****

Программирование на языке Java.

**Swing**.



Компиляция Latul Gh., USM

**Создание графического пользовательского интерфейса (GUI –ГПИ) на Java**

**Swing**

Оглавление

[1. Java Foundation Classes 4](#_Toc21887643)

[2. AWT 4](#_Toc21887644)

[3. Компоненты Swing – это легковесные компоненты AWT 5](#_Toc21887645)

[4. Архитектура MVC 7](#_Toc21887646)

[5. Модель событий 20](#_Toc21887647)

[5.1 Наблюдатели 20](#_Toc21887648)

[5.2 Слушатели 22](#_Toc21887649)

[5.3 Схема именования событий JavaBeans 25](#_Toc21887650)

[5.4 Техника написания слушателей 30](#_Toc21887651)

[4.4.1 Адаптеры 31](#_Toc21887652)

[4.4.2 Каждому событию – по слушателю 34](#_Toc21887653)

[4.4.3 Внутренние классы 34](#_Toc21887654)

[4.4.4 Использование анонимных классов 36](#_Toc21887655)

[4.4.5 Диспетчеризация 39](#_Toc21887656)

[5.5 Создание собственных событий 41](#_Toc21887657)

[5.6 События от мыши и метод contains() 49](#_Toc21887658)

[6. За кулисами системы обработки событий 51](#_Toc21887659)

[6.1 Поток EventDispatchThread и очередь событий EventQueue 51](#_Toc21887660)

[6.2 Доставка событий методам processXXXEvent() 53](#_Toc21887661)

[6.3 Маскирование и поглощение событий 56](#_Toc21887662)

[6.4 Работа с очередью событий 57](#_Toc21887663)

[6.5 Влияние на программы потока EventDispatchThread 59](#_Toc21887664)

[7. Создание графического интерфейса пользователя с помощью библиотек AWT и Swing 63](#_Toc21887665)

[7.1. Иерархия схема классов Java Swing (схема) 63](#_Toc21887666)

[7.2. Основные классы библиотек AWT и Swing 63](#_Toc21887667)

[7.2.1 Класс Component 64](#_Toc21887668)

[7.2.2 Класс Container 65](#_Toc21887669)

[7.2.3 Классы окон 65](#_Toc21887670)

[7.2.3.1 Класс JFrame 65](#_Toc21887671)

[7.2.3.2 Класс JWindow 65](#_Toc21887672)

[7.2.3.3 Класс JDialog 66](#_Toc21887673)

[7.2.3.4 Класс JFileChooser 66](#_Toc21887674)

[7.3. Графический контекст, рисование 67](#_Toc21887675)

[7.4. Класс Color, установка цвета 68](#_Toc21887676)

[7.5. Класс Font, установка шрифтов 69](#_Toc21887677)

[7.6. Класс Polygon 70](#_Toc21887678)

[7.7. Графические изображения 71](#_Toc21887679)

[7.8. Элементы управления 72](#_Toc21887680)

[7.9. Текстовые метки 73](#_Toc21887681)

[7.10. Управляющие кнопки 74](#_Toc21887682)

[7.11. Флажки – переключатели 74](#_Toc21887683)

[7.1. Группы кнопок («радиопереключатели») 74](#_Toc21887684)

[7.12. Раскрывающиеся списки 75](#_Toc21887685)

[7.2. Списки 75](#_Toc21887686)

[7.3. Текстовые поля 76](#_Toc21887687)

[7.4. Текстовые области 77](#_Toc21887688)

[7.5. Менеджеры компоновки (размещения) – Layout Manager 77](#_Toc21887689)

[8. Примеры Swing приложений 83](#_Toc21887690)

[9. Механизм управления событиями в java. 88](#_Toc21887691)

[Библиография и ссылки 90](#_Toc21887692)

[Приложения. Anexe 91](#_Toc21887693)

[1. Приложение Часы – Clock 91](#_Toc21887694)

[2. Вывод картинки в окне - Frame 92](#_Toc21887695)

[3. Использование таблиц swing - JTable 93](#_Toc21887696)

[4. Использование таблиц Swing - JTable (Render-Визуализатор) 96](#_Toc21887697)

[5. Пример вывода информации из базы данных в таблицу Jtable Swing 102](#_Toc21887698)

# Java Foundation Classes

Одной из самых больших и самых важных частей платформы J2SE является набор библиотек под общим названием Java Foundation Classes (JFC). Именно эти библиотеки предназначены для создания эффектных и отточенных пользовательских интерфейсов. Ниже перечислены библиотеки, входящие в набор Java Foundation Classes:

* ***Swing***. Важнейшая часть JFC; содержит компоненты для создания пользовательского интерфейса, такие как таблицы и текстовые поля, а также инструменты для работы с этими компонентами.
* ***Java2D***. Вторая по важности в JFC библиотека, позволяющая применять в своем приложении современную двухмерную графику, в том числе аффинные преобразования, дробные координаты, сглаживание, расширенные операции с растрами и многое другое. Библиотека Java2D встроена в Swing – все компоненты последней используют ее для вывода своих данных на экран, хотя и неявно.
* ***Accessibility***. Набор классов и интерфейсов, следующих промышленному стандарту и наделяющих приложения средствами поддержки пользователей с ограниченными возможностями. С помощью Accessibility можно, например, передать текстовое описание интерфейса системе синтеза речи, что позволит работать с программой пользователям с нарушениями зрения. Во всех компонентах Swing интерфейсы Accessibility уже реализованы и после небольшой настройки самый обычный интерфейс моментально превращается в специализированный, ориентированный на пользователей с ограниченными возможностями. Для приложений высшего уровня качества это свойство Swing просто незаменимо.
* ***Drag’n’Dro****p*. Дополнение, позволяющее приложению взаимодействовать с приложениями операционной системы пользователя или другими Java-приложениями с помощью технологии перетаскивания (drag and drop). Подобная возможность очень удобна для пользователя и позволяет ему сразу же забыть о том, что приложение написано на Java и не имеет практически никаких связей с его операционной системой. Напрямую с этим дополнением работать, как правило, не приходится, так как Swing берет многие детали процесса на себя.

Ядром Java Foundation Classes без всяких сомнений является библиотека Swing, остальные части набора классов так или иначе встроены в нее или предоставляют для компонентов этой библиотеки дополнительные возможности. Создавать пользовательский интерфейс своих приложений вы будете именно с помощью Swing, и именно эту библиотеку мы будем изучать. Для начала мы познакомимся с основными свойствами Swing, ее общей структурой, а затем перейдем к компонентам библиотеки. Они позволят создавать фантастические пользовательские интерфейсы всего одним мановением руки.

# AWT

ррррр

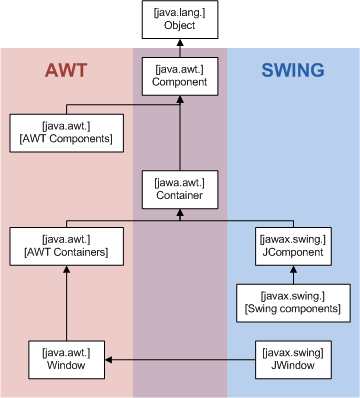
# Компоненты Swing – это легковесные компоненты AWT

Важнейшим отличием Swing от AWT является то, что компоненты Swing вообще не нуждаются в поддержке операционной системы и поэтому гораздо более стабильны, и быстры. Такие компоненты в Java называются ***легковесными*** (lightweight). Одна из особенностей легковесных компонентов состоит в том, что они «не используют код, зависящий от платформы»). AWT в своем составе имеет и легковесные компоненты.

Итак, *легковесный компонент* – это просто область в пространстве экрана, занимаемом Java-приложением. Главные его атрибуты – это координаты в окне и размер. Для операционной системы легковесный компонент *вообще не существует*, потому что представляет собой всего лишь часть какого-то окна. Всю работу по поддержке легковесных компонентов берут на себя библиотека AWT и виртуальная машина Java. Для программиста не существует никакого отличия между легковесными и обычными компонентами, которые по аналогии стали называть ***тяжеловесными***(heavyweight), то есть имеющими связь с операционной системой и представленными в своем собственном окне. Связь с системой обеспечивают тяжеловесные контейнеры (обычно окна), в которых размещаются легковесные компоненты.

Благодаря избавлению от сомнительных связей с разнообразными операционными системами легковесные компоненты полностью находятся во власти программиста на Java и могут следовать всем его указаниям.

Легковесные компоненты позволили совершенно безболезненно перейти от работы с прямоугольными компонентами, залитыми цветом фона, к работе с компонентами абсолютно любой, даже самой фантастичной, формы. Действительно, прорисовка легковесного компонента – это всего лишь прорисовка области окна вашего приложения, и ни операционная система, ни виртуальная машина Java не заставляли закрашивать эту область одним цветом. Вы вообще могли не рисовать свой компонент, несмотря даже на то, что он присутствовал на экране. Такая гибкость, позволяющая, в частности, создавать прозрачные компоненты и компоненты любой формы, дорогого стоила – ведь все эти возможности доставались программисту практически «даром», не требуя хитроумных приспособлений для своей реализации.



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Component** | | | | | |
|  | | |  | | |
| **Container** | | | | | |
|  | | |  | | |
| **JComponent** | | | | | |
|  | | |  | | |
|  |  |  | |  |  |

java.awt

java.swing

**Компоненты Swing**

Рис. 1.2. Диаграмма наследования компонентов библиотеки Swing

Легко видеть, что библиотека Swing обзавелась новым базовым классом, который стал называться JComponent и который был унаследован от абстрактного класса Container, определяющего поведение контейнеров AWT. Создатели Swing постарались максимально облегчить переход от AWT к новой библиотеке, практически полностью сохранив в ней имена классов AWT и их методов. Все компоненты AWT имеют своих наследников в Swing, и имена классов этих компонентов отличаются лишь префиксом «J» (например, тяжеловесная кнопка Button имеет в Swing свой легковесный аналог – кнопку JButton). Новые классы Swing также имели полный набор методов старых классов AWT, что делало процесс перехода с AWT на Swing простым и безболезненным (достаточно было импортировать пакет javax.swing и добавить к именам классов букву «J»). Все это обусловило молниеносное распространение новой библиотеки и еще сильнее подогрело интерес к ней.

Эффективная прорисовка компонентов Swing на экране – одна из самых важных обязанностей класса JComponent, и выполняет он ее очень качественно. Включив в базовый класс Swing двойную буферизацию, разработчики библиотеки избавили эту библиотеку от проблем, связанных с мерцанием и медленной скоростью вывода на экран.

Хотя Swing и основана на AWT, разница между двумя этими библиотеками велика.

Swing предоставляет свои, слегка измененные ***тяжеловесные*** контейнеры высшего уровня: окна JWindow, JDialog и JFrame. Перечисленные классы имеют всю необходимую поддержку для компонентов Swing, которую обеспечивает так называемая корневая панель (root pane) – особый контейнер Swing.

**Совместное использование компонентов AWT и Swing.**  Нет никаких препятствий для совместного использования графических компонентов этих двух библиотек. Вы можете спокойно добавлять компоненты AWT и Swing в свой контейнер, потому что в основе их (как и в основе любой библиотеки пользовательского интерфейса, даже от стороннего производителя) лежат базовые классы Component и Container. Вопрос в том, будет ли подобная конструкция правильно работать. Решена также проблема нахождения тяжеловесных компонент «под» легковесными, однако избегать совместного использования легковесных и тяжеловесных компонентов (Swing позволяет создать любой интерфейс, не прибегая к AWT).

# Архитектура MVC

Архитектура, предназначенная для создания легко расширяемых и настраиваемых пользовательских интерфейсов называется архитектура ***модель – вид – контроллер*** (**Model/View/Controller**, MVC). Впервые эта архитектура появилась в языке в Smalltalk (~1980 г.), именно тогда родились основные концепции и внешний вид большинства известных компонентов. Идея этих компонентов затем была использована в Macintosh, после чего перешла к многочисленным последователям Macintosh. Эта архитектура остается одним из самых удачных объектно-ориентированных решений, и поэтому часто используется и сегодня.

Как нетрудно догадаться по названию, MVC состоит из трех частей.

***Модель*** (model) хранит данные компонента и позволяет легко, не обращаясь к самому компоненту, изменять или получать эти данные. Например, раскрывающийся список позволяет вывести на экран перечень элементов (обычно это строки). Вместо того чтобы включать методы для манипуляции элементами списка в класс раскрывающегося списка, можно предоставить отдельный класс, работающий исключительно с данными. Такой подход позволит разработчику сосредоточиться именно на той задаче, которой он занимается в данный момент: можно сначала подготовить данные (считать их из файла или сетевого соединения, отсортировать, локализовать и т. п.), а потом уже передать их раскрывающемуся списку для вывода на экран. Хранение данных отдельно от самого компонента также позволяет изменять структуру данных модели, не меняя функций компонента. Самое же главное состоит в том, что отдельная модель позволяет появляться «хорошим» классам, которые описывают данные в языке решаемой задачи, а не в языке той или иной библиотеки пользовательского интерфейса. Если нам понадобится список книг в библиотеке, мы назовем класс именно так, а не будем встраивать его в мудреные классы, подгоняя под нужды интерфейса.

***Вид*** (view) выводит данные на экран для представления их пользователю. Отделение вида от данных позволяет представлять одни и те же данные совершенно разными способами. Например, текст формата HTML можно вывести в разном виде: провести разметку документа, разместить изображения и ссылки, использовать различные шрифты, а можно показать HTML-документ как код, который состоит из набора тегов и текста среди них. Между тем данные для этих разных видов требуются одни и те же (текст формата HTML). Вспоминая пример с раскрывающимся списком, можно сказать, что он является видом, представляющим на экране набор элементов. Данные раскрывающегося списка можно было бы представить и в другом виде, например, в таблице.

***Контроллер*** (controller) определяет, как должны реагировать вид и данные модели в ответ на действия пользователя. Наличие в MVC контроллера позволяет использовать одни и те же данные и виды в разных целях. HTML-страница, например, может быть показана в браузере или в визуальном средстве создания страниц. Браузер может задействовать контроллер, который при щелчке на ссылке переходит на страницу, указанную в ссылке (полностью меняет данные модели, загружая в нее новую порцию HTML-текста), а визуальное средство, скорее всего, использует контроллер, вызывающий при щелчке на ссылке редактор свойств этой ссылки (который меняет лишь часть данных модели, относящихся к ссылке). Раскрывающемуся списку также не помешает пара контроллеров: один для списка, не позволяющего редактирование элементов, а другой для редактируемого списка. Не всегда, но, тем не менее, довольно часто, контроллер также выполняет надзирательные функции и не позволяет пользователям портить модели некорректными данными, проводя проверку их на правильность перед отсылкой в модель.

На рис.1 схематически изображено взаимодействие трех компонент MVC архитектуры рис. 1.





Рис. 1. Взаимодействия между моделью, видом и контроллером

Пользователь взаимодействует с программой, совершая различные действия (нажимая клавиши, перемещая мышь, щелкая ее кнопками и т. п.); информация о его действиях поступает в контроллер. **Контроллер** определяет, как обработать эти действия, и посылает сообщения **модели** и/или **виду**. При этом используются следующие обозначения: сплошные стрелки – это обращение к конкретному объекту с известным типом, а пунктирные стрелки – это оповещения заранее неизвестных объектов об изменении ситуации в системе. Предпочтительнее задействовать механизм оповещения (пунктирные стрелки), так как он позволяет избежать сильной связи между объектами, а это повышает гибкость системы. Особенно важен механизм оповещения (шаблон observer) для модели, что и показано на диаграмме. Как видно, модель на самом деле не знает ни о присоединенном к ней виде, ни об используемом контроллере. Она пассивна: ее данные меняет контроллер (или вы сами), а информацию об этих изменениях она рассылает заинтересованным объектам, которые заранее неизвестны. Благодаря этому модель по-настоящему независима от остальных частей MVC, и ее легко использовать с разными видами, контроллерами и несколькими компонентами. В качестве слушателя оповещений, которые рассылает модель при изменении своих данных, чаще всего выступает вид (он может быть не один), обновляющий изображение компонента в соответствии с новыми данными. Например, при нажатии пользователем клавиши в текстовом поле контроллер (если он допускает ввод текста) вставляет в модель новый символ, соответствующий нажатой клавише, модель оповещает присоединенный к ней вид об изменении, а вид отображает новый символ на экране

Кроме такого, наиболее естественного способа работы с компонентом (все изменения происходят в ответ на действия пользователя), можно непосредственно взаимодействовать с отдельными частями архитектуры MVC. Иногда это может быть очень кстати. Например, вы можете использовать данные модели, не затрагивая контроллер и вид, манипулировать ими, и все изменения, что в них происходят, автоматически появятся на экране. Можно не ждать событий от пользователя, а программно генерировать их в контроллере (реакция на них со стороны модели и вида будет такой же) – это может быть весьма удобным при автоматизированном тестировании интерфейса или при обучении. Очень сильной стороной архитектуры MVC является также ее динамичность: никто не запрещает вам менять виды, контроллеры и модели прямо во время работы программы. Одним словом, архитектура MVC совсем не тривиальна, и возможности ее действительно велики.

Замечание: контроллер одна из самых запутанных частей MVC, зачастую, в программах от него просто избавляются.

**Недостатки MVC.**

Слабым звеном оказывается контроллер. Контроллеру просто необходимо знать хотя бы что-то о том, какие данные хранит модель, как она это делает и как эти данные меняются. Например, нажатие клавиши для текстового поля означает вставку символа в документ, для кнопки – изменение состояния на нажатое, для раскрывающегося списка – выбор нового элемента. Каждый раз контроллер меняет довольно специфичные данные модели, и основное преимущество от использования его в качестве отдельного элемента системы – возможность заменять одни контроллеры другими – просто теряется.

То же самое можно сказать и о связи контроллера и вида. Эта связь сильная: вид хранит ссылку на контроллер, а контроллер на вид. Получается, что хотя формально MVC отделяет контроллер, последний фактически «намертво» привязан к определенному виду и модели этого вида. Если компонент предполагает несколько сложных реакций на действия пользователя (к таким компонентам можно отнести, например, текстовые компоненты), контроллер в той или иной форме можно оставить, в противном случае он лишь вносит дополнительные сложность и путаницу.

Решение напрашивается само собой – надо просто соединить в единое целое контроллер и вид, образовав визуально-поведенческую (*look and feel*) часть компонента. Это целое и будет общаться с моделью. Такой подход не просто больше подходит для Java, но еще и более удобен для программирования: например, если компонент нужно отключить, то логичнее вызвать какой-то метод компонента (который сразу отключит обработку событий и сменит внешний вид), а не тасовать контроллеры и виды.

Немногие языки реализуют каноническую архитектуру MVC. Большинство объединяют контроллер и вид, чтобы упростить библиотеку и улучшить взаимодействие ее частей.

**Решение Swing – представители пользовательского интерфейса**

Итак, мы выяснили, что классическая архитектура MVC не очень хорошо подходит для Java: разделение контроллеров, моделей и видов не совсем оправдано. Гораздо более простым в реализации и дальнейшем использовании оказывается решение, совмещающее вид и контроллер в одно целое. Именно такое решение было выбрано разработчиками Swing. Посмотрим, что у них получилось (рис. 1.4).





Рис. 1.4. Организация представителей. Архитектура *Model-Delegate*

Как видно из диаграммы, разработчики Swing объединили вид и контроллер в новый элемент, который назвали представителем (delegate) пользовательского интерфейса (User Interface, UI). Теперь все действия пользователя поступают не в контроллер, определяющий реакцию на них, а в этот новый элемент, в котором происходит значительная часть работы. Он определяет, нужно ли реагировать на них (так как контроллер теперь находится внутри, исчезает необходимость переделывать его, чтобы отключить реакцию на действия пользователя – свойство «включено/выключено» стало свойством представителя) и, если нужно, то сразу же без генерации ка ких-либо событий и изменения данных меняет вид (это происходит быстро – вид и контроллер находятся в одном месте и имеют исчерпывающую информацию друг о друге, благодаря этому пользовательский интерфейс быстро реагирует на любые изменения и позволяет легко воспроизвести самые сложные операции по изменению данных), а уже после этого представитель говорит модели о том, что данные изменились, и их необходимо обновить. Модель обновляет хранящиеся в ней данные и оповещает заинтересованных субъектов (чаще всего того же представителя) об изменениях. В ответ внешний вид компонента окончательно обновляется, чтобы соответствовать новым данным.

Если рассматривать диаграмму подробней, оказывается, что схема работы новой системы очень проста, но весьма эффективна. Отдельного контроллера теперь нет, и ему не придется прикладывать титанических усилий, чтобы оставаться многократно используемым и полезным элементом системы, способным работать с любой моделью. Вид четко знает, какой элемент он представляет на экране, то есть данные модели, как бы она ни была реализована, он выведет на экран надлежащим образом (с видом вообще нет проблем). Находящийся теперь с ним в «одном флаконе» контроллер может проще решить свою задачу обработки событий пользователя: когда нужно, он сразу изменяет вид (это необходимо для смены ка ких-либо аспектов вида, не связанных с данными модели, как правило, эстетических, во имя удобства пользователя). Взаимодействие контроллера и вида реализовано внутри представителя пользовательского интерфейса, поэтому происходит быстро. Завершающим этапом является отправка совмещенным видом-контроллером сообщения модели о необходимости обновления данных (представитель знает, как работает отображаемая им модель) и рассылка моделью оповещения о новых данных. Это оповещение дает возможность всем присоединенным к модели видам прорисовать новые данные, а остальным слушателям узнать, что пользователь внес в данные изменения.

Надо сказать, что разделение функций компонента только на две части (UI-представителя и модель) как нельзя лучше подходит для Swing. Разработчики Swing не стремились создать ка кой-то новый тип пользовательского интерфейса, они, прежде всего, должны были обеспечить поддержку компонентами всех известных платформ таким образом, чтобы Java-приложения по возможности внешне не отличались от приложения для конкретной платформы. UI-представите ли позволяют легко реализовать такое поведение. Оставляя функции компонента и модели данных в стороне, нужно лишь изменить UI представителя, чтобы он реагировал на действия, характерные для ка кой-то платформы, и выводил компонент на экран в привычном для пользователей этой платформы виде.

