Прикладное программирование

Лекция №7.

- Организация графического интерфейса (GUI) в Java приложениях
- Компоненты графического интерфейса в Java приложениях
- Выводы графики

Диаграмма консольного приложения табулирования функции

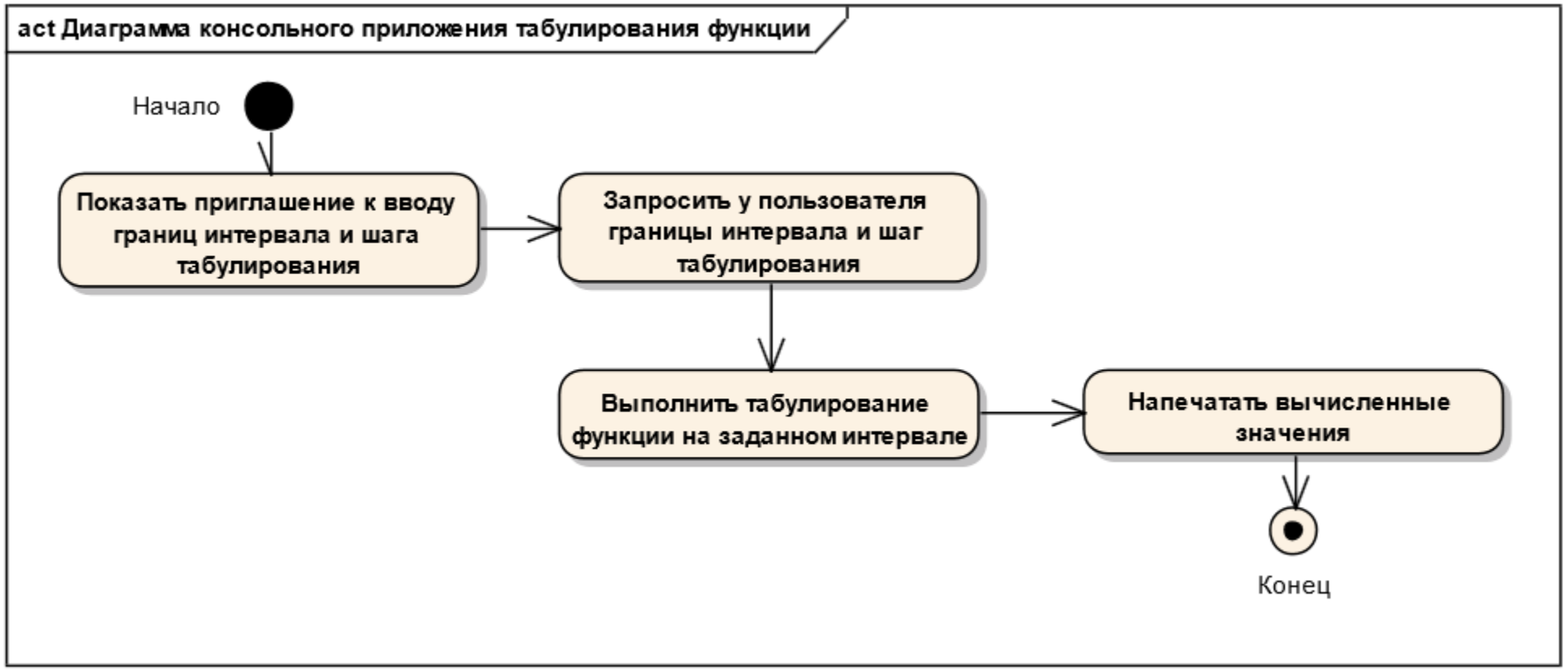
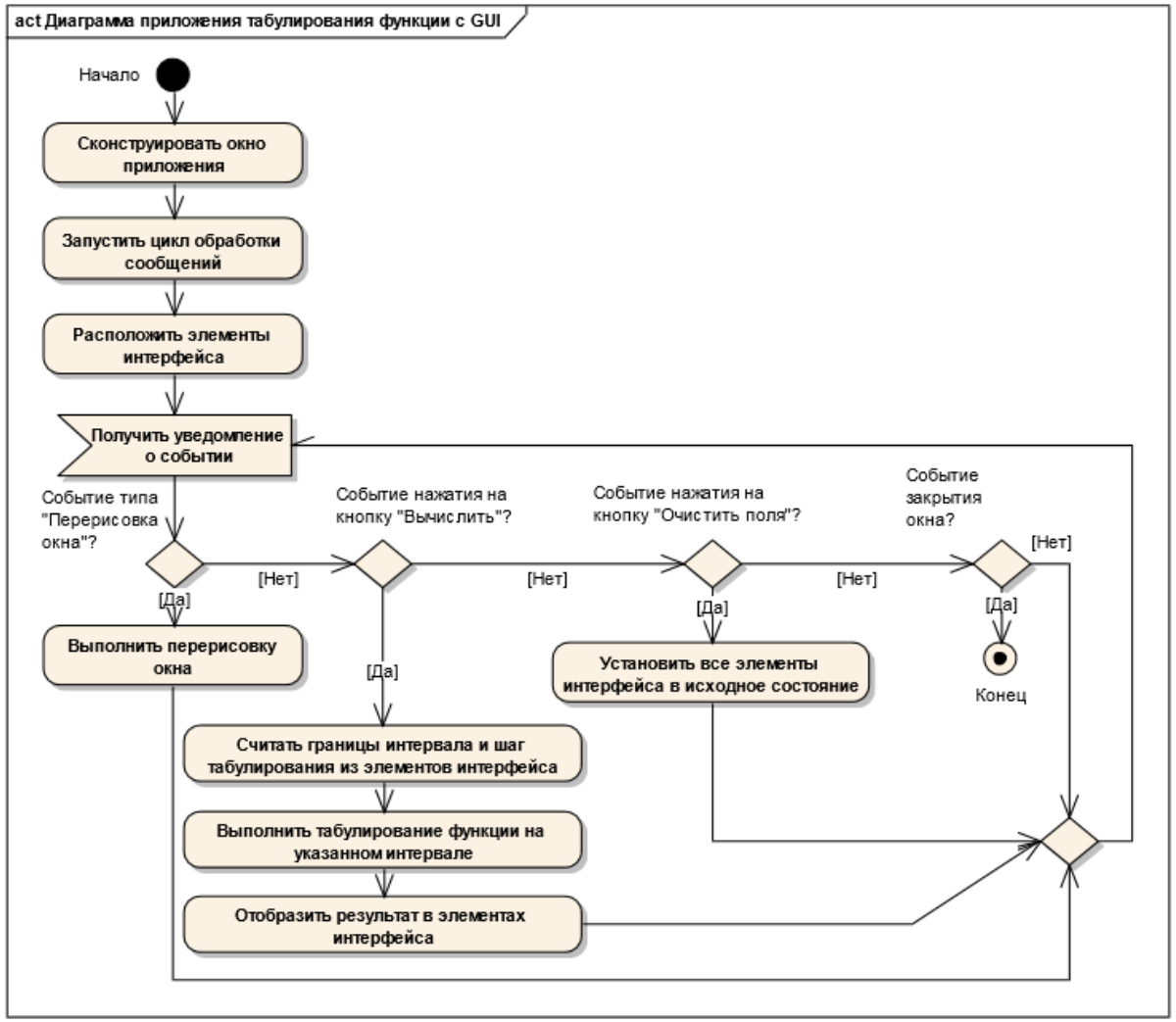


Диаграмма приложения табулирования функции с GUI



Отличие консольного приложения и приложения с GUI

В отличие от простейшего консольного **однопоточного** приложения, простейшее приложение с графическим интерфейсом обладает как минимум **двумя потоками**.

В первом потоке запускается главный метод **main()** главного класса приложения, который:

1. создает экземпляр окна;
2. задает реакцию на закрытие окна приложения (завершение);
3. отображает окно на экране.

При этом создается **второй поток** для обработки поступающих сообщений о событиях, **main()** завершается, но приложение продолжает функционировать.

Во втором потоке в цикле анализируется очередь поступивших событий, для которых вызываются соответствующие обработчики.

Окно/Frame – основа приложения с GUI

В отличие от консольного приложения, приложение с графическим (оконным) интерфейсом обладает одним или несколькими окнами верхнего уровня (т.е. окнами, которые **не являются вложенными в другие окна**), называемыми в Java **фреймами**.

Фрейм содержит обязательные компоненты, такие как:

- Полоса заголовка, содержащая название окна;
- Пиктограмма для системного меню;
- Набор кнопок управления состоянием окна (минимизации, максимизации, закрытия и т.п.).

Окно/Frame – основа приложения с GUI

Фрейм является контейнером, а значит может содержать вложенные элементы графического интерфейса пользователя:

- Надписи,
- Поля ввода,
- Выпадающие списки,
- Полосы прокрутки,
- Меню,
- Все элементы, которые позволяют организовать взаимодействие с пользователем, **реагируя на его действия** – нажатие клавиш на клавиатуре, движения и щелчки мышью.

Библиотеки компонентов GUI

С выходом **Java 1.0** для разработки GUI была предложена библиотека классов **«абстрактного оконного инструментария»** (**AWT**), основанная на **делегировании механизмов создания и поведения компонентов встроенному инструментарию GUI ОС.**

Это должно было позволить создавать переносимые приложения, внешний вид которых определялся той ОС, в которой работало приложение.

Тем не менее, отличия в поведении одних и тех же компонентов в различных ОС и серьезная разница в возможностях и количестве доступных компонентов на различных платформах существенно ограничили полезность и применимость данной библиотеки.

Библиотеки компонентов GUI

Альтернативным подходом, предложенным Netscape в 1996 году, стала библиотека основных классов **Интернет IFC (Internet Foundation Classes)**, основанная на **принципе отображения всех компонентов интерфейса в пространстве пустых окон**.

Единственным требованием к использованию **IFC** было наличие механизмов создания окна и рисования на его поверхности.

При этом компоненты **IFC** выглядели и вели себя одинаково, не завися от платформы, на которой запущено приложение.

В дальнейшем, совместно с компанией Sun, данный подход был усовершенствован, и результатом сотрудничества стала **библиотека GUI компонентов,** названная **Swing**.

Представление фреймов в библиотеках AWT и Swing

Для представления фреймов в библиотеке **AWT** существует класс **Frame**.

Так как классы **AWT** тесно связаны с встроенными компонентами ОС, то внешний вид окна (рамка, заголовок, иконки, управляющие кнопки) будет зависеть от ОС, в которой работает приложение.

В библиотеке **Swing** для представления фрейма существует класс **JFrame**. Так как **JFrame** является потомком **Frame**, то его внешний вид также отображается оконной системой ОС, а не **Swing**.

