

Клас ScrollPane

Класс-контейнер **ScrollPane** используется для представления в большом окне любого объекта, включая рисунки или таблицы.

Конструктор определяет новую панель прокрутки: **ScrollPane()**

Методы класса:

public Point getScrollPosition()

возвращает текущие координаты точки прокрутки

public void setScrollPosition(int x, int y)
public void setScrollPosition(Point p)

устанавливают новое значение точки прокрутки

public void setWheelScrollingEnabled(boolean handleWheel)

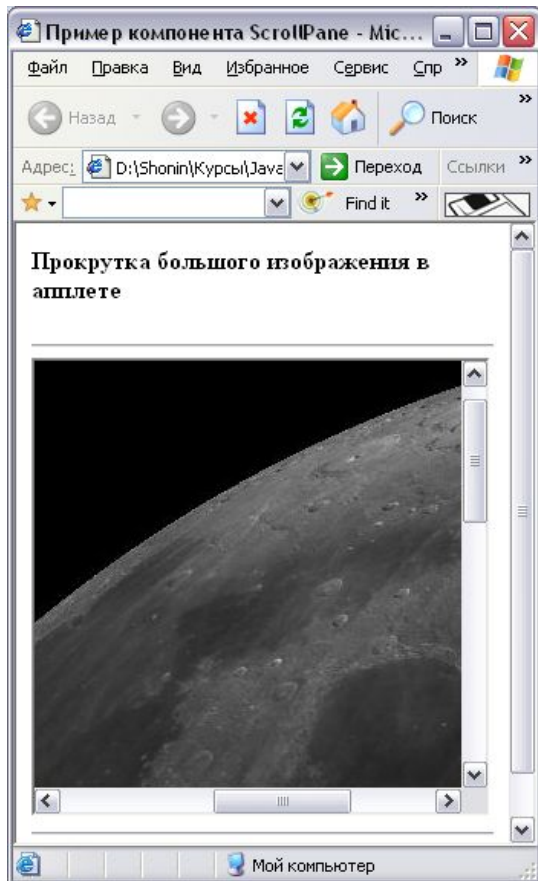
включает/выключает реагирование на прокручивание колесика мыши

public boolean isWheelScrollingEnabled()

проверяет, включена ли реакция на прокручивание колесика мыши.

Добавление панели прокрутки в апплет или приложение производится с помощью метода **add()**

```
ScrollPane myScrollPane =  
new ScrollPane(ScrollPane.SCROLLBARS_ALWAYS);  
add (myScrollPane);
```



Параметр конструктора может иметь значение одного из следующих свойств:

ScrollPane.SCROLLBARS_ALWAYS —
линейки прокрутки выводятся всегда;

ScrollPane.SCROLLBARS_NEVER —
линейки прокрутки никогда не выводятся;

ScrollPane.SCROLLBARS_AS_NEEDED —
линейки прокрутки выводятся при
необходимости.

Клас Window

Окно (Window) является родительским классом для классов **Dialog** и **Frame**.

Конструктор создает невидимое окно, которое ведет себя подобно диалоговому окну:

```
public Window(Frame parentFrame)
```

Класс содержит следующие методы:

```
public void show()
```

— показывает окно

```
public void hide()
```

— прячет окно

```
public boolean isShowing()
```

— показывает состояние окна

```
public void pack()
```

— устанавливает размер окна таким, чтобы все его компоненты имели предпочтительные размеры

public void toFront()

– перевод окна поверх текущего окна

public void toBack()

– перевод окна позади текущего окна

public void dispose()

– закрытие окна

Включение и удаление блоков прослушивания для окон выполняется с помощью соответственно методов

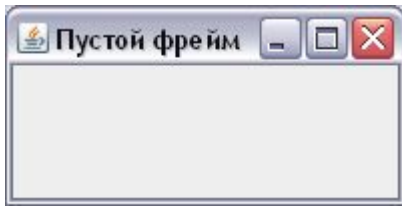
public void addWindowListener(WindowListener l)

public void removeWindowListener(WindowListener l)

Объекты класса **Window** не создаются непосредственно. Вместо этого используется подкласс этого класса – **Frame**.

Клас Frame

Фреймы (Frames) представляют собой элементы AWT, с помощью которых можно заставить апплет работать вне окна Web-браузера. Также фреймы используются также при создании графических приложений.



Фрейм можно создать конструктором:

```
public Frame()  
public Frame(String frameTitle)
```

После создания, перед отображением фрейма на экране, необходимо задать его размеры и положение на экране, используя методы

```
public void setSize(int x, int y)  
public void setLocation(int x, int y)
```

Если положение фрейма не задано он выводится в левом верхнем углу экрана.

public void isVisible(boolean visibility)

делает фрейм видимым
или невидимым

Невидимый фрейм продолжает существовать. По умолчанию фрейм **невидим**, поэтому его необходимо при работе программы сначала перевести в видимое состояние.

