**ЛАБОРАТОРНАЯ РАБОТА №5**

**ПОЛЬЗОВАТЕЛЬСКИЙ ИНТЕРФЕЙС**

**Цель**: проектирование простого графического интерфейса пользователя c добавлением основных элементов управления библиотеки Swing (либо JavaFX), реализация действий к соответствующим элементам.

Современные программы нуждаются в графическом интерфейсе пользователя (GUI). Пользователи отвыкли работать через консоль: они управляют программой и вводят входные данные посредством так называемых элементов управления (в программировании их также называют визуальными компонентами), к которым относятся кнопки(JButton), текстовые поля(JTextField), выпадающие списки(JComboBox) и т.д.

В Java есть три библиотеки визуальных компонентов для создания графического интерфейса пользователя. Самая ранняя из них называется AWT. Считается, что при ее проектировании был допущен ряд недочетов, вследствие которых с ней довольно сложно работать. Библиотека Swing разработана на базе AWT и заменяет большинство ее компонентов своими, спроектированными более тщательно и удобно. Третья, самая новая библиотека, называется SWT.

Каждая библиотека предоставляет набор классов для работы с кнопками, списками, окнами, меню и т.д., но эти классы спроектированы по-разному: они имеют различный набор методов с разными параметрами, поэтому «перевести» программу с одной библиотеки на другую (например, с целью увеличения быстродействия) не так-то просто.

Каждая GUI-программа запускается в окне и по ходу работы может открывать несколько дополнительных окон.

В библиотеке Swing описан класс JFrame, представляющий собой окно с рамкой и строкой заголовка (с кнопками «Свернуть», «Во весь экран» и «Закрыть»). Оно может изменять размеры и перемещаться по экрану.

Конструктор JFrame() без параметров создает пустое окно. Конструктор JFrame(String title) создает пустое окно с заголовком title.

Чтобы написать простейшую программу, выводящую на экран пустое окно, нам потребуется еще три метода:

* *setSize(int width, int height)* — устанавливает размеры окна. Если не задать размеры, окно будет иметь нулевую высоту независимо от того, что в нем находится и пользователю после запуска придется растягивать окно вручную. Размеры окна включают не только «рабочую» область, но и границы и строку заголовка.
* *setDefaultCloseOperation(int operation)* — позволяет указать действие, которое необходимо выполнить, когда пользователь закрывает окно нажатием на крестик. Обычно в программе есть одно или несколько окон при закрытии которых программа прекращает работу. Для того, чтобы запрограммировать это поведение, следует в качестве параметра operation передать константу EXIT\_ON\_CLOSE, описанную в классе JFrame.
* *setVisible(boolean visible)* — когда окно создается, оно по умолчанию невидимо. Чтобы отобразить окно на экране, вызывается данный метод с параметром true. Если вызвать его с параметром false, окно снова станет невидимым.

*Пример 5.1*. Программа, создающая окно, выводящая его на экран и завершающая работу, после нажатия на кнопку, закрывающую окно.

import javax.swing.JFrame;

public class Panel{

public static void main(String[] args) {

JFrame testWindow = new JFrame("Пробное окно");

testWindow.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

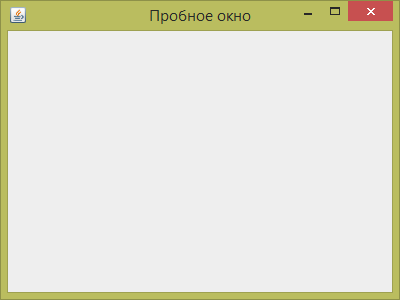
testWindow.setSize(400, 300);

testWindow.setVisible(true);

}

}

В результате мы получим:

  
Рис. 5.1. Пробное окно

Как правило, перед отображением окна, необходимо совершить гораздо больше действий, чем в этой простой программке. Необходимо создать множество элементов управления, настроить их внешний вид, разместить в нужных местах окна. Кроме того, в программе может быть много окон и настраивать их все в методе main() неудобно и неправильно, поскольку нарушает принцип инкапсуляции: держать вместе данные и команды, которые их обрабатывают. Логичнее было бы, чтобы каждое окно занималось своими размерами и содержимым самостоятельно. Поэтому наш пример было бы записать правильнее так:

public class Panel extends JFrame {

public Panel(){

super("Пробное окно");

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

setSize(250, 100);

}

}

public class Main {

public static void main (String [] args) {

JFrame testWindow = new Panel();

testWindow.setVisible(true);

}

}

Из примера видно, что окно описывается в отдельном классе, являющемся наследником JFrame и настраивающее свой внешний вид и поведение в конструкторе (первой командой вызывается конструктор суперкласса). Метод main() содержится в другом классе, ответственном за управление ходом программы. Каждый из этих классов очень прост, каждый занимается своим делом, поэтому в них легко разбираться и легко сопровождать (т.е. совершенствовать при необходимости).

Обратите внимание, что метод setVisible() не вызывается в классе Panel, что вполне логично: за тем, где какая кнопка расположена и какие размеры оно должно иметь, следит само окно, а вот принимать решение о том, какое окно в какой момент выводится на экран — прерогатива управляющего класса программы.

Напрямую в окне элементы управления не размещаются. Для этого служит панель содержимого, занимающая все пространство окна. Обратиться к этой панели можно методом getContentPane() класса JFrame. С помощью метода add(Component component) можно добавить на нее любой элемент управления.

*Пример 5.2*. Добавление кнопки на панель.

JButton myButton = new JButton("Кнопка");

getContentPane().add(myButton);

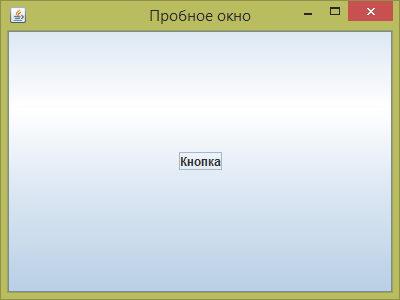
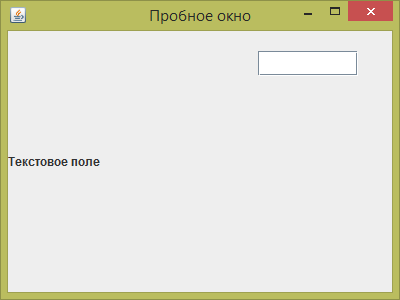
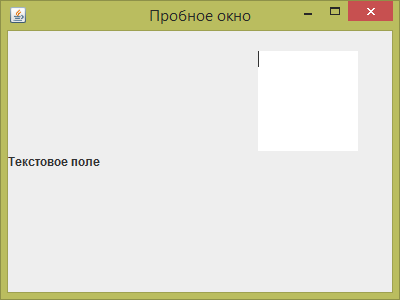
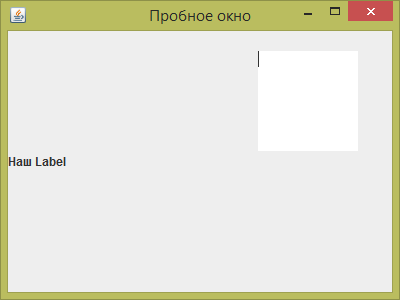
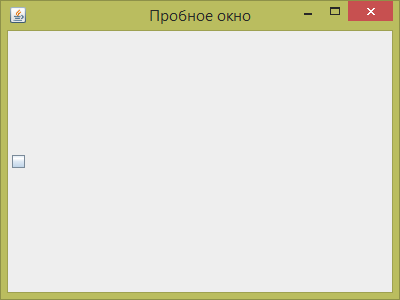
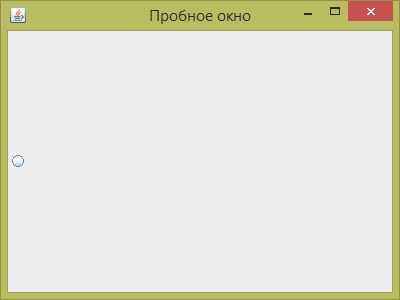
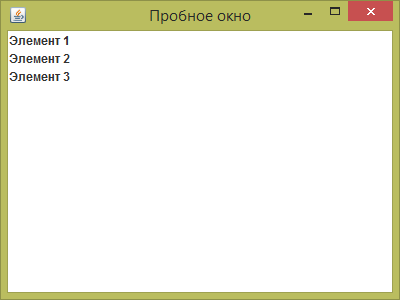
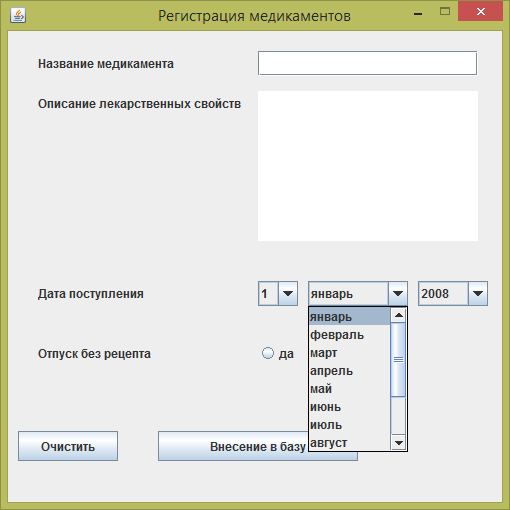


Рис. 5.2. Кнопка

Существует много различных элементов управления библиотеки Swing. Наиболее распространены из них такие, как:

* Текстовое поле(*JTextField*) 
* Текстовая область(*JTextArea*) 
* Лэйбл(*JLabel*) 
* Флажок(*JCheckBox*) 
* Радиокнопка(*JRadioButton*) 
* Список(*JList*) 
* Раскрывающиеся списки(*JComboBox*) 
* и так далее.

