**Библиотека Swing**

**Библиотека Swing**

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

Каждый из современных языков программирования предоставляет множество библиотек для работы со стандартным набором элементов управления. Напомним, что под библиотекой в программировании набор готовых классов и интерфейсов, предназначенных для решения определенного круга задач.

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

Каждая библиотека предоставляет набор классов для работы с кнопками, списками, окнами, меню и т.д., но эти классы спроектированы по-разному: они имеют различный набор методов с разными параметрами, поэтому «перевести» программу с одной библиотеки на другую (например, с целью увеличения быстродействия) не так-то просто. Это почти как перейти с одного языка программирования на другой: все языки умеют делать одно и то же, но у каждого из них свой синтаксис, своя программная структура и свои многочисленные хитрости.

По этой причине вместо того, чтобы делать обзор всех трех библиотек, мы постараемся получше разобраться в одной из них — библиотеке Swing. Полноценный графический интерфейс может быть разработан с ее помощью.

**Окно JFrame**

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

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

*об окнах Swing*

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

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

setSize(int width, int height) — устанавливает размеры окна. Если не задать размеры, окно будет иметь нулевую высоту независимо от того, что в нем находится и пользователю после запуска придется растягивать окно вручную. Размеры окна включают не только «рабочую» область, но и границы и строку заголовка.

setDefaultCloseOperation(int operation) — позволяет указать действие, которое необходимо выполнить, когда пользователь закрывает окно нажатием на крестик. Обычно в программе есть одно или несколько окон при закрытии которых программа прекращает работу. Для того, чтобы запрограммировать это поведение, следует в качестве параметра operation передать константу EXIT\_ON\_CLOSE, описанную в классе JFrame.

setVisible(boolean visible) — когда окно создается, оно по умолчанию невидимо. Чтобы отобразить окно на экране, вызывается данный метод с параметром **true**. Если вызвать его с параметром **false**, окно снова станет невидимым.

Теперь мы можем написать программу, которая создает окно, выводит его на экран и завершает работу после того, как пользователь закрывает окно.

**import** javax.swing.\*;

**public class** MyClass {

**public static void** main (**String** [] args) {

JFrame myWindow = **new** JFrame("Пробное окно");

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

myWindow.setSize(400, 300);

myWindow.setVisible(**true**);

}

}

Обратите внимание, для работы с большинством классов библиотеки Swing понадобится импортировать пакет java.swing.**\***

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

*В файле SimpleWindow.java:*

**public class** SimpleWindow **extends** JFrame {

SimpleWindow(){

**super**("Пробное окно");

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

setSize(250, 100);

}

}

*В файле Program.java:*

**public class** Program {

**public static void** main (**String** [] args) {

JFrame myWindow = **new** SimpleWindow();

myWindow.setVisible(**true**);

}

}

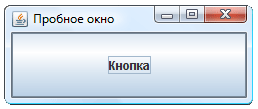
Из примера видно, что окно описывается в отдельном классе, являющемся наследником JFrame и настраивающее свой внешний вид и поведение в конструкторе (первой командой вызывается конструктор суперкласса). Метод main() содержится в другом классе, ответственном за управление ходом программы. Каждый из этих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

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

**Панель содержимого**

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

В примерах этого занятия мы будем использовать только один элемент управления — кнопку (не вдаваясь в подробности ее устройства). Кнопка описывается классом JButton и создается конструктором с параметром типа String — надписью.

Добавим кнопку в панель содержимого нашего окна командами:

JButton newButton = **new** JButton();

getContentPane().add(newButton);

В результате получим окно с кнопкой. Кнопка занимает всю доступную площадь окна. Такой эффект полезен не во всех программах, поэтому необходимо изучить различные способы расположения элементов на панели.

**Класс Container (контейнер)**

Элементы, которые содержат другие элементы, называются контейнерами. Все они являются потомками класса Container и наследуют от него ряд полезных методов:

add(Component component) — добавляет в контейнер элемент component;

remove(Component component) — удаляет из контейнера элемент component;

removeAll() — удаляет все элементы контейнера;

getComponentCount() — возвращает число элементов контейнера.

Кроме перечисленных в классе Container определено около двух десятков методов для управления набором компонентов, содержащихся в контейнере. Как видно, они похожи на методы класса-коллекции. Это неудивительно, ведь по сути контейнер и является коллекцией, но коллекцией особого рода — визуальной. Кроме хранения элементов контейнер занимается их пространственным расположением и прорисовкой. В частности, он имеет метод getComponentAt(int x, int y), возвращающий компонент, в который попадает точка с заданными координатами (координаты отсчитываются от левого верхнего угла компонента) и ряд других. Мы не будем подробно рассматривать абстрактный контейнер, а сразу перейдем к его наиболее часто используемому потомку — классу JPanel.

**Класс JPanel (панель)**

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

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

**Менеджер последовательного размещения FlowLayout**

Самый простой менеджер размещения — FlowLayout. Он размещает добавляемые на панель компоненты строго по очереди, строка за строкой, в зависимости от размеров панели. Как только очередной элемент не помещается в текущей строке, он переносится на следующую. Лучше всего пронаблюдать это на примере. Изменим конструктор класса SimpleWindow следующим образом:

SimpleWindow(){

**super**("Пробное окно");

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

JPanel panel = **new** JPanel();

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

panel.add(**new** JButton("Кнопка"));

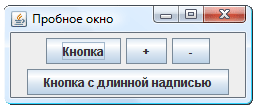
panel.add(**new** JButton("+"));

panel.add(**new** JButton("-"));

panel.add(**new** JButton("Кнопка с длинной надписью"));

setContentPane(panel);setSize(250, 100);

}

Менеджеры расположения описаны в пакете java.awt. Не забывайте импортировать нужные классы.

Пронаблюдайте за поведением окна, появляющегося после запуска программы. Четыре кнопки в нем расположены как слова в текстовом редакторе (при выравнивании по центру). Эффект будет лучше заметен, если изменять размеры окна во время работы программы.

Проанализируем текст примера. Новый менеджер расположения FlowLayout создается конструктором без параметров. Обратите внимание, в программе не используется промежуточная переменная. То есть вместо двух команд:

FlowLayout newLayout = **new** FlowLayout();

panel.setLayout(newLayout);

Мы используем одну:

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

Это вполне допустимо в тех случаях, когда в дальнейшем нам не потребуется обращаться к создаваемому объекту (что справедливо для данного примера). Мы создаем менеджер расположения, тут же привязываем его к панели — и все. Теперь панель и менеджер сами найдут друг с другом общий язык.

*о взаимоотношениях панели и ее менеджера*

Точно также мы добавляем на панель новые кнопки. Мы нигде больше не пытаемся обратиться к этим кнопкам в программе, поэтому заводить под них переменные нет смысла.

Метод setContentPane(JPanel panel) позволяет заменить панель содержимого окна.

**Менеджер граничного размещения BorderLayout**

Менеджер размещения BorderLayout разделяет панель на пять областей: центральную, верхнюю, нижнюю, правую и левую. В каждую из этих областей можно добавить ровно по одному компоненту, причем компонент будет занимать всю отведенную для него область. Компоненты, добавленные в верхнюю и нижнюю области, будут растянуты по ширине, добавленные в правую и левую — по высоте, а компонент, добавленный в центр, будет растянут так, чтобы полностью заполнить оставшееся пространство панели.

При добавлении элемента на панель с менеджером размещения BorderLayout, необходимо дополнительно указывать в методе add(), какая из областей имеется в виду. Для этого служат строки с названиями сторон света: "North", "South", "East", "West" и "Center". Но вместо них рекомендуется использовать константы, определенные в классе BorderLayout: NORTH, SOUTH, EAST, WEST и CENTER (поскольку в строке можно допустить ошибку и не заметить этого, а при попытке написать неправильно имя константы компилятор выдаст предупреждение). Если же использовать метод add() как обычно, с одним параметром, элемент будет добавлен в центр.

Панель содержимого имеет именно такое расположение, именно поэтому кнопка и занимала все окно целиком (она была добавлена в центральную область). Чтобы пронаблюдать эффект BorderLayout, добавим кнопки во все пять областей:

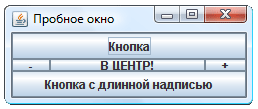
SimpleWindow(){

super("Пробное окно");

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

getContentPane().add(**new** JButton("Кнопка"), BorderLayout.NORTH);getContentPane().add(**new** JButton("+"), BorderLayout.EAST);getContentPane().add(**new** JButton("-"), BorderLayout.WEST);getContentPane().add(**new** JButton("Кнопка с длинной надписью"), BorderLayout.SOUTH);getContentPane().add(**new** JButton("В ЦЕНТР!"));setSize(250, 100);

}

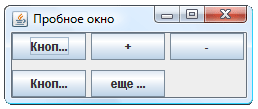
Эффект будет хорошо наблюдаться, если изменять размеры окна.

Данное размещение не случайно используется в панели содержимого по умолчанию. Большинство программ пользуются областями по краям окна, чтобы расположить в них панели инструментов, строку состояния и т.п. А ограничение на один компонент в центральной области абсолютно не существенно, ведь этим компонентом может быть другая панель со множеством элементов и с любым менеджером расположения.

**Менеджер табличного размещения GridLayout**

GridLayout разбивает панель на ячейки одинаковой ширины и высоты (таким образом окно становится похожим на таблицу). Каждый элемент, добавляемый на панель с таким расположением, целиком занимает одну ячейку. Ячейки заполняются элементами по очереди, начиная с левой верхней.

Этот менеджер, в отличие от рассмотренных ранее, создается конструктором с параметрами (четыре целых числа). Необходимо указать количество столбцов, строк и расстояние между ячейками по горизонтали и по вертикали. Выполните следующий пример и пронаблюдайте эффект.

SimpleWindow(){

super("Пробное окно");

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

JPanel panel = **new** JPanel();

panel.setLayout(**new** GridLayout(2,3,5,10));

panel.add(**new** JButton("Кнопка"));

panel.add(**new** JButton("+"));

panel.add(**new** JButton("-"));

panel.add(**new** JButton("Кнопка с длинной надписью"));

panel.add(**new** JButton("еще кнопка"));

setContentPane(panel);setSize(250, 100);

}

**Менеджер блочного размещения BoxLayout и класс Box**

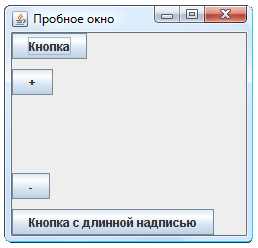
Менеджер BoxLayout размещает элементы на панели в строку или в столбец.

Обычно для работы с этим менеджером используют вспомогательный класс Box, представляющий собой панель, для которой уже настроено блочное размещение. Создается такая панель не конструктором, а одним из двух статических методов, определенных в классе Box: createHorizontalBox() и createVerticalBox().

Элементы, добавленные на панель с блочным размещением, выстраиваются один за другим. Расстояние между элементами по умолчанию нулевое. Однако вместо компонента можно добавить невидимую «распорку», единственная задача которой — раздвигать соседние элементы, обеспечивая между ними заданное расстояние. Горизонтальная распорка создается статическим методом createHorizontalStrut(int width), а вертикальная — методом createVerticalStrut(int height). Оба метода определены в классе Box, а целочисленный параметр в каждом из них определяет размер распорки.

Кроме того, на такую панель можно добавить еще один специальный элемент — своеобразную «пружину». Если размер панели будет больше, чем необходимо для оптимального размещения всех элементов, те из них, которые способны растягиваться, будут стараться заполнить дополнительное пространство собой. Если же разместить среди элементов одну или несколько «пружин», дополнительное свободное пространство будет распределяться и в эти промежутки между элементами. Горизонтальная и вертикальная пружины создаются соответственно методами createHorizontalGlue() и createVerticalGlue().

Понять особенности работы этого менеджера лучше на наглядном примере. Мы расположим четыре кнопки вертикально, поставив между двумя центральными «пружину», а между остальными — распорки в 10 пикселов.