Пока фрейм существует, он занимает часть ресурсов оконной системы, в которой выполняется приложение или апплет. Если фрейм больше не нужен, его следует уничтожить, используя метод **dispose**, определенный в классе **Window**.

public String getTitle() и public void setTitle(String newTitle)

— определение и задание заголовка, отображаемого в верхней части фрейма;

public void setResizable(boolean allowResizing)

— установка или отмена запрета изменения размеров фрейма

**public Image getIconImage()
public void setIconImage(Image image)**

— определение и установка значка-изображения для данного фрейма;

Створення графічного додатку

1. Объявить класс в приложении как класс, расширяющий класс **Frame**.
 2. Определить в классе конструктор, в котором должны быть заданы следующие операторы:
 - 2.1. **super("имя");** – для вызова конструктора класса **Frame**.
 - 2.2. **setSize(ширина, высота);** – установка размеров окна.
 - 2.3. **addWindowListener(this)** – добавить блок прослушивания окна с обработкой закрытия окна с помощью метода **windowClosing()**, а также (при необходимости) других методов интерфейса **WindowListener** или класса **WindowAdapter**;
 - 2.4. **setVisible(true);** – сделать окно видимым.
- В конструкторе могут быть определены и другие действия.
3. Создать в методе **main** с помощью конструктора новый объект класса, расширяющего класс **Frame**.
 4. При необходимости выполнить первоначальную прорисовку окна с помощью метода **paint()**.

Вставка изображений

Язык Java имеет готовые классы, способные загружать файлы изображений, эти классы обеспечивают работу только с изображениями, представленными в графических форматах **GIF, PNG** и **JPEG**.

Вставка изображения в графические приложения выполняются двумя различными способами.

Методы класса **Toolkit** из пакета **java.awt**:

```
public abstract Image getImage(String filename)
public abstract Image getImage(URL url)
```

Обращение к этим методам из компонента выполняется через метод класса **Component**:

```
public Toolkit getToolkit()
```

В общем случае обращение можно сделать через статический метод класса **Toolkit**:

```
public static Toolkit getDefaultToolkit()
```


Пример:

```
Image img = getToolkit().getImage("D:\\images\\ myimage.gif");  
Image img =  
Toolkit.getDefaultToolkit().getImage("D:\\images\\myimage.gif");
```

Класс **Toolkit** содержит пять методов **createImage()**, возвращающих ссылку на объект типа **Image**.

```
public abstract Image createImage (String fileName)
```

– создает изображение из содержимого графического файла **filename**

```
public abstract Image createImage (url address)
```

– создает изображение из содержимого графического файла по адресу **address**

Созданное изображение выводится на экран в методе **paint()** одним из перегружаемых методов **drawImage()** класса **Graphics**.



Техніка анімації

Анимация выполняется в программах на языке Java с помощью потоков.

1. В классе должен быть реализован интерфейс **Runnable**.
2. Задается целое число или числа – интервалы задержки между кадрами (в миллисекундах).
3. Объявляется переменная класса **Thread** для потока анимации.
4. В методе **start()** апплета или **main()** графического приложения создается новый поток для объявленной переменной и для него запускается метод **start()** интерфейса **Runnable**.
5. В методе **stop()** апплета или **main()** графического приложения производится приостановка потока (путем присвоения потоку анимации значения **null**).
6. В методе **run()** потоку анимации обычно присваивается минимальный приоритет для того, чтобы он не мешал выполнению других потоков программы, а затем запускается цикл анимации. Задержка между кадрами анимации задается с помощью метода **sleep()** класса **Thread**.

При работе программ, выполняющих перемещение объектов на экране дисплея, хорошо заметно мелькание. Это происходит потому, что происходит перерисовка всей фигуры, которая заметна глазу и вызывает эффект мелькания. Стандартный выход из такой ситуации — техника, называемая **двойной буферизацией** (double-buffering).

Идея, лежащая в основе двойной буферизации, состоит в том, что вне экрана создается изображение, и все рисование происходит именно на этом изображении. Закончив рисование, можно скопировать полученное изображение на экран посредством одного метода, так что обновление экрана произойдет мгновенно.

В объявлениях, находящихся в начале класса, следует объявить объект **Image**, который будет служить внеэкранным изображением:

```
private Image offScreenImage;
```

Добавить строку в метод, где изображение будет инициализировано:

```
offScreenImage = createImage(size().width, size().height);
```

Другим источником мелькания служит метод **update()**. Стандартный метод **update()** в апплете очищает область рисования, а затем вызывает метод **paint()**. Чтобы избавиться от мелькания из-за очистки экрана достаточно переопределить **update()** следующим образом:

```
public void update(Graphics g) {  
// Получение графического контекста для внеэкранного изображения  
Graphics offScreenGraphics = offScreenImage.getGraphics();  
// Очистка внеэкранного изображения:  
// 1. Выбор в качестве цвета рисования цвета фона апплета  
offScreenGraphics.setColor(getBackground());  
// 2. Закраска всей области этим цветом  
offScreenGraphics.fillRect(0, 0, size().width, size().height);  
// 3. Возврат прежнего значения цвета рисования  
offScreenGraphics.setColor(g.getColor()) ;  
// Вызов метода paint  
paint(offScreenGraphics) ;  
// Копирование внеэкранного изображения на экран  
g.drawImage(offScreenImage, 0, 0, this);  
}
```

При анимации, особенно с использованием последовательно сменяемых файлов изображений, иногда изображение меняется не плавно, а скачками. Это происходит в тех случаях, когда графические файлы загружаются недостаточно быстро для гладкого воспроизведения анимации.

Для устранения этого недостатка используется класс **MediaTracker**, в котором реализовано отслеживание состояния мультимедийных элементов.

Этот класс содержит конструктор

MediaTracker(Component comp)

обычно в качестве компонента используется ключевое слово **this**.

Для сглаживания анимации с помощью класса **MediaTracker** необходимо:

1. Добавить в программу переменную класса **MediaTracker**.
2. В методе **init()** апплета создать новый объект класса **MediaTracker**.
3. Каждое изображение по завершению загрузки передать на контроль объекту класса **MediaTracker** с помощью метода **addImage()** для отслеживаемого объекта.
4. Прежде, чем организовать циклическое выполнение метода **run()**, вызвать метод **waitForID()** или **waitForAll()**.
5. Прежде, чем выводить изображение на экран с помощью метода **paint()**, проверить его загрузку с помощью метода **checkID()** или **checkAll()**.

