Тема: JavaBeans Components I

План занятия:

1. Основы компонентного подхода к разработке ПО
2. Компонентная модель JavaBeans
3. Основы работы с JavaBeans
    a. Основные соглашения («шаблон»)
    b. Упаковка компонент в jar архив
4. Поддержка свойств (properties)
    a. Простые свойства (simple properties)
    b. Индексируемые свойства (indexed properties)
    c. Поддержка изменения свойств
        i. Связанные свойства (bound properties)
        ii. Свойства с ограничениями (constrained properties)
5. Поддержка событий (events)
    a. Модель обработки событий
    b. Шаблон Observer
    c. Создание своего события

Литература

1. Englander R. *Developing JAVA Beans*: - O'Reilly, 1997 – 231 p.
2. Wang A. J., Qian K. *Component-oriented programming*: - John Wiley & Sons. Inc., 2005.–334p.
3. *Trail: JavaBeans* (TM): https://docs.oracle.com/javase/tutorial/javabeans/
4. Кей Хорстманн, Гари Корнелл «*Java. Библиотека профессионала. Том 2*»
5. Патрик Нимейер, Дэниэл Леук. *Программирование на Java* (Исчерпывающее руководство для профессионалов). М.: Эксмо, 2014
6. Эллиот Расти Гарольд. *JavaBeans*. Издательство: Лори, 1999 - 327 с.
7. Роберт Орфали, Дан Харки. *Java и CORBA в приложениях клиент-сервер*. Издательство: Лори, 2000 - 734 с.

Основы компонентного подхода к разработке ПО

На данном занятии мы познакомимся с таким понятим, как компоненты *Java Beans*. В то время, когда язык и платформа *Java* начинала свое развитие, в обычной инженерии был очень популярен компонентный подход. Так, например, инженеры-разработчики оборудования вычислительных систем использовали различные готовые крупные компоненты, из которых относительно просто можно построить сложную вычислительную систему. Компоненты можно разделить по «степени крупности» (интеграции). Так,

можно выделить «атомарные» стандартные компоненты, такие как резисторы, конденсаторы транзисторы, катушки индуктивности и т.д. На следующем уровне можно выделить интегральные схемы, которые с одной стороны состоят из более простых компонентов, а с другой сами являются компонентами и предлагают еще больше функциональных возможностей. Дальше можно выделить более крупные блоки, состоящие из большого количества интегральных микросхем, которые предоставляют еще большую функциональность и непосредственно используются для построения вычислительных систем. При этом каждый из этих компонентов подчиняется определенным стандартом и их можно использовать многократно. Их не нужно компоновать заново всякий раз, когда требуется создать новую систему. Одни и те же компоненты можно использовать для создания разнотипных систем. Все это возможно потому, что поведение подобных компонент заранее известно и хорошо документировано.

В отрасли разработки программного обеспечения предпринимались попытки воспользоваться преимуществами такого многократного использования компонентов и их способностью к взаимодействию. Для этого необходимо было разработать такую архитектуру компонентов, которая позволила бы составлять программы из стандартных блоков, в том числе и предлагаемых сторонними производителями.

Был предложен компонентно-ориентированный поход к разработке программного обеспечения. Вообще-то считается, что данный подход появился в конце 1980-х годов, когда Н. Вирт разработал язык программирования «Оберон» и для него предложил паттерн написания блоков. Данный паттерн был сформирован при изучении проблемы «хрупких» базовых классов, возникающей при построении объёмной иерархии классов[1]. Паттерн заключался в том, что компонент компилируется отдельно от других, а на стадии выполнения — необходимые компоненты подключаются динамически. Компонентно-ориентированный подход может применяться во многих языках программирования с помощью стандартных конструкций (таких как: классы, интерфейсы, пакеты, модули). Компонентно-ориентированное программирование в применении к объектно-ориентированным языкам программирования подразумевает набор ограничений, накладываемых на объектные механизмы, направленные, прежде всего, на повышение надежности больших программных комплексов. Компонентно-ориентированный подход базируется на понятии компонента –

[1] Проблема хрупкого базового класса заключается в том, что малейшие правки в деталях реализации базового класса могут привнести ошибку в производные классы. В худшем случае это приводит к тому, что любая успешная модификация базового класса требует предварительного изучения всего дерева наследования, и зачастую невозможна (без создания ошибок) даже в этом случае. Проблема хрупкого базового класса сильно снижает ценность наследования. В общем случае проблема не решаема, и является одним из существенных недостатков ООП.

модульного, замкнутого, саморазвертывающегося объекта с качественно определенной функциональностью, который может быть объединен с другими компонентами посредством его интерфейса.

Наверное, наиболее явно это подход можно показать на примере таких сред разработки, как *Delphi* или *Visual Basic*: есть набор готовых компонент и возможность строить приложения из этих готовых визуальных блоков. От программиста требуется выбрать эти компоненты и настроить их на совместное использование. Кроме того стандартных компонент, разработчик может выбрать компонент стороннего производителя, разобраться в его функциях и внедрить в свое приложение. А после выхода новой версии компонента может относительно просто внедрить его функциональные возможности в уже существующий прикладной код.

В *Java* именно такую архитектуру и предлагают компоненты *Java Beans*. Когда-то это было новым, инновационным подходом, а теперь в какой-то мере стало стандартом. На этих двух лекциях мы разберемся с особенностями такого подхода.

Компонентная модель JavaBeans

JavaBeans - это компонентная архитектура для *Java*. Это набор правил для написания программных элементов с возможностью многократного использования, которые стандартным образом могут быть связаны друг с другом в режиме «подключи и работай» для создания больших и сложных приложений. Соблюдение этих правил позволяет также использовать стандартные, готовые инструменты разработки с поддержкой *JavaBeans* (например, все стандартные IDE), которые могут автоматически распознавать функции этих компонентов. В некоторых интегрированных средах разработки (IDE) даже можно создавать части приложений, просто подключая готовые компоненты Java. Эти компоненты можно создавать самостоятельно, а также приобретать в готовом виде у сторонних производителей.

JavaBeans — классы в языке *Java*, написанные по определённым правилам. Они используются для объединения нескольких объектов в один для удобной работы и взаимодействия. Спецификация определяет *JavaBeans* как повторно используемые программные компоненты, которыми можно управлять, используя графические конструкторы и средства *IDE*. *JavaBeans* обеспечивают основу для многократно используемых, встраиваемых и модульных компонентов *ПО*. Компоненты *JavaBeans* могут принимать различные формы, но наиболее часто они применяются в элементах графического пользовательского интерфейса. Одна из целей создания *JavaBeans* — взаимодействие с похожими компонентными структурами. Например, Windows-программа, при наличии соответствующего моста или

объекта-обёртки, может использовать компонент *JavaBeans* так, будто бы он является компонентом *COM* или *ActiveX*.

В исходной документации по *JavaBeans* указано: «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». Таким образом, здесь речь идет о компонентном программировании и *JavaBeans* — это технология создания и использования программных компонент (обычно визуальных, хотя не обязательно). В *JavaBeans* программные компоненты, которые являются как бы кирпичиками программы, называются *Beans* (англ. *bean* — фасоль, кофейное зерно). В литературе такие компоненты часто называют *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 *JavaBeans* также неявно подразумевает наличие такой среды, но никоим образом не определяет ее. Соответствующие диалоговые среды разработки есть и разрабатываются новые. Эти среды отличаются друг от друга, иногда значительно, но все опираются на *JavaBeans*, который является в этом смысле некоторым стандартом.

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.

