

# Лабораторная работа № 3

## Создание графического пользовательского интерфейса на основе стандартных компонент

### Оглавление

Цель лабораторной работы .....	2
1 Построение окон графического пользовательского интерфейса с использованием библиотеки Swing .....	2
1.1 Классы для создания окон. Внутренняя организация окон верхнего уровня .....	2
1.2 Создание фреймов. Класс <i>JFrame</i> . .....	5
1.2 Использование панелей. Класс <i>JPanel</i> . .....	7
1.3 Управление размещением компонентов в контейнере .....	9
1.3.1 Граничная компоновка. Класс <i>BorderLayout</i> . .....	9
1.3.2 Плавающая компоновка. Класс <i>FlowLayout</i> . .....	10
1.3.3 Коробочная компоновка. Класс <i>BoxLayout</i> . .....	12
1.3.4 Табличная компоновка. Класс <i>GridLayout</i> . .....	14
1.4 Создание стандартных диалоговых окон .....	15
2 Компоненты графического пользовательского интерфейса библиотеки Swing .....	18
2.1 Создание меток. Класс <i>JLabel</i> . .....	18
2.2 Создание текстовых полей. Класс <i>TextField</i> . .....	19
2.3 Создание кнопок. Класс <i>Button</i> . .....	20
2.4 Создание радиокнопок. Класс <i>JRadioButton</i> . .....	21
2.5 Создание флажков. Класс <i>JCheckBox</i> . .....	22
2.6 Создание комбинированных списков. Класс <i>JComboBox</i> . .....	23
3 Обработка событий .....	24
3.1 События типа <i>ActionEvent</i> . .....	24
3.2 События типа <i>ItemEvent</i> . .....	25
3.3 События нажатия клавиш клавиатуры. Класс <i>KeyEvent</i> . .....	26
3.4 События мыши. Класс <i>MouseEvent</i> . .....	27
4 Базовое приложение для лабораторной работы .....	29
4.1 Структура приложения и размещение элементов интерфейса .....	29
4.2 Реализация главного окна приложения .....	31
4.2.1 Поля класса <i>MainFrame</i> . .....	31
4.2.2 Реализация конструктора класса <i>MainFrame</i> . .....	31
4.2.3 Реализация метода <i>main</i> . .....	35
5 Задания .....	37
5.1 Вариант сложности А .....	37
5.2 Вариант сложности В .....	38

5.3 Вариант сложности С.....	39
Приложение 1. Исходный код базового приложения .....	40

## Цель лабораторной работы

Получить практические навыки создания Java-приложений с графическим пользовательским интерфейсом на основе библиотеки Swing.

### 1 Построение окон графического пользовательского интерфейса с использованием библиотеки Swing

Большинство современных программ обладают графическим пользовательским интерфейсом (Graphical user interface – GUI), наличие которого делает приложение намного более удобным в использовании по сравнению с консольными приложениями.

Для построения GUI на базе Java SE могут использоваться библиотеки AWT и Swing.

**Библиотека AWT** (Abstract Window Toolkit) основана на идее о делегировании создания визуальных компонентов операционной системе (ОС), в то время как само Java-приложение должно лишь задавать их положение и поведение. Такой подход имеет следующие недостатки:

- сложность реализации компонентов GUI, по причине различий в их поведении на различных ОС;
- проблемы с реализацией некоторых компонентов по причине отсутствия их аналогов на некоторых ОС.

**Библиотека Swing** основана на идее рисования содержимого компонентов на поверхности окна и требует от ОС только возможности создавать окна и рисовать на их поверхности. Данная библиотека успешно зарекомендовала себя при создании мультиплатформенного GUI для Java-приложений. Ее компоненты располагаются в пакете *javax.swing*.

GUI приложения в рамках данной лабораторной работы будет основан на компонентах библиотеки Swing.

#### 1.1 Классы для создания окон. Внутренняя организация окон верхнего уровня.

GUI Java-приложений на основе Swing включает одно или несколько окон верхнего уровня, которые не являются вложенными в другие окна и создаются с использованием одного из классов: *JFrame*, *JWindow* и *JDialog*, включенных в иерархию, представленную на рисунке 1.

Класс *Component* является базовым для всех элементов GUI, и определяет объекты, которые:

- имеют графическое представление;
- могут отображаться на экране;
- могут взаимодействовать с пользователем.

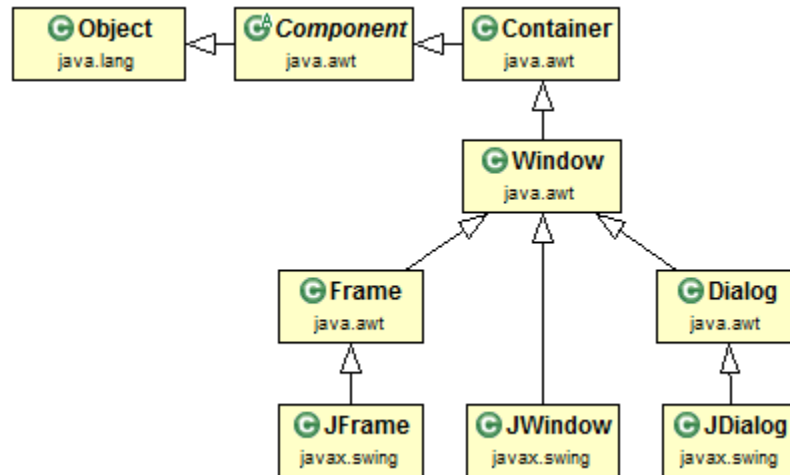


Рис 1. Иерархия классов для создания окон верхнего уровня в Java

Класс *Container* задает поведение элементов GUI, которые могут содержать другие элементы.

Классы *Frame* библиотеки AWT и *JFrame* библиотеки Swing являются основой для создания окон, которые имеют пиктограмму для системного меню, полосу заголовка для размещения названия окна, видимую границу вокруг окна и кнопки управления (для минимизации, максимизации и закрытия окна). Такие окна в Java называются *фреймами*.

Классы *Window* библиотеки AWT и *JWindow* библиотеки Swing являются основой для создания окон, которые не имеют заголовка, границы и кнопок управления.

Классы *Dialog* библиотеки AWT и *JDialog* библиотеки Swing являются основой для создания диалоговых окон, в большинстве случаев, предназначенных для ввода информации пользователем. Отличием такого вида окон от фреймов является отсутствие возможности минимизации и максимизации (нет соответствующих кнопок управления, а соответствующие пункты системного меню недоступны).

Примеры внешнего вида всех трех типов окон представлены на рисунке 2.

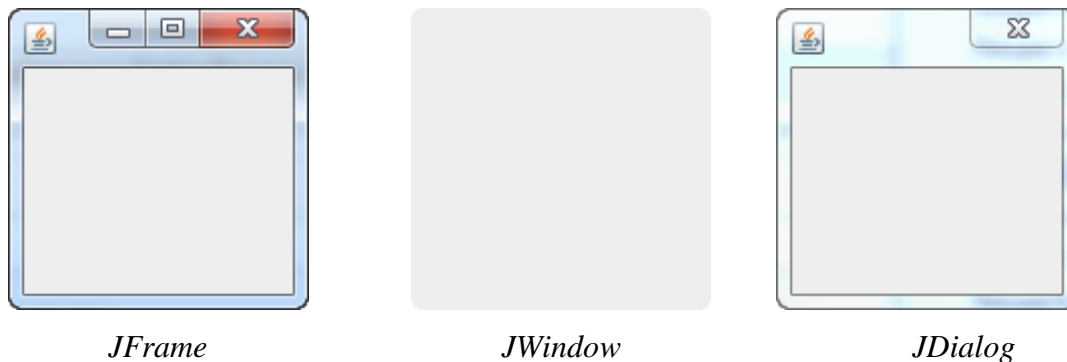


Рис 2. Пример внешнего вида окон верхнего уровня в Java

Классы библиотеки *Swing* *JFrame*, *JWindow* и *JDialog* отличаются от соответствующих суперклассов библиотеки *AWT*, тем что могут размещать внутри себя графические компоненты, принадлежащие библиотеке *Swing*.

Каждое из окон верхнего уровня библиотеки *Swing* реализует интерфейс *RootPaneContainer*, и делегирует операции, связанные с управлением внутренним содержимым *корневой панели*, которая представлена объектом класса *JRootPane*. Использование данной панели позволяет универсализировать управление подчиненными объектами графического интерфейса и таким образом избежать повторной реализации связанного с этим кода для окон различного типа. Корневая панель создается автоматически при создании окна и ссылка на ее объект может быть получена путем вызова метода *getRootPane()*. Корневая панель позволяет организовать в рамках окна структуру панелей, которая представлена на рисунке 3.

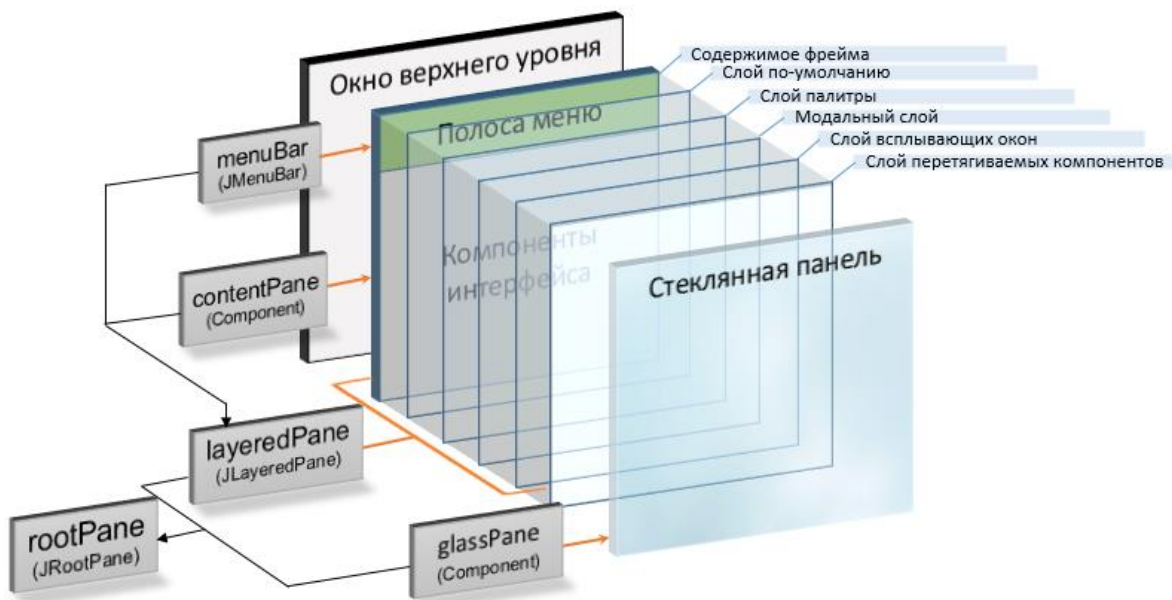


Рис 3. Структура панелей, организуемая классом *JRootPane*

Корневая панель использует для управления содержимым окна две панели: *многослойную панель* (*layeredPane* – объект класса *JLayeredPane*) и *стеклянную панель* (*glassPane* – объект класса *Component*).

Многослойная панель используется для создания набора слоев, в которых могут располагаться графические компоненты, размещенные в окне. Каждый такой слой характеризуется некоторой глубиной расположения, что позволяет отображать отдельные интерфейсные компоненты с наложением. Для практических целей наибольший интерес представляет слой содержимого фрейма (смотри рисунок 3), в котором размещаются:

- компонент *основного меню приложения*, который является необязательным, но если присутствует, то располагается вдоль верхней границы окна;
- *панель содержимого (contentPane)*, которая непосредственно предназначена для добавления интерфейсных компонентов, которые будут содержаться в окне и занимает все оставшееся пространство.

*Стеклянная панель* (по умолчанию является невидимой) располагается поверх всего остального содержимого корневой панели и дает возможность при ее переводе в видимое состояние рисовать поверх всех остальных компонентов окна, а также перехватывать входные сообщения, предназначенные для корневой панели.

## 1.2 Создание фреймов. Класс *JFrame*.

При построении GUI Java приложения наиболее часто основные окна программ создаются на базе класса *JFrame*, который позволяет управлять видом, размером и положением конструируемого окна. Некоторые методы класса *JFrame* приведены в таблице 1.