SimpleWindow(){

super("Пробное окно");

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

Box box = Box.createVerticalBox();

box.add(**new** JButton("Кнопка"));

box.add(Box.createVerticalStrut(10));

box.add(**new** JButton("+"));

box.add(Box.createVerticalGlue());

box.add(**new** JButton("-"));

box.add(Box.createVerticalStrut(10));

box.add(**new** JButton("Кнопка с длинной надписью"));

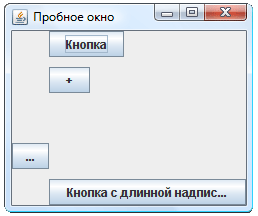
setContentPane(box);setSize(250, 100);

}

**Особенности выравнивания элементов**

В примере с вертикальной панелью все кнопки оказались выровнены по левому краю. Такое выравнивание по горизонтали принято по умолчанию.

Однако при разработке окна программы может понадобиться, чтобы какие-то элементы были выровнены иначе, например, по правому краю или по центру. Для того, чтобы установить выравнивание любого визуального компонента (например, кнопки или панели), используются методы setAlignmentX(float alignment) — выравнивание по горизонтали и setAlignmentY(float alignment) — выравнивание по вертикали. В качестве параметра проще всего использовать константы, определенные в классе JComponent. Для выравнивания по горизонтали служат константы LEFT\_ALIGNMENT (по левому краю), RIGHT\_ALIGNMENT (по правому краю) и CENTER\_ALIGNMENT (по центру). Для выравнивания по вертикали — BOTTOM\_ALIGNMENT (по нижнему краю), TOP\_ALIGNMENT (по верхнему краю) и CENTER\_ALIGNMENT (по центру).

Однако выравнивание работает несколько иначе, чем ожидается. Чтобы это обнаружить, изменим предыдущий пример, выровняв третью кнопку по правому краю. Для этого заменим строку:

box.add(**new** JButton("-"));

На три других:

JButton rightButton = **new** JButton("-");

rightButton.setAlignmentX(JComponent.RIGHT\_ALIGNMENT);

box.add(rightButton);

Нам пришлось ввести переменную для обращения к этой кнопке, поскольку теперь нам нужно выполнить с ней не одно, а два действия: установка выравнивания по правому краю и добавление в панель. Прежний прием — одновременное создание кнопки и передача ее в качестве параметра в метод — здесь не сработает.

После запуска программы мы увидим окно, в котором кнопки расположены не так, как, наверное, ожидалось. Мы привыкли, что выравнивание по правому краю прижимает объект к правому краю контейнера, но в данном случае перестроились все элементы, причем кнопка с выравниванием по правому краю оказалась самой левой.

Объяснение просто. При выравнивании по правому краю объект не прижимается к правому краю компонента. Вместо этого он прижимается правым краем к невидимой линии выравнивания. Все остальные компоненты прижимаются к этой линии своим левым краем, поэтому и получается наблюдаемый эффект.

Единственная трудность для начинающего разработчика может оказаться в том, что не всегда легко понять, где именно пройдет эта линия. Ее положение зависит от размеров и выравнивания всех элементов контейнера. Однако легко запомнить простое правило: если все элементы в контейнере выровнены одинаково, мы получим привычное поведение (как это и было в предыдущем примере, когда все компоненты были выровнены влево и линия в результате прижалась к левому краю панели.

*о выравнивании элементов*

Попробуйте поэкспериментировать с вертикальной панелью, задавая различное выравнивание для ее элементов, чтобы интуитивно понять логику размещения линии выравнивания. Изменяйте размеры окна во время работы программы, чтобы увидеть как меняется положение этой линии.

**Ручное размещение элементов**

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

Координаты элемента можно задать одним из следующих методов:

setLocation(int x, int y),

setLocation(Point point)

Эти методы работают аналогично, устанавливая левый верхний угол элемента в точку с заданными координатами. Разница в способе задания точки. Можно представить точку двумя целыми числами, а можно объектом класса Point. Класс Point по сути представляет собой ту же пару чисел, его конструктор имеет вид Point(int x, int y). Получить доступ к отдельной координате можно методами getX() и getY().

Можно задаться вопросом: зачем использовать класс Point, если можно просто передать пару чисел? Но дело в том, что многие полезные методы возвращают результат — координаты некоторой точки — в виде объекта этого класса. Например, метод getLocation(), возвращающий координаты элемента. Предположим, нам нужно поместить элемент b в точности в то место, которое занимает элемент a. Этого легко добиться одной строкой:

b.setLocation(a.getLocation());

Размер элемента задается одним из двух методов:

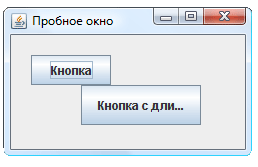
setSize(int width, int height),

setSize(Dimension size)

Эти методы работают одинаково — разница, как и в прошлый раз, в способе передачи параметра. Класс Dimension, аналогично классу Point, просто хранит два числа, имеет конструктор с двумя параметрами: Dimension(int width, int height) и позволяет получить доступ к своим составляющим — ширине и высоте — с помощью простых методов getWidth() и getHeigth(). Для того, чтобы получить текущий размер элемента, можно воспользоваться методом getSize(), возвращающего объект класса Dimension. Элемент b можно сделать точно такого же размера, как элемент a, выполнив команду:

b.setSize(a.getSize());

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

SimpleWindow(){

super("Пробное окно");

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

JPanel panel = **new** JPanel();

panel.setLayout(**null**);

JButton button = **new** JButton("Кнопка");

button.setSize(80, 30);

button.setLocation(20,20);

panel.add(button);

button = **new** JButton("Кнопка с длинной надписью");

button.setSize(120, 40);

button.setLocation(70,50);

panel.add(button);

setContentPane(panel);

setSize(250, 150);

}

Мы используем одну и ту же переменную button для обращения к обеим кнопкам (причем, второй раз ее описывать не нужно). В самом деле, осуществив все необходимые операции с первой кнопкой и зная, что обращаться к ней нам больше не понадобится, мы используем «освободившуюся» переменную для манипуляций со второй.

**Автоматическое определение размеров компонентов**

Если у панели есть любой менеджер размещения, она игнорирует явно заданные размеры и координаты всех своих элементов. В этом легко убедиться, заменив в предыдущем примере команду panel.setLayout(**null**) на panel.setLayout(**new** FlowLayout()). Менеджер размещения сам определяет координаты и размеры всех элементов.

Способ определения координат элементов очевидным образом вытекает из алгоритмов работы каждого менеджера и, таким образом, детально рассмотрен нами выше.

Мы также отмечали, что в некоторых случаях компоненты стараются заполнить все доступное им пространство. Например, всю центральную область в случае менеджера BorderLayout или всю ячейку в менеджере GridLayout. А в панели с менеджером FlowLayout, напротив, элементы никогда не пытаются выйти за определенные границы. Рассмотрим, что это за границы.

Каждый визуальный компонент имеет три типа размеров: минимально допустимый, максимально допустимый и предпочтительный. Узнать, чему равны эти размеры для данного компонента можно с помощью соответствующих методов:

getMinimumSize(),

getPreferredSize(),

getMaximumSize().

Методы возвращают результат типа Dimension. Они запрограммированы в соответствующем классе. Например, у кнопки минимальный размер — нулевой, максимальный размер не ограничен, а предпочтительный зависит от надписи на кнопке (вычисляется как размер текста надписи плюс размеры полей).

Менеджер FlowLayout всегда устанавливает предпочтительные размеры элементов. Менеджер BorderLayout устанавливает предпочтительную ширину правого и левого, а также предпочтительную высоту верхнего и нижнего. Остальные размеры подгоняются под доступное пространство панели. Менеджер GridLayout пытается подогнать размеры всех элементов под размер ячеек. Менеджер BoxLayout ориентируется на предпочтительные размеры.

Когда элемент старается занять все доступное ему пространство, он «учитывает» пожелания не делаться меньше своих минимальных или больше максимальных.

Всеми тремя размерами можно управлять с помощью соответствующим методов set:

setMinimumSize(Dimension size),

setPreferredSize(Dimension size),

setMaximumSize(Dimension size).

Чаще всего используется простой прием, когда элементу «не рекомендуется» увеличиваться или уменьшаться относительно своих предпочтительных размеров. Это легко сделать командой:

element.setMinimumSize(element.getPreferredSize());

**«Упаковка» окна**

В рассмотренных выше примерах мы явно задавали размер окна методом setSize(). Но когда используется какой-либо менеджер расположения, расставляющий элементы и изменяющий их размеры по собственным правилам, трудно сказать заранее, какие размеры окна будут самыми подходящими.

Безусловно, наиболее подходящим будет вариант, при котором все элементы окна имеют предпочтительные размеры или близкие к ним**\***.

Если вместо явного указания размеров окна, вызвать метод pack(), они будут подобраны оптимальным образом с учетом предпочтений всех элементов, размещенных в этом окне.

Оцените работу этого метода, заменив в каждом из вышеприведенных примеров команду

setSize(250, 100);

на команду

pack();

Заметьте, что когда панель не имеет метода размещения, эта команда не работает (поскольку панель не имеет алгоритма для вычисления своего предпочтительного размера).

**Упражнение**

Как уже отмечалось, элементом панели может быть другая панель. Создайте панель с тремя кнопками и менеджером размещения FlowLayout и панель с двумя кнопками и менеджером размещения BoxLayout (горизонтальным). Разместите обе панели в главном окне (не изменяя менеджера размещения у панели содержимого): одну в центр, а другую вдоль любой стороны окна.

**Рамки**

Когда панели служат не просто для размещения элементов в соответствии с алгоритмом некоторого менеджера, а для визуального отделения их друг от друга, они оформляются с помощью рамок.

Рамка панели устанавливается методом setBorder(Border border). Параметром метода выступает рамка — объект класса Border. Это абстрактный класс, поэтому для создания рамки используются его наследники:

EmptyBorder — пустая рамка, позволяет создать отступы вокруг панели. Размеры отступов задаются в конструкторе четырьмя целыми числами.

TitledBorder — рамка с заголовком. Простейший конструктор имеет один параметр типа String (текст заголовка). Заголовок может размещаться вдоль любой стороны рамки, иметь различные начертания.

EtchedBorder — рамка с тиснением. Может быть вогнутой или выпуклой.

BevelBorder — объемная рамка (выпуклая или вогнутая). Можно настроить цвета, требуемые для получения объемных эффектов.

SoftBevelBorder — то же самое, что BevelBorder, но позволяет дополнительно скруглить углы.

LineBorder — простая рамка, нарисованная сплошной линией. Можно выбирать цвет и толщину линии, скруглить углы.

MatteBorder — рамка из повторяющегося рисунка.

CompoundBorder — объединяет две рамки, передаваемые в качестве параметров конструктору в одну новую рамку.

Все перечисленные классы описаны в пакете javax.swing.border.

Рассмотрим пример. В этом примере мы создадим шесть панелей с различными рамками и разместим их в виде таблицы. Чтобы не описывать шесть раз процедуру создания новой панели, вынесем ее в отдельный метод:

private JPanel createPanel(Border border, String text) {

JPanel panel = **new** JPanel();

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

panel.add(**new** JButton(text));

panel.setBorder(**new** CompoundBorder(**new** EmptyBorder(12,12,12,12), border));

return panel;

}

Метод createPanel() создает панель с кнопкой во весь свой размер. В качестве параметра передается надпись на кнопке и рамка, которую необходимо добавить к панели. Рамка добавляется не напрямую, а путем композиции с пустой рамкой. Этот прием часто используется, чтобы рамка не прилипала к краю панели.

Теперь шесть раз воспользуемся этим методом в конструкторе окна программы.

SimpleWindow(){

super("Пробное окно");

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

JPanel panel = **new** JPanel();

panel.setLayout(**new** GridLayout(2,3,5,10));

panel.add(createPanel(**new** TitledBorder("Рамка с заголовком"), "TitledBorder"));

panel.add(createPanel(**new** EtchedBorder(), "EtchedBorder"));

panel.add(createPanel(**new** BevelBorder(BevelBorder.LOWERED), "BevelBorder"));

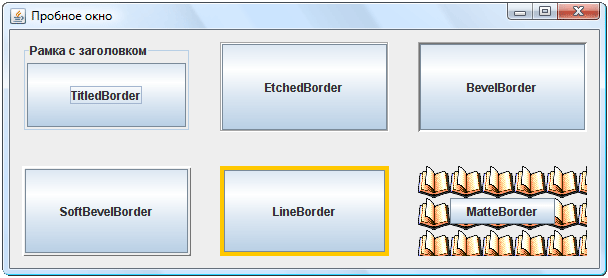
panel.add(createPanel(**new** SoftBevelBorder(BevelBorder.RAISED), "SoftBevelBorder"));

panel.add(createPanel(**new** LineBorder(Color.ORANGE, 4), "LineBorder"));

panel.add(createPanel(**new** MatteBorder(**new** ImageIcon("1.gif")), "MatteBorder"));

setContentPane(panel);pack();

}



Этот пример показывает, с помощью каких конструкторов создаются различные рамки и как они выглядят. В нем использованы два новых класса: Color и ImageIcon.