В технологии *JavaBeans* был сделан упор на универсальность, что позволило не только создавать компоненты, но и визуальные среды, использующие эти компоненты. Была предпринята попытка должным образом учесть интересы всех указанных групп действующих лиц. В технологии есть средства, позволяющие разработчикам визуальных сред подключать различные компоненты в палитру доступных компонент, что вызывает необходимость иметь средства анализа готовых компонент. Кроме того, должны быть правила разработки компонент, с тем, чтобы разработанная компонента могла быть интегрирована в визуальную среду. И, наконец, нужно определить 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, все фирмы разработчики визуальных сред разработки 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.

Основы работы с JavaBeans

В документации еще 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 все эти возможности подробнее.

Начнем с самого простого, с *persistence* ("*сохраняемости*"). Это свойство обеспечивается выполнением следующего требования. Каждый бин в заголовке описания класса должен содержать "*implements java.io.Serializable*" (или "*java.io.Externalizable*"), 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.

Основные соглашения

Чтобы класс мог работать как *bean*, он должен соответствовать определённым соглашениям об именах методов, конструкторе и поведении. Эти соглашения дают возможность создания инструментов, которые могут использовать, замещать и соединять *JavaBeans*.

Правила описания гласят:

- Класс должен иметь конструктор без параметров, с модификатором доступа *public*. Такой конструктор позволяет инструментам создать объект без дополнительных сложностей с параметрами.
- Свойства класса должны быть доступны через *get*, *set* и другие методы (так называемые методы доступа), которые должны подчиняться стандартному соглашению об именах. Это легко позволяет инструментам автоматически определять и обновлять содержание *bean*'ов. Многие инструменты даже имеют специализированные редакторы для различных типов свойств.
- Класс должен быть сериализуем. Это даёт возможность надёжно сохранять, хранить и восстанавливать состояние *bean* независимым от платформы и виртуальной машины способом.
- Класс должен иметь переопределенные методы *equals()*, *hashCode()* и *toString()*.

Так как требования в основном изложены в виде соглашения, а не интерфейса, некоторые разработчики рассматривают *JavaBeans*, как *Plain Old Java Objects*, которые следуют определённым правилам именования.

Кстати, что бы класс считался бином не обязательно выполнять все требования. В ряде случаев допускается не переопределять *equals()* и *hashCode()*. Кроме того, есть еще некоторые дополнительные возможности, предоставляемые технологией, часть из которых мы и рассмотрим.

Рассмотрим пример создания невизуального компонента в соответствии с указанными правилами (пример из проекта, созданного в *IntelliJ IDEA*):

```
package test1;

public class PersonBean implements java.io.Serializable {

    private String name;
    private boolean inUkraine;

    // Методы геттеры (get) и сеттеры (set)
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public boolean isInUkraine() {
        return inUkraine;
    }

    public void setInUkraine(boolean inUkraine) {
        this.inUkraine = inUkraine;
    }

    //Переопределенные методы equals() и hashCode()
    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (o == null || getClass() != o.getClass()) {
            return false;
        }
        PersonBean that = (PersonBean) o;
        if (inUkraine != that.inUkraine) {
            return false;
        }
        return !(name != null ? !name.equals(that.name) : that.name != null);
    }

    @Override
    public int hashCode() {
        int result = name != null ? name.hashCode() : 0;
        result = 31 * result + (inUkraine ? 1 : 0);
        return result;
    }

    //Переопределенный метод toString()
    @Override

    public String toString() {
        return "PersonBean{" +
                "name='" + name + '\'' +
                ", inUkraine=" + inUkraine +
                '}';
```

```
    }
}
```

Рассмотрим пример использования указанного компонента:

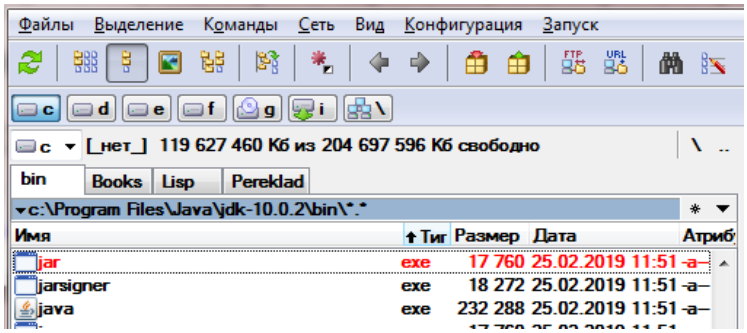
```
package test1;

import java.io.*;

public class TestPerson {
    public static void main(String[] args) {
        File file = new File("comp.ser");
        if (file.exists()) {
            System.out.println("Read components...");
            PersonBean person1 = null;
            PersonBean person2 = null;
            try {
                ObjectInputStream ois = new ObjectInputStream(
                                                new FileInputStream(file));
                person1 = (PersonBean) ois.readObject();
                person2 = (PersonBean) ois.readObject();
                ois.close();
            } catch (IOException | ClassNotFoundException e) {
                e.printStackTrace();
            }
            System.out.println("\t"+person1+"\n\t"+person2);
        } else {
            System.out.println("Create Components:");
            PersonBean person1 = new PersonBean();
            person1.setName("Bob");
            person1.setInUkraine(true);
            PersonBean person2 = new PersonBean();
            person2.setName("Bil");
            person2.setInUkraine(false);
            System.out.println("\t"+person1+"\n\t"+person2);

            System.out.println("Save components...");
            try {
                ObjectOutputStream oos = new ObjectOutputStream(
                                                new FileOutputStream(file));
                oos.writeObject(person1);
                oos.writeObject(person2);
                oos.flush();
                oos.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

}
```

Отметим, что применение стандартных компонент *JavaBeans* не ограничивается клиентскими программами. Такие технологии, как

Java Server Faces (*JSF*) и *Java Server Pages* (*JSP*), основываются на компонентной модели *JavaBeans*.

Упаковка компонент в jar архив

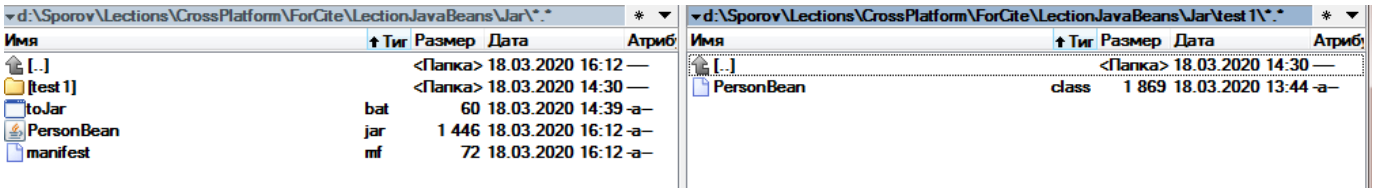
Для того чтобы подготовить компонент к использованию в контейнере, нужно упаковать его вместе с необходимыми ему файлами и ресурсами в файл JAR (https://docs.oracle.com/javase/tutorial/deployment/jar/index.html). Так как один компонент может иметь много вспомогательных файлов и, кроме того, файл JAR может содержать несколько компонентов, то нужно указать манифест, в котором должно быть описано, какие элементы файла являются компонентами. Файлы JAR создаются при помощи утилиты jar. Она расположена в директории, где хранятся исполняемые файлы *JDK*.



