Возможности класса **java.awt.Desktop** позволяют запустить браузер с помощью одной строки:

```
/* пример # 22 : запуск браузера:  DesktopTest.java */
package chapt13;
import org.jdesktop.jdic.desktop.*;
import java.net.*;

public class DesktopTest {
  public static void main(String[] args) throws Exception {
    Desktop.browse(new URL("http://java.sun.com"));
  }
}
```

При необходимости можно легко использовать веб-браузер в приложении:

```
/* пример # 23 : использование web- браузера:  BrowserTest.java */
package chapt13;
import org.jdesktop.jdic.browser.WebBrowser;
import java.net.URL;
import javax.swing.JFrame;

public class BrowserTest {
  public static void main(String[] args) throws Exception {

    WebBrowser browser = new WebBrowser();
    browser.setURL(new URL("http://www.netbeans.org"));
    JFrame frame = new JFrame("Использование Browser");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().add(browser);
    frame.pack();
    frame.setSize(640,480);
    frame.setVisible(true);
  }
}
```

Визуальные компоненты JavaBeans

Визуальные компоненты являются классами языка Java, объекты которых отображаются визуально в проектируемом приложении с помощью средств визуальной разработки. Визуальные компоненты являются удобным средством при создании пользовательских интерфейсов. Обычно часть визуальной разработки приложений состоит в перетаскивании компонентов на форму и в определении их свойств, событий и обработчиков событий. При этом компонент представляет собой объект класса, для которого, кроме данных и методов, дополнительно установлены свойства и события класса. Свойства и события устанавливаются через имена методов на основе соглашения об именовании методов.

JavaBeans (бин) – многократно используемый программный компонент, которым можно манипулировать визуально при создании приложения. Бин реализуется в одном или нескольких взаимосвязанных классах. Основной класс бина должен иметь конструктор по умолчанию, который вызывается визуальной средой при создании экземпляра бина.

Каждый бин должен поддерживать следующие возможности:

- интроспекцию, позволяющую средам разработки анализировать, из чего состоит и как работает данный бин;
- поддержку событий (**events**) ;
- поддержку свойств (**properties**) ;
- сохраняемость (**persistence**).

Каждый бин должен быть сериализуемым. Визуальная среда при сохранении скомпилированного приложения сохраняет настройки компонента, сделанные пользователем в процессе разработки приложения путем сериализации бина. При повторной загрузке приложения эти настройки восстанавливаются. Для этого среда разработки десериализует бины из файла.

Свойства бинов

Каждый бин имеет свойства (**properties**), которые определяют, как он будет работать и как выглядеть. Эти свойства являются **private** или **protected** полями класса, которые доступны через специальные методы **get**Имя**()** и **set**Имя**()** (getters и setters), называемые также аксессорами. Так, утверждение "данный бин имеет свойство **name** типа **String**" означает, что у этого бина

```
//есть поле
   private String name;
//есть get-метод
   public String getName(){
      return name;
   }
//есть set-метод
   public void setString(String name){
      this.name = name;
   }
```

Пусть рассматривается, к примеру, компонент **JLabel**. Во-первых, **JLabel** удовлетворяет интерфейсу **Serializable**, во-вторых, имеет конструктор по умолчанию **public JLabel()** и, в-третьих, имеет ряд методов аксессоров, например **public String getText()**, **public void setText(String text)**. Исходя из этого, можно сделать выводы, что **JLabel** является бином, имеющим свойство **text**. Значение этого свойства отображается как заголовок метки. Кроме того, **JLabel** имеет и другие свойства.

Для свойства типа **boolean** в бинах вместо **get**-методов может быть использован **is**-метод. Например, **JLabel** имеет **boolean**-свойство **enabled**, унаследованное от класса **Component**. Для доступа к этому свойству имеются методы

```
public boolean isEnabled()
public void setEnabled(boolean b)
```

Правила построения методов доступа к атрибутам (аксессоров):

```
public void setИмяСвойства(ТипСвойства value);
public ТипСвойства getИмяСвойства();
public boolean isИмяСвойства().
```

Свойства бинов могут быть как базовых типов, так и объектными ссылками. Свойства могут быть индексированными, если атрибут бина массив. Для индексированных свойств выработаны следующие правила. Они должны быть описаны как поля-массивы, например

```
private String[] messages;
```

и должны быть объявлены следующие методы:

```
public ТипСвойства getИмяСвойства(int index);
public void setИмяСвойства(int index, ТипСвойства value);
public ТипСвойства [] getИмяСвойства();
public void setИмяСвойства(ТипСвойства [] value).
```

Так, для приведенного выше примера должны быть методы

```
public String getMessages(int index);
public void setMessages(int index, String message);
public String[] getMessages();
public void setMessages(String[] messages).
```

Кроме аксессоров, бин может иметь любое колличество других методов.