Подробнее о Swing

```
1  package hello;
2
3  import java.awt.GridLayout;
11
12 public class HelloSwingGUI {
13
14     private static void createAndShowGUI() {
15         JFrame frame = new JFrame("HelloWorldSwing");
16         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
17         frame.setSize(400,400);
18         frame.setLayout(new GridLayout(3, 1));
19
20         frame.getContentPane().add(new JLabel("Hello World"));
21         JButton button = new JButton("Close");
22         frame.getContentPane().add(button);
23         frame.getContentPane().add(new JLabel("Bottom Label"));
24
25         button.addActionListener(new ActionListener() {
26             public void actionPerformed(ActionEvent e) {
27                 System.exit(0);
28             }
29          });
30         frame.setVisible(true);
31     }
32
33     public static void main(String[] args) {
34         SwingUtilities.invokeLater(new Runnable() {
35             public void run() {
36                 createAndShowGUI();
37             }
38         });
39     }
40 }
41
```

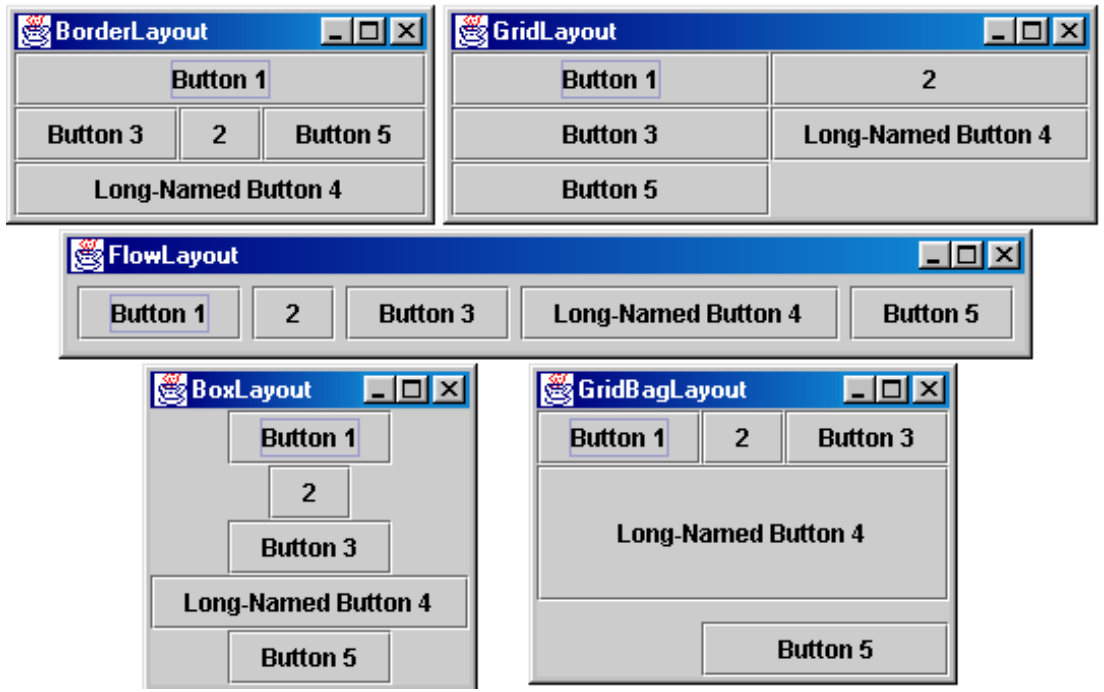
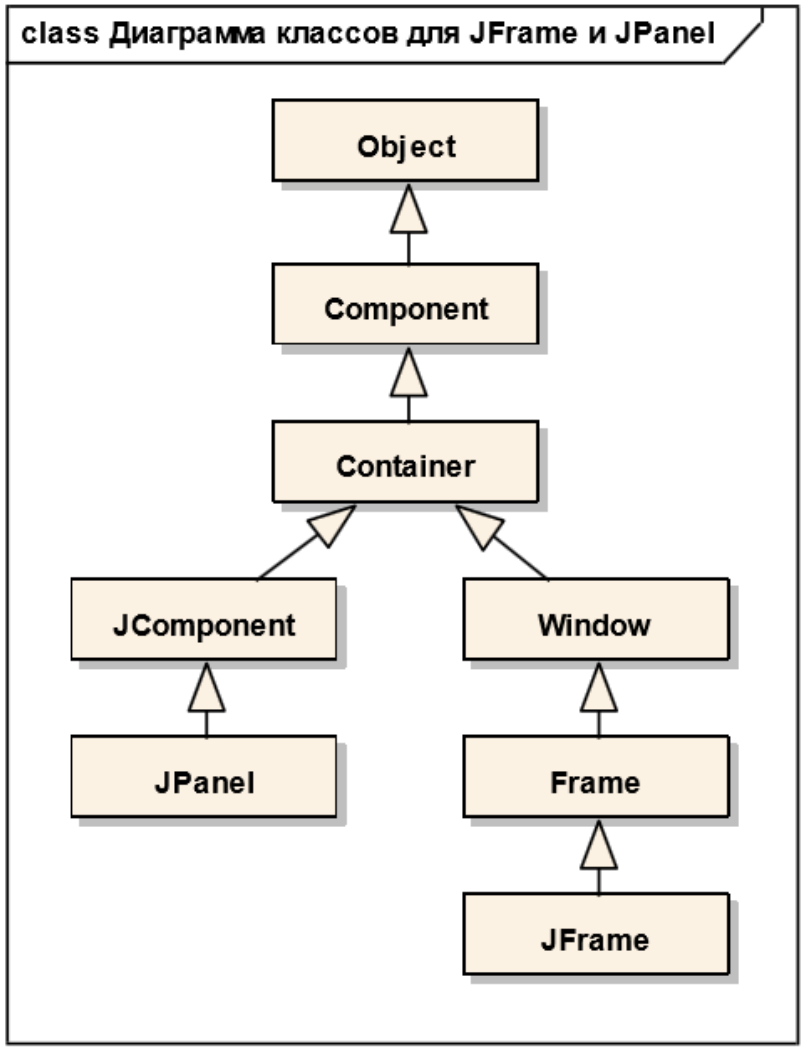
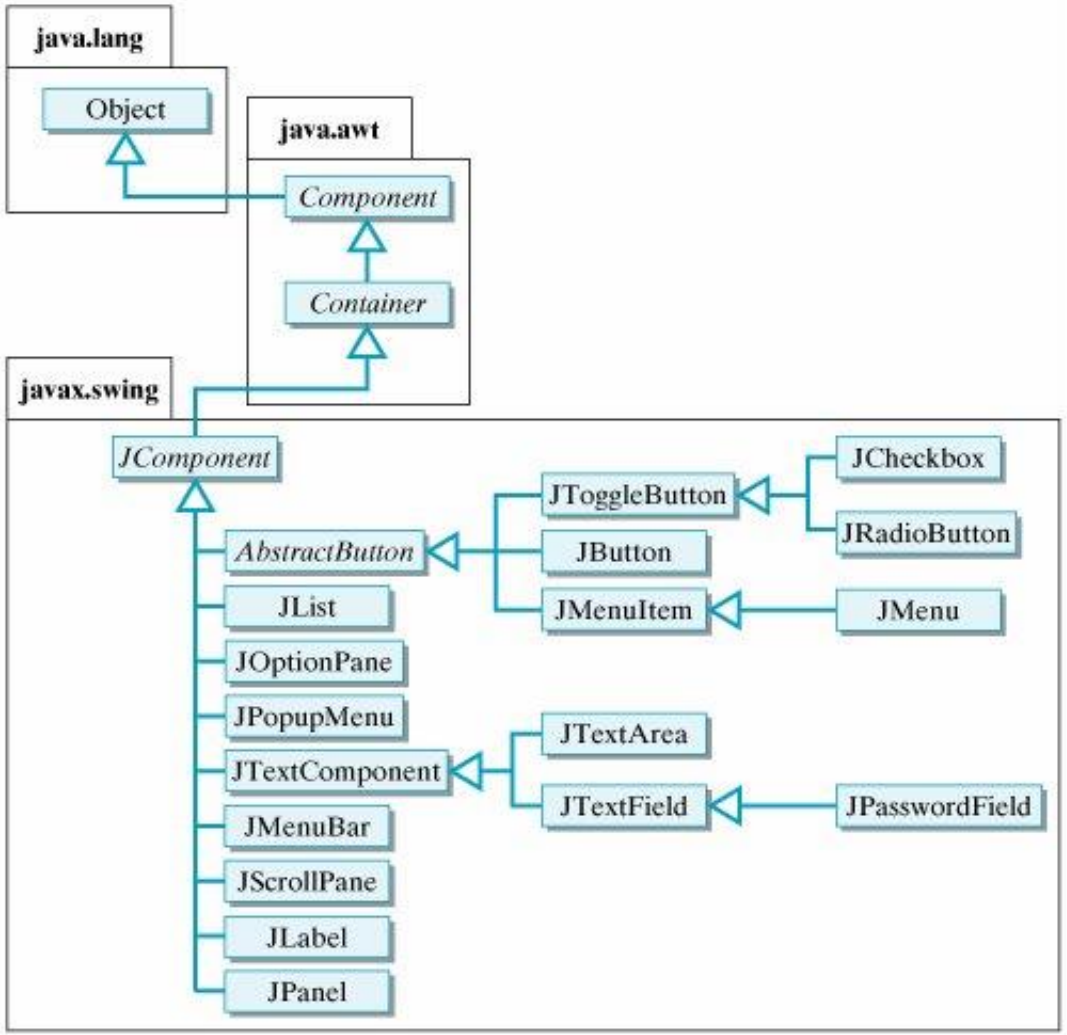


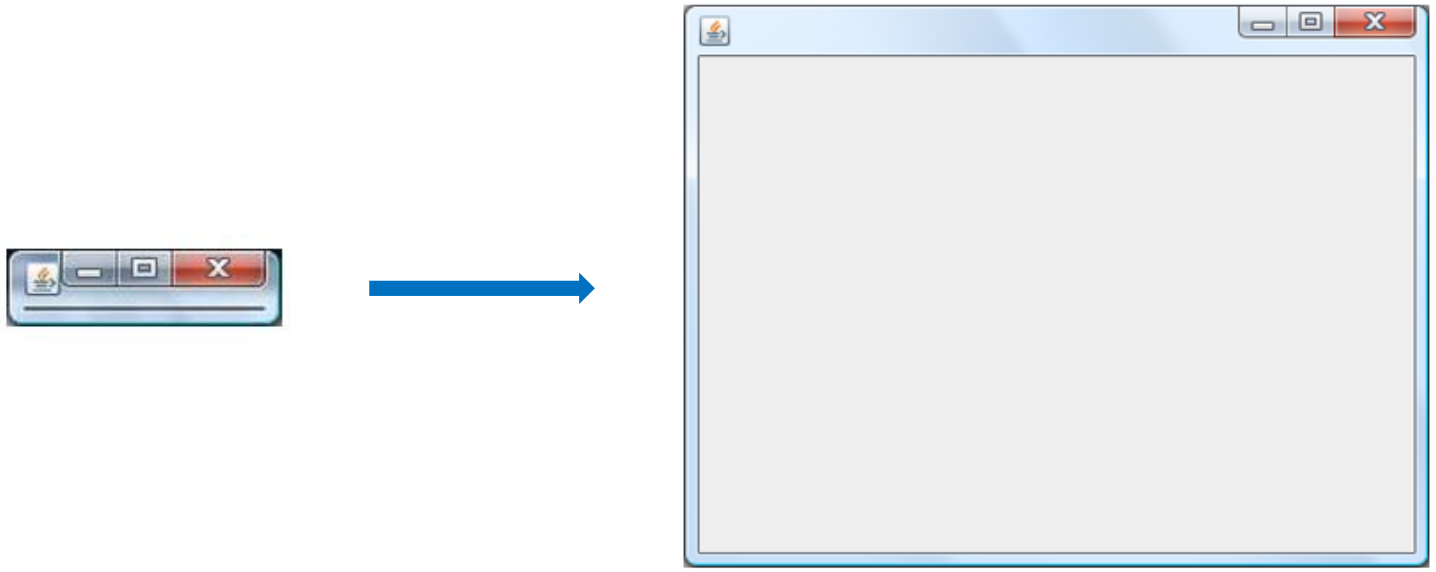
Диаграмма классов для Swing





Создание простейшего фрейма в Java

```
public static void main(String[] args) {
    // Конструируем экземпляр фрейма
    JFrame frame = new JFrame();
    // Задаём реакцию на нажатие кнопки закрытия фрейма
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    // Показываем фрейм на экране
    frame.setVisible(true);
}
```



Некоторые методы класса JFrame

boolean isVisible()	Позволяет проверить видимость компонента
void setVisible(boolean b)	Задает видимость компонента
boolean isEnabled()	Проверяет доступность компонента
void setEnabled(boolean b)	Делает компонент (окно) доступным
Point getLocation()	Возвращает положение верхнего левого угла компонента по отношению к левому верхнему углу компонента-контейнера
setLocation(int x, int y)	Задает положение верхнего левого угла компонента относительно верхнего левого угла компонента-контейнера. В случае окна (контейнера верхнего уровня) – относительно верхнего левого угла экрана.
Dimension getSize()	Возвращает текущие размеры компонента
setSize(int w, int h) **setSize(Dimension d)**	Задает размеры компонента
setResizable(boolean b)	Задает возможность изменения размеров окна
setTitle(String title)	Задает название окна в заголовке
setIconImage(Image icon)	Задает изображение для иконки окна

Получение платформенно-зависимой информации от ОС

В некоторых случаях, например при позиционировании или масштабировании окна, для получения ряда характеристик (разрешения экрана), необходимо взаимодействие с **ОС**, в которой выполняется приложение.