Панель JPanel — это элемент управления, представляющий собой прямоугольное пространство, на котором можно размещать другие элементы. JPanel является потомком класса Container. Элементы добавляются и удаляются методами, унаследованными от класса Container.

В примере с кнопкой мы наблюдали, как добавленная на панель содержимого кнопка заняла все ее пространство. Это происходит не всегда. На самом деле у каждой панели есть так называемый менеджер размещения, который определяет стратегию взаимного расположения элементов, добавляемых на панель. Его можно изменить методом setLayout(LayoutManager manager). Но чтобы передать в этот метод нужный параметр, необходимо знать, какими бывают менеджеры.

**Ручное размещение элементов**

Если в качестве менеджера размещения панели установить null, элементы не будут расставляться автоматически. Координаты каждого элемента необходимо в этом случае указать явно, при этом они никак не зависят от размеров панели и от координат других элементов. По умолчанию координаты равны нулю (т.е. элемент расположен в левом верхнем углу панели). Размер элемента также необходимо задавать явно (в противном случае его ширина и высота будут равны нулю и элемент отображаться не будет).

Координаты элемента можно задать следующим методом:

*setLocation(int x, int y)*

Размер элемента задается следующим методом:

*setSize(int width, int height)*

Создадим панель, с которой не будет связано никакого менеджера размещения и вручную разместим на ней две кнопки.

*Пример 5.3.*

public class Panel extends JFrame{

Panel(){

super("Пробное окно");

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

JPanel panel = new JPanel();

panel.setLayout(null);

JButton button = new JButton("Кнопка1");

button.setSize(100, 50);

button.setLocation(40,20);

panel.add(button);

button = new JButton("Кнопка2");

button.setSize(120, 40);

button.setLocation(150,50);

panel.add(button);

setContentPane(panel);

setSize(250, 150);

}

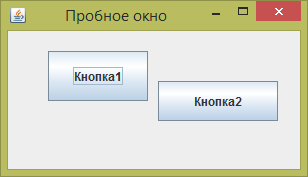


Рис. 5.3. Произвольное размещение

Таблица 5.1. Основные методы классов библиотеки Swing

|  |  |
| --- | --- |
| Метод | Что делает |
| ***Component*** | |
| От своего предка Component все визуальные элементы перенимают общее для них всех поведение, связанное с их визуальной и функциональной сторонами. Вот список основных, выполняемых компонентами, функций и методов для их реализации: | |
| [Font](http://spec-zone.ru/RU/Java/Docs/7/api/java/awt/Font.html) getFont()  void setFont([Font](http://spec-zone.ru/RU/Java/Docs/7/api/java/awt/Font.html) f) | Получение и установка шрифта надписей на компоненте |
| void  [setForeground](http://spec-zone.ru/RU/Java/Docs/7/api/javax/accessibility/AccessibleComponent.html#setForeground(java.awt.Color))([Color](http://spec-zone.ru/RU/Java/Docs/7/api/java/awt/Color.html) c)  [Color](http://spec-zone.ru/RU/Java/Docs/7/api/java/awt/Color.html) getForeground() | Получение и установка цвета самого шрифта |
| Color getBackground()  void  setBackground(Color с) | Получение и установка цвета фона, на котором отображается текст; |
| getMaximumSize()  void  setMaximumSize(Dimension d) | Получение и установка максимального размера; |
| getMinimumSize()  void  setMinimumSize(Dimension d) | Получение и установка минимального размера; |
| getPreferredSize()  void  setPreferredSize(Dimension d) | Получение и установка желательного размера; |
| void  setLocation(int x, int y)  void  setLocation(Point p) | Задание координат и размера компонента (x, y – координаты левой верхней точки; р – объект класса Point) |
| void  setSize(int w, int h)  void  setSize(Dimension d) | Задание размера компонента  (w – ширина, h – высота; d – объект класса Dimension) |
| void  setBounds(int x,int y, int w, int h)  void  setBounds(Rectangle r) | Задание координат и размера компонента  (x, y – координаты левой верхней точки; w – ширина, h – высота; r – объект класса Rectangle) |
| paint()  update()  repaint() | Отображение компонента |
| ***JButton*** | |
| void setText(String) | Установить надпись на компоненте |
| String getText() | Считать надпись на компоненте |
| Void setActionCommand(String) | Установить имя команды, выполняемой при нажатии кнопки |
| String getActionCommand(void) | Считать имя команды, выполняемой при нажатии кнопки |
| ***JLabel*** | |
| void setText(String) | Установить текст в *лэйбл* |
| String getText() | Получить *лэйбл* |
| Vjid setAlignment(int align)  Int getAlignment() | Выравнивание текста |
| void setIcon(Icon) | Графическое изображение в *лэйбле* |
| Icon getIcon() | Получение графического изображения с *лэйбла* |
| ***JCheckBox*** | |
| boolean getSelected()  void setSelected(boolean ) | Управление состоянием флажков (то есть стоит там галочка или нет) |
| ***JComboBox*** | |
| addItem(Object) | Добавление новый пунктов в список |
| void removeItem(String)  void remove ItemAt(int)  void RemoveAllItems() | Удаление строк из списка |
| String getSelectedItem()  int getSelectedIndex() | Определение текущего выбранного элемента в списке |
| int getItemCount() | Получение количества элементов в списке |
| void setSelectedIndex (int index)  void setSelectedItem (Object obj) | Выбор пункта из списка в ходе программы |
| void setEditable(boolean) | Сделать список редактируемым |
| ***JList*** | |
| Object getSelectedValue()  int getSelectedIndex() | Определение выбранного элемента списка |
| ***JTextField*** | |
| String getText() | Получение строки из поля |
| void setText(String) | Установление в поле текста |
| boolean isEditable()  void setEditable(boolean) | Сделать поле редактируемым |
| ***JTextArea*** | |
| void append(String) | Добавление строки в конец текста |
| void insert(String. int) | Вставки строки в область |
| Void replaceRange(String str, int, int) | Замена символов от s1 до s2 -1 строкой str |

**События**

В основу *Java*-программирования наряду с другими положен механизм обработки событий.

Событие – объект, который описывает изменение состояния источника (нажатие кнопки, выбор пункта меню, разворачивание, сворачивание окошка, нажатие клавиши и т.д.). Источник – это объект, генерирующий события. Одно и то же событие может быть значимым для одних объектов и не существенным для других.

В *Java* различают два механизма обработки событий:

1) с помощью метода *handleEvent()* (применялся до версии *jdk 1.1* (*Java 1.0*));

2) с помощью модели делегирования событий.

Далее будем рассматривать особенности обработки событий с применением второго механизма. В соответствии с моделью делегирования событий в обработке событий участвуют 2 объекта: источник (*source*) и блок прослушивания (*listener* − интерфейс для перехода конкретного вида события от конкретного компонента). Источник − объект, генерирующий событие. Блок прослушивания – объект, получающий уведомление о возникновении события, зарегистрированного одним или несколькими источниками, путем вызова одного из его методов (методов блока прослушивания) для приема и обработки этих уведомлений.

Методы обработки событий находятся в пакете *java.awt.event*.

**Классы событий**

В корне иерархии классов событий *Java* находится класс *EventObject*, находящийся в пакете *java.util.* Класс *EventObject* содержит 2 метода:

* *Object getSource( )* – возвращает источник события;
* *toString( )* – возвращает название этого события в виде строки.

В *Java* определены несколько типов событий (табл. 5.2).

Таблица 5.2 – Классы событий

|  |  |
| --- | --- |
| Класс событий | Описание |
| ActionEvent | Генерируется, когда нажата кнопка, дважды щелкнут элемент списка или выбран пункт меню |
| AdjustmentEvent | Генерируется при манипуляциях с полосой прокрутки |
| ComponentEvent | Генерируется, когда компонент скрыт, перемещен, изменен в размере или становится видимым |
| ContainerEvent | Генерируется, когда компонент добавлен или удален из контейнера |
| FocusEvent | Генерируется, когда компонент получает или теряет фокус |
| ItemEvent | Генерируется, когда помечен флажок или элемент списка, сделан выбор элемента в списке, выбран или отменен элемент меню с меткой |
| KeyEvent | Генерируется, когда получен ввод с клавиатуры |
| MouseEvent | Генерируется, когда объект перетащен мышью (*dragged*), перемещен (*moved*), произошел щелчок (*clicked*), нажата (*pressed*) или отпущена (*released*) кнопка мыши, указатель мыши входит или выходит в/за границы компонента |
| TextEvent | Генерируется, когда изменено значение текстового поля |
| WindowEvent | Генерируется, когда окно активизировано, закрыто, развернуто, организован выход из него |

**Класс ActionEvent**

Определяет четыре целочисленные константы, которые можно использовать для идентификации любых модификаторов, связанных с событием действия: *ALT\_MASK*, *CTRL\_MASK*, *META\_MASK* и *SHIFT\_MASK*. Кроме того, существует целочисленная константа *ACTION\_PERFORMED*, которую можно применять для идентификации action-события.

