Лабораторная работа 6. Построение графических интерфейсов с помощью инструментов Swing

# 1. Основные теоретические сведения

## 1.1. Библиотека Swing

### 1.1.1. Структура компонент Swing

Компоненты Swing можно разделить на следующие типы:

* контейнеры верхнего уровня (классы **JWindow**, **JFrame**, **JDialog** и **JApplet**);
* специализированные контейнеры (классы **JInternalFrame**, **JLayeredPane**, **JRootPane** и **JOptionPane**);
* общецелевые контейнеры (классы **JPanel**, **JScrollPane**, **JSplitPane**, **JTabbedPane** и **JToolBar**);
* компоненты управления (классы **JButton**, **JCheckBox**, **JRadioButton**, **JToggleButton**, **JComboBox**, **JList**, **JMenuBar**, **JMenu**, **JMenuItem**, **JCheckboxMenuItem**, **JRadioButtonMenuItem**, **JSeparator** и **JSlider**);
* нередактируемые информационные компоненты (классы **JLabel**, **JProgressBar** и **JToolTip**);
* редактируемые информационные компоненты (классы **JColorChooser**, **JFileChooser**, **JTable**, **JTree**, **JTextField**, **JPasswordField**, **JTextArea**, **JEditorPane** и **JTextPane**).

### 1.1.2. Контейнеры верхнего уровня и специализированные контейнеры

Так же, как и для AWT, для создания окон графических приложений используется не класс **JWindow**, а класс **JFrame** (окна, создаваемые классом **JWindow** не содержат наименования окна и кнопок управления окном). Приложения с графическим интерфейсом использует, по крайней мере, один фрейм. Апплеты также могут использовать фреймы.

Для создания окон, которые зависят от другого окна (например, исчезают, когда сворачивается окно, в котором они используются) применяются диалоговые окна класса **JDialog**.

Апплеты, использующие компоненты Swing, должны быть подклассами класса **JApplet**.

Любая программа, которая использует компоненты Swing, содержит, по крайней мере, один контейнер верхнего уровня. Этот контейнер является корнем иерархии контейнеров, содержащих все компоненты Swing.

Как правило, отдельное графическое приложение имеет, по крайней мере, одну иерархию контейнеров, в которой корнем является **JFrame**. Диалоговое окно или апплет также образуют иерархию контейнеров, корнем которой является **JDialog** или **JApplet**. Например, если приложение содержит одно главное окно и два диалоговых окна, то она содержит три иерархии контейнеров.

### 1.1.3. Корневая панель и панель содержимого

Каждый контейнер верхнего уровня базируется на промежуточном, скрытом, контейнере, называемом корневой панелью (root pane). Корневая панель определена в классе **JRootPane**.

Сама корневая панель обычно не используется, а используются ее компоненты, которые корневая панель (класс) предоставляет фрейму (или другому контейнеру верхнего уровня). Корневая панель содержит следующие компоненты:

* слоистая панель (layered pane);
* панель содержимого (content pane);
* строка меню (menu bar) – необязательный компонент;
* стеклянная панель (glass bar).

Единственным обязательным контейнером верхнего уровня является панель содержимого.

Панель содержимого содержит все компоненты Swing (кнопки, надписи, текстовые поля и т.д.). Поскольку для контейнеров верхнего уровня содержимое окна определяется с помощью **JRootPanel** и должно, в отличие от окон AWT, определяться вручную, для добавления компонент или установки менеджера компоновки используются не методы **add**() и **setLayout**(), а методы получения и установки панели содержимого:

**public Container getContentPane**() и L

**public void setContentPane(Container** contentPane**)**,

которые определены в классах **JFrame**, **JDialog**, **JApplet** и **JInternalFrame**.

Так, добавление текстового поля во фрейм, диалоговое окно, апплет или внутренний фрейм с установкой менеджера компоновки **FlowLayout** выглядит следующим образом:

**Container** contentPane = **getContentPane**();

**JTextField** inputField = **new JTextField**(15);

contentPane.**setLayout**(**new FlowLayout**());

contentPane.**add**(inputField);

Следует отметить, что для всех контейнеров верхнего уровня (включая **JApplet**) менеджером компоновки по умолчанию является **BorderLayout**.

К контейнеру верхнего уровня может быть добавлена строка меню (menu bar). Строка меню также располагается внутри главного контейнера, но за пределами панели содержимого.

### 1.1.4. Класс JFrame

Фрейм в Swing создается с помощью конструкторов **JFrame**() или **JFrame** (**String** *name*), где *name* – имя фрейма.

Для фрейма можно использовать приведенные выше методы получения и установки корневой панели, слоистой панели, панели содержимого, строки меню и стеклянной панели, причем должно быть получено, по крайней мере, значение панели содержимого.

Другим отличием **JFrame** от **Frame** является наличие у **JFrame** свойства, определяющего операцию закрытия окна по умолчанию. Для **Frame** по умолчанию ничего не происходит, если делается попытка закрыть окно, а окно **JFrame** прячется.

Метод **public void setDefaultCloseOperation**(**int** *operation*) класса **JFrame** позволяет определить три операции, которые могут выполняться при закрытии окна **JFrame**:

1. **DO\_NOTHING\_ON\_CLOSE** – ведет себя так же, как AWT Frame;
2. **HIDE\_ON\_CLOSE** – поведение по умолчанию (для того, чтобы окно снова появилось на экране, необходимо использовать метод **setVisible(true)**;
3. **DISPOSE\_ON\_CLOSE** – окно закрывается.

### 1.1.5. Диалоговые окна

Диалоговые окна можно создавать либо с помощью класса **JDialog**, либо используя класс **JOptionPane**.

Основными конструкторами класса **JDialog** являются:

* **JDialog**();
* **JDialog**(**Dialog** *owner*);
* **JDialog**(**Dialog** *owner*, **boolean** *modal*);
* **JDialog**(**Dialog** *owner*, **String** *title*);
* **JDialog**(**Dialog** *owner*, **String** *title*, **boolean** *modal*);
* **JDialog**(**Frame** *owner*);
* **JDialog**(**Frame** *owner*, **boolean** *modal*);
* **JDialog**(**Frame** *owner*, **String** *title*);
* **JDialog**(**Frame** *owner*, **String** *title*, **boolean** *modal*).

Первый конструктор создает немодальное диалоговое окно без имени. Параметры *owner*, *title* и *modal* в остальных конструкторах задают соответственно родителя данного диалогового окна, имя окна и окна (**true** – модальное окно или **false** – немодальное окно).

Окно класса **JDialog** имеет те же черты, что **JFrame** и прямое использование объектов класса **JDialog** аналогично прямому использованию объекта **JFrame**.

Класс **JOptionPane** облегчает вывод стандартных диалоговых или информационных окон.

Для класса **JOptionPane** определены следующие основные конструкторы:

1. **JOptionPane**();
2. **JOptionPane**(**Object** *message*, **int** *messageType*, **int** *optionType*, **Icon** *icon*, **Object**[] *options*, **Object** *initialValue*);

Кроме этого определены конструкторы, содержащие соответственно только первый параметр, только первые два параметра, только первые три параметра, только первые четыре параметра, только первые пять параметров второго конструктора.

Все использования этого класса обычно сводятся к вызову одного из его методов:

* **static int showConfirmDialog**(**Component** *parentComponent*, **Object** *message*, **String** *title*, **int** *optionType*, **int** *messageType*, **Icon** *icon*) – задает вопрос на который требуется ответ типа **Yes**, **No** или **Cancel**. Существуют также следующие три метода **showConfirmDialog**(): метод, содержащий первые два параметра, метод, содержащий первые четыре параметра и метод, содержащий первые пять параметров, приведенного выше метода;
* **static Object showInputDialog**(**Component** *parentComponent*, **Object** *message*, **String** *title*, **int** *messageType*, **Icon** *icon*, **Object**[] *selectionValues*, **Object** *initialValue*) – позволяет ввести некоторый текст. Существуют также следующие пять методов **showInputDialog**(): метод, содержащий второй параметр, метод, содержащий второй и седьмой параметры, метод, содержащий первые два параметра, метод, содержащий первый, второй и седьмой параметры, и метод, содержащий первые три параметра и седьмой параметр;
* **static void showMessageDialog**(**Component** *parentComponent*, **Object** *message*, **String** *title*, **int** *messageType*, **Icon** *icon*) – позволяет вывести некоторое сообщение. Существуют также следующие два метода **showMessageDialog**(): метод, содержащий первые два параметра и метод, содержащий первые четыре параметра;
* **static int showOptionDialog**(**Component** *parentComponent*, **Object** *message*, **String** *title*, **int** *optionType*, **int** *messageType*, **Icon** *icon*, **Object**[] *options*, **Object** *initialValue*) – объединяет функции перечисленных выше диалоговых окон.

Параметры в приведенных конструкторах и методах имеют следующий смысл:

* *parentComponent* – определяет компонент, являющийся родителем данного диалогового окна (если этот параметр равен null, в качестве родителя предполагается фрейм по умолчанию);
* *message* – описательное сообщение, помещаемое в диалоговое окно (следует иметь в виду, что это объект класса Object);
* *selectionValues* – массив объектов, рассматриваемый как набор сообщений;
* *icon* – изображение, выводимое в диалоге (значение по умолчанию определяется значением параметра *messageType*);
* *messageType* – определяет тип сообщения. Допустимы следующие значения: **ERROR\_MESSAGE**, **INFORMATION\_MESSAGE**, **WARNING\_MESSAGE**, **QUESTION\_MESSAGE** и **PLAIN\_MESSAGE**;
* *optionType* – определяет набор кнопок, которые будут выведены в нижней части диалогового окна: **DEFAULT\_OPTION**, **YES\_NO\_OPTION**, **YES\_NO\_CANCEL\_OPTION** и **OK\_CANCEL\_OPTION**;
* *options* – более детальное описание набора кнопок, которые будут выведены в нижней части диалогового окна (если задан объект класса **Component**, он прямо добавляется в нижнюю часть диалогового окна, если объект типа **Icon**, создается кнопка класса **JButton**, для объектов других классов производится преобразование объекта в строку, которая выводится на кнопке класса **JButton**);
* *title* – имя диалогового окна;
* *initialValue* – вводимое значение по умолчанию.

Для методов и допустимы следующие возможные значения возвращаемого значения: **YES\_OPTION**, **NO\_OPTION**, **CANCEL\_OPTION**, **OK\_OPTION** и **CLOSED\_OPTION**.

### 1.1.6. Общецелевые контейнеры

#### Класс JPanel

Класс **JPanel** является вариантом класса **Panel** в AWT. Четыре конструктора класса:

* **JPanel**();
* **JPanel**(**LayoutManager** *layout*);
* **JPanel**(**boolean** *isDoubleBuffered*);
* **JPanel**(**LayoutManager** *layout*, **boolean** *isDoubleBuffered*).

обеспечивают создание объекта **JPanel** соответственно под управлением менеджера компоновки **FlowLayout**, под управлением заданного менеджера компоновки, с встроенной поддержкой двойной буферизации и, наконец, под управлением заданного менеджера компоновки и с встроенной поддержкой двойной буферизации.

Работа с объектами класса **JPanel** не отличается от работы с объектами класса **Panel**, за исключением того, что панель добавляются не во фрейм, а в панель содержимого фрейма.

#### Класс JTabbedPane

Панель с вкладками (tabbed pane) введена в Swing вместо использовавшегося в AWT менеджера **CardLayout**, т.е. менеджер компоновки **CardLayout** в Swing не реализован. Каждая вкладка имеет заголовок. Когда пользователь выбирает вкладку, ее содержимое становится видимым. Только одна из вкладок может быть выбрана одновременно. Панель с вкладками обычно используется для установки параметров конфигурации.

Панель с вкладками реализована классом **JTabbedPane**, который имеет следующие конструкторы:

* **JTabbedPane**();
* **JTabbedPane**(**int** *tabPlacement*);
* **JTabbedPane**(**int** *tabPlacement*, **int** *tabLayoutPolicy*).

Первый конструктор создает новую панель с вкладками, в которой заголовки вкладок расположены сверху, а второй позволяет определить расположение заголовков вкладок (**JTabbedPane**.**TOP** – сверху, **JTabbedPane**.**BOTTOM** – снизу, **JTabbedPane**.**RIGHT** – вправо или **JTabbedPane**.**LEFT** – влево). Третий конструктор дополнительно задает правило расположения вкладок (**JTabbedPane.WRAP\_TAB\_LAYOUT** – вкладки располагаются в несколько рядов, если они не помещаются в одном ряду – значение по умолчанию и **JTabbedPane.SCROLL\_TAB\_LAYOUT** – вкладки прокручиваются, если они не помещаются в одном ряду).

Формирование в программе панели с вкладками выполняется с помощью следующей процедуры:

1. Создать объект **JTabbedPane**.
2. Для добавления каждой вкладки в панель вызвать один из методов **addTab**(): **public void addTab**(**String** *title*, **Icon** *icon*, **Component** *component*, **String** *tip*), **public void addTab**(**String** *title*, **Icon** *icon*, **Component** *component*), **public void addTab**(**String** *title*, **Component** *component*). Аргументы этих методов определяют заголовок вкладки *title*, компоненту *component*, который она содержит, изображение *icon* и всплывающую подсказку *tip*.
3. Добавить панель с вкладками в панель содержания окна.

Для отслеживания выбора той или иной вкладки используется блок прослушивания **ChangeListener**, который добавляется или удаляется с помощью методов **public void addChangeListener**(**ChangeListener** *l*) и **public void removeChangeListener**(**ChangeListener** *l*) класса **JTabbedPane**. В интерфейсе **ChangeListener** определен единственный метод **public void stateChanged**(**ChangeEvent** e), в котором с помощью методов **public Component getSelectedComponent**() или **public int getSelectedIndex**() класса **JTabbedPane** можно получить выбранный компонент или индекс выбранного компонента.