Таблица 1 Некоторые методы класса *JFrame*

Имя метода	Назначение
<i>boolean isVisible()</i> <i>void setVisible(boolean v)</i>	Проверяет ( <i>isVisible</i> )/задает ( <i>setVisible</i> ) видимость окна
<i>Point getLocation()</i> <i>void setLocation(int x, int y)</i>	Получает ( <i>getLocation</i> )/задает ( <i>setLocation</i> ) положение верхнего левого угла окна по отношению к левому верхнему углу экрана
<i>Dimension getSize()</i> <i>void setSize(int w, int h)</i> <i>void setSize(Dimension d)</i>	Получает ( <i>getSize</i> )/задает ( <i>setSize</i> ) размер окна
<i>boolean isResizable()</i> <i>void setResizable(boolean v)</i>	Проверяет ( <i>isResizable</i> )/задает ( <i>setResizable</i> ) возможность измерения размеров окна
<i>String getTitle()</i> <i>void setTitle(String title)</i>	Получает ( <i>getTitle</i> )/задает ( <i>setTitle</i> ) название окна в заголовке
<i>Image getIconImage()</i> <i>void setIconImage(Image img)</i>	Получает ( <i>getIconImage</i> )/задает ( <i>setIconImage</i> ) изображение для иконки окна

Имя метода	Назначение
<code>int getDefaultCloseOperation();</code> <code>void setDefaultCloseOperation(int dco)</code>	Получает ( <code>getDefaultCloseOperation()</code> )/задает ( <code>setDefaultCloseOperation()</code> ) операцию, выполняемую при закрытии окна. Возможные значения: <ul style="list-style-type: none"> <li>• <code>DO_NOTHING_ON_CLOSE</code> – ничего не предпринимать;</li> <li>• <code>HIDE_ON_CLOSE</code> – автоматически скрыть фрейм, без его удаления из памяти;</li> <li>• <code>DISPOSE_ON_CLOSE</code> – автоматически скрыть фрейм, и удалить его из памяти путем вызова метода <code>dispose()</code>;</li> <li>• <code>EXIT_ON_CLOSE</code> – закончить работу приложения</li> </ul>

Поскольку окна верхнего уровня позиционируются в рамках экрана, то их наиболее удобное положение и размеры зависят от настроек операционной системы (например, размеров экрана). По этой причине при создании окна необходимо взаимодействие с ОС, в которой выполняется приложение.

Для получения информации о системных настройках можно воспользоваться методами класса *java.awt.Toolkit*, часть из которых представлена в таблице 2.

Таблица 2 Некоторые методы класса *Toolkit*

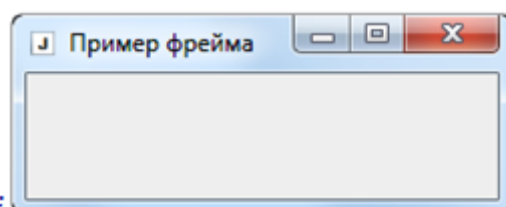
Имя метода	Назначение
<code>static Toolkit getDefaultToolkit()</code>	Возвращает ссылку на объект типа <i>Toolkit</i> , доступный по умолчанию
<code>int getScreenResolution()</code>	Возвращает разрешение экрана в точках на дюйм
<code>Dimension getScreenSize()</code>	Возвращает размер экрана в пикселях
<code>Image getImage(String filename)</code>	Возвращает объект типа <i>Image</i> , представляющий изображение, загруженное из файла <i>filename</i> , формат которого GIF, JPEG или PNG.

Ниже приведен пример программного кода приложения, состоящего из одного окна, размещенного по центру экрана и имеющего название и иконку, загруженную из файла.

```
public class MainFrame extends JFrame{
    public MainFrame() {
        super();

        //Получение сведений о размере экрана
        Toolkit toolkit = Toolkit.getDefaultToolkit();
        int scrWidth = toolkit.getScreenSize().width;
        int scrHeight = toolkit.getScreenSize().height;

        int frWidth = 300, frHeight = 200;
```



```

//Центрирование в пределах экрана и установка размера фрейма
setLocation((scrWidth - frWidth)/2, (scrHeight - frHeight)/2);
setSize(frWidth, frHeight);

//Заголовок и иконка
setTitle("Пример фрейма");
setIconImage(toolkit.getImage("icon.gif"));
}

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

```

## 1.2 Использование панелей. Класс *JPanel*.

Для удобства использования GUI, компоненты, добавляемые в панель содержимого окна, должны быть упорядочены (сгруппированы) в соответствии со своим назначением. При создании GUI на базе *Swing* отдельные подгруппы компонентов добавляются в контейнеры, в качестве которых часто выступают панели, представленные объектами класса *JPanel* и способные включать другие панели, что позволяет строить *дерево компонентов*, корнем которого является панель содержимого.

Для манипулирования компонентами класс *JPanel* предоставляет ряд методов, наследуемых от своих суперклассов *JComponent*, *Container*, и *Component*. Некоторые из них представлены в 3.

Таблица 3 Методы панели для манипулирования компонентами

Имя метода	Назначение
<i>Component add(Component c)</i> <i>Component add(Component c, int index)</i> <i>void add(Component c, Object obj)</i> <i>void add(Component c, Object obj, int index)</i>	Добавляет компонент <i>c</i> в панель. Параметр <i>index</i> задает индекс компонента в контейнере. Параметр <i>obj</i> содержит дополнительную информацию для используемого менеджера компоновки
<i>int getComponentCount()</i>	Возвращает количество компонентов в панели
<i>Component getComponent(int index)</i> <i>Component getComponentAt(int x, int y)</i> <i>Component getComponentAt(Point point)</i>	Позволяет получить ссылку на компонент, который содержится в панели либо по индексу, либо по координатам точки в рамках панели
<i>Component[] getComponents()</i>	Позволяет получить массив компонентов панели
<i>void remove(Component c)</i>	Позволяет удалять компоненты из панели



<code>void <b>remove</b>(int index)</code> <code>void <b>removeAll</b>()</code>	
--	--




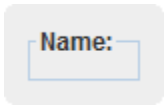
Кроме использования в качестве контейнера других компонентов, панели предоставляют возможность управлять своим внешним видом, путем вызова методов для задания *рамки* и *цвета фона*, представленных в таблице 4.

Таблица 4 Методы для управления внешним видом панелей

Имя метода	Назначение
<code>Border <b>getBorder</b>()</code> <code>void <b>setBorder</b>(Border b)</code>	Возвращает ( <i>getBorder</i> )/устанавливает ( <i>setBorder</i> ) объект типа <i>Border</i> , задающий вид рамки для компонента.
<code>Color <b>getBackground</b>()</code> <code>void <b>setBackground</b>(Color c)</code>	Возвращает ( <i>getBackground</i> )/устанавливает ( <i>setBackground</i> ) объект типа <i>Color</i> , задающий цвет фона для компонента.

Для задания рамок различного вида могут использоваться классы пакета *java.swing.border*, реализующие интерфейс *Border*, часть которых описана в таблице 5.

Таблица 5 Некоторые классы для задания формы границ компонента

Имя класса	Краткое описание	Вид границы
<i>BevelBorder</i>	Скошенная граница из двух линий	
<i>EmptyBorder</i>	Невидимая граница, имеющая определенную толщину	-
<i>EtchedBorder</i>	Выгравированная граница	
<i>LineBorder</i>	Граница в виде линии определенного цвета и толщины	
<i>TitledBorder</i>	Граница с заголовком	

Для быстрого создания объектов приведенных выше классов границ можно использовать набор статических методов класса *javax.swing.BorderFactory*. Например, для задания границы в виде линии можно использовать следующий код:

```

JPanel panel = new JPanel();
panel.setBorder(BorderFactory.createLineBorder(Color.BLUE, 2));

```



Для быстрого задания цвета, можно использовать статические поля класса *Color*, содержащие объекты для наиболее часто используемых цветов (например, *Color.black*, *Color.red*, и т.д.):

```
JPanel panel = new JPanel();  
panel.setBackground(Color.red);
```

### 1.3 Управление размещением компонентов в контейнере.

Для адаптации GUI приложения к изменению настроек ОС, размера окон, а также работе под управлением различных ОС используются **менеджеры компоновки**, управляющие расположением и абсолютными размерами компонентов в контейнере (панели содержимого окна, панелях и т.д.). Менеджеры компоновки являются объектами классов, реализующих интерфейс *java.awt.LayoutManager*. Для получения ссылки на менеджер компоновки контейнера используется метод *getLayout*, а для его задания метод *setLayout*.

Для работы любого менеджера компоновки необходимо указать:

- как компоненты располагаются относительно друг друга и границ окна;
- набор размеров для компонента (с использованием методов класса *java.awt.Component*), включающий: *минимальный размер* (*setMinimumSize*), *предпочтительный размер* (*setPreferredSize*) и *максимальный размер* (*setMaximumSize*);
- дополнительные параметры, определяемые классом используемого менеджера компоновки (если таковые есть).

На основании предоставленной информации менеджер компоновки с учетом установок ОС рассчитывает абсолютное положение и размеры каждого компонента в контейнере. Некоторые из классов менеджеров компоновки описаны в последующих подразделах.

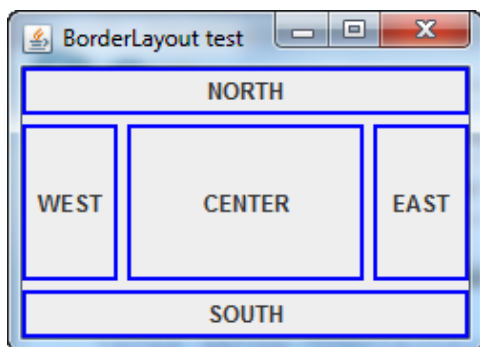
#### 1.3.1 Граничная компоновка. Класс *BorderLayout*.

Класс *java.awt.BorderLayout* позволяет расположить 5 компонент, четыре компоненты по границам контейнера и одну в центре (по умолчанию используется в панели содержимого).

Результат использования данного менеджера в окне, содержащем пять меток (которые создаются и настраиваются дополнительно написанным методом *createLabel*), для каждой из которых установлена граница в виде линии и некоторая надпись приведен на рисунке 4.

В качестве параметров конструктора класса *BorderLayout*, задаются расстояния в пикселях между компонентами по горизонтали и вертикали.

При добавлении компонентов в контейнер при помощи метода *add*, можно вторым аргументом указать непосредственное положение компонента в контейнере (при отсутствии второго аргумента компонент будет располагаться по центру).



```
JPanel p = new JPanel(new BorderLayout(5, 5));  
p.add(createLabel("NORTH"), BorderLayout.NORTH);  
p.add(createLabel("SOUTH"), BorderLayout.SOUTH);  
p.add(createLabel("WEST"), BorderLayout.WEST);  
p.add(createLabel("EAST"), BorderLayout.EAST);  
p.add(createLabel("CENTER"), BorderLayout.CENTER);  
  
setContentPane(p);
```

Рис 4. Пример использования менеджера компоновки *java.awt.BorderLayout*

Порядок расчета положения и размеров компонент:

- расчет положения начинается с верхней и нижней компонент, которые сохраняют высоту, не менее предпочтительной, а по ширине занимают все пространство контейнера;
- затем располагаются боковые компоненты, сохраняющие ширину, не менее предпочтительной, а по высоте занимающие столько, сколько составляет высота контейнера минус высоты верхней и нижней компонент минус вертикальные зазоры (они задаются в конструкторе класса *BorderLayout*);
- центральная компонента занимает оставшееся пространство в середине. Предпочтительные размеры для центральной компоненты во внимание не принимаются.

### 1.3.2 Плавающая компоновка. Класс *FlowLayout*.

Класс *FlowLayout* используется для задания «плавающей» компоновки (по умолчанию используется в *JPanel*).