Таким образом, слегка изменив и во многом упростив изначальную архитектуру MVC, создатели Swing сумели сохранить ее основные достоинства: простое изменение внешнего вида и поведения компонентов (это осуществляется заменой UI-представителя компонента), а также мощь модельного программирования (программист по-прежнему может использовать различные модели для одного компонента, менять данные и манипулировать ими, не заботясь об обновлении вида и о типе компонента, описывать данные в терминах решаемой задачи). Не совсем правильно говорить, что в Swing задействована архитектура MVC (в библиотеке реализована архитектура из двух частей, а не из трех), поэтому часто говорят об использовании в Swing отношения ***модель-представитель*** (*model delegate*), или об ***архитектуре с разделенной моделью***(*separable model architecture*).

**Как все работает**

Кажется, мы дошли до сути механизма, обеспечивающего компонентам Swing различные поведение и внешний вид. Имеется UI-пред ста ви тель, обрабатывающий события пользователя и рисующий компонент на экране; есть модель, хранящая данные компонента. Непонятно одно – как этот механизм взаимодействует с классами библиотеки Swing, такими как кнопки (JButton) или списки (JList). Работать с ними просто – вы создаете экземпляр класса кнопки и добавляете его в контейнер. Где же UI-представитель? Очевидно, что не в классах компонентов, иначе менять их внешний вид и поведение было бы невозможно (пришлось бы переписывать все эти классы для поддержки другого внешнего вида). Рисунок 1.5 иллюстрирует роль, которую играет класс компонента во взаимоотношениях UI-пред ста ви те ля, модели и конечных пользователей (к ним относятся программисты-клиенты Swing). Можно сказать, что класс компонента – это точка приложения сил архитектуры «модель-представитель», в нем сосредотачивается информация о том, как UI-представитель взаимодействует с некоторой моделью. Модель не знает, с каким UI-представителем она сотрудничает и какой компонент ее использует, все что известно о модели – это то, что она есть. Раз модель существует, на нее должна быть ссылка. Хранится эта ссылка в классе компонента. UI-представитель связывается с моделью только через класс компонента. Прямой связи нет, и это главное условие гибкости и взаимозаменяемости. Таким образом осуществляется обмен данными между видом и моделью. Программисту, намеревающемуся использовать некоторый компонент, не придется думать о том, какой UI-представитель задействован в данный момент и как его соединить с моделью. От него требуется лишь настроить модель (или применить модель по умолчанию) и передать ее компоненту. Компонент знает, какого UI-представителя нужно использовать (он получает эту информацию от менеджера внешнего вида UIManager, о котором мы вскоре поговорим), и готов к работе.

Если использовать терминологию шаблонов проектирования, то можно сказать, что с точки зрения UI-представителей и моделей классы компонентов Swing действуют как посредники (mediators), обеспечивая слабую связанность системы. С точки же зрения программистов-клиентов Swing классы компонентов являются фасадами (facade) для архитектуры «модель-представитель» (применяя компонент, не обязательно задумываться о том, что происходит внутри него).

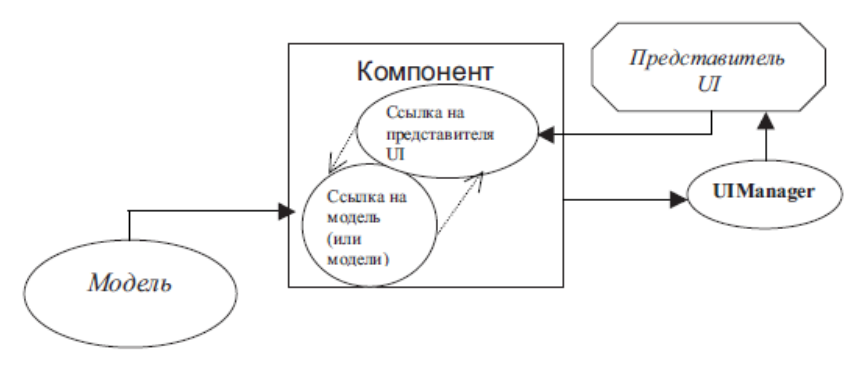


Рис. 1.5. Взаимоотношения UI-представителя и модели

Следует четко осознать, что именно классы компонентов (такие как JButton и JTable) являются основной частью библиотеки Swing. Может показаться, что они не так уж важны: ведь в них не происходит ни прорисовка компонента, ни обработка событий, ни манипуляция данными. Однако это не так: UI-представители и модели являются лишь частью внутреннего механизма библиотеки, а сами они не представляют большого интереса. Компонент просто делегирует к ним запросы: представитель осуществляет прорисовку и обработку событий, а модель хранит данные. Как мы уже выяснили, это обеспечивает великолепную гибкость библиотеки. Главными остаются компоненты Swing – все свойства (название кнопки, данные таблицы и т. п.) принадлежат им, они являются компонентами JavaBeans, ими вы манипулируете в своей программе или в визуальном средстве, именно они получают события от операционной системы и ресурсы для прорисовки, которые затем передают для выполнения действий в UI-представителей.

**Управление внешним видом и поведением программы**

В библиотеке Swing довольно много компонентов, и каждый из них имеет своего UI-представителя, ответственного за обработку событий и прорисовку компонента на экране. Рано или поздно настает момент, когда внешний вид и поведение вашего Java приложения приходится менять (например, чтобы оно выглядело одинаково с приложениями той платформы, на которой ему приходиться работать). Если бы разработчику пришлось менять UI-представителя индивидуально для каждого компонента, это было бы не только утомительно и долго, но и внесло бы множество ошибок. Поэтому в Swing управление внешним видом осуществляется в специальном классе UIManager. Он позволяет вам установить внешний вид и поведение для всех компонентов библиотеки сразу. Для этого нужно лишь вызвать статический метод этого класса setLookAndFeel() и передать в него объект класса LookAndFeel. Объект LookAndFeel – это хранитель информации об определенном внешнем виде и поведении программы, в нем содержится информация о UI-представителях, название внешнего вида, а также методы, упрощающие работу класса UIManager. По умолчанию компоненты автоматически «выбирают» себе UI-представителя именно с помощью класса UIManager. Внешние виды для компонентов Swing, которые поставляются c пакетом разработки JDK 1.6 последних ревизий (и список не планируется дополнять в новой версии 1.7), перечислены в табл. 1.1.

Таблица 1.1. Доступные внешние виды компонентов Swing

|  |  |  |
| --- | --- | --- |
| Название | **Местонахождение** | **Предназначение** |
| Внешний вид и поведение приложений на Java (внешний вид Metal, его современный подвид называется Ocean, или внешний вид, не зависящий от платформы) | Пакет javax.swing.plaf.metal  Класс MetalLookAndFeel | Именно этот внешний вид используется в Swing по умолчанию в версиях до 1.7 (если вы явно не установите другой). Специально разработан создателями Swing, для того чтобы придать приложениям на Java собственный уникальный вид. Подходит для всех платформ. Позволяет до некоторого предела менять цвета, шрифты и другие аспекты внешнего вида, но не слишком сильно – для этого предназначены так называемые «темы». |
| Полностью настраиваемый внешний вид Synth | Пакет javax.swing.plaf.synth Класс SynthLookAndFeel | Пакет, позволяющий менять внешний вид приложений с помощью изменяемых «обоев» (skins) или цветов. Сам он не определяет никакого вида, это система для поддержки реализации внешнего вида с помощью изображений или цветов. Может быть настроен как через классы, так и через файл настроек в формате XML. |
| Платформенно-независимый внешний вид Nimbus, предположительно замена для Metal | Пакет com.sun.java.swing.plaf. nimbus Класс NimbusLookAndFeel | Внешний вид нового поколения, построенный на базе настраиваемого вида Synth. Преимуществом данного вида без сомнения является возможность легко и самыми малыми штрихами менять внешний вид компонентов, «подгоняя» их под свои нужды. Внешне более тяготеет к Unix-стилю. Еще одним плюсом является использование векторной графики и соответственно независимость от разрешения экрана и возможность менять размеры компонентов. |
| Внешний вид и поведение Windows-приложени | Пакет com.sun.java.swing.plaf. windows Класс WindowsLookAndFeel | Этот внешний вид предназначен для эмуляции Windows-приложений. Его можно использовать только при работе под Windows. Отличается при работе на Windows XP и других версиях некоторыми визуальными отличиями и эффектами. Внешний вид Windows Vista пока что не реализован. |
| Внешний вид и поведение Unix-приложений | Пакет com.sun.java.swing.plaf. motif Класс MotifLookAndFeel | Позволяет приложениям на Java выглядеть аналогично Unix-приложениям с использованием среды CDE/Motif. Подходит для любых платформ (не только для Unix) |
| Mac Look & Feel |  |  |

Внешний вид Mac, так же как и внешний вид Windows, можно использовать только на соответствующей платформе (таковы требования корпораций Apple и Microsoft.

Специально для получения внешнего вида, соответствующего платформе, на которой работает приложение, в классе **UIManager** определен метод **getSystemLookAndFeel()**.

Лучше всего менять внешний вид и поведение перед тем, как на экране появится окно вашего приложения. Хотя никто не запрещает вам менять внешний вид прямо во время работы программы, в таком случае компоненты не изменятся автоматически, и вам придется вызывать специальный метод класса **SwingUtilities**, чтобы обновить их. Оптимальным для приложения является использование одного внешнего вида и одного варианта поведения: если вы создадите эффектное приложение с внешним видом Windows, полностью следуя рекомендациям Microsoft, вы не сможете перенести его на Unix (запрещено), и рекомендации Microsoft для интерфейса Unix не подходят, и это еще мягко сказано. Здесь на передний план выходит независимый от платформы внешний вид, **Nimbus** или **Metal**, специально созданный для Java-приложений.

Подключаемые внешний вид и поведение (Pluggable Look And Feel, PLAF) – одно из самых мощных свойств Swing. Никто не запрещает вам создать абсолютно новаторский трехмерный интерфейс или интерфейс, основанный только на системе синтеза и распознавания речи. Причем вам не нужно будет изменять ни строчки кода в вашем приложении.

**Специальные средства для пользователей с ограниченными возможностями**

Разработчики Swing учли важность этой поддержки и встроили во все компоненты особую информацию. Эта информация (она может быть передана специальным средствам, которые обработают ее надлежащим образом) описана во внутренних классах компонентов, использующих библиотеку Accessibility (эта библиотека относится к набору Java Foundation Classes). Итак, в каждом компоненте библиотеки Swing имеется внутренний класс, имя которого составляется из названия класса компонента и слова «Accessible», например, в классе кнопки (**JButton**) имеется внутренний класс **AccessibleJButton**. В классах **AccessibleXXX** содержится исчерпывающая информация о компонентах Swing (набор «ролей», которые исполняют графические компоненты в пользовательском интерфейсе; действия, которые можно совершать над компонентом, основные свойства компонента, например, текст надписи). Если на компьютере установлено специальное средство, оно находит эту информацию (вызывая для этого метод **getAccessibleContext()**, встроенный в общий класс всех компонентов Swing **JComponent**) и выводит ее в надлежащем виде (например, для человека с расстройством зрения интерфейс будет «прочитан» путем синтеза речи). Практически все известные средства для «чтения» интерфейсов или работы через специальные устройства поддерживают библиотеку Accessibility, к примеру, известная программа JAWS.

Т.е. вы просто пишите приложение, совершенно не задумываясь о том, что в дальнейшем оно может быть использовано человеком с ограниченными возможностями, и, тем не менее, вся необходимая информация будет на месте.

**Локализация приложений**

Библиотека Swing великолепно справляет с задачей представления приложений на нескольких языках, во многом потому, что сам язык Java создавался как язык, способный корректно работать в таких ситуациях. Вынос всего текста, используемого в приложении, во внешние файлы .properties, поддержка этими файлами всего спектра символов Unicode, а значит всех мировых языков, хранение всех строк в самой программе как Unicode, позволяют безболезненно переводить программу на любые языки. Поддержка часовых поясов, дат, времени, чисел всех мировых культур также встроена в Java и доступна в Swing.

# Модель событий

Графический пользовательский интерфейс (GUI) относится к *системам, управляемым по событиям* (event-driven systems). При запуске программы вы создаете пользовательский интерфейс, а затем ждете наступления некоторого события: нажатия клавиши, движения мыши или изменения компонента системы. При наступлении события программа выполняет необходимые действия, а затем снова переходит к ожиданию.

Библиотека Swing использует систему обработки событий JavaBeans.

Основным достоинством системы обработки событий Swing следует признать то, что как бы вы ни писали свою программу, какие бы способы создания интерфейса ни применяли, код, отвечающий за создание интерфейса, будет отделен от кода, обрабатывающего события. Это позволит и легко обновлять программу, и легко понимать ее.

# Наблюдатели

В программировании довольно часто возникает ситуация, когда при изменении данных в одном месте программы необходимо тут же обновить данные в другом месте программы. При изменении состояния одного объекта необходимо каким-либо образом изменить другой объект. Самое простое решение – организация непосредственной связи между объектами, то есть поддержание ситуации, когда объекты хорошо осведомлены о существовании друг друга, получим сильно связанные объекты – два объекта хранят ссылки друг на друга. Следовательно, при изменении программы придется переписывать много кода. Лучшее решение – использовать шаблон наблюдателя (observer).

Итак, объект, за которым ведется наблюдение, называется *субъектом* (subject). Объект, заинтересованный в изменениях субъекта, называются *наблюдателем* (observer). В обязанности субъекта входит добавление наблюдателей (в идеале количество наблюдателей произвольно), отправка наблюдателям сообщений об изменениях своего состояния и отсоединение ранее добавленных наблюдателей. Наблюдатели проще субъектов: они определяют методы, которые следует вызывать субъекту, для того чтобы сообщить о своих изменениях (рис. 2.1).

**Субъект (subject)** add() remove() notify()

notify() вызывает метод update()

**Наблюдатель (observer)** update()

**Наблюдатель (observer)** update()

**Наблюдатель (observer)** update()

Рис. 2.1. Взаимоотношения наблюдателей и субъектов

Рисунок показывает, что субъект обладает тремя методами: метод **add()** позволяет добавить очередного наблюдателя (как правило, наблюдатели хранятся в виде списка, что позволяет иметь произвольное их количество и легко манипулировать ими); метод **remove()** позволяет удалить ранее добавленного наблюдателя; метод **notify()** сообщает наблюдателям, добавленным ранее методом **add()**, о смене состояния субъекта. Для этого он вызывает определенный во всех объектах-наблюдателях специальный метод, в нашем случае это метод **update()**. В свою очередь наблюдатели знают, что очередной вызов метода **update()** означает смену состояния субъекта, и выполняют в этом методе все необходимые действия. Чтобы субъекты и наблюдатели смогли работать друг с другом, их функции описываются в базовых классах (или интерфейсах), и объекты, которым необходимо знать друг о друге, но которым нежелательно быть сильно связанными, наследуют от этих базовых классов или реализуют интерфейсы и начинают работать как субъекты и наблюдатели.

Объекты, изменения в которых могут быть интересны окружающим, становятся **субъектами** (то есть просто поддерживают список наблюдателей), а заинтересованным в их изменениях объектам нужно лишь определить методы наблюдателей и зарегистрировать свой интерес к субъектам. Никто не запрещает субъектам самим быть наблюдателями, и наоборот. В Java субъекты и наблюдатели могут работать особенно эффективно – возможность определять внутренние классы позволяет одному классу одновременно иметь информацию о нескольких совершенно разных субъектах и эффективно обрабатывать ее.

**Cубъектами** являются различные графические компоненты, из которых состоит интерфейс – кнопки, списки, текстовые поля, а программист определяет, что в них происходит (щелчки на кнопках, перемещения мыши), описывает объекты-наблюдатели и решает, как поступить при смене состояния интересующего его компонента. По сути, все действует по законам архитектуры «модель-вид-контроллер», только на более высоком уровне. Компонент представляет собой вид, в модели хранятся данные, так, как это удобно программе, ну а наблюдатель является контроллером, только уже более высокого уровня, который определяет что будет происходить в программе в ответ на происходящие события, то есть контроллером всей программы или ее части.

И модели, и UI-представители одновременно представляют собой субъектов и наблюдателей – при изменении данных модели она уведомляет об этом UI-представителя (выступающего в качестве наблюдателя), и тот обновляет внешний вид компонента в соответствии с новыми данными. Если же в ответ на действие пользователя меняется внешний вид компонента (а за это отвечает UI-представитель), то уже модель становится наблюдателем и получает уведомление о том, что данные необходимо изменить.

Cистема обработки событий Swing основана на отношении вида субъект-наблюдатель. **Субъектами** являются компоненты Swing (кнопки JButton, списки JList и т. п.), а наблюдателями – специальные объекты, которые называют ***слушателями***. Для того чтобы узнать о ка ком-либо событии, надо написать соответствующего слушателя и присоединить его к компоненту

# Слушатели

***Событие*** (event) в пользовательском интерфейсе – это либо непосредственное действие пользователя (щелчок или движение мыши, нажатие клавиши), либо изменение состояния ка кого-либо компонента интерфейса (например, щелчок мыши может привести к нажатию кнопки). ***Источником события*** (event source) в Swing может быть любой компонент, будь то кнопка, надпись с текстом или диалоговое окно. Для того чтобы узнавать в своей программе о происходящих в компоненте событиях, нам необходимо сообщить компоненту о своей заинтересованности. Сделать это можно, передав компоненту ***слушателя*** (listener) определенного события. Слушатель – этот тот самый наблюдатель (разница только в названиях), которому компонент будет сообщать о происходящих в нем событиях. Каждому типу события соответствует свой слушатель. Так гораздо проще следить именно за теми событиями, что нас интересуют, а не за всеми подряд – код, обрабатывающий щелчки мыши, будет отделен от кода, ответственного за нажатия клавиш. Обязанности слушателей (то есть методы, которые в них должны быть определены) описаны в соответствующих интерфейсах. Вы реализуете в своем классе нужный вам интерфейс слушателя, передаете его в интересующий вас компонент и спокойно ждете наступления события. Давайте рассмотрим простой пример, в котором мы попытаемся узнать о нажатиях клавиш при активном окне.

Пример обработки нажатия клавиш при активном окне.

// FirstEvents.java

// События - нажатия клавиш на клавиатуре

**import** javax.swing.\*;

**import** java.awt.event.\*;

**public** **class** FirstEvents **extends** JFrame {

**public** FirstEvents() {

**super**("FirstEvents");

// при закрытии окна - выход

setDefaultCloseOperation(***EXIT\_ON\_CLOSE***);

// регистрируем нашего слушателя

addKeyListener(**new** KeyL());

// выводим окно на экран

setSize(200, 200);

setVisible(**true**);

}

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

SwingUtilities.*invokeLater*(**new** Runnable() {

**public** **void** run() {

**new** FirstEvents();

}

});

}

}

// этот класс будет получать извещения о событиях

**class** KeyL **implements** KeyListener {

// печать символа

**public** **void** keyTyped(KeyEvent k) {

System.***out***.println(k);

}

// нажатие клавиши

**public** **void** keyPressed(KeyEvent k) {

System.***out***.println(k);

}

// отпускание нажатой клавиши

**public** **void** keyReleased(KeyEvent k) {

System.***out***.println(k);

}

}

java.awt.event.KeyEvent[KEY\_TYPED,keyCode=0,keyText=Unknown keyCode: 0x0,keyChar='**a**',keyLocation=KEY\_LOCATION\_UNKNOWN,rawCode=0,primaryLevelUnicode=0,scancode=0,extendedKeyCode=0x0] on frame0

java.awt.event.KeyEvent[KEY\_RELEASED,keyCode=65,keyText=**A**,keyChar='**a**',keyLocation=KEY\_LOCATION\_STANDARD,rawCode=65,primaryLevelUnicode=97,scancode=30,extendedKeyCode=0x41] on frame0

Пример очень прост – мы создаем класс, унаследованный от окна **JFrame**, устанавливаем для него размер методом **setSize()**, указываем, что при закрытии окна следует завершить работу приложения методом **setDefaultCloseOperation()**, и выводим окно на экран. Правда сам запуск приложения происходит через класс **SwingUtilities**, однако такова архитектура Swing. Посмотрим, как создается слушатель события.

Прежде всего, необходимо написать класс, ***реализующий интерфейс слушателя***. Для получения информации о нажатиях клавиш используется интерфейс **KeyListener**, он реализован в классе **KeyL**. В этом интерфейсе определены три метода, каждый из которых вызывается при наступлении определенного события: **keyPressed()** и **keyReleased()** – при нажатии и отпускании клавиши, **keyTyped()** – при печати символа (когда нажимается и отпускается клавиша, соответствующая печатному символу). В качестве параметра каждому методу передается объект **KeyEvent**, который используется для получения дополнительной информации о событии (кода клавиши, источника события и т. д.).

Последний этап – регистрация нашего слушателя в интересующем нас компоненте. Компонент у нас в программе только один – это наше окно. Для регистрации слушателя событий от клавиатуры мы вызываем метод **addKeyListener()**, в который передаем ссылку на объект класса **KeyL**. После этого остается только запустить программу.

Подобным образом обрабатываются все события в Swing. Когда вы собираетесь обработать ка кое-то событие, то, прежде всего, выясняете, какой слушатель получает сообщения о событиях этого типа, создаете класс, реализующий обязанности слушателя, и регистрируете его в интересующем вас компоненте. Это позволяет идентифицировать и обрабатывать лишь те события, которые нас интересуют, не выискивая нужное в потоке всех событий; причем события обрабатываются отдельно от компонентов, где эти события происходят, что позволяет точно отделять части программы, связанные с созданием пользовательского интерфейса, от частей, реализующих деловую логику программы.

# Схема именования событий JavaBeans

Все компоненты Swing являются компонентами JavaBeans. Основой JavaBeans является соглашение об именах, которое позволяет визуальным средствам легко узнавать, какими свойствами обладает компонент. Для этого компонент определяет набор методов со специальными именами **get/set**. Методы эти служат для считывания и записи значений свойств компонента.

Чтобы событие стало доступно визуальному средству разработки, необходимо проделать описанную ниже процедуру.

1. Определить **класс**, в котором будет храниться информация о произошедшем событии (что это будет за информация, определяет создатель события). Класс должен быть унаследован от базового класса java.util.EventObject и иметь название вида **XXXEvent**, где «**XXX**» – это название события. Например, **KeyEvent**.