#### Класс JSplitPane

Расщепленные панели (класс **JSplitPane**) используются для разделения выводимой панели на две компоненты. Конструкторы класса:

* **JSplitPane**();
* **JSplitPane**(**int** *newOrientation*);
* **JSplitPane**(**int** *newOrientation*, **boolean** *newContinuousLayout*);
* **JSplitPane**(**int** *newOrientation*, **Component** *newLeftComponent*, **Component** *newRightComponent*);
* **JSplitPane**(**int** *newOrientation*, **boolean** *newContinuousLayout*, **Component** *newLeftComponent*, **Component** *newRightComponent*)

задают в параметре *newOrientation* выравнивание слева направо с использованием константы **JSplitPane.HORIZONTAL\_SPLIT** или сверху вниз с использованием константы **JSplitPane.VERTICAL\_SPLIT**, в параметре *newContinuousLayout* непрерывную перерисовку компонент при перемещении разделителя (значение true) или перерисовку компонент после окончания перемещения разделителя (значение false), а в параметрах *newLeftComponent* и *newRightComponent* – компоненты расщепленной панели.

Положение разделителя в панели задается с помощью метода **public void setDividerLocation**(**int** *location*), который задает в пикселях позицию x или y разделителя (в зависимости от вертикальной или горизонтальной ориентации панелей). Метод **public void setOneTouchExpandable**(**boolean** *newValue*) позволяет задать на разделительной полосе кнопки (со стрелками) для быстрого сворачивания и разворачивания одной из панелей расщепленной панели.

### 1.1.7. Классы компонентов

#### Интерфейс Icon и класс ImageIcon

Значки (icons) реализуются в Swing с помощью интерфейса **Icon** (на самом деле они не являются компонентами и их можно использовать практически со всеми компонентами Swing). Реализацией интерфейса **Icon**, создающей значок из изображения, является класс **ImageIcon**. Двумя основными конструкторами этого класса являются:

* **ImageIcon**(**String** *filename*);
* **ImageIcon**(**URL** *url*).

Первая форма использует изображение в файле с именем *filename* (используется в графических приложениях) а вторая форма – в ресурсе, расположенном по URL-адресу *url* (используется в апплетах Swing)

В отличие от класса **Image**, в классе **ImageIcon** изображение загружается предварительно с использованием методов класса **MediaTracker**.

Для создания собственных изображений можно непосредственно реализовывать интерфейс **Icon**.

#### Класс JLabel

Надписи Swing являются объектами класса **JLabel**. Этот объект может отображать тексты и/или значки. Основными его конструкторами являются:

* **JLabel**(**Icon** *image*);
* **JLabel**(**String** *text*);
* **JLabel** (**String** *text*, **Icon** *image*, **int** *horizontalAlignment*).

В этих конструкторах *text* и *image* – текст и значок, используемый в надписи. Параметр *horizontalAlignment* определяет выравнивание и имеет значения ***LEFT***, ***RIGHT*** или ***CENTER***. Эти константы определены в интерфейсе **SwingConstants**, наряду с несколькими другими, используемыми классами системы Swing.

Значок и текст, связанный с надписью, можно считывать и записывать следующими методами:

* **public Icon getIcon**();
* **public String getText**();
* **public void setIcon**(**Icon** *image*);
* **public void setText**(**String** *text*);

Для обработки событий в объектах класса **JLabel** можно использовать интерфейс **MouseListener** и класс **MouseEvent**.

#### Класс AbstractButton

Кнопки Swing являются подклассами класса **AbstractButton**. Этот класс содержит много методов, которые позволяют управлять поведением кнопок, флажков и переключателей. Например, можно определять различные значки для отображения компонента, когда он отжат (disabled), нажат (pressed), или выбран (selected). Некоторый значок можно использовать как значок "наезда" (rollover), который отображается, когда курсор мыши установлен поверх этого компонента ("наехал" на него). Ниже приведены описания методов, которые управляют этим поведением:

* **public void setDisabledIcon**(**Icon** *image*);
* **public void setPressedIcon** (**Icon** *image*);
* **public void setSelectedIcon**(**Icon** *image*);
* **public void setRolloverIcon**(**Icon** *image*).

Текст, связанный с кнопкой, можно читать и записывать с помощью следующих методов:

* **public String getText**();
* **public void setText**(**String** *text*).

В классе **AbstractButton** определены также методы **public int getMnemonic**() и **public void setMnemonic**(**int** *mnemonic*) соответственно для получения и установки «горячих» клавиш. В качестве параметра при установке «горячей» клавиши задается оно из статических свойств класса **KeyEvent**. Эти свойства имеют общее имя **VK\_XXX**, где XXX – имя одной из клавиш, например, VK\_D.

При нажатии кнопки конкретные подклассы **AbstractButton** генерируют события **Action**. Блоки прослушивания регистрируют и отменяют регистрацию для этих событий с помощью следующих методов:

* **void addActionListener**(**ActionListener** *l*);
* **void removeActionListener**(**ActionListener** *l*).

#### Класс JButton

Класс JButton является подклассом класса **AbstractButton** и обеспечивает функциональные возможности кнопки. Этот класс позволяет связать с кнопкой изображение, текст или и то и другое. Некоторые из его конструкторов:

* **JButton**(**Icon** *image*);
* **JButton**(**String** *text*);
* **JButton**(**String** *text*, **Icon** *image*).

Здесь *text* и *image* – надпись и изображение, используемые для кнопки.

Класс **JButton** наследует свойства интерфейса **SwingConstants**, а также свойства и методы класса **AbstractButton**.

Для обработки событий в объектах класса **JButton** можно использовать интерфейс **ActionListener** и класс **ActionEvent**.

#### Флажки и переключатели

Класс **JCheckBox**, который обеспечивает функциональные возможности флажка, является конкретной реализацией класса **AbstractButton**. Некоторые из его конструкторов:

* **JCheckBox**(**Icon** *image*);
* **JCheckBox**(**Icon** *image*, **boolean** *state*);
* **JCheckBox**(**String** *text*);
* **JCheckBox**(**String** *text*, **boolean** *state*);
* **JCheckBox** (**String** *text*, **Icon** *image*);
* **JCheckBox**(**String** *text*, **Icon** *image*, **boolean** *state*).

В этих конструкторах используются следующие параметры: *image* — изображение для флажка, *text* – надпись. Если *state* имеет значение ***true***, то флажок первоначально выбран.

Состояние флажка может быть изменено с помощью метода **void public void setSelected**(**boolean** *b*), а получить состояние флажка можно с помощью метода **public boolean isSelected**().

Переключатели поддерживаются классом **JRadioButton**, который также является конкретной реализацией класса **AbstractButton**. Ниже приведены некоторые из его конструкторов:

* **JRadioButton**(**Icon** *image*);
* **JRadioButton**(**Icon** *image*, **boolean** *state*);
* **JRadioButton**(**String** *text*);
* **JRadioButton**(**String** *text*, **boolean** *state*);
* **JRadioButton**(**String** *text*, **Icon** *image*);
* **JRadioButton**(**String** *text*, **Icon** *image*, **boolean** *state*).

где параметры имеют тот же смысл, что и для класса **JCheckBox**.

Переключатели должны быть объединены в группу, где в каждый момент может быть выбран только один элемент. Например, если пользователь щелкает по переключателю, который находится в группе, любой предварительно нажатый переключатель в этой группе автоматически сбрасывается. Чтобы создать группу кнопок, создается объект класса **ButtonGroup**. Для этой цели используется единственный конструктор класса **public ButtonGroup**().

Элементы к группе переключателей добавляются с помощью метода **public void add**(**AbstractButton** *a*), а количество переключателей в группе можно получить с помощью метода **public int getButtonCount**().

Родительским классом и для **JCheckBox** и для **JRadioButton** является класс **JToggleButton**, который не имеет эквивалента в AWT. Он ведет себя как объект класса **Button**, который после нажатия на него остается в нажатом состоянии.

Для обработки событий в объектах классов **JCheckBox** и **JRadioButton** можно использовать интерфейс **ItemListener** и класс **ItemEvent**.

#### Класс JComboBox

Вместо класса **Choice** в AWT, в системе Swing обеспечивает комбинированное поле (combo box) – комбинация текстового поля и раскрывающегося списка (класс **JComboBox**). Комбинированное поле обычно отображает один вход (элемент) списка. Однако оно может также отображать и раскрывающийся список, который дает возможность пользователю выбирать различные элементы. Вы также можете ввести с клавиатуры свое значение элемента списка в текстовое поле. Два самых распространенных конструктора:

* **JComboBox**();
* **JComboBox**(**Object** [] *items*)

позволяют создать пустой список с выбором или задать массив объектов данного списка.

Элементы добавляются к списку выбора с помощью метода **void addItem**(**Object** *obj*), где *obj* – объект, который будет добавлен к комбинированному полю.

Остальные методы класс аналогичны соответствующим методам класса **Choice**, однако добавлены методы **public void setEditable**(**boolean** *aFlag*) и **public boolean isEditable**(), позволяющие установить возможность ввода значения в список и проверить флажок возможности ввода своего значения в список.

Для обработки событий, связанных с комбинированным списком, можно использовать либо блоки прослушивания **ItemListener**, либо блоки прослушивания **ActionListener**.

#### Класс JList

Аналогом класса **List** в Swing является класс **JList**. Два самых распространенных конструктора этого класса

* **JList**();
* **JList**(**Object** [] *items*)

позволяют задать либо пустой список, либо список, содержащий заданные объекты (обратите внимания, что элементами списков в Swing могут быть не только строки, но произвольные объекты) не выполняется в **JList** автоматически, а выполняется с помощью помещения списка в объект класса **JScrollPane**.

Набор методов в классе **JList** существенно расширен по сравнению с классом **List** (так, например, есть возможность задавать цвет текста и фона для выделенных элементов) и, кроме того, вместо интерфейса **ItemListener** для списков **JList** используется интерфейс **ListSelectionListener**, блок прослушивания которого добавляется и удаляется с помощью методов **public void addListSelectionListener**(**ListSelectionListener** *listener*) и **public void removeListSelectionListener**(**ListSelectionListener** *listener*) класса **JList**.

Метод **public void valueChanged**(**ListSelectionEvent** *e*) интерфейса обрабатывает события, связанные с выбором элементов списка, а метод **public Object**[] **getSelectedValues**() класса **JList** позволяет получить список выбранных объектов.

#### Класс JSpinner

Класс **JSpinner** определяет «вращающийся» список. Этот класс похож на класс **JComboBox**, но однострочный список можно прокручивать в двух направлениях: вверх и вниз с помощью двух кнопок справа от поля выбора.

Для класса **JSpinner** определены два конструктора:

* **JSpinner**();
* **JSpinner**(**SpinnerModel** *model*).

Второй конструктор создает объект для заданной модели «вращающего» списка. В пакете определены классы для наиболее употребительных моделей «вращающего» списка: **SpinnerListModel** (для списков), **SpinnerNumberModel** (для чисел) и **SpinnerDateModel** (для дат).

Установить или получить модель можно либо с помощью методов **void setModel**(**SpinnerModel** *model*) и **SpinnerModel getModel**().

Изменить выводимое в поле значение можно с помощью метода **void setValue**(**Object** *value*), а получить текущее значение поля можно с помощью метода **Object getValue**() класса **JSpinner**.

Установить конкретную модель для списков можно с помощью следующих конструкторов:

* **SpinnerListModel**();
* **SpinnerListModel**(**List** *values*);
* **SpinnerListModel**(**Object**[] *values*);

Первый конструктор создает модель для пустого списка, второй для списка объектов коллекции **List**, а третий для массива объектов класса **Object**.

Метод **void setValue**(**Object** *obj*) изменяет элемент в поле вывода и уведомляет блок прослушивания **ChangeListener**. Методы

* **Object getValue**();
* **Object getNextValue**();
* **Object getPreviousValue**().

позволяют получить соответственно текущее значение поля вывода, следующее значение поля вывода и предыдущее значение поля вывода. Если текущий элемент – последний, метод **getNextValue**() возвращает ***null***, а если текущий элемент – первый, метод **getPreviousValue**() возвращает ***null***.

Основными конструкторами модели чисел являются:

* **SpinnerNumberModel**();
* **SpinnerNumberModel**(**int** *value*, **int** *minimum*, **int** *maximum*, **int** *stepSize*)

где *value* – выводимое в поле вывода значение, *minimum* – минимальное значение списка, maximum – максимальное значение списка, *stepSize* – шаг списка. Пустой конструктор устанавливает список с нулевым значением в поле вывода, без ограничений на максимальное и минимальное значение и с шагом, равным 1.

Значения указанных параметров также можно установить с помощью методов

* **void setValue**(**Object** *value*);
* **void setMinimum**(**Comparable** *minimum*);
* **void setMaximum**(**Comparable** *maximum*);
* **void setStepSize**(**Number** *stepSize*);

а получить их значения – с помощью методов

* **Object getValue**();
* **Object getNextValue**();
* **Object getPreviousValue**();
* **Comparable getMinimum**();
* **Comparable getMaximum**();
* **Number getStepSize**().

Для работы с датами определены следующие конструкторы:

* **SpinnerDateModel**();
* **SpinnerDateModel**(**Date** *value*, **Comparable** *start*, **Comparable** *end*, **int** *calendarField*)

где *value* – выводимое значение даты, *start* и *end* – начальное и конечное значение даты, а допустимые значения для *calendarField*:

***Calendar.ERA*** – эра (значение ***GregorianCalendar.BC*** – до нашей эры или **GregorianCalendar.AD** – наша эра);

***Calendar.YEAR*** и ***Calendar.MONTH*** – значение года и месяца (от 0 до 11);

***Calendar.WEEK\_OF\_YEAR*** и ***Calendar.WEEK\_OF\_MONTH*** – неделя года и неделя месяца;

***Calendar.DAY\_OF\_MONTH*** и ***Calendar.DAY\_OF\_YEAR*** – день месяца и день года;

***Calendar.DAY\_OF\_WEEK*** и ***Calendar.DAY\_OF\_WEEK\_IN\_MONTH*** – день недели (от 1 до 7) и день недели в месяце (1-7 дни месяца – 1 неделя, 8-14 – 2 неделя и т.д.);

***Calendar.AM\_PM*** – до полудня – значение **Calendar.AM** или после полудня – значение **Calendar.PM**;

***Calendar.HOUR*** и ***Calendar.HOUR\_OF\_DAY***, - час в формате от 0 до 12 или от 0 до 24;

***Calendar.MINUTE***, ***Calendar.SECOND*** и ***Calendar.MILLISECOND*** – минута, секунда и миллисекунда.

Значения указанных параметров также можно установить с помощью методов:

* **void setValue**(**Object** *value*);
* **void setStart**(**Comparable** *start*);
* **void setEnd**(**Comparable** *end*);
* **void setCalendarField**(**int** *calendarField*)

а получить их значения – с помощью методов

* **Object getValue**();
* **Object getNextValue**();
* **Object getPreviousValue**();
* **Comparable getStart**();
* **Comparable getEnd**();
* **Number getCalendarField** ()

Метод **Date getDate**() возвращает текущий элемент в последовательности дат в виде объекта класса **Date**.