Данная компоновка располагает компоненты в контейнере горизонтально друг за другом в порядке добавления в контейнер пока хватает места до границы окна, после чего начинается новая строка компонентов. При изменении размеров контейнера, менеджер автоматически обновляет размещение компонентов для заполнения доступного пространства. Если пространства по вертикали недостаточно для размещения всех строк, то последние компоненты могут оказаться либо частично, либо полностью за пределами видимой области. На рисунке 5 представлено поведение окна с «плавающей» компоновкой, установленной для панели, куда добавлены три компонента: метка, кнопка и текстовое поле.

```

JPanel p = new JPanel(new FlowLayout());
p.add(createLabel("Label"));
p.add(createButton("Button"));
p.add(createTextField("Text field"));
setContentPane(p);

```

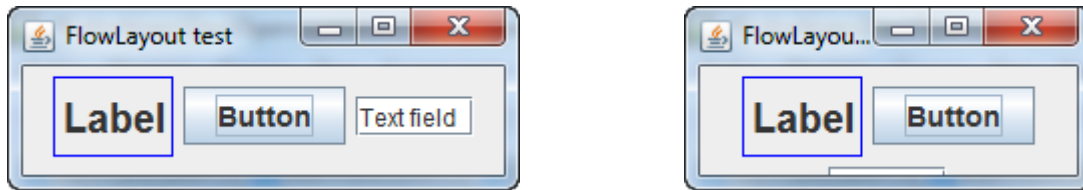


Рис 5. Поведение плавающей компоновки при различных размерах окна

Для выравнивания всей цепочки компонентов контейнера по горизонтали можно использовать режимы выравнивания, приведенные в таблице 6.

Таблица 6 Основные режимы выравнивания цепочки компонентов контейнера по горизонтали при использовании менеджера компоновки *java.awt.FlowLayout*

Имя параметра	Назначение
<i>CENTER</i>	Выравнивание по центру (по умолчанию)
<i>LEFT</i>	Выравнивание по левому краю
<i>RIGHT</i>	Выравнивание по правому краю

Для задания режима выравнивания по горизонтали используется либо один из конструкторов, принимающих его значение, либо метод *setAlignment*. Для получения текущего выравнивания используется метод *getAlignment*.

По вертикали по умолчанию компоненты каждой строки центрируются (рисунок 5). Можно установить режим выравнивания по *базовой линии*, вызвав метод *setAlignOnBaseline* объекта класса *FlowLayout*. Для компонентов, имеющих текстовое содержимое, *базовой линией* называется линия, на которой расположены нижние края его букв (без учета частей некоторых букв, таких как 'y', 'g' и т.д., уходящих ниже строки). Результат использования выравнивания по *базовой линии* приведен на рисунке 6.

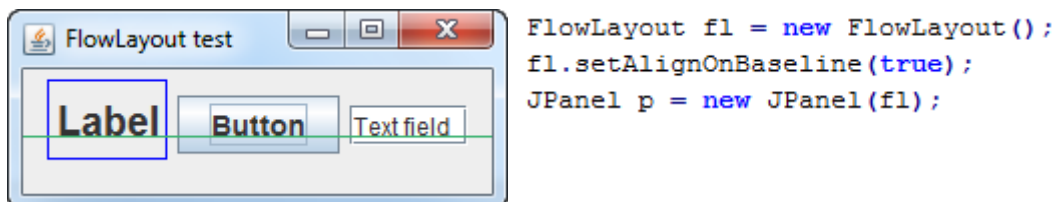


Рис 6. Выравнивание по базовой линии при использовании плавающей компоновки (базовая зеленая линия нарисована дополнительно в графическом редакторе)

Класс *FlowLayout* позволяет задавать расстояние в пикселях между компонентами в строке (метод *setHgap*) и между строками (метод *setVgap*). По умолчанию эти расстояния равны 5 пикселям. Те же расстояния могут быть заданы с использованием соответствующего конструктора. Для получения этих расстояний используется методы *getHgap* и *getVgap*, соответственно.

### 1.3.3 Коробочная компоновка. Класс *BoxLayout*.

Класс *javax.swing.BoxLayout* позволяет задать «коробочную» компоновку, которая позволяет выстроить компоненты либо в строку (горизонтальная коробочная компоновка), либо в столбец (вертикальная коробочная компоновка).

Примеры горизонтального и вертикального вариантов коробочной компоновки приведены на рисунке 7:

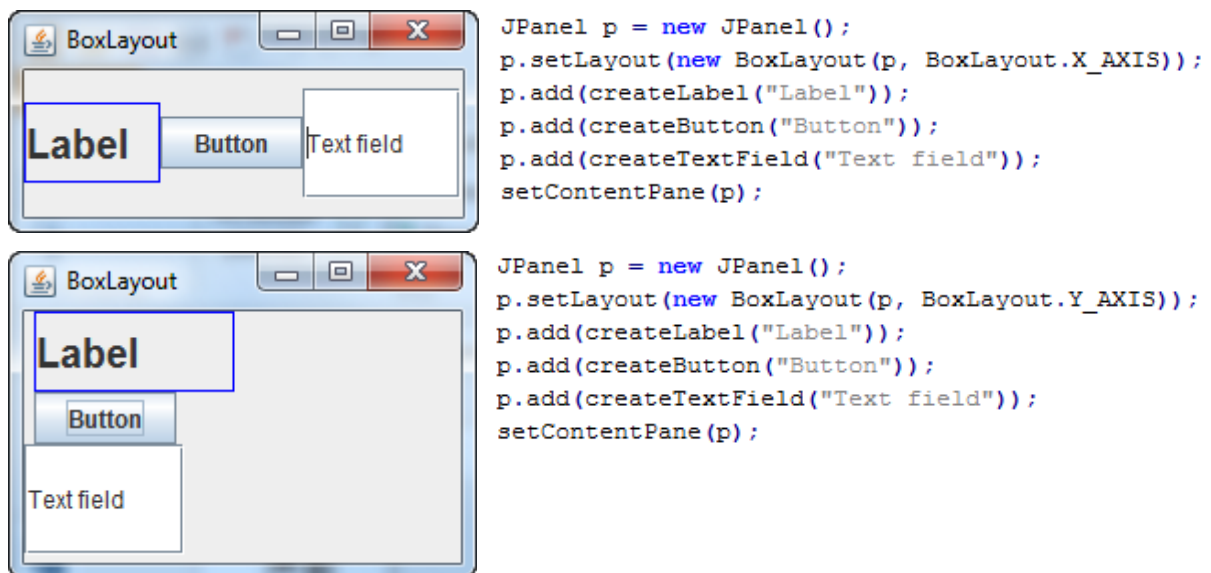


Рис 7. Горизонтальный и вертикальный варианты коробочной компоновки

Ориентация коробочной компоновки может быть задана вторым аргументом конструктора класса *BoxLayout* (рисунок 7).

Порядок вычисления размера и положения компонентов при коробочной компоновке:

- **Вдоль выбранной оси** (например, ось X при горизонтальной компоновке) компоненты по умолчанию выстраиваются вплотную друг к другу в порядке их добавления в контейнер. При расчете используется следующий алгоритм:
  1. вычисляется сумма предпочтительных размеров отдельных компонентов;
  2. если сумма предпочтительных размеров меньше соответствующего размера контейнера, то компоненты расширяются, но не более их максимального размера;
  3. если сумма предпочтительных размеров больше соответствующего размера контейнера, то компоненты сжимаются, но не менее их минимального размера;
  4. если минимальный размер достигнут, а места все равно не хватает, то часть компонент не будет показана.

- **Вдоль другой оси** (например, ось Y при горизонтальной компоновке):
  1. вычисляется максимально возможный размер для всех компонентов с учетом доступного места в контейнере и максимальных размеров отдельных компонент;
  2. делается попытка растянуть все компоненты до этого уровня;
  3. для компонентов, которые не растягиваются до этого уровня выравнивание определяется с использованием их методов *getAlignmentY* при горизонтальной компоновке и *getAlignmentX* при вертикальной компоновке. Эти методы возвращают число типа *float* в интервале [0, 1], которое используется для выравнивания компонента в рамках размера самого большого компонента в цепочке (например, при горизонтальной компоновке в рамках высоты самого высокого компонента от его нижней границы до верхней).

Для упрощения работы с коробочными компоновками можно использовать класс *javax.swing.Box*, который представляет контейнер с уже установленной коробочной компоновкой.

Для создания контейнера с предустановленной ориентацией можно использовать статические методы данного класса *createHorizontalBox()* и *createVerticalBox()*.

Для задания расстояний между компонентами в контейнере, используются специальные невидимые компоненты, которые можно создавать при помощи статических методов класса *Box*, представленных в таблице 7.

Таблица 7 Статические методы класса *Box* для создания невидимых компонент

Имя метода	Назначение
<i>Component</i> <i>createHorizontalStrut(int size)</i> <i>Component</i> <i>createVerticalStrut(int size)</i>	Возвращает ссылку на горизонтальную ( <i>createHorizontalStrut</i> )/ вертикальную ( <i>createVerticalStrut</i> ) распорку длины <i>size</i> пикселей. Распорки имеют фиксированный размер и задают фиксированное расстояние между компонентами, между которыми они располагаются в списке компонентов
<i>Component</i> <i>createRigidArea(Dimension d)</i>	Создает и возвращает ссылку на область, размер которой по горизонтали и вертикали задается параметром <i>d</i> , типа <i>Dimension</i>
<i>Component</i> <i>createHorizontalGlue()</i> <i>Component</i> <i>createVerticalGlue()</i>	Создает и возвращает ссылку на объект «клей», который будет забирать под себя все свободное пространство по горизонтали ( <i>createHorizontalGlue</i> )/ вертикали ( <i>createVerticalGlue</i> )

Пример использования класса *Box*, а также распорки и клея при горизонтальной компоновке компонентов в контейнере приведены на рисунке 8.

```

Box p = Box.createHorizontalBox();
p.add(createLabel("Label"));
p.add(Box.createHorizontalStrut(5));
p.add(createButton("Button"));
p.add(Box.createHorizontalGlue());
p.add(createTextField("Text field"));
setContentPane(p);

```

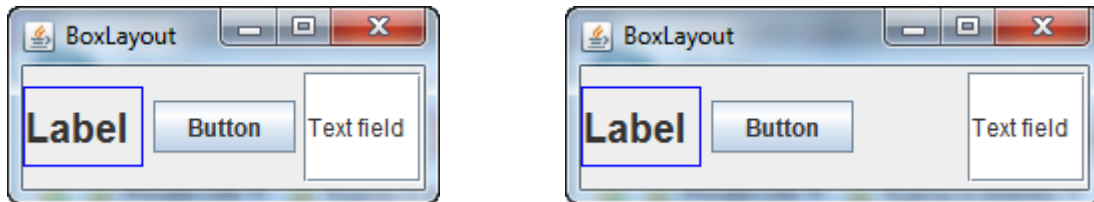


Рис 8. Использование невидимых элементов при коробочной компоновке

Как видно из рисунка 8, при растяжении окна по горизонтали расстояние между меткой и кнопкой, где расположена распорка, не изменилось, а расстояние между кнопкой и текстовым полем, где расположен клей, забрало под себя все свободное пространство, образовавшееся при растяжении.

### 1.3.4 Табличная компоновка. Класс *GridLayout*.

Класс *java.awt.GridLayout* задает табличную компоновку, при которой компоненты размещаются в ячейках таблицы строка за строкой в порядке их добавления в контейнер. Размеры компонентов в ячейке подгоняются под размеры ячейки. На рисунке 9 приведен пример использования табличной компоновки, где в качестве компонентов для панели содержимого использованы 8 меток с названиями, отражающими порядок их добавления в контейнер.

```

JPanel p = new JPanel(new GridLayout(2,2,10,5));
for(int i=0; i<8; i++)
    p.add(createLabel(i));
setContentPane(p);

```



Рис 9. Пример использования табличной компоновки

При вызове конструктора класса *GridLayout* первых два параметра задают *предпочтительное число строк и столбцов* в таблице, а вторые два параметра расстояние между компонентами в строках и столбцах, выраженное в пикселях.



Действительное число строк и столбцов в таблице вычисляется по следующим правилам:

1. если предпочтительное число строк не равно 0, то действительное число строк равно предпочтительному, а действительное число столбцов вычисляется на основании заданного числа строк и количества компонентов в контейнере;
2. если предпочтительное число строк равно 0, то действительное число столбцов равно предпочтительному, а действительное число строк вычисляется на основании заданного числа столбцов и количества компонентов в контейнере.

Задание предпочтительных значений как строк, так и столбцов равными 0 недопустимо и приводит к генерации исключения.

## 1.4 Создание стандартных диалоговых окон

Для создания простейших диалоговых окон можно использовать статические методы класса *JOptionPane* представленные в таблице 8.