Подобная информация получается с помощью класса **Toolkit**, для получения экземпляра которого следует воспользоваться методом **getDefaultToolkit()**:

```
Toolkit kit = Toolkit.getDefaultToolkit();
```

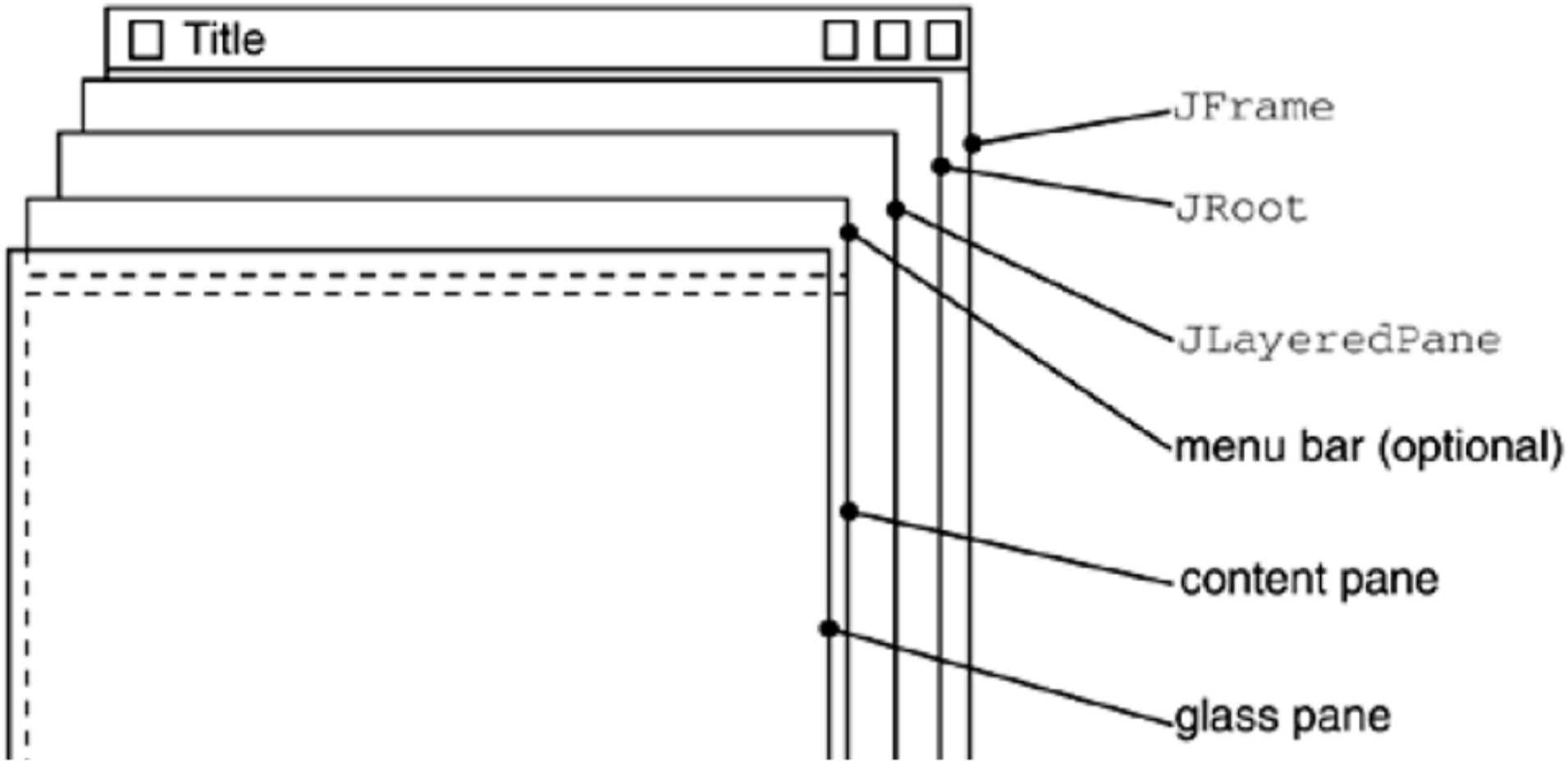
Получить размеры экрана можно следующим образом:

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

Операция загрузки изображения (зависящая от операционной системы) также выполняется с помощью **Toolkit**:

```
Image img = kit.getImage("icon.gif");
setIconImage(img);
```

Внутренняя структура фрейма



Компоновка элементов интерфейса внутри пространства фрейма

На ряде платформ и в многих средах разработки элементы интерфейса размещаются разработчиком в окне с помощью специализированного редактора, после чего их положение и размеры остаются неизменными.

Сложность данного подхода заключается в его низкой универсальности и переносимости:

- Пользователь может установить в ОС увеличенный размер шрифта, и размер элементов интерфейса будет недостаточным для того, чтобы вместить надписи, напечатанные шрифтом такого размера.
- При локализации разработанного приложения для другого региона или страны может оказаться, что названия элементов интерфейса, переведенные на другой язык, будут длиннее, и не смогут поместиться в область, зарезервированную разработчиком.

Компоновка элементов интерфейса внутри пространства фрейма

В Java принята концепция размещения элементов интерфейса на основе **менеджеров компоновки**, которые располагают и масштабируют компоненты друг относительно друга.

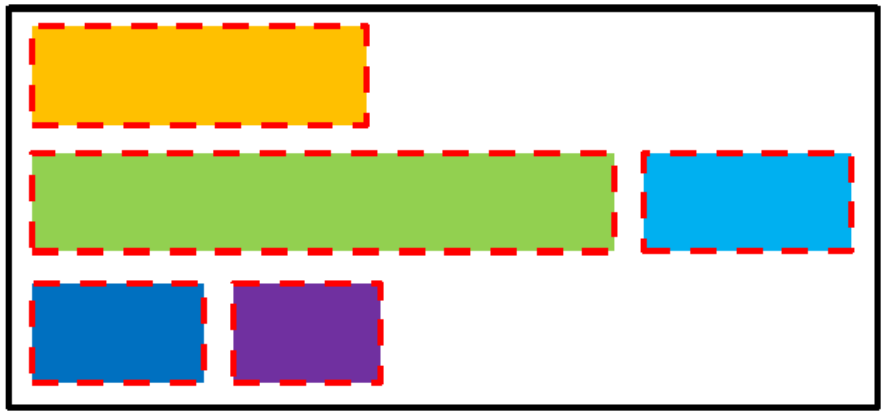
Разработчик задает **общие правила размещения**. Таким образом, вместо жесткого задания размеров и привязки элементов интерфейса имеет место их **динамическое позиционирование** с учетом правил, заданных разработчиком.

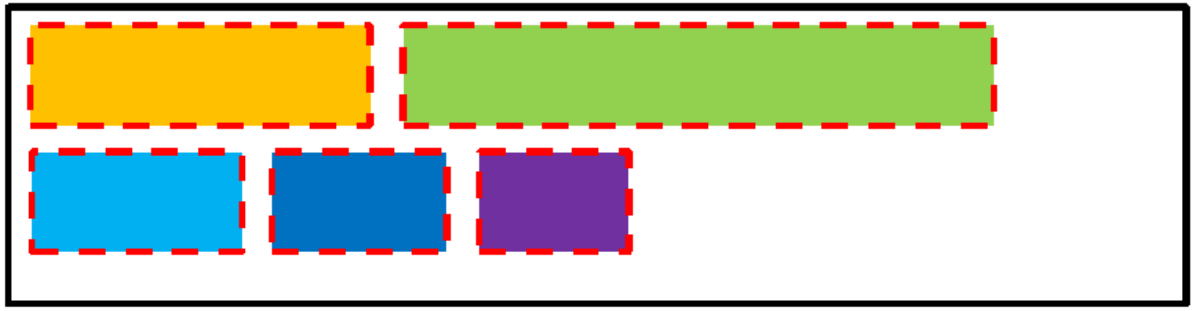
«Плавающая» компоновка – FlowLayout

Менеджер **FlowLayout** устанавливает компоненты **слева направо** и при заполнении **переходит на строку вниз**.

Смена менеджера компоновки осуществляется следующим образом:

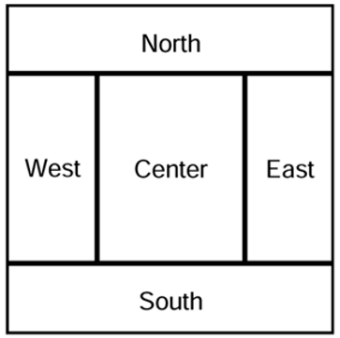
```
container.setLayout(new FlowLayout(FlowLayout.LEFT));
```





«Граничная» компоновка – BorderLayout

По умолчанию в **Swing** используется менеджер **BorderLayout**, в нем определены следующие константы для установки компонентов.



- BorderLayout.NORTH (верх)
- BorderLayout.SOUTH (низ)
- BorderLayout.EAST (справа)
- BorderLayout.WEST (слева)
- BorderLayout.CENTER (середина до других компонентов или до краев) – *по умолчанию*

Принцип размещения:

- Сначала рассчитывается место, занимаемое краевыми компонентами,
- все оставшееся место отводится центральной области.

При изменении размеров контейнера изменяются размеры только центральной области.

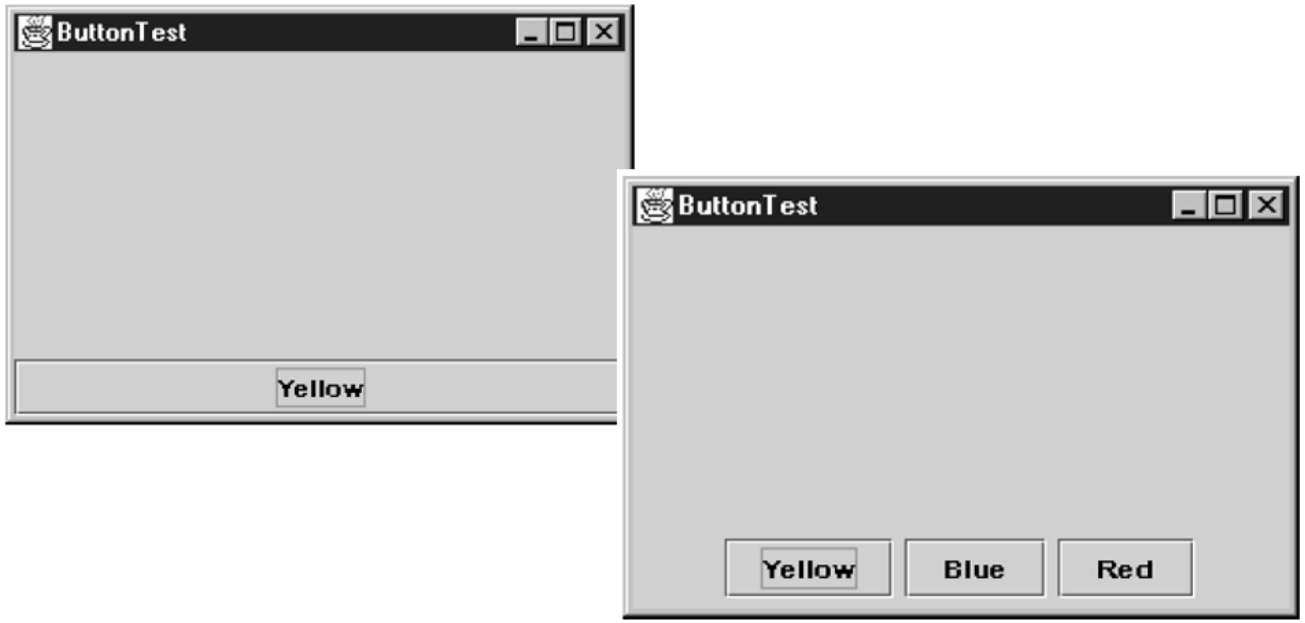
Пример задания граничной компоновки

```
// Создать экземпляр граничной компоновки с промежутками
// между ячейками по горизонтали - 10, по вертикали - 5
BorderLayout myLayout = new BorderLayout(10, 5);
// Добавить myComponent1 в верхнюю часть компоновки (север)
myLayout.add(myComponent1, BorderLayout.NORTH);
// Добавить myComponent2 в нижнюю часть компоновки (юг)
myLayout.add(myComponent2, BorderLayout.SOUTH);
// Установить граничную компоновку в качестве активной
// компоновки контейнера container
container.setLayout(myLayout);
```

Специфика граничной компоновки

В отличие от плавающей компоновки, граничная компоновка изменяет размер всех компонентов для заполнения доступного пространства.

Для предотвращения масштабирования компонентов следует использовать контейнер **«панель» (JPanel)**, у которого по умолчанию используется плавающая компоновка.