Имеет два конструктора:

*ActionEvent* ( *Object src, int type, String cmd* );

*ActionEvent* ( *Object src, int type, String cmd, int modifiers* );

*src* – ссылка на объект, который генерирует события (для следующих описываемых классов значение аналогичное);

*type* – тип события (для следующих описываемых классов значение аналогичное);

*cmd* – командная строка события;

*modifiers* – указывает, какие клавиши-модификаторы были нажаты при генерации события (*Alt, Ctrl, Shift* ).

Например, когда кнопка нажата, генерируется *action*-событие, которое имеет имя команды, равное метке или надписи на этой кнопке.

**FocusEvent**

Событие этого класса идентифицируется константой *FOCUS\_GAINED* и *FOCUS\_LOST*. Конструкторы класса:

*FocusEvent* ( *Component src, int type* );

*FocusEvent* ( *Component src, int type, boolean temporaryFlag* );

*temporaryFlag* – устанавливается как *true*, если cобытие фокуса временное, иначе – *false*.

**ItemEvent**

Существует два типа *Item*-событий, которые определяются константами:

*DESELECTED* – пользователь отменил выбор элемента;

*SELECTED* – выбрал элемент списка.

Конструктор класса:

*ItemEvent* ( *ItemSelectable src, int type, Object entry, int state* );

*entry* – передает конструкторы элемент, который генерировал *Item*-событие ;

*state* – состояние этого элемента.

Для того чтобы получить ссылку на объект *ItemSelectable*, используется метод *getItemSelectable*( ).

**KeyEvent**

Имеется три типа *Key*-событий, которые идентифицируются тремя

константами:

*KEY\_PRESSED* – клавиша нажата;

*KEY\_RELEASED* – клавиша отпущена;

*KEY\_TYPED* – генерируется только при нажатии символьной клавиши.

Конструктор класса:

*KeyEvent* ( *Component src, int type, long when, int modifiers, int code*);

*when* – параметр, передающий конструктору системное время, когда была нажата клавиша;

*modifiers* – параметр, указывающий, какие модификаторы были нажаты вместе с клавишей;

*code* – параметр, передающий конструктору код клавиши.

**MouseEvent**

Существует семь типов *Mouse*-событий, которые идентифицируются семью константами:

*MOUSE\_CLICKED* – пользователь щелкнул кнопкой мыши;

*MOUSE\_DRAGGED* – пользователь перетащил мышь;

*MOUSE\_ENTERED* – указатель мыши введен в компонент;

*MOUSE\_EXITED* – указатель мыши выведен из компонента;

*MOUSE\_MOVED* – мышь передвинута;

*MOUSE\_PRESSED* – кнопка мыши нажата;

*MOUSE\_RELEASED* – кнопка мыши освобождена.

Конструктор класса:

*MouseEvent* (*Component src, int type, long when, int modifiers, int x, int y, int clicks, boolean triggersPopup* );

*x, y* – координаты мыши;

*clicks* – подсчитывается количество щелчков;

*triggersPopup* – показывает, приводит ли это событие к появлению раскрывающегося меню; если да, то значение параметра соответствует *true*.

*int getX*( ); *int getY*( ); − методы для получения координат мышки.

**TextEvent**

*TEXT\_VALUE\_CHANGED* – событие, определяющее ввод текста пользователем в текстовое поле.

Конструктор класса:

*TextEvent* (*Object src, int type*);

**WindowEvent**

Существует семь типов событий *WindowEvent*:

WINDOW\_ACTIVATED – окно активизировано;

WINDOW\_CLOSED – окно закрыто;

WINDOW\_DEACTIVATED – окно деактивизировано;

WINDOW\_DEICONIFIED – окно развернуто из пиктограммы;

WINDOW\_ICONIFIED – окно свернуто в пиктограмму;

WINDOW\_OPENED – окно открыто;

WINDOW\_CLOSING – пользователь потребовал закрытия окна.

Конструктор класса:

*WindowEvent* (*Window src, int type*);

Метод *Window getWindow*(); − возвращает *Window*-объект, который сгенерировал это событие.

**Элементы-источники событий**

В таблице 5.3 приведены некоторые элементы-источники событий, применяемые в Java, и их описание.

Таблица 5.3 – Элементы-источники событий

|  |  |  |
| --- | --- | --- |
| Источник событий  1 | Описание  2 | |
| Button (кнопка) | Генерирует *action*-события, в тот момент когда нажимается кнопка | |
| Checkbox (флажок) | Генерирует *item*-события, когда флажок устанавливается/сбрасывается | |
| Choice (список с  выбором) | Генерирует *item*-события, когда изменяется выбор элемента в списке с выбором | |
| List (список) | Генерирует *action*-события, когда на элементе списка выполнен двойной щелчок (мышью). Генерирует item-события, когда элемент выделяется или снимается выделение | |
| MenuItem (пункт мею) | Генерирует *action*-события, когда пункт меню выделен. Генерирует события элемента, когда пункт меню с меткой выделен или выделение отменяется. | |
| Scrollbar (полоса прокрутки) | Генерирует *adjustment*-события при манипуляциях с полосой прокрутки | |
| TextField и TextArea  (текстовое поле и текстовая область) | | Генерирует *text*-события, когда пользователь вводит символ |
| Window (окно) | | Генерирует *window*-события, когда окно активизируется, закрывается, деактивизируется, сворачивается в пиктограмму, разворачивается из пиктограммы, открывается или выполняется выход из него (*quit*). |

**Интерфейсы прослушивания событий**

Модель делегирования событий содержит две части: источник событий и блоки прослушивания событий. Блоки прослушивания событий создаются путем реализации одного или нескольких интерфейсов прослушивания событий. Эти интерфейсы определены в пакете *java.awt.event*. Когда событие происходит, источник события вызывает соответствующий метод, определенный блоком прослушивания, и передает ему объект события в качестве параметра.

В табл. 5.4 приведены интерфейсы прослушивания событий и их методы. Когда класс реализует какой-нибудь из этих интерфейсов, то все методы интерфейса должны быть реализованы в этом классе. В случае, если среди методов интерфейса вам необходимы не все, а только некоторые из них, то для остальных методов в качестве реализации можно оставить пустые скобки {}.

Таблица 5.4 – Интерфейсы прослушивания событий

|  |  |  |
| --- | --- | --- |
| Интерфейс | Описание, определяемые методы | |
| 1 | 2 | |
| ActionListener | Определяет один метод для приема action-событий:  *void actionPerformed(ActionEvent ae)* | |
| AdjustmentListener | Определяет один метод для приема adjustment-событий:  *void adjustmentValueChanged(AdjustmentEvent ae)* | |
| FocusListener | Определяет два метода для приема focus-события  *void focusGained(FocusEvent fe)*  *void focusLost(FocusEvent fe)* | |
| ItemListener | Определяет один метод, распознающий события изменения состояние элемента  *void itemStateChanged(ItemEvent ie)* | |
| KeyListener | | Определяет три метода, распознающих события клавиатуры  *void keyPressed(KeyEvent ke)*  *void keyReleased(KeyEvent ke)*  *void keyTyped(KeyEvent ke)* |
| MouseListener | | Определяет пять методов, распознающих события щелчка, входа в границы компонента, выхода из границ, нажатия/ отпускания клавиши мыши  *void mouseClicked(MouseEvent me)*  *void mouseEntered(MouseEvent me)*  *void mouseExited(MouseEvent me)*  *void mousePressed(MouseEvent me)*  *void mouseReleased(MouseEvent me)* |
| MouseMotionListener | | Определяет два метода, распознающих события перетаскивания/ перемещения мыши  *void mouseDragged(MouseEvent me)*  *void mouseMoved(MouseEvent me)* |
| TextListener | | Определяет один метод, связанный с событием изменения текстового значения  *void textChanged(TextEvent te)* |
| WindowListener | | Определяет семь методов, связанных с окошком – событиями активации и т.д.  *void windowActivated(WindowEvent we)*  *void windowClosed(WindowEvent we)*  *void windowClosing(WindowEvent we)*  *void windowDeactivated(WindowEvent we)*  *void windowDeiconified(WindowEvent we)*  *void windowIconified(WindowEvent we)*  *void windowOpened(WindowEvent we)* |

**Классы пакета AWT: Component, Window, Frame**

**Класс Component**

Абстрактный класс, инкапсулирующий все элементы визуального интерфейса пользователя. Все управляющие компоненты окна пользователя являются подклассами класса Component. В данном классе определено более 100 методов, которые отвечают за управление событиями, позиционирование, управление размерами, управление цветами, перерисовку.

