Конспект лекций по Java. Занятие 18

В.Фесюнов

1. События и их реализация в JFC

Понятие **"событие"** используется во многих современных системах разработки программного обеспечения. Реализация событийной модели различна в разных системах, но принципы во многом схожи.

Событие (event) — это то, что может произойти. Например, нажатие клавиши, передвижение курсора мыши и даже изменения свойства какого-то объекта. В программе все время что-то происходит и это тоже события с абстрактной точки зрения. Но событие как программный термин используется тогда, когда мы говорим о событиях, которые могут произойти в произвольный момент времени и мы можем реализовать реакцию на это событие в виде некоторого программного кода, который должен быть выполнен, когда данное событие произошло.

Для большинства событий характерна **"объектность"** . Так, если мы ловим событие нажатия на клавишу, то для нас важно не только само это событие, но и некоторый объект, например, поле ввода, в котором находился курсор в момент нажатия клавиши. Более того, для разных полей хотелось бы иметь возможность написать разные фрагменты кода по перехвату этого события.

Также для события характерно наличие каких-либо параметров данного события. Например, нас не просто интересует сам факт нажатия на клавишу, а еще и то, какая именно клавиша была нажата. В другом случае нас может интересовать другая информация, например, при клике мышкой — позиция курсора мыши.

В объектно-ориентированных языках для объединения параметров события обычно применяют некоторый объект, ассоциированный с данным событием, который хранит необходимую информацию по данному событию. Обычно такой объект называют

Copyright © 1999-2004 Javable.com. All rights reserved.

объектом-событием *(Event Object)* или просто **событием** *(Event)* .

И последнее, что нужно сказать о термине **событие** , как об абстрактном понятии, это то, что события бывают низкоуровневыми и высокоуровневыми. Это в некоторой степени условное деление, но о нем нужно знать. Приведем пример.

Нажатие на клавишу *Tab* можно ловить как событие нажатия на клавишу. Но, с другой стороны, нажатие на *Tab* приводит к переходу фокуса на другую визуальную компоненту и это тоже можно ловить как событие (но уже совершенно другое). Из этого примера видно, что одно действие может приводить к порождению целого ряда событий. Соответственно у нас есть выбор, какие из этих событий мы собираемся обрабатывать.

При написании системных библиотек мы наверняка захотим обрабатывать низкоуровневые события, типа нажатия на клавишу, и сами будем порождать другие события, более высокоуровневые. При написании прикладных программ, напротив, лучше иметь дело с более абстрактными событиями, например — нажатие на кнопку.

Разберем последний пример подробнее. Формально, кнопка — это область экрана, на которой нарисовано нечто, что похоже на кнопку. Нажатие на кнопку — это клик мышкой в момент, когда ее курсор находится над этой областью, или нажатие клавиши *Enter* или *"пробел"* , когда фокус находится на данной кнопке, а в некоторых случаях, даже нажатие некоторой "горячей клавиши", ассоциированной с данной кнопкой. Система программирования данного языка ловит все эти низкоуровневые события и генерирует для нас более высокоуровневое событие — нажатие на кнопку. Если она (система программирования) это делает, то мы в программе можем реализовать обработку нажатия на кнопку и не задумываться над этими всеми деталями.

2. Событийная модель в JFC

Теперь рассмотрим непосредственно событийную модель Java.

Во-первых, нужно отметить, что она полностью реализована средствами стандартной библиотеки Java (**JFC** — Java Foundation Classes).

Во-вторых, следует сказать, что событийная модель, в свою очередь, используется в

Copyright © 1999-2004 Javable.com. All rights reserved.

других программных парадигмах JFC как средство реализации или составная часть. К ним относятся **MVC-архитектура** (от Model-View-Controller), **Java Beans** и другие понятия.

Так что знать основы событийной модели Java очень важно.

Мы не будем рассматривать ее полностью, во всех аспектах, а лишь с точки зрения реального использования. Внутренняя реализация событийной модели и ряд нюансов останутся нерассмотренными. Более подробную информацию можно найти в документации. См., например, http://java.sun.com/products/javabeans/docs/spec.html или http://java.sun.com/docs/books/tutorial/uiswing/overview/event.html .