Таблица 8 Методы класса *JOptionPane* для создания диалоговых окон

Название метода	Назначение
<i>showMessageDialog</i>	выводит на экран модальное диалоговое окно сообщения, которое может содержать три компонента: пиктограмму, некоторое сообщение и единственную кнопку ОК
<i>showConfirmDialog</i>	выводит на экран модальное диалоговое окно подтверждения, которое предлагает пользователю сделать выбор одного из возможных действий, от которых может зависеть дальнейшее поведение программы
<i>showOptionDialog</i>	выводит на экран модальное диалоговое окно, аналогичное методу <i>showConfirmDialog</i> , но с возможностью задавать любой набор кнопок
<i>showInputDialog</i>	выводит на экран модальное диалоговое окно, которое позволяет пользователю либо ввести некоторую строку, либо выбрать ее из множества предложенных вариантов

Класс *JOptionPane* содержит несколько вариантов каждого из методов, представленных в таблице 8. Определения каждого из этих методов с максимальным числом аргументов имеют вид:

```
static void showMessageDialog(Component parentComponent, Object message,  
                             String title, int messageType, Icon icon);  
static int showConfirmDialog(Component parentComponent, Object message,  
                             String title, int optionType, int messageType, Icon icon);  
static int showOptionDialog(Component parentComponent, Object message,  
                             String title, int optionType, int messageType, Icon icon,  
                             Object[] options, Object initialValue);
```







```
static Object showInputDialog( Component parentComponent, Object message,
    String title, int messageType, Icon icon, Object[] selectionValues,
    Object initialValue);
```

Аргументы с одинаковым именем имеют одинаковое назначение во всех методах, приведенных в таблице 8. Краткое описание аргументов представлено ниже:

- *parentComponent* – ссылка на родительский компонент;
- *message* – ссылка на объект, отображаемый в области сообщения. Чаще всего это ссылка на объект класса *String*, однако в общем случае можно передавать и ссылки на объекты другого типа, например, *Icon* для отображения картинки;
- *title* – строка, задающая заголовок окна;
- *messageType* – целое число, задающее вид пиктограммы, отображаемой в окне, в случае если аргумент *icon* не задан или равен *null*. Константы, передаваемые в качестве аргумента *messageType* приведены в таблице 9.

Таблица 9 Значения аргумента *messageType* и соответствующие им пиктограммы для стиля Java

Значение	Пиктограмма
<i>ERROR_MESSAGE</i>	
<i>INFORMATION_MESSAGE</i>	
<i>WARNING_MESSAGE</i>	
<i>QUESTION_MESSAGE</i>	
<i>PLAIN_MESSAGE</i>	Пиктограмма отсутствует

- *icon* – позволяет задать нестандартное изображение к качестве пиктограммы;
- *optionType* – определяет один из стандартных наборов кнопок, отображаемых в диалоговом окне, если иной набор не задан массивом *options*. Возможные значения *optionType* приведены в таблице 10.

Таблица 10 Значения аргумента *optionType*

Значение	Набор кнопок
<i>DEFAULT_OPTION</i>	Одна кнопка <i>Ok</i>
<i>YES_NO_OPTION</i>	Кнопки <i>Yes/No</i>
<i>YES_NO_CANCEL_OPTION</i>	Кнопки <i>Yes/No/Cancel</i>
<i>OK_CANCEL_OPTION</i>	Кнопки <i>Ok/Cancel</i>

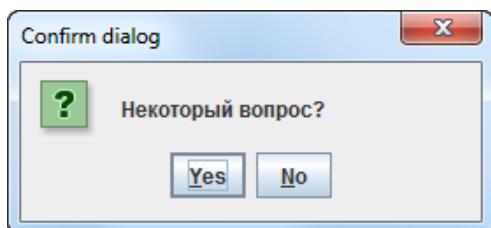
- *options* – массив, позволяющий задать количество и содержимое (название, изображение или непосредственно компонент) кнопок диалогового окна;
- *initialValue* – является ссылкой на один из объектов массива *options* и позволяет выбрать кнопку, на которой будет изначально находиться фокус после показа диалогового окна;
- *selectionValues* – определяет вид области для ввода значений. Если данный аргумент равен *null*, в области ввода значений диалога будет отображаться текстовое поле, в противном случае область ввода будет отображать выпадающий список, а элементы массива *selectionValues* будут использоваться в качестве его возможных значений;
- *initialSelectionValue* – задает либо текст по умолчанию показанный в поле ввода, либо ссылается на строку, выбранную в выпадающем списке.

Примеры использования описанных выше методов и внешний вид созданных диалоговых окон приведены на рисунках 10 – 13.



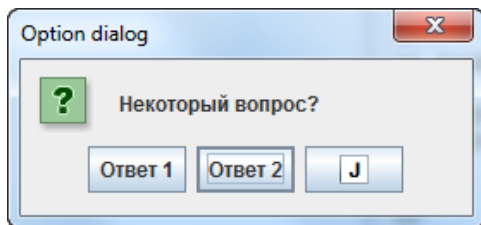
```
JOptionPane.showMessageDialog(
    null, "Some message",
    "Message dialog",
    JOptionPane.INFORMATION_MESSAGE);
```

Рис 10. Пример использования метода *showMessageDialog*



```
JOptionPane.showConfirmDialog(
    null, "Некоторый вопрос?", "Confirm dialog",
    JOptionPane.YES_NO_OPTION,
    JOptionPane.QUESTION_MESSAGE);
```

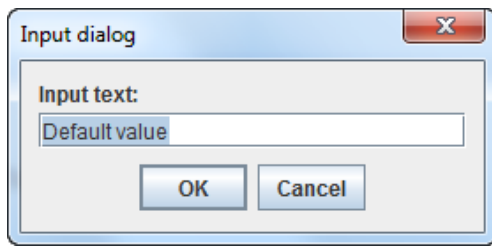
Рис 11. Пример использования метода *showConfirmDialog*



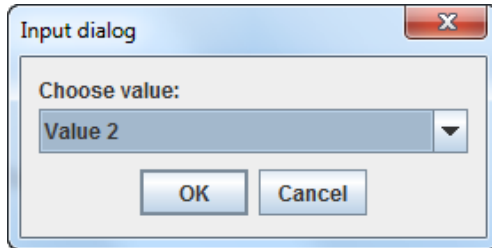
```
Object[] btns = new Object[3];
btns[0] = "Ответ 1";
btns[1] = "Ответ 2";
btns[2] = new ImageIcon(
    Toolkit.getDefaultToolkit().getImage("icon.gif"));

JOptionPane.showOptionDialog(
    null, "Некоторый вопрос?", "Option dialog",
    JOptionPane.YES_NO_OPTION,
    JOptionPane.QUESTION_MESSAGE, null, btns, btns[1]);
```

Рис 12. Пример использования метода *showOptionDialog*



```
JOptionPane.showInputDialog(  
    null, "Input text:", "Input dialog",  
    JOptionPane.PLAIN_MESSAGE, null, null,  
    "Default value");
```



```
Object[] values = new Object[3];  
values[0] = "Value 1";  
values[1] = "Value 2";  
JOptionPane.showInputDialog(  
    null, "Choose value:", "Input dialog",  
    JOptionPane.PLAIN_MESSAGE, null, values,  
    values[1]);
```

Рис 13. Примеры диалоговых окон, созданных при помощи метода *showInputDialog*

Для создания других стандартных диалоговых окон можно использовать следующие классы:

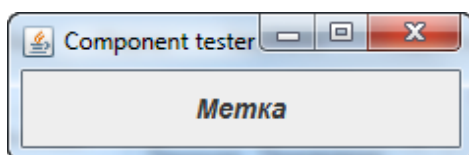
- Класс *JFileChooser* – используется для показа модального диалогового окна, позволяющего выбрать файлы;
- Класс *JColorChooser* используется для отображения модального окна выбора цвета.

## 2 Компоненты графического пользовательского интерфейса библиотеки Swing

Библиотека Swing содержит ряд готовых компонент для построения GUI, которые представлены в виде классов, унаследованных от класса *JComponent*.

### 2.1 Создание меток. Класс *JLabel*.

Класс *JLabel* позволяет создавать надписи и изображения, которые не могут быть изменены пользователем. Текст надписи (объект типа *String*) и/или изображение (объект типа *Icon*) могут быть переданы как параметры конструктора, а для их изменения после создания объекта метки могут использоваться методы *setText* и *setIcon*, соответственно. Внешний вид текста, отображаемого в метке, зависит от выбранного шрифта, который может быть изменен путем вызова метода *setFont* (рисунок 14).



```
JLabel comp = new JLabel("Метка");  
comp.setFont(new Font("SansSerif",  
    Font.BOLD|Font.ITALIC, 14));
```

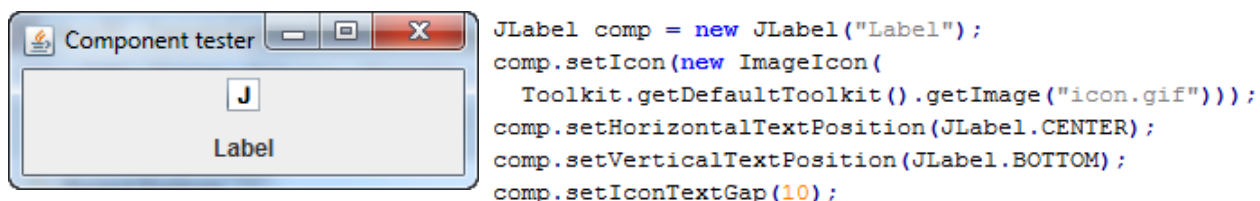


Рис 14. Примеры создания меток

Выравнивание всего содержимого в рамках метки можно задавать методами *setHorizontalAlignment* и *setVerticalAlignment*. Кроме того, можно задавать положение текста относительно изображения (методы *setHorizontalTextPosition* и *setVerticalTextPosition*), а также расстояние в пикселях между текстом и изображением (метод *setIconTextGap*).

Для оформления содержимого метки также допускается использовать HTML разметку, как показано в примере на рисунке 15.

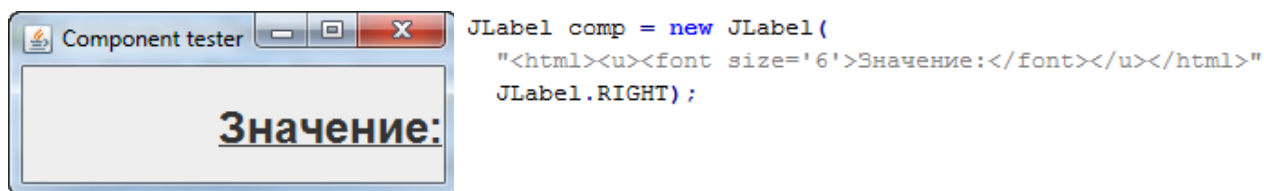


Рис 15. Использование HTML разметки для оформления содержимого метки

## 2.2 Создание текстовых полей. Класс *JTextField*.

Класс *JTextField* позволяет создать текстовое поле для ввода одной строки. Строковое значение, для размещения в текстовом поле может быть задано как аргумент конструктора или при помощи метода *setText*. Получить содержимое поля в виде строки можно при помощи метода *getText*. Класс содержит также конструктор, позволяющий задать вторым аргументом количество символов, которые будут одновременно видны в текстовом поле. Эту же величину можно задать при помощи метода *setColumns*. Метод *setEditable* позволяет задать разрешено ли пользователю редактировать значение поля. Примеры создания текстовых полей приведены на рисунке 16. Выравнивание содержимого тестового поля по горизонтали можно задавать при помощи метода *setHorizontalAlignment*.