**Класс Window**

Создает окно верхнего уровня на рабочем столе. Он расширяется классом Frame, который и представляет интерфейсное окно, окно с меню, обрамлением, необходимое для создания графического приложения с его компонентами.

**Класс Frame**

Инкапсулирует полноценное окно, имеющее строку заголовка, строку меню, обрамление и углы, изменяющие размеры окна.

Для создания окна *Frame* существуют два конструктора:

*Frame ();*

*Frame ( String Zagolovok);*

Для установления размера фрейма существуют следующие методы:

*void setSize ( int Width, int Height );*

*void setSize ( Dimension size );*

*Dimension* – класс, содержащий поля *width* и *height*.

Метод, позволяющий сделать окно видимым:

*void setVisible ( boolean visibleFlag );*

Пример кода для создания фреймового окна показан ниже.

Пример

import java.awt.\*;

public class NewFrame extends Frame

{

TextArea ta;

public NewFrame ( String title )

{

super ( title );

setSize(300,200);

//…

}

public static void main ( String args [ ] )

{

NewFrame nf = new NewFrame (“Мой фрейм”);

nf.show ( );

}

}

Некоторые методы класса *Frame*:

*String getTitle( );* – получить заголовок окна;

*void setTitle (String);* – установить заголовок окна;

*void setResizable (boolean);* - разрешить изменение размеров окна;

*boolean isResizable();* - возвратить true, если размер окна можно изменять, иначе false.

**Элементы управления Label, Button, Checkbox, Choice, List, Scrollbar**

Элемент управления – это компоненты, которые предоставляют пользователю различные способы взаимодействия с приложением (например кнопки, флажки, полосы прокрутки и т.п.)

Элементы управления представлены следующими классами:

*Label* – с помощью класса *Label* можно создавать текстовые строки в окне *Java*-программ. По умолчанию текст будет выровнен влево, но используя методы

setAligment (Label.CENTER );

setAligment (Label.RIGHT );

строку можно выровнять по центру и по правому краю. Можно создавать выводимый текст либо при создании объекта класса *Label*, либо создать пустой объект и уже затем определить его текст вызовом метода *setText( )*.

Для этого класса существуют три конструктора:

Label first = new Label ( );

Label second = new Label (“some text”);

Label third = new Label (“some text”, Label.CENTER);

*Button* – представляет на экране кнопку. Имеет два конструктора:

Button first = new Button ( );

Button second = new Button (“some text”);

Сделать кнопку неактивной можно методом *void disable( )*.

Следующий пример демонстрирует обработку кнопки.

Пример 5.4

Листинг ButtonDemo.java

import java.awt.\*;

import java.awt.event.\*;

public class ButtonDemo extends Frame

implements ActionListener,WindowListener{

Button btn;

Label lb;

int count;

public ButtonDemo(){

super("Фреймовое окно с кнопкой");

setLayout(new FlowLayout(FlowLayout.LEFT));

btn=new Button("Нажмите кнопку");

setSize(300,200);

btn.addActionListener(this);

lb=new Label("Здесь текстовое поле");

count=0;

add(btn);

add(lb);

setVisible(true);

addWindowListener(this);

}

public void actionPerformed(ActionEvent ae) {

count++;

lb.setText("Кнопка нажата "+count+" раз");

}

public void windowClosing(WindowEvent we){

this.dispose();

}

public void windowActivated(WindowEvent we){};

public void windowClosed(WindowEvent we){};

public void windowDeactivated(WindowEvent we){};

public void windowDeiconified(WindowEvent we){};

public void windowIconified(WindowEvent we){};

public void windowOpened(WindowEvent we){};

public static void main(String args[])

{ButtonDemo bd=new ButtonDemo();

}

}

*Checkbox* – отвечает за создание и отображение кнопок с независимой фиксацией. Эти кнопки имеют два состояния: включено и выключено. Щелчок по такой кнопке приводит к тому, что ее состояние меняется на противоположное. Если разместить несколько кнопок с независимой фиксаций внутри элемента класса *CheckboxGroup*, то вместо них мы получаем кнопки с зависимой фиксацией. Для такой группы кнопок характерно то, что в один и тот же момент может быть включена только одна кнопка. Если нажать какую-либо кнопку из группы, то ранее нажатая кнопка будет отпущена.

*Choice* – создает раскрывающийся список.

Пример реализации списка из трех пунктов.

Пример 5.5

Choice choice = new Choice ( );

choice.addItem (“First”);

choice.addItem (“Second”);

choice.addItem (“Third”);

Методы класса *Choice*:

*int countItems( )* – считать количество пунктов в списке;

*String getItem(int)* – возвратить строку с определенным номером в списке;

*void select(int)* – выбрать строку с определенным номером.

*List* – по назначению похож на *Choice*, но предоставляет пользователю не раскрывающийся список, а окно с полосами прокрутки. Такое окно содержит пункты выбора.

Создать объект класса *List* можно двумя способами:

1. Создать пустой список и добавить в него пункты методом *addItem()*. При этом размер списка будет расти при добавлении пунктов.

Пример 5.6

List list1 = new List ( );

list1.addItem (“1”);

list1.addItem (“2”);

list1.addItem (“3”);

2. Создать пустой список, добавить пункты при помощи *addItem ( )*, при этом можно ограничить количество видимых в окне списка пунктов. Ниже показан пример, демонстрирующий список, в котором видно 2 элемента.

Пример 5.7

List list2 = new List (2, true );

list2.addItem (“1”);

list2.addItem (“2”);

list2.addItem (“3”);

Некоторые полезные методы класса *List*:

*String getItem(int)* – получить текст пункта с номером *int*;

*int countItems( )* – посчитать количество пунктов в списке;

*void clear( )* – очистить список;

*void delItem(int)* – удалить из списка пункт с номером *int*;

*void delItems(int, int)* – удалить из списка элементы с int по *int*;

*int getSelectedIndex( )* – получить порядковый номер выделенного элемента списка;

*void select(int)* – выделить элемент списка с определенным номером.

Следующий пример демонстрирует обработку списка, раскрывающегося списка и кнопки с независимой фиксацией.

Пример 5.8

Листинг ListDemo.java

import java.awt.\*;

import java.awt.event.\*;

public class ListDemo extends Frame implements ItemListener{

List lst;Checkbox chb;Choice ch;

public ListDemo(){

super("Фреймовое окно");

setLayout(new FlowLayout(FlowLayout.CENTER));

setSize(300,200);

lst = new List (2, false );

lst.addItem ("1 BSUIR");

lst.addItem ("2 BSEU");

lst.addItem ("3 BSU");

chb=new Checkbox("Кнопка с независимой фиксацией");

ch=new Choice();

ch.add("Сюда переносятся строки со списка");

add(lst);

add(ch);

add(chb);

setVisible(true);

lst.addItemListener(this);

}

public void itemStateChanged(ItemEvent ie){

ch.addItem(lst.getSelectedItem());

}

public static void main(String args[]){

ListDemo list=new ListDemo();

}

}

*Scrollbar* – определяет полосу прокрутки. Создать полосу прокрутки можно следующим образом:

*new Scrollbar ( );*

*new Scrollbar ( Scrollbar.VERTICAL );*

*new Scrollbar* ( <ориентация>, <текущее значение>, <видно>, <минимальное значение>, <максимальное значение>);

<ориентация> – ориентация полосы, которая задается константами *Scrollbar.HORIZONTAL*, *Scrollbar.VERTICAL*.

<текущее значение> – начальное значение, в которое помещается бегунок полосы прокрутки.

<видно> – сколько пикселов прокручиваемой области видно и на сколько пикселов эта область будет прокручена при щелчке мышью на полосе прокрутки.

<минимальное значение> – минимальное значение полосы прокрутки.

<максимальное значение> – максимальное значение полосы прокрутки.

**Элементы управления TextField и TextArea**

Эти два класса позволяют отображать текст с возможностью его выделения и редактирования. Это по сути маленькие текстовые редакторы – однострочный (*TextField*) и многострочный (*TextArea*).

Создать текстовое поле и текстовую область можно следующими способами:

*TextField tf = new TextField (50);*

*TextArea ta = new TextArea (5, 30);*

Чтобы запретить или разрешить редактирование текста в окне, можно возпользоваться методом *void setEditable(boolean).*

*tf.setEditable (false);*

*ta.setEditable (false);*

Некоторые методы классов *TextField* и *TextArea*:

*String getText( )* – читать текст;

*void setText(String)* – отобразить текст;

*void selectAll( )* – выделить весь текст;

*int getColumns( )* – возвратить количество символов строки.

Специфические методы *TextField*:

*void setEchoChar(char)* – установить символ маски (при вводе паролей);

*char getEchoChar( )* − узнать символ маски.