Использование плавающей компоновки внутри граничной

```
// Создать экземпляр граничной компоновки с промежутками
// между ячейками по горизонтали – 10, по вертикали – 5
BorderLayout myLayout = new BorderLayout(10, 5);
// Создать новый экземпляр контейнера JPanel
JPanel myPanel = new JPanel();
// Добавить в контейнер myComponent1
myPanel.add(myComponent1);
// Добавить в контейнер myComponent1
myPanel.add(myComponent2);
// Добавить контейнер myPanel в центральную часть
// граничной компоновки
myLayout.add(myPanel, BorderLayout.CENTER);
// Установить граничную компоновку в качестве
// активной компоновки контейнера container
container.setLayout(myLayout);
```

«Табличная компоновка» – GridLayout

GridLayout это менеджер, который помещает компоненты в таблицу.

Табличная компоновка организует все компоненты в строки и столбцы, как в таблице. Et особенностью является равенство размеров всех ячеек.

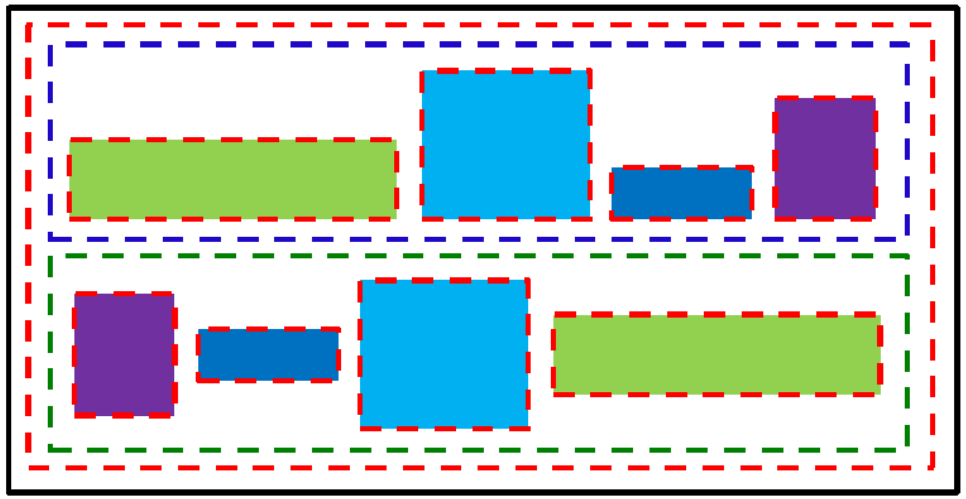
Добавление компонентов происходит с помощью метода **add(Component c)** последовательно:

- сначала заполняется 1-ый столбец 1-ой строки,
- потом 2-ой столбец 1-ой строки и т.д.

«Коробочная» компоновка – BoxLayout

BoxLayout позволяет управлять размещением компонентов, отдельно в вертикальном либо горизонтальном направлении помещая их, друг за другом, и управлять пространством между компонентами, используя вставки.

```
// создать «коробку» с горизонтальной укладкой
Box b1 = Box.createHorizontalBox();
// создать «коробку» с вертикальной укладкой
Box b2 = Box.createVerticalBox();
```

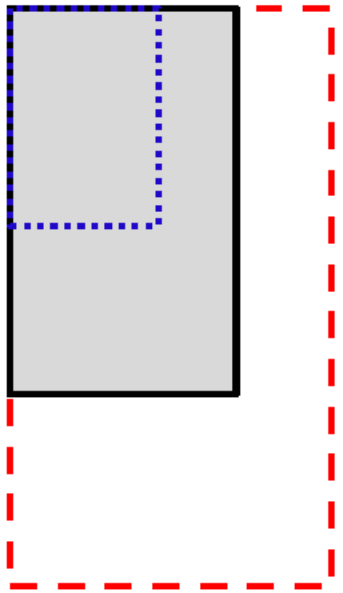


Вычисление размеров компонентов в коробочной компоновке

При размещении каждый компонент имеет три размера: **желаемый**, **максимальный** и **минимальный**.

При вычислении размеров менеджер руководствуется алгоритмом:

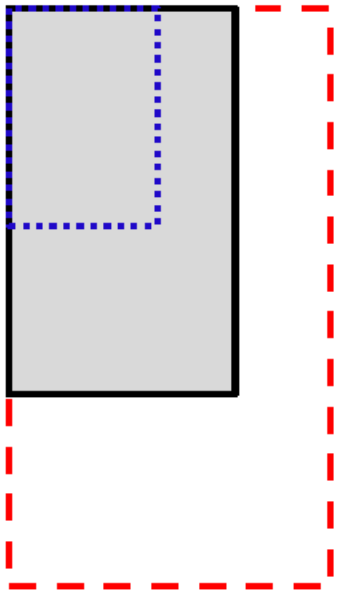
- Вычислить максимальную высоту самого высокого компонента.
- Попытаться растянуть высоту всех компонентов до этого уровня.
- Если какой-либо компонент не желает растягиваться до такой высоты, то определить его вертикальное выравнивание посредством метода **getAlignmentY().**



Вычисление размеров компонентов в коробочной компоновке

При вычислении размеров менеджер руководствуется алгоритмом:

- Получить желаемую ширину каждого компонента. Сложить полученные для всех компонентов значения.
- Если общая желаемая ширина меньше доступного места, расширить компоненты до их максимальной ширины. После этого выстроить компоненты слева направо без дополнительных пробелов между ними. Если желаемая ширина больше доступного места, то сжать компоненты до их минимального размера, но не более. Если все компоненты не помещаются даже при минимальной ширине, показать не все из них.



Предотвращение масштабирования элементов коробочной компоновки

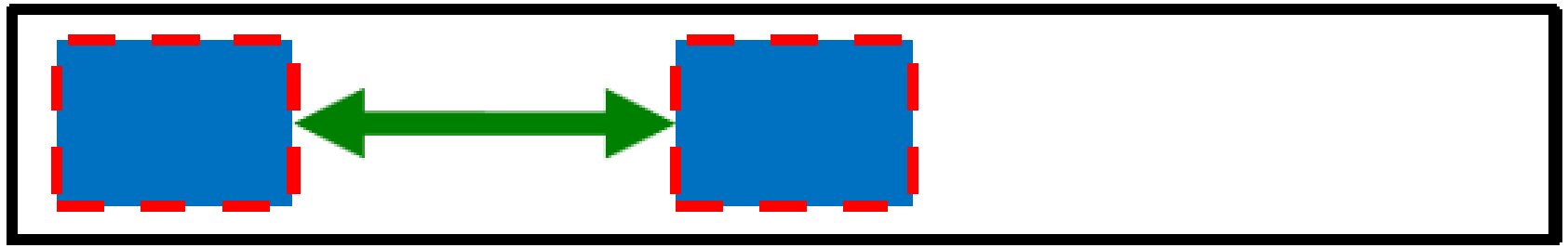
Коробочная компоновка увеличивает размер компонентов за пределы желаемого, поэтому для компонентов, масштабирование которых нежелательно (например, поля ввода), необходимо принудительно сделать максимальный размер равным желаемому:

```
textField.setMaximumSize(textField.getPreferredSize());
```

Использование «наполнителей» коробочной компоновки

Распорки (**strut**), добавляют фиксированный промежуток между компонентами:

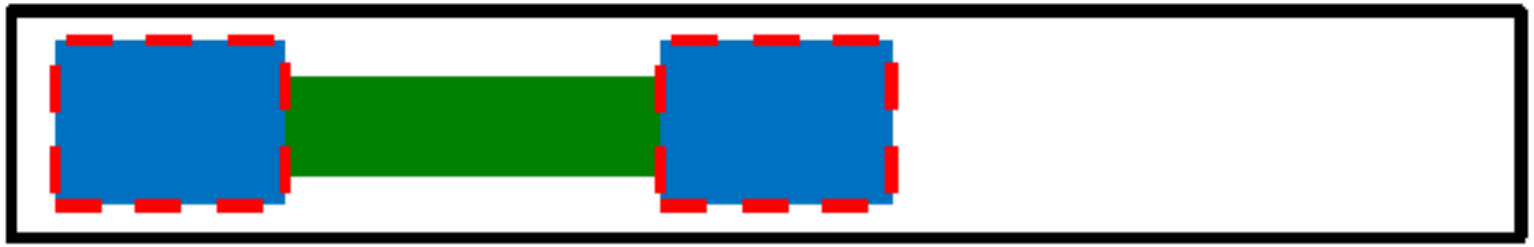
- box.add(label);
- box.add(Box.**createHorizontalStrut**(100));
- box.add(textField);



Использование «наполнителей» коробочной компоновки

Неподвижные области (**rigid area**), добавляют не только промежуток, но и **минимальный/максимальный/желаемый** размер по другому направлению:

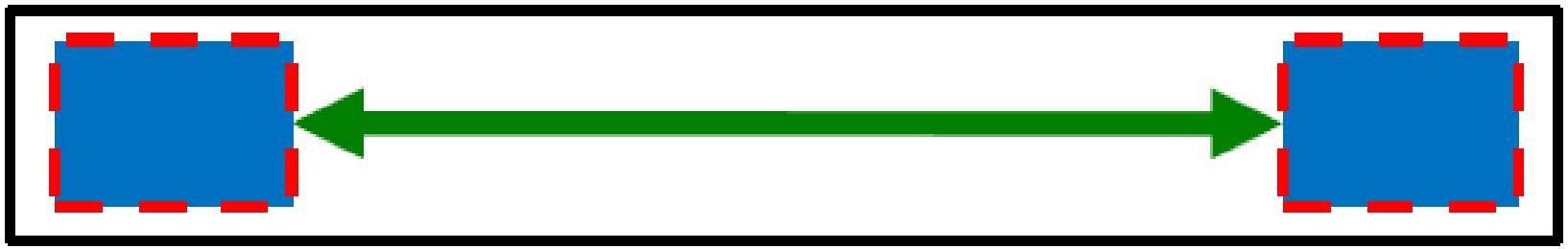
- box.add(Box.**createRigidArea**(new Dimension(100, 50));



Использование «наполнителей» коробочной компоновки

Клей (**glue**), максимальным образом удаляет компоненты друг от друга:

- box.add(button1);
- box.add(Box.**createGlue**());
- box.add(button2);



Произвольная компоновка

В некоторых случаях может оказаться необходимым разместить компоненты в фиксированных областях – **абсолютное позиционирование**.

Этот подход не рекомендуется для переносимых приложений, но может успешно использоваться для быстрого создания прототипов приложений.

В этом случае необходимо:

1. установить менеджер компоновки равным **null**;
2. добавить компоненты в контейнер;
3. указать размер и положение каждого компонента.

```
contentPane.setLayout(null);
JButton ok = new JButton("Ok");
contentPane.add(ok);
// Левый угол в точке (10,10), ширина – 30, высота – 15
ok.setBounds(10, 10, 30, 15);
```

Достойны упоминания

- **CardLayout** менеджер предназначен для использования нескольких менеджеров.
- **GroupLayout** менеджер имеет возможность независимо устанавливать горизонтальное и вертикальное расположение компонентов на форме. Он использует два типа добавления компонентов **параллельный** и **последовательный объединенный с иерархическим составом**.
    1. Последовательным добавляет компоненты просто помещая один за другим, точно так же как **BoxLayout** или **FlowLayout** вдоль одной оси. Положение каждого компонента определяется относительно предыдущего компонента.
    2. Помещает компонентов параллельно относительно друг друга в то же самом месте. Они добавляются к верху формы или выравниваются к основанию вдоль вертикальной оси. Вдоль горизонтальной оси они устанавливаются влево или по центру, если у компонентов разный размер.

Достойны упоминания

- **SpringLayout** очень гибкий менеджер, но и очень сложный для ручного кодирования изначально проектировался для использование в средах автоматического проектирования GUI например таких как NetBeans. Особенности его работы заключается в установки отношении между краями компонентов.

- Если необходимо самостоятельно расположить компоненты, то можно воспользоваться менеджером **NullLayout** установив в метод **setLayout()** значение **null**.