Для обработки событий в объектах класса **JSpinner** можно использовать интерфейс **ChangeListener** и класс **ChangeEvent**.

#### Класс JTextComponent

Родительским классом текстовых компонент в Swing является класс **JTextComponent**. Он обеспечивает функциональные возможности, которые являются общими для всех текстовых компонент.

Возможности компонента **JTextComponent** библиотеки Swing намного превосходят требования, предъявляемые к обычному текстовому полю или надписи. Данный компонент предоставляет разработчикам большой набор методов работы с текстом, например:

* **public void copy**() – копировать в буфер обмена;
* **public void cut**() – вырезать в буфер обмена;
* **public void paste**() – вставить из буфера обмена;
* **public String getSelectedText**() – получить выделенный текст;
* **public int getSelectionStart**() – определить точку начала выделенного текста;
* **public int getSelectionEnd**() – определить точку конца выделенного текста;
* **public String getText**() – ввести текст из поля;
* **public void setText**(**String** *text*) – поместить текст в поле;
* **public void setEditable** (**boolean** *b*) – разрешить редактирование;
* **public void setCaretPosition**(**int** *position*) – поместить курсор в указанную позицию.

#### Классы JTextField, JPasswordField и JTextArea

Подклассы класса **JTextComponent** – **JTextField** и **JTextArea** практически аналогичны классам **TextField** и **TextArea** пакета AWT. Между ними существуют лишь следующие отличия:

* строки ввода паролей задаются в Swing с помощью отдельного класса **JPasswordField** (конструкторы этого класса аналогичны конструкторам класса **JTextField**, метод **public char[] getPassword**() позволяет получить значение введенного пароля, а методы **public void setEchoChar(char c)** и **public char getEchoChar**() – установить и получить значение эхо-символа);
* прокрутка объекта класса **JTextArea** (так же, как и объекта класса **JList**) выполняется с помощью помещения списка в объект класса **JScrollPane**.

Для обработки событий в объектах класса **JButton** можно использовать интерфейс **ActionListener** и класс **ActionEvent**. Однако обычно с текстовыми компонентами связана кнопка, для которой и выполняется обработка события.

### 1.1.8. Меню

Компоненты **JMenuBar**, **JMenu**, **JMenuItem** и **JCheckBoxMenuItem** в Swing по своим функциональным возможностям и использованию аналогичны компонентам **MenuBar**, **Menu**, **MenuItem** и **CheckboxMenuItem**, используемым в AWT.

Новый компонент **JRadioButtonMenuItem** позволяет организовать в меню переключатели. Группа альтернативных переключателей формируется следующим образом: сначала создается объект класса **ButtonGroup**, а затем в этот объект с помощью метода **add**() добавляются объекты класса **JRadioButtonMenuItem**.

Классы, реализующие меню в Swing, отличаются от соответствующих классов AWT следующими основными особенностями:

* поскольку все классы, кроме **JMenuBar**, являются подклассами класса **JAbsractButton**, для меню и пунктов меню можно задавать не только текст, но и изображения (для этого в конструкторы этих классов введен параметр **Icon** *image* и добавлены методы, связанные с изображениями);
* строка меню устанавливается с помощью метода **public void setJMenuBar**(**JMenuBar** *menubar*) класса **JFrame**;
* добавлен интерфейс **MenuListener** для обработки событий связанных с меню (объектами класса **JMenu**). Методы этого интерфейса **public void menuSelected**(**MenuEvent** *e*), **public void menuDeselected**(**MenuEvent** *e*), **public void menuCanceled**(**MenuEvent** *e*) позволяют обрабатывать события, связанные с выбором меню, отменой выбора меню и отменой меню. Блоки прослушивания для меню добавляются и удаляются с помощью методов **public void addMenuListener**(**MenuListener** *l*) и **public void removeMenuListener**(**MenuListener** *l*);
* мнемонические клавиши («горячие» клавиши) для меню и пунктов меню в отличие от AWT действуют только тогда, когда соответствующие меню или пункты меню выведены на экране. Для того, чтобы клавиши действовали независимо от того, виден ли соответствующий элемент меню на экране, необходимо задать такие клавиши с помощью метода **public void setAccelerator**(**KeyStroke** *keyStroke*);
* для пунктов меню введен интерфейс **MenuDragMouseListener**, методы которого **public void menuDragMouseEntered**(**MenuDragMouseEvent** *e*), **public void menuDragMouseExited**(**MenuDragMouseEvent** *e*), **public void menuDragMouseDragged**(**MenuDragMouseEvent** *e*), **public void menuDragMouseReleased**(**MenuDragMouseEvent** *e*) позволяют организовать обработку событий, связанных с перемещением курсора мыши над пунктами меню (например, изменения вида пункта меню при наведении на него курсора мыши). Блоки прослушивания для меню добавляются и удаляются с помощью методов **public void addMenuDragMouseListener**(**MenuDragMouseListener** *l*) и **public void removeMenuDragMouseListener**( **MenuDragMouseListener** *l*);
* кроме этого, для пунктов меню введен интерфейс **MenuKeyListener**, методы которого **public void menuKeyTyped**(**MenuKeyEvent** *e*), **public void menuKeyPressed**(**MenuKeyEvent** *e*), **public void menuKeyReleased**(**MenuKeyEvent** *e*) позволяет организовать обработку событий, связанных с нажатием и отпусканием, а также отдельно с нажатием и отдельно с отпусканием клавиш, связанных с пунктами меню.

### 1.1.9. Бегунки и подсказки

#### Классы JScrollBar и JSlider

Полоса прокрутки (класс **JScrollBar**) в Swing практически не отличается от полосы прокрутки (класс **Scrollbar**) в AWT.

Бегунки реализуются в Swing с помощью класса **JSlider**. Этот компонент, как и **JScrollBar**, позволяет выбрать значение с помощью перемещения бегунка. Основными конструкторами класса **JSlider** являются

* **JSlider**();
* **JSlider**(**int** *orientation*);
* **JSlider**(**int** *min*, **int** *max*);
* **JSlider**(**int** *min*, **int** *max*, **int** *value*);
* **JSlider**(**int** *orientation*, **int** *min*, **int** *max*, **int** *value*)

где параметр *orientation* задает ориентацию, а параметры *min*, *max* и *value* соответственно минимальное, максимальное и текущее значение бегунка.

Для бегунков можно задать линейки с рисками большой или маленькой высоты с использованием методов **public void setMajorTickSpacing**(**int** *n*) и **public void setMinorTickSpacing**(**int** *n*), где параметр n задает расстояние между рисками в пикселях.

Для отслеживания событий бегунка используется интерфейс **ChangeListener** и класс **ChangeEvent**.

#### Класс JToolTip

«Всплывающие» подсказки (текст, появляющийся, если навести на некоторое время указатель мыши на какой-либо объект) реализуются в Swing с помощью класса **JToolTip**. Объект этого класса создается с помощью конструктора **JToolTip**().

Прикрепить подсказки к компоненту и получить компонент, к которому прикреплена данная подсказка, можно с помощью методов **void setComponent**(**JComponent** *comp*) и **JComponent getComponent**(), а установка и получение текста подсказки выполняется с помощью методов **void setTipText**(**String** *tipText*) и **String getTipText**().

Методы класса **JComponent public void setToolTipText**(**String** *text*) и **public String getToolTipText**() позволяют установить и получить подсказку непосредственно для любого графического компонента.

## 1.2. Основные менеджеры компоновки Java

### 1.2.1. Принципы функционирования менеджеров компоновки

При добавлении компонентов в контейнер обычно не приходится указывать положение каждого компонента в пределах контейнера. С помощью менеджеров компоновки можно указать, как те или иные компоненты должны быть расположены по отношению к другим компонентам. Точные значения координат вычисляет сам менеджер. Таким образом, облегчается построение программ, не зависящих от используемого Web-браузера и разрешения дисплея.

**LayoutManager** является интерфейсом в библиотеке классов Java, описывающим как класс **Container** и менеджер компоновки взаимодействуют между собой. Он объявляет несколько методов, которые:

* располагают контейнер в заданной панели – метод **public void layoutContainer** (**Container** *parent*);
* информируют менеджер компоновки о добавлении и удалении компонентов из контейнера – методы **public void addLayoutComponent**(**String** *name*, **Component** *comp*) и **public void removeLayoutComponent**(**Component** *comp*);
* задают размеры компонентов, которыми они управляют – методы **public Dimension minimumLayoutSize**(**Container** *parent*) и **public Dimension preferredLayoutSize** (**Container** *parent*).

Дополнительный интерфейс с методами позиционирования и проверки, **LayoutManager2**, был добавлен в JDK 1.1. В этом интерфейсе объявлен метод **public Dimension maximumLayoutSize**(**Container** *parent*) и методы, возвращающие выравнивание по осям x и y:

* **public float getLayoutAlignmentX**(**Container** *target*);
* **public float getLayoutAlignmentY**(**Container** *target*).

Кроме того, добавлен метод **public void addLayoutComponent**(**Component** *comp*, **Object** *constraints*), добавляющий заданный компонент с использованием заданного объекта-ограничителя и метод **public void invalidateLayout** (**Container** *target*), делающий контейнер недействительным.

В AWT определены пять различных типов менеджеров компоновки:

* **FlowLayout** – последовательное расположение;
* **GridLayout** – табличное расположение;
* **BorderLayout** – полярное расположение;
* **CardLayout** – блокнотное расположение;
* **GridBagLayout** – ячеистое расположение.

Все эти менеджеры реализуют интерфейс **LayoutManager**, а менеджеры **BorderLayout**, **CardLayout** и **GridBagLayout** – также интерфейс **LayoutManager2**. Установка менеджера компоновки для контейнера выполняется с помощью метода **public void setLayout**(**LayoutManager** *manager*), где параметр manager определяет объект одного из классов менеджеров компоновки, например: **setLayout**(**new GridLayout**(3,2));

Метод **public LayoutManager getLayout**() возвращает менеджер компоновки для данного контейнера.

Для определения промежутков между границей контейнера и содержащихся в нем компонент используется класс **Insets**. В данном классе содержатся свойства

* **public int** top,
* **public int** left,
* **public int** bottom,
* **public int** right,

задающие отступ сверху, справа, снизу и слева (в пикселях). Менеджер компоновки создает объект класса **Insets**, обратившись к конструктору класса **public Insets**(**int** *top*, **int** *left*, **int** *bottom*, **int** *right*), который возвращает объект **Insets**. Параметры *top*, *left*, *bottom* и *right* определяют промежутки (в пикселях) соответственно от верхней, левой, нижней и правой границ контейнера. Например, если требуется оставить пространство шириной 20 пикселей снизу и сверху и 10 пикселей по бокам, нужно определить в программе следующий метод **insets**():

public Insets insets()

{

return new Insets (20, 10, 20, 10);

}

Для выполнения своей работы менеджер компоновки должен связываться с объектом класса **Container**. Если контейнер не имеет связанного с ним менеджера компоновки, он просто помещает компоненты с использованием методов класса **Component**

* **public void setBounds**(**Rectangle** *r*), передвигающим и изменяющим размеры компонента так, чтобы он вписывался в заданный прямоугольник,
* **public void setLocation**(**Point** *p*), перемещающим компонент или
* **public void setSize**(**Dimension** *d*), изменяющим размеры компонента.

Если контейнер имеет связанный с ним менеджер компоновки, контейнер запрашивает менеджер компоновки о позиционировании и размерах своих компонент перед тем, как они будут нарисованы. Сам менеджер компоновки не выполняет рисования, он просто решает, какой размер и позицию должен иметь каждый компонент и вызывает методы **setBounds**(), **setLocation**() и/или **setSize**() для каждого из этих компонентов.

Как уже указывалось выше, **LayoutManager** связан с объектом класса **Container** вызовом метода **setLayout**(), например:

**Panel** p = **new Panel**();

p.**setLayout**(**new BorderLayout**());

Некоторые контейнеры, такие как **Panel**, обеспечивает конструктор, который в качестве аргумента воспринимает менеджер компоновки:

**Panel** *p* = **new Panel**(**new BorderLayout**());

При создании собственных менеджеров компоновки также должен быть реализован интерфейс **LayoutManager** и, возможно, **LayoutManager2**.

Собственные менеджеры расположения обычно используются в тех случаях, когда возможности, предоставляемые стандартными менеджерами, являются недостаточными.

В тех случаях, когда требуется вручную расположить компоненты, используется абсолютное позиционирование компонент. Поскольку даже когда менеджер компоновки не задан, действует менеджер по умолчанию (**FlowLayout** для **Panel** и **BorderLayout** для **Frame**), необходимо отключить менеджеры компоновки, действующие по умолчанию, с помощью оператора **setLayout**(***null***);

### 1.2.2. Менеджер FlowLayout

Класс **FlowLayout** рассматривает как контейнер набор строк. Этот менеджер используется по умолчанию в объектах классов **Panel** и **Applet**.

Это самый простой из менеджеров компоновки AWT. Его стратегия компоновки являются следующей:

* учитывать предпочитаемый размер всех содержащихся компонент.
* скомпоновать как можно больше компонент горизонтально внутри контейнера.
* начать новую строку компонент, если они еще есть.
* если все компоненты не соответствуют размерам, то, компоненты, которые не помещаются, не выводятся.

