Конспект лекций по Java. Занятие 25

В.Фесюнов

1. Технология JavaBeans

1.1. Что такое JavaBeans

В исходной документации по JavaBeans от Sun определено: "Целью технологии JavaBeans является определение модели программных компонент такой, что фирмы-разработчики (*third party firms*) могут создавать и устанавливать Java-компоненты, которые могут быть скомпонованы конечными пользователями в законченные приложения".

Т.е. здесь речь идет о компонентном программировании и JavaBean — это технология создания и использования программных компонент (обычно визуальных, хотя не обязательно). В JavaBean программные компоненты, которые являются как бы кирпичиками программы, называются **Beans** (в переводе — бобы). Мы будем далее их именовать **бинами** .

В компонентном программировании подразумевается наличие не только самих компонент, но и некоторой визуальной среды разработки, позволяющей в диалоге строить программу из этих компонент. Причем, результат процесса сразу виден на экране. Технология JavaBean также неявно подразумевает наличие такой среды, но никоим образом не определяет ее. Соответствующие диалоговые среды разработки есть и, наверное, еще будут созданы новые. Эти среды отличаются друг от друга, иногда значительно, но все опираются на JavaBean, который является в этом смысле некоторым стандартом.

По адресу http://jsp2.java.sun.com/products/javabeans/software/bdk_download.html можно загрузить **Bean Development Kit** (**BDK**), основу которого составляет **BeanBox** , но это скорее демонстрационное средство, а не инструмент для программирования.

В процессе компонентного программирования можно выделить три группы действующих лиц, или три роли. Во-первых, это конечный пользователь, т.е. прикладной программист, который в визуальной среде собирает программу из отдельных компонент. Во-вторых, это разработчик готовых компонент. И, в-третьих, это разработчик визуальных сред компоновки программ.

Соответственно, тут возможны варианты. Можно, например, создать среду и технологию построения компонент под эту среду. Именно так поступила MicroSoft, создав Visual Basic и технологии OLE и ActiveX. А можно сделать универсальную технологию, которая позволяла бы не только создавать компоненты, но и визуальные среды, использующие эти компоненты. JavaBean создана в расчете именно на этот вариант.

Кроме того, нужно учитывать, что универсальная технология, претендующая на стандарт, должна учитывать интересы всех указанных групп действующих лиц. Она должна иметь средства, позволяющие разработчикам визуальных сред подключать различные компоненты в палитру доступных компонент, что вызывает необходимость наличия средств анализа готовых компонент. Она должна иметь правила разработки компонент, с тем, чтобы разработанная компонента могла быть интегрирована в визуальную среду. И, наконец, она должна определять средства связи компонент, которые прикладной программист использует для объединения готовых компонент в законченное приложение.

Три указанные роли являются, конечно, некоторым идеалом и в реальности эти роли зачастую пересекаются. Так, фирмы разработчики визуальных сред (таких как, JBuilder, Semantec Cafe, VisualJ и др.) включают в состав своих продуктов разработанные ими библиотеки, содержащие бины; разработчики прикладного ПО в процессе разработки не только используют существующие бины, но и создают свои.

1.2. Что такое Bean

В документации от Sun бин определяется так: "A Java Bean is a reusable software component that can be manipulated visually in a builder tool." ("Java Bean это многократно используемый программный компонент, которым можно манипулировать визуально в (визуальных) средах разработки").

В простейшем случае бин — это отдельный класс, представляющий определенную компоненту. В более сложных случаях — это набор взаимосвязанных классов, каждый из которых играет определенную роль. Так многие классы стандартной библиотеки Java являются бинами, например, `JLabel, JTextField` и др.

Основной класс бина должен удовлетворять одному требованию — он должен иметь конструктор по умолчанию (default constructor). Это требование естественно. Предполагается, что визуальная среда будет создавать экземпляры бинов и использовать для этого конструкторы по умолчанию. Есть и другие требования к бинам. Мы их рассмотрим далее.

Бины могут быть совершенно разными как по размерам, сложности, так и по области применения. Каждый конкретный бин может поддерживать ту или иную степень функциональности, но типичные универсальные возможности, которые обеспечивает бин следующие.

- Поддерживает "интроспекцию" (introspection), что позволяет средам разработки анализировать из чего состоит и как работает данный бин.
- Обеспечивает настраиваемость (customization), т.е. возможность изменять внешний вид (положение, размеры и т.п.) и поведение данного бина.
- Обеспечивает поддержку "событий" (events) как средства связи данного бина с программой и другими бинами.
- Обеспечивает поддержку свойств или атрибутов (properties), которые используются, в частности, для настройки (например, ширина, высота, количество каких-либо составных подкомпонент и т.п.).
- Поддерживает "сохраняемость" (persistence). Это необходимо для того, чтобы после настройки конкретного бина в некоторой визуальной среде разработки была возможность сохранить параметры настройки, а потом их восстановить.