Класс Color предназначен для работы с цветом. В нем есть несколько констант, описывающих наиболее распространенные цвета. В частности, к таковым относится Color.ORANGE.

Класс ImageIcon описывает графическое изображение. Параметр его конструктора — это путь к файлу, из которого изображение может быть загружено. В примере используется относительное имя файла «1.gif». Чтобы объект ImageIcon был успешно создан, файл с таким именем должен быть помещен в папку проекта.

**Основные визуальные компоненты Swing**

**Класс JComponent**

Все визуальные компоненты библиотеки Swing унаследованы от класса JComponent. Сам этот класс является абстрактными и непосредственно не используется, но все визуальные компоненты наследуют его методы. Рассмотрим наиболее полезные из них.

setEnabled(boolean enabled) используется для управления активностью компонента. При вызове этого метода с параметром **false** компонент переходит в неактивное состояние. Для каждого наследника JComponent эта «неактивность» может быть переопределена по-разному. Например, неактивная кнопка не нажимается, не реагирует на наводящуюся мышь и отображается монохромным серым цветом. Метод isEnabled() возвращает **true**, если элемент активен и **false** в противном случае.

setVisible(boolean visible) управляет видимостью компонента. Мы уже использовали его для отображения окна JFrame. Большинство элементов управления, в отличие от окна, по умолчанию являются видимыми (поэтому мы не вызывали данный метод после создания кнопок в примерах предыдущей главы). Метод isVisible() возвращает **false**, если элемент невидим и true в противном случае.

С помощью метода setBackground(Color color) можно изменить цвет заднего фона компонента. Однако эффект будет иметь место лишь в том случае, если компонент непрозрачен (некоторые компоненты, например метка JLabel по умолчанию являются прозрачными). Непрозрачность устанавливается методом setOpaque(boolean opaque) с параметром **true**. Методы getBackground() и isOpaque() возвращают текущий цвет заднего фона и непрозрачность компонента.

**Метка JLabel**

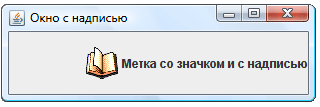
В большинстве визуальных библиотек *метка* — один из самых простейших компонентов. Она представляет собой обычный текст, который выводится в заданном месте окна и используется для вывода вспомогательной текстовой информации: подписи к другим элементам, инструкции и предупреждения для пользователя. В Swing метка позволяет достичь более интересных эффектов. Во-первых, помимо текста можно использовать значок. Во-вторых, с ее помощью можно выводить отформатированный текст.

Текст и значок метки можно задать в ее конструкторе. У нее есть несколько конструкторов с различными параметрами, в частности:

JLabel(String text) — создает метку с надписью text

JLabel(Icon image) — создает метку со значком image

JLabel(String text, Icon image, int align) — создает метку с надписью text и значком image. Третий параметр задает выравнивание текста вместе со значком. В качестве него может быть использована одна из констант, описанных в интерфейсе SwingConstants: LEFT, RIGHT, CENTER.

Для примера создадим окно с меткой, созданной при помощи третьего конструктора. Как и на прошлом занятии, мы будем использовать два класса, один из которых назовем SimpleWindow и унаследуем его от класса окна JFrame. В его конструкторе будут создаваться и размещаться все элементы окна. Второй класс будет создавать это окно и отображать его на экране (код будет таким же, как в примерах предыдущей главы).

Напишем в конструкторе класса SimpleWindow следующий код:

SimpleWindow(){

**super**("Окно с надписью");

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

JLabel label = **new** JLabel("Метка со значком и с надписью", **new** ImageIcon("1.gif"), SwingConstants.RIGHT);

getContentPane().add(label);

pack();

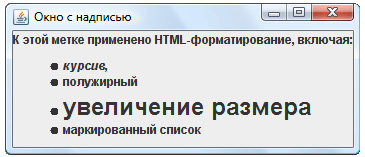
}

Чтобы убедиться, что выравнивание по правому краю работает, необходимо немного растянуть окно, чтобы ширина метки стала больше оптимальной.

В библиотеке Swing метка (и не только она) может быть настроена для отображения отформатированного текста в формате HTML. Для этого необходимо, чтобы строка, устанавливаемая в качестве надписи метки, начиналась с тега <html>. После этого можно использовать в строке любый теги языка HTML версии 3.2, и они будут преобразовываться в соответствующие атрибуты форматирования. В этом легко убедиться, изменив в предыдущем примере строку с вызовом конструктора на:

JLabel label = **new** JLabel("<html>К этой метке применено " +"HTML-форматирование, включая: <ul><li> <i>курсив</i>," +"<li><b>полужирный</b> <li><font size = +2> увеличение размера </font>" +"<li>маркированный список </ul>");

Поскольку текст нашей надписи достаточно длинный, строка для удобства восприятия разбивается на части и используется оператор +.

Перечислим основные методы класса JLabel:

getText() — возвращает текущий текст надписи метки

setText(String text) — задает новый текст надписи

getIcon() — возвращает значок метки

setIcon(Icon image) — устанавливает новый значок. В качестве значка обычно используется объект уже знакомого нам простого класса ImageIcon (как в вышеприведенном примере).

getVerticalAlignment(), setVerticalAlignment(int align), getHorizontalAlignment(), setHorizontalAlignment(int align) — эти четыре метода позволяют получить текущее или установить новое выравнивание (по горизонтали и вертикали) метки относительно ее границ. Возможные положения описаны в интерфейсе SwingConstants.

getVerticalTextPosition(), setVerticalTextPosition(int align), getHorizontalTextPosition(), setHorizontalTextPosition(int align) — эти четыре метода позволяют получить текущее или установить новое выравнивание текста относительно значка. Возможные положения описаны в интерфейсе SwingConstants.

getIconTextGap(), setIconTextGap(int gap) — позволяет получить или задать расстояние между текстом и значком метки в пикселах.

**Кнопка JButton**

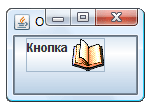
Мы постоянно использовали кнопки в предыдущей главе, хотя и не касались возможностей их настройки. *Кнопка* — это прямоугольник с текстом (и/или значком), по которому пользователь щелкает, когда хочет выполнить какое-то действие (или о чем-то сигнализировать).

Кнопка создается одним из пяти конструкторов, в частности JButton(), JButton(String text), JButton(Icon icon), JButton(String text, Icon icon), параметры которых говорят сами за себя. Пятый конструктор мы рассмотрим в следующей главе.

Кроме обычного значка можно назначить кнопке еще несколько — для различных состояний. Метод setRolloverIcon(Icon icon) позволяет задать значок, который будет появляться при наведении на кнопку мыши, setPressedIcon(Icon icon) — значок для кнопки в нажатом состоянии, setDisableIcon(Icon icon) — значок для неактивной кнопки. Каждому из этих методов соответствует метод get.

Метод setMargin(Insets margin) позволяет задать величину отступов от текста надписи на кнопке до ее полей. Объект класса Insets, который передается в этот метод, может быть создан конструктором с четырьмя целочисленными параметрами, задающими величину отступов: Insets(int top, int left, int bottom, int right). Метод getMargin() возвращает величину текущих отступов в виде объекта того же класса.

Все методы класса JLabel, описанные в предыдущем разделе, присутствуют и в классе JButton. С помощью этих методов можно изменять значок и текст надписи на кнопке, а также управлять их взаимным расположением друг относительно друга и относительно края кнопки (с учетом отступов).

Посредством методов setBorderPainted(boolean borderPainted), setFocusPainted(boolean focusPainted), setContentAreaFilled(boolean contentAreaFilled) можно отключать (параметром **false**) и включать обратно (параметром **true**) прорисовку рамки, прорисовку фокуса (кнопка, на которой находится фокус, выделяется пунктирным прямоугольником) и закраску кнопки в нажатом состоянии.

Для примера создадим кнопку со значком и с надписью, изменим ее отступы и расположение текста относительно значка (текст будет выровнен влево и вверх относительно значка).

SimpleWindow(){

**super**("Окно с кнопкой");

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

JButton button = **new** JButton("Кнопка", **new** ImageIcon("1.gif"));

button.setMargin(**new** Insets(0, 10, 20, 30));

button.setVerticalTextPosition(SwingConstants.TOP);

button.setHorizontalTextPosition(SwingConstants.LEFT);

getContentPane().add(button);

pack();

**Компоненты JToggleButton, JCheckBox, JRadioButton**

Компонент JToggleButton представляет собой кнопку, которая может находиться в двух состояниях: нажатом и отпущенном. Когда пользователь щелкает мышкой по такой кнопке, она изменяет свое состояние. Именно таким образом ведут себя кнопки форматирования на инструментальной панели текстового редактора. Кнопка [I] не только устанавливает или убирает курсивное начертание в выделенном тексте, но и сигнализирует о его наличии или отсутствии.

Основной конструктор — JToggleButton(String text, Icon icon, boolean selected) создает кнопку с заданными надписью, значком и текущим состоянием. Кнопку можно перевести в требуемое состояние программным путем, вызвав метод setSelected(boolean selected). Метод isSelected() возвращает **true**, если кнопка выбрана (т.е. находится в нажатом состоянии) и **false** в противном случает.

От класса JToggleButton унаследован класс JCheckBox — флажок. Этот класс имеет точно такой же набор конструкторов и методов, т.е. не расширяет функциональность предка. Единственное различие между ними — во внешнем виде: JCheckBox выглядит не как кнопка, а как небольшой квадратик, в котором можно поставить или убрать галочку.

Аналогичным образом ведет себя класс JRadioButton — переключатель или радиокнопка, внешне выглядящая как пустой кружок, когда она не выделена и кружок с точкой в выделенном состоянии.

Несмотря на то, что классы JCheckBox и JRadioButton ведут себя абсолютно одинаково (и аналогично их общему предку JToggleButton), их принято использовать в различных ситуациях. В частности, JRadioButton предполагает выбор единственной альтернативы из нескольких возможных: несколько таких объектов объединяются в одну группу (чаще всего эта группа визуально обозначается рамкой) и при выборе одного из элементов группы предыдущий выбранный элемент переходит в состояние «не выбран».

Для того, чтобы получить такое поведение, используется специальный контейнер ButtonGroup — взаимоисключающая группа (создается конструктором без параметров). Если добавить в один такой контейнер несколько элементов JRadioButton, то выбранным всегда будет только один из них.

В принципе, в ButtonGroup могут быть добавлены не только переключатели, но и флажки, и кнопки выбора (и даже обычные кнопки). Но при разработке интерфейса следует следовать устоявшемуся подходу, согласно которому во взаимоисключающую группу следует объединять объекты JRadioButton (и, в некоторых случаях JToggleButton), но не JCheckBox.

Метод add(AbstractButton button)**\*** добавляет элемент в группу. Метод getElements() возвращает все ее элементы в виде коллекции Enumeration. По коллекции можно пройтись итератором и найти выделенный элемент.

Рассмотри пример, в котором создаются две кнопки выбора, два флажка и два переключателя. Кнопки выбора и переключатели объединены в группы ButtonGroup. Для того, чтобы обвести каждую пару элементов рамкой, необходимо расположить каждую пару элементов на отдельной панели.