Специфические методы для *TextArea*:

*int getRows( )* – считать количество строк в окне;

*void insertText(String, int)* – вставить текст в определенной позиции int;

*void replaceText(String, int, int)* – заменить текст между заданными начальной и конечной позицией.

Следующий пример демонстрирует приложение с элементами управления: кнопкой (*Button*), списком (*List*), раскрывающимся списком(*Choice*), текстовой строкой (*Label*), текстовым полем (*TextField*). Введенное в текстовом поле слово при нажатии кнопки добавляется как в список, так и в раскрывающийся список. Также реализован механизм закрытия фрейма.

Пример 5.9

Листинг GUISample.java

import java.io.\*; //импортирование пакета ввода-вывода

import java.awt.\*; //импортирование пакета awt

import java.awt.event.\*; //импортирование пакета поддержки событий

public class GUISample extends Frame{ //объявление класса GUISample

Button b1 = new Button("Add"); //создание кнопки с надписью "Add"

Choice ch1=new Choice(); //создание раскрывающегося списка

TextField tf1 = new TextField(); //создание текстового поля (строки

//ввода)

Label label1 = new Label("Enter your text here:"); //создание текстовой

//строки

List l1 = new List(); //создание списка

public GUISample(){ //объявление конструктора класса

setLayout(null); //отключение менеджера компоновки

setSize(600,400); //установка размеров фрейма

setTitle("This is my Frame"); //установка заголовка фрейма

setBackground(Color.cyan); //установка цвета заднего фона фрейма

add(b1); //добавление кнопки к окну

b1.setBounds(220,200,84,24); //установка размеров кнопки

b1.setForeground(Color.black); //установка цвета переднего фона кнопки

b1.setBackground(Color.magenta); //установка цвета заднего фона кнопки

add(ch1); //добавление раскрывающегося списка к окну

ch1.setBounds(50,120,120,20); //установка размеров раскрывающегося

//списка

add(tf1); //добавление текстового поля к окну

tf1.setBounds(200,80,120,20); //установка размеров текстового поля

add(label1); //добавление текстовой строки к окну

label1.setBounds(200,55,120,20); //установка размеров текстовой строки

add(l1); //добавление списка к окну

l1.setBackground(Color.white); //установка цвета заднего фона списка

l1.setBounds(350,120,200,216); //установка размеров списка

/\*регистрация блока прослушивания событий типа WindowEvent\*/

addWindowListener(new WindowClose());

/\*регистрация блока прослушивания событий типа ActionEvent\*/

b1.addActionListener(new ButtonAdd());

}

/\*объявление класса-адаптера для обработки Window-событий\*/

class WindowClose extends WindowAdapter {

/\*метод, который вызывается при закрытии окна\*/

public void windowClosing(WindowEvent we) {

setVisible(false); //фрейм-окно становится невидимым

}

}

/\*объявление класса для обработки Action-событий (класс ButtonAdd реализует интерфейс ActionListener)\*/

class ButtonAdd implements ActionListener {

/\*реализация метода, который вызывается при наступлении action-события\*/

public void actionPerformed(ActionEvent event) {

/\*добавление текста из текстового поля в раскрывающийся список\*/

ch1.add(tf1.getText());

/\*добавление текста из текстового поля в список\*/

l1.add(tf1.getText(),2);

}

}

static public void main(String args[]){ //объявление метода main()

GUISample MyFrame=new GUISample(); //создание экземпляра класса GUISample

MyFrame.setVisible(true); //выведение окна на экран дисплея

} }

Результаты работы программы представлены на рис. 5.4.



Рис. 5.4. Результат работы программы

**Диалоговые окна**

Диалоговые окна подобны фрейм-окнам за исключением того, что они – всегда дочерние окна для окна верхнего уровня. Кроме того, диалоговые окна не имеют строки меню. В других отношениях они функционируют подобно фреймовым окнам. Можно, например, добавить к ним элементы управления тем же способом, каким добавляются элементы управления к фреймовому окну. Диалоговые окна могут быть модальными или немодальными. Когда модальное диалоговое окно активно, весь ввод направляется к нему, пока оно не будет закрыто. Это означает, что невозможно обратиться к другим частям программы до тех пор, пока не закрыто диалоговое окно. Когда немодальное диалоговое окно активно, фокус ввода может быть направлен другому окну программы. Таким образом, другие части программы остаются активными и доступными. Диалоговые окна обслуживает класс *Dialog*. Обычно используются следующие конструкторы класса:

*Dialog(Frame parentWindow, boolean mode);*

*Dialog(Frame parentWindow, String title, boolean mode);*

*parentWindow* – владелец диалогового окна. Если *mode* имеет значение *true*, диалоговое окно является модальным. Иначе оно немодальное. Заголовок диалогового окна можно передать через параметр *title*.

Следующий пример демонстрирует фреймовое окно с меню, из которого (выбором пунтов меню *File→DemoDialog*) вызывается модальное диалоговое окно.

Пример 5.10

Листинг Frame1.java

import java.awt.\*;

import java.awt.event.\*;

public class Frame1 extends Frame

implements ActionListener,WindowListener{

Menu file;MenuItem item1;

public Frame1(){

super("Фреймовое окно с меню");

setSize(500,300);

//создать строку главного меню и добавить его во фрейм

MenuBar mbar=new MenuBar();

setMenuBar(mbar);

//создать элемент меню

file=new Menu("File");

mbar.add(file);

file.add(item1=new MenuItem("DemoDialog"));

item1.addActionListener(this);

setVisible(true);

addWindowListener(this);

}

public void actionPerformed(ActionEvent ae) {

DemoDialog d=new DemoDialog(this,"Диалоговое окно",true);

}

public void windowClosing(WindowEvent we){

this.dispose();

}

public void windowActivated(WindowEvent we){};

public void windowClosed(WindowEvent we){};

public void windowDeactivated(WindowEvent we){};

public void windowDeiconified(WindowEvent we){};

public void windowIconified(WindowEvent we){};

public void windowOpened(WindowEvent we){};

public static void main(String args[])

{Frame1 f=new Frame1();

}

class DemoDialog extends Dialog implements ActionListener{

Button btn;

public DemoDialog(Frame1 ff, String title,boolean b){

super(ff,title,b);

setLayout(new FlowLayout(FlowLayout.LEFT));

btn=new Button("Закрыть");

setSize(300,200);

add(btn);

btn.addActionListener(this);

setVisible(true);

}

public void actionPerformed(ActionEvent ae){

this.dispose();

}

}

}

На рисунке 5.5 показан результат работы программы:



Рис. 5.5. Результат работы программы, приведенной в листинге Frame1.java

**Подведем итоги**

Ну вот мы установили на панели все нам необходимые элементы, разместили их так, как нам необходимо, и следующим шагом будет прикрепление к этим элементам соответствующего действия.

Обработка любого события (нажатие кнопки, щелчок мышью и др.) состоит в связывании события с методом, его обрабатывающим. Принцип обработки событий базируется на модели делегирования событий. В этой модели имеется блок прослушивания события *(EventListener)*, который ждет поступления события определенного типа от источника, после чего обрабатывает его и возвращает управление. Источник – это объект, который генерирует событие. Источником событий могут являться элементы управления: кнопки *(JButton,JCheckbox, JRadioButton),* списки, кнопки-меню. События могут генерироваться фреймами и апплетами, как mouse- и key-события. События генерируются окнами при развертке, сворачивании, выходе из окна.

Если изменяется его внутреннее состояние, например, изменился размер, изменилось значение поля, произведен щелчок мыши по форме или выбор значения из списка, после генерации объекта-события пересылается для обработки зарегистрированному в источнике блоку прослушивания как параметр его методов – обработчиков событий.

Блоки прослушивания *Listener* представляют собой объекты классов, реализующих интерфейсы прослушивания событий, определенных в пакетах *java.awt.event* и *javax.swing.event*. Соответствующие методы, объявленные в используемых интерфейсах, необходимо явно реализовать при создании собственных классов прослушивания. Эти методы и являются обработчиками события. Передаваемый источником блоку прослушивания объект-событие является аргументом обработчика события. Объект класса – блока прослушивания события необходимо зарегистрировать в источнике методом

источник.*add*Событие*Listener***(**объект\_прослушиватель**);**

После этого объект-слушатель (*Listener*)будет реагировать именно на данное событие и вызывать метод «обработчик события». Такая логика обработки событий позволяет легко отделить интерфейсную часть приложения от функциональной, что считается необходимым при проектировании современных приложений. Удалить слушателя определенного события можно с помощью метода *remove*Событие*Listener()*.

Когда событие происходит, все зарегистрированные блоки прослушивания уведомляются и принимают копию объекта события. Таким образом источник вызывает метод-обработчик события, определенный в классе, являющемся блоком прослушивания, и передает методу объект события в качестве параметра. В качестве блоков прослушивания на практике используются внутренние классы. В этом случае в методе, регистрирующем блок прослушивания в качестве параметра, используется объект этого внутреннего класса.