Высота каждой строки определяется по высоте находящихся в ней компонентов. Менеджер **FlowLayout** добавляет компоненты слева направо. Если следующий компонент не помещается в текущей строке, происходит переход на новую строку, которая опять начинается слева. Можно размещать строки с выравниванием влево, вправо и по центру. По умолчанию используется выравнивание по центру.

Конструктор **public FlowLayout**(**int** *alignment*, **int** *hgap*, **int** *vgap*) создает объект **FlowLayout**. Параметр *alignment* регулирует выравнивание строк, а возможные типы выравнивания задаются следующими переменными класса **FlowLayout** с модификаторами **public static final int**: ***LEFT*** (влево), ***RIGHT*** (вправо) и ***CENTER*** (по центру). Параметры *hgap* и *vgap* задают размеры промежутков между компонентами по вертикали и горизонтали (в пикселях). Так, чтобы расположить компоненты с правым выравниванием с промежутком 10 пикселей по вертикали и 5 пикселей по горизонтали, надо создать объект **FlowLayout** следующим образом:

myFlowLayout = **new FlowLayout**(**FlowLayout**.***RIGHT***, 10, 5);

Можно также использовать конструкторы **public FlowLayout**(**int** *alignment*) и **public FlowLayout**() (по умолчанию *hgap* и *vgap* равны 5 пикселей и выравнивание производится по центру).

Предпочтительным для **FlowLayout** будет размещение всех его компонент в одном ряду. Это означает, что его предпочтительная высота должна быть максимальной высотой компонент плюс промежуток, равный *vgap*. Предпочтительная ширина должна быть суммой всех значений ширины содержащихся компонент плюс расстояния между компонентами плюс расстояния между компонентом и границами ряда, равные *hgap* (*vgap* и *hgap* являются общими свойствами большинства стандартных менеджеров компоновки и определяют, как далеко компоненты размещаются друг от друга).

Класс **FlowLayout**, помимо методов интерфейса **LayoutManager**, обеспечивает также следующие методы:

* **public int getAlignment**() и **public void setAlignment**(**int** *align*) – получение и установка выравнивания;
* **public int getVgap**() и **public void setVgap**(**int** *vgap*) – получение и установка промежутка по вертикали между компонентами;
* **public int getHgap**() и **public void setHgap**(**int** *hgap*) – получение и установка промежутка по горизонтали между компонентами.

Менеджер **FlowLayout** ориентирован по горизонтали. Если необходима вертикальная компоновка, нужно написать свой собственный менеджер (или использовать менеджер **BoxLayout** из библиотеки Swing).

### 1.2.3. Менеджер GridLayout

Класс **GridLayout** разделяет контейнер на клетки одинакового размера (по типу таблицы). При добавлении компонентов в контейнер **GridLayout** конструктор **public GridLayout**(**int** *rows*, **int** *cols*, **int** *hgap*, **int** *vgap*) размещает их в клетках слева направо и сверху вниз. Параметры *rows* и *cols* задают количество строк или столбцов таблицы, *hgap* а и *vgap* – соответственно промежутки между столбцами и строками. Если задано количество строк, количество столбцов будет вычислено. Если, наоборот, задать количество столбцов, вычисляться будет количество строк. Если добавить шесть компонентов в объект **GridLayout**, содержащий две строки, он создаст три столбца. Если создается **GridLayout** с заданным числом строк, следует в качестве числа столбцов указать значение 0. Если же надо задать число столбцов, то вместо числа строк следует задать значение 0 (если оба значения ненулевые, будет использоваться только количество строк, а число столбцов будет вычислено, исходя из количества строк и числа компонентов в контейнере). Так предложение **GridLayout** *myGridLayout* = **new GridLayout**(0, 4, 8, 10); создает объект **GridLayout** с четырьмя столбцами, промежутком между столбцами 8 пикселей, а между строками — 10 пикселей. Задание нулевых значений и для rows и для cols приводит к возбуждению исключения **IllegalArgumentException**.

Можно также создать объект **GridLayout** с помощью конструктора **public GridLayout**(**int** *rows*, **int** *cols*) (в этом случае промежутков между столбцами и между строками не будет).

Конструктор **public GridLayout**() задает объект с одной строкой и одной колонкой.

Менеджер **GridLayout** старается установить размер каждого компонента к максимальной предпочитаемой ширине и максимальной предпочитаемой высоте (все компоненты **GridLayout** будут одного и того же размера). Поэтому предпочитаемая ширина *pw* и высота *ph* для **GridLayout** рассчитывается по следующей формуле:

pw = (maxPrefWidth \* cols) + (hgap \* (cols+1))

ph = (maxPrefHeight \* rows) + (vgap \* (rows+1))

Класс **GridLayout**, помимо методов интерфейса **LayoutManager**, обеспечивает также следующие методы:

* **public int getAlignment**() и **public void setAlignment**(**int** *align*) – получение и установка выравнивания;
* **public int getVgap**() и **public void setVgap**(**int** *vgap*) – получение и установка промежутка по вертикали между компонентами;
* **public int getHgap**() и **public void setHgap**(**int** *hgap*) – получение и установка промежутка по горизонтали между компонентами;
* **public int getRows**() и **public void setRows**(**int** *rows*) – получение и установка количества строк;
* **public int getColumns**() и **public void setColumns**(**int** *rows*) – получение и установка количества столбцов.

### 1.2.4. Менеджер BorderLayout

Класс **BorderLayout** разделяет контейнер на пять областей, условно называемых "***North***" (север), "***South***" (юг), "***East***" (восток), "***West***" (запад) и "***Center***" (центр).

Этот менеджер используется по умолчанию в объектах класса **Frame** и его подклассах.

На самом деле из пяти приведенных областей можно использовать, в зависимости от задачи, две или три области.

Менеджер **BorderLayout** требует ограничителя при добавлении компонент. Ограничитель может быть одной из следующих переменных класса **BorderLayout** с модификаторами **public static final int**: ***NORTH***, ***SOUTH***, ***EAST***, ***WEST*** или ***CENTER***.

Эти ограничители определяются с помощью следующего метода класса **Container**: **public void add**(**Component** *component*, **Object** *constraint*), например: **add**(**new Button**("Север"), **BorderLayout**.***NORTH***);

Объект **BorderLayout** создается с помощью одного из конструкторов:

* **public BorderLayout**();
* **public BorderLayout**(**int** *hgap*, **int** *vgap*)

где параметры *vgap* и *hgap* задают промежутки по вертикали и горизонтали (в случае конструктора без параметров между компонентами нет промежутков).

Класс **BorderLayout** не позволяет добавить в данную область более одного компонента. Если попытаться добавить два компонента в одну и ту же область, виден будет лишь тот из них, который был добавлен последним.

Класс **BorderLayout**, помимо методов интерфейсов **LayoutManager** и **LayoutManager2**, обеспечивает также следующие методы:

* **public int getVgap**() и **public void setVgap**(**int** *vgap*) – получение и установка промежутка по вертикали между компонентами;
* **public int getHgap**() и **public void setHgap**(**int** *hgap*) – получение и установка промежутка по горизонтали между компонентами.

### 1.2.5. Менеджер компоновки BoxLayout

Компоновка **BoxLayout** в целом подобна стандартной компоновке библиотеки AWT **FlowLayout**, за исключением того, что при работе с новой компоновкой можно определить используемую ось (с помощью статических переменных класса **BoxLayout**: ***Х\_AXIS*** или ***Y\_AXIS***). В отличие от компоновки **GridLayout**, здесь каждый компонент может занимать ячейку различного размера. Для использования компоновки **BoxLayout** с ориентацией по оси Y достаточно поместить в программу следующий оператор:

**setLayout**(**new BoxLayout**(this, **BoxLayout**.***Y\_AXIS***));

Если в программе желательно применить компоновку типа **BoxLayout**, в качестве контейнера вместо класса **JPanel** следует использовать класс **Box** (для этого класса компоновка **BoxLayout** используется по умолчанию). Кроме того, класс **Box** предоставляет несколько дополнительных методов, обеспечивающих полный контроль над размещением компонентов в окне.

Так, в число размещаемых на панели компонентов можно включить невидимые компоненты-заполнители трех видов.

Первый вид заполнителя – невидимая разделительная область (rigid area), имеющая фиксированные размеры. Она создается с помощью метода **public static Component createRigidArea**(**Dimension** *d*) и вставляется между компонентами, создавая промежуток фиксированного размера между ними.

Заполнитель второго вида – невидимая «распорка» (strut) имеет только один фиксированный размер. У горизонтальной распорки, создаваемой статическим методом **public static Component createHorizontalStrut**(**int** *width*) фиксирована только ширина. При горизонтальном расположении компонентов распорку можно использовать для создания промежутков между компонентами, а при вертикальном расположении — для задания ширины всей панели.

У вертикальной распорки, создаваемой статическим методом **public static Component createVerticalStrut**(**int** *height*) фиксирована высота. Она используется аналогично горизонтальной распорке.

Третий вид заполнителя – невидимая «надувная подушка» (glue), «раздуваясь», заполняет все выделенное ей пространство, раздвигая остальные компоненты и прижимая их к краям панели, если они имеют фиксированный максимальный размер. Этот заполнитель создается одним из статических методов:

* **public static Component createGlue**() – «подушка» раздувается во все стороны;
* **public static Component createHorizontalGlue**() – «подушка» раздается в ширину;
* **public static Component createVerticalGlue**() – «подушка» раздается в высоту.

Кроме этих трех компонентов-разделителей можно использовать невидимый компонент с фиксированным минимальным, максимальным и предпочтительным размерами. Он является объектом класса **Filler**, вложенного в класс **Box**, и создается конструктором **Box.Filler**(**Dimension** *min*, **Dimension** *pref*, **Dimension** *max*). Преимущество этого объекта в том, что он может поменять размеры с помощью метода **void changeShape**(**Dimension** *min*, **Dimension** *pref*, **Dimension** *max*).

### 1.2.6. Менеджер компоновки SpringLayout

Менеджер компоновки **SpringLayout** использует для определения положения и размеров компонента использует координаты x, у и размерами width, height, задаваемыми объектами абстрактного класса **Spring**.

Каждый объект этого класса содержит четыре величины: минимальный, максимальный и предпочтительный размеры, а также текущий размер. Эти размеры могут использоваться как размеры промежутков между компонентами. Объект класса **Spring** задается а методом **public static Spring constant**(**int** *min*, **int** *pref*, **int** *max*), с заданным минимальным, предпочитаемым и максимальным значениями, а метод **public static Spring constant**(**int** *size*) возвращает объект класса **Spring** с совпадающими между собой размерами, равными *size*.

Текущий размер *value*, устанавливается методом **public abstract void setValue**(**int** *value*).

Минимальный, максимальный, предпочитаемый и текущий размер можно получить с помощью методов

* **public abstract int getMinimumValue**(),
* **public abstract int getMaximumValue**(),
* **public abstract int getPreferredValue**(),
* **public abstract int getValue**().

При использовании менеджером **SpringLayout** нескольких объектов класса для учета их взаимодействия в классе **Spring** определены несколько методов.

Так, метод

* **public static Spring max**(**Spring** *s1*, **Spring** *s2*) возвращает новый объект, размеры которого составлены из наибольших значений объектов *s1* и *s2*,
* **public static Spring minus**(**Spring** *s*) возвращает новый объект, размеры которого равны размерам объекта *s* с обратным знаком,
* **public static Spring sum**(**Spring** *s1*, **Spring** *s2*) возвращает новый объект, размеры которого равны сумме соответствующих размеров объектов *s1* и *s2*.

Объект класса **Spring** используется для построения объекта вложенного класса **SpringLayout**.**Constraints**.

Объект этого класса содержит координаты *x* и *у*, ширину *width* и высоту *height*, являющиеся объектами класса **Spring**. Менеджер размещения **SpringLayout** меняет положение левого верхнего угла компонента в заданных объектами x и y пределах. Поэтому координаты обозначаются статическими строковыми константами ***WEST*** и ***NORTH*** класса **SpringLayout**. Величины x + width и y + height обозначаются статическими строковыми константами ***EAST*** и ***SOUTH*** и определяют положение правого нижнего угла компонента.

Для получения и установки всех этих значений в классе **SpringLayout**.**Constraints** определены методы:

* **void setX**(**Spring** *x*) и **Spring getX**() – установка и получение координаты *x*;
* **void setY**(**Spring** *y*) и **Spring getY**() – установка и получение координаты y;
* **void setWidth**(**Spring** *width*) и **Spring getWidth**() – установка и получение ширины;
* **void setHeight**(**Spring** *height*) и **Spring getHeight**()– установка и получение высоты.

Методы **public void setConstraint**(**String** *edgeName*, **Spring** *s*) и **public Spring getConstraint**(**String** *edgeName*) устанавливают и выдают объект класса Spring по заданному имени NORTH, WEST, SOUTH или EAST.

### 1.2.7. Менеджер компоновки GroupLayout

Менеджер **GroupLayout** иерархически группирует компоненты для их позиционирования в контейнере и поддерживает два типа групп. Группы первого типа выстраивают свои элементы последовательно, группы второго типа – параллельно четырьмя разными способами. Группирование выполняется с помощью объектов класса **Group**.

Каждая группа может содержать любое количество объектов класса **Group** или **Component**, а также промежуток (gap). Компоненты менеджера похожи на пружины. Каждый компонент имеет диапазон размеров, задаваемый минимальным, предпочитаемым и максимальным размерами, определяемыми с помощью методов **getMinimumSize**(), **getPreferredSize**() и **getMaximumSize**() класса **Component**. Промежуток также рассматривается как невидимый компонент с минимальным, предпочитаемым и максимальным размером. Кроме того, менеджер поддерживает предпочитаемый промежуток, который можно получить с помощью метода **getPreferredGap**() класса **LayoutStyle**.

Диапазон для объекта класса определяется типом группы: для последовательных групп диапазон равен сумме элементов, для параллельных – диапазону для максимального элемента.