Елементи управління AWT

Наряду со средствами рисования, использования цветов и шрифтов в пакете **java.awt** определены также элементы управления.

Элементы управления (controls) – это компоненты, которые предоставляют пользователю различные способы взаимодействия с приложением. Эти элементы реализованы в следующих классах пакета **java.awt**:

- Класс **Button** – кнопки;
- Класс **Label** – текст;
- Класс **Checkbox** – флажки и переключатели;
- Класс **Choice** – раскрывающиеся списки;
- Класс **List** – списки;
- Класс **TextField** – строки ввода;
- Класс **TextArea** – поля ввода;
- Класс **ScrollBar** – полосы прокрутки;
- Класс **Canvas** – рисунки.

Интерфейс ActionListener и класс(ActionEvent)

Для классов **Button**, **List** и **TextField** определен блок прослушивания **ActionListener**

```
public void actionPerformed(ActionEvent e)
```

вызывается, когда происходит действие для заданного объекта

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

Интерфейсы ItemListener, ItemSelectable и класс ItemEvent

Для классов **Choice**, **Checkbox** и **List** определен блок прослушивания **ItemListener**

```
public void itemStateChanged(ItemEvent e)
```

вызывается, когда происходит выбор или отмена выбора элементов в объектах

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

Обработка семантических событий

Включение блока прослушивания, как и для низкоуровневых событий, выполняется с помощью соответствующего метода **add****()**. Для операции включения необходимо указать, для какого объекта AWT добавляется данный блок прослушивания, причем обычно, как и для адаптера, метод блока прослушивания переопределяется непосредственно в параметре соответствующего метода **add****()**

```
Button myButton;  
...  
myButton.addActionListener (  
new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        // Операторы, реализующие обработку          //  
нажатия кнопки myButton  
    }  
}  
);
```


Клас Label

Текст (Label) представляет собой информационную текстовую строку. Для класса **Label** в AWT не определено никаких событий.

Создание строки :

```
public Label()  
public Label(String text)  
public Label(String text, int alignment)
```

Выравнивание текста

```
public void setAlignment(int alignment)  
public int getAlignment()
```

Изменение текста в
информационной строке

```
public void setText(String text)  
public String getText()
```



Клас Button

Кнопки (Buttons) используются в панелях инструментов, диалоговых окнах, обычных окнах и в других компонентах

Создание кнопки:

```
public Button()  
public Button(String label)
```

После создания кнопки ее необходимо добавить в контейнер. Поскольку любой апплет уже является контейнером, можно добавить кнопку в апплет или графическое приложение

Изменение надписи на кнопке:

```
public void setLabel(String label)  
public String getLabel().
```

```
public void setActionCommand(String command)
```

позволяет установить имя команды для события связанного с этой кнопкой
Включение/выключения блока прослушивания событий, связанных с кнопкой:

```
public void addActionListener(ActionListener l)  
public void removeActionListener(ActionListener l).
```

Обработка события, связанного с нажатием кнопки, выполняется с помощью метода **actionPerformed()** интерфейса **ActionListener**.

Клас Checkbox

Класс **Checkbox** отображает два вида графических объектов: флажок и переключатель.

Флажок состоит из текста и состояния. Текст — это строка, которая выводится рядом с самим флажком, а состояние — логическая переменная, указывающая, установлен флажок (значение **true**) или сброшен (значение **false**). При каждом щелчке по флажку он изменяет состояние на противоположное.

Переключатель или радиокнопка (**Radiobutton**) представляет собой разновидность флажка. Переключатели также имеют текст и два состояния («включен» и «выключен»), но они всегда объединяются в группы. В любой момент может быть включен только один переключатель из группы.

Создание флажка:

```
public Checkbox()  
public Checkbox(String label)  
public Checkbox(String label, boolean state)
```

Для создания группы переключателей необходимо создать соответствующее количество переключателей и объединить их в группу. Группа создается с помощью конструктора класса **CheckboxGroup**:

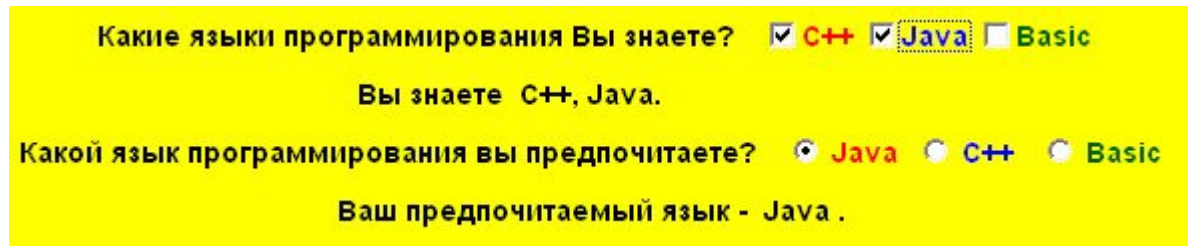
```
public CheckboxGroup()
```

Создав объект класса **CheckboxGroup**, можно помещать в эту группу переключатели, передавая объект **CheckboxGroup** в качестве параметра с помощью конструкторов класса **Checkbox**:

```
public Checkbox(String label, CheckboxGroup group, boolean state)
public Checkbox(String label, boolean state, CheckboxGroup group)
```

При объединении переключателей в группу последний из переключателей, созданных с параметром **true**, будет иметь состояние «включен», остальные — «выключен».