2. Создать **интерфейс** слушателя, в который будет приходить информация о событии. Название интерфейса должно иметь следующий вид: **XXXListener**. Например, **KeyListener**. Этот интерфейс должен быть унаследован от базового интерфейса всех слушателей событий **java.util.EventListener** (это пустой интерфейс без методов, он просто помечает то, что унаследованный от него интерфейс является слушателем). В интерфейсе может быть определено сколь угодно много методов, единственное требование к этим методам – наличие параметра типа **XXXEvent**. Никаких других параметров у методов быть не должно.

3. Включить **поддержку события в класс** компонента, в котором это событие может происходить. Чтобы сделать это, необходимо определить два метода: один для присоединения слушателей, другой для их отсоединения. Названия методов должны выглядеть следующим образом: **addXXXListener()** – для метода, присоединяющего слушателей, и **removeXXXListener()** – для метода, отсоединяющего слушателей. В примере присоединение слушателя происходило с помощью метода **addKeyListener()**, определенного в классе окон **JFrame** (на самом деле этот метод определен в базовом классе всех компонентов **Component**, и добавлять слушателей клавиатуры можно к любому компоненту).

Если данные требования выполнены, то визуальное средство легко найдет все типы событий, поддерживаемые компонентом, соответствующих им слушателей и определит методы для работы с этими слушателями. Интуитивно понятно, как найти необходимые классы и интерфейсы для обработки событий.

**Стандартные события**

События в Java условно разделяются на ***низкоуровневые*** (low-level events) и ***высокоуровневые*** (high-level events). К низкоуровневым событиям относят те, что происходят непосредственно в результате действий пользователя: это движения мыши, передача фокуса ввода от одного приложения другому, нажатия клавиш и т. п. Они поступают в Java-программу от операционной системы или от внутренних механизмов виртуальной машины. Высокоуровневые события происходят в результате изменения состояния компонента. Такие события поступают не от операционной системы, а создаются самим компонентом. Процесс создания события еще называют ***запуском*** (fire). Во многих компонентах Swing вы можете увидеть методы с именами вида **fireXXX()**; именно в таких методах создаются объекты с информацией о событиях, которые затем рассылаются слушателям. Часто события высокого уровня возникают после того, как происходят несколько событий низкого уровня (например, кнопка сообщает о своем нажатии, после того как над ней была нажата и отпущена кнопка мыши).

Начнем мы с низкоуровневых событий (табл. 2.1). Эти события могут возникать в любом графическом компоненте, унаследованном от класса java.awt.Component (правда, есть несколько исключений).

Низкоуровневые события легко отличить от высокоуровневых – все они унаследованы от особого базового класса **AWTEvent**.

**Таблица 2.1**. Основные низкоуровневые события

|  |  |  |
| --- | --- | --- |
| **Краткое описание события** | **Методы слушателя** | **Источник события** |
| Событие от клавиатуры. Описано в классе **KeyEvent**. Возникает, когда пользователь нажимает клавишу | keyPressed(KeyEvent), keyReleased(KeyEvent), keyTyped(KeyEvent) | Все компоненты (наследники класса java. awt.Component) |
| Нажатия и отпускания кнопок мыши и попадание курсора в область компонента. Класс **MouseEvent** позволяет следить за состоянием кнопок мыши и контролировать нахождение указателя мыши в определенной области | mouseClicked(MouseEvent), mouseEntered(MouseEvent), mouseExited(MouseEvent), mousePressed(MouseEvent), mouseReleased(MouseEvent) | Все компоненты |
| Перемещение мыши. Класс события тот же – **MouseEvent**, но вот слушатель называется по-другому – **MouseMotionListener** | mouseDragged(MouseEvent), mouseMoved(MouseEvent) | Все компоненты |
| Прокрутка колесика мыши. Класс **MouseWheelEvent** | mouseWheelMoved(  MouseWheelEvent) | Все компоненты |
| Передача фокуса ввода. Класс **FocusEvent** | focusGained(FocusEvent), focusLost(FocusEvent) | Все компоненты |
| Добавление или удаление компонентов в контейнере. Класс **ContainerEvent** | componentAdded(ContainerEvent), componentRemoved(Container Event) | Все контейнеры (наследники класса java. awt.Container) |
| Изменения состояния окна. Класс **WindowEvent**. Позволяет узнать о перемещении, свертывании, закрытии окна и т. д. | windowActivated(WindowEvent), windowClosed(WindowEvent), windowClosing(WindowEvent), windowDeactivated(WindowEvent), windowDeiconifi ed(WindowEvent), windowIconifi ed(WindowEvent), windowOpened(WindowEvent) | Окна (наследники класса java.awt. Window), например, окна с рамкой (JFrame) или диалоговые окна (JDialog) |

В таблице нет названий слушателей и методов, предназначенных для добавления и удаления этих слушателей. Такие названия легко получить самим согласно правилам.

Рассмотрим небольшой пример:

// LowLevelEvents.java

// Наблюдение за основными низкоуровневыми событиями

**import** javax.swing.\*;

**import** java.awt.event.\*;

**import** java.awt.\*;

**public** **class** LowLevelEvents **extends** JFrame {

// сюда мы будем выводить информацию

**private** JTextArea out;

**public** LowLevelEvents() {

**super**("LowLevelEvents");

// при закрытии окна - выход

setDefaultCloseOperation(***EXIT\_ON\_CLOSE***);

// добавим текстовое поле

add(**new** JScrollPane(out = **new** JTextArea()));

// и кнопку

JButton button = **new** JButton("Источник событий");

add(button, "South");

// регистрируем нащего слушателя

OurListener ol = **new** OurListener();

button.addKeyListener(ol);

button.addMouseListener(ol);

button.addMouseMotionListener(ol);

button.addMouseWheelListener(ol);

button.addFocusListener(ol);

// выводим окно на экран

setSize(400, 300);

setVisible(**true**);

}

// внутренний класс - слушатель событий

**class** OurListener **implements** MouseListener, KeyListener, MouseMotionListener, MouseWheelListener, FocusListener {

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

out.append(e.toString() + "\n");

}

**public** **void** mousePressed(MouseEvent e) {

out.append(e.toString() + "\n");

}

**public** **void** mouseReleased(MouseEvent e) {

out.append(e.toString() + "\n");

}

**public** **void** mouseEntered(MouseEvent e) {

out.append(e.toString() + "\n");

}

**public** **void** mouseExited(MouseEvent e) {

out.append(e.toString() + "\n");

}

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

out.append(e.toString() + "\n");

}

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

out.append(e.toString() + "\n");

}

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

out.append(e.toString() + "\n");

}

**public** **void** mouseDragged(MouseEvent e) {

out.append(e.toString() + "\n");

}

**public** **void** mouseMoved(MouseEvent e) {

out.append(e.toString() + "\n");

}

**public** **void** focusGained(FocusEvent e) {

out.append(e.toString() + "\n");

}

**public** **void** focusLost(FocusEvent e) {

out.append(e.toString() + "\n");

}

**public** **void** mouseWheelMoved(MouseWheelEvent e) {

out.append(e.toString() + "\n");

}

}

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

SwingUtilities.*invokeLater*(**new** Runnable() {

**public** **void** run() {

**new** LowLevelEvents();

}

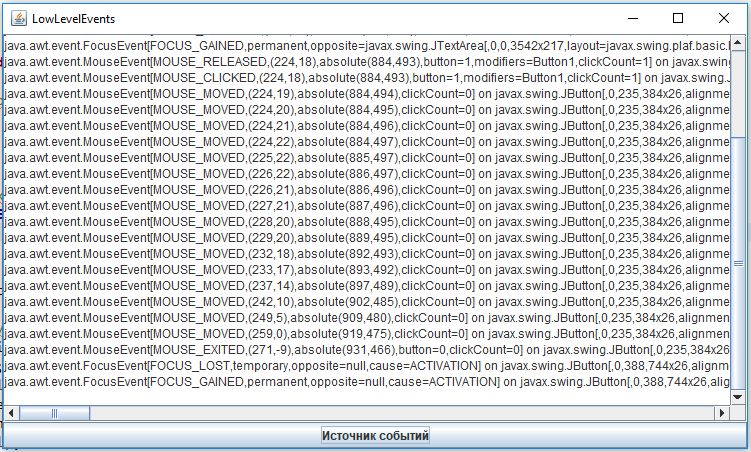
});

}

}

В этом примере создается окно, в центр добавляется текстовое поле (помещенное в панель прокрутки JScrollPane), а в нижнюю часть окна помещается простая кнопка JButton, которая и будет служить источником событий. Далее к кнопке присоединяются слушатели разнообразных событий.

Интерфейсы слушателей реализованы в классе OurListener. В примере этот класс реализует обязанности сразу пяти слушателей, так что методов в нем довольно много. Каждый из этих методов просто выводит информацию о происшедшем событии в текстовое окно, где ее можно изучить. Запустив приложение и совершая различные действия с кнопкой, вы сможете увидеть, когда, как и какие события в ней происходят. Посмотрите на рисунке, как работает эта программа.



**Таблица 2.2.** Наиболее часто используемые высокоуровневые события

|  |  |  |
| --- | --- | --- |
| **Краткое описание события** | **Методы слушателя** | **Источник события** |
| Событие **PropertyChangeEvent**. Обеспечивает работу механизма привязанных свойств JavaBeans | propertyChange (PropertyChangeEvent) | Практически все графические компоненты JavaBeans (в том числе все компоненты Swing) |
| Событие **ChangeEvent**. Запускается некоторыми компонентами и моделями для сообщения о своих изменениях | stateChanged  (ChangeEvent) | Некоторые компоненты Swing. Многие модели используют это событие для связи с UI- представителями |
| Событие **ActionEvent**. Сообщает о действии над компонентом | actionPerformed  (ActionEvent) | Компоненты Swing, у которых есть какое-то «главное» действие (например, у кнопки – нажатие) |

# Техника написания слушателей

Использование слушателей для обработки событий очень удобно не только потому, что позволяет разделять места возникновения событий (пользовательский интерфейс) и места их обработки. Слушатели также позволяют обрабатывать события так, как вам нужно, подстраивать схему обработки событий под свою программу, а не проектировать программу с тем расчетом, что в ней придется обрабатывать события. Вы не думаете о том, что придется делать при обработке событий, а просто пишете код, и в нужный момент используете один из доступных вариантов обработки события. В этом разделе мы рассмотрим наиболее популярные варианты обработки событий с помощью слушателей.

# Адаптеры

Интерфейсы некоторых слушателей, содержат не один, а несколько методов. В некоторых слушателях методов довольно много (например, в слушателе оконных событий WindowListener). Это вполне логично – каждый метод отражает свое небольшое изменение в компоненте. Создание для каждого простого события отдельного слушателя привело бы к появлению невообразимого количества интерфейсов с расплывчатым предназначением, а обработка сложного события в одном методе неудобна – придется предварительно выяснять, что же именно произошло.

Однако оказывается, что использование интерфейсов слушателей с несколькими методами тоже не совсем удобно. Чаще всего при обработке события какого-либо типа нас интересует нечто конкретное (например, щелчок мыши или закрытие окна), а не все возможные варианты происходящего события. Тем не менее, при реализации интерфейса Java обязывает определить все методы этого интерфейса, даже те, которые нас вообще не интересуют. Это не только неудобно, но и вносит некоторую путаницу в код, в котором появляются «пустые», ничего не выполняющие методы.

Чтобы избежать этих неудобств, в дополнении к слушателям библиотека предоставляет специальные классы, называемые адаптерами (adapters). Адаптер – это просто класс, который реализует интерфейс определенного слушателя, а значит, все его методы. Однако методы слушателя адаптер оставляет пустыми, без всякой реализации, что позволяет нам вместо реализации интерфейса слушателя наследовать адаптер от класса адаптера. При наследовании в дочерний класс переходят все методы базового класса, и остается переопределить только те из них, которые нас интересуют. Попробуем ис- пользовать адаптеры в следующем примере.

// Adapters.java

// Использование адаптеров вместо интерфейсов

**import** javax.swing.\*;

**import** java.awt.event.\*;

**import** java.awt.\*;

**public** **class** Adapters **extends** JFrame {

**public** Adapters() {

**super**("Adapters");

// при закрытии окна - выход

setDefaultCloseOperation(***EXIT\_ON\_CLOSE***);

// регистрируем слушателя

addMouseListener(**new** MouseL());

// выводим окно на экран

setSize(200, 200);

setVisible(**true**);

}

// наследуем от адаптера

**class** MouseL **extends** MouseAdapter {

// следим за щелчками мыши в окне

@Override

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

System.***out***.println(e);

}

}

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

//invokeLater заставляет doRun.run () выполняться асинхронно

//в потоке диспетчеризации событий AWT.

SwingUtilities.*invokeLater*(**new** Runnable() {

**public** **void** run() {

**new** Adapters();

}

});

}

//Подсистема оконной обработки JRE через SwingUtilities.invokeLater ()

//запускает три потока за раз:

// «AWT-Windows» (поток демона),

// «AWT-Shutdown» и «AWT-EventQueue-0».

// «AWT-EventQueue-0» известен как Thread-Dispatching Thread (EDT),

// который является единственным потоком, ответственным за обработку

// всех событий (например, нажатием кнопок) и обновлением дисплея

// для обеспечения безопасности потоков в графическом интерфейсе

// операций и манипулирования компонентами графического интерфейса.

}

java.awt.event.MouseEvent[MOUSE\_CLICKED,(60,112),absolute(60,112),

button=1,modifiers=Button1,clickCount=1] on frame0

java.awt.event.MouseEvent[MOUSE\_CLICKED,(78,85),absolute(78,85),

button=1,modifiers=Button1,clickCount=1] on frame0

В примере создается небольшое окно, к которому добавляется слушатель событий от мыши MouseListener. Однако реализовывать интерфейс слушателя мы не стали, а создали новый внутренний класс MouseL, унаследованный от класса адаптера MouseAdapter. Вместо того чтобы определять все методы слушателя (а их ни много, ни мало – целых пять, можете убедиться в этом, заглянув в табл. 2.1), мы переопределили только один. Здесь нас интересовало только, когда пользователь щелкает кнопками мыши в окне, так что нам пригодился метод mouseClicked(). Остальные четыре метода остались в стороне, как будто их и не было.

Для всех низкоуровневых событий из пакета java.awt.event, слушатели которых состоят более чем из одного метода, имеются адаптеры. Чаще всего они и используются при обработке событий, и только в тех редких случаях, когда программу интересуют все события определенного рода, задействуются интерфейсы. Узнать название класса адаптера очень просто: если слушатель называется XXXListener, то имя адаптера выглядит как XXXAdapter. Но вот для высокоуровневых событий компонентов Swing имеются только слушатели, адаптеров для них вы не найдете. Видимо, разработчики решили, что если хоть какое-то событие от компонента обрабатывается, то информация о нем нужна полная. Довольно странное решение.

Единственный недостаток адаптеров – меньший контроль со стороны компилятора и отсюда большая вероятность возникновения ошибки при работе с ними. Если методы интерфейса должны быть определены (иначе программа не будет компилироваться), то при наследовании от класса адаптера никаких ограничений нет, например, в только что рассмотренном примере в классе MouseL можно было определить следующий метод:

**M**ouseClicked(MouseEvent e)

Этот метод очень похож на правильный метод слушателя, предназначенный для получения сообщений о щелчках на кнопках мыши, но с его помощью вы не обработаете никакие события – первая буква оказалась прописной. Надо использовать аннотацию @Override которая, не дает коду компилироваться, если метод не переопределен из родительского класса

SwingUtilities.*invokeLater* выполняет ваш *Runnable* в потоке диспетчеризации событий AWT. Но зачем вам это делать? Swing не является потокобезопасным, и все изменения объектов Swing должны выполняться в Event Dispatch Thread. Поскольку структуры данных Swing не являются потокобезопасными, поэтому предоставить программистам легко достижимый способ предотвращения одновременного доступа к ним, дизайнеры Swing установили правило, что весь код, который обращается к ним, должен работать в одном потоке. Это происходит автоматически для кода обработки событий и отображения кода обслуживания, но если вы инициировали длительное действие - в новом потоке, конечно, как вы можете сигнализировать о его прогрессе или завершении? Вам нужно изменить элемент управления Swing, и вы должны сделать это из потока, отправляемого событиями. Следовательно, *invokeLater* запустит кусок кода в AWT-потоке. Это позволяет вам изменять графический интерфейс из других потоков.

# Каждому событию – по слушателю

Хорошо известно, что программа читается гораздо чаще, чем пишется. Процесс поддержки программ вообще занимает львиную долю времени разработки, поэтому удобочитаемость кода очень важна. А одним из важнейших условий удобочитаемости кода является четкое разделение его на объекты, каждый из которых занимается своим делом и не затрагивает другие. Однако, когда дело касается создания пользовательских интерфейсов, соблюсти это правило не так-то просто. Программе приходится работать по событиям пользовательского интерфейса, учитывать особенности этих событий, деловая логика программы смешивается с графическим интерфейсом, что приводит к совершенно неудобоваримому и плохо поддерживаемому коду.

В системе событий Swing каждому событию сопоставляется свой слушатель, который может быть описан весьма далеко от места возникновения события. Каждый слушатель представляет собой класс – обычный, внутренний или анонимный, в котором сосредотачивается обработка события. При таком подходе программа становится простой и понятной: с одной стороны, имеются классы и объекты, представляющие деловую логику программы, с другой – объекты и классы, реализующие ее пользовательский интерфейс. Отдельно от тех и других находятся объекты, обрабатывающие события, именно они увязывают происходящее в пользовательском интерфейсе с изменениями в деловой логике, и именно такой подход к обработке событий является оптимальным и наиболее естественным: обрабатывая событие, создавайте класс, и в этом классе объединяйте деловую логику и графический интерфейс. Следуя этой элементарной технике, можно четко структурировать код и обеспечить его высокое качество, не прилагая никаких сверхусилий.

# Внутренние классы

Чаще всего обработка событий происходит во внутренних классах, относящихся к классу, в котором создается основная часть пользовательского интерфейса. Причину этого понять несложно: когда происходит событие, нас интересует не только сам факт появления этого события, но и то, в каком состоянии находятся компоненты пользовательского интерфейса (каков текст в текстовых полях, сколько выбрано элементов в списке и т. п.). Конечно, более заманчиво выглядела бы обработка событий в отдельных классах (так было бы еще проще определять, где и какие события обрабатывает программа, а также обновлять эти классы и наследовать от них). Но в таком случае возникла бы сильная связь между этими классами и классами пользовательского интерфейса, а это никому не нужно, потому что требует дополнительной работы и вносит путаницу.

Пример использования внутреннего класса.

// InnerClassEvents.java

// Внутренние классы для обработки событий

**import** javax.swing.\*;

**import** java.awt.\*;

**import** java.awt.event.\*;

**public** **class** InnerClassEvents **extends** JFrame {

**private** JTextField text;

**private** JButton button;

**public** InnerClassEvents() {

**super**("InnerClassEvents");

// при закрытии окна - выход

setDefaultCloseOperation(***EXIT\_ON\_CLOSE***);

// последовательное расположение

setLayout(**new** FlowLayout());

// добавим текстовое поле

add(text = **new** JTextField(10));

// и кнопку

add(button = **new** JButton("Нажмите"));

// будем следить за нажатиями кнопки

button.addActionListener(**new** ButtonL());

// выводим окно на экран

pack();

setVisible(**true**);

}

// класс - слушатель нажатия на кнопку

**class** ButtonL **implements** ActionListener {

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

System.***out***.println(text.getText());

}

}

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

SwingUtilities.*invokeLater*(

**new** Runnable() {

**public** **void** run() {

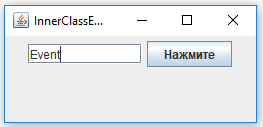
**new** InnerClassEvents();

}

});

}

}



В примере показана классическая ситуация: имеется текстовое поле и кнопка – мы добавили их в наше окно, предварительно установив для него последовательное расположение компонентов. Пользователь что-то вводит в поле и щелкает на кнопке, а программа должна обработать введенные им данные. Использование внутреннего класса для обработки события щелчка на кнопке (слушателя ActionListener) дает нам возможность без помех получить доступ к текстовому полю text и содержащийся в нем текст. Используй мы отдельный класс, нам пришлось бы каким-то образом заполучить ссылку на объект нашего окна, более того, в классе окна InnerClassEvents нам пришлось бы либо объявить текстовое поле открытым для доступа (public), либо добавить новый метод, возвращающий текст, набранный в текстовом поле.

Таким образом, внутренние классы – это практически оптимальный механизм обработки событий, позволяющий одновременно отделить место обработки события от места его возникновения и иметь полный доступ к элементам пользовательского интерфейса. Но возможна ситуация накопления огромного количества внутренних классов внутри одного из больших элементов системы, например главного окна приложения.

# Использование анонимных классов

Самый быстрый способ создавать классы – создание ***анонимных*** (anonymous classes) классов. В том месте программы, где вам понадобится какой-то класс, вы не создаете его в отдельном файле с отдельным именем, а пишете начинку этого класса (методы и т. п.) прямо на месте. В результате скорость написания программы немного возрастает, хотя страдает ее читабельность. Можно использовать анонимные классы и для создания слушателей событий. Рассмотрим пример:

// AnonymousClassEvents.java

// Анонимные классы для обработки событий

**import** javax.swing.\*;

**import** java.awt.event.\*;

**import** java.awt.\*;

**public** **class** AnonymousClassEvents **extends** JFrame {

**public** AnonymousClassEvents() {

**super**("AnonymousClassEvents");

// анонимный класс присоединяется прямо на месте

// выход из приложения при закрытии окна

addWindowListener(**new** WindowAdapter() {//анонимный класс

**public** **void** windowClosing(WindowEvent e) {

System.*exit*(0);

}

});

// добавим кнопку

JButton button = **new** JButton("Нажмите меня");

getContentPane().add(button);

// слушатель создается в методе

button.addActionListener(getButtonL());

// выводим окно на экран

pack();

setVisible(**true**);

}

// этот метод создает слушателя для кнопки

**public** ActionListener getButtonL() {

**return** **new** ActionListener() { //анонимный класс (лучше чем

//WindowAdapter

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

System.***out***.println("ActionListener");

}

};

}

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

SwingUtilities.*invokeLater*(

**new** Runnable() {

**public** **void** run() {

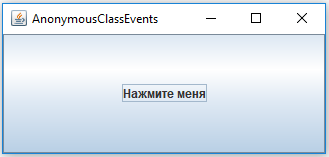
**new** AnonymousClassEvents();

}

});

}

}



ActionListener

ActionListener

В этом очень простом примере создается окно, в которое помещается кнопка. Для обработки закрытия окна мы создаем собственного слушателя оконных событий WindowEvent, и делаем это с помощью анонимного класса, который наследует от класса WindowAdapter и при закрытии окна (методом windowClosing()) завершает работу приложения. Все происходит прямо на месте: и регистрация слушателя, и создание слушателя, и его описание. Пожалуй, быстрее обработать событие невозможно. Однако видно, что получавшийся код весьма запутан и плохо управляем: нет никакой возможности получить ссылку на объект-слушатель, нельзя унаследовать от него, анонимный класс не может получить доступ к переменным внешней области видимости, которые не были объявлены неизменными (final). Есть более удобный способ работы с анонимными слушателями – их можно создавать ***в специальных методах***. В нашем примере – это метод getButtonL(), возвращающий слушателя нажатий кнопки (который просто выводит сообщение о нажатии в стандартный поток вывода). Он предоставляет чуть больше возможностей и удобства: класс находится в отдельном методе, метод легко найти, его можно переопределить.