Базовые классы, на которых основана событийная модель, находятся в пакете `java.util` .

Собственно объект-событие (*event state object*) это — объект класса, порожденного от класса `EventObject` . Как указывалось, этот объект является носителем параметров произошедшего события. Рассмотрим документацию по `EventObject` .

Из нее можно сделать вывод, что минимальный набор параметров события — это объект источник события, который может быть получен методом

public Object getSource()

класса `EventObject` . Поскольку `EventObject` является базовым классом для всех остальных классов-событий, то данный метод присутствует во всех этих классах.

Второй вывод, который можно сделать из документации, это само наличие объекта - источника события. Т.е. любое событие в JFC всегда порождается некоторым объектом, какого-то класса.

Итак, мы имеем два действующих лица — **объект-источник** и **объект-событие** . Третьим действующим лицом является **"слушатель"** (*лисенер, listener*).

3. Слушатели событий

Со слушателями событий мы уже встречались в примерах. Сейчас разберем это понятие подробнее.

Copyright © 1999-2004 Javable.com. All rights reserved.

Слушателем может быть объект любого класса, удовлетворяющего определенному интерфейсу — интерфейсу-слушателю. Все интерфейсы-слушатели порождены от базового интерфейса `EventListener` . Если мы посмотрим документацию по этому интерфейсу, то увидим, что он не имеет никаких методов. Это означает, что все слушатели имеют различные методы для прослушивания событий.

Все интерфейсы-слушатели (за редким исключением) имеют имена *XXXListener* . Здесь под *XXX* подразумевается некоторое имя события. Например, `ActionListener` или `AWTEventListener` , и т.д.

Объекты — источники событий должны быть объектами класса, который имеет методы для регистрации слушателя *addXXXListener* и отключения слушателя *removeXXXListener* . Соответственно, каждый конкретный класс — источник событий позволяет подключать к объекту-источнику только слушатели определенных типов, для которых в классе есть методы подключения *addXXXListener* .

Работа модели событий основана на том, что при наступлении какого-либо события объект-источник вызывает определенные методы интерфейса *XXXListener* для всех зарегистрировавшихся объектов-слушателей.

Слушатели представлены в JFC в виде интерфейсов, а не классов потому, что по смыслу они сами по себе ничего не должны делать. Они предоставляют нам возможность написать программный код, который будет выполняться при наступлении какого-либо события.

Рассмотрим для примера простейший интерфейс-слушатель `ActionListener` из пакета `java.awt.event` . Он содержит единственный метод

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

Этот метод вызывается, когда выполнено некоторое действие.

Что это за действие и когда оно происходит, самим этим интерфейсом не определено. Но в ряде классов имеются методы `addActionListener` , которые позволяют зарегистрировать слушателя типа `ActionListener` для прослушивания тех или иных событий. Что это за события с содержательной точки зрения (т.е. в каких случаях они происходят) — это описано в классе, имеющем метод `addActionListener` .

Copyright © 1999-2004 Javable.com. All rights reserved.

По рассмотренным примерам мы знаем, что для объекта класса `JButton` можно зарегистрировать слушателя типа `ActionListener` . Обратимся к документации по классу `JButton` .

В документации мы увидим, что в самом классе `JButton` метод `addActionListener` не определен, но в его базовом классе `AbstractButton` такой метод есть.

К сожалению, в документации по данному методу не описано, когда вызывается метод `actionPerformed` интерфейса `ActionListener` и можно лишь догадаться, что именно при нажатии данной кнопки. Документация лишь отсылает нас к соответствующему руководству How to Use Buttons, Check Boxes, and Radio Buttons .

Вспомним рассмотренные ранее примеры. Так в программе Dialog4.java из 9-го занятия присутствует такой фрагмент.

```
btn.addActionListener(new ActionListener(){
     public void actionPerformed(ActionEvent e) {
         lbl.setText("Нажата кнопка");
     }
});
```

В данном фрагменте использован аппарат анонимных классов для сокращения объема кода. Реализуем данный фрагмент в другом эквивалентном виде, расписав подробно каждый этап.

```
// 1. Класс ButtonListener1 реализует интерфейс ActionListener
class ButtonListener1 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        lbl.setText("Нажата кнопка");
    }
}
// 2. Создаем объект слушатель класса ButtonListener1
ActionListener listener = new ButtonListener1();
// 3. Для объекта — источника событий (btn) регистрируем слушателя (listener)
btn.addActionListener(listener);
```

В таком варианте более наглядно видно, что будет происходить. При нажатии на кнопку объект `btn` сгенерирует событие `ActionEvent` , т.е. фактически он создаст

Copyright © 1999-2004 Javable.com. All rights reserved.