Интроспекция бинов при помощи Reflection API

Под интроспекцией понимается процесс анализа bean-компонента для установления его возможностей, свойств и методов. Для интроспекции можно воспользоваться классами и методами из библиотеки Reflection API. При использовании бина визуальная среда должна знать полное имя класса бина. По строковому имени класса статический метод **forName(String className)** класса **java.lang.Class** возвращает объект класса **Class**, соответствующий данному бину. Далее с помощью метода класса **Class getField()**, **getMethods()**, **getConstructors()** можно получить необходимую информацию о свойствах и событиях класса.

В частности, можно получить список всех **public**-методов данного класса. Исследуя их имена, можно выделить из них аксессоры и определить какие атрибуты (свойства) есть у данного бина и какого они типа. Все остальные методы, не распознанные как аксессоры, являются bean-методами.

В результате соответствующая визуальная разработки может построить диалог, в котором будет предоставлена возможность задавать значения этих атрибутов. Наличие конструктора по умолчанию позволяет построить объект bean-класса, **set**-методы позволяют установить в этом объекте значения атрибутов, введенные пользователем, а благодаря сериализации объект с заданными атрибутами можно сохранить в файл и востановить значение объекта при следующем сеансе работы с данной визуальной средой. Более того, можно изобразить на экране внешний вид бина (если это визуальный бин) в процессе разработки и менять этот вид в соответствии с задаваемыми пользователем значениями атрибутов.

События

Еще одним важным аспектом технологии JavaBeans является возможность бинов взаимодествовать с другими объектами, в частности, с другими бинами. JavaBeans реализует такое взаимодействие путем генерации и прослушивания событий.

В приложении к бинам взаимодействие объектов с бином через событийную модель выглядит так. Объект, который интересуется тем, что может произойти во внешнем по отношению к нему бине, может зарегистрировать себя как слушателя (**Listener**) этого бина. В результате при возникновении соответствующего события в бине будет вызван определенный метод данного объекта, которому в качестве параметра будет передан объект-событие (**event**). Причем если зарегистрировалось несколько слушателей, то эти методы будут последовательно вызваны для каждого слушателя.

Такой механизм взаимодействия является очень гибким, поскольку два объекта – бин и его слушатель – связаны только посредством данного метода и параметра-события.

Одним из способов экспорта событий является использование связанных свойств. Когда значение связанного свойства меняется, генерируется событие и передается всем зарегистрированным слушателям посредством вызова метода **proertyChange()**.

Создание и использование связанного свойства

Разберемся практически, как создавать и использовать связанные свойства. Начнем с события, которое должно быть сгенерированно при изменении связанного свойства. Это событие класса **java.beans.PropertyChangeEvent** (см. документацию).

Далее можно действовать по следующей инструкции.

1. Для регистрации/дерегистрации слушателя необходимо в бине реализовать два метода:
addPropertyChangeListener(PropertyChangeListener p) и **removePropertyChangeListener(PropertyChangeListener p)**;
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 **firePropertyChange()** класса **java.beans.PropertyChangeSupport**;
4. В классе-слушателе реализовать интерфейс **PropertyChangeListener**, т.е. в заголовке класса записать "**implements PropertChangeListener**", 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 реализован в п.1. Лучше всего это сделать сразу после порождения объекта-слушателя, например:

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

где **myBean** – создаваемый 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.

Следует разобрать подробнее пункты 2 и 3.

Сейчас необходимо реализовать генерацию событий. 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. нужно обеспечить наличие в бине некоторого списка слушателей, а также методы **addPropertyChangeListener()** и **removePropertyChangeListener()**.

К счастью, 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.

Предложенный механизм будет рассмотрен в следующем примере.