Каждый интерфейс, включаемый в блок прослушивания, наследуется от интерфейса *EventListener* и предназначен для обработки определенного типа событий. При этом он содержит один или несколько методов, которые всегда принимают объект события в качестве единственного параметра и вызываются в определенных ситуациях.

Таблица 5.5. События и слушатели, которые с ними работают

|  |  |
| --- | --- |
| Событие | Слушатель |
| **Библиотека java.awt.event** | |
| **ActionEvent**– генерируется: при нажатии кнопки; двойном щелчке клавишей мыши по элементам списка; при выборе пункта меню; | ActionListener |
| **AdjustmentEvent** – генерируется при изменении полосы прокрутки; | AdjustmentListener |
| **ComponentEvent** – генерируется, если компонент скрыт, перемещен, изменен в размере или становится видимым; | ComponentListener |
| **FocusEvent** – генерируется, если компонент получает или теряет фокус ввода; | FocusListener |
| **TextEvent**– генерируется при изменении текстового поля; | TextListener |
| **ItemEvent**– генерируется при выборе элемента из списка. | ItemListener |
| Класс **InputEvent**является абстрактным суперклассом событий ввода (для клавиатуры или мыши). | interface InputListener |
| **KeyEvent -** генерируется при вводе с клавиатуры | KeyListener |
| **MouseEvent -** генерируется, если произошло событие мыши | MouseListener |
| **Библиотека javax.swing.event** | |
| [**ChangeEvent**](http://docs.oracle.com/javase/7/docs/api/javax/swing/event/ChangeEvent.html) – генерируется при изменении состояния в источники события. | ChangeListener |
| **HyperlinkEvent** – генерируется, когда что-то изменилось относительно гипертекстовой ссылки. | HyperlinkListener |
| [**ListDataEvent**](http://docs.oracle.com/javase/7/docs/api/javax/swing/event/ListDataEvent.html) - определяет событие, которое инкапсулирует изменения в списке. | ListDataListener |
| **MenuEvent** – генерируется, когда меню, которое является источником события, было размещено, выбрано или отменено. | MenuListener |
| **MenuKeyEvent –** генерируется, когда элемент меню получил KeyEvent. | MenuKeyListener |
| **TableModelEvent –** генерируется при изменении модели таблицы. | TableModelListener |
| [**TreeModelEvent**](http://docs.oracle.com/javase/7/docs/api/javax/swing/event/TreeModelEvent.html) – генерируется при изменений TreeModel, уведомляет слушателей об изменении. | TreeModelListener |
| [**TreeSelectionEvent**](http://docs.oracle.com/javase/7/docs/api/javax/swing/event/TreeSelectionEvent.html) – событие, которое характеризует изменение текущего выбора. | TreeSelectionListener |

Как только пользователь нажимает кнопку, создается ActionEvent событие, которое передается слушателям кнопки. Для того, чтобы организовать слушателя, Swing предоставляет интерфейс ActionListener, который необходимо реализовать. Интерфейс ActionListener требует только реализации одного метода — actionPerformed. Пример класса, реализующего интерфейс ActionListener представлен ниже.

Пример. 5.11.

public class OurActionListener implements ActionListener {

public void actionPerformed(ActionEvent e) {

//Код, который нужно выполнить при нажатии

}

}

После того, как обработчик создан, его необходимо добавить к кнопке. Делается это при помощи метода addActionListener. В качестве параметра методу передается обработчик. Например, это можно сделать вот так:

Пример 5.12.

JButton button = new JButton("Кнопка");

ActionListener actionListener = new OurActionListener();

button.addActionListener(actionListener);

Здесь мы создаем сначала кнопку. Потом создаем экземпляр нашего слушателя OurActionListener, а затем добавляем его в качестве слушателя к кнопке с помощью вызова addActionListener и передаем ему экземпляр обработчика.

Или же таким образом:

JButton button = new JButton("Кнопка");

button.addActionListener(new OurActionListener());

Вообще слушателей может быть неопределенное количество. Если нам будет необходимо два или более слушателей, которые должны будут по-разному реагировать на нажатие кнопки, то для каждого из них вызовем addActionListener.

Пример 5.13. Создать панель, на которой будут находиться 2 текстовых поля и одна кнопка. При нажатии на кнопку текст из первого текстового поля переместится во второе, а второе в первое, то есть поменяются местами.

public class User // класс, который содержит метод main()

{

public static void main(String[] args) {

Panel testPanel = new Panel();

testPanel.setVisible(true);

}

}

public class Panel extends JFrame{

JButton button;

JTextField text1, text2;

Panel(){

super("Пробное окно");

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

setSize(500, 200);

button = new JButton("Поменять местами");

text1 = new JTextField();

text2 = new JTextField();

setLayout(null);

button.setSize(300, 50);

button.setLocation(90,100);

text1.setSize(200, 50);

text1.setLocation(20,20);

text2.setSize(200, 50);

text2.setLocation(250,20);

add(button);

add(text1);

add(text2);

button.addActionListener(new ButtonActionLisener());

}

public class ButtonActionLisener implements ActionListener{

public void actionPerformed(ActionEvent e) {

if(e.getSource() == button ){

String str1 = text1.getText();

String str2 = text2.getText();

text1.setText(str2);

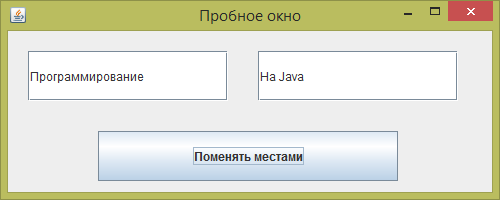
text2.setText(str1);

}

}

}

}



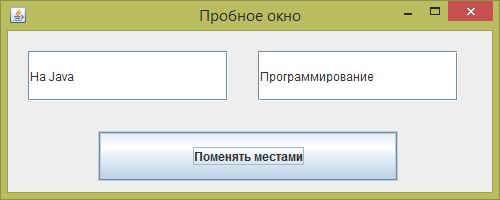


Рис. 5.5.

Таким же образом действие реализуются и для других элементов управления.

Пример 5.14. Создать панель, на которой будут находиться 2 радиокнопки и текстовое поле. При выборе первой из радиокнопок в текстовом поле высвечивается текст “Павел”, при выборе второй “Ольга”.

public class User // класс, который содержит метод main()

{

public static void main(String[] args) {

Panel testPanel = new Panel();

testPanel.setVisible(true);

}

}

public class Panel extends JFrame{

JRadioButton button1, button2;

JTextField text1;

ButtonGroup bg;

Panel(){

super("Пробное окно");

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

setSize(500, 200);

button1 = new JRadioButton("Мужской");

button2 = new JRadioButton("Женский");

text1 = new JTextField();

bg = new ButtonGroup();

bg.add(button1);

bg.add(button2);

setLayout(null);

button1.setSize(100, 50);

button1.setLocation(50,20);

button2.setSize(100, 50);

button2.setLocation(200,20);

text1.setSize(200, 50);

text1.setLocation(50,80);

add(text1);

add(button1);

add(button2);

button1.addActionListener(new ButtonActionLisener());

button2.addActionListener(new ButtonActionLisener());

}

public class ButtonActionLisener implements ActionListener{

public void actionPerformed(ActionEvent e) {

{

if(e.getSource() == button1)

text1.setText("Павел");

if(e.getSource() == button2)

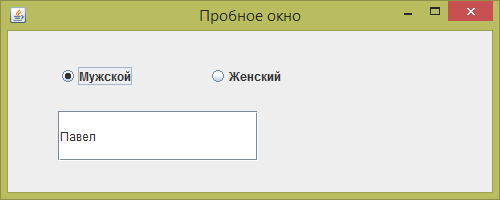
text1.setText("Ольга");

}

}

}

}



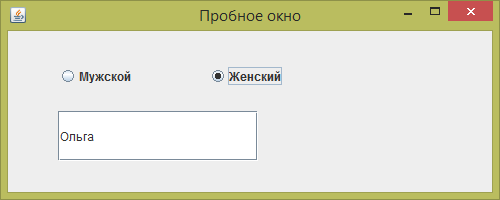


Рис. 5.7

Пример 5.15. Создать панель, на которой будут список и текстовое поле. Пример демонстрирует использование MouseListener. При выборе элемента из списка он появляется в текстовом поле, при вождении мышки по области демонстрируется использование методов MouseEvent.

public class User // класс, который содержит метод main()

{

public static void main(String[] args) {

Panel testPanel = new Panel();

testPanel.setVisible(true);

}

}

public class Panel extends JFrame{

Object box;

JList list;

JTextField text1;

String[] myList = {"Некрасова Ирина", "Лепель Андрей", "Иванов Николай", "Руденко Екатерина", "Некрасов Павел", "Мирницкая Анастасия"};

Panel(){

super("Пробное окно");

setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*);

setSize(500, 200);

list = new JList(myList);

text1 = new JTextField();

setLayout(null);

list.setSize(150, 110);

list.setLocation(10,10);

text1.setSize(200, 50);

text1.setLocation(200,50);

add(text1);

add(list);

list.addMouseListener(new ButtonActionLisener());