Однако анонимные классы в любом случае ведут к запутыванию кода, создаются ли они прямо на месте или с помощью специальных методов. Их можно использовать, если вы хотите быстро набросать простую программу и забыть о ней. Но в больших и сложных программах, склонных расти и совершенствоваться и которые надо будет поддерживать, анонимные классы могут превратить вашу жизнь в ад. От «раскиданных» по разным местам фрагментов классов, что-то зачем-то делающих, чтение и отладка кода проще не станут. В таких случаях лучше предпочесть «настоящие» именованные внутренние классы или использовать специальные приемы, повышающие гибкость программы. Некоторые наиболее популярные из них мы сейчас рассмотрим.

# Диспетчеризация

Вне всяких сомнений, техника снабжения каждого события собственным слушателем, располагающимся в отдельном классе, является самой распространенной и по праву: она действительно разделяет места возникновения и обработки события и позволяет создавать кристально чистый код. Есть и другой способ обработки событий, в котором используется противоположная идея: обработка событий происходит в одном классе (или в нескольких, но не в таком умопомрачительном количестве, как в предыдущих вариантах). Техника эта называется *диспетчеризацией* (dispatching), или *перенаправлением* (forwarding), и довольно часто используется в визуальных средствах разработки интерфейса.

Суть этой техники такова: вы обрабатываете схожие события в одном слушателе, не плодя море классов, сортируете происходящие события по источникам (местам, где события происходят) и вызываете для каждого события обрабатывающий его метод (то есть слушатель не обрабатывает событие, а выступает диспетчером, отправляющим его в нужное место, отсюда и название техники). Чтобы все стало окончательно ясно, рассмотрим небольшой пример.

// ForwardingEvents.java

// Техника диспетчеризации событий

**import** javax.swing.\*;

**import** java.awt.\*;

**import** java.awt.event.\*;

**public** **class** ForwardingEvents **extends** JFrame {

**public** ForwardingEvents() {

**super**("ForwardingEvents");

// при закрытии окна - выход

setDefaultCloseOperation(***EXIT\_ON\_CLOSE***);

// последовательное расположение

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

// добавим пару кнопок

button1 = **new** JButton("ОК");

button2 = **new** JButton("Отмена");

getContentPane().add(button1);

getContentPane().add(button2);

// будем следить за нажатиями кнопок

Forwarder forwarder = **new** Forwarder();

button1.addActionListener(forwarder);

button2.addActionListener(forwarder);

// выводим окно на экран

pack();

setVisible(**true**);

}

JButton button1, button2;

// класс - слушатель нажатия на кнопку

**class** Forwarder **implements** ActionListener {

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

// рассылаем события по методам

**if** (e.getSource() == button1) onOK(e);

**if** (e.getSource() == button2) onCancel(e);

}

}

// обработка события от кнопки "ОК"

**public** **void** onOK(ActionEvent e) {

System.***out***.println("onOK()");

}

// обработка события от кнопки "Отмена"

**public** **void** onCancel(ActionEvent e) {

System.***out***.println("onCancel()");

}

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

SwingUtilities.*invokeLater*(

**new** Runnable() {

**public** **void** run() {

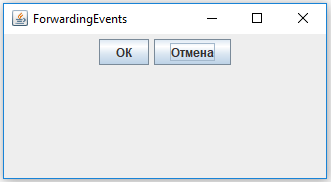
**new** ForwardingEvents();

}

});

}

}



onOK()

onCancel()

В данном примере создается окно, в которое помещено две кнопки. К каждой кнопке присоединен слушатель событий Forwarder, следящий за нажатиями кнопок, причем слушатель этот создается только один раз (что без сомнений позволяет экономить память). В самом слушателе проделывается немудреная работа: при возникновении события от кнопки выясняется, в какой именно кнопке произошло это событие, после чего вызывается метод с соответствующим названием. Слушатель Forwarder можно расширить, чтобы он поддерживал гораздо большее число кнопок, и при этом не придется создавать новые классы – достаточно будет лишь определить новые методы. Если в дальнейшем понадобится модифицировать работу приложения, это будет несложно: надо унаследовать новый класс от класса окна и переопределить интересующие нас методы, например onOK().

Диспетчеризация имеет свои преимущества: код получается более компактным. Однако есть здесь и свои ограничения: то, как события рассылаются по методам, целиком зависит от классов слушателей, подобных Forwarder, и если событие от какого-то компонента не обрабатывается этим классом, то надо писать слушатель самому. Если компонентов в интерфейсе достаточно много, и для каждого из них создается свой метод, обрабатывающий некоторое событие, получится гигантский класс, «битком набитый» методами, а работать с такими классами всегда неудобно; более того, появление таких классов свидетельствует о нарушении основополагающего правила объектно-ориентированного программирования: каждый класс решает собственную небольшую задачу.

Вывод: создавайте для слушателей событий отдельные классы.

**Замечание**. ***Проблема висячих ссылок***. Если вы пишете программу с заранее неизвестным количеством компонентов, не забывайте отсоединять от удаляемых компонентов ранее присоединенных слушателей методом removeXXXListener(), особенно если эти слушатели используются многократно и привязаны к каким-либо еще компонентам системы, которые еще активны и не стали мусором. Будет утечка памяти. Сборщик мусора не удалит объекты, у которых есть явные ссылки на слушателей. Висячие ссылки – это проблема не системы обработки событий Swing, а побочный эффект использования шаблона проектирования «наблюдатель».

# Создание собственных событий

Иногда бывают нужны компоненты, функционал которых стандартные компоненты библиотеки Swing обеспечить не могут. В таком случае придется создать собственный компонент, унаследовав его от какого-либо компонента библиотеки или написав «с нуля» (то есть унаследовав от базового компонента JComponent библиотеки Swing или, если вы хотите создать компонент «с чистого листа», от базового класса Component библиотеки AWT). У нового компонента, если он выполняет не самые простые функции, наверняка будут какие-то события, и программистам-клиентам компонента (если компонент окажется удачным, возможности для его многократного использования обязательно найдутся) необходимо предоставить способ обработки этих событий.

В качестве примера рассмотрим простую кнопку: небольшой прямоугольник с рамкой, при нажатии которого происходит некоторое событие.

Прежде всего необходимо создать **класс события**. Как вы помните из описания схемы событий JavaBeans, этот класс должен быть унаследован от класса java.util.EventObject и иметь название вида XXXEvent, где XXX – название события. Вот что получается для нашей кнопки:

// com.usm.swing.event/ButtonPressEvent.java

// класс событий для кнопки SimpleButton

**package** com.usm.swing.event;

**import** java.util.EventObject;

**public** **class** ButtonPressEvent **extends** EventObject {

// конструктор. требует задать источник события

**public** ButtonPressEvent(Object source) {

**super**(source);

}

}

В событии не будет храниться никакой дополнительной информации, поэтому класс события очень прост. Заметьте, что конструктор класса требует указать источник события; как правило, это компонент, в котором событие произошло. Источник события нужно задавать для любого события, унаследованного от класса EventObject, а получить его позволяет метод getSource() того же базового класса. Таким образом, при обработке любого события JavaBeans вы можете быть уверены в том, что источник этого события всегда известен. И, наконец, обратите внимание на то, что класс нового события разместился в пакете com.usm.swing.event. При создании других событий можно добавлять в них подходящие поля и методы, которые сделают работу с событием более комфортной.

Далее нам нужно описать интерфейс слушателя нашего события. Данный интерфейс будут реализовывать программы-клиенты компонента, заинтересованные в нажатиях кнопки. Интерфейс слушателя, следующего стандарту JavaBeans, должен быть унаследован от интерфейса java.util.EventListener. В последнем нет ни одного метода, он служит «отличительным знаком», показывая, что наш интерфейс описывает слушателя событий. Итак:

//com.usm.swing.event/ButtonPressListener.java

//Интерфейс слушателя события нажатия кнопки

**package** com.usm.swing.event;

**import** java.util.EventListener;

**public** **interface** ButtonPressListener **extends** EventListener {

// данный метод будет вызываться при нажатии кнопки

**void** buttonPressed(ButtonPressEvent e);

}

В интерфейсе слушателя определен всего один метод buttonPressed(), который и будет вызываться при нажатии создаваемой кнопки. В качестве параметра этому методу передается объект события ButtonPressEvent, так что заинтересованный в нажатии кнопки программист, реализовавший интерфейс слушателя, будет знать подробности о событии. Интерфейс слушателя, так же как и класс события, разместился в пакете com.usm.swing.event.

Теперь остается включить поддержку события в класс самого компонента. Для этого в нем нужно определить пару методов для присоединения и отсоединения слушателей ButtonPressListener, эти методы должны следовать схеме именования событий JavaBeans. В нашем случае методы будут именоваться addButtonPressListener() и removeButtonPressListener(). Слушатели, которых программисты регистрируют в данных методах, будут оповещаться о нажатии кнопки. Существует два основных способа регистрации слушателей в компоненте и оповещения их о происходящих событиях.

* Регистрация *единичного* (unicast) слушателя. В классе компонента определяется единственная ссылка на слушателя события, так что узнавать о событии может только один слушатель одновременно. При присоединении нового слушателя старый слушатель, если он был, перестает получать оповещения о событиях, так как ссылка на него теряется.
* Регистрация *произвольного* (multicast) количества слушателей. В компоненте заводится список, в котором и хранятся все присоединяемые к компоненту слушатели.
* При возникновении события все находящиеся в списке слушатели получают о нем полную информацию.
* В подавляющем большинстве ситуаций используется гораздо более гибкий и удобный второй способ с произвольным количеством слушателей, практически все события стандартных компонентов Swing (это же относится и к компонентам AWT), низкоуровневые и высокоуровневые, поддерживают произвольное количество слушателей. Для нас не составит особого труда реализовать список слушателей в нашем компоненте, так что мы тоже выбираем второй способ. Все готово для того, чтобы приступить к созданию самого компонента:

//Пример компонента со своим собственным событием

**package** com.usm.swing;

**import** javax.swing.\*;

**import** java.awt.\*;

**import** java.awt.event.\*;

**import** java.util.\*;

**import** com.usm.swing.event.\*;

**public** **class** SimpleButton **extends** JComponent {

// список слушателей

**private** ArrayList<ButtonPressListener> listenerList = **new** ArrayList<ButtonPressListener>();

// один объект-событие на все случаи жизни

**private** ButtonPressEvent event = **new** ButtonPressEvent(**this**);

// конструктор - присоединяет к кнопке слушателя

// событий от мыши

**public** SimpleButton() {

addMouseListener(**new** PressL());

// зададим размеры компонента

setPreferredSize(**new** Dimension(100, 100));

}

// присоединяет слушателя нажатия кнопки

**public** **void** addButtonPressListener(ButtonPressListener l) {

listenerList.add(l);

}

// отсоединяет слушателя нажатия кнопки

**public** **void** removeButtonPressListener(ButtonPressListener l) {

listenerList.remove(l);

}

// прорисовывает кнопку

**public** **void** paintComponent(Graphics g) {

// зальем зеленым цветом

g.setColor(Color.***green***);

g.fillRect(0, 0, getWidth(), getHeight());

// рамка

g.setColor(Color.***black***);

g.draw3DRect(0, 0, getWidth(), getHeight(), **true**);

}

// оповещает слушателей о событии

**protected** **void** fireButtonPressed() {

**for** (ButtonPressListener l : listenerList) {

l.buttonPressed(event);

}

}

// внутренний класс, следит за нажатиями мыши

**class** PressL **extends** MouseAdapter {

// нажатие мыши в области кнопки

**public** **void** mousePressed(MouseEvent e) {

// оповестим слушателей

fireButtonPressed();

}

}

}

Мы создаем очень простой компонент SimpleButton: это прямоугольник с рамкой, который тем не менее обладает собственным событием ButtonPressEvent. Компонент унаследован от базового класса JComponent библиотеки Swing, так что все действия, связанные с его прорисовкой, мы поместили в метод paintComponent(). В конструкторе происходит настройка основных механизмов компонента: присоединяем к нему слушателя событий от мыши, реализованного во внутреннем классе PressL, а также задаем размер компонента методом setPreferredSize() (по умолчанию размеры компонента считаются нулевыми). В слушателе PressL будем отслеживать нажатия кнопок мыши, для этого нам понадобится метод mousePressed(). Как только пользователь щелкнет кнопкой мыши в области, принадлежащей создаваемому компоненту, будет вызван метод fireButtonPressed(), обязанностью которого является оповещение слушателей о событии. Кстати, название вида fireXXX() (или fireXXXEvent()) является неофициальным стандартом для методов, «запускающих» высокоуровневые события, хотя такие методы и не описаны в спецификации JavaBeans.

Слушатели ButtonPressListener будут храниться в списке ArrayList, адаптированном только под хранение объектов ButtonPressListener. Благодаря мощи стандартного списка ArrayList методы для присоединения и отсоединения слушателей совсем просты: им нужно лишь использовать соответствующие возможности списка. Также просто и оповещение слушателей о событиях: для каждого элемента списка мы вызываем метод buttonPressed(), передавая ему в качестве параметра объект-событие. Объект-событие у нас один на компонент, и именно он передается всем слушателям: на самом деле, зачем создавать для каждого слушателя новое событие, если в нем не хранится ничего, кроме ссылки на источник события, то есть на сам компонент. Мы не включили в компонент поддержку многозадачности: если регистрировать слушателей будут несколько потоков одновременно, у нашего компонента могут возникнуть проблемы (для эффективной работы список ArrayList рассчитан на работу только с одним потоком в каждый момент времени). Но исправить это легко: просто объявите методы для присоединения и отсоединения слушателей как synchronized. Аналогичное действие проделайте и с методом fireButtonPressed() (он тоже работает со списком). Есть и еще один способ включить для компонента поддержку многозадачного окружения: с помощью класса java.util.Collections и статического метода synchronizedList(). Последний метод вернет вам версию списка ArrayList со встроенной поддержкой многозадачности. Впрочем, как мы узнаем из следующей главы, работать с компонентами Swing из нескольких потоков без специальных усилий нельзя, так что смысла в добавлении слушателей из нескольких потоков как правило никакого нет.

Ну а теперь давайте проверим работу нашего нового компонента и посмотрим, будет ли обрабатываться принадлежащее ему событие. Вот простой пример:

// SimpleButtonTest.java

// ќбработка событи¤ нового компонента

**import** javax.swing.\*;

**import** com.usm.swing.\*;

**import** com.usm.swing.event.\*;

**import** java.awt.\*;

**public** **class** SimpleButtonTest **extends** JFrame {

**public** SimpleButtonTest() {

**super**("SimpleButtonTest");

setDefaultCloseOperation(***EXIT\_ON\_CLOSE***);

// создаем кнопку и присоединим слушателей

SimpleButton button = **new** SimpleButton();

// анонимный класс (слушатель)

button.addButtonPressListener(

**new** ButtonPressListener() {

**public** **void** buttonPressed(ButtonPressEvent e) {

System.***out***.println("1!");

}

});

// внутренний класс (слушатель)

button.addButtonPressListener(**new** ButtonL());

// добавим кнопку в окно

JPanel contents = **new** JPanel();

contents.add(button);

setContentPane(contents);

// выведем окно на экран

setSize(400, 300);

setVisible(**true**);

}

**class** ButtonL **implements** ButtonPressListener {

//реализация слушателя во внутреннем классе

**public** **void** buttonPressed(ButtonPressEvent e) {

System.***out***.println("2!");

}

}

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

SwingUtilities.*invokeLater*(**new** Runnable() {

**public** **void** run() {

**new** SimpleButtonTest();

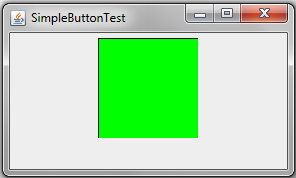
}

});

}

}

Результат работы программы:



К кнопке присоединено два слушателя ButtonPressListener.

Вывод на консоль при щелчке мышкой по зеленому квадрату (кнопка SimpleButton).

1!

2!

1!

2!

**Список EventListenerList**

В пакете javax.swing.event, который предназначен для хранения событий и слушателей компонентов библиотеки Swing, есть интересный инструмент – специализированный список EventListenerList. Этот список позволяет хранить в одном месте произвольное количество слушателей любого типа, лишь бы они были унаследованы от «отличительного знака» слушателей – интерфейса EventListener. Для того чтобы можно было легко различать слушателей разных типов, при добавлении в список каждому слушателю требуется сопоставить его «тип», который представлен объектом Class. Таким образом список EventListenerList состоит из пар вида «объект Class – слушатель EventListener». Для добавления в список новой такой пары служит метод add(). Если бы мы в только что разобранном нами примере использовали для хранения слушателей вместо ArrayList список EventListenerList, то для присоединения слушателей нам понадобилось бы написать следующий код:

listenerList.add(ButtonPressListener.class, l);

Получить слушателей определенного типа позволяет метод getListeners(), которому необходимо передать объект Class, определяющий тип слушателей. Данный метод возвращает массив слушателей нужного нам типа. Для получения слушателей ButtonPressListener пригодился бы такой код:

EventListener[] listeners = listenerList.getListeners(ButtonPressListener.class);

С полученным массивом слушателей легко работать: для каждого элемента массива (все элементы массива имеют базовый тип EventListener) вы можете смело провести преобразование к указанному вами в методе getListeners() типу слушателя (у нас это ButtonPressListener) и вызвать определенные в интерфейсе слушателя методы, которые и сообщат слушателям о том, что в компоненте произошли некоторые события. Помимо метода getListeners() можно также использовать метод getListenersList(), возвращающий все содержащиеся в списке пары Class – EventListener в виде массива. С массивом, полученным из метода getListenersList(), работать сложнее: приходится самостоятельно проверять, принадлежит ли слушатель нужному нам типу и учитывать тот факт, что каждый слушатель хранится вместе с объектом Class. Метод getListeners() без сомнения проще и удобнее.

Практически все компоненты библиотеки Swing, обладающие собственными событиями, и стандартные модели с поддержкой слушателей используют для хранения слушателей список EventListenerList. Если в вашем компоненте или модели имеется событие только одного типа (и соответственно, слушатель одного типа), проще задействовать обычный список, вроде ArrayList. Но если событий и слушателей несколько, преимущества EventListenerList выходят на первый план: достаточно единственного экземпляра такого списка для хранения всех регистрируемых в вашем компоненте или модели слушателей; не нужно понапрасну расходовать память на несколько списков и выполнять излишнюю работу. Требуется лишь правильно указывать, слушателя какого типа вы собираетесь добавить в список, удалить из него (для этого предназначен метод remove()) или получить.

# События от мыши и метод contains()

Стоит упомянуть одну особенность работы с событиями от мыши. Дело в том, что они весьма зависимы от области на экране, которую занимает интересующий вас компонент: только в том случае, когда курсор мыши находится в области компонента, мы можем сказать, что происходят перемещения курсора, щелчки или перетаскивания. Заведует же определением местонахождения той или иной точки экрана в области компонента метод contains(), определенный в базовом классе всех компонентов Java Component.

По умолчанию нам, как правило, ничего не приходится делать – область, которую выделяет компоненту контейнер (панель, окно или апплет) и является той зоной, в которой происходят все события от мыши. Это понятное и прозрачное поведение. Но бывает, что нам нужно модифицировать это поведение, особенно если компонент не прямоугольной формы или имеет какие-то «окна», сквозь которые события от мыши проникают к компонентам, лежащим ниже в контейнере. Ситуация в Swing не такая уж редкая. В качестве быстрого и понятного примера можно посмотреть на такую кнопку:

// ContainsTest.java

// Изменение поведения мыши и метод contains()

**import** javax.swing.\*;

**import** java.awt.\*;

**public** **class** ContainsTest **extends** JFrame {

**public** ContainsTest() {

**super**("ContainsTest");

// при закрытии окна - выход

setDefaultCloseOperation(***EXIT\_ON\_CLOSE***);

// добавим кнопку и переопределим метод

JButton button = **new** JButton("Невидима") {

@Override

**public** **boolean** contains(**int** x, **int** y) {

// не содержим ни одной точки

**return** **false**;

}

};

// настроим панель содержимого и выведем окно на экран

setLayout(**new** FlowLayout());

add(button);

setSize(300, 200);

setVisible(**true**);

}

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

SwingUtilities.*invokeLater*(

**new** Runnable() {

**public** **void** run() {

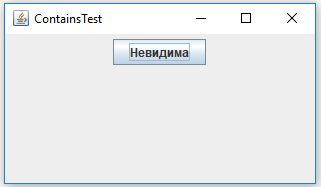
**new** ContainsTest();

}

});

}

}



Кнопка ничем не отличается от своих собратьев за исключением метода contains() – этот метод для переданных ему координат точки должен сообщить, принадлежит ему точка или нет. Мы говорим, что нам чужда любая точка из нашей области, и запустив пример, вы увидите, что происходит. Кнопка не нажимается мышью, хотя она доступна и включена, но свободно нажимается с клавиатуры (пробельной клавишей).

Следует помнить, что необходимо переопределять именно метод, которому передаются два целых числа, координаты компонента X и Y, а не второй вариант метода с объектом Point, он системой не вызывается. Ну и координаты исчисляются в системе самого компонента – (0,0) обозначает начало компонента, и максимум не будет превышать длину и ширину компонента, так работает система рассылки событий.