объект класса `ActionEvent` и вызовет метод `actionPerformed` с данным объектом в качестве параметра для всех зарегистрировавшихся слушателей.

Таким образом, в нужный момент будет вызван метод `actionPerformed` , который мы запрограммировали в классе `ButtonListener1` . В нашем примере этот метод никак не использует информацию из объекта события (параметр `e`). В данном случае она не нужна. Все, что нужно, нам уже известно — произошло нажатие на кнопку `btn` .

Представим себе другую ситуацию. Пусть у нас в программе несколько кнопок. В этом случае у нас есть выбор из двух вариантов реализации реакции на нажатие кнопок. Первый состоит в создании для каждой кнопки отдельного класса, отдельного объекта этого класса и реализации в каждом из методов `actionPerformed` необходимых действий. Тогда, как и в примере, нам не потребуется информация из параметра метода `actionPerformed` .

Второй вариант состоит в создании одного класса-слушателя, одного объекта-слушателя и в программировании всех действий в рамках одного метода `actionPerformed` . Вот здесь нам обязательно потребуется параметр этого метода, т.к. только с его помощью мы сможем узнать, какая именно кнопка была нажата.

Приведем в шаблонном виде реализацию данного варианта

```
JButton btn1, btn2, btn3;
...
ActionListener listener = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        JButton btn = (JButton)e.getSource();
        if ( btn == btn1 ) {
             здесь действия по нажатию на кнопку btn1
        } else if ( btn == btn2 ) {
             здесь действия по нажатию на кнопку btn2
        } else if ( btn == btn3 ) {
             здесь действия по нажатию на кнопку btn3
        }
    }
}
```


Copyright © 1999-2004 Javable.com. All rights reserved.

```
btn1.addActionListener(listener);
btn2.addActionListener(listener);
btn3.addActionListener(listener);
```

Оба варианта являются приемлемыми и в каждой конкретной ситуации можно выбрать один из них.

- Отметим, что, если бы не аппарат анонимных классов, который дает возможность записать реализацию слушателей в компактной форме, то второй вариант был бы более предпочтительным.
- Данный пример демонстрирует еще одну особенность слушателей. Мы не только можем к одному объекту — источнику событий подключить ряд слушателей, но и одного слушателя подключить к нескольким объектам — источникам событий.

Интерфейс `ActionListener` очень прост, он содержит всего один метод. Но существуют и более сложные интерфейсы слушателей, например, `WindowListener` , с которым мы тоже уже встречались в примерах.

Обратимся опять к документации. Интерфейс `WindowListener` имеет такие методы.

```
public void windowOpened(WindowEvent e)
```

Вызывается при открытии окна (когда оно становится видимым в первый раз).

```
public void windowClosed(WindowEvent e)
```

Вызывается при закрытии окна.

```
public void windowActivated(WindowEvent e)
```

Вызывается при активизации окна.

```
public void windowDeactivated(WindowEvent e)
```

Вызывается при деактивизации окна.

```
public void windowClosing(WindowEvent e)
```

Вызывается при попытке закрыть окно.

```
public void windowIconified(WindowEvent e)
```


Copyright © 1999-2004 Javable.com. All rights reserved.

Вызывается при сворачивании окна в иконку на панели.

```
public void windowDeiconified(WindowEvent e)
```

Вызывается при развертывании окна из иконки в нормальное состояние.

Как видим здесь довольно много методов. Они вызываются в определенных ситуациях в некоторой последовательности. Так, например, при сворачивании окна в иконку будет вызван сначала метод `windowDeactivated` , потом `windowIconified` .

Обычно для интерфейсов-слушателей, имеющих более одного метода, создаются вспомогательные классы-адаптеры. В этих классах все, или большинство методов данного интерфейса реализованы и обычно ничего не выполняют.