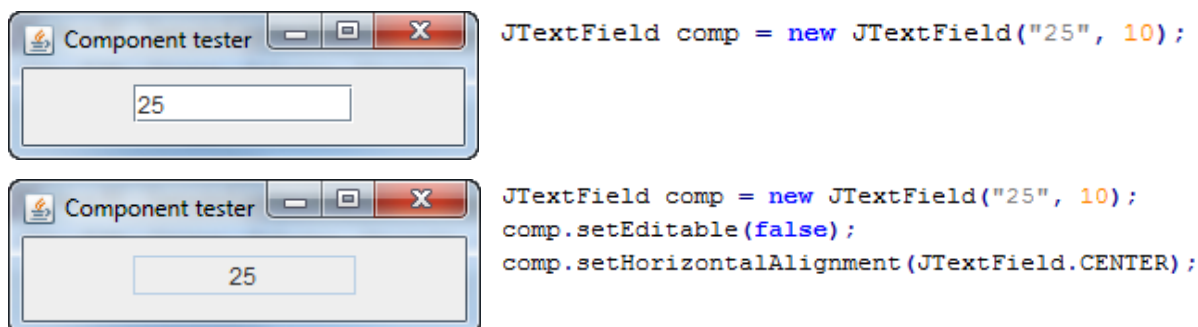
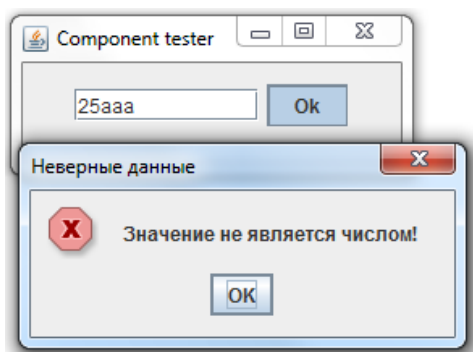


Рис 16. Примеры создания текстовых полей

Для организации проверки правильности введенного пользователем значения можно использовать метод *setInputVerifier*, куда в качестве аргумента передается ссылка на объект класса, унаследованного от абстрактного класса *InputVerifier*. Пример создания такого объекта и его использования с текстовым полем, предназначенным для ввода чисел, приведен на рисунке 17. В приведенном примере, если пользователь после ввода текста в текстовое поле пытается выйти из редактирования (например, нажать кнопку “Ok”), то управление передается методу *verify* класса *NumberVerifier*, который возвращает *true*, если введенная строка является числом, и выдает сообщение об ошибке и возвращает *false* в противном случае. Если данный метод вернул значение *false*, то фокус принудительно возвращается в текстовое поле.



```
class NumberVerifier extends InputVerifier {
    public boolean verify(JComponent input) {
        JTextField tf = (JTextField) input;
        try{
            Double.valueOf(tf.getText());
        }
        catch(NumberFormatException ex){
            JOptionPane.showMessageDialog(null,
                "Значение не является числом!",
                "Неверные данные",
                JOptionPane.ERROR_MESSAGE);
            return false;
        }

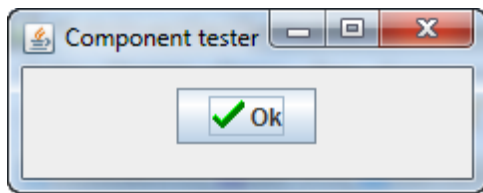
        return true;
    }
}

...
JTextField comp = new JTextField("25", 10);
comp.setInputVerifier(new NumberVerifier());
...
```

Рис 17. Пример использования проверки введенных данных для *JTextField*

### 2.3 Создание кнопок. Класс *JButton*.

Класс *JButton* позволяет создать прямоугольную кнопку, содержащую надпись и/или иконку. Кнопка поддерживает управление взаимным положением надписи и иконки аналогично тому как это было описано для класса *JLabel* (методы *setHorizontalTextPosition*, *setVerticalTextPosition* и *setIconTextGap*). Пример создания кнопки приведен на рисунке 18.



```
JButton comp = new JButton("Ok");
comp.setIcon(new ImageIcon(
    Toolkit.getDefaultToolkit().getImage(
        "OkImage.gif")));
comp.setHorizontalTextPosition(JLabel.RIGHT);
comp.setVerticalTextPosition(JLabel.CENTER);
comp.setIconTextGap(2);
```

Рис 18. Пример создания кнопки

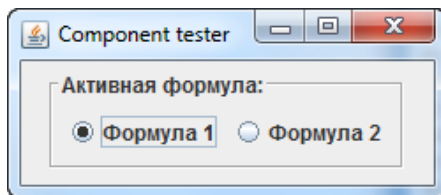
Для организации реакции приложения на нажатие кнопки пользователем необходимо создать и добавить к кнопке специальный объект-слушатель, реализующий интерфейс *ActionListener*, который будет выполнять необходимые действия. Данный объект добавляется с помощью метода *AddActionListener*. Для получения более подробной информации об организации слушателей смотри раздел 3.

## 2.4 Создание радиокнопок. Класс *JRadioButton*.

Класс *JRadioButton* создает радиокнопку круглой формы, которая может находиться в выбранном, либо не выбранном состоянии. Радиокнопки обычно используются в приложениях не по отдельности, а в наборе из нескольких штук, объединенных в группу. В рамках такой группы может быть выбрана только одна радиокнопка, при выборе которой состояние остальных радиокнопок группы переключается в невыбранное. Тем самым в рамках графического интерфейса реализуется возможность выбирать одну установку из нескольких возможных. Для объединения набора радиокнопок в группу в Java используется специальный объект класса *ButtonGroup*, который при выборе пользователем одной из радиокнопок обеспечивает перевод остальных в невыбранное состояние.

Пример создания группы радиокнопок приведен на рисунке 19. В данном примере создается две радиокнопки, которые далее для обеспечения их совместной работы добавляются в объект *bg* класса *ButtonGroup*. Метод *bg.setSelected* обеспечивает перевод первой радиокнопки в выбранное состояние по умолчанию. Стоит отметить, что объект *bg* отвечает исключительно за состояние радиокнопок, а для их отображения на экране они должны быть как обычно добавлены в контейнер (объект *comp* в приведенном примере).





```
JRadioButton btn1 =
    new JRadioButton("Формула 1");
JRadioButton btn2 =
    new JRadioButton("Формула 2");

ButtonGroup bg = new ButtonGroup();
bg.add(btn1);
bg.add(btn2);
bg.setSelected(btn1.getModel(), true);

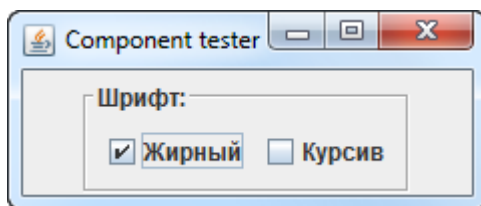
JPanel comp = new JPanel();
comp.setBorder(
    new TitledBorder(
        new EtchedBorder(),
        "Активная формула:"));
comp.add(btn1);
comp.add(btn2);
```

Рис 19. Пример создания группы радиокнопок

Для организации реакции на нажатие радиокнопки (помимо изменения состояния самих радиокнопок) необходимо создавать и добавлять к ней слушатель событий типа *ActionListener*.

## 2.5 Создание флажков. Класс *JCheckBox*.

Класс *JCheckBox* позволяет создавать интерфейсный компонент в виде флажка. Такого рода элементы обычно используются для задания отдельных параметров, для которых возможны два значения: “да” (флажок установлен) / “нет” (флажок не установлен). Пример фрагмента кода программы, использующей флажки приведен на рисунке 20.



```
JCheckBox bold =
    new JCheckBox("Жирный", true);
JCheckBox italic =
    new JCheckBox("Курсив", false);

JPanel comp = new JPanel();
comp.setBorder(new TitledBorder(
    new EtchedBorder(), "Шрифт:"));
comp.add(bold);
comp.add(italic);
```

Рис 20. Пример использования флажков

В приведенном на рисунке 20 примере используются конструкторы класса *JCheckBox*, позволяющие задать для каждого флажка надпись и его первоначальное состояние (*true* – отмечен, *false* – не отмечен). Как надпись, так и состояние флажка могут в дальнейшем изменяться программно с использованием методов *setText* и *setSelected* соответственно. Для выполнения программного кода при смене состояния флажка, этот программный код



должен быть написан в классе-слушателе, реализующем интерфейс *ActionListener*, объект которого добавляется к флажку с использованием метода *addActionListener*, либо реализующем интерфейс *ItemListener*, объект которого добавляется к флажку с использованием метода *addItemListener*.

## 2.6 Создание комбинированных списков. Класс *JComboBox*.

Класс *JComboBox* используется для создания комбинированных списков, которые являются комбинацией текстового поля и выпадающего списка возможных значений. Такие списки предназначены для того, чтобы позволить пользователю выбирать один из возможных вариантов для некоторой настройки. Пример создания комбинированного списка приведен на рисунке 21. Как видно из данного рисунка добавление элементов в выпадающий список объекта класса *JComboBox* выполняется с помощью метода *addItem*. Комбинированный список может работать в двух режимах: с возможностью редактирования значения в текстовом поле и без такой возможности. Если возможность вводить значение, которого нет в выпадающем списке, необходима, то такой режим работы может быть задан путем вызова метода *setEditable(true)*. Для получения и задания индекса элемента, выбранного в комбинированном списке, используются методы *getSelectedIndex* и *setSelectedIndex*.

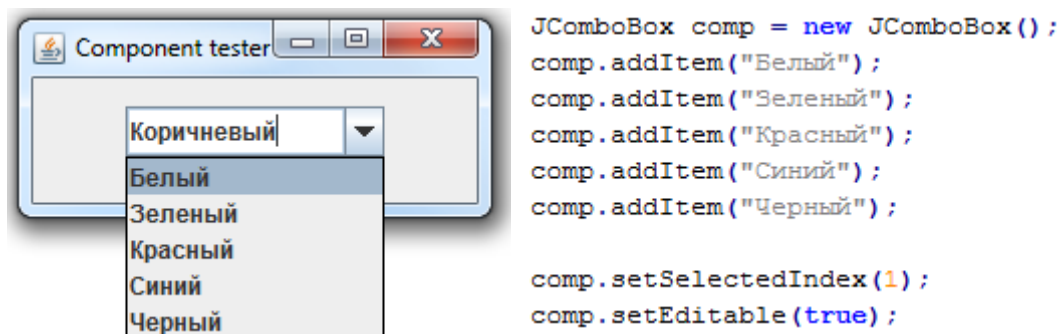


Рис 21. Пример использования комбинированного списка

Для отслеживания изменения выбранного значения в комбинированном списке, можно использовать объекты-слушатели, реализующие интерфейс *ActionListener* (метод для добавления *addActionListener*), либо интерфейс *ItemListener* (метод для добавления *addItemListener*). Первый из слушателей будет получать событие при изменении значения в текстовом поле (если оно редактируемое, то после завершения редактирования). Второй слушатель срабатывает при изменении выбранного значения, причем количество событий при этом два: одно, содержащее информацию об объекте, с которого убирается выделение, а второе – об объекте, который становится выделенным.

## 3 Обработка событий

Для организации реакции Java-приложений с графическим пользовательским интерфейсом, на действия пользователя в программах на Java используется *модель делегирования обработки событий*. При описании принципов, заложенных в основу данной модели, используются следующие понятия:

- *событие* – объект класса, унаследованного от *java.util.EventObject*, содержащий сведения о произошедшем событии;
- «*слушатель*» – объект класса, реализующего специальный интерфейс слушателя событий, выполняющий определенные действия при возникновении события;
- *источник событий* – объект, регистрирующий заинтересованных слушателей и передающий им ссылки на объект, описывающий произошедшее событие.

Цикл обработки событий в рамках модели делегирования обработки событий включает следующие этапы:

1. В источнике событий (*eventSourceObject*) регистрируются ссылки на объекты-слушатели данного события (*eventListenerObject*), которые включаются во внутренний список источника. Это делается при помощи программного кода, аналогичного следующему шаблону:

```
eventSourceObject.addEventListener(eventListenerObject);
```

В качестве источника событий часто выступают объекты, соответствующие компонентам графического пользовательского интерфейса (кнопки, пункты меню, и т.д.);

2. При возникновении события источник создает объект соответствующего класса, описывающего данное событие;
3. Источник событий вызывает метод обработки события, для каждого зарегистрированного в нем слушателя, передавая в данный метод ссылку на объект события в качестве аргумента. Название вызываемого метода слушателя определяется, интерфейсом слушателя.

В основе иерархии классов событий располагается класс *java.util.EventObject*, который является суперклассом для всех событий. Конструктор данного класса принимает в качестве параметра ссылку на объект, генерирующий событие. Далее ссылка на источник события может быть получена путем вызова метода *getSource*.

### 3.1 События типа *ActionEvent*.

Класс *ActionEvent* представляет часто используемое событие, происходящее при нажатии различных кнопок, выборе пунктов меню и элементов списка. Для получения информации о событии, можно использовать методы, реализованные в классе *ActionEvent*, которые представлены в таблице 11.