SimpleWindow(){

**super**("Пример с кнопками выбора, флажками и переключателями");

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

ImageIcon icon = **new** ImageIcon("1.gif"); // будем использовать один значок на все случаиBox mainBox = Box.createVerticalBox();

Box box1 = Box.createVerticalBox();

JToggleButton tButton1 = **new** JToggleButton("Кнопка выбора 1");

JToggleButton tButton2 = **new** JToggleButton("Кнопка выбора 2", icon);

ButtonGroup bg = **new** ButtonGroup(); // создаем группу взаимного исключения

bg.add(tButton1);

bg.add(tButton2); // сделали кнопки tButton1 и tButton2 взаимоисключающими

box1.add(tButton1);box1.add(tButton2); // добавили кнопки tButton1 и tButton2 на панель box1

box1.setBorder(**new** TitledBorder("Кнопки выбора"));

Box box2 = Box.createVerticalBox();

JCheckBox check1 = **new** JCheckBox("Флажок 1");

JCheckBox check2 = **new** JCheckBox("Флажок 2", icon);

box2.add(check1);

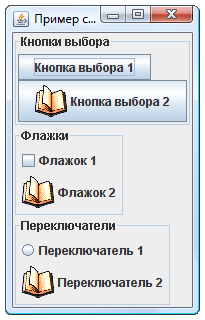
box2.add(check2); // добавили флажки на панель box2

box2.setBorder(**new** TitledBorder("Флажки"));

Box box3 = Box.createVerticalBox();

JRadioButton rButton1 = **new** JRadioButton("Переключатель 1");

JRadioButton rButton2 = **new** JRadioButton("Переключатель 2", icon);

bg = **new** ButtonGroup(); // создаем группу взаимного исключенияbg.add(rButton1);bg.add(rButton2); // сделали радиокнопки взаимоисключающими

box3.add(rButton1);

box3.add(rButton2); // добавили радиокнопки на панель box3

box3.setBorder(**new** TitledBorder("Переключатели"));

mainBox.add(box1);

mainBox.add(box2);

mainBox.add(box3);

setContentPane(mainBox);

pack();

}

Запустив пример, мы можем пронаблюдать особенности работы кнопок выбора, флажков и переключателей. В частности, мы видим, что у флажков или переключателей рисунок заменяет элемент выделения. Но рисунок не показывает, выбран ли данный объект, что может сбить пользователя с толку. Необходимо установить отдельный рисунок для выделенного состояния, что достигается методом setSelectedIcon(Icon icon). Добавьте в нужные места команды:

check2.setSelectedIcon(**new** ImageIcon("2.gif"));

и

rButton2.setSelectedIcon(**new** ImageIcon("2.gif"));

Пронаблюдайте произведенный эффект. Не забудьте, что файл 2.gif, равно как и файл 1.gif должны находиться в доступном для программы месте: в директории вашего проекта.

**Упражнение.**

В рассмотренном примере рамки имеют различную ширину. Измените пример таким образом, чтобы все рамки были одинаковы по ширине. Подсказка: используйте другой менеджер расположения главной панели (вместо BoxLayout).

**Текстовое поле JTextField**

*Текстовое поле* — простой и часто используемый компонент, предназначенный для ввода небольших по объему (записываемых в одну строку) текстовых данных. Для создания текстового поля чаще всего используются конструкторы:

JTextField(int columns) — создает пустое текстовое поле, ширина которого достаточна для размещения columns символов. При этом пользователь может вводить в текстовое поле строку какой угодно длины: она просто будет прокручиваться.

JTextField(String text) — создает текстовое поле с начальным текстом text.

JTextField(String text, int columns) — устанавливает и ширину и начальный текст.

Занести текст в поле можно методом setText(String text). Метод getText() возвращает содержимое текстового поля целиком, а getText(int offset, int length) — фрагмент содержимого длины length, начиная с символа offset.

Часть текста в поле может выделяться (как программным путем, так и в результате действий пользователя). Метод getSelectedText() позволяет получить выделенную часть текста. Заменить выделенный текст другим можно с помощью метода replaceSelection(String content). Методы getSelectionStart() и getSelectionEnd() возвращают границы выделенного участка, а методы setSelectionStart(int start) и setSelectionEnd(int end) изменяют их.

Метод getCaretPosition() возвращает позицию курсора (каретки) в текстовом поле, а метод setCaretPosition(int position) позволяет задать ее программно. Методом setCaretColor(Color color) можно изменить цвет курсора.

По умолчанию текст в поле прижимается к левому краю. Изменить это можно методом setHorizontalAlignment(int align), в качестве параметра передается одна из констант выравнивания, определенных в этом же классе JTextField: LEFT, CENTER, RIGHT.

**Поле для ввода пароля JPasswordField**

JPasswordField является прямым потомком JTextField, поэтому для него справедливо все сказанное выше. Отличие заключается в том, что весь введенный в него текст скрыт от посторонних глаз: он заменяется звездочками или другим символом, установить который позволяет метод setEchoChar(char echo), а получить — getEchoChar().

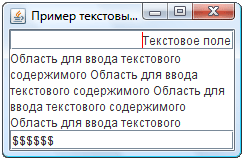
Чаще всего JPasswordField применяется для ввода пароля. Метод getText() позволяет получить этот пароль, но пользоваться им не рекомендуется (злоумышленник может проанализировать содержимое оперативной памяти и перехватить этот пароль). Вместо него следует использовать метод getPassword(), возвращающий массив символов char[]. После того, как введенный пароль будет обработан (например, сравнен с реальным паролем) рекомендуется заполнить этот массив нулями, чтобы следов в оперативной памяти не осталось.

**Область для ввода текста JTextArea**

JTextArea также является потомком JTextField и наследует все его методы. В отличие от текстового поля область для ввода текста позволяет ввести не одну строку, а несколько. В связи с этим JTextArea предлагает несколько дополнительных функций. Во-первых, это способность переносить слова на соседнюю строку целиком, которой управляет метод setWrapStyleWord(boolean wrapStyle). Если вызвать этот метод с параметром true, то слова не будут разрываться в том месте, где они «натыкаются» на границу компонента, а будут целиком перенесены на новую строку. Во-вторых, это способность переносить текст (то есть длинные строки будут укладываться в несколько строк вместо одной, уходящей за границы компонента. Этой способностью управляет метод setLineWrap(boolean lineWrap). Методы isWrapStyleWord() и isLineWrap() возвращают текущее состояние данных способностей (**true** — активирована и **false** — деактивирована).

При создании JTextArea чаще всего используют конструктор JTextArea(int rows, int columns), устанавливающий высоту (количество строк) и ширину (количество символов) компонента.

Для работы со своим содержимым JTextArea дополнительно предлагает два удобных метода. Метод append(String text) добавляет строку text в конец уже имеющегося текста, а метод insert(String text, int position) вставляет ее в позицию position.

Пронаблюдаем эти три компонента на наглядном примере. Создадим простое окно, в котором разместим их с помощью менеджера BorderLayout.

SimpleWindow(){

**super**("Пример текстовых компонентов");

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

JTextField textField = **new** JTextField("Текстовое поле", 20);

textField.setCaretColor(Color.RED);textField.setHorizontalAlignment(JTextField.RIGHT);

JPasswordField passwordField = **new** JPasswordField(20);

passwordField.setEchoChar('$');

passwordField.setText("пароль");

JTextArea textArea = **new** JTextArea(5, 20);

textArea.setLineWrap(true);

textArea.setWrapStyleWord(true);**for** (**int** i = 0; i <= 20; i++)textArea.append("Область для ввода текстового содержимого ");

getContentPane().add(textField, BorderLayout.NORTH);

getContentPane().add(textArea);

getContentPane().add(passwordField, BorderLayout.SOUTH);

pack();

}

Для того, чтобы лучше понять особенности работы текстовой области, замените по очереди **true** на **false** в вызовах методов setLineWrap() и setWrapStyleWord(). Пронаблюдайте за изменением работы компонента. Изменяйте размеры окна, чтобы видеть, каким образом текст перестраивается под доступное ему пространство.

**Упражнение.**

Расположите в окне те же самые три компонента, но с помощью менеджера размещения FlowLayout, который всегда устанавливает для компонентов их предпочтительные размеры. Пронаблюдайте за тем, как ведет себя область JTextArea при добавлении в нее нового текста.

**Панель прокрутки JScrollPane**

Наблюдая за поведением компонента JTextArea в предыдущем примере, легко можно обнаружить проблемы, которые возникают, когда тексту становится «тесно» в рамках отведенного места. В зависимости от используемого менеджера расположения текст либо обрезается, уходя за границы компонента, либо раздвигает эти границы (но в любом случае остается ограничен размером окна). В таких случаях типично использование полос прокрутки, но в Swing полосы прокрутки сами собой не появляются.

К счастью, добавить к компоненту полосы прокрутки на самом деле очень просто. Для этого служит компонент JScrollPane — панель прокрутки. Чаще всего она просто «надевается» на требуемый объект посредством собственного конструктора, принимающего этот объект в качестве параметра. Например, чтобы текстовая область textArea из предыдущего примера обрела полосы прокрутки, необходимо заменить команду

getContentPane().add(textArea);

на команду

getContentPane().add(**new** JScrollPane(textArea));

В этой команде создается панель с полосами прокрутки, в нее помещается объект textArea, а сама панель добавляется в панель содержимого окна. Теперь текст свободно прокручивается. А в случае применения менеджера FlowLayout или BoxLayout компонент JTextArea не будет подстраиваться под свое содержимое (будет иметь предпочтительный размер, соответствующий параметрам конструктора) и, при необходимости, отображать полоски прокрутки.

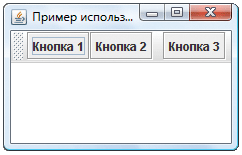
Полезными методами JScrollPane являются:

setHorizontalScrollBarPolicy(int policy) — позволяет задать стратегию работы с горизонтальной полосой прокрутки. Возможные значения представлены константами HORIZONTAL\_SCROLLBAR\_ALWAYS (отображать всегда), HORIZONTAL\_SCROLLBAR\_AS\_NEEDED (отображать при необходимости) и HORIZONTAL\_SCROLLBAR\_NEVER (не отображать никогда). Данные константы определены в интерфейсе ScrollPaneConstants.

*об интерфейсах с константами*

setVerticalScrollBarPolicy(int policy) позволяет задать стратегию работы с вертикальной полосой прокрутки посредством констант VERTICAL\_SCROLLBAR\_ALWAYS, VERTICAL\_SCROLLBAR\_AS\_NEEDED и VERTICAL\_SCROLLBAR\_NEVER.**\***

**Инструментальная панель JToolBar**



Большинство программных продуктов предоставляют удобные инструментальные панели, расположенные вдоль границ окна программы и содержащие кнопки, выпадающие списки и другие элементы управления, обычно соответствующие командам меню. В Swing для инструментальных панелей разработан визуальный компонент JToolBar, в котором заложена просто потрясающая функциональность.

На этот раз начнем сразу с примера. Создадим окно с менеджером расположения BorderLayout, разместим по центру область для ввода текста JTextArea, а к верхней границе прикрепим инструментальную панель с тремя кнопками и одним разделителем:

SimpleWindow(){

**super**("Пример использования JToolBar");

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

JTextArea textArea = **new** JTextArea(5, 20);

getContentPane().add(textArea);

JToolBar toolBar = **new** JToolBar("Инструментальная панель");

toolBar.add(**new** JButton("Кнопка 1"));

toolBar.add(**new** JButton("Кнопка 2"));

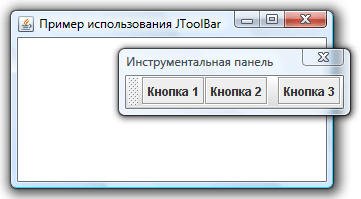
toolBar.addSeparator();

toolBar.add(**new** JButton("Кнопка 3"));

getContentPane().add(toolBar, BorderLayout.NORTH);

pack();

}

Запустите пример и поэкспериментируйте с инструментальной панелью. Попробуйте отсоединить ее от верхней границы окна и прикрепить к какой-либо другой. Отсоедините ее от границ окна так, чтобы панель стала самостоятельным окном. При этом панель всегда отображается над родительским окном, даже если именно оно, а не панель является активным. Если закрыть самостоятельную панель кнопкой с крестиком, она вернется в свое окно, в то место, где она была закреплена последний раз.