Это сделано из практических соображений. Если создавать класс, удовлетворяющий, скажем, интерфейсу `WindowListener` , то в нем нужно реализовать все методы данного интерфейса. В то время, как чаще всего нас интересует один или несколько методов данного интерфейса. Тут на помощь приходят классы-адаптеры. Свой класс можно унаследовать от класса-адаптера и переопределить в нем нужные методы.

Именно так мы и делали в примерах для организации завершения приложения при закрытии главного окна.

```
  WindowListener wndCloser = new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  };
  addWindowListener(wndCloser);
```

Здесь для построения анонимного класса использован не интерфейс `WindowListener` , как мы делали при работе с `ActionListener` , а класс-адаптер `WindowAdapter` . В нем все методы интерфейса `WindowListener` реализованы, и мы можем переопределить только один интересующий нас метод — `windowClosing` .

4. События и потоки

Copyright © 1999-2004 Javable.com. All rights reserved.

Мы не рассматривали потоки и не сможем себе сейчас ясно представить, как они связаны со слушателями.

Но, забегая вперед, надо сказать, что такая связь во многих случаях имеется и выражается она в следующем. Обработка событий визуальной библиотеки Java реализована таким образом, что на время работы некоторого метода какого-то слушателя, никакие действия со стороны пользователя не вызывают никакого эффекта на данное приложение, перерисовка визуальных компонент приложения не выполняется. Это следует учитывать и не выполнять долго работающие алгоритмы внутри слушателя.

В тех случаях, когда требуется выполнение какого-то долго работающего алгоритма, следует предпринимать специальные меры для того, чтобы не заблокировать работу приложения.

Применение событий и слушателей на практике еще не раз встретится нам в ходе дальнейшего изложения. В частности, в следующей теме, посвященной визуальному представлению информации в виде списков или таблиц.

5. Класс JList библиотеки Swing

Вернемся к домашнему заданию на сегодняшнее занятие — к приложению "Записная книжка". Мы создали уже достаточно "умное" приложение, но все же оно не лишено недостатков.

Основной из них заключается в том, что мы можем просматривать записи нашей записной книжки только по одной, и не видим всю картину в целом. Т.е. нам желательно было бы поместить на окно нашего приложения список со скроллингом, вмещающий все записи и дать возможность выбора и редактирования конкретной записи.

Для этого нам требуется познакомиться с возможностями создания таблиц.

Библиотека Swing содержит два основных средства работы с таблицами — это классы `JList` и `JTable`.

По сравнению с `JList` класс `JTable` содержит расширенные средства. Так `JList`

Copyright © 1999-2004 Javable.com. All rights reserved.

ориентирован на просмотр одной колонки, а `JTable` позволяет просматривать несколько колонок, а также редактировать поля колонок.

Однако, во многих случаях предпочтительнее использовать все-таки именно `JList` , хотя-бы потому, что он проще. Базовые возможности, заложенные в основу библиотеки Swing, позволяют изменять поведение класса `JList` в весьма широких пределах, обеспечивая, в частности, и возможность вывода списка, состоящего из нескольких колонок.

Рассмотрим документацию по `JList` . Класс `JList` имеет 4 конструктора.

```
public JList()
```

Создает пустой список.

```
public JList(Object[] listData)
```

Список на основе массива объектов.

```
public JList(Vector listData)
```

Список из элементов вектора.

- Лучше было бы иметь конструктор public **JList** (List listData)

```
public JList(ListModel dataModel)
```

Самый абстрактный и гибкий вариант. Можно создать свой класс, реализующий интерфейс `ListModel` и обеспечивающий элементы для списка из любого источника (обычно такой класс строят на основе абстрактного класса `AbstractListModel` , в котором уже реализована часть необходимой функциональности).

Если не говорить о конструкторе по умолчанию, то самый простой конструктор — это конструктор, принимающий массив объектов. Этот конструктор применим тогда, когда у нас есть готовый массив объектов, и мы хотим отобразить его в виде списка на экране.

Единственный вопрос состоит в том, как `JList` отображает эти объекты, ведь список должен состоять из строк.