Дополнительные средства взаимодействия с пользователем

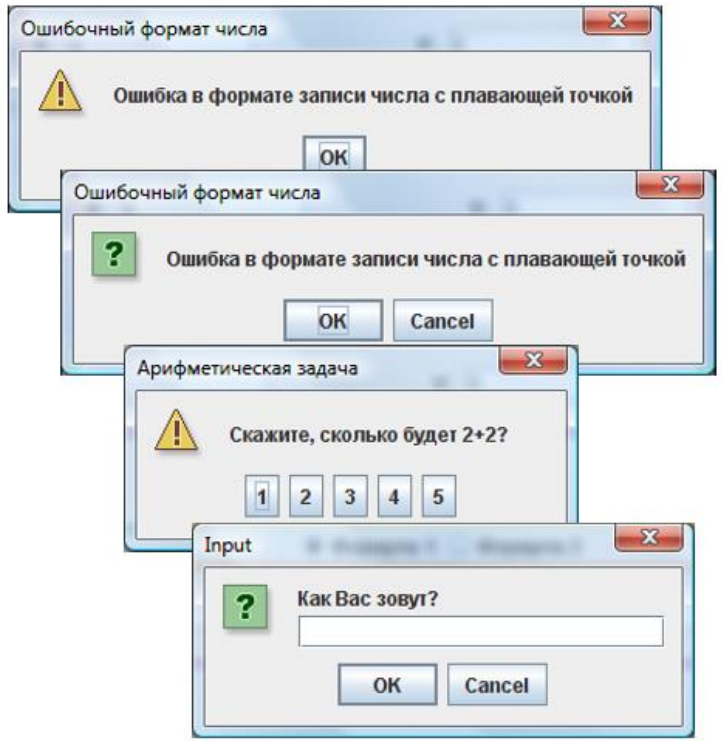
Для показа диалоговых окон пользователю предназначен класс **JOptionPane**, четыре статических метода которого позволяют организовать различные формы взаимодействия:

`showMessageDialog`

`showConfirmDialog`

`showOptionDialog`

`showInputDialog`



Задание реакции на взаимодействие пользователей с компонентами

Приложения, реализующие графический интерфейс, часто сталкиваются с необходимостью **обработки реакции на действия пользователя** с компонентами интерфейса.

Используемая в Java модель (**делегирование обработки событий**) основывается на следующих базовых понятиях:

- **событие** – объект, содержащий сведения о произошедшем событии, являющийся экземпляром потомка класса **java.util.EventObject**;
- **«слушатель»** – экземпляр класса, реализующий специальный интерфейс слушателя событий;
- **источник событий** – объект, регистрирующий заинтересованных слушателей и передающий им экземпляры объектов, описывающих произошедшие события (например, кнопка, полоса прокрутки).

Цикл обработки событий модели делегирования обработки

Весь цикл обработки событий состоит из следующих этапов:

1. Слушатель событий регистрируется у источника событий (источник включает слушателя во внутренний список заинтересованных слушателей) обращаясь к его методу:

   eventSourceObject.**addEventListener**(eventListenerObject);

2. При возникновении события источник событий создает экземпляр класса соответствующего класса, описывающего произошедшее событие.

Цикл обработки событий модели делегирования обработки

3. Источник событий для каждого заинтересованного слушателя вызывает метод обработки события, определенный интерфейсом слушателя, передавая в качестве аргумента экземпляр события.

4. Большинство компонентов пользовательского интерфейса генерируют событие типа **ActionEvent**, сообщающее о действии пользователя. Объекты, заинтересованные в получении уведомлений о событиях данного типа, должны реализовывать интерфейс **ActionListener**, требующий наличия метода **actionPerformed(ActionEvent event).**

Объявление и создание слушателей событий

```
class OkButtonActionListener implements ActionListener
   private Component parent;
   public OkButtonActionListener(Component parent) {
      this.parent = parent;
   }
   public void actionPerformed(ActionEvent ev) {
      JOptionPane.showMessageDialog(parent, "Нажата кнопка
OK", "Событие", JOptionPane.INFORMATION_MESSAGE);
   }
}
class MyFrame extends JFrame {
   public MyFrame() {
      …
      JButton buttonOK = new JButton("OK");
      buttonOK.addActionListener(new
OkButtonActionListener(this));
      …
   }
   …
}
```

Объявление и создание слушателей событий с помощью анонимных классов

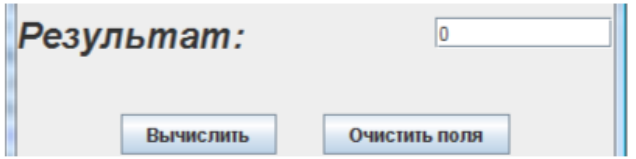
```
class MyFrame extends JFrame {
   public MyFrame() {
      …
      JButton buttonOK = new JButton("OK");
      buttonOK.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent ev) {
            JOptionPane.showMessageDialog(MyFrame.this,
"Нажата кнопка OK",
"Событие", JOptionPane.INFORMATION_MESSAGE);
         }
      });
      …
   }
   …
}
```

Компонент «Надпись» - JLabel

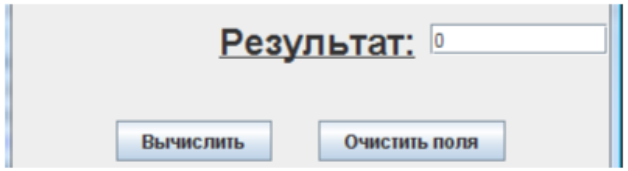
Компонент представляет надпись в окне, не имеет визуального декорирования (например, рамки) и не взаимодействует с пользователем.

Текст, отображаемый надписью, может содержать HTML-разметку. Исходный текст надписи задается в конструкторе, для изменения надписи во время работы приложения следует использовать метод **setText(String newText)**.

```
JLabel label1 = new JLabel("<html><i><font
size='6'>Результат:</font></i></html>");
container.add(label1);
```




```
JLabel label2 = new JLabel("<html><u><font
size='6'>Результат:</font></u></html>", JLabel.RIGHT);
container.add(label2);
```



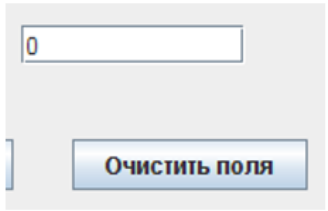
Компонент «Поле ввода» - JTextField

Компонент представляет прямоугольное поле ввода для одной строки.

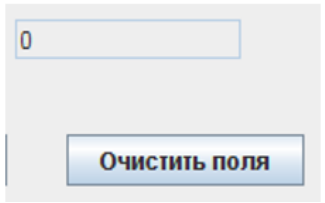
Для считывания значения поля используется метод **getText()**, для установки – **setText(String newText)**.

Метод **setEditable(Boolean)** позволяет задать, разрешено ли пользователю редактировать значение поля. Начальное значение поля ввода задается в конструкторе, вторым аргументом конструктору передается максимальное число символов, которое может быть введено в данное поле.

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




```
textField.setEditable(false);
```



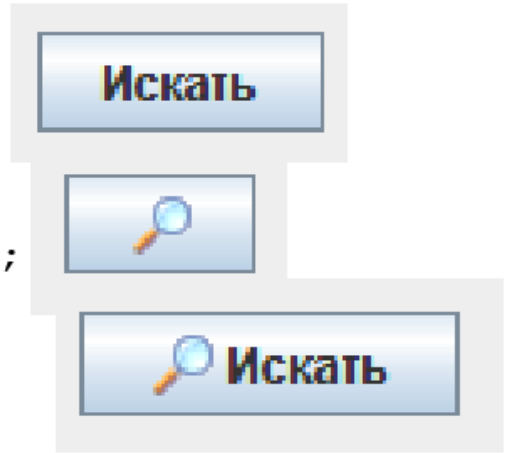
Компонент «Кнопка» - JButton

Компонент представляет прямоугольную кнопку, снабженную надписью и (возможно) иконкой.

При нажатии на кнопку генерируется событие **ActionEvent**, передаваемое всем заинтересованным слушателям.

ActionEvent, передаваемое всем заинтересованным слушателям.

```
// кнопка с надписью
JButton b1 = new JButton("Искать");
// кнопка с иконкой
JButton b2 = new JButton(icon);
// кнопка с надписью и иконкой
JButton b3 = new JButton("Искать", iconOK);
container.add(b1);
container.add(b2);
container.add(b3);
```



Компонент «Кнопка» - JButton

Для создания объекта-иконки на основе графического файла следует применять конструктор **ImageIcon(String filename)**:

```
JButton b4 = new JButton(new ImageIcon("search.gif"));
```

После создания экземпляра кнопки необходимо зарегистрировать слушателя, заинтересованного в уведомлениях о нажатии на ней:

```
b3.addActionListener(okButtonActionListener);
```

Компонент «Радио-кнопка» - JRadioButton

Компонент представляет круглую кнопку, позволяющую выбрать один (и только один) вариант из целой группы.

Когда выбирается другая кнопка, то выделение предыдущей кнопки автоматически пропадает. Обеспечение подобного поведения требует включения ряда радио-кнопок в группу кнопок (**ButtonGroup**).

Именно объект класса **ButtonGroup** и несет ответственность за «выключение» включенной кнопки при активации новой. При щелчке на радио-кнопку, как и для обычной кнопки, генерируется событие типа **ActionEvent**.

```
ButtonGroup myButtons = new ButtonGroup();
// вариант 1 по умолчанию «включен»
JRadioButton radio1 = new JRadioButton("Вариант 1", true);
myButtons.add(radio1);
// вариант 2 по умолчанию «выключен»
JRadioButton radio2 = new JRadioButton("Вариант 2", false);
myButtons.add(radio2);
```

Компонент «Радио-кнопка» - JRadioButton

Для определения слушателей событий щелчков на радио-кнопках необходимо действовать по аналогии с добавлением слушателей на обычные кнопки:

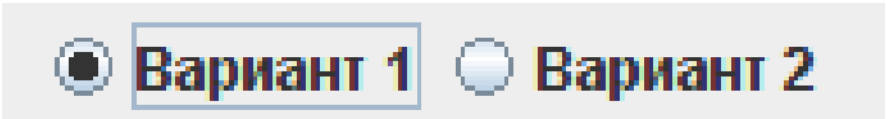
```
radio1.addActionListener(radio1ActionListener);
radio2.addActionListener(radio2ActionListener);
```

Компонент «Радио-кнопка» - JRadioButton

Добавление радио-кнопок в группу не избавляет от необходимости добавить каждую из них в объект контейнер, в противном случае она не будет отображена.

Объект **ButtonGroup** отвечает только за единственность выделения кнопки в группе, но не за ее размещение на экране:

```
container.add(radio1);
container.add(radio2).
myFrame.getContentPane().add(container, BorderLayout.NORTH);
```



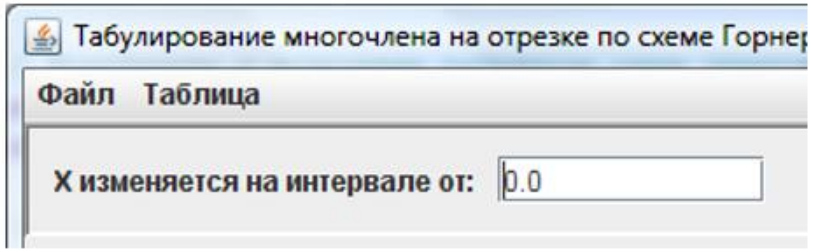
Компонент «Полоса меню» - JMenuBar

Чтобы создать меню приложения необходимо создать полосу меню. Полоса меню создается с помощью обращения к конструктору:

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

Полоса меню является таким же компонентом, как и другие объекты библиотеки **Swing**, и может быть размещена внутри любого компонента. В большинстве случаев мы будем размещать ее в верхней части главного окна. Это делается обращением к методу **setJMenuBar()** класса **Frame**:

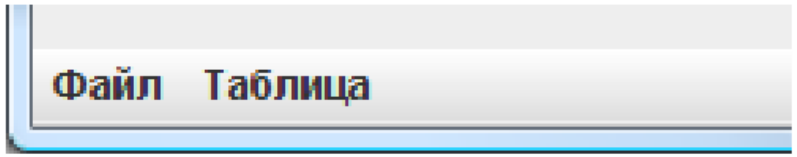
```
setJMenuBar(menuBar);
```



Компонент «Полоса меню» - JMenuBar

Выполнение следующего фрагмента кода обеспечит размещение полосы меню в нижней части главного окна приложения.

```
// Блок 3 – размещение меню в нижней части окна
getContentPane().add(menuBar, BorderLayout.SOUTH);
```



Компонент «Полоса меню» - JMenuBar

Любое меню (**верхнего уровня**, пункты которого размещены вверху окна, или **вложенного уровня**) представляется экземпляром класса **JMenu**, который создается обращением к конструктору **JMenu(menuName)**, принимающему в качестве аргумента имя создаваемого меню:

```
JMenu fileMenu = new JMenu("Файл");
```

Для добавления меню верхнего уровня в полосу меню, а также для добавления вложенных меню в меню-контейнеры применяется метод add():

```
menuBar.add(fileMenu);
JMenu nestedMenu = new JMenu("Экспортировать данные");
fileMenu.add(nestedMenu);
```

Компонент «Элемент меню» - JMenuItem

Наиболее простым способом создания элемента меню является конструирование экземпляра **JMenuItem** с помощью метода **add()** родительского меню.