Итак, весьма простой пример продемонстрировал весьма впечатляющие возможности инструментальной панели. Подведем итог, перечислив самые полезные методы JToolBar:

Конструктор JToolBar(String title) создает горизонтальную панель с заданным заголовком. Горизонтальная панель предназначена для прикрепления к верхней либо нижней границе родительской панели (имеющей расположение BorderLayout). Для создания вертикальной панели используется конструктор JToolBar(String title, int orientation), где параметр orientation задается константой VERTICAL из интерфейса SwingConstants. Также доступны конструкторы JToolBar() и JToolBar(int orientation), создающие панель без заголовка.

setFloatable(boolean floatable) — разрешает либо запрещает (по умолчанию разрешает) пользователю откреплять панель от места ее начального расположения. Ему соответствует метод isFloatable() возвращающий **true**, если откреплять панель разрешено.

add(Component component) — добавляет на инструментальную панель новый элемент управления. Взаимосвязанные группы элементов управления принято разделять с помощью линии или пустого пространства. Метод addSeparator() добавляет такой разделитель.

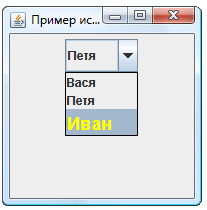
**Выпадающий список JComboBox**

Выпадающий список — весьма распространенный элемент управления. Он содержит множество вариантов, из которых пользователь может выбрать один и только один, либо (если выпадающий список это позволяет) ввести свой собственный.

Создать выпадающий список можно конструктором по умолчанию JComboBox(), после чего добавлять в него элементы методом addItem(Object item), добавляющим новый элемент в конец списка, или методом insertItemAt(Object item, int index), позволяющим уточнить позицию, в которую требуется вставить элемент. Однако проще использовать конструктор, в котором сразу указываются все элементы выпадающего списка. Таких конструкторов два: JComboBox(Object[] elements) и JComboBox(Vector elements). Работают они одинаково, так что это вопрос удобства разработчика: использовать массив или вектор.

Чаще всего в выпадающий список добавляют строки, однако, как это следует из сигнатур описанных выше методов, он может содержать вообще любые объекты. Любой объект преобразуется к строке методом toString(), именно эта строка и будет представлять его в выпадающем списке.

Метод getItemAt(int index) позволяет обратиться к произвольному элементу.

Метод removeAllItems() удаляет из JComboBox все элементы, а метод removeItem(Object item) — конкретный элемент (при условии, что он содержался в списке).

Метод getSelectedIndex() позволяет получить индекс выбранного пользователем элемента (элементы нумеруются начиная с нуля), а метод getSelectedItem() возвращает сам выбранный объект. Сделать конкретный элемент выбранным можно и программно, воспользовавшись методом setSelectedIndex(int index) или setSelectedItem(Object item).

Чтобы пользователь мог ввести свой вариант, который не присутствует в списке, должен быть вызван метод setEditable(boolean editable) с параметром **true**. Ему соответствует метод isEditable().

Рассмотрим пример, в котором создается выпадающий список из 3 элементов и выбирается 2-й. Строка, представляющая третий элемент, использует HTML-теги. Как показывает результат, они работают не только в метках**\***.

SimpleWindow(){

**super**("Пример использования JComboBox");

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

String[] elements = **new** String[] {"Вася", "Петя","<html><font size = +1 color = yellow>Иван</font>"};

JComboBox combo = **new** JComboBox(elements);

combo.setSelectedIndex(1);

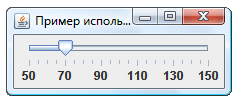
JPanel panel = **new** JPanel();panel.add(combo);

setContentPane(panel);

setSize(200,200);

}

**Ползунок JSlider**

Ползунок позволяет пользователю выбрать некоторое число из диапазона доступных значений, наглядно представив этот диапазон. Против наглядности у ползунка есть один недостаток: он занимает достаточно много места.

Основной конструктор ползунка: JSlider(int orientation, int min, int max, int value). Первый параметр — ориентация ползунка (HORIZONTAL или VERTICAL). Остальные параметры указывают соответственно минимальное, максимальное и текущее значение. Изменить эти значения позволяют методы setOrientation(int), setMinimum(int min), setMaximum(int max), setValue(int value), а получить текущие — соответствующие им методы get. Чаще всего, конечно, используется метод getValue() — чтобы определить, какое значение выбрал при помощи ползунка пользователь.

Шкала ползунка может быть украшена делениями. Метод setMajorTickSpacing(int spacing) позволяет задать расстояние, через которое будут выводиться большие деления, а метод setMinorTickSpacing(int spacing) — расстояние, через которые будут выводиться маленькие деления. Метод setPaintTicks(boolean paint) включает или отключает прорисовку этих делений. Метод setSnapToTicks(boolean snap) включает или отключает «прилипание» ползунка к делениям: если вызвать этот метод с параметром **true**, пользователь сможет выбрать при помощи ползунка только значения, соответствующие делениям. Наконец, метод setPaintLabels(boolean paint) включает или отключает прорисовку меток под большими делениями.

Пример использования перечисленных методов:

SimpleWindow(){

**super**("Пример использования JSlider");

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

JSlider slider = **new** JSlider(JSlider.HORIZONTAL, 50, 150, 70);

slider.setMajorTickSpacing(20);

slider.setMinorTickSpacing(5);

slider.setPaintTicks(true);

slider.setPaintLabels(true);

slider.setSnapToTicks(true);

JPanel panel = **new** JPanel();

panel.add(slider);

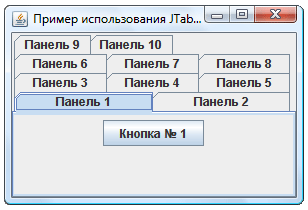
setContentPane(panel);

pack();

}

**Панель со вкладками JTabbedPane**

Многим программам бывает необходимо разместить в одном окне большое количество элементов управления, некоторые из которых (такие как списки, деревья, текстовые области и т.д.) могут к тому же занимать приличное пространство. Чаще всего такая необходимость возникает, когда для работы программе необходимо множество входных данных. Для того, чтобы не вводить дополнительных окон и не перегружать интерфейс, часто используется панель с закладками. Ее можно воспринимать как множество страниц (вкладок), каждая из которых занимает все доступное пространство, за исключением полосы с краю (это может быть любой край), где отображаются ярлычки с названиями страниц. Когда пользователь щелкает по ярлычку, открывается соответствующая ему страница. На каждой странице размещено несколько элементов управления (как правило, они группируются по смыслу).

Создать панель со вкладками можно простым конструктором, в котором определяется только месторасположение ярлычков (LEFT, RIGHT, TOP или BOTTOM). Но иногда бывает полезен конструктор JTabbedPane(int orientation, int layout), где второй параметр принимает значения, соответствующие константам SCROLL\_TAB\_LAYOUT (если все ярлычки не помещаются, появляется полоса прокрутки) или WRAP\_TAB\_LAYOUT (ярлычки могут располагаться в несколько рядов).

После этого можно добавлять вкладки методом addTab(), имеющим несколько вариантов. В частности, метод addTab(String title, Component tab) добавляет закладку с указанием текста ярлычка, а метод addTab(String title, Icon icon, Component tab) позволяет задать также и значок к ярлычку. В качестве вкладки обычно служит панель с размещенными на ней элементами управления.

Создадим панель с десятью вкладками, на каждой из которых поместим по кнопке. Все эти вкладки создадим в цикле **for**, чтобы не писать много кода.

SimpleWindow(){

**super**("Пример использования JTabbedPane");

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

JTabbedPane tabbedPane = **new** JTabbedPane(JTabbedPane.TOP, JTabbedPane.WRAP\_TAB\_LAYOUT);

**for** (**int** i = 1; i <= 10; i++) {JPanel panel = **new** JPanel();

panel.add(**new** JButton("Кнопка № " + i));

tabbedPane.add("Панель " + i, panel);

}

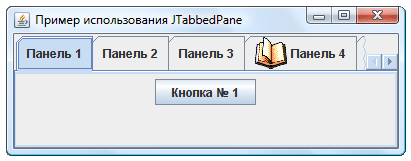
getContentPane().add(tabbedPane);

setSize(300,200);

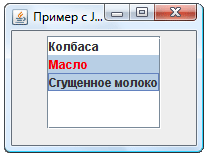
}

**Упражнение.**

Измените пример, чтобы вкладки располагались не в несколько рядов, а прокручивались и, кроме того, добавьте к ярлыку четвертой вкладки значок. Результат должен выглядеть так, как изображено на рисунке.



**Список JList**

Список JList — это один из сложных компонентов, для эффективной работы с которыми необходимо понимание основ библиотеки Swing, в частности, концепции «Модель-Вид». Компоненты JTree (дерево) и JTable (таблица) еще сложнее и в данном пособии не рассматриваются. Что касается списка, то некоторая часть его возможностей может быть использована без углубления в детали.

Список содержит группу элементов, аналогично выпадающему списку JComboBox, но обладает двумя отличительными особенностями. Во-первых, на экране видны одновременно несколько элементов списка. Во-вторых, пользователь может выбрать в списке не один элемент, а несколько (если установлен соответствующий режим выделения).

Создать список можно с помощью конструктора, работающего на основе массива Object[] или вектора Vector (аналогично JComboBox). Метод setVisibleRowCount(int count) устанавливает количество видимых элементов списка. Остальные элементы будут уходить за его пределы или прокручиваться, если поместить список в JScrollPane (что рекомендуется).

По умолчанию пользователь может выбрать в списке любое число элементов, держа нажатой клавишу Ctrl. Это можно изменить, вызвав метод setSelectionMode(int mode), где параметр задается одной из констант класса ListSelectionModel:

SINGLE\_SELECTION — может быть выделен только один элемент,

SINGLE\_INTERVAL\_SELECTION — может быть выделено несколько элементов, но составляющих непрерывный интервал,

MULTIPLE\_INTERVAL\_SELECTION — может быть выделено произвольное количество смежных и несмежных элементов.

Выделенный элемент списка (если он один) можно получить методом getSelectedValue(). Если таких несколько, метод вернет первый из них. Метод getSelectedValues() возвращает все выделенные элементы списка в виде массива Object[]. Аналогично работают методы getSelectedIndex() и getSelectedIndices(), только возвращают они не сами выделенные элементы, а их индексы. Всем этим методам соответствуют методы set, так что выделить элементы списка можно и программно.

Следующий пример иллюстрирует некоторые из этих возможностей JList:

SimpleWindow(){

**super**("Пример с JList");

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

Object[] elements = **new** Object[] {"Колбаса", "<html><font color = red>Масло", "Сгущенное молоко"};

JList list = **new** JList(elements);

list.setVisibleRowCount(5);

list.setSelectionMode(ListSelectionModel.SINGLE\_INTERVAL\_SELECTION);

list.setSelectedIndices(**new** int[] {1,2});

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

getContentPane().add(**new** JScrollPane(list));

setSize(200,150);

}

Для того, чтобы эффективно добавлять и удалять элементы из списка, управлять их отображением, добавлять и удалять выделенные элементы поодиночке, необходимо познакомиться с моделью данных списка, что выходит за рамки настоящего пособия.

**Окно входа в систему**

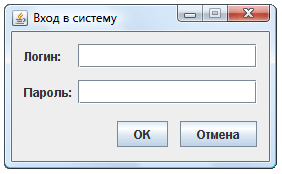
В качестве реального примера законченного окна, иллюстрирующего положения этой и предыдущей глав, рассмотрим окно для входа в систему (которое является необходимой частью каждого из итоговых заданий). Это окно содержит два текстовых поля для ввода логина и пароля, подписи к этим полям и кнопки «OK» и «Отмена».

В первую очередь необходимо сделать на бумаге (или с помощью специализированного программного средства рисования интерфейсов, такого как Microsoft Visio) эскиз будущего окна, чтобы определиться, где должны быть расположены элементы управления и какой менеджер расположения должен быть выбран.