Если *JDK* установлен правильно, с указанием всех необходимых системных переменных (напр. PATH), то вызывать утилиту jar можно просто по имени. Иначе нужно или добавить соответствующее значение в переменную PATH, или вызывать утилиту jar по точному указанию месторасположения. Формат вызова утилиты:

jar *options file*

Рассмотрим ее применение на нашем примере. Предположим, что у нас уже есть откомпилированные файлы классов. Как и положено, они сохранены в директории с именем пакета (test1), а наш текущий каталог – это каталог, в котором находится папка test1.



Текущий рабочий каталог указан на левом рисунке. Как уже было сказано, в *jar* архив должен быть включен *файл манифеста* (*manifest file*), в котором будут указано, какие файлы классов в архиве являются компонентами *JavaBeans* и должны быть включены в интегрированную среду разработки. Создадим обычный текстовый файл **manifest.mf** в текущем каталоге и заполним его следующим содержимым:

```
Manifest-Version: 1.0

Name: test1/PersonBean.class
Java-Bean: True
```

В файле манифеста для компонент *JavaBeans* сначала указывается заголовок, который отделен двоеточием от его значения. Здесь показываем, что манифест соответствует спецификациям манифеста для версии 1.0. Далее указываем, какие компоненты являются компонентами *JavaBeans*. Нужно обратить внимание, что между заголовком манифеста с определением версии и названием компонента *JavaBeans* должна находиться пустая строка. Нужно обратить внимание на то, что для любой операционной системы в файле манифеста для разделения каталогов используются прямые слэши. Кроме того, не рекомендуется включать компоненты *JavaBeans* в каталог по умолчанию, так как в некоторых средах могут возникнуть проблемы при загрузке компонентов *JavaBeans* из пакета по умолчанию.

Если компонент *JavaBeans* содержит несколько файлов классов, то в файле манифеста нужно указать только файлы классов для тех компонентов *JavaBeans*, которые нужно отразить на панели инструментов. Например, в случае если бы у нас было два компонента *JavaBeans*, которые нужно было включить в один *jar* файл, то формат файла манифеста был бы таким:

```
Manifest-Version: 1.0

Name: test1/PersonBean.class
Java-Bean: True

Name: test1/NextJavaBean.class
Java-Bean: True
```

После последней строки в файле манифеста должен быть перевод строки. Дело в том, что некоторые интегрированные среды разработки предъявляют

чрезвычайно высокие требования к формату файла манифеста, например, в концах строк недопустимы пробелы, после определения версии и между записями компонентов *JavaBeans* должна быть указана пустая строка и вслед за последней строкой файла также должна размещаться еще одна пустая строка.

После того, как будет сформирован файл манифеста можно запустить архиватор jar. Для получения короткой справки по программе можно выполнить в командном окне команду:

```
jar --help
```

Выполняем команду по сборке архива:

```
jar.exe cvmf manifest.mf PersonBean.jar -C . .\test1\*.class
```

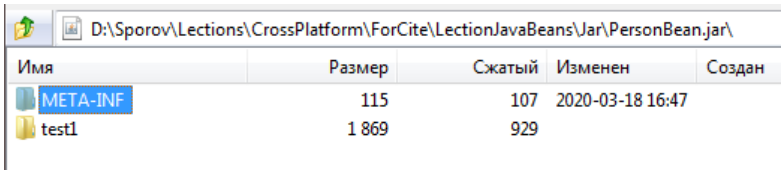
Кратко рассмотрим указанные опции:

- c – создается новый архивный файл;
- v – во время работы утилиты должен быть предоставлен подробный отчет о ходе ее выполнения;
- m – место имени файла манифеста в списке файлов;
- f – место имени архива в списке файлов.

Здесь важен порядок следования ключей. С каком порядке стоят опции m и f, в таком же порядке должны стоять имена manifest- и jar- файлов, соответственно. В нашей команде manifest.mf – это имя файла манифеста, в котором мы указали компонент. В принципе, этот файл может называться как угодно, на указанное – это фиксировано стандартом имя файла манифеста внутри jar-архива. Имя PersonBean.jar – это имя создаваемого jar архива. Конструкция «-C . .\test1*.class» означает: изменить рабочую директорию на текущий каталог, и в нем из директории test1 включить в архив все файлы с расширением имени class. По сути, все указанные команды должны выполняться из корня структуры классов нашего проекта.

```
d:\Sporov\Lections\CrossPlatform\ForCite\LectionJavaBeans\Jar>jar.exe cvmf manif
est.mf PersonBean.jar -C . .\test1\*.class
added manifest
adding: test1/PersonBean.class(in = 1869) (out= 929)(deflated 50%)
```

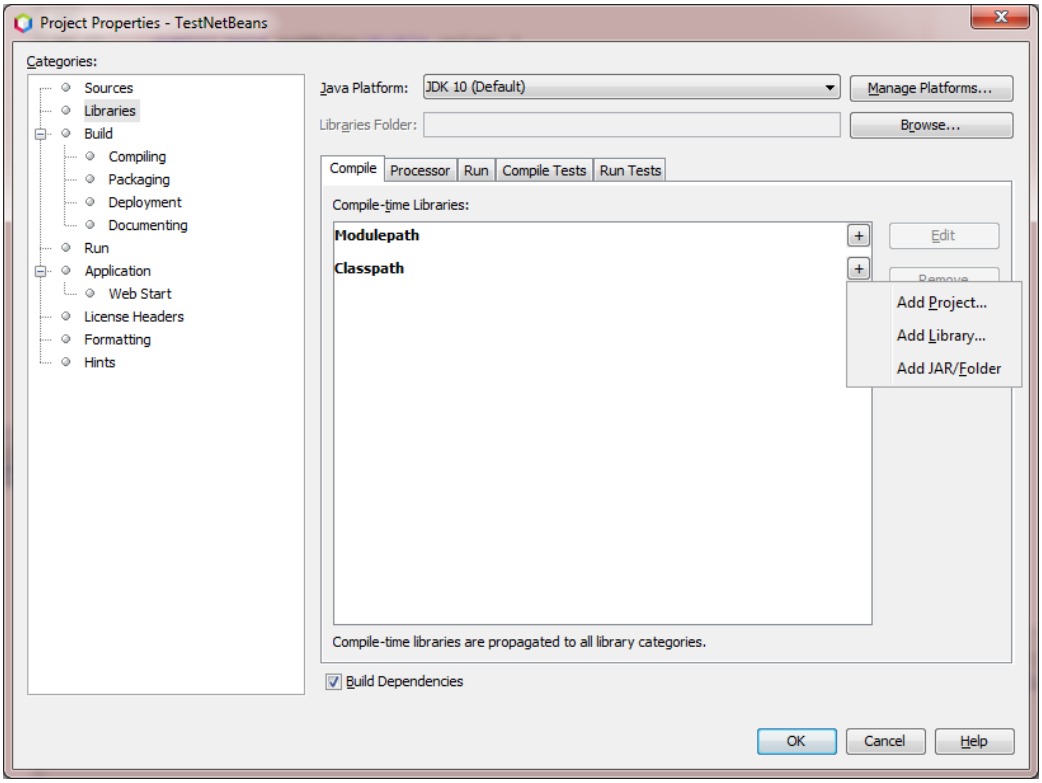
В результате в текущей директории будет создан файл PersonBean.jar, структуру которого можно исследовать с помощью обычного архиватора:



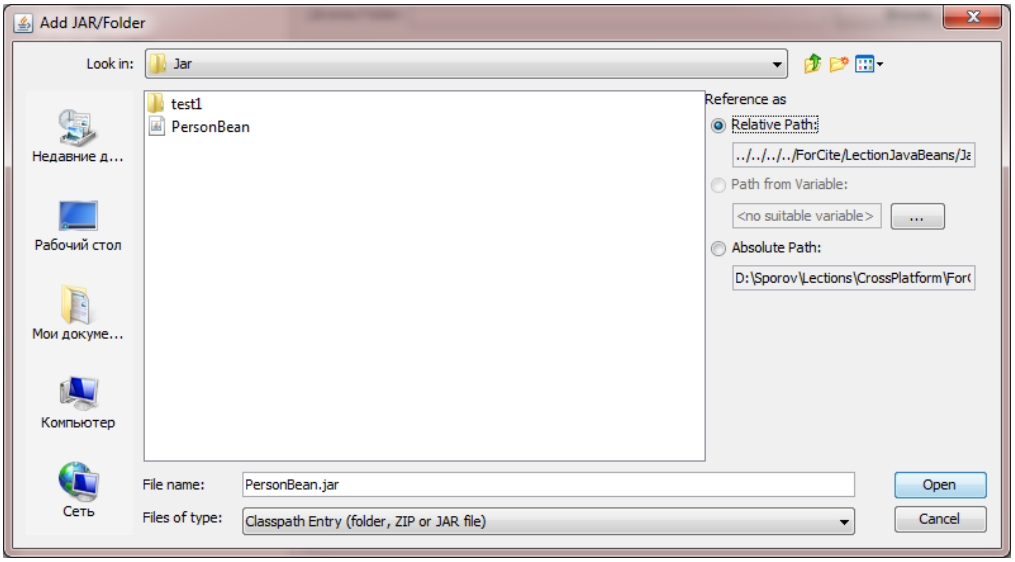
В каталоге META-INF располагается файл манифеста MANIFEST.MF, а в каталоге test1 – class файлы.

После этого, сформированный jar архив можно передавать для использования. Если бы у нас в архиве хранились визуальные компоненты, то ими можно было бы воспользоваться и настраивать их в визуальном редакторе интерфейсов (это посмотрим чуть позже). А пока можно просто подключить этот файл как библиотеку.

Протестируем этот компонент. Создадим стандартным образом новый проект с помощью *Apache Net Beans IDE*. В проекте создадим новый пакет (test). К проекту подключим созданный компонент в jar файле. Для этого откроем диалоговое окно со свойствами проекта (*File | Project Properties* (*ProjectName*)):



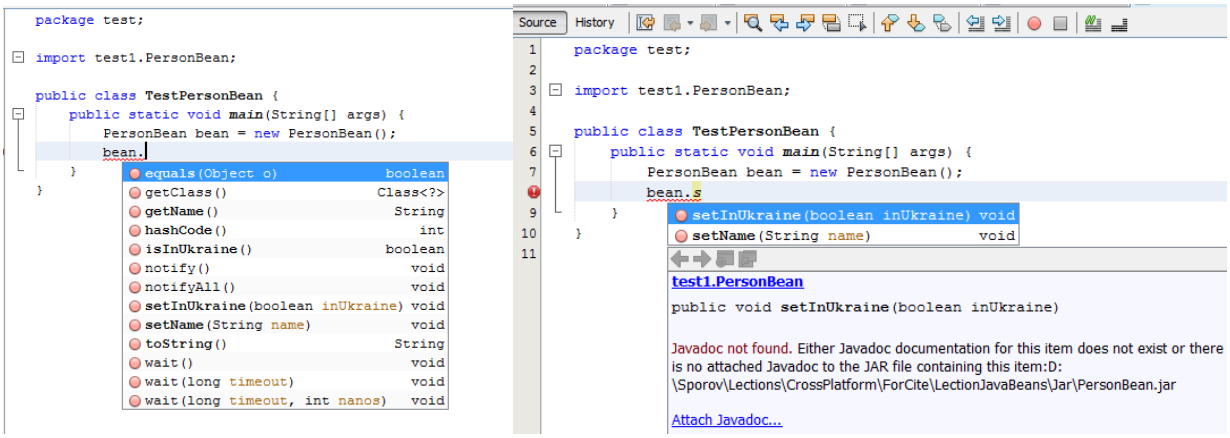
В разделе библиотеки добавим jar файл (пункт *Add JAR/Folder*).



Указываем месторасположение архива (можно указать, чтобы система сохранила в настройках относительный путь к архиву). И после этого приступаем к созданию кода программы:

```
package test;

import test1.PersonBean;

public class TestPersonBean {
    public static void main(String[] args) {
        PersonBean bean = new PersonBean();
        bean.setName("Bill");
        bean.setInUkraine(true);
        System.out.println(bean);
    }
}
```

Система автоматически импортирует нужный класс и нормально будет нормально выдавать подсказки:



Удобство использования компонент будет особенно заметно для сложных визуальных компонент с дополнительными возможностями, которые мы рассмотрим на этом и следующем занятиях.

Поддержка свойств

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 чтения и/или модификации через специальные public методы. Создатели спецификации языка *Java* не стали включать специальные ключевые слова для поддержки визуального программирования, поэтому анализ компонентов *JavaBeans* и определение их свойств и событий осуществляется другими способами. Если автор компонента следует стандартным соглашениям об именовании, то интегрированная среда разработки может использовать для определения нужных свойств и событий компонентов *JavaBeans* механизм отражения (*Reflection API*). Кроме того, можно создать специальный информационный класс компонента, который содержит описание компонента, необходимое среде разработки для организации работы с ним. Эту возможность рассмотрим на следующем занятии, а пока рассмотрим более простой способ – соглашения об именах. В документах по технологии *JavaBeans* стандартные соглашения об именах называют *шаблонами проектирования* (*design patterns*), но они не имеют ничего общего с теми шаблонами проектирования, которые используются в объектно-ориентированном программировании.

Значение свойств компонентов устанавливается методом записи, а получается оно методом получения. 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.

Простые свойства

Простыми свойствами называют те свойства, которые содержат только одно значение, например строку или число. Простые свойства 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). Именно такие свойства были указаны в нашем примере бина. Простое свойство легко программируется с помощью соглашения об именах.

Рассмотрим правила построение аксессоров для случая простых свойств:

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

Свойство может быть определено только для чтения, если для него предусмотрен get метод, но отсутствует set метод. Следует отметить, что аксессоры могут выполнять произвольные действия, а не только возвращать или задавать значение поля.

Создавая методы доступа следует обращать внимание на использование прописных букв в их именах. Так, одно из рассматриваемых в примере свойств должно иметь имя name (буква n — строчная), а в методах get и set указывается прописная буква N (getName(), setName()). Для определения имени свойства анализатор компонентов интегральной среды разработки использует процедуру приведения к нижнему регистру (*decapitalization*). При этом к нижнему регистру преобразуется первый символ, следующий за префиксом get или set. Благодаря такому подходу формируются имена свойств и методов, привычные для большинства программистов.

Если же прописными являются две первые буквы свойства (такому свойству, например, соответствует метод getURL()), то регистр первой буквы не изменяется.

Для свойств типа boolean (inUkraine в нашем примере) вместо get-метода рекомендуется использовать is-метод. Для доступа к этому свойству применялись методы

```
public boolean isInUkraine();
public void setInUkraine(boolean b);
```

Таким образом, утверждение о том, что данный бин имеет свойство name типа PropertyType, доступное для чтения и записи означает, что у этого бина

- есть поле private Type name;
- есть get-метод:

```
public PropertyType getName() {
  return name;
}
```

- есть set-метод

```
public void setName(PropertyType 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.

Индексируемые свойства

Кроме простых свойств бины поддерживают индексированные свойства. С помощью таких свойств можно работать со сложными полями, например с массивами. В случае использования индексированного свойства необходимо предоставить две пары методов get и set: одну для массива целиком и одну для его отдельных элементов.

Для индексированных свойств выработаны следующие правила. Они должны быть описаны как поля-массивы, например,

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

и должны быть определены 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);
```

Следует помнить, что для расширения массива нельзя использовать метод set<Property_name>(int index, <Property_type> value); для этого необходимо создать новый массив и передать его методу set<Property_name> (<Property_type>[] value).