Таблица 11 Некоторые методы класса *ActionEvent*

Название метода	Описание
<i>String</i> <b>getActionCommand()</b>	Возвращает значение типа <i>String</i> , ассоциированное с выполненным действием. Данная строка может быть задана путем вызова метода <i>setActionCommand</i> , объекта, генерирующего событие
<i>int</i> <b>getModifiers()</b>	Возвращает значение, задающее набор клавиш-модификаторов (из набора Alt, Ctrl, Shift), которые были нажаты при генерации события. Для извлечения информации о том, была ли нажата отдельная клавиша, используются соответствующие константы, определенные в классе <i>ActionEvent</i> . Например, для определения того, была ли нажата клавиша Shift, используется выражение:  <code>actionEvent.getModifiers() &amp; ActionEvent.SHIFT_MASK</code>
<i>long</i> <b>getWhen()</b>	Возвращает время, когда произошло событие

Для обработки данного события класс слушателя должен реализовывать интерфейс *ActionListener* и переопределять его единственный абстрактный метод *actionPerformed*. Ниже приведен пример создания слушателя для кнопки.

```
public class MyListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        //Код, вызываемый при обработке события
    }
}
...
JButton btn = new JButton("Нажми меня");
btn.addActionListener(new MyListener());
...
```

### 3.2 События типа *ItemEvent*.

Класс *ItemEvent* представляет событие, которое генерируется, если некоторый элемент стал выбранным, либо наоборот не выбранным. Примерами интерфейсных компонентов, генерирующих подобные, события являются флажки (класс *JCheckBox*) и комбинированные списки (класс *JComboBox*). Интерфейсные элементы, способные генерировать события типа *ItemEvent*, должны реализовывать интерфейс *ItemSelectable*. Методы, предоставляемые классом *ItemEvent*, для получения информации о произошедшем событии приведены в таблице 12

Таблица 12 Некоторые методы класса *ItemEvent*

Название метода	Описание
<i>Object</i> <b>getItem()</b>	Возвращает ссылку на элемент, состояние которого изменилось
<i>ItemSelectable</i> <b>getItemSelectable()</b>	Возвращает ссылку на объект, вызвавший событие
<i>int</i> <b>getStateChange()</b>	Возвращает тип измерения состояния: выбран, либо не выбран

Для обработки событий типа *ItemEvent*, используются объекты-слушатели, класс которых реализует интерфейс *ItemListener* и переопределяет его абстрактный метод *itemStateChanged*. Фрагмент программного кода, где анонимный слушатель события типа *ItemEvent* создается для комбинированного списка, приведен ниже.

```
JComboBox cb = new JComboBox();
cb.addItemListener(new ItemListener() {
    public void itemStateChanged(ItemEvent arg0) {
        System.out.println("Item: " + arg0.getItem());
    }
});
```

### 3.3 События нажатия клавиш клавиатуры. Класс *KeyEvent*.

Класс *KeyEvent* представляет событие, которое генерируется при нажатии клавиши на клавиатуре. Событие генерируется только тогда, когда компонент пользовательского интерфейса находится в фокусе. При нажатии и отпускании некоторой клавиши на клавиатуре генерируется сразу несколько событий, представленных классом *KeyEvent* и идентифицируемых отдельными константами, определенными в этом классе и приведенными в таблице 13.

Таблица 13 Типы событий, представимые классом *KeyEvent*

Идентификатор события	Описание события
<i>KEY_PRESSED</i>	генерируется если была нажата некоторая клавиша
<i>KEY_TYPED</i>	генерируется только при нажатии клавиши, соответствующей некоторому символу <i>Unicode</i> . В остальных случаях (например, если была нажата клавиша Alt, F1, Ctrl, и т. д.) данное событие не генерируется
<i>KEY_RELEASED</i>	генерируется если была отпущена некоторая клавиша

Наиболее полезными с практической точки зрения методами класса *KeyEvent* являются методы, представленные в таблице 14.

Таблица 14 Некоторые методы класса *KeyEvent*

Название метода	Описание
<i>char getKeyChar()</i>	позволяет вернуть <i>Unicode</i> символ, введенный с клавиатуры. Если клавише не соответствует символ, то метод возвращает <i>CHAR_UNDEFINED</i>
<i>int getKeyCode()</i> <i>int getExtendedKeyCode()</i>	возвращает целочисленный код нажатой клавиши при обработке событий с идентификаторами <i>KEY_PRESSED</i> и <i>KEY_RELEASED</i> . Для событий с <i>KEY_TYPED</i> , возвращается значение <i>VK_UNDEFINED</i> . Для метода <i>getExtendedKeyCode</i> возвращаемое значение зависит от раскладки клавиатуры.

Название метода	Описание
<i>int</i> <i>getExtendedKeyCodeForChar(int c)</i>	статический метод, который возвращает целочисленный код для символа <i>Unicode</i> , переданного в качестве аргумента. Может использоваться при идентификации значения, возвращаемого методами <i>getKeyCode</i> / <i>getExtendedKeyCode</i>
<i>String</i> <i>getKeyText(int keyCode)</i>	статический метод, который возвращает строковое описание, соответствующее коду <i>keyCode</i>

С целью обработки событий типа *KeyEvent* используются классы, реализующие интерфейс *KeyListener*, который содержит три абстрактных метода *keyPressed*, *keyTyped* и *keyReleased*. Каждый из этих методов вызывается при обработке событий типа *KeyEvent* с одноименными константами (смотри таблицу 13). Поскольку часто требуется использовать только некоторые из трех перечисленных выше методов интерфейса *KeyListener*, то для упрощения написания классов для слушателей можно использовать вспомогательный класс *KeyAdapter*, который реализует интерфейс *KeyListener* и содержит пустые реализации всех его методов. Порождая класс слушатель от *KeyAdapter* достаточно переопределить лишь те методы, которые действительно необходимо использовать. Пример создания и добавления слушателя клавиатуры, в котором используется анонимный класс, порожденный от *KeyAdapter* приведен ниже.

```
//Получаем ссылку на панель содержимого окна
Container cp = getContentPane();
//Устанавливаем для нее возможность получать фокус
cp.setFocusable(true);
//Добавляем слушателя на отпускание клавиши
cp.addKeyListener(new KeyAdapter() {
    public void keyReleased(KeyEvent e) {
        //Выводим описание отпущенной клавиши в консоль
        System.out.println(KeyEvent.getKeyText(e.getExtendedKeyCode()));
    }
});
```

### 3.4 События мыши. Класс *MouseEvent*.

Класс *MouseEvent* используется для генерации событий мыши. События такого типа генерируются компонентами пользовательского интерфейса при перемещениях мыши в рамках компонента и нажатиях ее кнопок. В зависимости от действий пользователя возможна генерация событий типа *MouseEvent* с идентификаторами, представленными в таблице 15.

Таблица 15 Типы событий, представимые классом *MouseEvent*

Идентификатор события	Описание события
<i>MOUSE_MOVED</i>	генерируется при перемещении мыши
<i>MOUSE_DRAGGED</i>	генерируется если мышь перемещается и при этом нажата кнопка мыши
<i>MOUSE_ENTERED</i>	генерируется если указатель мыши был введен в компонент
<i>MOUSE_EXITED</i>	генерируется если указатель мыши покинул компонент
<i>MOUSE_PRESSED</i>	генерируется если была нажата кнопка мыши
<i>MOUSE_RELEASED</i>	генерируется если была отпущена кнопка мыши
<i>MOUSE_CLICKED</i>	генерируется при щелчке (нажатии и последующем отпускании) кнопкой мыши на компоненте

При обработке с событий мыши наиболее часто оказываются востребованными методы класса *MouseEvent*, приведенные в таблице 16.

Таблица 16 Некоторые методы класса *MouseEvent*

Название метода	Описание
<i>int getButton()</i>	Возвращает идентификатор кнопки мыши, которая поменяла свое состояние
<i>int getClickCount()</i>	Возвращает число нажатий мыши, ассоциированное с данным событием
<i>Point getPoint()</i> <i>int getX()</i> <i>int getY()</i>	Позволяют получить координаты точки, где находился указатель мыши в момент генерации события, рассчитанные относительно левого верхнего угла компонента
<i>int getXOnScreen()</i> <i>int getYOnScreen()</i>	Позволяют получить координаты точки в рамках экрана, где находился указатель мыши в момент генерации события
<i>boolean isPopupTrigger()</i>	Позволяет определить может ли данное событие вызвать показ всплывающего меню, связанного с компонентом

Для обработки событий мыши класс слушателя должен реализовывать следующие интерфейсы:

- *MouseListener* – для событий с идентификаторами *MOUSE\_ENTERED*, *MOUSE\_EXITED*, *MOUSE\_PRESSED*, *MOUSE\_RELEASED* и *MOUSE\_CLICKED*;
- *MouseMotionListener* – для событий с идентификаторами *MOUSE\_MOVED* и *MOUSE\_DRAGGED*.

Методы, определенные в данных интерфейсах, имеют названия аналогичные соответствующим идентификаторам. Для предотвращения необходимости реализовывать все методы, определенные в вышеуказанных интерфейсах, можно порождать класс слушателя от суперкласса *MouseAdapter*, который содержит пустые реализации этих методов. Пример создания и добавления слушателя мыши, в котором используется анонимный класс, порожденный от *MouseAdapter* приведен ниже.



```

//Получаем ссылку на панель содержимого окна
Container cp = getContentPane();
//Добавляем слушателя на перемещение мыши
cp.addMouseMotionListener(new MouseAdapter(){
    public void mouseMoved(MouseEvent e){
        //Выводим координаты мыши в консоль
        System.out.println("x: " + e.getX() + " ; y: " + e.getY());
    }
});

```

## 4 Базовое приложение для лабораторной работы

**Задание:** составить программу с графическим пользовательским интерфейсом для вычисления значений функции одного аргумента. Вид главного окна приложения представлен на рисунке 22.

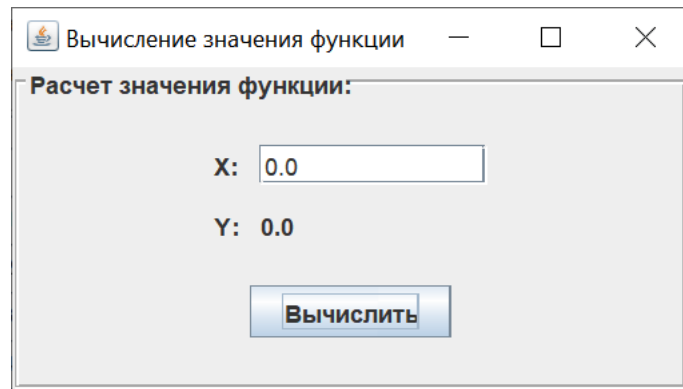


Рис 22. Внешний вид главного окна приложения

### 4.1 Структура приложения и размещение элементов интерфейса

Базовое приложение в данной лабораторной работе содержит единственный класс *MainFrame*, который унаследован от класса *javax.swing.JFrame* и предназначен для создания и отображения на экране главного окна графического пользовательского интерфейса. Программный код в рамках класса *MainFrame* выполняет следующие задачи:

- создают и настраивают все элементы графического пользовательского интерфейса, которые будут содержаться в окне;
- организует расположение элементов графического пользовательского интерфейса в рамках окна;
- обеспечивает взаимодействие элементов графического пользовательского интерфейса с пользователем;
- обеспечивает отображение главного окна приложения вместе с его внутренним содержимым на экране при запуске приложения на выполнение.



Для размещения элементов интерфейса в пространстве фрейма использован установленный в нем по умолчанию менеджер «граничной» компоновки, в рамках которого задействована центральная область. В данную область помещен контейнер с вертикальной коробочной компоновкой, который содержит все остальные элементы пользовательского интерфейса и имеет границу с заголовком «Расчет значения функции:» (рисунок 23). Верхняя, нижняя, левая и правая области граничной компоновки фрейма не используются и имеют нулевой размер.