Менеджер **GroupLayout** рассматривает каждую ось независимо, т.е. существует группа, представляющая вертикальную ось и группа, представляющая горизонтальную ось. Вертикальная ось определяет минимальный, предпочитаемый и максимальный размер по этой оси, установку координаты y и высоты компонент, а горизонтальная ось – минимальный, предпочитаемый и максимальный размер по этой оси, установку координаты x и ширины компонент. Каждый компонент должен быть задан и в вертикальной и в горизонтальной группе.

Конструктор **public GroupLayout**(**Container** *host*) задает групповую компоновку для контейнера host.

Метод **public void setAutoCreateGaps**(**boolean** *autoCreatePadding*) задает (при значении параметра *autoCreatePadding*, равном **true**) или отменяет (при значении параметра *autoCreatePadding*, равном **false**) автоматическую вставку промежутков между компонентами.

Горизонтальная и вертикальная группы создаются с помощью методов

* public void setHorizontalGroup(GroupLayout.Group group)
* public void setVerticalGroup(GroupLayout.Group group)

Оба эти метода содержат в качестве параметра объект group вложенного класса Group, который задает позиции и размеры компонент вдоль горизонтальной или вертикальной оси.

Последовательная или параллельная группы создаются с помощью методов

* **public GroupLayout.SequentialGroup createSequentialGroup**()
* **public GroupLayout.ParallelGroup createParallelGroup**()
* **public GroupLayout.ParallelGroup createParallelGroup** (**GroupLayout.Alignment** *alignment*)
* **GroupLayout.ParallelGroup createParallelGroup**(**GroupLayout.Alignment** *alignment*, **boolean** *resizable*)

Предпоследний и последний методы задают (с помощью параметра alignment) выравнивание компонент в группе с помощью перечислимых переменных класса **GroupLayout.Alignment**: ***LEADING*** (выравнивание к началу группы), ***TRAILING*** (выравнивание к концу группы), ***CENTER*** (выравнивание по центру) и ***BASELINE*** (выравнивание по базовой линии). Последний метод задает также (с помощью параметра resizable) возможность изменения размера компонента.

Методы

* **public void linkSize**(**Component**... *components*)
* **public void linkSize**(**int** *axis*, **Component**... *components*)

позволяют установить одинаковый размер для всех заданных в качестве параметров компонент группы (второй метод – только для заданной оси, которая задается статическими константами ***HORIZONTAL*** и ***VERTICAL*** класса **SwingConstants**).

## 1.3. Обработка событий в Java

### 1.3.1. Модель делегирования событий в Java

Операционная система, например Microsoft Windows, отслеживает все события, происходящие в системе. Система направляет эти события в соответствующие целевые объекты. Если, к примеру, пользователь щелкает по окну апплета, система создает событие mouse-down (нажатие клавиши мыши) и отсылает его данному окну для обработки. Затем это окно может выполнить некоторую операцию или попросту передать событие обратно системе для обработки по умолчанию.

В языке Java события представляют собой объекты некоторых классов.

В JDK 1.0 существовал единственный класс **Event**, описывающий все события, на которые программа может реагировать, а также определяющий методы по умолчанию для получения информации о конкретном событии.

Начиная с JDK 1.1, в Java были изменена модель обмена информацией о событиях, названная моделью делегирования событий. Старая модель и класс **Event** были сохранены для возможности использования старых программ, однако использовать старую модель не рекомендуется.

Модель делегирования событий работает следующим образом. Каждый элемент взаимодействия между интерфейсом пользователя и программой определяется как событие. Классы приложений выясняют свою заинтересованность в некотором ожидаемом определенным компонентом событии путем опроса компонента и выдачи ему предложения поместить в список сведения о блоке прослушивания данного компонента (listener). Когда происходит некоторое событие, источник события уведомляет о нем зарегистрированные блоки прослушивания.

События, как и исключения, образуют иерархию классов. Корнем этой иерархии является класс **EventObject**, расширяющий класс **Object**.

Для класса определены два метода:

* **public Object getSource**();
* **public String toString**(),

определяющие соответственно объект – источник события и строковое представление объекта-события.

События, связанные с графическим интерфейсом пользователя, в свою очередь, образуют ветвь в иерархии классов событий, корнем которой является класс **AWTEvent**, расширяющий класс **EventObject**.

Существуют два различных типа событий – *низкоуровневые* и *семантические*.

Низкоуровневые события связаны с физическими аспектами интерфейса пользователя – щелчками мыши, нажатиями клавиш клавиатуры и т.д. Низкоуровневые события обрабатывает класс ComponentEvent и его потомки.

Семантические события строятся на базе низкоуровневых событий. Например, для выбора команды меню может потребоваться раскрыть список команд первым щелчком мыши, а затем вторым щелчком выбрать в нем требуемую команду. Эта последовательность низкоуровневых событий может быть объединена в одно семантическое событие. Все остальные события, кроме событий ветви ComponentEvent, являются семантическими (эти компоненты будут рассмотрены далее)

### 1.3.2. Классы и интерфейсы обработки низкоуровневых событий

В каждом классе событий определены соответствующие свойства и методы для обработки событий.

Почти каждый тип события, за исключением **PaintEvent** и **InputEvent**, имеет связанный с ним интерфейс или интерфейсы блоков прослушивания, в которых объявлены методы обработки данного типа события. Эти интерфейсы являются расширениями интерфейса **EventListener**, не определяющего никаких переменных и методов.

Рассмотрим основные свойства и методы классов низкоуровневых событий и соответствующих им блоков прослушивания.

#### Класс ComponentEvent и интерфейс ComponentListener

Класс **ComponentEvent** обрабатывает события, связанные с изменением компонента. Причину изменения можно определить с помощью следующих статических final переменных класса типа int:

* COMPONENT\_MOVED – компонент переместился;
* COMPONENT\_RESIZED – компонент изменил размер;
* COMPONENT\_HIDDEN – компонент убран с экрана;
* COMPONENT\_SHOWN – компонент появился на экране;
* COMPONENT\_FIRST – первый номер в списке идентификаторов, используемых для событий компонента;
* COMPONENT\_LAST – последний номер в списке идентификаторов, используемых для событий компонента.

Метод класса **public Component getComponent**() возвращает компонент – источник события.

Интерфейс **ComponentListener** содержит объявления следующих методов обработки событий компонент:

* **public void componentResized**(**ComponentEvent** *e*) – обработка события изменения размера компонента;
* **public void componentMoved** (**ComponentEvent** *e*) – обработка события перемещения компонента;
* **public void componentShown**(**ComponentEvent** *e*) – обработка события появления компонента на экране;
* **public void componentHidden**(**ComponentEvent** *e*) – обработка события удаления компонента с экрана.

#### Класс ContainerEvent и компонент ContainerListener

Класс **ContainerEvent** обрабатывает события добавления или удаления компонента и содержит следующие статические final переменные типа int:

* COMPONENT\_ADDED – в контейнер добавлен компонент;
* COMPONENT\_REMOVED – из контейнера удален компонент;
* CONTAINER\_FIRST – первый номер в списке идентификаторов, используемых для событий контейнера;
* CONTAINER\_LAST – последний номер в списке идентификаторов, используемых для событий контейнера.

Методы класса

* **public Component getChild**()
* **public Container getContainer**()

возвращают соответственно компонент, на который действует событие и контейнер – источник события.

Интерфейс **ContainerListener** содержит объявления следующих методов:

* **public void componentAdded**(**ContainerEvent** *e*) – обработка события добавления компонента в контейнер;
* **public void componentRemoved**(**ContainerEvent** *e*) – обработка события удаления компонента из контейнера.

#### Класс FocusEvent и интерфейс FocusListener

Класс **FocusEvent** обрабатывает события приобретения или потери фокуса и содержит следующие статические final переменные типа int:

* FOCUS\_GAINED – получение фокуса ввода;
* FOCUS\_LOST – потеря фокуса ввода;
* FOCUS\_FIRST – первый номер в списке идентификаторов, используемых для событий фокуса;
* FOCUS\_LAST – последний номер в списке идентификаторов, используемых для событий фокуса.

Метод класса

* **public Component getOppositeComponent**() возвращает другой компонент, связанный со сменой фокуса, а метод
* **public boolean isTemporary**() возвращает true, если событие, связанное с изменением фокуса, является временным и false – в противном случае.

В интерфейсе **FocusListener** содержатся объявления двух методов:

* **public void focusGained**(**FocusEvent** *e*) – обработка события получения фокуса;
* **public void focusLost**(**FocusEvent** *e*) – обработка события потери фокуса.

#### Класс WindowEvent и интерфейсы WindowListener, WindowStateListener и WindowFocusListener

Класс **WindowEvent** обрабатывает события, связанные с изменением состояния окна и содержит следующие статические final переменные типа int:

* WINDOW\_OPENED – окно открылось;
* WINDOW\_CLOSED – окно закрылось;
* WINDOW\_CLOSING – попытка закрытия окна;
* WINDOW\_ACTIVATED – окно получило фокус;
* WINDOW\_DEACTIVATED – окно потеряло фокус;
* WINDOW\_ICONIFIED – окно свернулось в ярлык;
* WINDOW\_DEICONIFIED – окно развернулось;
* WINDOW\_GAINED\_FOCUS – окно получило фокус;
* WINDOW\_LOST\_FOCUS – окно потеряло фокус;
* WINDOW\_STATE\_CHANGED – любое изменение состояния окна из перечисленных выше;
* WINDOW\_FIRST – первый номер в списке идентификаторов, используемых для событий окна;
* WINDOW\_LAST – последний номер в списке идентификаторов, используемых для событий окна.

Методы класса

* **public int getNewState**()
* **public int getOldState**()

для события WINDOW\_STATE\_CHANGED возвращает новое или предыдущее состояние окна. Возвращаемое значение можно сравнивать на равенство со следующими статическими целыми final переменными класса Frame:

* NORMAL – нормальное состояние окна;
* ICONIFIED – свернутое окно;
* MAXIMIZED\_HORIZ – увеличение до максимального размера по горизонтали;
* MAXIMIZED\_VERT – увеличение до максимального размера по вертикали;
* MAXIMIZED\_BOTH – увеличение до максимального размера по горизонтали и вертикали.

Методы

* **public Window getOppositeWindow**()
* **public Window getWindow**()

возвращает соответственно другое окно, которое затронуло данное событие и окно – источник события.

Интерфейс **WindowListener** содержит объявления следующих методов обработки событий окна:

* **public void windowOpened**(**WindowEvent** *e*) – обработка открытия окна;
* **public void windowClosing**(**WindowEvent** *e*) – обработка попытки закрытия окна;
* **public void windowClosed**(**WindowEvent** *e*) – обработка закрытия окна;
* **public void windowIconified**(**WindowEvent** *e*) – обработка сворачивания окна;
* **public void windowDeiconified**(**WindowEvent** *e*) – обработка разворачивания окна;
* **public void windowActivated**(**WindowEvent** *e*) – обработка активизации окна;
* **public void windowDeactivated**(**WindowEvent** *e*) – обработка деактивизации окна.

Начиная с версии 1.4 добавлены два новых интерфейса для обработки событий окна.

Интерфейс **WindowStateListener**, содержит объявление метода **public void windowStateChanged**(**WindowEvent** e), который обрабатывает событие изменения окна.

Второй интерфейс **WindowFocusListener** содержит объявления следующих двух методов:

* **public void windowGainedFocus**(**WindowEvent** *e*) – обработка события приобретения окном фокуса;
* **public void windowLostFocus**(**WindowEvent** *e*) – обработка события потери окном фокуса.

#### Класс InputEvent

Класс **InputEvent** является корневым классом для всех событий ввода уровня компонента. События ввода направляются блокам прослушивания до их обычной обработки источником, который вызвал данное событие. Это позволяет блокам прослушивания и подклассам компонент «потреблять» событие так, чтобы источник не обрабатывал их обычным образом. Например, «потребление» события нажатия кнопки предотвращает активизацию кнопки.

Класс содержит следующие статические final переменные типа int, которые используются при обработки событий мыши и клавиатуры:

* BUTTON1\_DOWN\_MASK, BUTTON2\_DOWN\_MASK и BUTTON3\_DOWN\_MASK – нажатие соответственно первой, второй или третьей кнопки мыши;
* CTRL\_DOWN\_MASK, ALT\_DOWN\_MASK и SHIFT\_DOWN\_MASK – нажатие соответственно клавиш Ctrl, Alt или Shift.

Методы класса

* **public void consume**()
* **public boolean isConsumed**()

соответственно «потребляет» событие так, чтобы оно не обрабатывалось обычным образом и проверяет, не использовалось ли потребление события.

Методы

* **public int getModifiers**()
* **public static String getModifiersExText**(**int** *modifiers*)

соответственно возвращает модификатор для данного события (его значение совпадает с одной из определенных выше переменных) и получает текстовое значение для нажатых клавиш модификаторов, например, "Shift", "Button1" или "Ctrl+Shift".

Метод **public int getModifiersEx**() позволяет получить расширенные значения модификаторов, которые представляют состояние всех клавиш-модификаторов и кнопок мыши.

Метод **public long getWhen**() позволяет получить отметку о времени возникновения события (в миллисекундах с 1 января 1970 года).

Методы

* **public boolean isAltDown**()
* **public boolean isControlDown**()
* **public boolean isShiftDown**()

позволяют определить были ли нажаты соответственно клавиши Alt, Ctrl или Shift во время возникновения события.

#### Класс KeyEvent и интерфейс KeyListener

Класс **KeyEvent** обрабатывает события клавиатуры и содержит следующие статические final переменные типа int:

* KEY\_PRESSED – клавиша нажата;
* KEY\_RELEASED – клавиша отпущена;
* KEY\_TYPED – введен символ;
* KEY\_LOCATION\_LEFT – клавиша расположена слева на клавиатуре (для нескольких одинаковых клавиш);
* KEY\_LOCATION\_RIGHT – клавиша расположена справа на клавиатуре (для нескольких одинаковых клавиш);
* KEY\_LOCATION\_NUMPAD – клавиша расположена на цифровой клавиатуре (для нескольких одинаковых клавиш);
* KEY\_LOCATION\_STANDARD – положение клавиши стандартное (для тех клавиш, у которых нет совпадающих с ними клавиш);
* KEY\_LOCATION\_UNKNOWN – положение клавиши не определилось (обычно для события KEY\_TYPED).