Здесь применен достаточно стандартный для JFC прием. В списке отображаются

Copyright © 1999-2004 Javable.com. All rights reserved.

строки, полученные путем вызовов метода `toString` для объектов, входящих в список.

Рассмотрим такой фрагмент

```
String[] data = {"one", "two", "three", "four"};
JList dataList = new JList(data);
JScrollPane scrollPane = new JScrollPane(dataList);
```

Здесь сформирован список `JList` на основе массива, этот список помещен внутрь `JSrollPane` (как и в случае с `JTextArea` , который мы рассматривали ранее). Осталось только добавить `scrollPane` на какую-то панель нашего приложения.

- Помещать `JList` внутрь `JSrollPane` — стандартный прием для обеспечения скроллируемости списка.

Второй конструктор применим тогда, когда у нас есть коллекция объектов класса `Vector` .

Но оба эти конструктора позволяют создать лишь статические списки. После создания списка нет никакой возможности изменить его.

Если обратиться к документации по `JList` , то мы там не найдем ни одного метода для добавления чего-либо в список. Это связано с тем, что класс `JList` является довольно сложной по структуре конструкцией и, как и многие другие классы Swing, построен на базе архитектуры MVC (*Model-View-Controller*).

Рассмотрение архитектуры MVC это отдельная тема, которую мы сейчас не будем затрагивать. Мы просто разберемся, как нам построить модифицируемый список.

Для этого нужно обратить внимание на интерфейс **ListModel** и классы **AbstractListModel** и **DefaultListModel** .

Список типа `JList` работает с данными, составляющими собственно список, не напрямую, а через специальный объект "модель данных". Интерфейс `ListModel` определяет минимальный набор методов, который требуется классу `JList` от "модели данных". Если обратиться к документации, то можно увидеть, что эти методы позволяют узнать количество элементов в списке (метод `getSize`), выбрать элемент

Copyright © 1999-2004 Javable.com. All rights reserved.

списка (метод `getElementAt`), а также зарегистрировать и отключить слушателей (методы `addListDataListener` и `removeListDataListener`).

Класс `AbstractListModel` является абстрактным вспомогательным классом, в котором реализован набор методов, необходимых для поддержки слушателей типа `ListDataListener` . Т.е. все реальные классы, реализующие "модель данных" для `JList` удобно строить на базе класса `AbstractListModel` — тогда не придется реализовывать методы `addListDataListener` и `removeListDataListener` .

И, наконец, класс `DefaultListModel` . Этот класс удовлетворяет интерфейсу `ListModel` и, что важно для нас, имеет методы для добавления элементов в список (методы `add` и `addElement`), модификации существующих элементов (`setElementAt`) и удаления из списка (ряд методов `remove...`).

- Можно построить и свой класс, реализующий интерфейс `ListModel` , определив в нем любые другие методы и придав ему требуемые свойства. Но, обычно достаточно класса `DefaultListModel` .

Резюмируя все это, можно сделать вывод, что для реализации модифицируемого списка типа `JList` можно

- Построить объект класса `DefaultListModel` .
- Построить объект класса `JList` с использованием конструктора

  ```
  public JList(ListModel dataModel)
  ```

  передав ему в качестве параметра построенный объект класса `DefaultListModel` .
- Выполнять модификации списка через объект класса `DefaultListModel` — визуальное представление списка при этом будет меняться автоматически.

5.1. Внешний вид отображаемого списка

Как уже упоминалось, внутренний список может быть списком объектов, а для внешнего его представления формируются строки с использованием метода `toString` , который применяется ко всем объектам списка. Соответственно, на экране отображается список строк, сформированных путем вызова `toString` .

В простейших случаях этого достаточно. Но возможности класса `JList` по внешнему представлению списка этим не ограничиваются.

Copyright © 1999-2004 Javable.com. All rights reserved.

Визуальное представление строк списка может быть более сложным, чем просто строка. Например, можно реализовать представление данных в несколько колонок или вставить в строки пиктограммы.

Для того чтобы манипулировать внешним представлением строк списка, нужно знать следующее.

