }
}
```

- 1) одна кнопка с YES и одна кнопка NO;
- 2) одна кнопка с YES и две кнопки NO;
- 3) две кнопки с YES и одна кнопка NO;
- 4) две кнопки с YES и две кнопки NO.

Вопрос 13.5.

Объект **JCheckBox** объявлен следующим образом:

```
JCheckBox ob = new JCheckBox();
```

Какая из следующих команд зарегистрирует его в блоке прослушивания событий?

- 1) `ob.addItemListener()`;
- 2) `ob.addItemListener(this)`;
- 3) `addItemListener(this)`;
- 4) `addItemListener()`;
- 5) ни одна из приведенных.»