**Резюме**

Итак, события могут возникать в любом компоненте Swing; для того чтобы узнать о них, надо создать соответствующего слушателя событий и зарегистрировать его в компоненте. В большинстве случаев обработка событий не заходит дальше этой простой цепочки действий. Однако при работе с системой событий возникают и более сложные ситуации, как правило, это связано с появлением при обработке событий различных потоков выполнения.

# За кулисами системы обработки событий

Присоединяя слушателей к компоненту, мы можем узнавать обо всех действиях пользователей. Однако иногда возникают более сложные задачи, требующие более изощренного контроля над происходящими событиями, например, если вам понадобится отфильтровать некоторые события или перехватить их раньше, чем они дойдут до слушателей. В этой главе мы познакомимся с «начинкой» системы обработки событий Swing и посмотрим, как она функционирует.

Программы, написанные на Java, выполняются виртуальной машиной Java (Java Virtual Machine, JVM), которая обеспечивает независимость от конкретной платформы. Виртуальная машина использует ресурсы операционной системы для решения своих задач, это утверждение верно и для событий пользовательского интерфейса. Низкоуровневые события (движения мыши, нажатия клавиш, закрытие окон) возникают именно в операционной системе, виртуальная машина перехватывает их и перенаправляет компонентам пользовательского интерфейса Java-приложения. Посмотрим теперь, как низкоуровневые события попадают к компонентам Java (к которым относятся и компоненты Swing).

# Поток EventDispatchThread и очередь событий EventQueue

Вспомним, как запускается Java-программа: вызывается статический метод main(). При этом виртуальная машина создает новый поток выполнения (который так и называется «main») и передает ему управление. Самое интересное происходит, когда в методе main() создается объект, каким-либо образом связанный с графической системой Java, то есть унаследованный от базового класса java.awt.Component. При этом происходит целая череда событий.

* Инициализируется динамическая библиотека Java, связывающая графическую подсистему Java с операционной системой, когда в этом возникает необходимость, как правило, с первыми же операциями внутри библиотеки AWT.
* Создаются необходимые помощники компонентов AWT, отвечающие за их связь с операционной системой.
* Компоненты выводятся на экран, и операционная система начинает посылать им события (своим «родным» тяжеловесным компонентам). Эти события преобразуются в объекты Java, которые помещаются в очередь событий EventQueue.
* Очередь событий проверяет наличие потока рассылки событий, и, так как приложение только начинает работу, впервые запускает поток выполнения EventDispatchThread, который связывается с очередью событий EventQueue и начинает рассылать события, хранящиеся в этой очереди событий, по соответствующим компонентам AWT.

Очередь событий – это экземпляр класса EventQueue из пакета java.awt, и на самом деле класс этот очень прост. Он представляет собой контейнер, работающий по принципу FIFO. В этой очереди графическая система Java хранит все приходящие в программу низкоуровневые события. Происходит это следующим образом: пользователь совершает действие, операционная система сообщает об этом, виртуальная машина Java создает соответствующий событию объект (например, объект класса MouseEvent, если пользователь щелкнул мышью) и добавляет его в конец очереди EventQueue. Класс EventQueue – это *одиночка* (singleton), сколько бы обычных окон, диалоговых окон или компонентов приложение не создавало, очередь событий всегда присутствует в единственном экземпляре. Единственное, что можно сделать, – это заменить существующую очередь событий своей собственной.

Далее вступает в действие специальный поток выполнения EventDispatchThread. Он циклически «подкачивает» (pump) события из очереди событий EventQueue (это делает метод pumpEvents()), и, если события в очереди имеются, он извлекает их и рассылает по соответствующим компонентам AWT (это выполняет метод dispatchEvent() класса EventQueue). Так как поток EventDispatchThread один, очередь событий одна, и события хранятся в порядке их поступления, обеспечивается последовательная обработка событий в соответствии с очередностью их поступления, то есть события обрабатываются одно за другим.

Роль потока EventDispatchThread в любой графической программе на Java трудно переоценить, это настоящий «властелин» этих программ. Именно в этом потоке происходит вся работа, и он никогда не останавливается. Мы отмечали, что графическая программа в промежутке между событиями простаивает, это так лишь с нашей стороны, с другой стороны всегда работает поток EventDispatchThread, вызывающий наш код только при наступлении событий (в графической программе наш код весьма пассивен). Стоит ему остановиться, как программа тут же прекратит свою работу. В этом предложении ответ на часто возникающий вопрос, почему программа, использующая графический интерфейс, не заканчивается после выполнения метода main(), как это делают обычные (консольные) программы.

В спецификации виртуальной машины Java сказано, что программа завершает свою работу, когда в ней не останется работающих потоков выполнения (не являющихся демонами). В графической программе такой поток выполнения остается – это как раз поток распределения событий EventDispatchThread

Низкоуровневые события, возникающие в операционной системе, рассылаются только тяжеловесным компонентам AWT, потому что легковесных компонентов (к которым относятся и компоненты Swing) операционная система не видит. Легковесные компоненты узнают о событиях, только не напрямую, а через один скрытый механизм.

# Доставка событий методам processXXXEvent()

Когда поток выполнения EventDispatchThread обнаруживает, что в очереди событий появилось новое событие, он извлекает его и проверяет тип этого события, чтобы определить, что делать с ним дальше. Если извлеченное из очереди событие относится к графическому компоненту (это может быть событие, относящееся к прорисовке, или более интересное нам событие, возникшее в результате действия пользователя), то поток EventDispatchThread вызывает метод dispatchEvent() очереди событий EventQueue. Этот метод не делает ничего экстраординарного, а просто узнает, к какому компоненту относится событие (то есть определяет источник события – об этом сообщает операционная система) и вызывает метод dispatchEvent() этого компонента.

Все компоненты AWT наследуют метод dispatchEvent() от своего базового класса java.awt.Component, но ни один из них не в состоянии вмешаться в процесс первоначальной обработки событий, потому что метод этот является неизменным (final) и переопределить его невозможно, т.к. в методе dispatchEvent() происходит самая важная внутренняя работа по распределению событий, в том числе координация действий с помощниками компонентов, преобразование событий PaintEvent в вызовы методов paint() и update(), а также целая череда действий, вмешиваться в которые нет смысла, потому что они относятся к внутренней реализации библиотеки и постоянно меняются без всякого предупреждения. В конце концов, метод dispatchEvent() передает событие (если это было событие в ответ на действие пользователя) методу processEvent().

Метод processEvent() – это следующий этап на пути следования событий к слушателям. Ничего особенного в нем не происходит, его основная функция – передать событие согласно его типу одному из методов processXXXEvent(), где «XXX» – название события, например Focus или Key. Именно методы processXXXEvent() распределяют приходящие события по слушателям. Например, метод processKeyEvent() определяет, что именно за событие от клавиатуры пришло, и вызывает соответствующий метод слушателя KeyListener (если конечно, такие слушатели были зарегистрированы в компоненте). После этого мы наконец-то получаем уведомление о событии в свою программу (посредством слушателя) и выполняем необходимые действия. Так заканчивается путь события, начинавшийся в очереди событий. В целом все выглядит так, как показано на рис. 3.1.

Поток рассылки событий EventDispatchThread

Очередь событий EventQueue

Действия пользователя

|  |  |  |  |
| --- | --- | --- | --- |
| Слушатели | | | |
|  | | |  |
|  | | processXXXEvent() | |  |
|  | | |  |
|  | | processEvent() | |  |
|  | | |  |
|  | Component.dispatchEvent() | | |  |

Рис. 3.1. Путь события к слушателям

Вся описанная цепочка действий относится к тяжеловесным компонентам AWT, которые «видны» операционной системе и которым она посылает низкоуровневые события. Как же легковесные компоненты получают уведомления о событиях? Оказывается, что происходит это с помощью тяжеловесного контейнера (обычно окна), в котором расположены легковесные компоненты. Он вмешивается в цепочку обработки событий еще на уровне метода dispatchEvent() и, прежде чем продолжить обработку события обычным образом, просматривает все содержащиеся в нем легковесные компоненты, проверяя, не принадлежит ли пришедшее событие им. Если это так, то обработка события тяжеловесным контейнером прекращается, а событие передается в метод dispatchEvent() соответствующего легковесного компонента. Эти действия встроены в базовый класс контейнеров java.awt.Container, и любой унаследованный от него контейнер может быть уверен, что содержащиеся в нем легковесные компоненты получат необходимую поддержку. Для программистов-клиентов Swing все эти хитрости остаются незаметными, и цепочка обработки событий кажется одинаковой как для тяжеловесного, так и для легковесного компонента.

Мы упомянули здесь методы processXXXEvent() потому, что они могут быть полезными и в обычных программах. Чаще всего эти методы используют там, где необходимо выполнить некоторое действие перед поступлением события к слушателю или вообще запретить распространение события. Для этого необходимо переопределить метод processXXXEvent(), соответствующий нужному нам событию. Рассмотрим простой пример, в котором мы попытаемся вмешаться в цепочку обработки событий от мыши:

// PreProcessMouse.java

// Перехват событий от мыши до их поступления к слушателям

**import** javax.swing.\*;

**import** java.awt.event.\*;

**public** **class** PreProcessMouse **extends** JFrame {

PreProcessMouse() {

**super**("PreProcessMouse");

// при закрытии окна - выход

setDefaultCloseOperation(***EXIT\_ON\_CLOSE***);

// добавим слушателя событий от мыши

addMouseListener(**new** MouseL());

// выводим окно на экран

setSize(200, 200);

setVisible(**true**);

}

// перехват событий от мыши

**public** **void** processMouseEvent(MouseEvent e) {

**if** (e.getClickCount() == 1) {

// один щелчок не пропускаем к слушателям

**return**;

}

// иначе вызываем метод базового класса

**else**

**super**.processMouseEvent(e);

}

// в этом слушателе будем следить за щелчками мыши

**class** MouseL **extends** MouseAdapter {

@Override

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

System.***out***.println("ClickCount: " + e.getClickCount());

}

}

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

SwingUtilities.*invokeLater*(**new** Runnable() {

**public** **void** run() {

**new** PreProcessMouse();

}

});

}

}

На консоль выводятся сообщения “ClickCount:” только, если было произведено больше 2-х щелчков мышки.

В примере мы создаем небольшое окно JFrame, при его закрытии приложение будет заканчивать свою работу. К этому окну (которое является обыкновенным компонентом, унаследованным от базового класса Component), мы присоединяем слушателя событий от мыши MouseL, который унаследован от адаптера MouseAdapter и следит за щелчками мыши в окне (о них сообщается в метод слушателя mouseClicked()). Однако слушатель этот – не единственное заинтересованное в щелчках мыши «лицо»: мы переопределили метод processMouseEvent() нашего окна JFrame. Как нам теперь известно, именно в этом методе происходит рассылка событий от мыши слушателям, в нем мы также отслеживаем щелчки мыши еще до поступления события к слушателям. Посмотрите, что происходит: в методе processMouseEvent() мы выясняем, сколько раз пользователь щелкнул мышью (не важно, правая эта кнопка или левая), и если количество щелчков равно единице, заканчиваем работу метода, что равносильно игнорированию всех присоединенных слушателей (рассылка событий происходит в методе базового класса, и если мы его не вызываем, то слушатели ничего не узнают). В противном случае, если количество щелчков больше единицы, мы вызываем метод базового класса, в обязанности которого входит рассылка события слушателям.

Запустив программу с примером, вы (глядя на консольный вывод) сможете увидеть, какие щелчки мыши добираются до слушателя событий MouseL. В методы processXXXEvent() попадают всевозможные события одного типа, для мыши это щелчки, перетаскивание, нажатия, вход в область компонента и выход из нее, и многое другое.

# Маскирование и поглощение событий

Исследование показывает, что в компоненты приходят уведомления абсолютно обо всех происходящих с ними низкоуровневых событиях, будь то события от мыши, клавиатуры, изменения в размерах контейнера или что-либо еще, причем приходят такие уведомления всегда, независимо от того, заинтересован компонент в этих событиях или нет, и в легковесные, и в тяжеловесные компоненты. В итоге получалось, что и для сложных компонентов, таких как таблицы, по-настоящему нуждающихся в информации о большей части событий, и для простых вспомогательных компонентов, подобных панелям, которые по большому счету нуждаются только в своевременной перерисовке, система присылала полную информацию обо всех происходящих событиях. И это не приводило к блестящей производительности.

Решением стало *маскирование событий* (event masking). С каждым компонентом графической системы, способным получать уведомления о событиях, теперь ассоциирована специальная маска в виде длинного (long) целого, которая и определяет, какие события компонент получает. По умолчанию отключено получение всех событий, кроме уведомлений о необходимости перерисовки. Но, как только вы (или другой компонент) присоединяете к компоненту слушателя события определенного типа, например, слушателя событий от мыши MouseListener, компонент тут же включает в свою маску признак обработки данного события (от мыши), так что система обработки событий начинает присылать ему все связанные с этой маской события. Принцип прост: нет слушателей – нет и событий, как только слушатели появляются – автоматически появляются события.

# Работа с очередью событий

Очередь EventQueue является настоящим ядром системы обработки событий: все события, так или иначе возникающие в вашей программе, проходят через нее, включая даже такие экзотические события, как уведомления о перерисовке PaintEvent. Можно использовать в программе свой вариант очереди событий, получая над системой обработки событий практически неограниченную власть.

Получить используемую в данный момент очередь событий позволяет метод getSystemEventQueue() класса Toolkit, а получить объект Toolkit можно методом getToolkit(), который имеется в каждом унаследованном от класса Component компоненте (кстати, класс Toolkit – это *абстрактная фабрика*, используемая для создания основных частей AWT). Полученный экземпляр очереди событий позволяет проделать многое: вы сможете узнать, какие события находятся в данный момент в очереди событий, вытащить их оттуда, поместить в очередь новые события (которые могут быть созданы вами вручную, а всем компонентам будет «казаться», что события эти возникли в результате действий пользователя). Помещение в очередь сгенерированных вами событий – прекрасный способ автоматизированного тестирования вашего интерфейса или демонстрации некоторых его возможностей. Приложение при этом будет вести себя точно так же, как если бы с ним работал самый настоящий пользователь. Рассмотрим небольшой пример:

// UsingEventQueue.java

// использование очереди событий

**import** java.awt.\*;

**import** java.awt.event.\*;

**import** javax.swing.\*;

**public** **class** UsingEventQueue **extends** JFrame {

**public** UsingEventQueue() {

**super**("UsingEventQueue");

// выход при закрытии окна

setDefaultCloseOperation(***EXIT\_ON\_CLOSE***);

// кнопка и ее слушатель

JButton button = **new** JButton("Генерировать событие");

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

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

// генерируем событие закрыти¤ окна

getToolkit().getSystemEventQueue()

.postEvent(**new** WindowEvent(UsingEventQueue.**this**, WindowEvent.***WINDOW\_CLOSING***));

}

});

// добавим кнопку в панель содержимого

setLayout(**new** FlowLayout());

add(button);

// выведем окно на экран

setSize(400, 300);

setVisible(**true**);

}

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

SwingUtilities.*invokeLater*(**new** Runnable() {

**public** **void** run() {

**new** UsingEventQueue();

}

});

}

}

В примере мы создаем небольшое окно, при закрытии которого программа будет завершать свою работу. В панель содержимого окна (для нее мы устанавливаем последовательное расположение компонентов FlowLayout) добавляется кнопка JButton с присоединенным к ней слушателем. При нажатии кнопки мы создаем событие WindowEvent типа WINDOW\_CLOSING, именно такое событие генерируется виртуальной машиной, когда пользователь пытается закрыть окно. Созданное событие (помимо типа ему нужно указать источник, то есть окно, которое закрывается пользователем) мы помещаем в очередь событий, используя для этого метод postEvent(). Запустив программу с примером, вы увидите, что при нажатии кнопки приложение заканчивает свою работу, точно так же, как если бы мы нажали кнопку закрытия окна. События, которые виртуальная машина генерирует в ответ на действия пользователя, вы можете с тем же успехом генерировать самостоятельно – результат будет точно таким же.

Заменить стандартную очередь событий собственным вариантом очереди позволяет метод push() класса EventQueue. Ему нужно передать экземпляр вашей очереди событий, унаследованной от класса EventQueue. Наибольший интерес при наследовании представляет метод dispatchEvent(), который, как мы знаем, вызывается при распределении событий всех типов. Переопределив данный метод, вы получите исчерпывающую информацию о том, как, когда и какие события распределяются в вашей программе. Это может быть весьма ценно при отладке и диагностике вашей программы или при реализации особого поведения ваших компонентов.

Есть чуть менее сложный способ узнать обо всех распределяемых в вашей программе низкоуровневых событиях – использовать особый слушатель AWTEventListener. Вы присоединяете его к системе обработки событий с помощью класса Toolkit, указывая при этом, о событиях каких типов вы хотите знать. После присоединения слушателя AWTEventListener события всех указанных вами типов будут перед распределением попадать этому слушателю.

# Влияние на программы потока EventDispatchThread

Рассмотрим поток распределения событий EventDispatchThread и роль, которую он играет в любой графической программе. Известно, что этот поток распределяет по компонентам события, находящиеся в очереди событий EventQueue, и запускается той самой очередью, когда в нее попадают первые события графической системы. Таким образом, если программа использует графические компоненты, она автоматически оказывается в многозадачном окружении, и придется принимать во внимание вопросы совместного использования ресурсов и их синхронизации.

В программе кроме потока рассылки событий всегда есть еще один поток выполнения с названием main. В нем выполняется метод main(), и он вполне может конфликтовать с потоком рассылки событий. Кроме того, важнейшим свойством любого пользовательского интерфейса является его *отзывчивость (Responsive Swing Application)*. Если не использовать для выполнения длинных сложных задач отдельные потоки (так чтобы не блокировать рассылку событий и работу программы), обеспечить отзывчивость интерфейса будет невозможно.

Прежде всего выясним, когда именно запускается поток рассылки событий. Если брать в расчет библиотеку AWT, основу Swing, то там считается, что поток рассылки событий запускается, как только какой-либо компонент переходит в *реализованное* (realized) состояние. Реализованное состояние компонента означает, что он становится видимым для операционной системы, начинает получать от нее сообщения о событиях и занимает некоторое место на экране или в памяти экрана (даже если он все еще невидим). При получении событий очередь событий запускает поток рассылки. Легковесные компоненты (к которым относятся и компоненты Swing), а их операционная система не видит, переходят в реализованное состояние одновременно с переходом в реализованное состояние тяжеловесных контейнеров (окон или апплетов), в которых они содержатся. В свою очередь, тяжеловесные контейнеры переходят в реализованное состояние после вызова одного из следующих методов:

* pack() – служит для придания окну оптимального размера;
* setVisible(true) или show() – выводит окно на экран.

Если вы вызвали один из этих методов, можете быть уверены в том, что поток рассылки событий уже начал свою работу, и вы находитесь в многозадачном окружении. До вызова одного из этих методов графические компоненты AWT являются просто объектами, и вы можете работать с ними из любого потока.

Вопросы синхронизации с потоком рассылки событий становятся особенно важны при работе с компонентами библиотеки Swing, потому что последние практически полностью лишены каких бы то ни было средств для работы в многозадачном окружении. Это сделано не случайно: при разработке библиотеки рассматривалось множество альтернатив, и было выяснено, что наделение компонентов механизмами поддержки многозадачного окружения приведет не только к их усложнению, но и к значительному падению скорости их работы. Поэтому с компонентами Swing работать имеет смысл *только* из потока рассылки событий, то есть только из слушателей или методов, служащих для обработки событий. Но тут возникает резонный вопрос: «А как же обеспечить отзывчивость интерфейса? Ведь длительные вычисления в потоке рассылки событий блокируют остальной интерфейс программы.»

Есть несколько методов, которые можно вызывать из любого потока выполнения, не опасаясь соревнования с потоком рассылки событий. Они обладают встроенной синхронизацией и всю работу с компонентами все равно вызовут из потока рассылки событий. Вот эти методы:

* repaint() – служит для перерисовки компонента;
* revalidate(), validate(), invalidate() – позволяют заново расположить компоненты в контейнере и удостовериться в правильности их размеров.

Таким образом, вы (в отдельном потоке выполнения) можете провести какие-либо сложные вычисления, изменить некоторые данные, а затем попросить компонент обновить себя, так чтобы он смог вывести на экран новую информацию. И это не приведет к конфликтам.