В нашем окне нет элементов, которые имеет смысл растягивать на все доступное пространство, поэтому будем использовать один из менеджеров, сохраняющих предпочтительные размеры компонентов, а именно — BoxLayout. Проще всего представить наше окно как три горизонтальные панели, объединенные в одной вертикальной. В первой из них будет надпись «Логин:» и текстовое поле. Во второй — надпись «Пароль:» и поле для ввода пароля. В третьей будут размещены две кнопки. При этом необходимо учесть следующее:

1. Кнопки «ОК» и «Отмена» принято прижимать к правому краю окна, поэтому в начале третьей горизонтальной панели необходимо добавить «пружину».
2. Должно выдерживаться аккуратное расстояние между элементами. В частности, для стиля Java разработаны следующие рекомендации. Тесно связанные элементы (такие как текстовое поле и подпись к нему) должны отстоять друг от друга на 6 пикселов. Логически сгруппированные элементы — на 12 пикселов (в нашем случае это две верхние панели и пара кнопок). Все остальные элементы должны находиться на расстоянии 17 пикселов друг от друга. Не следует забывать и про рамку окна.
3. Элементы должны быть аккуратно выровнены. В нашем случае надписи, которые мы собираемся поместить перед текстовыми полями, наверняка будут разной длины и из-за этого поля будут сдвинуты относительно друг друга, что не желательно. Поэтому следует принудительно задать у надписей одинаковую ширину.
4. При увеличении размеров окна текстовые поля будут неэстетично изменять свою высоту. Можно ее зафиксировать, а можно просто запретить окну изменять свои размеры после упаковки командой setResizable(false).

Итоговый код окна приведен ниже. Рекомендуется не только набрать и протестировать, но и внимательно разобрать его.



**public class** LoginWindow **extends** JFrame { /\* Для того, чтобы впоследствии обращаться к содержимому текстовых полей, рекомендуется сделать их членами класса окна \*/

JTextField loginField;

JPasswordField passwordField;

LoginWindow(){

**super**("Вход в систему");

setDefaultCloseOperation(EXIT\_ON\_CLOSE);// Настраиваем первую горизонтальную панель (для ввода логина)

Box box1 = Box.createHorizontalBox();

JLabel loginLabel = **new** JLabel("Логин:");

loginField = **new** JTextField(15);

box1.add(loginLabel);box1.add(Box.createHorizontalStrut(6));

box1.add(loginField);// Настраиваем вторую горизонтальную панель (для ввода пароля)

Box box2 = Box.createHorizontalBox();

JLabel passwordLabel = **new** JLabel("Пароль:");

passwordField = **new** JPasswordField(15);

box2.add(passwordLabel);

box2.add(Box.createHorizontalStrut(6));

box2.add(passwordField);// Настраиваем третью горизонтальную панель (с кнопками)

Box box3 = Box.createHorizontalBox();

JButton ok = **new** JButton("OK");

JButton cancel = **new** JButton("Отмена");

box3.add(Box.createHorizontalGlue());

box3.add(ok);

box3.add(Box.createHorizontalStrut(12));

box3.add(cancel);// Уточняем размеры компонентов

loginLabel.setPreferredSize(passwordLabel.getPreferredSize());// Размещаем три горизонтальные панели на одной вертикальной

Box mainBox = Box.createVerticalBox();

mainBox.setBorder(**new** EmptyBorder(12,12,12,12));

mainBox.add(box1);

mainBox.add(Box.createVerticalStrut(12));

mainBox.add(box2);

mainBox.add(Box.createVerticalStrut(17));

mainBox.add(box3);

setContentPane(mainBox);

pack();

setResizable(false);

}

}

**Заключение**

Мы сделали лишь краткий обзор компонентов Swing. Нами не были упомянуты некоторые полезные компоненты, такие как индикатор состояния процесса JProgressBar, счетчик JSpinner, панель с разделительной полосой JSplitPane, а также ряд мощных текстовых компонентов. Многие из рассмотренных в этой главе компонентов внушительными возможностями, оставшимися за рамками нашего обзора. И хотя на основе материала этой и следующей, а также предыдущей глав можно разработать полноценный графический интерфейс, необходимый для выполнения основного задания, для профессиональной работы с библиотекой Swing необходимо обратиться к полноценным учебникам и справочникам.

**Обработка событий в Swing**

Графический интерфейс пользователя включает в себя не только расположение в окне необходимых элементов управления, но и назначение реакции на пользовательские события. Большая честь действий в оконных программах выполняется в ответ на выбор пользователем команд меню, нажатие кнопок, а иногда даже просто в ответ на ввод нового символа в текстовое поле.

Таким образом, при разработке программы необходимо:

1. Выявить события, в ответ на которые потребуется реакция программы.
2. Написать код, реализующий эту реакцию (так называемый обработчик событий).
3. Связать обработчик события с соответствующим событием.

Первый пункт зависит от логики работы приложения и находится всецело на усмотрении программиста. Обработчик события — это, как правило, обычный метод и ничего особенного в его написании нет.

**Паттерн проектирования «наблюдатель»**

В разных языках программирования и визуальных библиотеках используются самые различные способы привязки процедуры (метода), которая должна вызываться в ответ на некоторое событие, к самому этому событию. В частности, в Delphi процедура просто должна иметь особое название, состоящее из имени компонента и типа обрабатываемого события.

В библиотеке Swing для привязки обработчика события к вызывающему его компоненту используется паттерн проектирования «наблюдатель».

Паттерны проектирования — это стандартные приемы объектно-ориентированного программирования, позволяющие оптимальным образом справиться с нетривиальными, но часто возникающими в программировании ситуации. Паттерн проектирования описывает классы, которые необходимо ввести для разрешения такой ситуации и взаимодействие между классами. В частности, ранее мы рассматривали паттерн проектирования «итератор», позволяющий обойти все элементы коллекции с помощью двух простейших методов в порядке, скрытом в самом итераторе.

Паттерн проектирования «наблюдатель» применяется, когда один объект должен оповещать другие о произошедших с ним изменениях или внешних воздействиях. Такой объект называется наблюдаемым, а объекты, которые следует оповестить — наблюдателями.

Для того, чтобы подобное взаимодействие было возможным, объект-наблюдатель должен иметь метод (или несколько методов) с заранее определенной сигнатурой (именем и параметрами). Когда с наблюдаемым объектом произойдет ожидаемое событие, он вызовет соответствующий метод у своего наблюдателя. В этом методе и будет запрограммирована реакция на событие.

Для того, чтобы наблюдаемый объект мог вызвать метод наблюдателя, он должен знать о том, что такой наблюдатель существует. Поэтому у наблюдаемого предварительно должен быть вызван специальный метод, регистрирующий его наблюдателя.

Заметим, что в данной схеме один наблюдатель может быть зарегистрирован у нескольких объектов (т.е. одинаково реагировать на изменения в каждом из них), а у одного объекта может быть несколько наблюдателей (т.е. при возникновении события выполняется несколько независимых методов-обработчиков). Это весьма увеличивает гибкость программирования.

**Механизм обработки событий библиотеки Swing**

В контексте графического интерфейса пользователя наблюдаемыми объектами являются элементы управления: кнопки, флажки, меню и т.д. Они могут сообщить своим наблюдателям об определенных событиях, как элементарных (наведение мышкой, нажатие клавиши на клавиатуре), так и о высокоуровневых (изменение текста в текстовом поле, выбор нового элемента в выпадающем списке и т.д.).

Наблюдателями должны являться объекты классов, поддерживающих специальные интерфейсы (вспомним, что в классе наблюдателя должны быть определенные методы, о которых «знает» наблюдаемый и вызывает их при наступлении события). Такие классы в терминологии Swing называются *слушателями*.

**Интерфейс MouseListener и обработка событий от мыши**

Мы рассмотрим технологию написания слушателей на примере слушателей событий мыши.

События от мыши — один из самых популярных типов событий. Практически любой элемент управления способен сообщить о том, что на него навели мышь, щелкнули по нему и т.д. Об этом будут оповещен все зарегистрированные слушатели событий от мыши.

Так например, кнопка для входа в систему из примера в конце предыдущей главы должна реагировать на щелчок по ней, проверяя имя и пароль, введенные пользователем.

Слушатель событий от мыши должен реализовать интерфейс MouseListener. В этом интерфейсе перечислены следующие методы:

public void mouseClicked(MouseEvent event) — выполнен щелчок мышкой на наблюдаемом объекте

public void mouseEntered(MouseEvent event) — курсор мыши вошел в область наблюдаемого объекта

public void mouseExited(MouseEvent event) — курсор мыши вышел из области наблюдаемого объекта

public void mousePressed(MouseEvent event) — кнопка мыши нажата в момент, когда курсор находится над наблюдаемым объектом

public void mouseReleased(MouseEvent event) — кнопка мыши отпущена в момент, когда курсор находится над наблюдаемым объектом

*о параметре типа MouseEnent*

Чтобы обработать нажатие на кнопку, требуется описать класс, реализующий интерфейс MouseListener, причем метод mouseClicked() должен содержать обработчик события. Далее необходимо создать объект этого класса и зарегистрировать его как слушателя интересующей нас кнопки. Для регистрации слушателя используется метод addMouseListener(MouseListener listener).

Опишем класс слушателя в пределах класса окна SimpleWindow, после конструктора. Обработчик события будет проверять, ввел ли пользователь логин «Иван» (пароль проверять не будем) и выводить сообщение об успехе или неуспехе входа в систему (работа стандартных диалоговых окон обсуждается дальше в этой главе):

**class** MouseL **implements** MouseListener {

**public void** mouseClicked(MouseEvent event) {

**if** (loginField.getText().equals("Иван"))JOptionPane.showMessageDialog(**null**, "Вход выполнен");

**else** JOptionPane.showMessageDialog(**null**, "Вход НЕ выполнен");

}

**public void** mouseEntered(MouseEvent event) {}

**public void** mouseExited(MouseEvent event) {}

**public void** mousePressed(MouseEvent event) {}

**public void** mouseReleased(MouseEvent event) {} }

Мы сделали слушателя вложенным классом класса SimpleWindow, чтобы он мог легко получить доступ к его внутренним полям loginField и passwordField. Кроме того, хотя реально мы обрабатываем только одно из пяти возможных событий мыши, описывать пришлось все пять методов (четыре имеют пустую реализацию). Дело в том, что в противном случае класс пришлось бы объявить абстрактным (ведь он унаследовал от интерфейса пустые заголовки методов) и мы не смогли бы создать объект этого класса. А мы должны создать объект слушателя и прикрепить его к кнопке. Для этого в код конструктора SimpleWindow() необходимо добавить команду:

ok.addMouseListener(**new** MouseL());

Это можно сделать сразу после команды:

JButton ok = **new** JButton("OK");

**Создание слушателей с помощью анонимных классов**

Чтобы кнопка ok обрела слушателя, который будет обрабатывать нажатие на нее, нам понадобилось описать новый (вложенный) класс. Иногда вместо вложенного класса можно обойтись анонимным. Как вы помните, анонимный класс не имеет имени и в программе может быть создан только один объект этого класса (создание которого совмещено с определением класса). Но очень часто, как и в нашем примере, слушатель пишется для того, чтобы обрабатывать события единственного объекта — в нашем случае кнопки ok, а значит, используется в программе только однажды: во время привязки к этому объекту. Таким образом, мы можем заменить вложенный класс анонимным. Для этого описание класса MouseL можно просто удалить, а команду

ok.addMouseListener(**new** MouseL());

заменить на:

ok.addMouseListener(**new** MouseListener() {

**public void** mouseClicked(MouseEvent event) {

**if** (loginField.getText().equals("Иван"))JOptionPane.showMessageDialog(**null**, "Вход выполнен");

**else** JOptionPane.showMessageDialog(**null**, "Вход НЕ выполнен");

}

**public void** mouseEntered(MouseEvent event) {}

**public void** mouseExited(MouseEvent event) {}

**public void** mousePressed(MouseEvent event) {}

**public void** mouseReleased(MouseEvent event) {}

}

);

Новый вариант выглядит более громоздко, чем первый. Злоупотребление анонимными классами может сделать программу плохо читаемой. Однако в результате все действия с кнопкой (создание, настройка ее внешнего вида и команды обработки щелчка по ней) не разнесены, как в случае вложенных классов, а находятся рядом, что облегчает сопровождение (внесение изменений) программы. В случае простых (в несколько строк) обработчиков разумно делать выбор в пользу анонимных классов.

Вообще, можно реализовать интерфейс слушателя в любом классе, добавив к его описанию директиву implements MouseListener и определив в нем необходимые методы. В частности, это может быть сам класс окна SimpleWindow. Но чаще всего пользуются анонимными и вложенными классами (не в последнюю очередь благодаря наличию адаптеров (см. далее).