Если в группе задаются флажки, а не переключатели, параметр **group** должен быть равен **null**.



Какие языки программирования Вы знаете? ☒ C++ ☒ Java ☐ Basic

Вы знаете C++, Java.

Какой язык программирования вы предпочитаете? ☒ Java ☐ C++ ☐ Basic

Ваш предпочитаемый язык - Java .

Получение и установка надписи для флажков и переключателей:

```
public String getLabel()  
public void setLabel(String label)
```

Получение и установка текущего состояния флажков и переключателей:

```
public boolean getState()  
public void setState(boolean state)
```

Определение и установка группы для переключателей

```
public CheckboxGroup getCheckboxGroup()  
public void setCheckboxGroup(CheckboxGroup g)
```

Включение/выключения блока прослушивания состояния флажков или переключателей производится с помощью методов класса **Checkbox**:

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

Клас Choice

Раскрывающийся список — это компонент, обеспечивающий выбор текстовой строки из раскрывающегося меню. Выбранная строка отображается на экране. Раскрывающийся список реализуется классом **Choice**.

public Choice()

задает пустой список

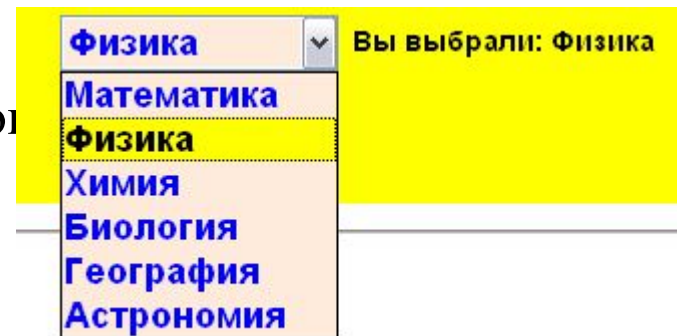
Создав раскрывающийся список, можно добавлять в него строки

public void add(String item)

public void insert(String item, int index)

После добавления всех элементов раскрывающегося списка в вывод апплета или приложения с помощью метода **add()** класса **Container** должен быть добавлен и сам список

Для обработки события **ItemEvent**, связанного с выбором элемента или элементов из раскрывающегося списка должен быть реализован интерфейс **ItemListener** и метод **itemStateChanged()**.



Клас List

Класс **List** позволяет создать **список** значений, из которых можно выбирать одно или несколько. При необходимости обеспечивается прокрутка списка.

Создание списков:

```
public List()  
public List(int rows)  
public List(int rows, boolean multipleMode)
```

Добавление элементов в список:

```
public void add(String item)
```

После добавления всех элементов раскрывающего списка в вывод апплета или приложения с помощью метода **add()** класса **Container** должен быть добавлен и сам список



Обработка событий, связанных с выбором элементов списка может производиться с помощью реализации интерфейса **ItemListener** в методе **itemStateChanged()** и/или с помощью реализации интерфейса **ActionListener** в методе **actionPerformed()**.

Класи TextField и TextArea

Для ввода строки текста AWT предоставляет компонент **поле ввода (TextField)**, представленный классом **TextField**.

```
public TextField().  
public TextField (int columns).  
public TextField(String text).  
public TextField (String text, int columns)
```

Добавление текстового поля в приложение или апплет выполняется с помощью метода **add()** класса **Container**

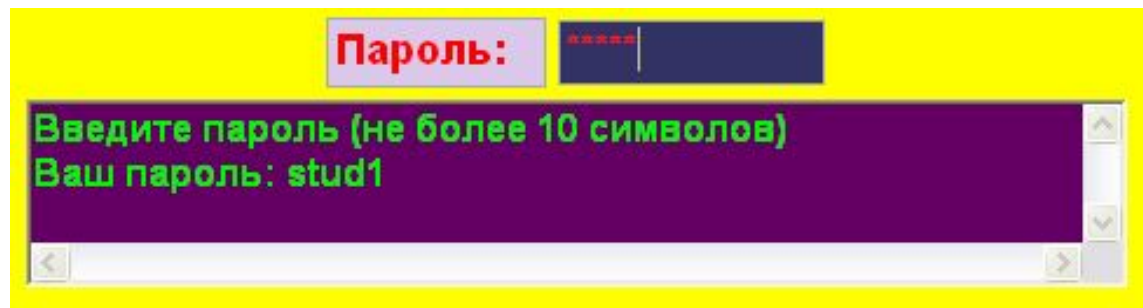
Класс **TextField** может использовать блок прослушивания **KeyListener** для обработки событий, связанных с вводом символа в текстовое поле, однако для него обычно используется интерфейс **ActionListener** для прослушивания событий, связанных с нажатием клавиши **Enter**.

Класс **TextField** может также использовать блок прослушивания **TextListener**, связанный с обработкой событий при изменении текста в поле.

Для ввода нескольких строк текста используется компонент AWT **область ввода**, представленная классом **TextArea**.

Конструкторы создания областей ввода практически совпадают с конструкторами создания поля ввода за исключением того, что при задании размера поля ввода необходимо задать не только число символов в строке, но и число строк.

```
public TextArea().  
public TextArea (String text).  
public TextArea(int rows, int columns).  
public TextArea (String text, int rows, int columns).  
public TextArea (String text, int rows, int columns, int scrollbars)
```



Клас Scrollbar

Класс **Scrollbar** обеспечивает базовые функции **прокрутки (scrolling)**. Несмотря на то, что некоторые компоненты добавляют линейки прокрутки автоматически, их можно создать явно в виде отдельных компонентов.