Таким образом, свойство список фамилий может иметь такой вид. В классе бина должно быть поле

private String[] names;

конструктор по умолчанию для инициализации пол и методы

public String getNames(int index);
public void setNames(int index, String name);
public String[] getNames();
public void setNames(String[] names);

Рассмотрим пример бина с простыми и индексируемыми свойствами:

```
package example1;

import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;
import java.util.Arrays;

public class BankAccountBean implements java.io.Externalizable {
    private String user;
    private double money;
    private double withdraw;
    private double[] limits;

    public BankAccountBean() {
        this.user = "NoName";
        this.money = 0.0;
        this.withdraw = 1000;
        this.limits = new double[2];
        limits[0] = 1000;
        limits[1] = 10000;
    }

    public String getUser() {
        return user;
    }

    public void setUser(String user) {
        this.user = user;
    }

    public double getMoney() {
        return money;
    }

    public void setMoney(double money) {
        this.money = money;
```

```
    }

    public double getWithdraw() {
        return withdraw;
    }

    public void setWithdraw(double withdraw) {
        this.withdraw = withdraw;
    }

    public double[] getLimits() {
        return limits;
    }

    public void setLimits(double[] limits) {
        this.limits = limits;
    }

    public double getLimits(int i) {
        return this.limits[i];
    }

    public void setLimits(int i, double limit) {
        this.limits[i] = limit;
    }

    @Override
    public void writeExternal(ObjectOutput objectOutput) throws IOException {
        //objectOutput.writeObject(this.user);
        objectOutput.writeObject(new
StringBuilder(this.user).reverse().toString());
        objectOutput.writeDouble(this.money);
        objectOutput.writeDouble(this.withdraw);
        objectOutput.writeInt(this.limits.length);
        for(double lim : this.limits)
            objectOutput.writeDouble(lim);
    }

    @Override
    public void readExternal(ObjectInput objectInput) throws IOException,
ClassNotFoundException {
        //this.user = (String)objectInput.readObject();
        this.user = new StringBuilder((String)
objectInput.readObject()).reverse().toString();
        this.money = objectInput.readDouble();
        this.withdraw = objectInput.readDouble();
        int len = objectInput.readInt();
        this.limits = new double[len];
        for (int i = 0; i < len; i++)
            this.limits[i] = objectInput.readDouble();
    }

    @Override
    public String toString() {
        return "BankAccountBean{" +
                "user='" + user + '\'' +
                ", money=" + money +
                ", withdraw=" + withdraw +
                ", limits=" + Arrays.toString(limits) +
```

```
                '}';
    }
}
```

Приведем пример кода, в котором используется этот бин.

```
package example1;

import java.io.*;

/* Basic example with properties */
public class Main {
    public static void serialize(Object obj, String fileName) {
        try {
            FileOutputStream fos = new FileOutputStream(fileName);
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(obj);
            oos.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static Object deSrialize(String fileName) {
        Object res = null;
        try {
            FileInputStream fis = new FileInputStream(fileName);
            ObjectInputStream ois = new ObjectInputStream(fis);
            res = ois.readObject();
            ois.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        return  res;
    }

    public static void main(String[] args) {
        BankAccountBean bean1 = new BankAccountBean();
        System.out.println("Initial state: " + bean1);
        System.out.println("Bean customization...");
        bean1.setUser("Lohankin");
        bean1.setMoney(50000);
        bean1.setWithdraw(5000);
        bean1.setLimits(0,1000);
        bean1.setLimits(1, 80000);
        System.out.println("Final state: " + bean1);
        System.out.println("Bean serialize / deserialize...");
        Main.serialize(bean1, "bean.ser");
        BankAccountBean bean2 = (BankAccountBean) Main.deSrialize("bean.ser");
        System.out.println("Serializable version: " + bean2);
    }
```

```
}
```

Результат работа программы

```
Initial state: BankAccountBean{user='NoName', money=0.0, withdraw=1000.0,
limits=[1000.0, 10000.0]}
Bean customization...
Final state: BankAccountBean{user='Lohankin', money=50000.0, withdraw=5000.0,
limits=[1000.0, 80000.0]}
Bean serialize / deserialize...
Serializable version: BankAccountBean{user='Lohankin', money=50000.0,
withdraw=5000.0, limits=[1000.0, 80000.0]}
```

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

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 визуальная 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.

Поддержка изменения свойств

Еще одним важным аспектом технологии *JavaBeans* является возможность бинов взаимодействовать с другими объектами, в частности, с другими бинами. Так, в ряде случаев бывает удобно не только изменить значение свойства, но и, кроме этого, уведомить заинтересованные компоненты о произошедших изменениях. Для этого в технологии *JavaBeans* предусмотрены *связанные свойства* (*bounded properties*) и свойства с ограничениями (*constrained properties*). Рассмотрим эти возможности поподробнее.

Связанные свойства

Связанные свойства уведомляют заинтересованные компоненты об изменении значения. Когда значение связанного свойства меняется, генерируется событие и передается всем зарегистрированным слушателям посредством вызова метода propertyChange.

Для создания связанного свойства необходимо обеспечить выполнение двух действий:

1. При изменении значения связанного свойства компонент *JavaBeans* должен послать событие PropertyChangeEvent всем зарегистрированным обработчикам. Такое изменение может произойти при вызове set-метода или при выполнении пользователем какого-то действия, например при редактировании текста или выборе файла.
2. Для автоматической регистрации обработчиков событий в компонент *JavaBeans* следует включить два метода:

```
void addPropertyChangeListener(PropertyChangeListener listener)
void removePropertyChangeListener(PropertyChangeListener listener)
```

3. Кроме того рекомендуется, но не требуется предоставить метод:

```
PropertyChangeListener[] getPropertyChangeListeners()
```

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, доступную по адресу: https://docs.oracle.com/javase/9/docs/api/java/beans/PropertyChangeEvent.html).

Далее можно реализовать следующий план действий.

1. Чтобы упростить реализацию связанных свойств рекомендуется воспользоваться возможностями, предоставляемыми существующим классом java.beans.PropertyChangeSupport (см. документацию https://docs.oracle.com/javase/7/docs/api/java/beans/PropertyChangeSupp

ort.html). Это класс может самостоятельно управлять работой обработчиков событий. Чтобы воспользоваться возможностями этого класса, необходимо в добавить в бин поле экземпляра этого класса:

```
private final PropertyChangeSupport pcs =
                                        new PropertyChangeSupport(this);
```

2. Для регистрации/дерегистрации слушателя необходимо в бине реализовать два метода:

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

и

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

3. В set-методе связанного свойства необходимо добавить вызов метода класса java.beans.PropertyChangeSupport — firePropertyChange .

```
public void setValue(Тип newValue) {
    Тип oldValue = this.value;
    this.value = newValue;
    this.pcs.firePropertyChange("value", oldValue, newValue);
}
```

Для уведомлении об изменении индексированного свойства используется вызов:

```
this.pcs.fireIndexedPropertyChange("PropertyName", index, oldValue,
                                                        newValue);
```

4. В классе-слушателе необходимо реализовать интерфейс PropertyChangeListener, т.е. в заголовке класса записать "implements PropertyChangeListener", а в теле класса реализовать единственный метод интерфейса

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

Объект PropertyChangeEvent (https://docs.oracle.com/javase/7/docs/api/java/beans/PropertyChangeEvent.html) содержит имя свойства, его старое и новые значения, получаемые с помощью методов getPropertyName(), getNewValue(), getOldValue(). Если тип свойства не является ссылочным, то объекты значений этого свойства представляют собой экземпляры классов-оболочек.