Например, если у нас есть объект **fileMenu** типа **JMenu**, то элемент **openFileMenuItem** типа **JMenuItem** создается следующим образом:

JMenuItem openFileMenuItem = **fileMenu.add**("Открыть");

Используя возвращенную ссылку на экземпляр **JMenuItem** можно прикрепить к нему слушателя событий типа **ActionEvent**.

Удобным способом для этого является применение анонимных классов:

```
openFileMenuItem.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent ev) {
                    // Код обработки события
            }
});
```

Компонент «Диалоговое окно выбора файла» - JFileChooser

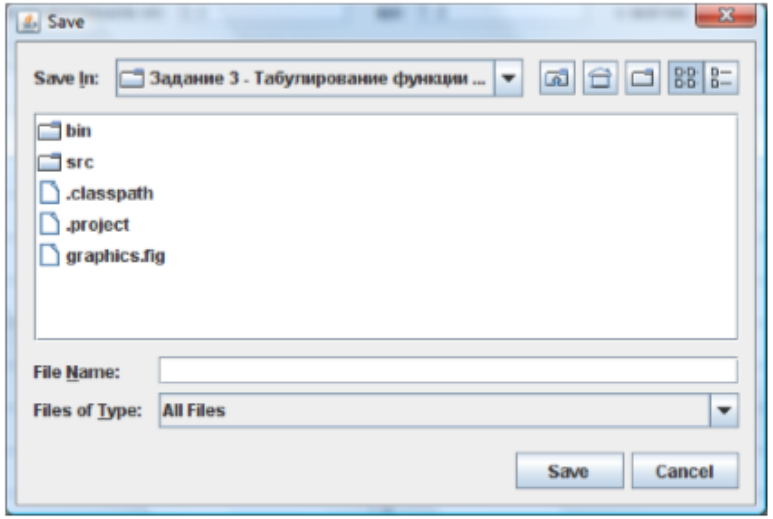
Компонент отображает диалоговое окно, выводящее списки папок и файлов файловой системы и позволяющий выбрать один или несколько файлов.

Так как создание экземпляра данного класса является ресурсоемкой операцией, рекомендуется повторно использовать созданные экземпляры, сохранив ссылку на объект во внутреннем поле объекта.

```
JFileChooser fileChooser = new JFileChooser();
```

Далее необходимо установить текущую директорию, а также (при необходимости) файл, выделенный по умолчанию:

```
fileChooser.setCurrentDirectory(new File("."));
```



Компонент «Диалоговое окно выбора файла» - JFileChooser

Диалоговое окно может быть показано как при открытии файла, так и при его сохранении.

В классе **JFileChooser** для этого предусмотрены два различных метода – **showOpenDialog**() (в этом случае кнопка будет иметь подпись «**Open**») и **showSaveDialog**() (кнопка будет иметь подпись «**Save**»).

В качестве аргумента любому из этих методов передается указатель на компонент, являющийся «родителем» диалогового окна (в большинстве случаев им будет являться ссылка на экземпляр главного окна приложения):

```
int result = fileChooser.showSaveDialog(MainFrame.this);
```

Компонент «Диалоговое окно выбора файла» - JFileChooser

Показ диалогового окна является модальной операций, возвращающей в целочисленной переменной результат операции:

- если пользователь закрыл окно успешно выбрав какой-либо файл, значение **result** будет равно **JFileChooser.APPROVE_OPTION**.
- если пользователь закрыл диалоговое окно, нажав «**Cancel**», результатом будет **JFileChooser.CANCEL_OPTION**.

Узнать (в случае успешного закрытия окна), какой файл был выбран пользователем, можно обратившись к методу **getSelectedFile()**, возвращающему объект класса **File**.

```
File selectedFile = fileChooser.getSelectedFile();
```

Компонент «Таблица» - JTable

Компонент **JTable** отображает двумерную матрицу объектов.

Он не хранит данные внутри себя, а получает их из **модели таблицы**. При создании экземпляра таблицы модель передается конструктору в качестве аргумента. Полученный в итоге компонент таблицы можно добавлять в окно приложения по аналогии с другими компонентами (кнопками, полями ввода и т.п.).

Следует отметить, что таблица обладает достаточно широкими возможностями поведения – позволяет выделять ячейки и строки, масштабировать столбцы и изменять их порядок следования.

```
// data - модель таблицы, содержащая данные для отображения
// Создаётся экземпляр таблицы
JTable table = new JTable(data);
// Таблица "оборачивается" в панель, обладающую
// возможностями прокрутки и устанавливается в качестве
// содержания окна
getContentPane().add(new JScrollPane(table),
BorderLayout.CENTER);
```

Представление данных для таблицы – класс AbstractTableModel

Модель таблицы выступает по отношению к компоненту JTable источником данных. Класс **AbstractTableModel** является абстрактным, поэтому создание его экземпляров невозможно, и он используется только в качестве базового класса для определения собственной модели данных.

Основные задачи модели таблицы:

- **определение размерности таблицы** (задание количества строк и столбцов модели);
- **задание имен столбцов** (для отображения заголовков столбцов);
- **задание типов данных в столбцах** (для использования совместно с визуализаторами ячеек);
- **получение доступа к элементу таблицы** (значение i-ой строки j-го столбца).

Визуализация ячеек таблицы

В библиотеке **Swing** имеются три встроенных визуализатора, автоматически выбираемых в зависимости от типа данных в столбцах таблицы:

Тип данных	Способ отображения
ImageIcon	Как картинка
Boolean	Как флаг
Object	Как строка

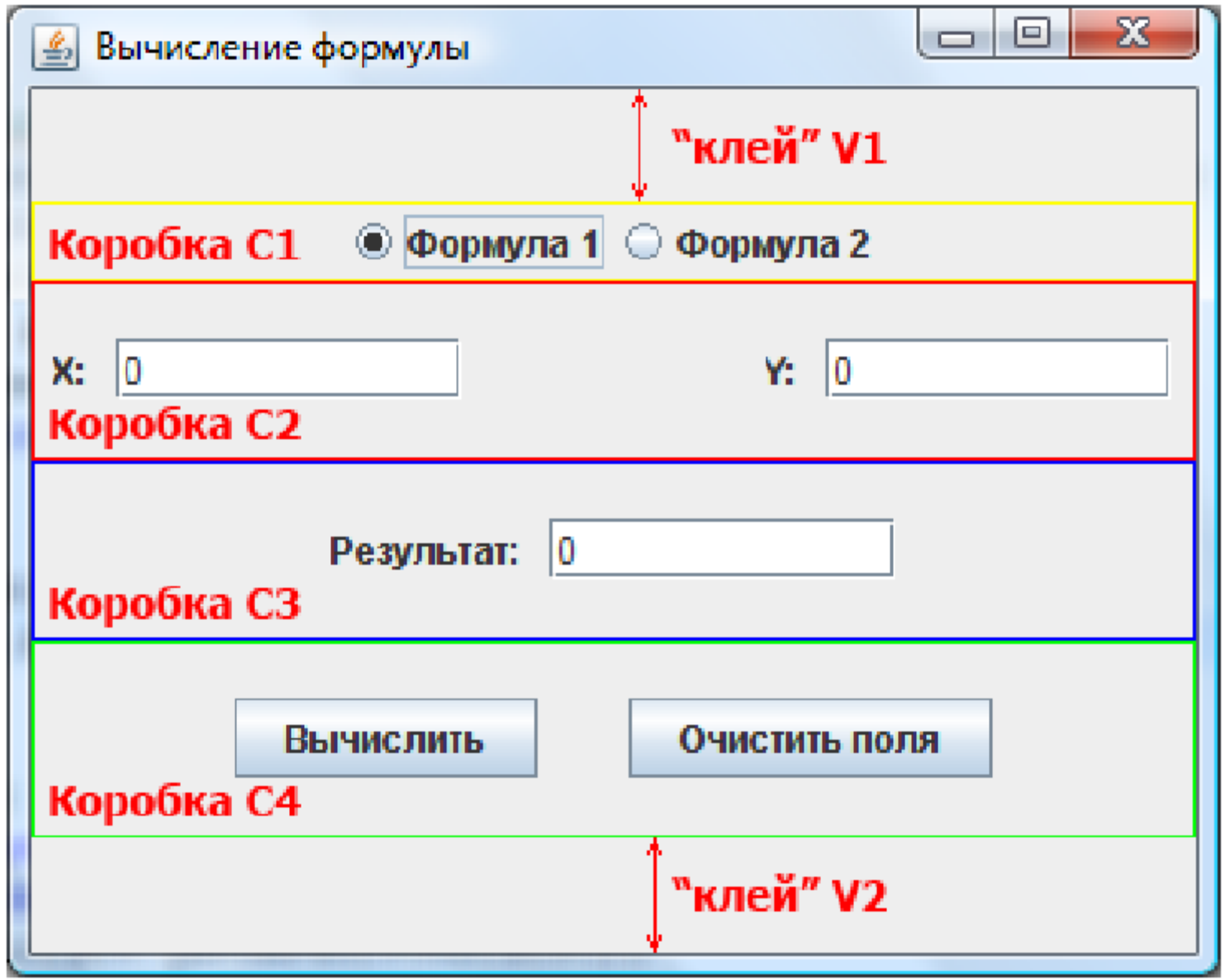
При необходимости задания собственного способа отображения данных модели (например, вместо показа кода цвета в шестнадцатеричной системе – заливка ячейки этим цветом) следует определить собственный **класс-визуализатор ячеек таблицы**, реализующий интерфейс **TableCellRenderer**.

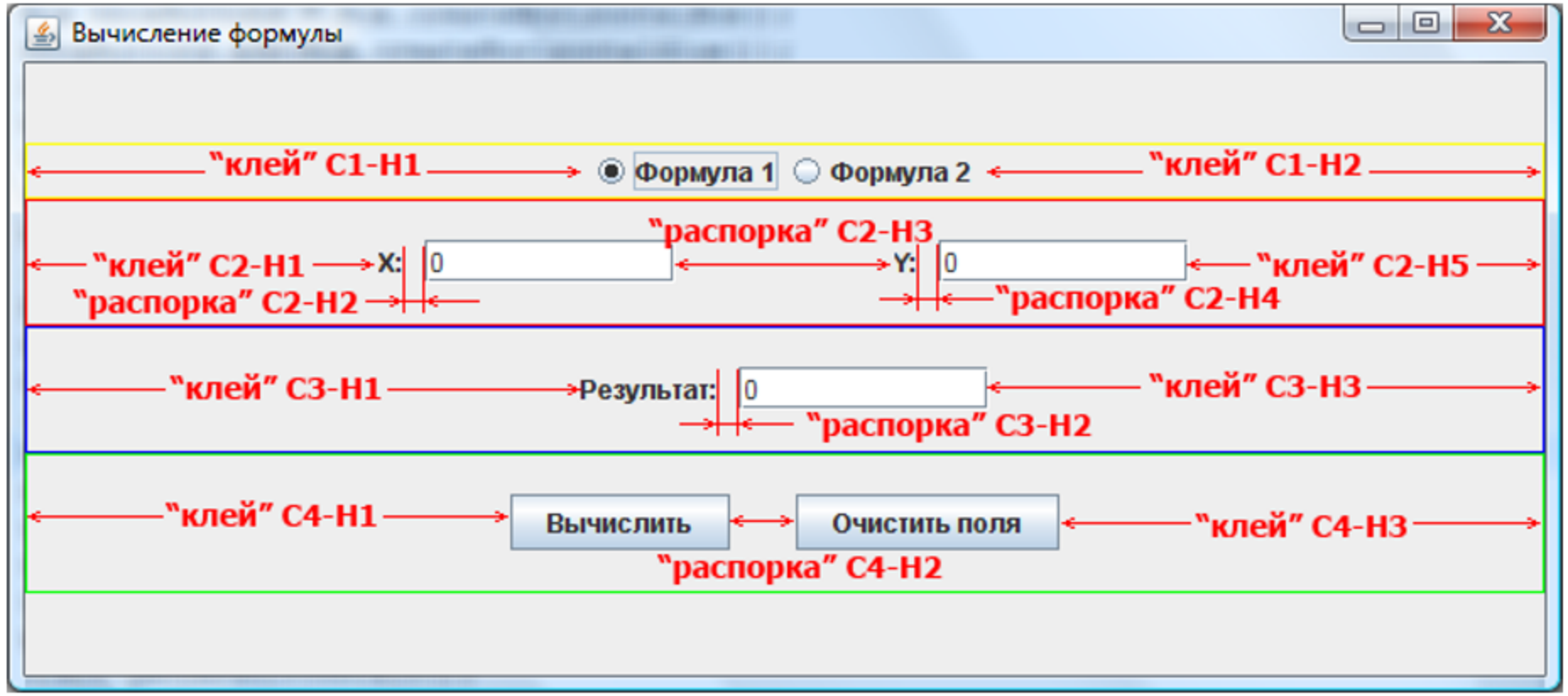
Визуализация ячеек таблицы – интерфейс TableCellRenderer

При необходимости задания собственного способа отображения данных модели следует определить собственный класс-визуализатор ячеек таблицы.