Управляющие элементы на полосе прокрутки управляют численным значением, определяющим текущее положение бегунка на полосе прокрутки. Программист может задавать верхний и нижний пределы прокрутки и текущее положение.

Управляющие элементы воздействуют на текущее положение с помощью трех единиц измерения:

- **Строка (Unit);**
- **Страница (Block);**
- **Абсолютная единица (Absolute).**

Кнопки со стрелками (arrow buttons) на концах полосы прокрутки воздействуют на текущее положение, используя в качестве единицы строку. Можно задать, сколько единиц будет прибавляться (или вычитаться) из текущего положения при нажатии на эти кнопки. По умолчанию это одна строка.

Абсолютные единицы применяются при перетаскивании бегунка мышью в том или ином направлении. Здесь можно задать минимальное и максимальное значение, но нельзя регулировать, на сколько страниц должно перемещаться текущее положение при заданном смещении бегунка.

Важно иметь в виду, что при прокрутке класс **Scrollbar** изменяет лишь свое состояние: он не может вызвать прокрутку каких-либо других компонентов.

```
public Scrollbar()  
public Scrollbar(int orientation)  
public Scrollbar(int orientation, int value, int visible,  
int minimum, int maximum)
```

Добавление полосы прокрутки выполняется с помощью метода **add()** класса **Container**

Для обработки событий клавиатуры должен быть реализован интерфейс блока прослушивания **AdjustmentListener** и переопределен его метод

```
public void adjustmentValueChanged(AdjustmentEvent e)
```

Обработка событий, связанных с полосой прокрутки выполняется в классе **AdjustmentEvent**.



Работа з курсором

Класс **Cursor** в пакете **java.awt** позволяет определить форму курсора

```
public Cursor(int type)
```

Кроме predefined курсоров можно задать свой курсор с типом **CUSTOM_CURSOR** с помощью метода класса **Toolkit** пакета **java.awt**

```
public Cursor createCustomCursor(Image cursor,  
Point hotSpot, String name)
```

Параметр **cursor** задает изображение, которое будет служить в качестве курсора, параметр **name** задает имя курсора (обычно задается значение **null**), параметр **hotSpot** задает точку фокуса курсора

Установить собственный курсор для какого-либо компонента, как и predefined курсор, можно с помощью метода **setCursor()**

Клас Canvas

Класс **Canvas** (рисунок) – компонент, не имеющий собственных функций. Он используется в основном для создания графических компонентов. Класс **Canvas** часто расширяется для создания новых типов компонент, например, кнопок с изображениями.

```
Canvas myCanvas = new Canvas();
```

Как правило, в программах создается производный от **Canvas** класс, который выполняет необходимые функции по рисованию.

Для рисования объектов необходимо переопределить методы **paint()** и/или **update()** класса **Canvas**

```
public class CanvasDemo extends Applet {  
    ImageButton imageButton = new ImageButton();  
    ...  
    class ImageButton extends Canvas {  
        public ImageButton () {  
            ... }  
        public void paint(Graphics g) {  
            ... } } }  
}
```

Менеджеры 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.

Интерфейс **LayoutManager** является интерфейсом в библиотеке классов Java, описывающим как класс **Container** и менеджер компоновки взаимодействуют между собой.

Методы интерфейса **LayoutManager**:

public void layoutContainer (Container parent)

располагает контейнер в заданной панели

public void addLayoutComponent(String name, Component comp)
public void removeLayoutComponent(Component comp)

добавляют и удаляют компоненты в контейнере

public Dimension minimumLayoutSize(Container parent)
public Dimension preferredLayoutSize (Container parent).

задают размеры компонентов, которыми они управляют

Дополнительный интерфейс с методами позиционирования и проверки, **LayoutManager2**, был добавлен в JDK 1.1.

В AWT определены пять различных типов менеджеров компоновки:

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

Установка менеджера компоновки для контейнера выполняется с помощью метода

```
public void setLayout(LayoutManager manager)
```

Для определения промежутков между границей контейнера и содержащихся в нем компонент используется класс **Insets**. В данном классе содержатся свойства, задающие отступ сверху, справа, снизу и слева (в пикселях)

```
public int top, public int left, public int bottom, public int  
right
```

Менеджер компоновки создает объект класса **Insets**, обратившись к конструктору класса, который возвращает объект **Insets**.

```
public Insets(int top, int left, int bottom, int right)
```

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**

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.

Менеджер FlowLayout

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

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

- учитывать предпочитаемый размер всех содержащихся компонент.
- компоновать 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 создает объект **FlowLayout**:

```
public FlowLayout(int alignment, int hgap, int vgap)
```

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**.

Пример

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



Менеджер GridLayout

Класс **GridLayout** разделяет контейнер на клетки одинакового размера (по типу таблицы). При добавлении компонентов в контейнер **GridLayout** конструктор размещает их в клетках слева направо и сверху вниз.

```
public GridLayout(int rows, int cols, int hgap, int vgap)
```

Параметры **rows** и **cols** задают количество строк или столбцов таблицы, **hgap** 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**.

Менеджер **GridLayout** старается установить размер каждого компонента к максимальной предпочитаемой ширине и максимальной предпочитаемой высоте.

Пример

```
GridLayout myGridLayout = new GridLayout(0, 2, 8, 10);
```

1	2
3	4
5	

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.