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 создания объекта-слушателя, например,

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

где myBean — наш бин (объект-бин, а не класс).

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

Рассмотрим пример работы со связанными свойствами:

```
package lectest1;

import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;
import java.util.Arrays;

public class TempBean implements java.io.Serializable {
   private static final long serialVersionUID = 1L;

   private double temperature;
   private double[] delta;

   public TempBean() {
      this.temperature = 0;
      this.delta = new double[2];
      delta[0] = -2;
      delta[1] = 8;
   }

   public double getTemperature() {
      return temperature;
   }

   public void setTemperature(double temperature) {
      //this.temperature = temperature;
      double old = this.temperature;
      this.temperature = temperature;
      changeSupport.firePropertyChange("temperature", old, temperature);
   }

   public double[] getDelta() {
      return delta;
   }

   public void setDelta(double[] delta) {
      this.delta = delta;
   }

   public double getDelta(int i) {
      return delta[i];
   }

   public void setDelta(int i, double delta) {
      this.delta[i] = delta;
   }
```

```
    @Override
    public String toString() {
        return "TempBean [temperature=" + temperature + ", delta="
                + Arrays.toString(delta) + "]";
    }


    /**/
    private PropertyChangeSupport changeSupport = new PropertyChangeSupport(this);

    public void addPropertyListener(PropertyChangeListener listener) {
        changeSupport.addPropertyChangeListener(listener);
    }

    public void removePropertyChangeListener(PropertyChangeListener listener) {
        changeSupport.removePropertyChangeListener(listener);
    }
    /**/
}
```

```
package lectest1;

import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;

public class TempListener implements PropertyChangeListener {

    @Override
    public void propertyChange(PropertyChangeEvent evt) {
        System.out.println(evt.getPropertyName() + " changed from " +
        evt.getOldValue() + " to " + evt.getNewValue());
    }
}
```

```
package lectest1;

import java.beans.Beans;
import java.io.IOException;

public class Termometer {

    private TempBean bean;

    public Termometer() {
        this.bean = new TempBean();
        this.bean.addPropertyListener(new TempListener());
    }

    public void changeTemp() {
        for(int i = 0; i < 10; i++) {
            bean.setTemperature(i*0.5);
        }
    }

    public static void main(String[] args) throws ClassNotFoundException, IOException {
        new Termometer().changeTemp();
        /*
temperature changed from 0.0 to 0.5
temperature changed from 0.5 to 1.0
```

```
temperature changed from 1.0 to 1.5
temperature changed from 1.5 to 2.0
temperature changed from 2.0 to 2.5
temperature changed from 2.5 to 3.0
temperature changed from 3.0 to 3.5
temperature changed from 3.5 to 4.0
temperature changed from 4.0 to 4.5
        */
    }

}
```

Свойства с ограничениями

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. Таким образом, будет необходимо 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.

Для создания свойства с ограничениями необходимо реализовать в составе компонента два метода, предназначенные для управления объектами VetoableChangeListener (https://docs.oracle.com/javase/7/docs/api/java/beans/VetoableChangeListener.html).

```
public void addVetoableChangeListener(VetoableChangeListener listener);
public void removeVetoableChangeListener(
                                    VetoableChangeListener listener);
```

Как и в случае связанных свойств, в пакете java.beans существует вспомогательнй класс VetoableChangeSupport (https://docs.oracle.com/javase/7/docs/api/java/beans/VetoableChangeSupport.html), предназначенный для управления обработчиками событий изменения, которые могут быть запрещены. Ваш бин-компонент должен содержать экземпляр этого класса.

```
private final VetoableChangeSupport vcs =
```

```
                                        new VetoableChangeSupport(this);
```

Добавление и удаление обработчиков делегируются данному объекту.

```
public void addVetoableChangeListener(
                                        VetoableChangeListener listener) {
        this.vcs.addVetoableChangeListener(listener);
}

public void removeVetoableChangeListener(
                                        VetoableChangeListener listener) {
        this.vcs.removeVetoableChangeListener(listener);
}
```

Обновление значения свойства с ограничениями включает в себя три этапа.

1. Оповещение обработчиков изменений свойств с ограничениями о намерении изменить значение. (Для этого используется метод fireVetoableChange() объекта класса VetoableChangeSupport).
2. Если ни один из обработчиков не сгенерирует исключение PropertyVetoException, значение свойства обновляется.
3. Передача всем обработчикам подтверждения о том, что изменения выполнены.

Рассмотрим схематрический пример кода:

```
public void setValue(String newValue) throws PropertyVetoException {
        String oldValue = this.value;
        this.vcs.fireVetoableChange("value", oldValue, newValue);
// Если следующая строка кода получит управление,
// значит, запрета на изменение значения не было
        this.value = newValue;
    }
```