Ну и, наконец, есть самый гибкий способ изменить компонент из другого потока. Если ваша задача, выполняющаяся в отдельном потоке, так или иначе приводит к необходимости изменить что-либо в компоненте, и в этом вам не могут помочь безопасные вызовы, остается только одно. Во избежание неприятностей с потоком рассылки событий и неожиданного тупика все действия с компонентами *все равно* нужно выполнять из потока рассылки событий. И у вас есть возможность выполнить некоторое действие в потоке рассылки событий. Для этого предназначены методы invokeLater() и invokeAndWait() класса EventQueue. Данным методам нужно передать ссылку на интерфейс Runnable, метод run() этого интерфейса будет выполнен потоком рассылки событий, как только он доберется до него. (Переданная в метод invokeLater() или invokeAndWait() ссылка на интерфейс Runnable «оборачивается» в событие специального типа InvocationEvent, обработка которого сводится к вызову метода run().) В итоге вы действуете следующим образом: выполняете в отдельном потоке сложные долгие вычисления, получаете некоторые результаты, а действия, которые необходимо провести после этого с графическими компонентами, выполняете в потоке рассылки событий с помощью метода invokeLater() или invokeAndWait(). Рассмотрим небольшой пример:

// InvokeLater.java

// Метод invokeLater() и работа с потоком рассылки событий

**import** java.awt.\*;

**import** java.awt.event.\*;

**import** javax.swing.\*;

**public** **class** InvokeLater **extends** JFrame {

**public** InvokeLater() {

**super**("InvokeLater");

// при закрытии окна - выход

setDefaultCloseOperation(***EXIT\_ON\_CLOSE***);

// добавим кнопку со слушателем

button = **new** JButton("Выполнить сложную работу");

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

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

// запустим отдельный поток

**new** ComplexJobThread().start();

button.setText("Подождите...");

}

});

// настроим панель содержимого и выведем окно на экран

setLayout(**new** FlowLayout());

add(**new** JTextField(20));

add(button);

setSize(300, 200);

setVisible(**true**);

}

**private** JButton button;

// поток, выполняющий "сложную работу"

**class** ComplexJobThread **extends** Thread {

**public** **void** run() {

**try** {

// изобразим задержку

*sleep*(3000);

// работа закончена, нужно изменить интерфейс

EventQueue.*invokeLater*(**new** Runnable() {

**public** **void** run() {

button.setText("Работа завершена");

}

});

} **catch** (Exception ex) {

ex.printStackTrace();

}

}

}

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

SwingUtilities.*invokeLater*(**new** Runnable() {

**public** **void** run() {

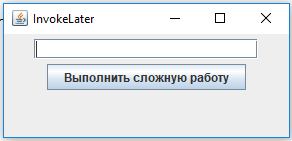
**new** InvokeLater();

}

});

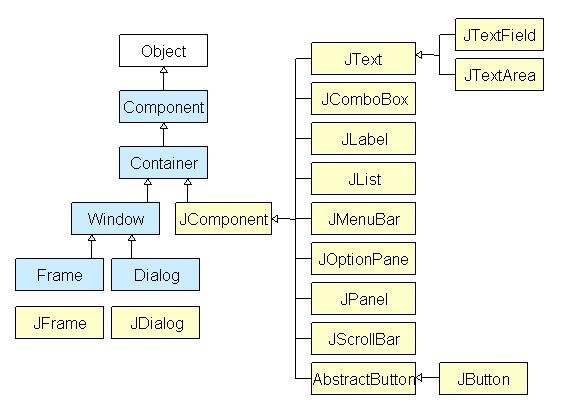
}

}



# Создание графического интерфейса пользователя с помощью библиотек AWT и Swing

# Иерархия схема классов Java Swing (схема)

Рис.7.1. Иерархия классов Java Swing

Классы контейнеров – это классы, на которых могут быть другие компоненты. Поэтому для создания графического интерфейса нам нужен хотя бы один объект-контейнер. Существует 3 типа контейнеров.

* **JPanel (Панель)**: это чистый контейнер и не является окном. Единственная цель – организовать компоненты для окна.
* **JFrame (Рамка)**: это полностью функционирующее окно со своим названием и значками.
* **JDialog (Диалог)**: его можно рассматривать как окно, которое позволяет организовать диалог с пользователем, например сохранить/загрузить файл.

<https://www.javatpoint.com/java-jcomponent>

<https://docs.oracle.com/javase/7/docs/api/>

# Основные классы библиотек AWT и Swing

Основное понятие, заложенное в графические библиотеки – *компонент графической системы*. Компонент – отдельный, полностью определенный элемент, который можно использовать в GUI независимо от других элементов. Например, это поле ввода, кнопка, строка меню, полоса прокрутки, «радиопереключатель» и т.п. Само окно приложения – тоже его компонент.

В Java компонентом считается объект класса Component или одного из классов, расширяющих Component. В классе Component определены общие методы работы с любым компонентом GUI. Этот класс – вершина иерархии графических библиотек, поэтому его необходимо рассмотреть подробнее.

# Класс Component

Класс Component является абстрактным и не может использоваться сам по себе, применяются только унаследованные от него подклассы. В этом классе определены методы, отвечающие за управление событиями, позиционирование и изменение размеров окна, перерисовку окна и т.п.

Компонент всегда занимает прямоугольную область окна со сторонами, параллельными сторонам экрана. В компоненте есть система координат. Ее начало – точка с координатами (0,0) – находится в левом верхнем углу компонента. Оси направлены так как показано на рис.9.2.

Ox

Oy

* *Рис.9.2*

Высоту и ширину компонента можно узнать с помощью метода

Dimension getSize()

Этот метод возвращает объект класса Dimension в полях width и height, которого хранятся ширина и высота компонента. Ту же задачу можно решить с помощью методов

int getWidth()

int getHeight()

класса Component. Разница заключается в том, что в объекте класса Dimension переменные width и height хранятся в виде чисел типа double, а методы класса Component возвращают числа типа int.

Чтобы установить положение и размеры компонента можно воспользоваться методом

setBounds(int x, int y, int width, int height)

Доступность компонента и его видимость на экране определяется с помощью методов

isEnabled()

isVisible()

Управлять этими параметрами можно с помощью методов

setEnabled(boolean b)

setVisible(boolean b)

# Класс Container

Каждый компонент перед выводом на экран должен быть помещен в *контейнер* – специальный объект, отвечающий за размещение компонентов на экране. Контейнер может содержать набор компонентов, которые добавляются к нему с помощью метода

Component add(Component comp)

и другие контейнеры. В языке Java контейнер – объект класса Container или всякого его расширения. Класс Container сам является невидимым компонентом, он расширяет класс Component. Все класс окон унаследованы от класса Container.

# Классы окон

# Класс JFrame

Контейнер JFrame это готовое окно со строкой заголовка, в которую помещены кнопки контекстного меню, сворачивания/разворачивания окна, закрытия приложения. Окно типа Frame имеет рамку, которая позволяет перемещать его и изменять его размеры с помощью мыши.

Конструкторы класса JFrame:

JFrame()

JFrame(String title)

Первый конструктор создает окно с пустой строкой заголовка. Второй – окно со строкой заголовка, задаваемой параметром title. Заголовок окна можно создать и позже – методом

setTitle(String title)

Созданное окно типа Frame, вначале является невидимым. Чтобы сделать его видимым, следует вызвать метод

setVisible(Boolean state)

класса Component, рассмотренный выше.

Окно типа JFrame после создания автоматически регистрируется в оконном менеджере графической оболочки.

# Класс JWindow

Контейнер JWindow – это пустое окно без рамки, строки заголовка, строки меню, полос прокрутки, т.е. это просто прямоугольная область на экране. Оно не регистрируется в оконном менеджере графической оболочки, следовательно его нельзя перемещать и менять размеры средствами операционной системы. Окно типа Window может быть создано только уже существующим окном-владельцем. Когда окно-владелец убирается с экрана, вместе с ним убирается и порожденное окно.

Конструкторы класса Window:

JWindow(JFrame f)

JWindow(JWindow w)

Первый конструктор – создает окно, владелец которого – фрейм f. Второй – окно, владелец которого – уже имеющееся окно или подкласс класса Window.

Уничтожить окно, освободив занимаемы им ресурсы, можно методом:

void dispose()

Окно типа Window можно использовать для создания всплывающих окон предупреждения., сообщений, подсказок.

# Класс JDialog

Класс JDialog – это окно фиксированного размера, предназначенное для ответа на сообщения приложения. Оно автоматически регистрируется в оконном менеджере графической оболочки, следовательно, его можно перемещать по экрану, менять размеры. Окно типа JDialog, как и его суперкласс – окно типа Window – обязательно должно иметь владельца, который указывается в конструкторе. Окно типа JDialog может быть модальным либо немодальным.

В классе JDialog есть семь конструкторов:

JDialog(JDialog owner)

Создает немодальное диалоговое окно с владельцем owner и пустой строкой заголовка

JDialog(JDialog owner, String title)

Создает немодальное диалоговое окно с владельцем owner и строкой заголовка title

JDialog(JDialog owner, String title, boolean modal)

Создает диалоговое окно с владельцем owner и строкой заголовка title. Окно будет модальным, если параметр modal == true.

Следующие 4 конструктора аналогичны, но создают диалоговые окна, принадлежащие окну типа Frame.

JDialog(JFrame owner)

JDialog(JFrame owner, boolean modal)

JDialog(JFrame owner, String title)

JDialog(JFrame owner, String title, boolean modal)

# Класс JFileChooser

Класс JFileChooser – это диалоговое окно, содержащее стандартное окно *File Load…/File Save As…*. Класс JFileChooser находится в пакете javax.swing.filechooser.

Основные конструкторы класса JFileChooser:

JFileChooser()

Создает окно со списком файлов из текущего каталога

JFileChooser(String path)

Создает окно со списком файлов из каталога, указанного в path

Методы File getSelectedFile() и String getName(File f) возвращают соответственно выбранный файл и его имя в виде строки. Можно установить начальный каталог для поиска файла методом void setCurrentDirectory(File f).

Пример:

1. JFileChooser chooser = new JFileChooser();
2. int returnVal = chooser.showOpenDialog(null);
3. if(returnVal == JFileChooser.APPROVE\_OPTION)
4. {
5. System.out.println("You chose to open this file: " + chooser.getSelectedFile().getName());
6. }

Ограничить множество отображаемых файлов можно создав специальный класс, унаследованный от абстрактного класса FileFilter, и реализовав в этом классе метод accept().

# Графический контекст, рисование

При создании компонента автоматически формируется его *графический контекст* (graphics context). Контекст содержит текущий и альтернативный цвет рисования и цвет фона, текущий шрифт для вывода текста, систему координат (см.рис.9.2). Управляет контекстом класс Graphics. Т.к. графический контекст сильно зависит от конкретной графической платформы, этот класс является абстрактным. Каждя JVM реализует методы этих классов, создает их экземпляры для компонента. Получить ссылку на созданный графический контекст можно с помощью метода

Graphics getGraphics()

После получения cсылки на графический контекст, можно пользоваться методами класса Graphics для работы с графикой и текстом. Основные методы класса Graphics представлены в таблице 9.1.

* *Таблица 9.1*

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| **drawLine(int x1, int y1, int x2, int y2)** | Вычерчивает текущим цветом отрезок прямой между точками (x1,y1) и (x2,y2) |
| **drawRect(int x, int y, int width, int height)** | Чертит прямоугольник со сторонами, параллельными краям экрана, задаваемый координатами верхнего угла (x,y), шириной width и высотой height |
| **draw3DRect(int x, int y, int width, int height, boolean raised)** | Чертит прямоугольник, «выделяющийся» из плоскости рисования, если raised = true и «вдавленный» в плоскость, если raised = false |
| **drawOval(int x, int y, int width, int height)** | Чертит овал, вписанный в прямоугольник, заданный аргументами метода (если width = height – окружность) |
| **drawArc(int x, int y, int width, int height, int startAngle, int arc)** | Чертит дугу овала, вписанного в прямоугольник, заданный первыми четырьмя аргументами. Дуга имеет величину arc градусов и отсчитывается от угла startAngle. Угол отсчитывается в градусах от оси Ox. Положительный угол отсчитывается против часовой стрелки, отрицательный – по часовой стрелке. |
| **drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)** | Чертит прямоугольник с закругленными краями. Закругления вычерчиваются четвертинками овалов, вписанных в прямоугольники шириной arcWidth и высотой arcHeight, построенные в углах основного прямоугольника. |
| **drawPolyline(int xPoints[], int yPoints[], int nPoints)** | Чертит ломаную линию с вершинами в точках (xPoints[i], yPoints[i]) и числом вершин nPoints |
| **drawPolygon(int xPoints[], int yPoints[], int nPoints)** | Чертит замкнутую ломаную с вершинами в точках (xPoints[i], yPoints[i]) и числом вершин nPoints |
| **drawPolygon(Polygon p)** | Чертит замкнутую ломаную, вершины которой заданы объектом p класса Polygon (этот класс подробнее рассматривается ниже) |
| **fillRect(int x, int y, int width, int height)** | Эти методы аналогичны соответствующим методам draw…(), но в отличие от них вычерчивают фигуры, залитые текущим цветом |
| **fill3DRect(int x, int y, int width, int height, boolean raised)** |
| **fillOval(int x, int y, int width, int height)** |
| **fillArc(int x, int y, int width, int height, int startAngle, int arc)** |
| **fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)** |
| **fillPolyline(int xPoints[], int yPoints[], int nPoints)** |
| **fillPolygon(int xPoints[], int yPoints[], int nPoints)** |
| **fillPolygon(Polygon p)** |
| **drawString(String s, int x,**  **int y)** | Выводит строку s. Вывод начинается  с позиции (x,y). |
| **drawBytes(byte b[],int offset, int length, int x, int y)** | Выводит length элементов массива b, начиная с индекса offset. Вывод начинается  с позиции (x,y). |
| **drawChars(char ch[], int offset, int length, int x, int y)** | Выводит length элементов массива ch, начиная с индекса offset. Вывод начинается  с позиции (x,y). |

# Класс Color, установка цвета

Цвет в Java – объект класса Color. В классе Color 7 конструкторов:

Color(int r, int g, int b)

Создается цвет, получающийся смешением красной (r), зеленой (g) и синей (b) составляющих (модель RGB). Значение интенсивности каждой составляющей меняется от 0 до 255.

Color(int rgb)

Все три составляющие цвета задаются одним числом. В битах 16-23 – красная составляющая, в битах 8-15 – зеленая, в битах 0-7 – синяя.

Color(float r, float g, float b)

Аналогичен первому, но составляющие модели RGB задаются вещественными числами от 0.0 до 1.0, что позволяет изменять их более плавно.

Color(int r, int g, int b, int a)

Color(float r, float g, float b, float a)

В этих конструкторах вводится четвертая составляющая (a), определяющая прозрачность цвета. Она может меняться от 255 (цвет совершенно не прозрачен – предыдущий цвет не виден) до 0(цвет абсолютно прозрачен – для каждого пиксела виден только предыдущий цвет).

Color(int rgba, boolean hasAlpha)

Данный конструктор учитывает прозрачность цвета исходя из значения параметра hasAlpha. Если hasAlpha = true, в качестве значения прозрачности используются биты 24-31 параметра rgba, если hasAlpha = false, прозрачность считается равной 255.

Color(ColorSpace cspace, float[] components, float alpha)

Данный конструктор позволяет создавать цвет в цветовых моделях, отличных от **RGB**. Например, CMYK, **HSB** и т.п. Тип цветовой модели задается параметром ColorSpace.

Если нет необходимости тщательно подбирать цвета, можно воспользоваться одной из статических констант типа Color, имеющихся в классе Color. Таких констант 13: black, blue, cyan, darkGray, gray, green, lightGray, magenta, orange, pink, red, white, yellow.

**Установка цвета**

Чтобы установить цвет фона для окна, используйте **setBackground()**.

Чтобы установить цвет переднего плана (цвет вывода текста, например), используйте **setForeground()**.Эти методы определены в Component и имеют следующую общую форму:

**void** **setBackground**(Color newColor)

**void** **setForeground**(Color newColor)

Здесь **newColor** специфицирует новый цвет. В следующем примере устанавливается зеленый цвет фона и красный цвет текста (используются предопределенные цвета):

**setBackground**(Color.green);

**setForeground**(Color.red);

Получить текущие установки цветов переднего плана и текста можно вызовами методов **getBackground()** и **getForeground()** соответственно. Они также определены в Component и показаны ниже:

Color **getBackground**()

Color **getForeground**()

# Класс Font, установка шрифтов

**Создание шрифта:**

Font StrongFont =

**new** Font("Helvetica", Font.BOLD | Font.ITALIC, 24);

getHeight

getMaxAscent

getMaxDescent

getSize

getStyle

Установка шрифта:

Font c = new Font("Courier", Font.PLAIN, 18);

g.setFont(c);

JLabel label = **new** JLabel("First Name");

label.setFont(**new** Font("Courier New", Font.ITALIC, 12));

label.setForeground(Color.GRAY);

Получение имеющихся в системе шрифтов:

String FontList[];

FontList = getToolkit().getFontList();

**for** (**int** i=0; i<FontList.lenght; i++) {

S.o.p(i + 'i' + FontList[i]);

}

# Класс Polygon

Этот класс предназначен для работы с многоугольниками. Объекты данного класса можно создать двумя конструкторами:

Polygon()

Создается пустой объект.

Polygon(int xPoints[], int yPoints[], int nPoints)

Создается многоугольник с вершинами (xPoints[i], yPoints[i]). Число вершин – nPoints.

Данный класс содержит методы, позволяющие легко проверить, лежит ли в многоугольнике заданная точка, отрезок или целый прямоугольник. Основные методы класса Polygon собраны в таблице 9.2.

* *Таблица 9.2*

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| boolean contains(double x, double y) | Проверяет, лежит ли точка (x,y) внутри многоугольника |
| boolean contains(double x, double y, double w, double h) | Проверяет, лежит ли прямоугольник, заданный аргументами метода внутри многоугольника |
| boolean contains(int x, int y) | Проверяет, лежит ли точка (x,y) внутри многоугольника |
| boolean contains(Point p) | Проверяет, лежит ли точка (x,y) внутри многоугольника |
| boolean intersects(double x, double y, double w, double h) | Проверяет, пересекается ли прямоугольник, заданный аргументами метода с многоугольником |

В следующем примере строится квадрат. Выбирается случайная точка, выводятся ее координаты и проверяется ее принадлежность квадрату.

Пример 9.2:

import java.awt.\*;

import java.awt.event.\*;

import javax.swing.\*;

import java.util.Random;

class Rect

{public static void main(String args[])

{int xPoints[] = new int[4];

int yPoints[] = new int[4];

int x,y;

JFrame f = new JFrame();

f.setBounds(10,10,500,200);

f.setVisible(true);

Graphics g = f.getGraphics();

g.setColor(new Color (0,0,255));

Random r = new Random();

xPoints[0]=20;yPoints[0]=70;

xPoints[1]=280;yPoints[1]=70;

xPoints[2]=280;yPoints[2]=150;

xPoints[3]=20;yPoints[3]=150;

x = r.nextInt (500);

y = r.nextInt (200);

Polygon p = new Polygon(xPoints,yPoints,4);

g.drawPolygon (p);

g.setColor(new Color (255,0,0));

if (p.contains (x,y)==true)

g.drawString ("Точка " + x + " " + y + " попадает

в прямоугольник", 20,50);

else

g.drawString ("Точка " + x + " " + y + " не попадает

в прямоугольник", 20,50);

f.addWindowListener (new WindowAdapter()

{public void windowClosing(WindowEvent ev)

{System.exit (0);

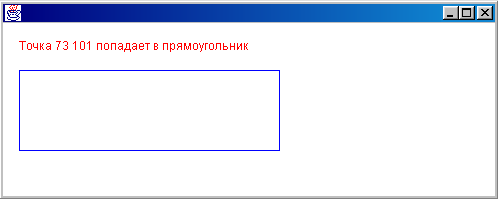
}

});

}

}

Вывод этой программы:



# Графические изображения

Image image = getImage(Url, "MorningGlory.jpg");

g.drawImage(image, 0, 0, this);

showDocument(URL);//????

showDocument(URL, target);

repaint() **–** используется для принудительного перерисовывания апплета.

repaint(time)**–** устанавливает крайний срок для перерисовки.

repaint(x, y, w, h) **–** ограничивается областью - прямоугольником.

repaint(time, x, y, w, h) **–** совмещение двух предыдущих.

В библиотеке Swing графические изображения инкапсулированы классом ImageIcon, реализующим интерфейс Icon.

Конструктор класса ImageIcon:

ImageIcon(String filename)

Основные методы класса ImageIcon:

int getIconHeight()

int getIconWidth()

void paintIcon(Component comp, Graphics g, int x, int y)

# Элементы управления

Библиотеки AWT и Swing поддерживает следующие основные элементы управления:

* Текстовые метки
* Кнопки
* Флажки
* «Радиопереключатели»
* Списки
* Поля со списком
* Текстовые поля
* Текстовые области
* Панели прокрутки
* Панели с вкладками
* Деревья
* Таблицы

Все элементы управления реализованы как объекты специальных классов, являющихся подклассами класса Component. Как уже говорилось ранее, каждый компонент (в т.ч. элемент управления) должен быть включен в некоторый контейнер (подкласс класса Container). В библиотеке AWT таким контейнером является само окно. Для добавления элемента управления к окну необходимо использовать одну из форм метода add( ), определенного в классе Container. Самая простая форма выглядит следующим образом:

Component add(Component obj)

Здесь obj – экземпляр элемента управления, который добавляется к окну. Метод возвращает ссылку на объект, который передается параметром obj. Сразу после добавления элемент будет выводиться на экран автоматически, когда отображается его «родительское» окно.

Удаление элемента управления из окна производится методом

void remove (Component obj).

Вызывая метод removeAll() можно удалить сразу все присоединенные к окну элементы управления.

Управлять положением элемента управления относительно окна, в котором он размещен, можно с помощью методов setBounds( ) и getBounds( ). Формат этих методов:

void setBounds(int x, int y, int width, int height)

void setBounds(Rectangle r)

Rectangle getBounds()

В компоненте хранятся координаты его левого верхнего угла в системе координат «родительского» окна. Узнать их можно с помощью метода:

Point getLocation()

а изменить с помощью методов:

void setLocation(int x, int y)

void setLocation(Point p)

Все элементы управления, за исключением текстовых меток, могут взаимодействовать с пользователем, генерируя некоторые события, которые в программе можно идентифицировать и обрабатывать. Модель обработки событий, реализованная в языке Java, будет рассматриваться ниже.

В библиотеке Swing компоненты перед выводом на экран должны быть помещены не в окно, а внутрь специального контейнера, который называется *панель содержания* (content pane). Этот контейнер не нужно создавать, его можно получить для окна подобно графическому контексту:

JFrame f = new JFrame("Test");

Container cp = f.getContentPane();

После этого все операции добавления/удаления компонентов выполняются для панели содержания, а не для самого окна.

# Текстовые метки

Самый простой для использования компонент графического интерфейса – текстовая метка (label). Текстовые метки – это объекты класса JLabel, содержащие одну строку статического текста и/или графическое изображение. Среди всех элементов графического интерфейса только метки являются пассивными, т.е. не поддерживают никакого взаимодействия с пользователем во время выполнения.

Конструкторы класса JLabel:

JLabel (String str)

JLabel (Icon id\_icon)

JLabel (String str, Icon id\_icon, int align)

создает метку, содержащую строку str и/или графическое изображение id\_icon. Выравнивание строки определяется параметром align. Значением align должна быть одна из трех констант, определенных в классе JLabel:

JLabel.LEFT - выравнивание по левому краю

JLabel.RIGHT - выравнивание по правому краю

JLabel.CENTER - выравнивание по центру

Текст в метке можно устанавливать или изменять с помощью метода

void setText(String str)

Текущую метку можно получить, используя метод

String getText( )

Управлять выравниванием текста можно с помощью методов

void setAlignment(int align)

int getAlignment( )

Графическое изображение в метке можно устанавливать или изменять методами:

void setIcon(Icon id\_icon)

Текущее графическое изображение можно получить, используя метод

Icon getIcon( )

# Управляющие кнопки

Управляющая кнопка – это объект класса JButton. Кнопка может содержать текстовую строку (надпись на кнопке) и/или графическое изображение. Она генерирует события в ответ на щелчок мышью.