Менеджер **BorderLayout** требует ограничителя при добавлении компонент. Ограничитель может быть одной из следующих констант 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),
```

Пример:

```
add(new Button("Север"), BorderLayout.NORTH);
```

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

Менеджер действует по следующему алгоритму:

1. области **BorderLayout.NORTH** и **BorderLayout.SOUTH** являются полезными, если необходимо связать высоту компонента с его предпочитаемой высотой;
2. области **BorderLayout.EAST** и **BorderLayout.WEST** являются полезными, если необходимо связать ширину компонента с его предпочитаемой шириной;
3. как только указанные выше компоненты становятся связанными, область **BorderLayout.CENTER** становится расширяющейся частью.



Менеджер CardLayout

Класс **CardLayout** рассматривает каждый компонент в контейнере как карточку или лист блокнота. В окне видна только одна карточка (лист), т.е. контейнер функционирует как стек для карточек. Карточки упорядочены в соответствии с порядком ввода элементов каждой карточки.

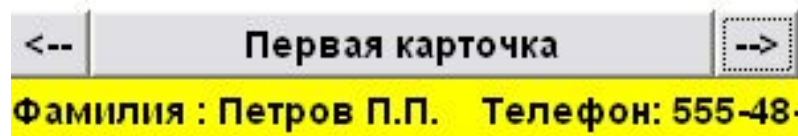
Конструктор:

```
public CardLayout ()  
public CardLayout (int hgap, int vgap)
```

Компоненты могут быть добавлены в объект **CardLayout** с использованием одного из следующих методов:

```
public void add(Component comp, Object constraints)  
public void add(Component comp, Object constraints, int index)
```

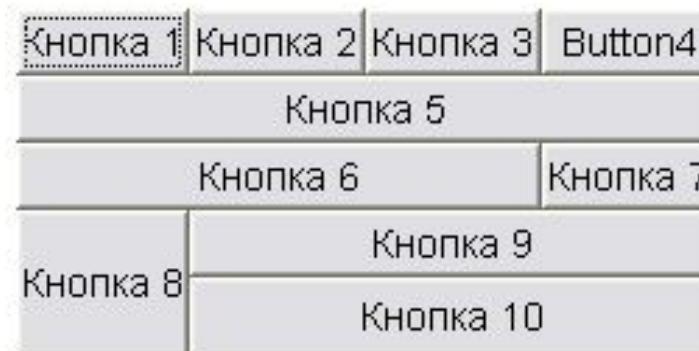
Первый метод добавляет компонент в конец списка компонент контейнера, второй — добавляет компонент в заданной позиции в контейнере.



Менеджер GridBagLayout

Класс **GridBagLayout** разделяет контейнер на клетки равного размера, как и **GridLayout**. Отличие **GridBagLayout** состоит в том, что он сам определяет, сколько требуется строк и столбцов, а также позволяет компоненту занимать при необходимости более одной клетки, фактически он может создавать табличный вывод более сложной структуры, чем менеджер **GridLayout**.

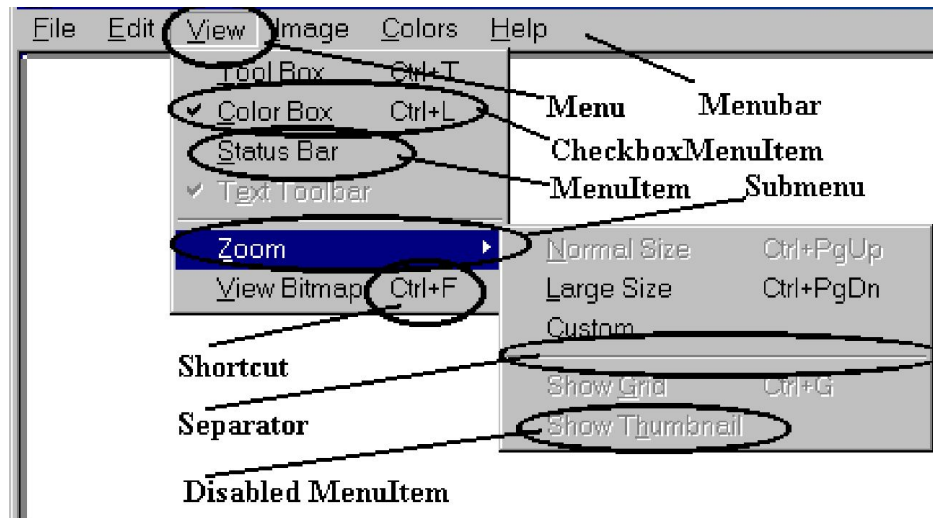
Область, занимаемая компонентом, называется его **ячейкой (display area)**. Прежде чем добавить компонент в контейнер, надо задать менеджеру **GridBagLayout** набор требований по размещению компонент. Эти требования выдаются в форме объекта **GridBagConstraints**.



Класс **GridBagConstraints** содержит ряд **public** переменных типа **int**, управляющих размещением компонента:

- **GridBagConstraints.gridx** и **GridBagConstraints.gridy**. Координаты клетки, куда будет помещен следующий компонент;
- **GridBagConstraints.gridwidth** — число клеток, которое занимает компонент по горизонтали и **GridBagConstraints.gridheight** — число клеток, которое занимает компонент по вертикали;
- **GridBagConstraints.fill** — сообщает объекту **GridBagLayout**, как поступить, если компонент меньше, чем предоставленная ему ячейка;
- **GridBagConstraints.ipadx** и **GridBagConstraints.ipady**. Указывает, сколько пикселей добавить к размерам компонента по осям x и y;
- **GridBagConstraints.insets**. Указывает, сколько места нужно оставить между границами компонента и краями ячейки;
- **GridBagConstraints.anchor**. Указывает, где должен располагаться компонент в пределах ячейки тогда, когда его размеры меньше размеров ячейки.

Створення меню і обробка подій в меню



Основные компоненты
меню в AWT Java

Чтобы у фрейма появилось раскрывающееся меню, с фреймом следует связать объект класса **MenuBar**. Он создается конструктором