Пусть имеется некоторый бин **SomeBean** с одним свойством **someProperty**:

```
/* пример # 24 : простой bean-класс :  SomeBean.java */
package chapt13;
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 свойство **someProperty** стало связанным:

```
/* пример # 25 : bean-класс со связанным свойством:  SomeBean.java */
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 реализованы 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/дерегистрации слушателей по собственно генерации событий "переадресуются" этому объекту, который за нас выполняет всю эту рутинную работу.

Так, например, метод **addPropertyChangeListener(PropertyChangeListener pcl)** созданного 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.

Ограниченные свойства (contrained properties)

Кроме понятия связанных свойств, в JavaBeans есть понятие ограниченных свойств (contrained properties). Ограниченные свойства введены для того, чтобы была возможность запретить изменение свойства бина, если это необходимо. Т.е. бин будет как бы спрашивать разрешения 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**, что заставляет перехватывать это исключение в точке вызова данного **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** и имеют вид **public void <PropertyName>**(ТипСвойства **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 одним методом **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.

В качестве примера вспомним класс **SomeBean**, рассмотренный ранее. Его свойство **someProperty()** реализовано как связанное. Переделаем пример и реализуем это свойство как ограниченное.

```
/* пример # 26 : bean-класс с ограниченным свойством :  SomeBean.java */
import java.beans.*;
    public class SomeBean {
           private String someProperty = null;
           private VetoableChangeSupport vcs;
           public SomeBean(){
             vcs = new VetoableChangeSupport(this);
           }
           public void addVetoableChangeListener
           (VetoableChangeListener pcl){
             vcs.addVetoableChangeListener(pcl);
           }
           public void removeVetoableChangeListener
           (VetoableChangeListener pcl){
             pcs.removePropertyChangeListener(pcl);
           }

           public String getSomeProperty(){
                 return someProperty;
           }
           public  void  setSomeProperty(String  value)
           throws
           PropertyVetoException{
                vcs.fireVetoableChange("someProperty",
                someProperty, value);
                someProperty = value;
           }
    }
```

Как видно, 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.

Задания к главе 13

Вариант А

1. Создать апплет. Поместить на него текстовое поле **`JTextField`**, кнопку **`JButton`** и метку **`JLabel`**. В метке отображать все введенные символы, разделяя их пробелами.
2. Поместить в апплет две панели **`JPanel`** и кнопку. Первая панель содержит поле ввода и метку “Поле ввода”; вторая – поле вывода и метку “Поле вывода”. Для размещения в окне двух панелей и кнопки “Скопировать” использовать менеджер размещения **`BorderLayout`**.
3. Изменить задачу 2 так, чтобы при нажатии на кнопку “Скопировать” текст из поля ввода переносился в поле вывода, а поле ввода очищалось.
4. Задача 2 модифицируется так, что при копировании поля ввода нужно, кроме собственно копирования, организовать занесение строки из поля ввода во внутренний список. При решении использовать коллекцию, в частности **`ArrayList`**.
5. К условию задачи 2 добавляется еще одна кнопка с надписью “Печать”. При нажатии на данную кнопку весь сохраненный список должен быть выведен в консоль. При решении использовать коллекцию, в частности **`TreeSet`**.
6. Написать программу для построения таблицы значений функции $y = a\sqrt{x} * \cos(ax)$. Использовать метку **`JLabel`**, содержащую текст “Функция: $y = a\sqrt{x} * \cos(ax)$”; панель, включающую три текстовых поля **`JTextField`**, содержащих значения параметра, шага (например, 0.1) и количества точек. Начальное значение x=0. С каждым текстовым полем связана метка, содержащая его название. В приложении должно находиться текстовое поле со скроллингом, содержащее полученную таблицу.
7. Создать форму с набором кнопок так, чтобы надпись на первой кнопке при ее нажатии передавалась на следующую, и т.д.
8. Создать форму с выпадающим списком так, чтобы при выборе элемента списка на экране появлялись GIF-изображения, двигающиеся в случайно выбранном направлении по апплету.
9. В апплете изобразить прямоугольник (окружность, эллипс, линию). Направление движения объекта по экрану изменяется на противоположное щелчком по клавише мыши. При этом каждый второй щелчок меняет цвет фона.
10. Создать фрейм с изображением окружности. Длина дуги окружности изменяется нажатием клавиш от 1 до 9.