Основные конструкторы класса JButton:

JButton(String str)

JButton(Icon id\_icon)

Методы, позволяющие управлять видом кнопки:

String getLabel( )

void setLabel(String str)

Icon getLabel( )

void setIcon(Icon id\_icon)

# Флажки – переключатели

Флажки-переключатели являются объектами класса JCheckBox. Самая общая форма конструктора:

JCheckBox (String str, Icon id\_icon, boolean state)

Управление состоянием флажка из программы осуществляется с помощью методов:

boolean getSelected()

void setSelected(boolean state)

# Группы кнопок («радиопереключатели»)

«Радиопереключатели» в библиотеке Swing реализованы как объекты класса JRadioButton. Самая общая форма конструктора:

JRadioButton (String str, Icon id\_icon, boolean state)

«Радиопереключатели» должны быть объединены в группу, в пределах которой будет обеспечиваться уникальность выбора элемента. Чтобы создать группу создается экземпляр объекта специального класса ButtonGroup. Далее все созданные радиопереключатели добавляются в созданную группу (что не отменяет необходимости их добавления к панели содержания).

Пример:

1. JFrame f = new JFrame();
2. Container cp = f.getContentPane();
3. JRadioButton jrb1 = new JRadioButton("a",false);
4. JRadioButton jrb2 = new JRadioButton("b",false);
5. JRadioButton jrb3 = new JRadioButton("c",true);
6. ButtonGroup bg = new ButtonGroup();
7. bg.add(jrb1);
8. bg.add(jrb2);
9. bg.add(jrb3);
10. cp.add(jrb1);
11. cp.add(jrb2);
12. cp.add(jrb3);

# Раскрывающиеся списки

Раскрывающийся список – компонент класса JComboBox. Создать раскрывающийся список можно с помощью одного из конструкторов:

JComboBox()

JComboBox(Vector v[])

JComboBox(Object obj[])

Далее в список можно добавлять новые пункты с помощью методов:

addItem(Object item)

Новые строки будут добавляться к списку в том порядке в котором выполнялись вызовы метода addItem( ).

Вставить строку в список в определенную позицию (index) можно методом:

insertItemAt(Object anObject, int index)

Удаление строк из списка:

void removeItem(String text)

void removeItemAt(int index)

void RemoveAllItems()

Для определения текущего выбранного элемента можно использовать методы

String getSelectedItem( )

int getSelectedIndex( )

Первый метод возвращает строку с именем элемента, а второй – номер выбранной строки. Строки в списке нумеруются от 0. По умолчанию выбранным считается нулевая строка.

Чтобы получить текущее количество элементов в списке используется метод:

int getItemCount( )

Зная номер строки в списке можно получить ее имя с помощью метода:

String getItemAt(int index)

Можно произвести выбор пункта в списке из программы с помощью методов:

void setSelectedIndex (int index)

void setSelectedItem (Object obj)

По умолчанию раскрывающийся список является не редактируемым. Чтобы сделать его редактируемым, необходимо использовать метод:

void setEditable (boolean state)

# Списки

Списки – объекты класса JList. В них допускается множественный выбор.

Стандартный объект класса JList – неизменяемый список, поэтому его необходимо заполнять данными в момент создания.

Конструкторы:

JList(Vector v[])

JList(Object obj[])

Кроме того объекты типа JList не поддерживают автоматическую прокрутку. Поэтому для того, чтобы элементы в списке могли прокручиваться, список необходимо поместить в специальный объект JScrollPane, а уже этот объект добавить в контейнер.

Пример:

JFrame f = new JFrame();

Container cp = f.getContentPane();

f.setBounds(100,100,300,300);

String s[] = {"1","2","3","4","5","6","7","8","9","10"};

JList jl = new JList(s);

jl.setBounds(10,10,30,30);

JScrollPane p = new JScrollPane(jl);

cp.add(p);

f.setVisible(true);

Для определения выбранного элемента (элементов) используются методы:

Object getSelectedValue()

Object[] getSelectedValues()

int getSelectedIndex( )

int[] getSelectedIndices( )

# Текстовые поля

Однострочная область ввода текста (текстовое поле) – объект класса JTextField. Текстовые поля дают возможность вводить строки, редактировать их с помощью клавиш-стрелок, Backspace, Delete, пользоваться буфером обмена.

Конструкторы:

JTextFiled( )

JTextFiled(int numChars)

JTextFiled(String str)

JTextFiled(String str, int numChars)

Первая форма создает пустое текстовое поле. Вторая текстовое поле шириной numChars символов. Третья – инициализирует текстовое поле строкой str. Четвертая – инициализирует текстовое поле и задает его ширину.

Основные методы класса JTextField:

String getText()

Возвращает строку, содержащуюся в поле ввода.

void setText(String str)

Устанавливает в поле ввода текст str.

String getSelectedText()

Пользователь может выделять часть текста в текстовом поле с помощью клавиатуры или мыши. Метод getSelectedText() возвращает выделенную часть текста.

void select(int start, int end)

Этот метод позволяет программно выделить часть текста в поле ввода начиная с позиции start до позиции end.

Следующие два метода управляют доступностью текстового поля для редактирования.

boolean isEditable( )

void setEditable(boolean canEdit)

В обоих методах true означает доступность текста для изменения.

Если необходимо, чтобы пользователь мог вводить текст без его отображения в текстовом поле (например, пароли), то нужно воспользоваться методом

void setEchoChar(char ch)

ch – символ, который будет отображаться вместо фактически вводимого.

# Текстовые области

Текстовая область (многострочное поле ввода) – объект класса JTextArea. Нажатие клавиши Enter переводит курсор в начало следующей строки. В текстовой области могут быть установлены линейки прокрутки.

Конструкторы:

JTextArea( )

JTextArea(int numLines, int numChars)

JTextArea(String str)

JTextArea(String str, int numLines, int numChars)

numLines определяет высоту текстовой области в строках, numChars определяет ширину текстовой области в символах, str – начальный текст.

Объект JTextArea поддерживает все методы, описанные выше для класса JTextField. В дополнение к ним поддерживаются методы:

void append (String str)

Строка str добавляется к концу текущего текста

void insert (String str, int index)

Строка str вставляется в позицию, указанную в параметре index.

void replaceRange(String str, int start, int end)

Заменяет символы от start до end-1 строкой str.

Для обеспечения горизонтальной и вертикальной прокрутки компоненты JTextField и JTextArea должны быть помещены в контейнер JScrollPane.

# Менеджеры компоновки (размещения) – Layout Manager

<http://www.quizful.net/post/swing-layout-managers>

Менеджер расположения (layout manager) определяет, каким образом на форме будут располагаться компоненты. Независимо от платформы, виртуальной машины, разрешения и размеров экрана менеджер расположения гарантирует, что компоненты будут иметь предпочтительный или близкий к нему размер и располагаться в том порядке, который был указан программистом при создании программы.

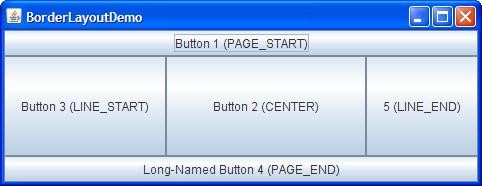
<http://java-online.ru/swing-layout.xhtml#gridlayout>

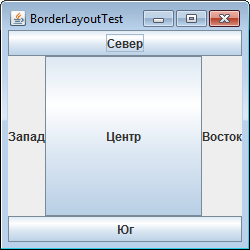
На странице представлено описание следующих менеджеров расположения и примеров:

* Полярное расположение [BorderLayout](http://java-online.ru/swing-layout.xhtml#borderlayout)
* Последовательное расположение [FlowLayout](http://java-online.ru/swing-layout.xhtml#flowlayout)
* Табличное расположение [GridLayout](http://java-online.ru/swing-layout.xhtml#gridlayout)
* Менеджер расположения [GridBagLayout](http://java-online.ru/swing-layout.xhtml#gridbaglayout)
* Менеджер расположения [CardLayout](http://java-online.ru/swing-layout.xhtml#cardlayout)
* Менеджер расположения [BoxLayout](http://java-online.ru/swing-layout.xhtml#boxlayout)
* Менеджер расположения [GroupLayout](http://java-online.ru/swing-layout.xhtml#grouplayout)
* Менеджер расположения [SpringLayout](http://java-online.ru/swing-layout.xhtml#springlayout)
* Пример [диалогового окна авторизации](http://java-online.ru/swing-layout.xhtml#logindialog)
* Пример собственного [менеджера расположения](http://java-online.ru/swing-layout.xhtml#verticallayout)

**Layout Manager –** используется для компоновки (или размещения) компонентов GUI внутри контейнера. Существует несколько менеджеров компоновки.

**BorderLayout –** размещает компоненты в пяти областях: сверху, снизу, слева, справа и в центре. Это менеджер компоновки по умолчанию для каждого JFrame.





**FlowLayout –** Менеджер последовательного расположения, менеджер компоновки по умолчанию для JPanel. Он просто располагает компоненты один за другим – слева направо, сверху вниз.

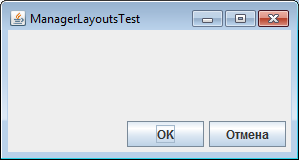


**GridLayout** – менеджер табличного расположения, представляет контейнер в виде таблицы с ячейками одинакового размера. Все ячейки таблицы имеют одинаковый размер, равный размеру самого большого компонента, находящегося в таблице.

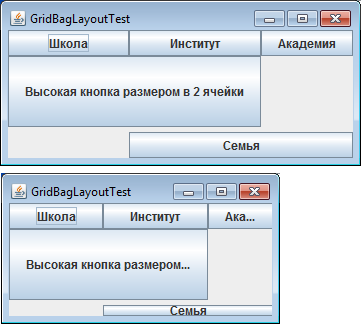




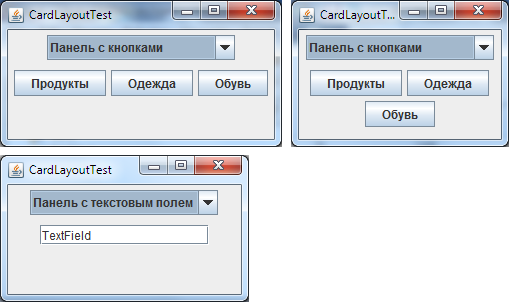
Можно совмещать менеджеры размещения, например, табличный GridLayout и последовательный FlowLayout.

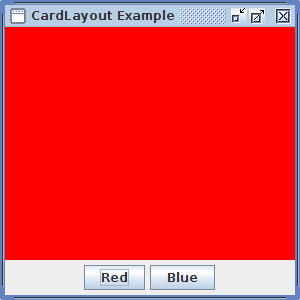


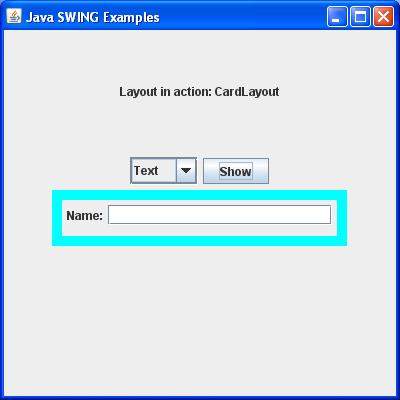
**GridBagLayout** – подобно табличному менеджеру устанавливает компоненты в таблицу, но дает возможность определять для компонентов разную ширину и высоту колонок и строк таблицы. Фактически GridBagLayout позволяет получить абсолютно любое расположение компонентов. При использовании менеджера расположения GridBagLayout необходимо настраивать класс GridBagConstraints для каждого добавляемого компонента.



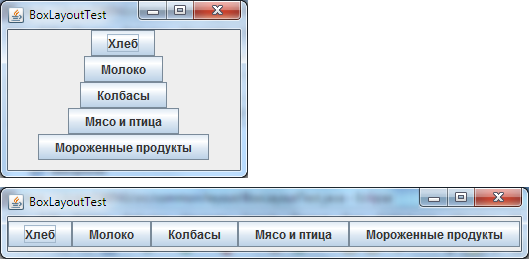
**CardLayout** можно использовать для создания так называемых вкладок (tabs), выбирая которые будут избранно открываться разные панели, занимающие одно и то же место в интерфейсе окна. В библиотеке Swing данную задачу можно решить с использованием класса JTabbedPane, организовывающего управление панелями с вкладками.



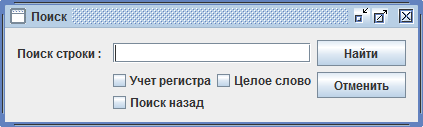




**BoxLayout** – позволяет управлять размещением компонентов либо в вертикальном, либо в горизонтальном направлениях и управлять пространством между компонентами, используя вставки. Для размещения компонентов в вертикальной плоскости необходимо конструктору передать константу BoxLayout.Y\_AXIS, для размещения в горизонтальной - BoxLayout.X\_AXIS.



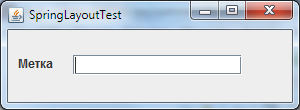
**GroupLayout** раскладывает компоненты по группам. Группы имеют горизонтальное и вертикальное направление и могут быть параллельными и последовательными. В последовательной группе у каждого следующего компонента координата вдоль оси на единицу больше (имеется в виду координата в сетке), в параллельной – компоненты имеют одну и ту же координату.

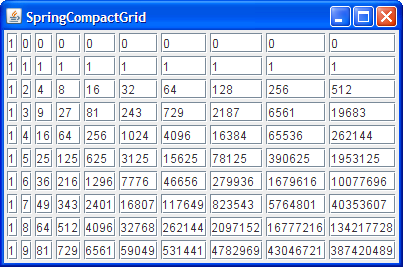


**SpringLayout**, действие его весьма специфично и не похоже на действие ни одного из уже рассмотренных выше менеджеров расположения.

С каждым компонентом ассоциируется особый информационный объект SpringLayout, который позволяет задать расстояние (в пикселах) между парой границ различных компонентов. Границ у компонента четыре — это его северная, восточная, западная и южная стороны. Можно задавать расстояние и между границами одного и того же компонента: к примеру, задав расстояние между северной и южной сторонами одного компонента, вы укажете его высоту. По умолчанию все компоненты имеют предпочтительный размер, однако SpringLayout учитывает и два остальных размера, не делая компоненты меньше минимального и больше максимального размеров.

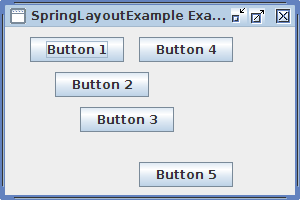
На первом этапе работы менеджера SpringLayout все компоненты находятся в начале координат контейнера и имеют предпочтительный размер. Чтобы разместить их по нужным позициям, обычно проводят следующие действия: первый компонент отделяют некоторым расстоянием от границы контейнера, второй отделяют от первого расстоянием между нужными границами, далее размещают третий компонент и т.д.





<https://docs.oracle.com/javase/tutorial/uiswing/layout/spring.html>

<http://www.java2s.com/Tutorial/Java/0240__Swing/SpringLayout.htm>



# Примеры Swing приложений

1. **Пример создания окна с меню**.

//Usually you will require both swing and awt packages

// even if you are working with just swings.

**import** javax.swing.\*;

**import** java.awt.\*;

**class** gui {

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

//Creating the Frame

JFrame frame = **new** JFrame("Frame MENU");

frame.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

frame.setSize(400, 400);

//Creating the MenuBar and adding components

JMenuBar mb = **new** JMenuBar();

JMenu m1 = **new** JMenu("FILE");

JMenu m2 = **new** JMenu("Help");

mb.add(m1);

mb.add(m2);

JMenuItem m11 = **new** JMenuItem("Open");

JMenuItem m22 = **new** JMenuItem("Save as");

m1.add(m11);

m1.add(m22);

//Creating the panel at bottom and adding components

JPanel panel = **new** JPanel(); // the panel is not visible in output

JLabel label = **new** JLabel("Enter Text");

JTextField tf = **new** JTextField(10); // accepts upto 10 characters

JButton send = **new** JButton("Send");

JButton reset = **new** JButton("Reset");

panel.add(label); // Components Added using Flow Layout

panel.add(tf);

panel.add(send);

panel.add(reset);

// Text Area at the Center

JTextArea ta = **new** JTextArea();

//Adding Components to the frame.

frame.getContentPane().add(BorderLayout.***SOUTH***, panel);

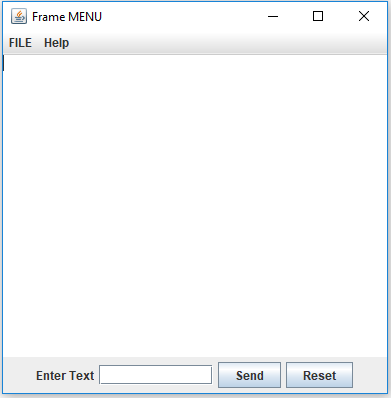
frame.getContentPane().add(BorderLayout.***NORTH***, mb);

frame.getContentPane().add(BorderLayout.***CENTER***, ta);

frame.setVisible(**true**);

}

}



1. **Label слушает Button и изменяет свой текст**

Приложение, в котором класс ChangingLabel определяет специальный тип метки, который является исполнителем действий (т. е. может принимать ActionEvents), реализуя интерфейс Он также имеет возможность изменять свой текст посредством метода changeLabel(). Метод actionPerformed() просто вызывает changeLabel(), когда получает ActionEvent.

В методе main ChangeLabel регистрируется, с помощью метода addActionListener() кнопки, как слушатель ActionListener кнопки. Мы можем это сделать, потому что JButtons создает ActionEvents, а ChangeLabel – ActionListener (то есть реализует интерфейс ActionListener).

В приведенном примере определены два класса, один из которых содержит программный код в методе static main(), а другой ChangeLabel – тип, используемый в программе (ChangeLabel). Включение метода main() программы в настраиваемый подкласс компонента является распространенной идиомой в программировании приложений Swing. При реорганизации таким образом Программа выглядит следующим образом:

**import** javax.swing.JButton;

**import** javax.swing.JFrame;

**import** javax.swing.JLabel;

**import** javax.swing.SwingConstants;

**import** java.awt.BorderLayout;

**import** java.awt.event.ActionEvent;

**import** java.awt.event.ActionListener;

// ActionEvent and ActionListener

// live in the java.awt.event package

// Application with a changing text field

**public** **class** ChangingLabel **extends** JLabel **implements** **ActionListener** {

// String constants which define the text that's displayed.

**private** **static** **final** String ***TEXT1*** = "Hello, World!";

**private** **static** **final** String ***TEXT2*** = "Hello again!";

// Instance field to keep track of the text currently

// displayed

**private** **boolean** usingText1;

// Constructor. Sets alignment and initial text, and

// sets the usingText1 field appropriately.

**public** ChangingLabel() {

**this**.setHorizontalAlignment(SwingConstants.***CENTER***);

**this**.setText(***TEXT1***);

usingText1 = **true**;

}

// A method to change the label text. If the

// current text is TEXT1, changes it to TEXT2,

// and vice-versa. The method is private here,

// because it is never called directly.

**private** **void** changeLabel() {

**if** (usingText1) {

**this**.setText(***TEXT2***);

usingText1 = **false**;

} **else** {

**this**.setText(***TEXT1***);

usingText1 = **true**;

}

}

// This method implements the ActionListener interface.

// Any time that an action is performed by a component

// that this object is registered with, this method

// is called. We can analyze the event object received

// here for more information if we want to, but it's not

// necessary in this application.

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

**this**.changeLabel();

}

// Main method. This is the code that is executed when the

// program is run

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

// Constructs the ChangingLabel.