**Класс MouseAdapter**

Программа стала выглядеть загроможденной главным образом из-за того, что помимо полезного для нас метода mouseClicked() нам пришлось определять пустые реализации всех остальных, не нужных методов. В принципе, этого можно избежать.

Класс MouseAdapter реализует интерфейс MouseListener, определяя пустые реализации для каждого из его методов. Можно унаследовать своего слушателя от этого класса и переопределить те методы, которые нам нужны.

В результате предыдущее описание слушателя будет выглядеть более компактно:

ok.addMouseListener(**new** MouseAdapter() {

**public void** mouseClicked(MouseEvent event) {

**if** (loginField.getText().equals("Иван"))JOptionPane.showMessageDialog(**null**, "Вход выполнен");

**else** JOptionPane.showMessageDialog(**null**, "Вход НЕ выполнен");

}

}

);

**Общая структура слушателей**

Кроме слушателей MouseListener визуальные компоненты Swing поддерживают целый ряд других слушателей (заметим, что в разных компонентах набор возможных слушателей может отличаться).

Каждый слушатель должен реализовывать интерфейс \*\*\*Listener, где \*\*\* — тип слушателя. Практически каждому из этих интерфейсов (за исключением тех, в которых всего один метод) соответствует пустой класс-заглушка \*\*\*Adapter. Каждый метод интерфейса слушателя принимает один параметр типа \*\*\*Event, в котором собрана вся информация, относящаяся к событию. Мы не будем подробно рассматривать методы получения этой информации. Как правило их немного и их назначение легко понять, а перечень методов даст контекстная подсказка Eclipse.

Чтобы привязать слушателя к объекту (который поддерживает соответствующий тип слушателей) используется метод add\*\*\*Listener(\*\*\*Listener listener).

Например, слушатель MouseListener должен реализовать интерфейс с таким же именем, которому соответствует класс-заглушка MouseAdapter. Методы этого интерфейса обрабатывают параметр типа MouseEvent, а регистрируется слушатель методом addMouseListener(MouseListener listener). Мы не будем повторять эту информацию для других типов слушателей, они работают аналогично.

**Слушатель фокуса FocusListener**

Слушатель FocusListener отслеживает моменты, когда объект получает фокус (то есть становится активным) или теряет его. Концепция фокуса очень важна для оконных приложений. В каждый момент времени в окне может быть только один активный (находящийся в фокусе) объект, который получает информацию о нажатых на клавиатуре клавишах (т.е. реагирует на события клавиатуры), о прокрутке колесика мышки и т.д. Пользователь активирует один из элементов управления нажатием мышки или с помощью клавиши Tab (переключаясь между ними).

Интерфейс FocusListener имеет два метода:

public void focusGained(FocusEvent event) — вызывается, когда наблюдаемый объект получает фокус

public void focusLost(FocusEvent event) — вызывается, когда наблюдаемый объект теряет фокус.

**Слушатель колесика мышки MouseWheelListener**

Слушатель MouseWheelListener оповещается при вращении колесика мыши в тот момент, когда данный компонент находится в фокусе. Этот интерфейс содержит всего один метод:

public void mouseWheelMoved(MouseWheelEvent event).

**Слушатель клавиатуры KeyListener**

Слушатель KeyListener оповещается, когда пользователь работает с клавиатурой в тот момент, когда данный компонент находится в фокусе. В интерфейсе определены методы:

public void mouseKeyTyped(KeyEvent event) — вызывается, когда с клавиатуры вводится символ

public void mouseKeyPressed(KeyEvent event) — вызывается, когда нажата клавиша клавиатуры

public void mouseKeyReleased(KeyEvent event) — вызывается, когда отпущена клавиша клавиатуры.

Аргумент event этих методов способен дать весьма ценные сведения. В частности, команда event.getKeyChar() возвращает символ типа char, связанный с нажатой клавишей. Если с нажатой клавишей не связан никакой символ, возвращается константа CHAR\_UNDEFINED. Команда event.getKeyCode() возвратит код нажатой клавиши в виде целого числа типа int. Его можно сравнить с одной из многочисленных констант, определенных в классе KeyEvent: VK\_F1, VK\_SHIFT, VK\_D, VK\_MINUS и т.д. Методы isAltDown(), isControlDown(), isShiftDown() позволяют узнать, не была ли одновременно нажата одна из клавиш-модификаторов Alt, Ctrl или Shift.

**Упражнение.**

В окне входа в систему добавьте возможность, чтобы при нажатии клавиши Esc в окне ввода логина его содержимое очищалось.

**Слушатель изменения состояния ChangeListener**

Слушатель ChangeListener реагирует на изменение состояния объекта. Каждый элемент управления по своему определяет понятие «изменение состояния». Например, для панели со вкладками JTabbedPane это переход на другую вкладку, для ползунка JSlider — изменение его положения, кнопка JButton рассматривает как смену состояния щелчок на ней. Таким образом, хотя событие это достаточно общее, необходимо уточнять его специфику для каждого конкретного компонента. В интерфейсе определен всего один метод:

public void stateChanged(ChangeEvent event).

**Слушатель событий окна WindowListener**

Слушатель WindowListener может быть привязан только к окну и оповещается о различных событиях, произошедших с окном:

public void windowOpened(WindowEvent event) — окно открылось.

public void windowClosing(WindowEvent event) — попытка закрытия окна (например, пользователя нажал на крестик). Слово «попытка» означает, что данный метод вызовется до того, как окно будет закрыто и может воспрепятствовать этому (например, вывести диалог типа «Вы уверены?» и отменить закрытие окна, если пользователь выберет «Нет»).

public void windowClosed(WindowEvent event) — окно закрылось.

public void windowIconified(WindowEvent event) — окно свернуто.

public void windowDeiconified(WindowEvent event) — окно развернуто.

public void windowActivated(WindowEvent event) — окно стало активным.

public void windowDeactivated(WindowEvent event) — окно стало неактивным.

**Слушатель событий компонента СomponentListener**

Слушатель ComponentListener оповещается, когда наблюдаемый визуальный компонент изменяет свое положение, размеры или видимость. В интерфейсе четыре метода:

public void componentMoved(ComponentEvent event) — вызывается, когда наблюдаемый компонент перемещается (в результате вызова команды setLocation(), работы менеджера размещения или еще по какой-то причине).

public void componentResized(ComponentEvent event) — вызывается, когда изменяются размеры наблюдаемого компонента.

public void componentHidden(ComponentEvent event) — вызывается, когда компонент становится невидимым.

public void componentShown(ComponentEvent event) — вызывается, когда компонент становится видимым.

**Слушатель выбора элемента ItemListener**

Слушатель ItemListener реагирует на изменение состояния одного из элементов, входящих в состав наблюдаемого компонента. Например, выпадающий список JComboBox состоит из множества элементов и слушатель реагирует, когда изменяется выбранный элемент. Также данный слушатель оповещается при выборе либо отмене выбора флажка JCheckBox или переключателя JRadioButton, изменении состояния кнопки JToggleButton и т.д. Слушатель обладает одним методом:

public void itemStateChanged(ItemEvent event).

**Универсальный слушатель ActionListener**

Среди многочисленных событий, на которые реагирует каждый элемент управления (и о которых он оповещает соответствующих слушателей, если они к нему присоединены), есть одно основное, вытекающее из самой сути компонента и обрабатываемое значительно чаще, чем другие. Например, для кнопки это щелчок на ней, а для выпадающего списка — выбор нового элемента.

Для отслеживания и обработки такого события может быть использован особый слушатель ActionListener, имеющий один метод:

public void actionPerformed(ActionEvent event).

У использования ActionListener есть небольшое преимущество в эффективности (так, при обработке нажатия на кнопку не надо реагировать на четыре лишних события — ведь даже если методы-обработчики пустые, на вызов этих методов все равно тратятся ресурсы). А кроме того очень удобно запомнить и постоянно использовать один класс с одним методам и обращаться к остальным лишь в тех относительно редких случаях, когда возникнет такая необходимость.

Обработка нажатия на кнопку ok в нашем примере легко переписывается для ActionListener:

ok.addMouseListener(**new** ActionListener() {

**public void** actionPerformed(ActionEvent event) {

**if** (loginField.getText().equals("Иван"))JOptionPane.showMessageDialog(**null**, "Вход выполнен");

**else** JOptionPane.showMessageDialog(**null**, "Вход НЕ выполнен");

}

}

);

**Работа с меню в библиотеке Swing**

Неотъемлемой частью современных оконных программ является меню, представляющее собой удобно сгруппированный набор команд. Меню бывает двух типов: главное и контекстное. *Главное меню* располагается вдоль верхней границы окна и содержит команды, относящиеся ко всему приложению (точнее, все команды, которые можно выполнить, работая с данным окном). *Контекстное меню* вызывается нажатием правой кнопки мышки на конкретном объекте и содержит команды, которые могут быть применены именно к этому объекту.

**Создание главного меню**

Главное меню окна представлено в Swing классом JMenuBar. По сути своей этот класс представляет собой панель с менеджером расположения BoxLayout (по горизонтали), в которую можно добавлять не только элементы меню, но и что угодно: хоть выпадающие списки, хоть панели с закладками. Однако для удобства пользования программой предпочтительнее использовать «традиционные» возможности меню.

Главное меню должно быть присоединено к окну методом setJMenuBar(JMenuBar menuBar).

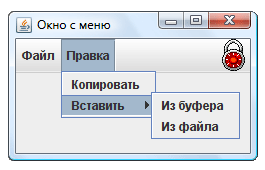
Элементами главного меню являются обычные меню — выпадающие прямоугольные блоки команд — объекты класса JMenu. Конструктор JMenu(String title) принимает один параметр: название меню, которое будет отображаться в строке главного меню.**\***

Меню, в свою очередь, состоит из *пунктов меню*, представленных классом JMenuItem. По логике работы пункты меню аналогичны кнопке JButton, то есть, при нажатии на него пользователем выполняется какое-то действие.

*об особых разновидностях пунктов меню*

Создать элемент меню можно пустым конструктором JMenuItem либо (что более востребовано) одним из конструкторов, в которые передается текст и/или значок элемента меню: JMenuItem(String text), JMenuItem(Icon icon), JMenuItem(String text, Icon icon). Конечно, в любой момент текст и значок можно сменить методами setText(String text) и setIcon(Icon icon) соответственно.

Элемент добавляется в меню методом add(JMenuItem item) класса JMenu. Чтобы отделить группы взаимосвязанных элементов меню, можно добавить между ними разделитель методом addSeparator() класса JMenu.

Кроме того, в меню можно добавить и другое меню. В этом случае образуется последовательность вложенных друг в друга подменю, что довольно часто встречается в современных программах. Но не следует увлекаться: глубина вложенности более трех уровней скорее всего приведет к неудобствам пользования программой.

Создадим главное меню окна, состоящее из двух подменю: «Файл» и «Правка», причем в меню «Правка» поместим выпадающее подменю. Кроме того, воспользуемся знаниями о менеджере расположения главного меню, чтобы добавить с правого края значок (наподобие того, как это сделано в браузере Internet Explorer).

SimpleWindow(){

**super**("Окно с меню");

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

JMenuBar menuBar = **new** JMenuBar();

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

fileMenu.add(**new** JMenuItem("Новый"));

fileMenu.add(**new** JMenuItem("Открыть", **new** ImageIcon("1.gif")));

fileMenu.add(**new** JMenuItem("Сохранить"));

fileMenu.addSeparator();

fileMenu.add(**new** JMenuItem("Выйти"));

JMenu editMenu = **new** JMenu("Правка");

editMenu.add(**new** JMenuItem("Копировать"));

JMenu pasteMenu = **new** JMenu("Вставить");

pasteMenu.add(**new** JMenuItem("Из буфера"));

pasteMenu.add(**new** JMenuItem("Из файла"));

editMenu.add(pasteMenu);

menuBar.add(fileMenu);

menuBar.add(editMenu);

menuBar.add(Box.createHorizontalGlue());

menuBar.add(**new** JLabel(**new** ImageIcon("2.gif")));

setJMenuBar(menuBar);

setSize(250,150);

}

**Обработка команд меню**

Как было сказано, каждый элемент меню по сути представляет собой кнопку, поэтому реагирует на те же события и позволяет присоединять к себе тех же слушателей, что и кнопка.**\*** Поэтому мы можем назначить реакцию на команды меню точно также, как делали это выше. Например, чтобы при выборе в меню элемента «Выйти» программа прекращала свою работу, следует заменить в примере команду

fileMenu.add(**new** JMenuItem("Выйти"));

на последовательность команд:

JMenuItem exit = **new** JMenuItem("Выйти");

exit.addActionListener(**new** ActionListener() {

**public void** actionPerformed(ActionEvent event) {

System.exit(0);

}

}

);

fileMenu.add(exit);

Командой System.exit(0) прекращается работа текущего приложения.

