! "#$%&'( )&'*+, %" -./.0 12#3(+& 45

В.Фесюнов

60 7&8#")"9+3 -./.:;.<=0 >?"@")A&#+&

6060 BC3D2##E& $C",$(C2 FGHI<J KLHK;LMN;=0 + $"PE(+30 >?"@")A&#+&

606060 B"D@2#+& + +$%")QD"C2#+& $C3D2##"9" $C",$(C20 >?"@")A&#+&

На прошлом занятии мы начали разбирать вопрос создания и использования связанного свойства. Мы определили инструкцию или краткий перечень действий для создания связанного свойства.

Но некоторые пункты инструкции требуют более подробного рассмотрения.

Приведем еще раз эту инструкцию.

- **1.** Для регистрации/дерегистрации слушателя необходимо в бине реализовать два метода:

```
public void addPropertyChangeListener(PropertyChangeListener pcl)
```

и

```
public void removePropertyChangeListener(PropertyChangeListener pcl)
```

- **2.** Чтобы не реализовывать их вручную лучше воспользоваться существующим классом `java.beans.PropertyChangeSupport` (см. документацию).
- **3.** В set-методе связанного свойства необходимо добавить вызов метода класса `java.beans.PropertyChangeSupport — firePropertyChange`.
- **4.** В классе-слушателе необходимо реализовать интерфейс

`PropertyChangeListener` , т.е. в заголовке класса записать " `implements PropertyChangeListener` ", а в теле класса реализовать метод

```
public void propertyChange(PropertyChangeEvent evt)
```

- **5.** Создать объект-слушатель и зарегистрировать его как слушателя нашего бина при помощи метода `addPropertyChangeListener` , который был нами реализован в п.1. Лучше всего это сделать сразу после порождения объекта-слушателя, например,

```
MyLitener obj = new MyListener();
myBean.addPropertyChangeListener(obj);
```

где `myBean` — наш бин (имеется в виду объект, а не класс).

Пункт 4-й должен быть реализован для каждого класса-слушателя, а п.5 — для каждого порожденного объекта-слушателя.

Разберемся подробнее с пунктами 2 и 3.

С событиями мы знакомились в рамках 20-го занятия. Но там мы рассматривали подключение слушателей к объектам источникам событий. Сейчас же нам нужно реализовать генерацию событий. Наш бин должен генерировать событие `PropertyChangeEvent` при изменении связанного свойства (п.3). Кроме того, согласно правилам событийной модели Java он должен обеспечивать регистрацию/дерегистрацию слушателей при помощи соответствующих методов `add...Listener` / `remove...Listener` (п.2).

Т.е. нам нужно обеспечить наличие в бине некоторого списка слушателей, а также методы `addPropertyChangeListener` и `removePropertyChangeListener` .

К счастью, нам не требуется программировать все это. Соответствующий инструментарий уже подготовлен в пакете `java.beans` — это класс `java.beans.PropertyChangeSupport` . Он обеспечивает регистрацию слушателей и методы `firePropertyChange` , которые можно использовать в тех местах, где требуется сгенерировать событие, т.е. в set-методах, которые изменяют значение связанных атрибутов.

Разберем это на примере.

Пусть мы имеем некоторый бин `SomeBean` с одним свойством `someProperty`:

```
public class SomeBean {

    private String someProperty = null;

    public SomeBean() {
    }

    public String getSomeProperty() {
        return someProperty;
    }

    public void setSomeProperty(String value) {
        someProperty = value;
    }

}
```

Переделаем его так, чтобы сойство `someProperty` стало связанным:

```
import java.beans.*;

public class SomeBean {

    private String someProperty = null;

    private PropertyChangeSupport pcs;

    public SomeBean() {
        pcs = new PropertyChangeSupport(this);
    }

    public void addPropertyChangeListener(PropertyChangeListener pcl) {
        pcs.addPropertyChangeListener(pcl);
    }
```

```
    public void removePropertyChangeListener(PropertyChangeListener pcl) {
        pcs.removePropertyChangeListener(pcl);
    }

    public String getSomeProperty() {
        return someProperty;
    }

    public void setSomeProperty(String value) {
        pcs.firePropertyChange("someProperty", someProperty, value);
        someProperty = value;
    }

}
```

Здесь мы реализовали пункты 1, 2 и 3 приведенной инструкции. Остальные пункты относятся к использованию связанного свойства и для их демонстрации потребуется более реальный пример.

Для обеспечения механизма генерации событий в классе `SomeBean` создан объект класса `PropertyChangeSupport` (поле `pcs`). И все действия по регистрации/дерегистрции слушателей, по собственно генерации событий "переадресуются" этому объекту, который за нас выполняет всю эту рутинную работу.