}

public class ButtonActionLisener implements MouseListener{

//  *Если пользователь нажал и отпустил одну из кнопок, вызывается метод mouseClicked.*

public void mouseClicked(MouseEvent e) {

list = (JList) e.getSource();

Object text = list.getSelectedValue();

text1.setText((String)text);

}

// *mouseEntered - данный метод будет вызываться системой у слушателя каждый раз, когда курсор мыши будет оказываться над компонентом.*

public void mouseEntered(MouseEvent e) {

list = (JList) e.getSource();

text1.setText("метод mouseEntered()");

}

// *mouseExited – данный метод срабатывает, когда убираем курсор мыши с компонента.*

public void mouseExited(MouseEvent e) {

list = (JList) e.getSource();

text1.setText("метод mouseExited()");

}

// *Навели на компонент, зажали кнопку — система вызвала mousePressed.*

public void mousePressed(MouseEvent e) {

list = (JList) e.getSource();

text1.setText("метод mousePressed()");

}

// *Отпускаем кнопку — система вызвала mouseReleased.*

public void mouseReleased(MouseEvent e) {

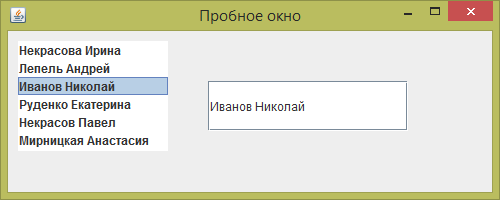
list = (JList) e.getSource();

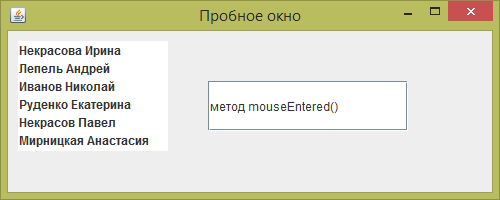
text1.setText("метод mouseReleased()");

}

}

}





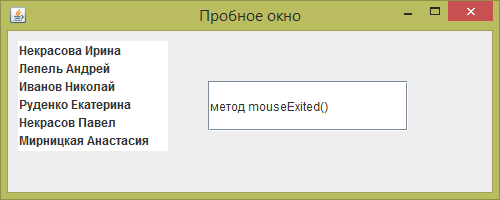


Рис. 5.8.

***Внутренние классы***

В литературе по Java встречаются такие термины, как "внутренние классы" (inner classes) и "вложенные классы" (nested classes). Итак, вложенный класс - это класс, который объявлен внутри объявления другого класса. Например:

class OutClass{

…

class NewClass{

…

}

…

}

**Для чего используются внутренние классы**

* Это хороший способ группировки классов, которые используются только в одном месте: если класс полезен только для одного другого класса, то логично будет держать их вместе. Вложение таких вспомогательных классов делает код более удобным.
* Инкапсуляция: допустим, есть два класса A и B, классу B требуется доступ к свойству класса A, которое может быть приватным. Вложение класса B в класс A решит эту проблему, более того сам класс B можно скрыть от внешнего использования.
* Улучшение читаемости и обслуживаемости кода: вложение малых классов в более высокоуровневые классы позволяет хранить код там, где он используется.

**Типы внутренних классов**

*1.Статические внутренние классы.*

Декларируются внутри основного класса и обозначаются ключевым словом static. Не имеют доступа к членам внешнего класса за исключением статических. Может содержать статические поля, методы и классы, в отличие от других типов внутренних классов.

class OutClass{

…

static class NewClass{

…

}

…

}

*2.Внутренние классы.*

Декларируются внутри основного класса. В отличие от статических внутренних классов, имеют доступ к членам внешнего класса. Не могут содержать (но могут наследовать) определение статических полей, методов и классов (кроме констант).

class OutClass{

…

class NewClass{

…

}

…

}

*3.Локальные классы.*

Декларируются внутри методов основного класса. Могут быть использованы только внутри этих методов. Имеют доступ к членам внешнего класса. Имеют доступ как к локальным переменным, так и к параметрам метода при одном условии - переменные и параметры используемые локальным классом должны быть задекларированы final. Не могут содержать определение (но могут наследовать) статических полей, методов и классов (кроме констант).

class OutClass{

void methodWithLocalClass(){

class NewClass{

…

}

}

…

}

*4. Анонимные классы.*

Декларируются внутри методов основного класса. Могут быть использованы только внутри этих методов. В отличие от локальных классов, анонимные классы не имеют названия. Главное требование к анонимному классу - он должен наследовать существующий класс или реализовывать существующий интерфейс. Не могут содержать определение (но могут наследовать) статических полей, методов и классов (кроме констант).

class OutClass{

void methodWithLocalClass()

{

*/\* При определении анонимного класса применен полиморфизм – переменная listener содержит экземпляр анонимного класса, реализующего существующий интерфейс ActionListener \*/*

ActionListener listener = new ActionListener()

{

public void actionPerformed (ActionEvent event)

{

}

};

}

**Пример выполнения индивидуального задания**

Пример 5.15.

Реализовать приложение, в котором будет осуществляться регистрация, поступающих в аптеку новых медикаментов. Введённые данные сохранять в файл. Реализовать механизм закрытия окна.

import java.awt.event.\*;

import java.io.\*;

import javax.swing.\*;

public class User {

public static void main(String[] args) {

Reader med = new Reader("Регистрация медикаментов");

med.setVisible(true);

med.setResizable(false);

med.setLocationRelativeTo(null);

}

}

public class Reader extends JFrame {

String text1;

int k;

Object boxA, boxB, boxC;

File file = new File("D://GUI.txt");

static JLabel *l1*, *l2*, *l3*, *l4*;

JComboBox box\_1, box\_2, box\_3;

JRadioButton flag1, flag2;

ButtonGroup bg;

static JButton *b*, *del*;

static JTextField *text*;

static JTextArea *area*;

static String[] *box1* = {"1","2","3","4","5","6","7","8","9","10","11","12","13","14","15","16","17","18","19","20","21","22","23","24","25","26","27","28","29","30","31"};

static String[] *box2* = {"январь","февраль","март","апрель","май","июнь","июль","август","сентябрь","октябрь","ноябрь","декабрь"};

static String[] *box3* = {"2008","2009","2010","2011","2012","2013","2014","2015","2016","2017","2018","2019","2020"};

public Reader(String str){

super(str);

setSize(500, 500);

setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*);

*b* = new JButton("Внесение в базу");

*del* = new JButton("Очистить");

*text* = new JTextField(9);

*area* = new JTextArea(9, 9);

*l1* = new JLabel("Название медикамента");

*l2* = new JLabel("Описание лекарственных свойств");

*l3* = new JLabel("Дата поступления");

*l4* = new JLabel("Отпуск без рецепта");

box\_1 = new JComboBox(*box1*);

box\_2 = new JComboBox(*box2*);

box\_3 = new JComboBox<Object>(*box3*);

flag1 = new JRadioButton("да");

flag2 = new JRadioButton("нет");

bg = new ButtonGroup();

bg.add(flag1);

bg.add(flag2);

setLayout(null);

*b*.setSize(200,30);

*b*.setLocation(150, 400);

*del*.setSize(100, 30);

*del*.setLocation(10, 400);

*text*.setSize(220,25);

*text*.setLocation(250, 20);

*area*.setSize(220,150);

*area*.setLocation(250,60);

*l1*.setSize(200,25);

*l1*.setLocation(30, 20);

*l2*.setSize(220, 25);

*l2*.setLocation(30, 60);

*l3*.setSize(200, 25);

*l3*.setLocation(30, 250);

*l4*.setSize(200, 25);

*l4*.setLocation(30, 310);

box\_1.setSize(40,25);

box\_1.setLocation(250,250);

box\_2.setSize(100, 25);

box\_2.setLocation(300, 250);

box\_3.setSize(70, 25);

box\_3.setLocation(410, 250);

flag1.setSize(40,25);

flag1.setLocation(250, 310);

flag2.setSize(50, 25);

flag2.setLocation(300, 310);

add(*b*);

add(*del*);

add(*text*);

add(*area*);

add(*l1*);

add(*l2*);

add(*l3*);

add(*l4*);

add(box\_1);

add(box\_2);

add(box\_3);

add(flag1);

add(flag2);

*b*.addActionListener(new ButtonActionListener());

flag1.addActionListener(new FlagActionListener());

flag2.addActionListener(new FlagActionListener());

*del*.addActionListener(new DelActionListener());

box\_1.addActionListener(new BoxActionListener());

box\_2.addActionListener(new BoxActionListener());

box\_3.addActionListener(new BoxActionListener());

}

public class ButtonActionListener implements ActionListener {

public void actionPerformed(ActionEvent e) {

try{

if(!file.exists()){

file.createNewFile();

}

FileWriter out = new FileWriter(file, true);

try{

String text1 = *text*.getText();