Рассмотрим эти возможности подробнее. И начнем с конца, с persistence ("сохраняемости"), как с самого простого.

Это свойство обеспечивается выполнением следующего требования. Каждый бин в заголовке описания класса должен содержать " `implements java.io.Serializable` ", т.е. бины должны быть сериализуемыми. Предполагается, что визуальная среда при сохранении скомпонованного приложения дополнительно сохраняет настройки компонент, сделанные пользователем в процессе

разработки приложения и делает она это путем сериализации бина, например, в некоторый файл. При повторном входе в среду разработки и загрузке приложения эти настройки восстанавливаются. Для этого среда разработки просто десериализует бины из файла.

1.3. Свойства бинов (Bean properties)

Обычно каждый бин имеет свойства, которые определяют, как он будет работать и/или как он будет выглядеть. Эти свойства являются `private` или `protected` полями класса бина, которые доступны для выборки и/или модификации через специальные `public` методы. Другими словами бин обеспечивает доступ к своим свойствам через `public` методы 'get...' и 'set...'. Эти методы называют аксессорами (accessor) или, жаргонно, getters и setters и имеют определенные правила построения. 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;
}
```

Такой подход соответствует общим принципам объектно-ориентированного программирования, когда внутренние поля класса недоступны непосредственно и могут быть извлечены/изменены только посредством вызова методов класса.

Рассмотрим, к примеру, `JLabel` по документации. 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 вывод, что `JLabel` является бином, имеющим атрибут (свойство) *text* . Кроме того, `JLabel` имеет и другие пары get-/set-методов, т.е. имеет и другие атрибуты.

Атрибуты бинов могут быть как элементарных типов (`int, long` и т.п.), так и стандартных типов Java (например, `String`), а также пользовательских типов (например, `MyType` , где `MyType` — класс, определенный пользователем).

- Для свойств типа `boolean` вместо get-метода используется 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)
```

1.3.1. Правила построение методов доступа к атрибутам (аксессоров)

Аксессоры строятся по следующим правилам

- public void set < Property_name> (< Property_type> value);
- public < Property_type> get < Property_name>();
- public boolean is < Property_name>();

Эти правила относятся к простым свойствам. Кроме того, свойства могут быть индексированными или, другими словами, атрибут бина может быть массивом.

Для индексированных свойств выработаны следующие правила.

Они должны быть описаны как поля-массивы, например,

private String[] messages;

и должны быть такие методы

- public < Property_type> get < Property_name>(int index);
- public void set < Property_name> (int index, < Property_type> value);
- public < Property_type>[] get < Property_name>();
- public void set < Property_name> (< Property_type>[] 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);

1.4. Bean-методы

Кроме аксессоров, бин может иметь любое количество других методов, как обычный класс Java.

1.5. Интроспекция бинов при помощи reflection API

Описанных выше правил достаточно для осуществления простейшей интроспекции бинов с использованием reflection API. Вспомним возможности интроспекции, рассмотренные нами ранее.

Для использования бина визуальная среда должна знать полное имя класса бина. По полному имени класса можно статическим методом `forName` класса `Class` получить объект класса `Class` для данного бина. И далее, используя возможности класса `Class` , получить всю необходимую информацию по данному методу.

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

1.6. Связанные свойства (bound properties) и события

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

События и событийную модель Java мы рассматривали на 20-м занятии. 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. Модификации в структуре и алгоритмах работы двух этих объектов очень редко влияют на эту связь.

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

Приведем реальный пример. Рассмотрим текстовый редактор, позволяющий редактировать один файл. Такой пример мы уже рассматривали (Dlg5.java, занятие 17) и реализовали текущий редактируемый файл просто как поле типа `File` с именем `currentFile` . Изначально переменная `currentFile` не установлена. Она устанавливается при сохранении или чтении файла.

Теперь, если мы захотим добавить в наше приложение вывод какой-то информации по данному файлу, например, вывод имени файла где-нибудь внизу или вверху экрана, нам придется внимательно отследить, где в программе может изменятся это поле и вставить в этих точках обновление имени файла на экране. В данном приложении это вполне приемлемо, так как оно является небольшим и обозримым, но в более сложных случаях это может вызвать проблемы.

Альтернативный вариант состоит в построении бина со связанным свойством `currentFile` . Тогда при любой модификации этого свойства будет генерироваться событие. После этого мы можем создавать любое количество объектов, интересующихся значением свойства `currentFile` . Нам не придется при этом вносить какие-либо изменения в разработанный уже бин. Просто каждый новый такой объект должен зарегистрировать себя как слушателя данного бина. Для этого он должен иметь метод `propertyChange` , а в нем должен быть код, реагирующий на изменение свойства `currentFile` .

1.6.1. Создание и использование связанного свойства

Разберемся практически, как создавать и использовать связанные свойства.

Начнем с события, которое должно быть сгенерировано 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:

```
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.