Для этого необходимо:

1. В модели таблицы определить тип данных в столбцах переопределив метод **getColumnClass**().
2. Создать класс, реализующий интерфейс визуализатора ячеек, переопределив метод **getTableCellRendererComponent**(), которому передаются в аргументах:
   - ссылка на экземпляр отображаемой таблицы;
   - значение в отображаемой ячейке;
   - флаг выделения ячейки;
   - флаг наличия фокуса ввода в ячейке;
   - позиция ячейки в таблице.
3. Связать визуализатор с определенным типом данных.


Вычисление формулы
"клей" V1
Коробка C1
Формула 1
Формула 2
X:
0
Y:
0
Коробка C2
Результат:
0
Коробка C3
Вычислить
Очистить поля
Коробка C4
"клей" V2


Вычисление формулы
“клей” С1-Н1
Формула 1
Формула 2
“клей” С1-Н2
“распорка” С2-Н3
“клей” С2-Н1
X:
0
Y:
0
“клей” С2-Н5
“распорка” С2-Н2
“распорка” С2-Н4
“клей” С3-Н1
Результат:
0
“клей” С3-Н3
“распорка” С3-Н2
“клей” С4-Н1
Вычислить
Очистить поля
“клей” С4-Н3
“распорка” С4-Н2

Метод paintComponent() – основа отображения вида компонента

В оконных приложениях Java за отображение любого компонента отвечает метод **paintComponent**(), получающий в качестве аргумента объект класса **Graphics**.

Этот объект представляет некоторый виртуальный холст, на котором происходит рисование, хранит набор параметров для отображения изображений и текста, а также предоставляет для этого ряд методов.

При каждой перерисовке окна (по любой причине) обработчик события перерисовки уведомляет компонент.

Это приводит к вызову метода **paintComponent**() всех компонентов интерфейса.

По этой причине, стандартным способом вывода графики в приложениях является использование классов-потомков стандартных компонентов пользовательского интерфейса библиотеки **Swing** (например, **JPanel**, **JButton** и т.д.), с переопределением в них метода **paintComponent**().

Компонент JPanel – основа для вывода графики

Одним из наиболее предпочтительных кандидатов на использование в качестве базового класса при разработке собственного компонента, отображающего графику на экране, является класс **JPanel**.

Причин для этого несколько:

- при компоновке панель масштабируется таким образом, чтобы заполнить все доступное пространство;
- панель является контейнером, т.е. может содержать вложенные компоненты;
- метод **paintComponent**() панели только закрашивает все ее пространство цветом заднего фона и ничего не отображает сверху.

Сам процесс вывода графики состоит из последовательности операций с виртуальным холстом, соответствующим пространству, занимаемому отображаемым компонентом.

Вывод графики в Java версии 1.0

В **Java версии 1.0** средства отображения графики были весьма простыми:

выбирался **цвет** и **режим «заливки» фона**, после чего вызывались методы класса **Graphics**, такие как **drawRect**() и **fillOval**().

Начало координат холста (точка с координатами (0,0)) располагалась в левом верхнем углу отображаемого компонента, а измерения выполнялись в точках, поэтому все координаты были целочисленными.

Вывод графики в Java версии 1.0

drawLine()	Рисует линию
drawRect()	Рисует контур прямоугольника
drawRoundRect()	Рисует контур прямоугольника с закругленными краями
draw3DRect()	Рисует контур прямоугольника с эффектом объема
drawPolygon()	Рисует замкнутый многоугольник
drawPolyline()	Рисует ломаную линию
drawOval()	Рисует овал
drawArc()	Рисует дугу

Вывод графики в Java начиная с версии 2.0

Начиная **с Java версии 2.0**, в состав J2SE включена библиотека **Java 2D**, реализующая широкий набор графических операций.

Ее особенностями являются:

- Управление пером, т.е. типом линий, которыми прорисовываются границы фигур.
- Создание заливки с однородными цветами, градиентами, повторяющимися шаблонами.
- Использование трансформаций для перемещения, масштабирования, поворота, растягивания фигур.

Вывод графики в Java начиная с версии 2.0

Начиная **с Java версии 2.0**, в состав J2SE включена библиотека **Java 2D**, реализующая широкий набор графических операций.

Ее особенностями являются:

- Использование кадрирования для ограничения области перерисовки.
- Использование правил композиции для описания способов объединения точек новой фигуры с существующими точками.
- Настройка ориентиров визуализации для задания компромисса между скоростью и качеством прорисовки.
- Вывод графики на основе объектных представлений геометрических фигур.
- Использование вещественных координат.

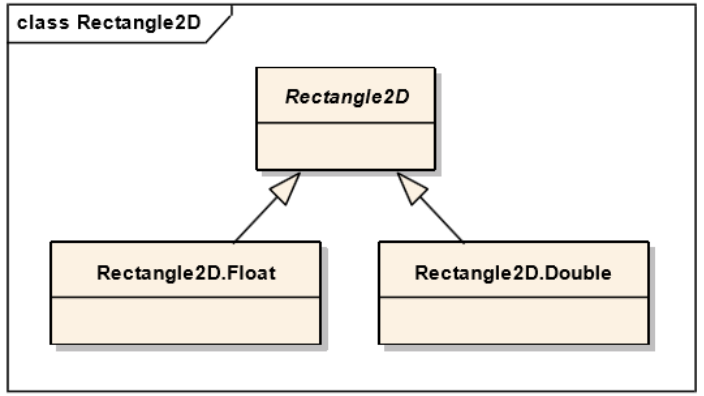
Вывод геометрических фигур с помощью их объектных представлений

Point2D	Представляет точку, с координатами (x,y). Используется для задания фигур, но сам фигурой не является.
Line2D	Фигура, представляющая линию
Rectangle2D	Фигура, представляющая прямоугольник
RoundRectangle2D	Фигура, представляющая прямоугольник с закруглѐнными краями
Ellipse2D	Фигура, представляющая эллипс (круг)
Arc2D	Фигура, представляющая дугу
QuadCurve2D	Фигура, представляющая кривую второго порядка
GeneralPath	Фигура, состоящая из множества сегментов, каждый из которых может быть линией, кривой второго или третьего порядка.
QubicCurve2D	Фигура, представляющая кривую третьего порядка

Использование вещественных координат в Java 2.0

Использование вещественных координат в ряде случаев является большим удобством, так как позволяет задавать координаты фигур в осмысленных единицах (метрах, дюймах, километрах), затем автоматически преобразуя их в точки на основе заданных коэффициентов масштаба.

В Java 2D для внутренних вычислений используются числа типа **float**, точности которых достаточно. Тем не менее, так как многие методы в Java работают с вещественными числами повышенной точности (**double**), а **float** и **double напрямую несовместимы!** (требуется явное преобразование типов), в Java 2D предусмотрены **две версии каждого класса фигур**: использующих координаты одинарной **float** и двойной точности **double**.



Процесс визуализации фигур

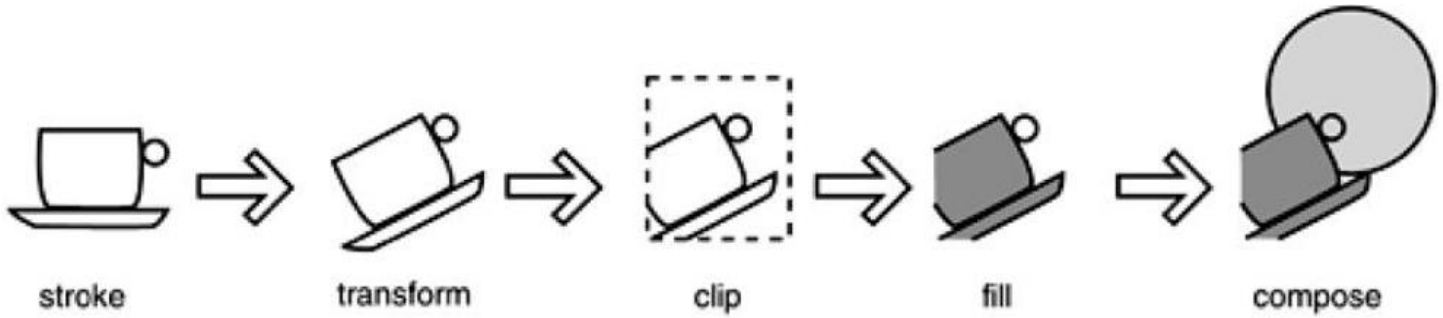
При инициализации экземпляров фигур библиотеки Java 2D непосредственной их визуализации не происходит.

Фигура отображается только при обращении к методу **draw()** или **fill()** класса **Graphics2D**. В этот момент новая фигура обрабатывается в **конвейере визуализации**.

Последовательность этапов следующая:

- прорисовывается контур фигуры;
- применяются заданные преобразования (сдвига, поворота и т.п.);
- фигура кадрируется (если фигура и область кадрирования не пересекаются, то процесс визуализации прекращается);
- оставшаяся после кадрирования часть фигуры закрашивается;
- точки закрашенной фигуры совмещаются с существующими точками.

Процесс визуализации фигур



Класс Graphics2D

Для доступа к расширенным возможностям библиотеки Java 2D необходимо в явном виде привести экземпляр класса **Graphics** (например, полученный как аргумент в методе **paintComponent**()) к классу **Graphics2D**, который является представлением поверхности, на которой происходит рисование, и предоставляет ряд методов для настройки параметров отображения и непосредственного рисования.

```
public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D canvas = (Graphics2D) g;
        // Дальнейшие действия с экземпляром класса
        // ...
}
```

Методы класса Graphics2D

draw(Shape)	Рисует контур фигуры с учѐтом настроек пера, кадрирования, трансформации, заливки, композиции
fill(Shape)	Заполняет внутреннюю часть фигуры с использованием настроек трансформации, кадрирования, заливки и композиции
Color getColor()	Возвращает текущий цвет холста
Font getFont()	Возвращает текущий шрифт холста
Paint getPaint()	Возвращает текущую заливку холста
Stroke getStroke()	Возвращает текущее перо холста
setColor(Color)	Устанавливает для холста текущий цвет
setFont(Font)	Устанавливает для холста текущий шрифт
setPaint(Paint	Устанавливает для холста текущую заливку
setStroke(Stroke)	Устанавливает для холста текущее перо

Работа с контекстом отображения – экологический подход

При изменении текущих настроек холста хорошим стилем считается сохранение предыдущих настроек до их использования и восстановление старых значений после использования.

В следующем фрагменте сохраняются и восстанавливаются значения текущего цвета и заливки:

```
// Сохранение текущих настроек
Color oldColor = canvas.getColor();
Paint oldPaint = canvas.getPaint();
// Установка новых настроек
canvas.setColor(Color.RED);
canvas.setPaint(Color.BLUE);
// Рисование с новыми настройками
// ...
// Восстановление старых настроек
canvas.setColor(oldColor);
canvas.setPaint(oldPaint);
```

Некоторые классы геометрических фигур

Для отображения линий в библиотеке **Java 2D** предназначен абстрактный класс **Line2D** с двумя потомками **Line2D.Float** (представляющий координаты с помощью **float**) и **Line2D.Double** (представляющий координаты с помощью **double**).

Для отображения эллипсов и окружностей в библиотеке **Java 2D** предназначен абстрактный класс **Ellipse2D** с двумя потомками **Ellipse2D.Float** (представляющий координаты с помощью **float**) и **Ellipse2D.Double** (представляющий координаты с помощью **double**).