String area2 = *area*.getText();

out.write(text1 + " - ");

out.write(area2 + "\n");

if(k == 1)

out.write("Отпуск медикамента осуществляется без рецепта. " + "\n");

else if(k == -1)

out.write("Отпуск медикамента осуществляется при наличии рецепта. " + "\n");

out.write(" (Дата поступления:" + boxA + " " + boxB + " " + boxC + ")");

} finally{

out.close();

}

}catch(IOException e1){

throw new RuntimeException(e1);

}

}

}

public class FlagActionListener implements ActionListener {

public void actionPerformed(ActionEvent e) {

k = 0;

if(e.getSource() == flag1){

k++;

}

else if(e.getSource() == flag2){

k--;

}

}

}

public class DelActionListener implements ActionListener{

public void actionPerformed(ActionEvent e) {

if(e.getSource() == *del*){

*text*.setText(null);

*area*.setText(null);

}

}

}

public class BoxActionListener implements ActionListener{

public void actionPerformed(ActionEvent e) {

if(e.getSource() == box\_1){

boxA = box\_1.getSelectedItem();

}

if(e.getSource() == box\_2){

boxB = box\_2.getSelectedItem();

}

if(e.getSource() == box\_3){

boxC = box\_3.getSelectedItem();

}

}

}

}

**Задания для самостоятельного выполнения**

Важно. Сначала выполнить задание из части 1, затем – задание из части 2.

Часть 1

1. Разработать приложение управления тремя списками, расположенными горизонтально. Приложение должно обеспечивать перемещение любого выбранного элемента или содержимого всего списка в следующий список по часовой стрелке: как из первого – во второй, из второго в третий, из третьего в первый. Элемент при перемещении должен исчезать из одного списка и появляться в другом. Помимо этого, приложение должно обеспечивать управление вторым списком – добавление нового элемента, редактирование, удаление.

2. Разработать приложение управления тремя списками, расположенными горизонтально. Приложение должно обеспечивать перемещение выбранного элемента из первого во второй, из второго в третий, из третьего в первый список и наоборот. Направление перемещения элемента из списка в список должно определяться выбором из набора флажков (*CheckboxGroup*). Элемент при перемещении должен исчезать из одного списка и появляться в другом. Помимо этого, приложение должно обеспечивать управление всеми списками – добавление нового элемента, редактирование, удаление.

3. Разработать приложение, обеспечивающее возможность множественного выбора элементов из списка. Выбранные элементы должны образовывать строку текста и помещаться в текстовое поле. Предусмотреть возможность вывода сообщения в диалоговое окно (*Dialog*) в случае, если суммарное количество символов будет превышать 100.

4.  Разработать приложение, реализующее калькулятор. Приложение должно иметь строку редактирования (*TextField*), набор кнопок 0…9, кнопки арифметических действий – суммирование, вычитание, деление, умножение, память.

5.  Разработать приложение, реализующее калькулятор. Приложение должно иметь две строки редактирования (*TextField*). Набор флажков (*CheckboxGroup*) определяет, какое арифметическое действие необходимо выполнить: суммирование, вычитание, деление, умножение, память.

5. Разработать приложение, обеспечивающее поиск в двух списках несовпадающих фрагментов текста. Строки, в которых будут найдены искомые фрагменты, должны быть выведены в диалогове окно (*Dialog*) (предполагается, что несколько строк может иметь такие фрагменты). Помимо этого приложение должно обеспечивать управление содержимым списков – добавление нового элемента, редактирование, удаление.

7. Разработать приложение управления тремя списками, расположенными на диалоге горизонтально. Приложение должно обеспечивать перемещение некоторого (указанного в наборе флажков (*CheckboxGroup*)) количества выбранных элементов из списка в список. Перемещение элементов осуществлять слева направо. Элемент при перемещении не исчезает, а выделяется. Помимо этого приложение должно обеспечивать заполнение помеченного флажком списка 10 строками. Предусмотреть очистку помеченного списка.

8.    Разработать приложение управления списком. Вывести два флажка (*Сheckbox*). При первом включенном флажке осуществляется выбор всех нечетных строк, при втором включенном флажке осуществляется выбор всех четных строк и перенос их в раскрывающийся список (*Choice*).

9.    Разработать приложение управления списком. Вывести два флажка (*Checkbox*). При первом включенном флажке осуществляется выбор всех нечетных строк и их удаление, при втором включенном флажке осуществляется выбор всех четных строк и перенос их во второй список. Предусмотреть обновление элементов списка и очистку второго списка.

10.  Разработать приложение, реализующее калькулятор. Приложение должно иметь три строки редактирования (*TextField*) – для двух операндов и результата. Набор флажков (*CheckboxGroup*) определяет, какое арифметическое действие необходимо выполнить: суммирование, вычитание, деление, умножение, очистку окон редактирования.

11.  Разработать приложение управления тремя списками («Фамилия», «Имя», «Отчество») и строки редактирования (*TextField*). В строку редактирования вводится информация в формате «Фамилия Имя Отчество». По завершении ввода фамилия должна появиться в списке «Фамилия», имя в списке «Имя», отчество в списке «Отчество». Предусмотреть вывод сообщения в диалогое окно (*Dialog*), если количество введенных в списки ФИО будет превышать 10.

12.  Разработать приложение управления тремя списками («Фамилия», «Имя», «Отчество») и строки редактирования (*TextField*). В строку редактирования вводится информация в формате «Фамилия Имя Отчество». По завершении ввода фамилия должна появиться в списке «Фамилия», имя в списке «Имя», отчество в списке «Отчество». Предусмотреть возможность множественного выбора фамилий или отчеств в зависимости от выбора в наборе флажков (*CheckboxGroup*) и вывода всех их в отсортированном порядке в диалоговое окно (*Dialog*).

13. Разработать приложение, обеспечивающее поиск в двух раскрывающихся списках (*Choice*) фрагмента текста. Набором флажков (*CheckboxGroup*) указывать, в каком списке будет осуществляться поиск. Строки, в которых будет найден искомый фрагмент, должны быть выделены (предполагается, что несколько строк может иметь искомый фрагмент). Помимо этого приложение должно обеспечивать управление содержимым списков – добавление нового элемента, редактирование, удаление.

14. Разработать приложение, обеспечивающее возможность множественного выбора элементов из списка. Выбранные элементы должны образовывать строку текста и выводиться в соседний список. Предусмотреть возможность вывода сообщения в диалоговое окно (*Dialog*) в случае, если суммарное количество символов будет превышать 100.

15.   Разработать приложение управления тремя списками («Фамилия», «Имя», «Отчество») и строкой редактирования (*TextField*). Для отображения строки редактирования вызывается диалоговое окно (*Dialog*). В строку редактирования вводится информация в формате «Фамилия Имя Отчество». По завершении ввода диалоговое окно закрывается, фамилия должна появиться в списке «Фамилия», имя в списке «Имя», отчество в списке «Отчество». Предусмотреть возможность множественного выбора фамилий и записи их в отсортированном порядке в четвертый список.

Часть 2

Разработать приложение с 5 (минимум) элементами управления библиотеки Swing (кнопка (*JButton*), список (*JList*), текстовое поле (*JTextField*), раскрывающий список (*JComboBox*), текстовая область (*JTextArea*), флажок (*JCheckBox*), радиокнопки (*JRadioButton*), лэйбл (*JLabel*)) исходя из задания вариантов. Подгрузку данных осуществлять из файла (при необходимости), сохранять данные также в файл. Обязательна реализация механизма закрытия окна. Также можно использовать систему подсветки данных, при желании.

1. Регистрация пластиковой карточки.
2. Регистрация поступлений в магазин игрушек (учесть дату поступления).
3. Регистрация пользователя в сети.
4. Заполнение анкеты.
5. Оформление заказа в интернет магазине.
6. Бронирование тура.
7. Оформление книги в электронной библиотеке.
8. Создание мини библиотеки автобиографий.
9. Опрос на конкретную тематику (Машины, мебели………).
10. Регистрация поступлений в продуктовый магазин (учесть дату поступления).
11. Регистрация посетителей салона по оказанию каких-то услуг.
12. Реализовать электронное меню для ресторана.
13. Создание архива бумаг\книг, занесение в базу данных о них.
14. Система бронирование авиабилета.
15. Бронирование талона в больницу с указанием причины посещения (обследование, болезнь…….), необходимые врачи для посещения и т.д.

**Контрольные вопросы**

1. Какие существуют в Java библиотеки визуальных компонентов для создания графического интерфейса пользователя?
2. Какие основные элементы управления библиотеки Swing вам известны?
3. Какое действие выполняет конструктор JFrame()?
4. Что такое менеджер размещения и для чего он служит?
5. При помощи каких методов задаются координаты и размер элементов?
6. Что такое *“источник”* события, и что им может являться?
7. Что такое ActionListener, и как метод он требует для реализации?
8. При помощи какого метода обработчик добавляется к кнопке?
9. Для чего нужны внутренние классы?
10. Какие типы внутренних классов вам известны?