Кроме того, в классе **KeyEvent** определены коды всех клавиш в виде констант, называемых виртуальными кодами клавиш (virtual key codes), например, VK\_F1, VK\_SHIFT, VK\_A, VK\_B, VK\_PLUS. Они перечислены в документации к классу KeyEvent. Фактическое значение виртуального кода зависит от языка и раскладки клавиатуры.

Методы класса

* **public char getKeyChar**()
* **public int getKeyCode**()
* **getKeyLocation**()

выводят соответственно значение символа, кода символа и положение клавиши на клавиатуре (его можно сравнить с одной из приведенных выше переменных класса).

Методы

* **public static String getKeyText**(**int** *keyCode*)
* **public static String getKeyModifiersText**(**int** *modifiers*)

возвращают соответственно строку, описывающую клавишу, например, "HOME", "F1" или "A".и текст, описывающий клавишу-модификатор, например, "Shift" или "Ctrl+Shift".

Метод

public boolean isActionKey()

проверяет, не является ли клавиша, вызвавшая событие клавишей «действия» (обычно клавиши «действия» – это клавиши, которые не являются клавишами символов или клавишами модификаторов, например, клавиша Esc).

Методы

* **public void setKeyChar**(**char** *keyChar*)
* **public void setKeyCode**(**int** *keyCode*)
* **public void setModifiers**(**int** *modifiers*)

позволяет установить соответственно значение символа, кода и клавиши-модификатора.

Кроме указанных свойств и методов, класс **KeyEvent** наследует все свойства и методы класса **InputEvent**.

Интерфейс **KeyListener** содержит объявления следующих методов:

* **public void keyTyped**(**KeyEvent** *e*) – обработка события ввода символа;
* **public void keyPressed**(**KeyEvent** *e*) – обработка события нажатия клавиши;
* **public void keyReleased**(**KeyEvent** *e*) – обработка события отпускания клавиши.

#### Класс MouseEvent и интерфейсы MouseListener и MouseMotionListener

Класс **MouseEvent** обрабатывает события клавиатуры и содержит следующие статические final переменные типа int:

* MOUSE\_PRESSED – нажатие кнопки мыши;
* MOUSE\_RELEASED – отпускание кнопки мыши;
* MOUSE\_CLICKED – щелчок кнопкой мыши;
* MOUSE\_MOVED – перемещение мыши;
* MOUSE\_DRAGGED – перемещение мыши с нажатой кнопкой;
* MOUSE\_ENTERED – появление курсора мыши в компоненте;
* MOUSE\_EXITED – выход курсора мыши из компонента;
* BUTTON1, BUTTON2 и BUTTON3 – первая, вторая и третья кнопки мыши;
* MOUSE\_WHEEL – вращение колесика мыши;
* NOBUTTON – кнопка мыши не использовалась во время события;
* MOUSE\_FIRST – первый номер в списке идентификаторов, используемых для событий мыши;
* MOUSE\_LAST – последний номер в списке идентификаторов, используемых для событий мыши.

Методы класса

* **public int getButton**()
* **public int getClickCount**()

позволяют определить соответственно, какая кнопка была нажата и количество щелчков кнопки.

Методы

* **public Point getPoint**()
* **public int getX**()
* **public int getY**()

определяет точку экрана, в которой произошло событие мыши или отдельно координаты x и y точки.

Метод **public boolean isPopupTrigger**() позволяет определить, является ли событие мыши событием вызова контекстного («всплывающего») меню (для Windows это нажатие правой кнопки мыши).

Метод **public static String getMouseModifiersText**(**int** *modifiers*) возвращает строку, идентифицирующую клавишу-модификатор (например, "Shift"), если она была нажата одновременно с кнопкой мыши.

Метод **public void translatePoint**(**int** *x*, **int** *y*) переводит точку координаты точки на экране, в которой произошло событие мыши, в новую позицию с горизонтальным смещением x и вертикальным смещением y.

Кроме указанных свойств и методов, класс **MouseEvent** наследует все свойства и методы класса **InputEvent**.

В интерфейсе **MouseListener** объявлены следующие методы:

* **public void mouseClicked**(**MouseEvent** *e*) – обработка события щелчка мыши;
* **public void mousePressed**(**MouseEvent** *e*) – обработка события нажатия кнопки мыши;
* **public void mouseReleased**(**MouseEvent** *e*) – обработка события отпускания кнопки мыши;
* **public void mouseEntered**(**MouseEvent** *e*) – обработка события входа курсора мыши в компонент;
* **public void mouseExited**(**MouseEvent** *e*) – обработка события выхода курсора мыши из компонента.

Интерфейс **MouseMotionListener** содержит объявления следующих методов:

* public void mouseDragged(MouseEvent e) – обработка события перемещения мыши при нажатой кнопке;
* public void mouseMoved(MouseEvent e) – обработка события перемещения мыши (при не нажатой кнопке мыши).

#### Класс MouseWheelEvent и интерфейс MouseWheelListener

Класс **MouseWheelEvent** обрабатывает события, связанные с вращением колесика мыши. Две статические final переменные типа int:

* WHEEL\_UNIT\_SCROLL
* WHEEL\_BLOCK\_SCROLL

представляют прокручивание колесика в «единицах» (как перемещение с помощью клавишей стрелок) и в «блоках» (как перемещение с помощью клавиш PageUp и PageDown).

Методы класса

* **public int getScrollType**()
* **public int getScrollAmount**()
* **public int getWheelRotation**()
* **public int getUnitsToScroll**()

позволяют получить соответственно тип вращения (равный одной из приведенных выше констант); количество «единиц», которое необходимо прокрутить в ответ на это событие; количество «щелчков», прокрученных колесиком мыши (отрицательное, если прокручивание вверх и положительное, если вниз) и количество «единиц» прокрутки (зависит от компьютерной платформы).

Интерфейс **MouseWheelListener** объявляет метод **public void mouseWheelMoved**(**MouseWheelEvent** *e*) для обработки события вращения колесика мыши.

### 1.3.3. Классы и интерфейсы обработки высокоуровневых событий

#### Интерфейс ActionListener и класс ActionEvent

Для классов **JButton**, **JList** и **JTextField** определен блок прослушивания **ActionListener** с единственным методом **public void actionPerformed**(**ActionEvent** *e*), вызываемым, когда происходит действие для заданного объекта класса **Button**, **List** или **TextField**.

Класс **ActionEvent** обрабатывает события, связанные с нажатием кнопки, выбором элемента из списка или с нажатием клавиши Enter в текстовом поле.

Для класса **ActionEvent** определены следующие статические final переменные типа int:

* ACTION\_PERFORMED – действие произошло;
* ALT\_MASK – нажата клавиша Alt;
* SHIFT\_MASK – нажата клавиша Shift;
* CTRL\_MASK – нажата клавиша Ctrl;
* ACTION\_FIRST – первый номер в списке идентификаторов, используемых для событий действия;
* ACTION\_LAST – последний номер в списке идентификаторов, используемых для событий действия.

Методы

* **public String getActionCommand**()
* **public int getModifiers**()
* **public long getWhen**()
* **public String paramString**()

позволяют получить соответственно командную строку, связанную с данным действием; ключ-модификатор для данного действия; время счетчика (в миллисекундах) для данного действия и строку параметра, идентифицирующего данное действие (обычно используется для отладки).

#### Интерфейсы ItemListener, ItemSelectable и класс ItemEvent

Для классов **JComboBox**, **JCheckbox**, **JRadioButton** и **JList** определен блок прослушивания **ItemListener** с единственным методом **public void itemStateChanged**(**ItemEvent** *e*), вызываемым, когда происходит выбор или отмена выбора элементов в объектах класса **Choice**, **Checkbox** или **List**.

Класс **ItemEvent** содержит переменные и методы для обработки события, связанные с выбором элемента или отменой выбора элемента.

Для класса **ItemEvent** определены следующие статические final переменные типа int:

* SELECTED – элемент выбран;
* DESELECTED – выборка элемента отменена;
* ITEM\_STATE\_CHANGED – состояние элемента изменено;
* ITEM\_FIRST – первый номер в списке идентификаторов, используемых для событий выбора элемента;
* ITEM\_LAST – последний номер в списке идентификаторов, используемых для событий выбора элемента.

Методы

* **public Object getItem**()
* **public int getStateChange**()
* **public String paramString**()

позволяют получить соответственно элемент, вызвавший событие; изменение состояния элемента (выбран или не выбран) и строку параметра, идентифицирующего выбор элемента (обычно используется для отладки).

Метод **public ItemSelectable getItemSelectable**() возвращает объект интерфейса **ItemSelectable**. Этот интерфейс предназначен для объектов, в которых допустим выбор одного или более элементов и содержит объявления методов

* **public void addItemListener**(**ItemListener** *l*)
* **public void removeItemListener**(**ItemListener** *l*)

для добавления и удаления блоков прослушивания выбора или отмены выбора элементов из списка, а также метод **public Object[] getSelectedObjects**(), который должен возвращать массив выбранных объектов или null, если не выбрано ни одного объекта.

### 1.3.4. Обработка событий в Java

Чтобы блок прослушивания мог фиксировать и обрабатывать события он должен быть включен с помощью методов добавления блоков прослушивания в класс (эти методы определены в классе Component).

Методы включения блоков прослушивания имеют следующий общий формат:

**public void addXXXX** (**XXXX** *l*)

где XXXX – имя соответствующего блока прослушивания, например включение в класс блока прослушивания для событий клавиатуры реализуется с помощью метода **public void addKeyListener** (**KeyListener** *l*).

Таким образом, для обработки события в каком либо классе необходимо:

1. Сделать доступным методы, определенные в соответствующем интерфейсе или интерфейсах, объявив класс как реализацию (implementation) соответствующего интерфейса или интерфейсов.
2. Включить необходимые блоки прослушивания с помощью соответствующего метода addXXXX.
3. Описать в программе все методы всех реализуемых интерфейсов (если какой-либо метод не используется в классе, он объявляется с пустым телом – {}). В методах интерфейсов обычно используются методы и свойства соответствующего класса обработки событий.

Скелет программы (для обработки событий клавиатуры) представлен ниже:

**class KeyTest implements KeyListener**

{

...

**addKeyListener**(имя-объекта-KeyListener);

...

**public void keyPressed**(**KeyEvent** *e*)

{

// Обработка события нажатия клавиши или пусто

}

**public void keyReleased**(**KeyEvent** *e*)

{

// Обработка события отпускания клавиши или пусто

}

**public void keyTyped**(**KeyEvent** *e*)

{

// Обработка ввода символа или пусто

}

Как видно из предыдущего примера, необходимо описывать метод, даже если он не используется. Для того, чтобы в классе можно было описывать только те методы, которые необходимо, в JDK были введены специальные абстрактные классы – адаптеры, содержащие объявления тех же методов, что и соответствующие блоки прослушивания. Имена этих классов формируются так же, как и блоков прослушивания, только вместо суффикса **Listener** в них используется суффикс **Adapter**, например, **KeyAdapter** или **MouseAdapter**.

Если создать внутри данного класса подкласс, расширяющий класс соответствующего адаптера (например, подкласс **myKeyAdapter**, расширяющий класс **KeyAdapter**), то внутри него можно переопределить только те методы адаптера, которые необходимо использовать. В этом случае в соответствующем методе addXXXX в качестве параметра можно задать экземпляр созданного подкласса, например,

**addKeyListener**(**new myKeyAdapter**());

Более компактной записью, которая часто используется программистами на Java, является непосредственное определение безымянного внутреннего класса в параметре соответствующего метода addXXXX, например:

**addKeyListener**

(

**new KeyAdapter**()

{

// Описание методов класса KeyAdapter

}

);

## 1.4. Визуальные средства разработки форм Swing в IntelliJ IDEA

### 1.4.1. Особенности работы стандартного плагина

В дистрибутив IntelliJ IDEA входит стандартный плагин для разработки форм **UI Designer**, который обладает рядом особенностей:

* При добавлении на форму элементов интерфейса данная IDE автоматически генерирует только поля класса. Для того, чтобы были сгенерированы методы создания интерфейса, необходимо собрать проект, так как эти методы генерируются только при сборке.
* Для того, чтобы форма и элементы интерфейса на ней могли корректно отображаться на экране, необходимо доработать сгенерированный класс формы: (а) Сделать его наследником класса **JFrame**. (б) Создать метод, который будет выводить форму на экран.
* Для того, чтобы иметь возможность обрабатывать событие в форме, следует самостоятельно создать класс слушателя с реализацией обработчика события и в файле формы подключить слушателя к компоненту формы.

Все вышеописанные нюансы – следствие издержек плагина UI Designer. Другие плагины для создания графического интерфейса лишены этих недостатков. Например, JFormDesigner. Но, к сожалению, многие из них платные. На данный момент, стоимость некоммерческой лицензии на тот же JFormDesigner составляет от 79$, коммерческой от 159$ в расчёте на одного пользователя.

NetBeans имеет в своем составе довольно качественный редактор форм.

### 1.4.2. Пример простого оконного приложения

Создадим простейшее оконное приложение, содержащее одну кнопку, щелчок по которой вызывает диалог приветствия. Для этого:

1. Создаем новый проект: **File - New Project…**
2. В папке **src** создадим окно формы: **New - GUI Form** и назовем ее ***GUIHello***.
3. В окне дизайнера (файл ***GUIHello.form***) перетаскиваем объект **JButton** на форму.
4. Отредактируйте класс ***GUIHello*** как указано в листинге.