```
public MenuBar()
```

Создав строку меню, ее можно добавить в фрейм

```
public void setMenuBar(MenuBar mb)
```

Раскрывающиеся меню являются объектами класса **Menu** и добавляются к объекту класса **MenuBar** с помощью метода класса **MenuBar**

```
public Menu add(Menu newMenu)
```

Пример

```
MenuBar myMenuBar = new MenuBar();  
myFrame.setMenuBar(myMenuBar);  
Menu fileMenu = new Menu("File");  
myMenuBar.add(fileMenu);
```

Некоторые оконные системы позволяют создавать меню, которые остаются на экране после того, как кнопка мыши отпущена. Такие меню называются **отрывными меню** (tear-off menu). При создании меню можно указать, что оно должно вести себя как отрывное с помощью конструктора класса **Menu**.

```
public Menu()  
public Menu(String label)  
public Menu(String menuLabel, boolean allowTearoff)
```

В каждом добавленном меню можно определять команды или пункты меню. **Пункт меню** (menu item) — это элемент меню, который выбирает пользователь. Меню может содержать отдельные пункты, а также подменю. Пункт меню можно задать либо как объект класса **String**, либо как объект класса **MenuItem**.

```
public void add(String label)  
public MenuItem add(MenuItem mi)
```

Вставить разделитель между пунктами меню можно также, задав в качестве текста для пункта меню строку "-".

В классе **MenuItem** определены следующие конструкторы

```
MenuItem()  
MenuItem(String label)  
MenuItem(String label, MenuShortcut s)
```

Третий конструктор дополнительно задает во втором параметре **«горячие»** или **«быстрые» клавиши** (shortcuts), которые позволяют выбрать пункт меню при нажатии этих клавиш.

Конструкторы класса **MenuShortcut**:

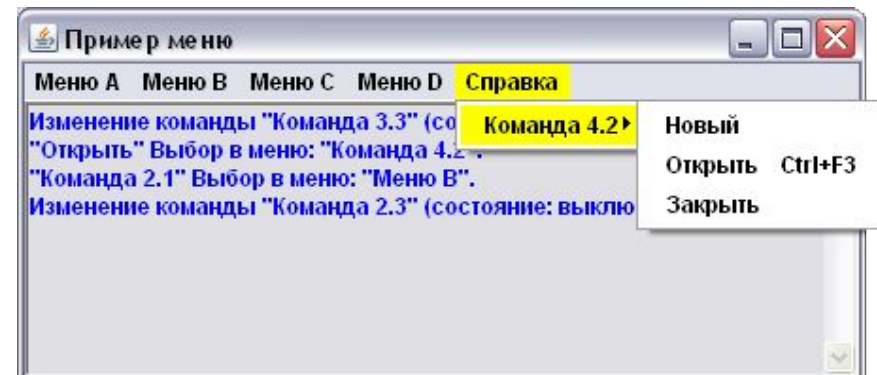
```
public MenuShortcut(int key)
public MenuShortcut(int key, boolean useShiftModifier)
```

Второй конструктор дополнительно задает реакцию на одновременное нажатие клавиши **key** и клавиши **Shift**, т.е. различать ли нажатие данной клавиши в верхнем или нижнем регистре. В качестве значения **key** задаются значения полей класса **KeyEvent**.

Пример

```
MenuItem saveMenuItem =
new MenuItem("Save", new MenuShortcut(KeyEvent.VK_S));
fileMenu.add(saveMenuItem);
```

Кроме отдельных пунктов и элементов-разделителей, можно также добавлять в меню **подменю** (submenu). Чтобы добавить подменю, достаточно создать новое меню и добавить его в существующее меню



Помимо обычных пунктов меню, можно создавать **пункты меню с флажком** (checkbox menu item). Такие пункты меню действуют аналогично флажкам. Когда пункт выбран первый раз, он становится включенным. В следующий раз при выборе пункт становится выключенным. Для создания пунктов меню с флажком используется отдельный класс **CheckboxMenuItem**. В этом классе определены три конструктора:

```
CheckboxMenuItem()  
CheckboxMenuItem(String itemLabel)  
CheckboxMenuItem(String itemLabel, boolean state)
```

Как правило, меню добавляются в главное меню слева направо. В то же время, во многих оконных системах предусмотрена возможность создавать специальное меню "**Справка**" ("**Help**"), расположенное в главном меню справа. Такое меню можно добавить, используя метод

```
public void setHelpMenu(Menu helpMenu)
```


Меню генерирует события только тогда, когда выбираются пункты меню. Однако добавление блоков прослушивания и обработку событий можно задавать как для меню, так и для отдельных пунктов меню, поскольку методы включения и удаления блоков прослушивания определены как в классе **Menu**, так и в классе **MenuItem**.