Рассмотрим пример работы со свойствами с ограничениями:

```
package lectest2;

import java.beans.PropertyVetoException;
import java.beans.VetoableChangeListener;
import java.beans.VetoableChangeSupport;
```

```
import java.util.Arrays;

public class TempBean {
   private static final long serialVersionUID = 1L;

   private double temperature;
   private double[] delta;

   public TempBean() {
      this.temperature = 0;
      this.delta = new double[2];
      delta[0] = -2;
      delta[1] = 8;
   }

   public double getTemperature() {
      return temperature;
   }

   public void setTemperature(double temperature) throws PropertyVetoException {
      try {
         double old = this.temperature;
         vetoChangeSupport.fireVetoableChange("temperature", old, temperature);
          this.temperature = temperature;
         System.out.println("The temperature was changed");
      } catch (PropertyVetoException e) {
            throw e;
      }
   }

   public double[] getDelta() {
      return delta;
   }

   public void setDelta(double[] delta) {
      this.delta = delta;
   }

   public double getDelta(int i) {
      return delta[i];
   }

   public void setDelta(int i, double delta) {
      this.delta[i] = delta;
   }

   @Override
   public String toString() {
      return "TempBean [temperature=" + temperature + ", delta="
            + Arrays.toString(delta) + "]";
   }


   /**/
   VetoableChangeSupport vetoChangeSupport = new VetoableChangeSupport(this);

   public synchronized void addVetoableChangeListener(VetoableChangeListener listener)
{
      vetoChangeSupport.addVetoableChangeListener(listener);
    }

   public synchronized void removeVetoableChangeListener(VetoableChangeListener
listener) {
      vetoChangeSupport.removeVetoableChangeListener(listener);
   }
```

```
    /**/
}
```

```
package lectest2;

import java.beans.PropertyChangeEvent;
import java.beans.PropertyVetoException;
import java.beans.VetoableChangeListener;

public class TempVetoableChangeListener implements VetoableChangeListener {

   private double threshold;

   public TempVetoableChangeListener() {
      this.threshold = 4.0;
   }

   public double getThreshold() {
      return threshold;
   }

   public void setThreshold(double threshold) {
      this.threshold = threshold;
   }

   @Override
   public void vetoableChange(PropertyChangeEvent evt)
         throws PropertyVetoException {
      System.out.println("Attempt to set " + evt.getPropertyName() + " from "
            + evt.getOldValue() + " to " + evt.getNewValue());
      boolean veto = (double) evt.getNewValue() > this.threshold;
      if (veto) {
         throw new PropertyVetoException("the reason for the veto", evt);
      }
   }
}
```

```
package lectest2;

import java.beans.PropertyVetoException;

public class Main {

   public static void main(String[] args) {
      TempBean bean = new TempBean();
       bean.addVetoableChangeListener(new TempVetoableChangeListener());
       //
       TempVetoableChangeListener lst = new TempVetoableChangeListener();
       lst.setThreshold(3.5);
       bean.addVetoableChangeListener(lst);
       //

       for(int i = 0; i < 12; i++) {
         try {
            bean.setTemperature(i*0.5);
         } catch (PropertyVetoException e) {
            System.out.println("\tThe temperature was NOT changed");
         }
      }
```

```
        System.out.println("Final Temperature: " + bean.getTemperature());
    }

}
/*
The temperature was changed
Attempt to set temperature from 0.0 to 0.5
Attempt to set temperature from 0.0 to 0.5
The temperature was changed
Attempt to set temperature from 0.5 to 1.0
Attempt to set temperature from 0.5 to 1.0
The temperature was changed
Attempt to set temperature from 1.0 to 1.5
Attempt to set temperature from 1.0 to 1.5
The temperature was changed
Attempt to set temperature from 1.5 to 2.0
Attempt to set temperature from 1.5 to 2.0
The temperature was changed
Attempt to set temperature from 2.0 to 2.5
Attempt to set temperature from 2.0 to 2.5
The temperature was changed
Attempt to set temperature from 2.5 to 3.0
Attempt to set temperature from 2.5 to 3.0
The temperature was changed
Attempt to set temperature from 3.0 to 3.5
Attempt to set temperature from 3.0 to 3.5
The temperature was changed
Attempt to set temperature from 3.5 to 4.0
Attempt to set temperature from 3.5 to 4.0
Attempt to set temperature from 4.0 to 3.5
   The temperature was NOT changed
Attempt to set temperature from 3.5 to 4.5
   The temperature was NOT changed
Attempt to set temperature from 3.5 to 5.0
   The temperature was NOT changed
Attempt to set temperature from 3.5 to 5.5
   The temperature was NOT changed
Final Temperature: 3.5
*/
```