Для задания сложных линий, состоящих из различных сегментов, в библиотеке **Java 2D** используется класс **GeneralPath** (общий путь).

Сегментами пути могут являться как прямые линии (**Line2D**), так и кривые второго (**QuadCurve2D**) и третьего (**QubicCurve2D**) порядка.

Выбор типа начертания линий – класс BasicStroke

Операция отображения **draw()** класса **Graphics2D** прочерчивает границу фигуры с использованием текущих настроек пера. По умолчанию, перо чертит сплошную линию толщиной в 1 точку.

С помощью метода **setStroke()** класса **Graphics2D** можно изменять настройки пера, передавая методу в качестве аргумента класс, реализующий интерфейс **Stroke**. В библиотеке **Java 2D** единственным классом, реализующим данный интерфейс, является класс **BasicStroke**.

Для инициализации объектов данного класса предложен ряд конструкторов, наиболее сложный из которых позволяет при создании нового пера задавать целый ряд параметров:

- толщину линий;
- оформление края линий;
- оформление смыкания линий;
- предельный угол смыкания линий;
- шаблон линий;
- начальное смещение в шаблоне.

Параметры начертания линий

- Толщина линии задается вещественным числом типа **float**, и может быть как целым, так и дробным значением.
- Для оформления края линий на выбор предложено три варианта: стиль **CAP_BUTT** делает край линии резко обрубленным; **CAP_ROUND** – завершает его полукругом; **CAP_SQUARE** – завершает его половиной квадрата (сторона которого равна толщине линии).

Параметры начертания линий

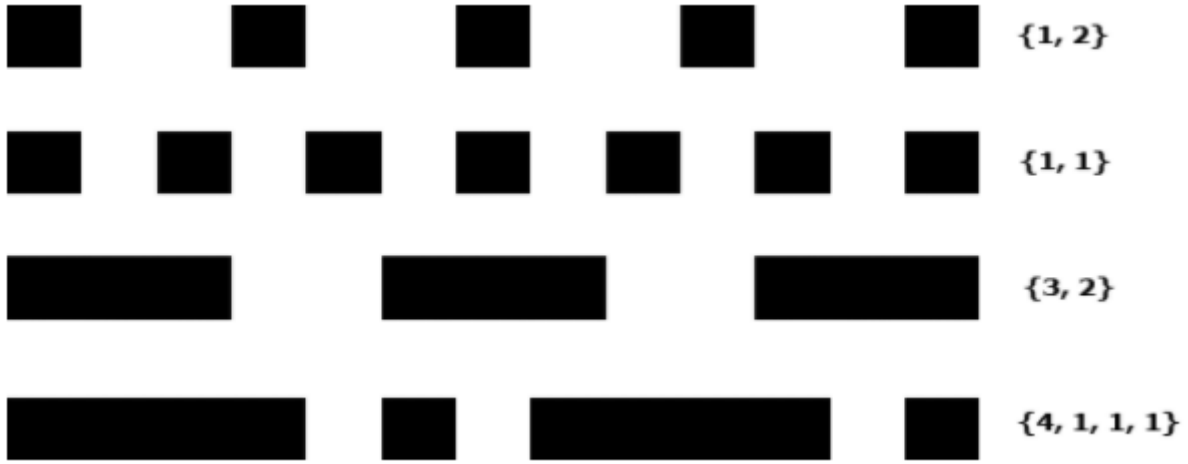
- Задание способа оформления смыкания линий позволяет указать, каким из трех вариантов будет оформлено смыкание двух линий: стиль **JOIN_BEVEL** соединяет смыкающиеся линии прямой, перпендикулярной биссектрисе угла между ними; **JOIN_ROUND** – дополняет конец каждой линии полукруглым завершением; **JOIN_MITER** – удлиняет обе линии, образуя острый «шип».

Параметры начертания линий (продолжение)

- Тип **JOIN_MITER** не приспособлен для оформления соединений линий, смыкающихся под небольшими углами, так как это может выразиться в очень длинных «шипах». Для предотвращения этого эффекта, вводится такой параметр как предельный угол смыкания линий (по умолчанию – 10 градусов).
- Если две линии смыкаются под углом, не превышающим значения предельного, вместо стиля **JOIN_MITER** используется стиль **JOIN_BEVEL**.

Параметры начертания линий (продолжение)

- Шаблон линий задается в как массив вещественных чисел типа **float**, каждое из которых определяет длину чередующихся отрезков «линия есть» - «линии нет».



- Параметр начального смещения в шаблоне задает, в каком месте шаблона должно начинаться отображение каждой линии (по умолчанию – это значение равно 0).

Выбор способа заливки – заливка однородным цветом

Операция заливки замкнутой области **fill**() класса **Graphics2D** использует текущие настройки заливки.

Для задания альтернативного способа заливки применяется метод **setPaint**(), которому в качестве аргумента необходимо передать класс, реализующий интерфейс **Paint**.

Одним из классов, его реализующих, является класс **Color**, поэтому для заполнения фигур однородным цветом необходимо обратиться к методу **setPaint**(), передав в качестве аргумента объект класса **Color**, например:

```
canvas.setPaint(Color.RED);
```

Выбор способа заливки – градиентная заливка

Более сложные способы заливки обеспечивает класс **GradientPaint** (градиентная заливка), интерполируя цветовые оттенки между двумя заданными значениями. Объекты класса инициализируются посредством конструктора, которому передаются две точки и цвета, которые должны быть в этих точках, например:

```
// Задать исходную точку для градиента
Point2D from = new Point2D.Double(0, 0);
// Задать конечную точку для градиента
Point2D to = new Point2D.Double(10, 10);
// Установить новый способ заливки фигур
canvas.setPaint(new GradientPaint(from,Color.RED, to,Color.BLUE));
```

Выбор способа заливки – градиентная заливка

Цвета изменяются вдоль линии, соединяющей две заданные точки, и постоянны на линиях, перпендикулярных ей. Точки, находящиеся далее краевых точек имеют цвета краевых точек.

Дополнительным параметром, который может быть передан конструктору, является флаг периодичности заливки. Если он установлен в **true**, то цвета циклически продолжают изменяться и за пределами отрезка, ограниченного двумя точками:

```
canvas.setPaint(new GradientPaint(from,Color.RED,to,Color.BLUE,true));
```

Выбор шрифта для надписей – класс Font

Для вывода текста и надписей используется метод **drawString**() класса **Graphics2D**, принимающий в качестве аргументов строку s и отображающий ее в точке с координатами (x, y).

Отображение осуществляется с использованием текущего шрифта (вначале, текущим шрифтом является шрифт по умолчанию).

Для отображения символов каким-либо особенным шрифтом, необходимо создать экземпляр класса **Font**, для чего указывается имя шрифта, его стиль и размер, например:

```
// Создать шрифт с именем “Serif”, жирный, размер 36 пунктов
Font myFont1 = new Font("Serif", Font.BOLD, 36);
```

Указание необходимой гарнитуры и начертания шрифта

Так как 100%-ного способа обеспечить наличие определенного шрифта на компьютере пользователя нет, то для задания некоторой определенности в библиотеке AWT определено пять логических имён шрифтов:

- SansSerif
- Serif
- Monospaced
- Dialog
- DialogInput

Указание необходимой гарнитуры и начертания шрифта

Представленные шрифты всегда связываются со шрифтами, которые действительно присутствуют на компьютере пользователя.

Например, в ОС Windows логическое имя ***SansSerif*** будет связано с шрифтом Arial.

При задании стиля шрифта возможны варианты:

- обычный (Font.PLAIN);
- жирный (Font.BOLD);
- курсив (Font.ITALIC);
- жирный курсив (Font.BOLD + Font.ITALIC).

Использование TrueType-шрифтов

Начиная с J2SE версии 1.3, стало возможным использовать шрифты **TrueType**.

Для использования шрифта его сначала нужно загрузить из потока ввода – обычно файла в файловой системе или удаленного файла.

Получив объект открытого потока ввода, следует обратиться к статическому методу **Font.createFont**(). Объект, возвращенный в качестве результата выполнения метода, будет представлять шрифт размером 1 пункт.

Для установки другого размера шрифта следует воспользоваться методом **deriveFont**(), например:

```
// Открыть поток чтения из файла mySuperFont.ttf
InputStream in = new FileInputStream(new
File("mySuperFont.ttf"));
// Создать объект шрифта TrueType размером 1 пт
Font mySuperFont = Font.createFont(Font.TRUETYPE_FONT, in);
// На основе созданного шрифта, создать шрифт размером 14пт
Font mySuperFont14pt = mySuperFont.deriveFont(14.0f);
```

Оценка размеров области, необходимой для размещения надписи

Достаточно часто возникает задача размещения текста или надписи определенным образом по отношению к ориентирам (например, центрирование на экране, выравнивание по верхнему краю рисунка и т.п.). Для ее выполнения необходимо знать ширину и высоту (в точках) надписи.

Эти размеры зависят от трех факторов:

- используемого шрифта;
- выводимой строки;
- устройства, на котором отображается текст (экран, принтер и т.п.).

Оценка размеров области, необходимой для размещения надписи

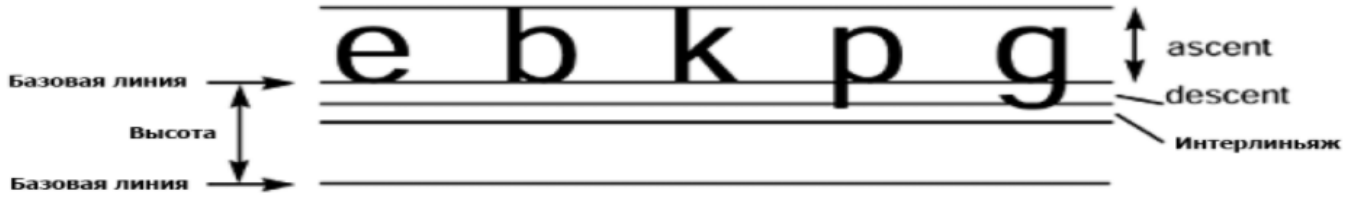
Для получения объекта, характеризующего свойства устройства, используется метод **getFontRenderContext**() класса **Graphics2D**. Возвращаемый им экземпляр класса **FontRenderContext** может использоваться для вычисления размеров надписи с помощью метода **getStringBounds**() класса **Font**:

```
// Создать объект контекста отображения текста
FontRenderContext context = canvas.getFontRenderContext();
// Получить размеры области для отображения надписи "Hello!"
Rectangle2D bounds = mySuperFont.getStringBounds("Hello!",
context);
```

Параметры, характеризующие анатомию шрифта

Для того, чтобы интерпретировать значения, возвращаемые методом **getStringBounds()**, следует знать некоторые термины, имеющие отношение к анатомии шрифта:

- **Базовая линия** – это воображаемая линия, на которой находятся нижние части большинства символов.
- **Аскент** – это расстояние от базовой линии до верхней части высоких символов, таких как b, k и т.д., или символов верхнего регистра.
- **Дескент** – это расстояние от базовой линии до нижней части низких символов, таких как p, g и т.д.
- **Интерлиньяж** – это расстояние между дескентом одной линии и аскентом другой.
- **Высота шрифта** – это расстояние между последовательными базовыми линиями, то есть аскент + дескент + интерлиньяж.



Интерпретация размеров области

Ширина прямоугольника, возвращаемого методом **getStringBounds**(), представляет ширину отображаемой надписи.

Высота прямоугольника равна сумме аскента и дескента. Начало координат прямоугольника находится на базовой линии. Верхняя укоордината прямоугольника отрицательна.

Т.О. можно вычислить все характеристики шрифта следующим образом:

- **double stringWidth = bounds.getWidth();**
- **double stringHeight = bounds.getHeight();**
- **double ascent = -bounds.getY();**
- **double descent = bounds.getHeight() + bounds.getY();**

Интерпретация размеров области

Для вычисления интерлиньяжа и полной высоты шрифта следует воспользоваться методом **getLineMetrics**() класса Font, который возвращает объект класса **LineMetrics**.

Методы-селекторы **getHeight**() и **getLeading**() полученного объекта позволяют узнать интерлиньяж и высоту шрифта:

- **LineMetrics metrics = mySuperFont.getLineMetrics(message, context);**
- **float fontHeight = metrics.getHeight();**
- **float leading = metrics.getLeading();**

Увидимся на следующей лекции!