**Создание контекстного меню**

Контекстное (или всплывающее) меню реализовано в классе JPopupMenu, очень похожему на класс JMenu. Отличительным методом этого класса является метод show(Component comp, int x, int y), отображающий меню в точке с заданными координатами относительно границ заданного компонента.

Как уже отмечалось, контекстное меню, как правило, отображается при щелчке правой кнопкой мыши над компонентом. Таким образом, чтобы отобразить меню, требуется добавить к этому компоненту слушателя мыши.

Рассмотрим пример, в котором к уже созданному нами окну добавляется метка с надписью и контекстное меню из двух элементов, связанное с этой меткой. Необходимо добавить в конструктор перед последней командой (setSize) следующий код:

label = **new** JLabel("КНИЖКА", **new** ImageIcon("1.gif"), JLabel.RIGHT);

JPanel panel = **new** JPanel();

panel.add(label);

popup = **new** JPopupMenu();

popup.add(**new** JMenuItem("Прочитать"));

popup.add(**new** JMenuItem("Сжечь"));

label.addMouseListener(**new** MouseAdapter(){

**public void** mouseClicked(MouseEvent event) {

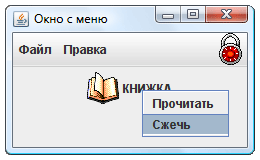
**if** (SwingUtilities.isRightMouseButton(event))popup.show(label, event.getX(), event.getY());

}

}

);

setContentPane(panel);

При этом в определении класса окна следует ввести два поля:

JPopupMenu popup;JLabel label;

Это необходимо для того, чтобы анонимный класс-слушатель мог обратиться к метке и к контекстному меню.

Приведенный пример также иллюстрирует применение полезного метода isRightMouseButton(MouseEvent event), определенного в классе вспомогательных утилит SwingUtilities. Метод получает информацию о событии мыши и отвечает на вопрос, была ли нажата именно правая кнопка мыши. Методы getX() и getY() возвращают координаты курсора мыши относительно наблюдаемого компонента.

**Интерфейс Action (действие) и класс AbstractAction**

Очень часто одна и та же команда дублируется в графическом интерфейсе в нескольких местах. Например, команда «Копировать» может присутствовать в главном меню окна, в контекстном меню некоторого объекта, а также в виде кнопки на инструментальной панели. Это необходимо для удобства пользования программой. Однако, привязывать ко всем этим объектам (кнопкам и командам меню) одинаковых слушателей довольно утомительно. Более того, все эти команды одинаково должны реагировать на изменения состояния программы: если копирование по каким-то причинам становится невозможным, то все элементы управления, представляющие данную команду, должны сделаться неактивными.

Для упрощения работы в таких ситуациях предусмотрен интерфейс Action. Он собирает в одном месте все, что относится к некоторому действию: обработчик, показатель активности, всплывающую подсказку, значок и т.д.

Интерфейс Action унаследован от интерфейса ActionListener, поэтому его главный метод — actionPerformed(ActionEvent event). Именно здесь программируется само действие. Но помимо этого посредством метода putValue(String key, Object value) можно задать одно из дополнительных свойств действия. Первый параметр — строковый идентификатор свойства. Он может принимать одно из значений, описанных в константах интерфейса Action. Второй параметр — объект, представляющий собой значение свойства.

Основные свойства интерфейса Action (точнее, соответствующие им константы):

NAME — имя действия,

SMALL\_ICON — значок, соответствующий действию,

SHORT\_DESCRIPTION — краткое описание действия (для всплывающей подсказки).

Метод setEnabled(boolean enabled) позволяет сделать действие активным или неактивным.

На основе созданного действия можно создавать некоторые элементы управления, передавая это действие в качестве единственного параметра конструктора. К таким элементам управления, в частности, относятся элементы меню и кнопки.

Недостаток интерфейса Action — в нем слишком много вспомогательных абстрактных методов (их семь, в том числе setEnabled() и putValue()) и программировать их достаточно утомительно. Поэтому обычно используется реализующий данный интерфейс класс AbstractAction, в котором «не заполнен» единственный метод — actionPerformed(), а его все равно необходимо определить для программирования сути действия.

Рассмотрим пример.

public class SimpleWindow extends JFrame {

private ExitAction exitAction;

SimpleWindow(){

super("Окно с меню");

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

exitAction = new ExitAction();

DeactivateAction deactivateAction = new DeactivateAction();

JMenuBar menuBar = new JMenuBar();

JMenu fileMenu = new JMenu("Файл");

fileMenu.add(new JMenuItem("Новый"));

fileMenu.addSeparator();

fileMenu.add(deactivateAction);

fileMenu.add(exitAction);

menuBar.add(fileMenu);

setJMenuBar(menuBar);

JToolBar toolBar = new JToolBar("Панель инструментов");

toolBar.add(exitAction);

toolBar.add(deactivateAction);

getContentPane().add(toolBar, BorderLayout.NORTH);

JPanel panel = new JPanel();

panel.add(new JButton(exitAction));

panel.add(new JButton(deactivateAction));

getContentPane().add(panel);

setSize(250,250);

}

class ExitAction extends AbstractAction {

ExitAction(){

putValue(Action.NAME, "Выйти");

putValue(Action.SHORT\_DESCRIPTION, "Программа перестанет работать, а окно исчезнет с экрана.");

putValue(Action.SMALL\_ICON, new ImageIcon("2.gif"));

}

public void actionPerformed(ActionEvent event) {

System.exit(0);

}

}

class DeactivateAction extends AbstractAction {

DeactivateAction(){

putValue(Action.NAME, "Запретить выход");

putValue(Action.SMALL\_ICON, new ImageIcon("1.gif"));

}

public void actionPerformed(ActionEvent event) {

if (exitAction.isEnabled()) {

exitAction.setEnabled(false);

putValue(Action.NAME, "Разрешить выход");

} else {

exitAction.setEnabled(true);

putValue(Action.NAME, "Запретить выход");

}

}

}

}

Здесь в окне описаны два внутренних класса-действия, одно из которых — это выход из программы. Объект этого класса, exitAction, является внутренним полем класса-окна и на его основе создается пункт меню и две кнопки. Второй класс-действие активирует и деактивирует объект exitAction (при этом изменяется соответствующий действию текст). Запустите пример и пронаблюдайте за тем, как выглядят и ведут себя объекты на основе Action, а затем внимательно проанализируйте его.

**Стандартные диалоговые окна**

**Класс JOptionPane**

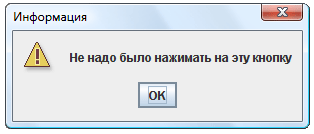
Неотъемлемой частью большинства программ являются небольшие диалоговые окна: для вывода пользователю сообщения (например, сообщения об ошибке), для вопроса, ответ на который важен для выполнения текущего действия (например, просьба подтвердить или отменить запрашиваемую операцию). Эти диалоги могут быть запрограммированы вручную на основе класса JFrame. Однако ввиду того, что они являются типичными для многих программ, Swing предоставляет в распоряжение программиста несколько готовых классов для работы с ними.

Чаще всего используется класс JOptionPane, содержащий несколько статических методов, отображающих стандартные диалоги.

Метод showMessageDialog() выводит на экран диалоговое окно, информирующее пользователя. Оно содержит надпись, значок и кнопку ОК. Существует несколько разновидностей этого метода с разными наборами параметров. Самый простой из них showMessageDialog(Component component, Object content) требует указания компонента, над которым должно появиться диалоговое окно и содержимого окна. Чаще всего содержимым окна является некоторая строка, а вместо первого параметра передается **null** — тогда окно появляется по центру экрана. Более «продвинутый» вариант showMessageDialog(Component component, Object content, String title, int type) позволяет задать также свой заголовок окна и выбрать его тип (влияет на иконку в окне): сообщение об ошибке (ERROR\_MESSAGE), предупреждение (WARNING\_MESSAGE), информация (INFORMATION\_MESSAGE).

Диалоговое окно является модальным. Это значит, что пока пользователь не нажмет в этом окне кнопку OK, программа окажется заблокирована — пользователь не сможет работать с другими окнами.

Пронаблюдаем действие метода на простом примере:



public class SimpleWindow extends JFrame {

**private** JButton button;

SimpleWindow(){

**super**("Предупреждающий диалог");

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

button = **new** JButton("Информация");

button.addActionListener(**new** ActionListener() {

**public void** actionPerformed(ActionEvent event) {

JOptionPane.showMessageDialog(button, "Не надо было нажимать на эту кнопку", "Информация", JOptionPane.WARNING\_MESSAGE);

}

}

);

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

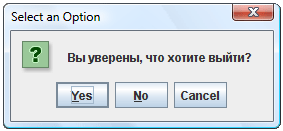
getContentPane().add(button);

setSize(200,150);

}

}

Кнопка button сделана полем класса окна, чтобы можно было получить к ней доступ из анонимного класса-слушателя.

Другое часто используемое диалоговое окно — окно вопроса. В этом окне несколько кнопок, одну из которых пользователь должен нажать. В программу, вызывающую это диалоговое окно, возвращается информация о выборе пользователя, на основе которой и строится дальнейший ход работы программы.

Данное окно отображается методом showConfirmDialog(Component component, Object content). Параметры этого метода идентичны по смыслу параметрам showMessageDialog(), но в диалоговом окне появится не одна кнопка, а три: «Yes», «No» и «Cancel». Более того, метод возвращает значение, которое можно сравнить с константами YES\_OPTION, NO\_OPTION и CANCEL\_OPTION. Логику работы с этим методом проиллюстрирует пример:

**public class** SimpleWindow extends JFrame {

JButton button;

SimpleWindow(){

**super**("Предупреждающий диалог");

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

button = **new** JButton("Выход");

button.addActionListener(**new** ActionListener() {

public void actionPerformed(ActionEvent event) {

**if** (JOptionPane.showConfirmDialog(button, "Вы уверены, что хотите выйти?") == JOptionPane.YES\_OPTION) System.exit(0);

}

}

);

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

getContentPane().add(button);

setSize(200,150);

}

}

Метод имеет еще четыре разновидности с разным набором параметров, позволяющих изменять заголовок и значок окна, а также набор кнопок.

**Класс JFileChooser**

Swing содержит готовое окно для выбора файла (полезное, например, для программирования пункта меню Файл --> Открыть). Объект класса JFileChooser создается простым конструктором без параметров, после чего может выводиться на экран методом showOpenDialog(). Этот метод возвращает результат действий пользователя по выбору файла, который сравнивается с одной из следующих констант:

APPROVE\_OPTION — выбор файла прошел успешно. Теперь можно методом getFile() получить выбранный файл.

CANCEL\_OPTION — пользователь отменил выбор файла, щелкнув на кнопке Cancel.

ERROR\_OPTION — при выборе файла произошла ошибка, либо пользователь закрыл диалоговое окно крестиком.

Метод showSaveDialog() отображает то же самое окно, но теперь оно работает в режиме сохранения. Пользователь выбирает директорию для сохранения файла и может ввести его имя. Метод возвращает результат того же типа, что и showOpenDialog(). Если выбор пути для сохранения прошел успешно, вызов метода getFile() вернут путь, куда пользователь желает сохранить файл.

Следует иметь в виду, что сам класс JFileChooser ничего не открывает и не сохраняет. Он только возвращает путь к выбранному пользователем файлу. А открыть его или сохранить файл в заданном месте должна уже сама программа.

Метод setDialogTitle(String title) позволяет задать окну заголовок.