Листинг класса ***GUIHello***:

import javax.swing.\*;

import java.awt.BorderLayout;

import java.awt.HeadlessException;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

public class GUIHello extends JFrame implements ActionListener {

private JPanel panel1;

private JButton clickButton;

public GUIHello() throws HeadlessException {

super("GUI Hello");

add(panel1);

pack();

setSize(300, 300);

// this.getContentPane().add(panel1);

setVisible(true);

clickButton.addActionListener(this);

}

@Override

public void actionPerformed(ActionEvent e) {

System.out.println("hello");

JOptionPane.showMessageDialog(null,"Hello");

}

}

# 2. Варианты заданий

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **№** | **Задачи** | **№** | **Задачи** | **№** | **Задачи** | **№** | **Задачи** |
| **1** | 1,5,16,30 | **7** | 8,13,18,22 | 13 | 2,14,21,28 | 19 | 8,15,22,24 |
| **2** | 2,13,17,29 | **8** | 9,14,23,30 | 14 | 3,5,19,29 | 20 | 9,13,21,25 |
| **3** | 3,15,18,28 | **9** | 5,10,24,29 | 15 | 4,11,26,30 | 21 | 10,14,20,26 |
| **4** | 4,14,19,27 | **10** | 11,15,25,28 | 16 | 5,13,20,25 | 22 | 5,11,19,27 |
| **5** | 5,6,20,26 | **11** | 5,12,14,26 | 17 | 5,7,12,22 | 23 | 12,15,18,28 |
| **6** | 7,15,17,21 | **12** | 1,13,20,27 | 18 | 14,6,23,29 | 24 | 1,13,17,29 |