ChangingLabel changingLabel = **new** ChangingLabel();

// Create the button

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

// Register the ChangingLabel as an action

// listener to the button. Whenever the

// button is pressed, its ActionEvent will

// be sent the ChangingLabel's

// actionPerformed() method, and the code

// there will be executed.

button.addActionListener(changingLabel);

// Create the frame

JFrame frame = **new** JFrame("Hello");

// Add the label and the button to the

// frame, using layout constants.

frame.add(changingLabel, BorderLayout.***CENTER***);

frame.add(button, BorderLayout.***SOUTH***);

frame.setSize(300, 300);

frame.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

frame.setVisible(**true**);

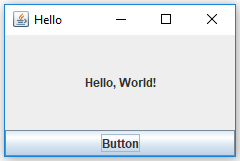
frame.setLocationRelativeTo(**null**);

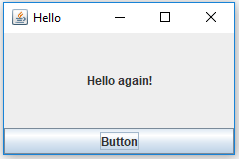
frame.toFront();

}

}

Результат работы программы:





# Механизм управления событиями в java.

Zdfsdfdfdsfdsfdsfd hfgjhdfjg fhgfjhgjf fjhgfjhgfj fghfjhg ffghfjhgjsfashjfl dlghfdskjh dflghfklhg ghrdfhglsghdflsgh

Многопоточность и асинхронность

# Библиография и ссылки

1. Oracle Java 3D API <https://www.oracle.com/technetwork/java/javase/tech/index-jsp-138252.html>
2. The Java™ Tutorials. How to Use HTML in Swing Components. <https://docs.oracle.com/javase/tutorial/uiswing/components/html.html>
3. JDK Release Notes. <https://www.oracle.com/technetwork/java/javase/jdk-relnotes-index-2162236.html>

# Приложения. Anexe

# Приложение Часы – Clock

**import** java.util.Calendar;

**import** java.text.SimpleDateFormat;

**import** java.awt.Font;

**import** javax.swing.SwingConstants;

**import** javax.swing.JFrame;

**import** javax.swing.JLabel;

**public** **class** Clock **extends** JFrame

{

**private** JLabel m\_time;

**private** Thread m\_thr;

**public** Clock()

{

m\_time = **new** JLabel();

m\_time.setHorizontalAlignment(SwingConstants.***CENTER***);

m\_time.setFont(**new** Font("Default", Font.***BOLD***+Font.***ITALIC***, 24));

getContentPane().add(m\_time);

setBounds(0,0,300,100);

m\_thr = **new** Thread() {

**public** **void** run()

{

**while**(**true**) {

setTime();

**try** {

Thread.*sleep*(500);

}

**catch**(Exception e) {}

}

}

};

m\_thr.start();

}

**public** **void** setTime()

{

SimpleDateFormat df = **new** SimpleDateFormat("dd.MM.yyyy HH:mm:ss");

m\_time.setText(df.format(Calendar.*getInstance*().getTime()));

}

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

{

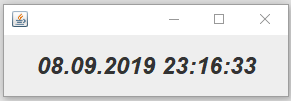
Clock cl = **new** Clock();

cl.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

cl.setVisible(**true**);

}

}



# Вывод картинки в окне - Frame

**import** java.awt.BorderLayout;

**import** java.awt.Graphics;

**import** java.awt.Image;

**import** java.awt.image.BufferedImage;

**import** java.io.File;

**import** java.io.IOException;

**import** javax.imageio.ImageIO;

**import** javax.swing.ImageIcon;

**import** javax.swing.JFrame;

**import** javax.swing.JLabel;

**import** javax.swing.JPanel;

**public** **class** TestImage **extends** JFrame {

**public** **static** BufferedImage *myImage*;

**public** **static** Image *visibleImage*;

**public** JFrame jf;

**public** **static** JPanel *jp* = **new** JPanel();

TestImage(){

JFrame jf = **new** JFrame();

jf.setSize(500,500);

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

jf.add(*jp*);

jf.setVisible(**true**);

repaint();

}

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

*visibleImage* = (Image) ImageIO.*read*(**new** File("src/duke-icon.png"));

TestImage t = **new** TestImage();

// setContentPane(new JLabel(new ImageIcon("resources/taverna.jpg")));

// BufferedImage myPicture = ImageIO.read(new File("path-to-file"));

JLabel picLabel = **new** JLabel(**new** ImageIcon(*visibleImage*));

*jp*.add(picLabel);

}

// @Override

// public void paint(Graphics g) {

// super.paint(g);

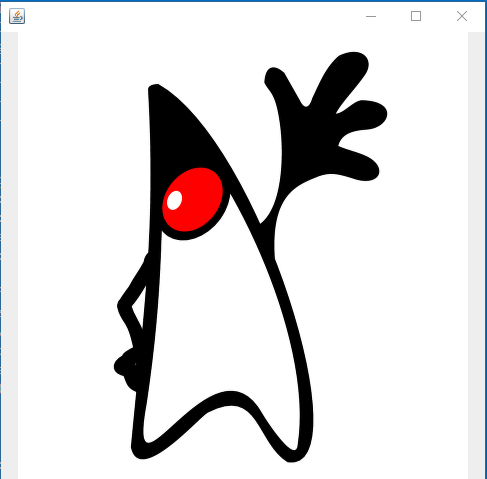
// g.drawImage(visibleImage, 0, 0, jf);

// //g.drawImage(img, 0, 0, width, height, null);

// }

}





# Использование таблиц swing - JTable

<https://docs.oracle.com/javase/tutorial/uiswing/components/table.html>

/\*

\* Copyright (c) 1995, 2008, Oracle and/or its affiliates. All rights reserved.

\*

\* Redistribution and use in source and binary forms, with or without

\* modification, are permitted provided that the following conditions

\* are met:

\*

\* - Redistributions of source code must retain the above copyright

\* notice, this list of conditions and the following disclaimer.

\*

\* - Redistributions in binary form must reproduce the above copyright

\* notice, this list of conditions and the following disclaimer in the

\* documentation and/or other materials provided with the distribution.

\*

\* - Neither the name of Oracle or the names of its

\* contributors may be used to endorse or promote products derived

\* from this software without specific prior written permission.

\*

\* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS

\* IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,

\* THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR

\* PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR

\* CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,

\* EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,

\* PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR

\* PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF

\* LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING

\* NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS

\* SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

\*/

/\*

\* SimpleTableDemo.java requires no other files.

\*/

//Использование таблиц swing - JTable

//https://docs.oracle.com/javase/tutorial/uiswing/components/table.html

**import** javax.swing.JFrame;

**import** javax.swing.JPanel;

**import** javax.swing.JScrollPane;

**import** javax.swing.JTable;

**import** java.awt.Dimension;

**import** java.awt.GridLayout;

**import** java.awt.event.MouseAdapter;

**import** java.awt.event.MouseEvent;

**public** **class** SimpleTableDemo **extends** JPanel {

**private** **boolean** DEBUG = **true**;

**public** SimpleTableDemo() {

**super**(**new** GridLayout(1,0));

String[] columnNames = {"First Name",

"Last Name",

"Sport",

"# of Years",

"Vegetarian"};

**final** JTable table = **new** JTable(data, columnNames);

table.setPreferredScrollableViewportSize(**new** Dimension(500, 70));

table.setFillsViewportHeight(**true**);

**if** (DEBUG) {//false mojno vivoditi i na console pri neobhodimosti

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

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

printDebugData(table);

}

});

}

//Create the scroll pane and add the table to it.

JScrollPane scrollPane = **new** JScrollPane(table);

//Add the scroll pane to this panel.

add(scrollPane);

}

Object[][] data = {

{"Kathy", "Smith",

"Snowboarding", **new** ~~Integer~~(5), **new** ~~Boolean~~(**false**)},

{"John", "Doe",

"Rowing", **new** ~~Integer~~(3), **new** ~~Boolean~~(**true**)},

{"Sue", "Black",

"Knitting", **new** ~~Integer~~(2), **new** ~~Boolean~~(**false**)},

{"Jane", "White",

"Speed reading", **new** ~~Integer~~(20), **new** ~~Boolean~~(**true**)},

{"Joe", "Brown",

"Pool", **new** ~~Integer~~(10), **new** ~~Boolean~~(**false**)}

};

//otladocinaia peciati DEBUG==true

**private** **void** printDebugData(JTable table) {

**int** numRows = table.getRowCount();

**int** numCols = table.getColumnCount();

javax.swing.table.TableModel model = table.getModel();

System.***out***.println("Value of data: ");

**for** (**int** i=0; i < numRows; i++) {

System.***out***.print(" row " + i + ":");

**for** (**int** j=0; j < numCols; j++) {

System.***out***.print(" " + model.getValueAt(i, j));

}

System.***out***.println();

}

System.***out***.println("--------------------------");

}

/\*\*

\* Create the GUI and show it. For thread safety,

\* this method should be invoked from the

\* event-dispatching thread.

\*/

**private** **static** **void** createAndShowGUI() {

//Create and set up the window.

JFrame frame = **new** JFrame("SimpleTableDemo");

frame.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

//Create and set up the content pane.

SimpleTableDemo newContentPane = **new** SimpleTableDemo();

newContentPane.setOpaque(**false**); //content panes must be opaque

frame.setContentPane(newContentPane);

//Display the window.

frame.pack();

frame.setVisible(**true**);

}

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

//Schedule a job for the event-dispatching thread:

//creating and showing this application's GUI.

javax.swing.SwingUtilities.*invokeLater*(**new** Runnable() {

**public** **void** run() {

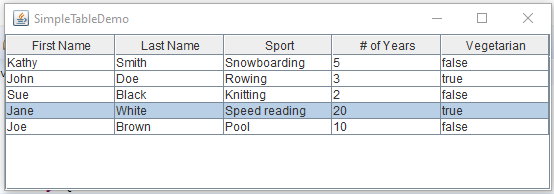
*createAndShowGUI*();

}

});

}

}



# Использование таблиц Swing - JTable (Render-Визуализатор)

<https://docs.oracle.com/javase/tutorial/uiswing/components/table.html>

/\*

\* Copyright (c) 1995, 2008, Oracle and/or its affiliates. All rights reserved.

\*

\* Redistribution and use in source and binary forms, with or without

\* modification, are permitted provided that the following conditions

\* are met:

\*

\* - Redistributions of source code must retain the above copyright

\* notice, this list of conditions and the following disclaimer.

\*

\* - Redistributions in binary form must reproduce the above copyright

\* notice, this list of conditions and the following disclaimer in the

\* documentation and/or other materials provided with the distribution.

\*

\* - Neither the name of Oracle or the names of its

\* contributors may be used to endorse or promote products derived

\* from this software without specific prior written permission.

\*

\* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS

\* IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,

\* THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR

\* PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR

\* CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,

\* EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,

\* PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR

\* PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF

\* LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING

\* NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS

\* SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

\*/

/\*

\* TableRenderDemo.java requires no other files.

\*/

//Использование таблиц Swing - JTable (Render-Визуализатор)

//https://docs.oracle.com/javase/tutorial/uiswing/components/table.html

**import** javax.swing.DefaultCellEditor;

**import** javax.swing.JComboBox;

**import** javax.swing.JFrame;

**import** javax.swing.JPanel;

**import** javax.swing.JScrollPane;

**import** javax.swing.JTable;

**import** javax.swing.table.AbstractTableModel;

**import** javax.swing.table.DefaultTableCellRenderer;

**import** javax.swing.table.TableCellRenderer;

**import** javax.swing.table.TableColumn;

**import** java.awt.Component;

**import** java.awt.Dimension;

**import** java.awt.GridLayout;

/\*\*

\* TableRenderDemo is just like TableDemo, except that it

\* explicitly initializes column sizes and it uses a combo box

\* as an editor for the Sport column.

\*/

**public** **class** TableRenderDemo **extends** JPanel {

**private** **boolean** DEBUG = **false**;

**public** TableRenderDemo() {

**super**(**new** GridLayout(1,0));

JTable table = **new** JTable(**new** MyTableModel());

table.setPreferredScrollableViewportSize(**new** Dimension(500, 70));

table.setFillsViewportHeight(**true**);

//Create the scroll pane and add the table to it.

JScrollPane scrollPane = **new** JScrollPane(table);

//Set up column sizes.

initColumnSizes(table);

//Fiddle with the Sport column's cell editors/renderers.

setUpSportColumn(table, table.getColumnModel().getColumn(2));

//Add the scroll pane to this panel.

add(scrollPane);

}

/\*

\* This method picks good column sizes.

\* If all column heads are wider than the column's cells'

\* contents, then you can just use column.sizeWidthToFit().

\*/

**private** **void** initColumnSizes(JTable table) {

MyTableModel model = (MyTableModel)table.getModel();

TableColumn column = **null**;

Component comp = **null**;

**int** headerWidth = 0;

**int** cellWidth = 0;

Object[] longValues = model.longValues;

TableCellRenderer headerRenderer =

table.getTableHeader().getDefaultRenderer();

**for** (**int** i = 0; i < 5; i++) {

column = table.getColumnModel().getColumn(i);

comp = headerRenderer.getTableCellRendererComponent(

**null**, column.getHeaderValue(),

**false**, **false**, 0, 0);

headerWidth = comp.getPreferredSize().width;

comp = table.getDefaultRenderer(model.getColumnClass(i)).

getTableCellRendererComponent(

table, longValues[i],

**false**, **false**, 0, i);

cellWidth = comp.getPreferredSize().width;

**if** (DEBUG) {

System.***out***.println("Initializing width of column "

+ i + ". "

+ "headerWidth = " + headerWidth

+ "; cellWidth = " + cellWidth);

}

column.setPreferredWidth(Math.*max*(headerWidth, cellWidth));

}

}

**public** **void** setUpSportColumn(JTable table,

TableColumn sportColumn) {

//Set up the editor for the sport cells.

JComboBox comboBox = **new** JComboBox();

comboBox.addItem("Snowboarding");

comboBox.addItem("Rowing");

comboBox.addItem("Knitting");

comboBox.addItem("Speed reading");

comboBox.addItem("Pool");

comboBox.addItem("None of the above");

sportColumn.setCellEditor(**new** DefaultCellEditor(comboBox));

//Set up tool tips for the sport cells.

DefaultTableCellRenderer renderer =

**new** DefaultTableCellRenderer();

renderer.setToolTipText("Click for combo box");

sportColumn.setCellRenderer(renderer);

}

**class** MyTableModel **extends** AbstractTableModel {

**private** String[] columnNames = {"First Name",

"Last Name",

"Sport",

"# of Years",

"Vegetarian"};

**private** Object[][] data = {

{"Kathy", "Smith",

"Snowboarding", **new** ~~Integer~~(5), **new** ~~Boolean~~(**false**)},

{"John", "Doe",

"Rowing", **new** ~~Integer~~(3), **new** ~~Boolean~~(**true**)},

{"Sue", "Black",

"Knitting", **new** ~~Integer~~(2), **new** ~~Boolean~~(**false**)},

{"Jane", "White",

"Speed reading", **new** ~~Integer~~(20), **new** ~~Boolean~~(**true**)},

{"Joe", "Brown",

"Pool", **new** ~~Integer~~(10), **new** ~~Boolean~~(**false**)}

};

**public** **final** Object[] longValues = {"Jane", "Kathy",

"None of the above",

**new** ~~Integer~~(20), Boolean.***TRUE***};

**public** **int** getColumnCount() {

**return** columnNames.length;

}

**public** **int** getRowCount() {

**return** data.length;

}

**public** String getColumnName(**int** col) {

**return** columnNames[col];

}

**public** Object getValueAt(**int** row, **int** col) {

**return** data[row][col];

}

/\*

\* JTable uses this method to determine the default renderer/

\* editor for each cell. If we didn't implement this method,

\* then the last column would contain text ("true"/"false"),

\* rather than a check box.

\*/

**public** Class getColumnClass(**int** c) {

**return** getValueAt(0, c).getClass();

}

/\*

\* Don't need to implement this method unless your table's

\* editable.

\*/

**public** **boolean** isCellEditable(**int** row, **int** col) {

//Note that the data/cell address is constant,

//no matter where the cell appears onscreen.

**if** (col < 2) {

**return** **false**;

} **else** {

**return** **true**;

}

}

/\*

\* Don't need to implement this method unless your table's

\* data can change.

\*/

**public** **void** setValueAt(Object value, **int** row, **int** col) {

**if** (DEBUG) {

System.***out***.println("Setting value at " + row + "," + col

+ " to " + value

+ " (an instance of "

+ value.getClass() + ")");

}

data[row][col] = value;

fireTableCellUpdated(row, col);

**if** (DEBUG) {

System.***out***.println("New value of data:");

printDebugData();

}

}

**private** **void** printDebugData() {

**int** numRows = getRowCount();

**int** numCols = getColumnCount();

**for** (**int** i=0; i < numRows; i++) {

System.***out***.print(" row " + i + ":");

**for** (**int** j=0; j < numCols; j++) {

System.***out***.print(" " + data[i][j]);

}

System.***out***.println();

}

System.***out***.println("--------------------------");

}

}

/\*\*

\* Create the GUI and show it. For thread safety,

\* this method should be invoked from the

\* event-dispatching thread.

\*/

**private** **static** **void** createAndShowGUI() {

//Create and set up the window.

JFrame frame = **new** JFrame("TableRenderDemo");

frame.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

//Create and set up the content pane.

TableRenderDemo newContentPane = **new** TableRenderDemo();

newContentPane.setOpaque(**true**); //content panes must be opaque

frame.setContentPane(newContentPane);

//Display the window.

frame.pack();

frame.setVisible(**true**);

}

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

//Schedule a job for the event-dispatching thread:

//creating and showing this application's GUI.

javax.swing.SwingUtilities.*invokeLater*(**new** Runnable() {

**public** **void** run() {

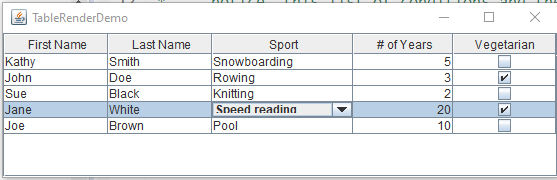
*createAndShowGUI*();

}

});

}

}



# Пример вывода информации из базы данных в таблицу Jtable Swing

**import** java.awt.BorderLayout;

**import** java.awt.Color;

**import** java.awt.Font;

**import** java.sql.Connection;

**import** java.sql.DriverManager;

**import** java.sql.ResultSet;

**import** java.sql.ResultSetMetaData;

**import** java.sql.SQLException;

**import** java.sql.Statement;

**import** java.util.ArrayList;

**import** java.util.Vector;

**import** java.util.logging.Level;

**import** java.util.logging.Logger;

**import** javax.swing.ImageIcon;

**import** javax.swing.JFrame;

**import** javax.swing.JPanel;

**import** javax.swing.JScrollPane;

**import** javax.swing.JTable;

**import** javax.swing.table.DefaultTableModel;

**public** **class** MyFrame **extends** JFrame {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**static** ArrayList<Book> *library* = **new** ArrayList<Book>();

**static** JTable *table* = **new** JTable();

**static** Object[] *columnsNameТ*;

**static** Object[] *arr*;

**public** MyFrame() {

**this**.setTitle("Bind JTable From MySQL DataBase");

**this**.setIconImage(**new** ImageIcon("src/book.png").getImage());

setLocationRelativeTo(**null**);

setSize(900, 500);// width, height

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

setVisible(**true**);

}

// create the connection

**static** Connection getConnection() {

Connection con = **null**;

**try** {

con = DriverManager.*getConnection*("jdbc:mysql://localhost/LIBRARY", "root", "root");

} **catch** (SQLException ex) {

Logger.*getLogger*(MyFrame.**class**.getName()).log(Level.***SEVERE***, **null**, ex);

}

**return** con;

}

// fill the an arrayList from database

**static** ArrayList<Book> getBook() {

// ArrayList<Book> library = new ArrayList<Book>();

Connection con = *getConnection*();

Statement st;

ResultSet rs;

Book book;

**try** {

st = con.createStatement();

rs = st.executeQuery("SELECT \* FROM book");

**while** (rs.next()) {

book = **new** Book(rs.getInt("bookId"), rs.getString("bookAuthor"), rs.getString("bookName"),

rs.getString("bookPublisher"), rs.getString("bookPrice"), rs.getString("bookQuantity"),

rs.getString("bookVolume"));

// для отладки Display values

System.***out***

.print(book.bookId + " " + book.bookAuthor + " \"" + book.bookName + "\" " + book.bookPublisher

+ " " + book.bookPrice + " " + book.bookQuantity + " " + book.bookVolume + " " + "\n");

*library*.add(book);

}

//получение названий колонок из базы данных - получение метаданных

ResultSetMetaData md = rs.getMetaData();

**int** columnCount = md.getColumnCount();

//создание вектора названий колонок из базы данных - store column names

Vector<String> columns = **new** Vector<String>(columnCount);

**for** (**int** i = 1; i <= columnCount; i++) {// store column names

columns.add(md.getColumnName(i));

// System.out.println("--" + md.getColumnName(i));//для отладки

}

*arr* = columns.toArray();//превращение Vector в массив Object[], понадобится для

// получение имен столбцов БД

// To remove previously added rows

// while(table.getRowCount() > 0)

// {

// ((DefaultTableModel) table.getModel()).removeRow(0);

// }

// int columnsT = rs.getMetaData().getColumnCount();

// while(rs.next())//не заходит сюда

// {

// columnsNameТ = new Object[columnsT];

// System.out.println("columnsТ "+ columnsT);

// for (int i = 1; i <= columnsT; i++)

// {

// columnsNameТ[i - 1] = rs.getObject(i);

// System.out.println("--"+columnsNameТ[i - 1] );//для отладки

// }

//// ((DefaultTableModel) table.getModel()).insertRow(rs.getRow()-1,columnsNameТ);

// }

} **catch** (SQLException ex) {

Logger.*getLogger*(MyFrame.**class**.getName()).log(Level.***SEVERE***, **null**, ex);

}

**return** *library*;

}

//??можно использовать для изменения размера шрифта

**private** **static** **void** enlargeFont(MyFrame window, **float** factor) {

window.setFont(window.getFont().deriveFont(window.getFont()

.getSize() \* factor));

}

/\*

\* create and populate a jtable from the arraylist who is

\* populated from mysql

\* database

\*/

**public** **static** **void** createAndShowGUI() {

// JTable table = new JTable();

DefaultTableModel model = **new** DefaultTableModel();

//установка названий колонок таблицы JTable table

//так лучше не делать, т.к. при изменении таблицы

//надо менять массив Object[] columnsName

//названия колонок можно получить при помощи метаданных

//см. rs.getMetaData()

// Object[] columnsName = new Object[7];

//

// columnsName[0] = "BookId";

// columnsName[1] = "bookAuthor";

// columnsName[2] = "bookName";

// columnsName[3] = "bookPublisher";

// columnsName[4] = "bookPrice";

// columnsName[5] = "bookQuantity";

// columnsName[6] = "bookVolume";

//

// model.setColumnIdentifiers(columnsName);

Object[] rowData = **new** Object[7];

**int** librarySize = *getBook*().size();

// for (int i=0; i<6;i++)

// System.out.println(" \*\*\*\*\*" + (String)arr[i]);//для отладки

// установка названий колонок таблицы JTable table

model.setColumnIdentifiers(*arr*);// без выполнения getBook() arr не инициализируется

**for** (**int** i = 0; i < librarySize; i++) {

rowData[0] = *library*.get(i).getBookId();

rowData[1] = *library*.get(i).getBookAuthor();

rowData[2] = *library*.get(i).getBookName();

rowData[3] = *library*.get(i).getBbookPublisher();

rowData[4] = *library*.get(i).getBookPrice();

rowData[5] = *library*.get(i).getBookQuantity();

rowData[6] = *library*.get(i).getBookVolume();

model.addRow(rowData);

}

*table*.getTableHeader().setFont(**new** Font("Arial", Font.***BOLD***, 15));// Изменение шрифта в заголовке JTable

*table*.setFont(**new** Font("Arial", Font.***PLAIN***, 15));// Изменение шрифта в JTable

// https://colorscheme.ru/

Color fg = **new** Color(0xFFFD73);

*table*.getTableHeader().setBackground(fg);

*table*.setBackground(**new** Color(0x7ff7f4));

*table*.setForeground(**new** Color(0x007f36));

*table*.setModel(model);

MyFrame window = **new** MyFrame();

JPanel panel = **new** JPanel();

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

JScrollPane pane = **new** JScrollPane(*table*);

panel.add(pane, BorderLayout.***CENTER***);

window.setContentPane(panel);

}

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

javax.swing.SwingUtilities.*invokeLater*(**new** Runnable() {

**public** **void** run() {

*createAndShowGUI*();

}

});

}

}