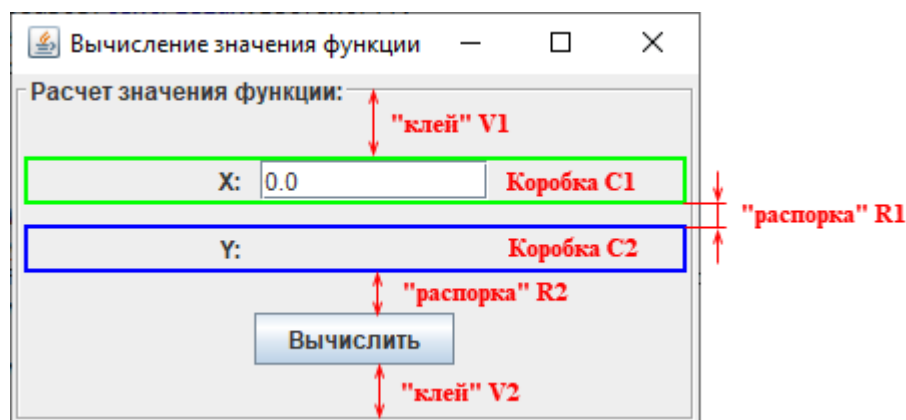


Рис 23. Организация размещения интерфейсных элементов в главном окне приложения

Как видно из представленного рисунка, в рамках контейнера с вертикальной коробочной компоновкой содержатся следующие элементы (сверху вниз):

- элемент *клей V1*, который забирает под себя все пространство по вертикали от верхней границы контейнера с вертикальной коробочной компоновкой до контейнера *коробка C1*;
- контейнер *коробка C1*, который содержит интерфейсные элементы для ввода значения аргумента  $X$  вычисляемой функции;
- элемент *распорка R1*, определяет фиксированное расстояние между контейнером *коробка C1* и контейнером *коробка C2*;
- контейнер *коробка C2*, который содержит интерфейсные элементы для вывода значения вычисляемой функции;
- элемент *распорка R2*, определяет фиксированное расстояние между контейнером *коробка C2* и кнопкой «Вычислить»;
- кнопка «Вычислить», при нажатии которой выполняется расчет значения функции с использованием значения аргумента, заданного в текстовом поле в рамках *коробки C1*, и последующим отображением значения функции в рамках *коробки C2*;
- элемент *клей V2*, который забирает под себя все пространство по вертикали от кнопки до нижней границы контейнера с вертикальной коробочной компоновкой.

При размещении элементов внутри контейнеров типа «горизонтальная коробка» (коробки *C1* и *C2*) применяются элементы типа «клей» и «распорка» (рисунок 24):

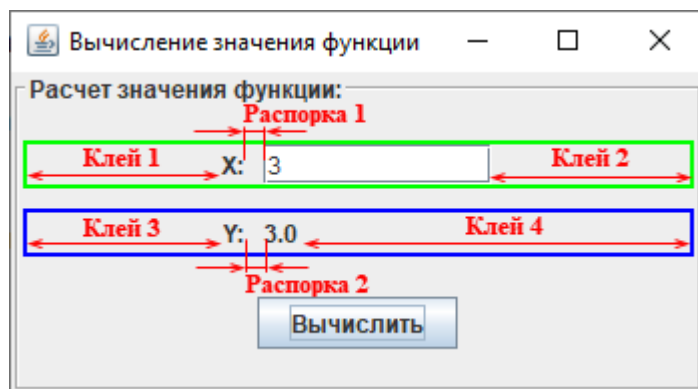


Рис 24. Компоновка элементов внутри горизонтальных контейнеров

## 4.2 Реализация главного окна приложения

Реализация класса *MainFrame* главного окна приложения включает ряд полей, конструктор и метод *main*. Детальное описание назначения этих составляющих класса и их содержимое приведены в последующих подразделах.

### 4.2.1 Поля класса *MainFrame*.

Набор полей класса *MainFrame* представлен набором констант, имеющих модификаторы *static final*, назначение которых описано в таблице 17.

Таблица 17 Поля класса *MainFrame*

Название поля	Порядок использования
<b>Константы</b>	
<i>int WIDTH</i>	Исходный размер окна по горизонтали
<i>int HEIGHT</i>	Исходный размер окна по вертикали

Фрагмент кода, определяющего поля класса *MainFrame* имеет вид:

```
// Константы с исходным размером окна приложения
private static final int WIDTH = 300;
private static final int HEIGHT = 200;
```

### 4.2.2 Реализация конструктора класса *MainFrame*.

Программный код конструктора главного окна приложения обеспечивает компоновку интерфейса пользователя, а также организацию взаимодействия элементов интерфейса с пользователем. Определение конструктора имеет вид:

```
public MainFrame() {
    //Код конструктора
}
```

Код конструктора начинается с вызова конструктора предка *JFrame*, куда передается заголовок окна, а также масштабирования и позиционирования фрейма в рамках экрана:

```
// Обязательный вызов конструктора предка
super("Вычисление значения функции");

// Установить размеры окна
setSize(WIDTH, HEIGHT);

Toolkit kit = Toolkit.getDefaultToolkit();
// Отцентрировать окно приложения на экране
setLocation((kit.getScreenSize().width - WIDTH)/2,
            (kit.getScreenSize().height - HEIGHT)/2);
```

Далее в коде конструктора класса *MainFrame* выполняется непосредственное создание и настройка элементов графического интерфейса.

Сначала создается объект текстового поля, используемый для ввода значений аргумента функции X:

```
// Создать текстовое поле для ввода значения длиной в 10 символов
// со значением по умолчанию 0.0
JTextField textFieldX = new JTextField("0.0", 10);
// Установить максимальный размер равный предпочтительному, чтобы
// предотвратить увеличение размера поля ввода
textFieldX.setMaximumSize(new Dimension(2*textFieldX.getPreferredSize().width,
                                          textFieldX.getPreferredSize().height));
```

Созданное текстовое поле будет изначально содержать значение 0 и вмещать по горизонтали 10 видимых позиций для символов, однако реальный размер поля при отображении в интерфейсе будет зависеть от используемого менеджера компоновки и способа вычисления ширины позиции для символа. Для предотвращения чрезмерного растяжения текстового поля по горизонтали устанавливаются его максимальные размеры путем вызова метода *setMaximumSize*.

После создания и настройки текстового поля для X, создается объект метки *labelY*, предназначенный для вывода значения функции:

```
// Создать метку для вывода значения функции
JLabel labelY = new JLabel("");
labelY.setMinimumSize(textFieldX.getMaximumSize());
labelY.setPreferredSize(textFieldX.getPreferredSize());
```

Далее создается верхняя панель *hboxXValue* (контейнер *коробка C1*, см. рисунок 23), которая располагает размещённые в ней интерфейсные элементы горизонтально друг за другом в порядке их добавления в контейнер. После этого в созданный контейнер добавляются необходимые элементы (смотри рисунок 24).

```
// Создать контейнер типа "коробка с горизонтальной укладкой"
Box hboxXValue = Box.createHorizontalBox();
// Добавить "клей"
hboxXValue.add(Box.createHorizontalGlue());
// Добавить подпись "X:"
hboxXValue.add(new JLabel("X:"));
// Добавить "распорку"
hboxXValue.add(Box.createHorizontalStrut(10));
// Добавить поле ввода начального значения X
hboxXValue.add(textFieldX);
// Добавить "клей"
hboxXValue.add(Box.createHorizontalGlue());
hboxXValue.setMaximumSize(new Dimension(hboxXValue.getMaximumSize().width,
hboxXValue.getPreferredSize().height));
```

В приведенном выше фрагменте программного кода сам контейнер *hboxXValue*, создается с использованием статического метода *createHorizontalBox* класса *Box*, что обеспечивает использование в данном контейнере менеджера коробочной компоновки сразу после его создания. Объект метки с надписью «X:» создается прямо при добавлении данной метки в контейнер. Надпись метки передается как параметр конструктора класса *JLabel*, а отдельная ссылочная переменная для хранения адреса объекта данной метки не создается, т.к. в дальнейшем работать с ее объектом нет необходимости. Для задания фиксированного расстояния в 10 пикселей по горизонтали между меткой и текстовым полем *textFieldX* создается при помощи метода *createHorizontalStrut* и добавляется в контейнер объект распорки.

Следующая часть кода конструктора класса *MainFrame* создает контейнер с горизонтальной укладкой, адрес которого хранится в переменной *hboxYValue* (контейнер *коробка C2*, см. рисунок 23), а также содержащиеся в нем элементы.

```
// Создать контейнер типа "коробка с горизонтальной укладкой"
Box hboxYValue = Box.createHorizontalBox();
// Добавить "клей"
hboxYValue.add(Box.createHorizontalGlue());
// Добавить подпись "Y:"
hboxYValue.add(new JLabel("Y:"));
// Добавить "распорку"
hboxYValue.add(Box.createHorizontalStrut(10));
// Добавить метку для вывода значения Y
hboxYValue.add(labelY);
// Добавить "клей"
hboxYValue.add(Box.createHorizontalGlue());
hboxYValue.setMaximumSize(new Dimension(hboxYValue.getMaximumSize().width,
hboxYValue.getPreferredSize().height));
```

Метки, содержащиеся в данной контейнере, предназначены для отображения надписи «Y:» и вывода значения функции, рассчитанного программой (объект *labelY*). Принципы компоновки объектов внутри контейнера *hboxYValue*

представлены на рисунке 24 и аналогичны описанным ранее принципам компоновки для контейнера *hboxXValue*.

Далее создается и настраивается кнопка «Вычислить».

```
JButton buttonCalc = new JButton("Вычислить");
buttonCalc.setAlignmentX(CENTER_ALIGNMENT);
// Задать действие на нажатие "Вычислить" и привязать к кнопке
buttonCalc.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ev) {
        //Считать значение X
        Double X = Double.parseDouble(textFieldX.getText());
        //Вывести значение X в качестве значения функции
        labelY.setText(Double.toString(X));
    }
});
```

Для создания кнопки используется класс *JButton*, в конструктор которого передается ее название. Вызов метода *setAlignmentX* с параметром *CENTER\_ALIGNMENT* позволяет обеспечить расположение кнопки горизонтально по центру при ее последующем (см. ниже) добавлении в контейнер с вертикальной коробочной компоновкой. Далее путем вызова метода *addActionListener* для кнопки задается объект-слушатель, содержащий код, который будет выполняться в ответ на ее нажатие. Для создания объекта-слушателя используются определяемый “на лету” анонимный (не имеющий явно указанного имени) класс, реализующий интерфейс *ActionListener*, и переопределяющий его метод *ActionPerformed*.

Рассмотрим содержимое метода *ActionPerformed* для кнопки «Вычислить». При щелчке на данную кнопку происходит чтение значения из текстового поля ввода *textFieldX* с его преобразованием из строкового представления в тип чисел с плавающей точкой при помощи метода *parseDouble*. Далее в базовом приложении полученное значение аргумента считается равным значению функции. Поэтому оно преобразуется обратно в строку при помощи метода *toString* класса *Double* и затем отображается в метке *labelY* путем вызова ее метода *setText*.

Для объединения ранее созданных компонент в один набор используется контейнер с вертикальной коробочной компоновкой, код для создания и заполнения которого элементами приведен ниже.

```
Box vboxCalculator = Box.createVerticalBox();
// Задать для контейнера тип рамки с заголовком
vboxCalculator.setBorder(BorderFactory.createTitledBorder(
    BorderFactory.createEtchedBorder(), "Расчет значения функции:"));
vboxCalculator.add(Box.createVerticalGlue());
vboxCalculator.add(hboxXValue);
vboxCalculator.add(Box.createVerticalStrut(10));
vboxCalculator.add(hboxYValue);
vboxCalculator.add(Box.createVerticalStrut(20));
vboxCalculator.add(buttonCalc);
vboxCalculator.add(Box.createVerticalGlue());
```

В данном фрагменте программного кода для создания контейнера с вертикальной коробочной компоновкой, адрес которого хранится в переменной *vboxCalculator*), используется статический метод *createVerticalBox*, класса *Box*. Для данного контейнера путем вызова метода *setBorder* устанавливается рамка с заголовком «Расчет значения функции:». Объект для рамки создается с использованием статического метода *createTitledBorder* класса *BorderFactory*. Кроме самого заголовка, в данный метод передается ссылка на объект класса *EtchedBorder* (создается при помощи метода *createEtchedBorder*), который определяет вид самой рамки по периметру контейнера. Структура расположения компонентов внутри контейнера *vboxCalculator* представлена на рисунке 23. Компоненты типа клей, создаваемый при помощи метода *createVerticalGlue*, и добавляемые первым и последним в контейнер *vboxCalculator*, забирают под себя все свободное от других компонентов пространство по вертикали при увеличении размера контейнера. Это дает возможность центрировать остальные компоненты в рамках *vboxCalculator*. Кроме компонент типа клей в данный контейнер добавлены ранее созданные контейнеры с горизонтальной компоновкой *hboxXValue* и *hboxYValue*, а также кнопка «Вычислить» (объект *buttonCalc*). Для задания фиксированных расстояний в пикселях между данными компонентами по вертикали используются компоненты типа распорка, которые создаются с использованием метода *createVerticalStrut* класса *Box* и добавляются в контейнер *vboxCalculator* между теми компонентами, между которыми необходимо задать фиксированное расстояние.

Завершается код конструктора главного окна приложения установкой контейнера *vboxCalculator* в центральную область панели содержимого главного окна:

```
// Установить коробку в панель содержимого  
getContentPane().add(vboxCalculator);
```

Это позволяет отобразить все содержимое контейнера *vboxCalculator* в главном окне приложения после его запуска.