Так, например, метод

public void addPropertyChangeListener(PropertyChangeListener pcl)

нашего класса просто обращается к одноименному методу класса `PropertyChangeSupport` .

В методе `setSomeProperty` перед собственно изменением значения свойства `someProperty` генерируется событие `PropertyChangeEvent` . Для этого вызывается метод `firePropertyChange` , который обеспечивает все необходимые для такой генерации действия.

Как видно из кода примера, результат не очень громоздкий, несмотря на то, что наш бин реализует достаточно сложное поведение.

606040 >?2'(+R&$'23 ?2P"(2

Добавим в наш пример редактирования файла (Dlg5.java) метку в нижнюю область экрана. Будем выводить в нее информацию: имя файла и количество строк файла. Для этого реализуем наш основной класс как бин со связанным свойством `currentFile`. А для метки построим класс-слушатель на основе `JLabel`, создадим объект этого класса, добавим его на экран и зарегистрируем как слушателя основного класса.

Проделайте это сами. Если не получится, решение можно загрузить отсюда: Dlg5.java .

6040 S9?2#+R&##E& $C",$(C2 FTH<=ML.N<;J KLHK;LMN;=0

Кроме понятия связанных свойств в JavaBeans есть понятие ограниченных свойств (constrained properties). Ограниченные свойства введены для того, чтобы была возможность запретить изменение свойства бина, если это необходимо. Т.е. бин будет как-бы спрашивать разрешение у зарегистрированных слушателей на изменение данного свойства. В случае, если слушатель не разрешает ему менять свойство, он генерирует исключение `PropertyVetoException`. Соответственно set-метод для ограниченного свойства должен иметь в своем описании throws PropertyVetoException. Это заставляет перехватывать это исключение в точке вызова этого set-метода. В результате прикладная программа, использующая этот бин, будет извещена, что ограниченное свойство не было изменено.

В остальном ограниченные свойства очень похожи на связанные свойства. Как и все свойства они имеют get- и set-методы. Но для них set-методы могут генерировать исключение PropertyVetoException. Т.е. они имеют вид

public void set < Property_name> (< Property_type> param) throws PropertyVetoException

Второе отличие заключается в именах методов для регистрации/ дерегистрации слушателей. Вместо методов `addPropertyChangeListener` и `removePropertyChangeListener` для ограниченных свойств применяются методы

addVetoableChangeListener(VetoableChangeListener v)

и

removeVetoableChangeListener(VetoableChangeListener v)

Здесь `VetoableChangeListener` — интерфейс с единственным методом

public void vetoableChange(PropertyChangeEvent evt) throws PropertyVetoException

По аналогии со вспомогательным классом `PropertyChangeSupport` , который используется при реализации связанных свойств, для ограниченных свойств в пакете `java.beans` есть вспомогательный класс `VetoableChangeSupport` . В нем реализованы алгоритмы, необходимые для поддержки событий ограниченных свойств.

Рассмотрим это на примере.

Вспомним класс `SomeBean` , рассмотренный ранее. Его свойство `someProperty` мы реализовали как связанное. Переделаем пример и реализуем его как ограниченное.

```
import java.beans.*;

public class SomeBean {

    private String someProperty = null;

    private VetoableChangeSupport vcs;

    public SomeBean() {
        vcs = new VetoableChangeSupport (this);
    }

    public void addVetableChangeListener(VetoableChangeListener pcl) {
        vcs.addVetoableChangeListener(pcl);
    }


    public void removeVetoableChangeListener(VetoableChangeListener pcl) {
        vcs.removeVetoableChangeListener(pcl);
    }

    public String getSomeProperty() {
        return someProperty;
    }

    public void setSomeProperty(String value) throws PropertyVetoException {
```

```
        vcs.fireVetoableChange("someProperty", someProperty, value);
        someProperty = value;
    }

}
```

Как видим, принципиально ничего не изменилось. Только вместо `PropertyChangeSupport` использован `VetoableChangeSupport` и в описании set-метода добавлено `throws PropertyVetoException`. Теперь `someProperty` является ограниченным свойством и зарегистрировавшийся слушатель может запретить его изменение.

Рассмотренные возможности организации связи бина с другими компонентами не являются единственно возможными. Бин, как и любой класс, может быть источником событий и/или слушателем. И эти события могут быть не связаны с изменением свойств бина.

В таких случаях обычно используют существующие события, типа `ActionEvent`, хотя можно построить и свои события. Лучше пользоваться существующими событиями, поскольку предполагается, что бин будет использоваться совместно с другими классами и бинами при разработке приложений и наличие в нем каких-то уникальных возможностей может помешать его интеграции в законченное приложение.