Строками отображаемого списка являются метки класса `JLabel` . По умолчанию при формировании визуального представления строки текст надписи метки формируется вызовом метода `toString` . Поэтому можно отображать список любых объектов, какой бы сложной структуры они не были. Нужно только позаботиться о создании метода `toString` , чтобы он выдавал то, что мы хотим видеть в строке на экране.

Эти строки-метки формируются объектом класса, удовлетворяющего интерфейсу `ListCellRenderer` . Опять же в `JList` для этого использован стандартный для Swing прием — применение "рендерера" (*renderer* — представитель, "отображатель" ячейки списка, визуализатор). Соответственно, объект "рендерер", формируемый классом `JList` по умолчанию, можно заменить другим, который формирует требуемый внешний вид меток. Для замены визуализатора класс `JList` имеет метод `setRenderer` (см. документацию).

Мы не будем более углубляться в эту тему. Хороший пример использования визуализатора приводится в главе 10 (Chapter10.doc) книги "Swing" by Matthew Robinson and Pavel Vorobiev Там реализован класс `TabListCellRenderer` — достаточно универсальный класс для представления данных в несколько колонок. Разделителями колонок служат символы табуляции '\t'. Размеры колонок могут быть установлены по желанию разработчика.

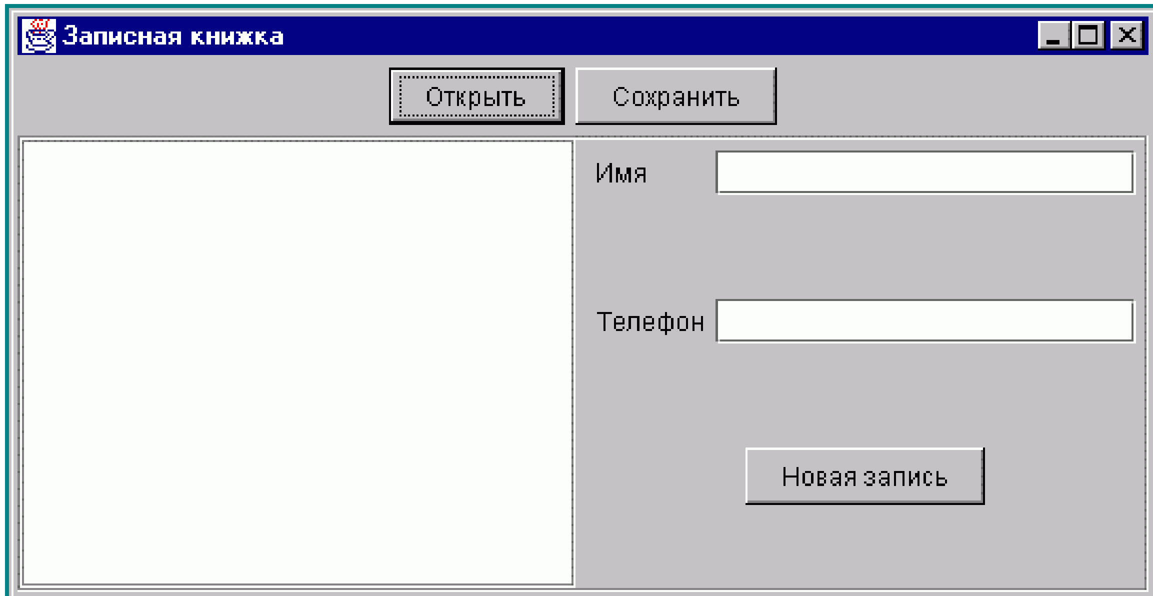
Для полноценной работы со списком `JList` нам еще необходимо разобраться с программированием действий по выбору элемента списка. Мы отложим это до следующего занятия, а сейчас попробуем применить то, что мы уже изучили, на практике.

6. Практическая работа

Переделаем наш пример так, чтобы слева от полей ввода находился список имен.

Copyright © 1999-2004 Javable.com. All rights reserved.

- Заменим ArrayList на DefaultListModel для облегчения работы с JList.
- Удалим btnLeft и btnRight со всем, что к ним относится.
- В классе Person реализуем метод toString().
- Изменим визуальное представление экрана так, чтобы он выглядел следующим образом:



- Реализуем сохранение списка в файле и его чтение из файла.

Copyright © 1999-2004 Javable.com. All rights reserved.