1. Ввод текста в графическое окно приложения. В окне определена строка меню (JMenuBar), в которой определены два меню (JMenu) – "Шрифт" и "Стиль". В меню "Шрифт" определены три пункта меню (JRadioButtonMenuItem): "Times New Roman" (шрифт по умолчанию), "Arial" и "Verdana". В меню "Стиль" определены четыре пункта меню (JRadioButtonMenuItem): "Простой" (шрифт по умолчанию), "Жирный", "Курсив" и "Жирный курсив". В текстовой панели (JTextPane) "Ввод текста" окна приложения вводится текст, набираемый на клавиатуре. При выборе одного из пунктов меню текст в панели выводится соответствующим шрифтом и/или стилем.
2. Ввод текста в графическое окно приложения. В верхней панели (JPanel) окна "Характеристики шрифта" определены следующие компоненты: надпись (JLabel) "Шрифт:", раскрывающееся меню (JComboBox), в котором три пункта: "Times New Roman" (шрифт по умолчанию), "Arial" и "Verdana", надпись (JLabel) "Стиль:" и раскрывающееся меню (JComboBox), в котором четыре пункта: "Простой" (шрифт по умолчанию), "Жирный", "Курсив" и "Жирный курсив". В нижней текстовой панели (JTextPane) "Ввод текста" окна приложения вводится текст, набираемый на клавиатуре. При выборе одного из пунктов меню текст в панели выводится соответствующим шрифтом и/или соответствующего стиля.
3. Изменение размера фигуры в графическом окне. В окне определена строка меню (JMenuBar), в которой определены меню (JMenu) "Параметр" и "Изменение". В меню "Параметр" определены два пункта меню (JCheckBoxMenuItem): "Ширина" (по умолчанию включена) и "Высота" (по умолчанию включена). В меню "Изменение" определены два пункта меню (JMenuItem): "Увеличить" и "Уменьшить". В центре панели (JPanel) окна "Вывод изображения" в графическом контексте задается (с помощью метода drawRect()) прямоугольник. При выборе одного из пунктов второго меню фигура скачкообразно (на 10 пикселей) увеличивается или уменьшается для заданного параметра или параметров в первом меню и перерисовывается. Если в первом меню не включен ни один из пунктов, фигура не меняется.
4. Изменение размера фигуры в графическом окне. В центре верхней панели (JPanel) "Вывод изображения" в графическом контексте задается (с помощью метода drawRect()) прямоугольник. В нижней панели (JPanel) "Изменение размера" задается надпись (JLabel) "Параметр:", два переключателя (JCheckBox): "Ширина:" (оба включены по умолчанию) и "Высота:", а также две кнопки (JButton): "Увеличить" и "Уменьшить". При выборе одной из радиокнопок и нажатии одной из двух кнопок фигура скачкообразно (на 10 пикселей) увеличивается или уменьшается для заданного параметра и перерисовывается. Если оба переключателя выключены, фигура не меняется.
5. Вывод строки заданным шрифтом и заданного цвета в графическом окне. В верхней панели (JPanel) "Управление выводом" задается надпись (JLabel) "Текст:" и текстовое поле (JTextField), надпись (JLabel) "Гарнитура:", вращающийся список (JSpinner) со значениями "Times New Roman" (шрифт по умолчанию), "Arial" и "Verdana", надпись (JLabel) "Цвет:" и вращающийся список (JSpinner) со значениями "Черный" (цвет по умолчанию), "Красный", "Зеленый" и "Синий", а также кнопка (JButton) "Вывести строку". В нижней панели (JPanel) "Вывод строки" выводится в графическом контексте (с помощью метода drawString()) в произвольном месте пустая строка. При задании текста строки в текстовом поле, параметров строки во вращающихся списках верхней панели и нажатии кнопки "Вывод строки" строка заданного содержимого и цвета выводится заданным шрифтом в нижней панели.
6. Вывод строки заданным шрифтом и заданного цвета в графическом окне. В верхней панели (JPanel) "Управление выводом" задается надпись (JLabel) "Строка:" и текстовое поле (JTextField), надпись (JLabel) "Гарнитура:", раскрывающееся меню (JComboBox), в котором три пункта: "Times New Roman" (шрифт по умолчанию), "Arial" и "Verdana", надпись (JLabel) "Цвет:" и раскрывающееся меню (JComboBox), в котором четыре пункта: "Черный" (цвет по умолчанию), "Красный", "Зеленый" и "Синий", а также кнопка (JButton) "Вывести строку". В нижней панели (JPanel) "Вывод строки" выводится в графическом контексте (с помощью метода drawString()) в произвольном месте пустая строка. При задании текста строки в текстовом поле, параметров строки в раскрывающихся меню верхней панели и нажатии кнопки "Вывод строки" строка заданного содержимого и цвета выводится заданным шрифтом в нижней панели.
7. Вывод фигур в графическом окне. В окне определена строка меню (JMenuBar), в которой определены два меню (JMenu) – "Цвет и "Вывод". В меню "Цвет" определены четыре пункта меню (JRadioButtonMenuItem): "Черный" (цвет по умолчанию), "Красный", "Зеленый" и "Синий". В меню "Вывод" определены три пункта меню (JCheckBoxMenuItem): "Квадрат", "Круг", и "Треугольник". В панели (JPanel) окна выводятся в графическом контексте (с помощью методов drawRect(), drawOval() и drawPolygon()) три фигуры: прямоугольник, круг и треугольник. При выборе пункта первого меню все фигуры заполняются заданным цветом. Команды второго меню отключают или включают вывод соответствующей фигуры на экран.
8. Вывод фигур в графическом окне. В окне определена строка меню (JMenuBar), в которой определены два меню (JMenu) – "Цвет и "Вывод". В меню "Цвет" определены четыре пункта меню (JRadioButtonMenuItem): "Черный" (цвет по умолчанию), "Красный", "Зеленый" и "Синий". В меню "Вывод" определены три пункта меню (JCheckBoxMenuItem): "Квадрат", "Круг", и "Треугольник". В верней панели окна панели (JPanel) окна выводятся в графическом контексте (с помощью методов drawRect(), drawOval() и drawPolygon()) три фигуры: прямоугольник, круг и треугольник. В нижней панели (JPanel) "Управление выводом" задается надпись (JLabel) "Цвет:", раскрывающееся меню (JComboBox), в котором определены четыре пункта: "Черный" (цвет по умолчанию), "Красный", "Зеленый" и "Синий", надпись (JLabel) "Вывод:" и три переключателя (JCheckBox): "Квадрат", "Круг", и "Треугольник". При выборе пункта раскрывающегося меню все фигуры заполняются заданным цветом. На экран выводятся только те фигуры, переключатели которых включены.
9. Изменение размера изображения в графическом окне. В верхней панели окна (JPanel) окна "Вывод изображения" в компоненте (JLabel) задается произвольное изображение. В нижней панели (JPanel) "Управление выводом" задаются две кнопки (JButton): "Увеличить" и "Уменьшить". При нажатии первой кнопки изображение увеличивается на 10 пикселей по ширине и высоте, при нажатии второй кнопки – уменьшается на то же количество пикселей.
10. Изменение размера изображения в графическом окне. В панели окна (JPanel) окна "Вывод изображения" в компоненте (JLabel) задается произвольное изображение. В панели инструментов окна задаются две кнопки (JButton): "Увеличить" и "Уменьшить". При нажатии первой кнопки изображение увеличивается на 10 пикселей по ширине и высоте, при нажатии второй кнопки – уменьшается на то же количество пикселей.
11. Перемещение фигуры в графическом окне. В центре верхней панели (JPanel) "Вывод изображения" в графическом контексте задается (с помощью метода drawRect()) прямоугольник. В нижней панели (JPanel) "Перемещение изображения" задаются следующие компоненты: надпись (JLabel) "Цвет:", раскрывающееся меню (JComboBox), в котором определены четыре пункта: "Черный" (цвет по умолчанию), "Красный", "Зеленый" и "Синий", а также четыре кнопки (JButton): "Влево", "Вверх" "Вправо" и "Вниз". При выборе цвета прямоугольник перерисовывается заданным цветом, а при нажатии одной из четырех кнопок изображение скачкообразно (на 10 пикселей) перемещается в этом направлении и перерисовывается.
12. Перемещение фигуры в графическом окне. В центре верхней панели (JPanel) "Вывод изображения" в графическом контексте задается (с помощью метода drawRect()) прямоугольник. В нижней панели (JPanel) "Перемещение изображения" задаются следующие компоненты: надпись (JLabel) "Цвет:", вращающийся список (JSpinner) со значениями "Черный" (цвет по умолчанию), "Красный", "Зеленый" и "Синий", а также четыре переключателя (JRadioButton): "Влево", "Вверх" "Вправо" и "Вниз" и кнопка (JButton) "Переместить". При выборе цвета прямоугольник перерисовывается заданным цветом, а при нажатии одной из четырех радиокнопок изображение скачкообразно (на 10 пикселей) перемещается в этом направлении и перерисовывается.
13. Ввод текста в графическое окно приложения. В верхней панели (JPanel) окна "Характеристики шрифта" определены следующие компоненты: надпись (JLabel) "Цвет:", раскрывающееся меню (JComboBox), в котором четыре пункта: "Черный" (цвет по умолчанию), "Красный", "Зеленый" и "Синий", надпись (JLabel) "Размер:" и три радиокнопки (JRadioButton) с надписями: "12pt" (размер по умолчанию), "14pt" и "16pt". В нижней текстовой панели (JTextPane) "Ввод текста" окна приложения вводится текст, набираемый на клавиатуре. При выборе одного из пунктов меню текст в панели выводится соответствующим цветом и/или соответствующего размера.
14. Ввод текста в графическое окно приложения. В панели инструментов окна (JToolBar) определены следующие компоненты (JButton) с надписями: "Черный" (цвет по умолчанию), "Красный", "Зеленый" и "Синий", "12pt" (размер по умолчанию), "14pt" и "16pt". В текстовой панели (JTextPane) "Ввод текста" окна приложения вводится текст, набираемый на клавиатуре. Надписи в кнопках должны быть выполнены соответствующим цветом и шрифтом. При выборе на панели инструментов цвета и/или размера текст в панели выводится соответствующим цветом и/или соответствующего размера.
15. Изменение фигуры в графическом окне. В центре верхней панели (JPanel) "Вывод прямоугольника" в графическом контексте рисуется (с помощью метода drawRoundRect()) прямоугольник со скругленными вершинами. В нижней панели (JPanel) "Параметры прямоугольника" задаются следующие компоненты: надпись (JLabel) "Цвет:", вращающийся список (JSpinner) со значениями: "Черный" (цвет по умолчанию), "Красный", "Зеленый" и "Синий", надпись (JLabel) "Ширина закругления", текстовое поле (JTextField), надпись (JLabel) "Высота закругления:", текстовое поле (JTextField) и кнопка (JButton) "Изменить прямоугольник". При наборе данных во вращающихся списках, вводе данных в текстовых полях (в пикселях) и при нажатии кнопки "Изменить прямоугольник" прямоугольник перерисовывается заданным цветом и с заданным новыми значениями ширины и высоты закруглений. Перед выводом фигуры выполняется проверка, введены ли в текстовых полях все данные и являются ли они целыми числами.
16. Изменение фигуры в графическом окне. В центре верхней панели (JPanel) "Вывод прямоугольника" в графическом контексте рисуется (с помощью метода drawRoundRect()) прямоугольник со скругленными вершинами. В нижней панели (JPanel) "Параметры прямоугольника" задаются следующие компоненты: надпись (JLabel) "Цвет:", раскрывающее меню (JComboBox) со значениями: "Черный" (цвет по умолчанию), "Красный", "Зеленый" и "Синий", надпись (JLabel) "Ширина закругления", вращающийся список (JSpinner), надпись (JLabel) "Высота закругления:", вращающийся список (JSpinner) и кнопка (JButton) "Изменить прямоугольник". При наборе данных во вращающихся списках, вводе данных в текстовых полях (в пикселях) и при нажатии кнопки "Изменить прямоугольник" прямоугольник перерисовывается заданным цветом и с заданным новыми значениями ширины и высоты закруглений. Диапазон изменения вращающихся списков – от 0 до 30 с шагом 1.
17. Рисование фигуры в графическом окне. В верхней панели (JPanel) "Вывод дуги" в графическом контексте рисуется (с помощью метода drawArc()) дуга. В нижней панели (JPanel) "Параметры дуги" задаются следующие компоненты: надпись (JLabel) "Координата X:", текстовое поле (JTextField), надпись (JLabel) "Координата Y:", текстовое поле (JTextField), надпись (JLabel) "Ширина:", текстовое поле (JTextField), надпись (JLabel) "Высота:", текстовое поле (JTextField), надпись (JLabel) "Начальный угол:", текстовое поле (JTextField), надпись (JLabel) "Конечный угол:", текстовое поле (JTextField) и кнопка (JButton) "Вывести дугу". При вводе данных (в пикселях и в градусах – для углов) и при нажатии кнопки "Вывести дугу" дуга перерисовывается с заданными параметрами. При вводе данных проверяется, все ли данные введены и являются ли данные целыми числами. Значения углов должны являться положительными или отрицательными целыми числами в диапазоне (по абсолютной величине) от 0 до 360.
18. Рисование фигуры в графическом окне. В верхней панели (JPanel) "Вывод дуги" в графическом контексте рисуется (с помощью метода drawArc()) дуга. В нижней панели (JPanel) "Параметры дуги" задаются следующие компоненты: надпись (JLabel) "Координата X:", текстовое поле (JTextField), надпись (JLabel) "Координата Y:", текстовое поле (JTextField), надпись (JLabel) "Ширина:", текстовое поле (JTextField), надпись (JLabel) "Высота:", текстовое поле (JTextField), надпись (JLabel) "Начальный угол:", две радиокнопки "Плюс" и "Минус", вращающийся список (JSpinner), надпись (JLabel) "Конечный угол:", две радиокнопки "Плюс" и "Минус", вращающийся список (JSpinner) и кнопка (JButton) "Вывести дугу". При вводе данных для координат и размера (в пикселях), набора данных для углов и при нажатии кнопки "Вывести дугу" дуга перерисовывается с заданными параметрами. При вводе данных проверяется, все ли данные введены и являются ли данные целыми числами. Значения данных во вращающемся списке – в диапазоне от 0 до 360 с шагом 45.
19. Изменение фигуры в графическом окне. В центре верхней панели (JPanel) "Вывод треугольника" в графическом контексте рисуется (с помощью метода drawPolygon()) треугольник. В нижней панели (JPanel) "Параметры треугольника" задаются следующие компоненты: надпись (JLabel) "Цвет:", раскрывающееся меню (JComboBox) с пунктами: "Черный" (цвет по умолчанию), "Красный", "Зеленый" и "Синий", надпись (JLabel) "Номер вершины:", раскрывающееся меню (JComboBox) с пунктами "1", "2" и "3", надпись (JLabel) "Координата X:", вращающийся список (JSpinner), надпись (JLabel) "Координата Y:", вращающийся список (JSpinner) и кнопка (JButton) "Изменить треугольник". При наборе данных во вращающихся списках (в пикселях) и при нажатии кнопки "Изменить треугольник" треугольник перерисовывается заданным цветом и с заданным новым положением одной из вершин. Диапазон изменения вращающихся списков – от 0 до 150 с шагом 1.
20. Изменение фигуры в графическом окне. В центре верхней панели (JPanel) "Вывод треугольника" в графическом контексте рисуется (с помощью метода drawPolygon()) треугольник. В нижней панели (JPanel) "Параметры треугольника" задаются следующие компоненты: надпись (JLabel) "Цвет:", вращающийся список (JSpinner) со значениями: "Черный" (цвет по умолчанию), "Красный", "Зеленый" и "Синий", надпись (JLabel) "Номер вершины:", вращающийся список (JSpinner) со значениями "1", "2" и "3", надпись (JLabel) "Координата X:", текстовое поле (JTextField), надпись (JLabel) "Координата Y:", текстовое поле (JTextField) и кнопка (JButton) "Изменить треугольник". При наборе данных во вращающихся списках (в пикселях), вводе данных в текстовых полях (в пикселях) и при нажатии кнопки "Изменить треугольник" треугольник перерисовывается заданным цветом и с заданным новым положением одной из вершин. Перед выводом фигуры выполняется проверка, введены ли в текстовых полях все данные и являются ли они целыми числами.
21. Изменение надписи в графическом окне. В центре верхней панели (JPanel) "Вывод надписи" в графическом контексте выводится (с помощью метода drawString()) произвольная надпись. В нижней панели (JPanel) "Параметры надписи" задаются следующие компоненты: надпись (JLabel) "Текст:", текстовое поле (JTextField), надпись (JLabel) "Цвет:", вращающийся список (JSpinner) со значениями: "Черный" (цвет по умолчанию), "Красный", "Зеленый" и "Синий", надпись (JLabel) "Размер:", вращающийся список (JSpinner) со значениями "10pt", "12pt" (размер по умолчанию), "14pt" и "16pt", надпись (JLabel) "Координата X:", текстовое поле (JTextField), надпись (JLabel) "Координата Y:", текстовое поле (JTextField), а также кнопка (JButton) "Вывести строку". Первоначально в текстовых полях "Координата X:" и "Координата Y:" устанавливаются координаты начала базовой линии надписи (текстовые поля заданы как нередактируемые). При вводе данных надпись меняет текст, цвет и/или размер. При щелчке мышью в области верхней панели в текстовых полях выводятся координаты точки щелчка, и надпись перерисовывается в этой точке.
22. Изменение надписи в графическом окне. В центре верхней панели (JPanel) "Вывод надписи" в графическом контексте выводится (с помощью метода drawString()) произвольная надпись. В нижней панели (JPanel) "Параметры надписи" задаются следующие компоненты: надпись (JLabel) "Текст:", текстовое поле (JTextField), надпись (JLabel) "Цвет:", четыре радиокнопки (JRadioButton) со значениями: "Черный" (цвет по умолчанию), "Красный", "Зеленый" и "Синий", вращающийся список (JLabel) "Размер:", три радиокнопки (JRadioButton) со значениями: "10pt", "12pt" (размер по умолчанию) и "14pt", надпись (JLabel) "Координата X:", текстовое поле (JTextField), надпись (JLabel) "Координата Y:", текстовое поле (JTextField), а также кнопка (JButton) "Вывести строку". Первоначально в текстовых полях "Координата X:" и "Координата Y:" устанавливаются координаты начала базовой линии надписи (текстовые поля заданы как нередактируемые). При вводе данных надпись меняет текст, цвет и/или размер. При щелчке мышью в области верхней панели в текстовых полях выводятся координаты точки щелчка и надпись перерисовывается в этой точке.
23. Рисование фигуры в графическом окне. В центре верхней панели (JPanel) "Вывод эллипса" в графическом контексте рисуется (с помощью метода drawOval()) эллипс. В нижней панели (JPanel) "Параметры эллипса" задаются следующие компоненты: надпись (JLabel) "Цвет:", четыре радиокнопки (JRadioButton) с надписями: "Черный" (цвет по умолчанию), "Красный", "Зеленый" и "Синий", надпись (JLabel) "Координата X центра:", текстовое поле (JTextField), надпись (JLabel) "Координата Y центра:", текстовое поле (JTextField), надпись (JLabel) "Полуось a:", текстовое поле (JTextField), надпись (JLabel) "Полуось b:", текстовое поле (JTextField) и кнопка (JButton) "Вывести эллипс". При вводе данных в текстовых полях (в пикселях) и при нажатии кнопки "Вывести эллипс" в верхней панели рисуется эллипс. Перед выводом фигуры выполняется проверка, введены ли все данные и являются ли они числами. Для вызова метода drawOval() выполняются следующие преобразования: координата X левого верхнего угла вычисляется как координата X центра минус значение a, координата Y левого верхнего угла вычисляется как координата Y центра минус значение b, ширина – как удвоенное значение a, а высота – как удвоенное значение b.
24. Рисование фигуры в графическом окне. В центре верхней панели (JPanel) "Вывод эллипса" в графическом контексте рисуется (с помощью метода drawOval()) эллипс. В нижней панели (JPanel) "Параметры эллипса" задаются следующие компоненты: надпись (JLabel) "Цвет:", раскрывающееся меню (JComboBox) с пунктами: "Черный" (цвет по умолчанию), "Красный", "Зеленый" и "Синий", надпись (JLabel) "Координата X центра:", вращающийся список (JSpinner), надпись (JLabel) "Координата Y центра:", вращающийся список (JSpinner), надпись (JLabel) "Полуось a:", вращающийся список (JSpinner), надпись (JLabel) "Полуось b:", вращающийся список (JSpinner) и кнопка (JButton) "Вывести эллипс". При наборе данных во вращающихся списках (в пикселях) и при нажатии кнопки "Вывести эллипс" в верхней панели рисуется эллипс. Диапазон изменения вращающихся списков – от 10 до 150 с шагом 1. Для вызова метода drawOval() выполняются следующие преобразования: координата X левого верхнего угла вычисляется как координата X центра минус значение a, координата Y левого верхнего угла вычисляется как координата Y центра минус значение b, ширина – как удвоенное значение a, а высота – как удвоенное значение b.
25. Ввод текста в графическое окно приложения. В верхней панели окна (JPanel) "Параметры шрифта" заданы следующие компоненты: надпись (JLabel) "Цвет:", четыре радиокнопки (JRadioButton) с надписями: "Черный" (цвет по умолчанию), "Красный", "Зеленый" и "Синий", надпись (JLabel) "Размер:" и текстовое поле (JTextField) (начальное значение этого текстового поля равно 12). В текстовой панели (JTextPane) "Ввод текста" окна приложения вводится текст, набираемый на клавиатуре. Надписи для радиокнопок должны быть выполнены соответствующим цветом. При выборе одной из радиокнопок цвета и ввода в текстовом поле целого числа – размера текста содержимое текстовой панели выводится соответствующим цветом и/или соответствующего размера. При вводе значения в текстовое поле проверяется, является ли это значение целым числом, содержащим одну или две цифры.
26. Ввод текста в графическое окно приложения. В верхней панели окна (JPanel) "Параметры шрифта" заданы следующие компоненты: надпись (JLabel) "Стиль:", раскрывающееся меню (JComboBox) с пунктами: "Простой" (стиль по умолчанию), "Жирный" и "Курсив", надпись (JLabel) "Размер:" и раскрывающееся меню (JComboBox) с пунктами: "10pt" (стиль по умолчанию), "12pt" (размер по умолчанию) и "14pt". В текстовой панели (JTextPane) "Ввод текста" окна приложения вводится текст, набираемый на клавиатуре. При выборе одного из пунктов раскрывающегося меню стиля и ввода в текстовом поле целого числа – размера текста содержимое текстовой панели выводится соответствующим цветом и/или соответствующего размера. При вводе значения в текстовое поле проверяется, является ли это значение целым числом, содержащим одну или две цифры.
27. Рисование фигуры в графическом окне. В верхней панели окна (JPanel) "Координаты вершин треугольника" заданы следующие компоненты: надпись (JLabel) "X1:", текстовое поле (JTextField), надпись (JLabel) "Y1:", текстовое поле (JTextField), надпись (JLabel) "X2:", текстовое поле (JTextField), надпись (JLabel) "Y2:", текстовое поле (JTextField), надпись (JLabel) "X3:", текстовое поле (JTextField), надпись (JLabel) "Y3:", текстовое поле (JTextField) и кнопка (JButton) "Вывести треугольник". В нижней панели окна (JPanel) "Вывод треугольника" в графическом контексте рисуется (с помощью метода fillPolygon()) треугольник, заполненный черным цветом. Рисование треугольника выполняется после введения координат вершин при нажатии кнопки "Вывести треугольник". Перед выводом треугольника проверяется задание всех координат, которые должны быть положительными целыми числами.
28. Рисование фигуры в графическом окне. В верхней панели окна (JPanel) "Координаты вершин треугольника" заданы следующие компоненты: надпись (JLabel) "X1:", вращающийся список (JSpinner), надпись (JLabel) "Y1:", вращающийся список (JSpinner), надпись (JLabel) "X2:", вращающийся список (JSpinner), надпись (JLabel) "Y2:", вращающийся список (JSpinner), надпись (JLabel) "X3:", вращающийся список (JSpinner), надпись (JLabel) "Y3:", вращающийся список (JSpinner) и кнопка (JButton) "Вывести треугольник". В нижней панели окна (JPanel) "Вывод треугольника" в графическом контексте рисуется (с помощью метода fillPolygon()) треугольник, заполненный черным цветом. Рисование треугольника выполняется после введения координат вершин при нажатии кнопки "Вывести треугольник". Координаты вершин должны быть в диапазоне от 0 до 100.
29. Изменение размера изображения в графическом окне. В верхней панели окна (JPanel) "Размер изображения" заданы следующие компоненты: надпись (JLabel) "Ширина:", текстовое поле (JTextField), надпись (JLabel) "Высота:", текстовое поле (JTextField) и кнопка (JButton): "Вывести изображение". В нижней панели окна (JPanel) окна "Вывод изображения" в компоненте (JLabel) задается произвольное изображение. При вводе размера изображения по ширине и высоте и нажатии кнопки "Вывести изображение" в нижней панели выводится масштабированное изображение с заданными размерами (в пикселях). Первоначально изображение имеет «естественный» размер. Перед выводом изображения выполняется проверка, заданы ли значения ширины и высоты и являются ли они целыми числами.
30. Изменение размера изображения в графическом окне. В верхней панели окна (JPanel) "Размер изображения" заданы следующие компоненты: надпись (JLabel) "Ширина:", бегунок (JSlider), надпись (JLabel) "Высота:", бегунок (JSlider) и кнопка (JButton): "Вывести изображение". В нижней панели окна (JPanel) окна "Вывод изображения" в компоненте (JLabel) задается произвольное изображение. При установке бегунков размера изображения по ширине и высоте и нажатии кнопки "Вывести изображение" в нижней панели выводится масштабированное изображение с заданными размерами (в пикселях). Первоначально изображение имеет «естественный» размер. Нижние и верхние границы бегунков должны иметь значения 10 и 150.

# 3. Ход выполнения работы

1. Создайте проект **Java**.
2. Добавьте в проект форму и разместите на ней компоненты в соответствии с вариантом индивидуального задания.
3. Отредактируйте файл класса формы, обеспечив функциональность программы в соответствии с вариантом индивидуального задания.
4. Добавьте unit-тесты для тестирования методов класса, воспользовавшись командой **Code-Generate…-Test...**, и выполните их тестирование.
5. Если обнаружены методы, не прошедшие тест, внесите необходимые изменения в код и повторите тест.
6. Вставьте необходимые комментарии и сгенерируйте справку Javadoc.