```
addActionListener(ActionListener l)  
removeActionListener(ActionListener l)
```

Обработка события, связанного с выбором меню или пункта меню выполняется с помощью метода

```
public void actionPerformed(ActionEvent e)
```

События в пунктах меню класса **CheckboxMenuItem** обрабатываются как события класса **ItemEvent**, для них надо добавлять или удалять блоки прослушивания с помощью методов

```
addItemListener(ItemListener l)  
removeItemListener(ItemListener l).
```

Обработка события в этом случае выполняется с помощью метода

```
public void itemStateChanged(ItemEvent e)
```

Клас PopupMenu

Меню, созданное с помощью класса **PopupMenu** («всплывающее» меню) работает точно так же, как и обычное меню, но выводится при щелчке мышью в какой-либо точке окна фрейма (возможно при выполнении заданных условий). В Windows такого рода меню называется контекстным меню (shortcut menu) и вызывается щелчком правой кнопки мыши.

Для класса **PopupMenu** определены конструкторы

```
public PopupMenu()  
public PopupMenu(String label)
```

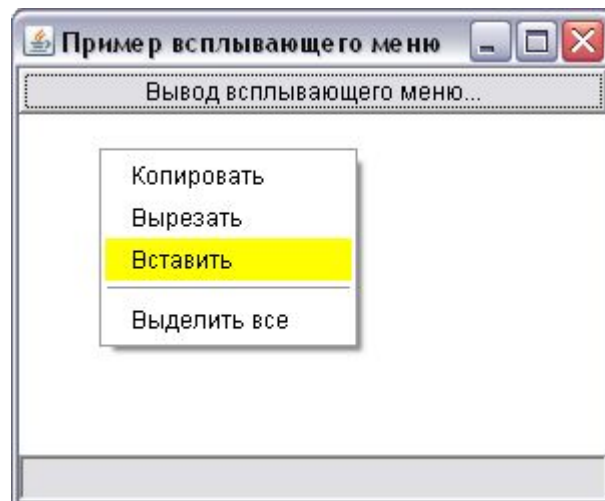
Поскольку класс **PopupMenu** является подклассом класса **Menu**, он наследует все методы класса **Menu**, в том числе и для манипуляций с объектами классов **MenuItem** и **CheckboxMenuItem**. В классе добавлен метод показывающий «всплывающее» меню в точке с координатами **x** и **y** относительно компонента **origin**:

```
public void show(Component origin, int x, int y),
```

Поскольку «всплывающее» меню обычно появляется в окне при щелчке мышью, необходимо использовать блоки прослушивания мыши, добавляемые и удаляемые с помощью методов

addListener(MouseListener I)
removeMouseListener(MouseListener I)

Обработка событий выбора обычных пунктов меню и меню с флажками производится точно также, как и в обычном меню.



Клас Dialog

Диалоговые окна (Dialogs) представляют собой раскрывающиеся окна, которые не настолько универсальны, как фреймы.

Можно создать *модальное (modal)* и *немодальное (non-modal)* диалоговое окно. **Если на экране появилось модальное диалоговое окно, ввод в другие окна временно запрещается.** Это полезно при создании диалогов, требующих ответа на принципиальный вопрос, такой как "Завершить работу?". Примером немодального диалога может служить панель управления, которая изменяет некоторые параметры приложения, в то время как программа продолжает выполняться.

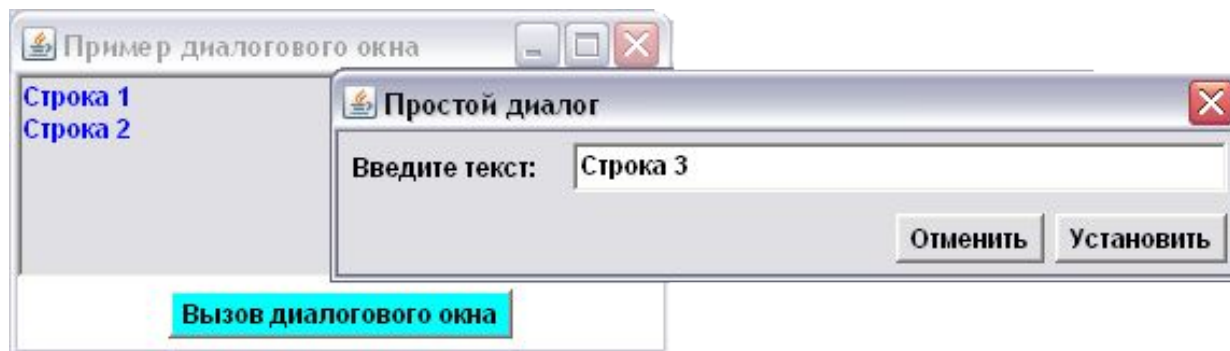
Чтобы создать диалоговое окно, необходимо иметь фрейм. Диалоговое окно не может принадлежать непосредственно апплету, но апплет может создать фрейм, а затем диалоговое окно.

Диалоговое окно создается с помощью одного из следующих конструкторов:

```
public Dialog(Dialog parentDialog)
public Dialog(Dialog parentDialog, String title)
public Dialog(Dialog parentDialog, String title, boolean isModal)
public Dialog(Frame parentFrame)
public Dialog(Frame parentFrame, String title)
public Dialog(Frame parentFrame, boolean isModal)
```

Пример

```
Dialog myDialog = new Dialog(myFrame, "My Dialog", true);
myDialog.show();
```



Клас **FileDialog**

В состав AWT включен класс **FileDialog**, который обеспечивает встроенное диалоговое окно, позволяющее выбрать определенный файл. По форме это стандартное модальное диалоговое окно, используемое операционной системой для открытия файлов.

После выбора файла или нажатия кнопки **Cancel** файловое диалоговое окно автоматически закрывается.

Для открытия этого окна достаточно создать объект класса **FileDialog** с помощью одного из следующих конструкторов:

FileDialog(Frame parent)
FileDialog(Frame parent, String title)
FileDialog(Frame parent, String title, int mode)

Параметр **parent** задает родительский фрейм, параметр **title** — имя файлового диалогового окна, а параметр **mode** — режим диалогового окна. Если **mode** имеет значение **FileDialog.LOAD**, то окно выбирает файл для чтения, если **FileDialog.SAVE** — то для записи (с целью сохранения).