#### **4.2.3 Реализация метода *main*.**

Метод *main*, реализованный в классе *MainFrame*, является точкой входа в приложение и вызывается первым при его запуске. Реализация данного метода имеет вид:



```
public static void main(String[] args) {  
    // Создать экземпляр главного окна  
    MainFrame frame = new MainFrame();  
    // Задать действие, выполняемое при закрытии окна  
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    //Показать главное окно приложения  
    frame.setVisible(true);  
}
```

Программный код метода *main* создает объект главного окна приложения, устанавливает режим выхода из приложения при закрытии главного окна (вызов метода *setDefaultCloseOperation*) и отображает главное окно на экране (вызов метода *setVisible*).



## 5 Задания

### 5.1 Вариант сложности А

- а) добавить в главное окно приложения панель «Параметры функции:», расположенную над панелью «Расчет значения функции:» (рисунок 25). При изменении размеров главного окна приложения панель «Параметры функции:» должна занимать весь размер главного окна по горизонтали и оставаться неизменной по вертикали;
- б) в панель «Параметры функции:» добавить метку «Р:» и справа от нее текстовое поле (рисунок 25) для ввода значения параметра  $P$ , который должен использоваться для вычисления значений функции  $Y$ , по формуле, указанной в варианте задания (таблица 18);
- в) справа от текстового поля для ввода значения параметра  $P$ , добавить две кнопки с названиями «-» и «+» (рисунок 25), при нажатии которых значение в текстовом поле для параметра  $P$ , должно соответственно уменьшаться/увеличиваться на 1;
- г) в панель «Параметры функции:» (рисунок 25), ниже расположения элементов для работы с параметром  $P$ , добавить метку, содержащую надпись «Формула:  $f(x) =$  », после которой должно следовать выражение, приведенное в варианте задания (таблица 18). При изменении размеров главного окна приложения метка должна располагаться по горизонтали по центру панели «Параметры функции:»;
- д) обеспечить при нажатии на кнопку «Вычислить» вычисление значения функции в соответствии с выражением, приведенным в варианте задания (таблица 18) и отображение полученного значения справа от метки «Y:». Выполнить обработку ошибок, которые могут возникать из-за некорректно введенных значений параметра  $P$  и аргумента функции  $X$ .

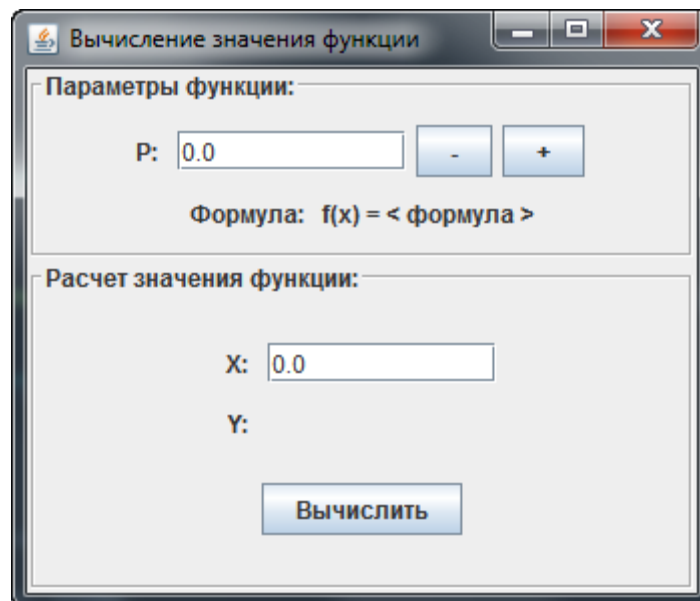


Рис 25. Шаблон внешнего вида окна приложения, после выполнения задания сложности А

Таблица 18 Варианты заданий для уровня сложности А

№ п/п	Функция для вычисления Y
1	$Y = X + P$
2	$Y = X - P$
3	$Y = P - X$
4	$Y = X * P$
5	$Y = X / P$
6	$Y = P / X$
7	$Y = (P - 1) * X$
8	$Y = P - 2X$
9	$Y = 2X + P$
10	$Y = X - 2P$
11	$Y = (X + 1) / P$
12	$Y = P * X - 1$

## 5.2 Вариант сложности В

- а) выполнить задание а) соответствующего варианта уровня сложности А;
- б) выполнить задание б) соответствующего варианта уровня сложности А;
- в) выполнить задание в) соответствующего варианта уровня сложности А;
- г) выполнить задание г) соответствующего варианта уровня сложности А;
- д) выполнить задание д) соответствующего варианта уровня сложности А;
- е) реализовать наряду с функцией из задания сложности А дополнительную функцию для вычисления значения  $Y$  в соответствии с вариантом задания (таблица 19);
- ж) добавить в панель «Параметры функции:» дополнительные элементы пользовательского интерфейса в соответствии с вариантом задания (таблица 19), которые:
  - располагаются по вертикали ниже элементов для работы с параметром  $P$ , но выше метки, показывающей формулу для функции;
  - располагаются по горизонтали друг за другом на некотором расстоянии в порядке их перечисления в задании, при этом весь их набор отцентрирован по горизонтали относительно размеров панели «Параметры функции» (смотри рисунок 25);
  - позволяют выбирать вычисляемую приложением функцию (базовую из варианта сложности А) или дополнительную из таблицы 19);
  - сразу после действий пользователя с ними изменяют выражение для вычисляемой функции, которое отображается в метке «Формула:  $f(x) =$  » (смотри рисунок 25);
  - сразу после запуска приложения имеют состояние, соответствующее вычислению базовой функции из варианта сложности А).

Таблица 19 (для уровня сложности В)

№ п/п	Дополнительная функция для вычисления Y	Дополнительные элементы панели «Параметры функции:»
1	$Y = X - P$	метка «Роль Р:» и не редактируемый комбинированный список с двумя значениями «Слагаемое» и «Вычитаемое»
2	$Y = P - X$	две радиокнопки: «Р - вычитаемое» и «Р - уменьшаемое»
3	$Y = P + X$	метка «Операция с X:» и не редактируемый комбинированный список с двумя значениями «Вычесть» и «Прибавить»
4	$Y = X/P$	флажок с надписью: «Обратное значение для Р»
5	$Y = P/X$	две радиокнопки: «Р - делимое» и «Р - делитель»
6	$Y = 1/X$	флажок с надписью: «Использовать Р»
7	$Y = (P + 1) * X$	метка «Изменение Р:» и не редактируемый комбинированный список с двумя значениями «Уменьшить на 1» и «Увеличить на 1»
8	$Y = P + 2X$	две радиокнопки: «Вычесть 2X» и «Прибавить 2X»
9	$Y = X + P$	флажок с надписью: «Удвоить X»
10	$Y = 2X - P$	метка «Удвоить:» и не редактируемый комбинированный список с двумя значениями «Аргумент X» и «Параметр Р»
11	$Y = (X + 1) * P$	две радиокнопки: «Р - делитель» и «Р - множитель»
12	$Y = P/X - 1$	флажок с надписью: «Обратное значение для X»

### 5.3 Вариант сложности С

- а) выполнить задание а) соответствующего варианта уровня сложности А;
- б) выполнить задание б) соответствующего варианта уровня сложности А;
- в) выполнить задание в) соответствующего варианта уровня сложности А;
- г) в панель «Параметры функции:»), ниже расположения элементов для работы с параметром  $P$ , добавить интерфейсный элемент, показывающий изображение вычисляемой функции, загруженное из файла с расширением .png (файлы изображений подготовить заранее на основе формул, набранных в редакторе формул MS Word). Изображение должно располагаться по горизонтали по центру панели «Параметры функции:»;
- д) убрать из интерфейса приложения кнопку «Вычислить» и обеспечить вычисление значения функции и отображение полученного значения справа от метки «Y:» автоматически сразу после внесения пользователем изменений в интерфейсных элементах приложения. При работе с текстовыми полями обеспечить завершение их редактирования (если введено корректное значение) при нажатии кнопки «Enter» на клавиатуре. Выполнить обработку ошибок, которые могут возникать из-за некорректно введенных значений параметра  $P$  и аргумента функции  $X$ ;
- е) выполнить задание е) соответствующего варианта уровня сложности В;
- ж) выполнить задание ж) соответствующего варианта уровня сложности В;

## Приложение 1. Исходный код базового приложения

### Главный класс приложения *MainFrame*

```
package mainPackage;

import java.awt.Dimension;
import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.BorderFactory;
import javax.swing.Box;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;

@SuppressWarnings("serial")
public class MainFrame extends JFrame {
    // Константы с исходным размером окна приложения
    private static final int WIDTH = 300;
    private static final int HEIGHT = 200;

    public MainFrame() {
        // Обязательный вызов конструктора предка
        super("Вычисление значения функции");

        // Установить размеры окна
        setSize(WIDTH, HEIGHT);

        Toolkit kit = Toolkit.getDefaultToolkit();
        // Отцентрировать окно приложения на экране
        setLocation((kit.getScreenSize().width - WIDTH)/2,
                    (kit.getScreenSize().height - HEIGHT)/2);

        // Создать текстовое поле для ввода значения длиной в 10 символов
        // со значением по умолчанию 0.0
        JTextField textFieldX = new JTextField("0.0", 5);
        // Установить максимальный размер равный предпочтительному, чтобы
        // предотвратить увеличение размера поля ввода
        textFieldX.setMaximumSize(
            new Dimension(2*textFieldX.getPreferredSize().width,
                          textFieldX.getPreferredSize().height));

        // Создать метку для вывода значения функции
        JLabel labelY = new JLabel("");
        labelY.setMinimumSize(textFieldX.getMaximumSize());
        labelY.setPreferredSize(textFieldX.getPreferredSize());

        // Создать контейнер типа "коробка с горизонтальной укладкой"
        Box hboxXValue = Box.createHorizontalBox();
        // Добавить "клей"
        hboxXValue.add(Box.createHorizontalGlue());
        // Добавить подпись "X:"
        hboxXValue.add(new JLabel("X:"));
        // Добавить "распорку"
        hboxXValue.add(Box.createHorizontalStrut(10));
        // Добавить поле ввода начального значения X
        hboxXValue.add(textFieldX);
    }
}
```

```

// Добавить "клей"
hboxXValue.add(Box.createHorizontalGlue());
hboxXValue.setMaximumSize(
    new Dimension(hboxXValue.getMaximumSize().width,
        hboxXValue.getPreferredSize().height));

// Создать контейнер типа "коробка с горизонтальной укладкой"
Box hboxYValue = Box.createHorizontalBox();
// Добавить "клей"
hboxYValue.add(Box.createHorizontalGlue());
// Добавить подпись "Y:"
hboxYValue.add(new JLabel("Y:"));
// Добавить "распорку"
hboxYValue.add(Box.createHorizontalStrut(10));
// Добавить метку для вывода значения Y
hboxYValue.add(labelY);
// Добавить "клей"
hboxYValue.add(Box.createHorizontalGlue());
hboxYValue.setMaximumSize(
    new Dimension(hboxYValue.getMaximumSize().width,
        hboxYValue.getPreferredSize().height));

JButton buttonCalc = new JButton("Вычислить");
buttonCalc.setAlignmentX(CENTER_ALIGNMENT);
// Задать действие на нажатие "Вычислить" и привязать к кнопке
buttonCalc.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ev) {
        //Считать значение X
        Double X = Double.parseDouble(textFieldX.getText());
        //Вывести значение X в качестве значения функции
        labelY.setText(Double.toString(X));
    }
});

Box vboxCalculator = Box.createVerticalBox();
// Задать для контейнера тип рамки с заголовком
vboxCalculator.setBorder(BorderFactory.createTitledBorder(
    BorderFactory.createEtchedBorder(),
    "Расчет значения функции:"));
vboxCalculator.add(Box.createVerticalGlue());
vboxCalculator.add(hboxXValue);
vboxCalculator.add(Box.createVerticalStrut(10));
vboxCalculator.add(hboxYValue);
vboxCalculator.add(Box.createVerticalStrut(20));
vboxCalculator.add(buttonCalc);
vboxCalculator.add(Box.createVerticalGlue());

// Установить коробку в панель содержимого
getContentPane().add(vboxCalculator);
}

public static void main(String[] args) {
    // Создать экземпляр главного окна
    MainFrame frame = new MainFrame();
    // Задать действие, выполняемое при закрытии окна
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    //Показать главное окно приложения
    frame.setVisible(true);
}
}

```