Следует отметить, что до тех пор, пока все зарегистрированные обработчики не согласятся с предложенными изменениями, значение свойства остается прежним. Однако, ни один обработчик не может быть уверенным, что изменения, против которых он не возражает, действительно будут выполнены. Единственный способ убедиться в том, что значение свойства обновлено, — использовать обработчик события, соответствующего изменению значения свойства. Рассмотрим пример такой совместной работы:

```
package example2;

import java.beans.*;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;
import java.util.Arrays;

public class BankAccountBean implements java.io.Externalizable {
    private String user;
    private double money;
```

```
private double withdraw;
private double[] limits;

//////////////////////////////////////////////////////////////////////////
private PropertyChangeSupport changeSupport = new PropertyChangeSupport(this);

public void addPropertyChangeListener(PropertyChangeListener listener) {
    changeSupport.addPropertyChangeListener(listener);
}

public void removePropertyChangeListener(PropertyChangeListener listener) {
    changeSupport.removePropertyChangeListener(listener);
}
//////////////////////////////////////////////////////////////////////////
VetoableChangeSupport vetoChangeSupport = new VetoableChangeSupport(this);

public void addVetoableChangeListener(VetoableChangeListener listener) {
    vetoChangeSupport.addVetoableChangeListener(listener);
}

public void removeVetoableChangeListener(VetoableChangeListener listener) {
    vetoChangeSupport.removeVetoableChangeListener(listener);
}
//////////////////////////////////////////////////////////////////////////

public BankAccountBean() {
    this.user = "NoName";
    this.money = 0.0;
    this.withdraw = 1000;
    this.limits = new double[2];
    limits[0] = 1000;
    limits[1] = 10000;
}

public String getUser() {
    return user;
}

public void setUser(String user) {
    String old = this.user;
    this.user = user;
    changeSupport.firePropertyChange("user", old, this.user);
}

public double getMoney() {
    return money;
}

public void setMoney(double money) throws PropertyVetoException {
    try {
        double old = this.money;
        vetoChangeSupport.fireVetoableChange("money", old, money);
        this.money = money;
        changeSupport.firePropertyChange("money", old, this.money);
        //System.err.println("The money has been changed");
    } catch (PropertyVetoException e) {
        //System.err.println("The money has NOT been changed");
        throw e;
    }

}

public double getWithdraw() {
    return withdraw;
}
```

```
    public void setWithdraw(double withdraw) {
        double old = this.withdraw;
        this.withdraw = withdraw;
        changeSupport.firePropertyChange("withdraw", old, this.withdraw);
    }

    public double[] getLimits() {
        return limits;
    }

    public void setLimits(double[] limits) {
        double old[] = this.limits;
        this.limits = limits;
        changeSupport.firePropertyChange("limits", old, this.limits);
    }

    public double getLimits(int i) {
        return this.limits[i];
    }

    public void setLimits(int i, double limit) {
        double old = this.limits[i];
        this.limits[i] = limit;
        changeSupport.fireIndexedPropertyChange("limits", i, old, this.limits[i]);
    }

    @Override
    public void writeExternal(ObjectOutput objectOutput) throws IOException {
        objectOutput.writeObject(new StringBuilder(this.user).reverse().toString());
        objectOutput.writeDouble(this.money);
        objectOutput.writeDouble(this.withdraw);
        objectOutput.writeInt(this.limits.length);
        for(double lim : this.limits)
            objectOutput.writeDouble(lim);
        objectOutput.writeObject(changeSupport);
        objectOutput.writeObject(vetoChangeSupport);
    }

    @Override
    public void readExternal(ObjectInput objectInput) throws IOException,
ClassNotFoundException {
        this.user = new StringBuilder((String)
objectInput.readObject()).reverse().toString();
        this.money = objectInput.readDouble();
        this.withdraw = objectInput.readDouble();
        int len = objectInput.readInt();
        this.limits = new double[len];
        for (int i = 0; i < len; i++)
            this.limits[i] = objectInput.readDouble();
        this.changeSupport = (PropertyChangeSupport) objectInput.readObject();
        this.vetoChangeSupport = (VetoableChangeSupport) objectInput.readObject();
    }

    @Override
    public String toString() {
        return "BankAccountBean{" +
                "user='" + user + '\'' +
                ", money=" + money +
                ", withdraw=" + withdraw +
                ", limits=" + Arrays.toString(limits) +
                '}';
    }
}
```

```
package example2;

import java.beans.*;

public class User  implements VetoableChangeListener, PropertyChangeListener,
java.io.Serializable {

    private BankAccountBean account;

    public User() {
        this.account = new BankAccountBean();
        account.addPropertyChangeListener(this);
        account.addVetoableChangeListener(this);
    }

    public BankAccountBean getAccount() {
        return account;
    }

    public void setAccount(BankAccountBean account) {
        this.account.removePropertyChangeListener(this);
        this.account.removeVetoableChangeListener(this);
        this.account = account;
        this.account.addPropertyChangeListener(this);
        this.account.addVetoableChangeListener(this);
    }

    public void changeMoney() throws PropertyVetoException {
        for (int i = 0; i < 10; i++) {
            account.setMoney((i+1)*5000);
        }
    }

    public void addMoney(double money) throws PropertyVetoException {
        account.setMoney(account.getMoney()+money);
    }

    @Override
    public void propertyChange(PropertyChangeEvent propertyChangeEvent) {
        if (propertyChangeEvent instanceof IndexedPropertyChangeEvent) {
            IndexedPropertyChangeEvent evt = (IndexedPropertyChangeEvent)
propertyChangeEvent;
            System.out.println("\t" + evt.getPropertyName() + "[" + evt.getIndex()  +
                    "] changed from " + evt.getOldValue() + " to " +
evt.getNewValue());
        } else {
            if(propertyChangeEvent.getPropertyName().equals("limits")) {
                System.out.println("\t" + propertyChangeEvent.getPropertyName() + "
changed from " +
                        java.util.Arrays.toString((double[])
propertyChangeEvent.getOldValue()) + " to " +
                        java.util.Arrays.toString((double[])
propertyChangeEvent.getNewValue()));;
            } else {
                System.out.println("\t" + propertyChangeEvent.getPropertyName() + "
changed from " +
                        propertyChangeEvent.getOldValue() + " to " +
propertyChangeEvent.getNewValue());
            }
        }
    }

    @Override
```

```
    public void vetoableChange(PropertyChangeEvent propertyChangeEvent) throws
PropertyVetoException {
        System.out.println("Attempt to set " + propertyChangeEvent.getPropertyName() +
" from " +
                propertyChangeEvent.getOldValue() + " to " +
propertyChangeEvent.getNewValue());
        boolean veto = (double) propertyChangeEvent.getNewValue() < 0;
        if (veto) {
            throw new PropertyVetoException("the reason for the veto",
propertyChangeEvent);
        }

    }

    @Override
    public String toString() {
        return "User{" +
                "account=" + account +
                '}';
    }
}
```

```
package example2;

import java.beans.PropertyVetoException;
import java.io.*;

public class Main {

    public static void serialize(Object obj, String fileName) {
        try {
            FileOutputStream fos = new FileOutputStream(fileName);
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(obj);
            oos.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static Object deSrialize(String fileName) {
        Object res = null;
        try {
            FileInputStream fis = new FileInputStream(fileName);
            ObjectInputStream ois = new ObjectInputStream(fis);
            res = ois.readObject();
            ois.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        return  res;
    }

    public static void main(String[] args) {
        BankAccountBean account = new BankAccountBean();
        account.setUser("Lohankin");
        try {
```

```
            account.setMoney(50000);
        } catch (PropertyVetoException e) {
            e.printStackTrace();
        }
        account.setWithdraw(5000);
        account.setLimits(0,1000);
        account.setLimits(1, 80000);
        System.out.println("Account: " + account);
        User vasisyalij = new User();
        vasisyalij.setAccount(account);
        System.out.println("User: " + vasisyalij);
        try {
            vasisyalij.changeMoney();
        } catch (PropertyVetoException e) {
            e.printStackTrace();
        }
        try {
            vasisyalij.addMoney(330);
        } catch (PropertyVetoException e) {
            e.printStackTrace();
        }
        account.setWithdraw(6000);
        vasisyalij.getAccount().setWithdraw(7500);
        Main.serialize(vasisyalij, "user.ser");
        User user = (User) Main.deSrialize("user.ser");
        System.out.println("Serializable version: " + user);
        user.getAccount().setLimits(new double[]{1.0, 2.0});
        user.getAccount().setLimits(1, 1000);
        user.getAccount().setLimits(0, 10);
        try {
            user.addMoney(-100000);
        } catch (PropertyVetoException e) {
            e.printStackTrace();
        }
    }
}

/*
Account: BankAccountBean{user='Lohankin', money=50000.0, withdraw=5000.0,
limits=[1000.0, 80000.0]}
        //money changed from 0.0 to -1000.0
User: User{account=BankAccountBean{user='Lohankin', money=50000.0, withdraw=5000.0,
limits=[1000.0, 80000.0]}}
Attempt to set money from 50000.0 to 5000.0
   money changed from 50000.0 to 5000.0
Attempt to set money from 5000.0 to 10000.0
   money changed from 5000.0 to 10000.0
Attempt to set money from 10000.0 to 15000.0
   money changed from 10000.0 to 15000.0
Attempt to set money from 15000.0 to 20000.0
   money changed from 15000.0 to 20000.0
Attempt to set money from 20000.0 to 25000.0
   money changed from 20000.0 to 25000.0
Attempt to set money from 25000.0 to 30000.0
   money changed from 25000.0 to 30000.0
Attempt to set money from 30000.0 to 35000.0
   money changed from 30000.0 to 35000.0
Attempt to set money from 35000.0 to 40000.0
   money changed from 35000.0 to 40000.0
Attempt to set money from 40000.0 to 45000.0
   money changed from 40000.0 to 45000.0
Attempt to set money from 45000.0 to 50000.0
   money changed from 45000.0 to 50000.0
Attempt to set money from 50000.0 to 50330.0
    money changed from 50000.0 to 50330.0
```

```
        withdraw changed from 5000.0 to 6000.0
        withdraw changed from 6000.0 to 7500.0
Serializable version: User{account=BankAccountBean{user='Lohankin', money=50330.0,
withdraw=7500.0, limits=[1000.0, 80000.0]}}
        limits changed from [1000.0, 80000.0] to [1.0, 2.0]
       limits[1] changed from 2.0 to 1000.0
       limits[0] changed from 1.0 to 10.0
Attempt to set money from 50330.0 to -49670.0
   Veto Exception
*/
```

Поддержка событий

Компоненты должны оповещать друг друга не только при изменении их свойств. Они могут уведомить друг друга о наступлении определенного события. Модель обработки событий представляет собой, по существу, модель обратных вызовов (*callback*). При создании компонента ему сообщается, какой метод или методы он должен вызывать при возникновении в нем определенного события. Эту модель очень легко использовать в языках, которые позволяют оперировать указателями на методы (чтобы определить обратный вызов, необходимо всего лишь передать указатель на функцию). Однако в *Java* указателей на методы нет, и для реализации новой модели необходимо определить класс, реализующий некоторый специальный интерфейс. Затем можно передать экземпляр такого класса компоненту, обеспечивая, таким образом, обратный вызов. Когда наступит ожидаемое событие, компонент вызовет соответствующий метод объекта, определенного ранее.

Модель обработки событий

Такая модель обработки событий используется как в пакете *AWT*, так и в *JavaBeans API*. В этой модели разным типам событий соответствуют различные классы *Java*. Каждое событие является подклассом класса java.util.EventObject (https://docs.oracle.com/javase/7/docs/api/java/util/EventObject.html).

Для каждого события существует порождающий его объект, который можно получить с помощью метода getSource.

Модель обработки событий базируется на концепции слушателя событий. Слушателем события является объект, заинтересованный в получении данного события. В объекте, который порождает событие (в источнике событий), содержится список слушателей, заинтересованных в получении уведомления о том, что данное событие произошло, а также методы, которые позволяют слушателям добавлять или удалять себя из этого списка. Когда источник порождает событие (или когда объект источника зарегистрирует событие, связанное с вводом информации пользователем), он оповещает все объекты слушателей событий о том, что данное событие произошло.

Источник события оповещает объект слушателя путем вызова специального метода и передачи ему объекта события (экземпляра подкласса EventObject). Для того чтобы источник мог вызвать данный метод, он должен быть реализован для каждого слушателя. Это объясняется тем, что все слушатели событий определенного типа должны реализовывать соответствующий интерфейс. Например, объекты слушателей событий ActionEvent должны реализовывать интерфейс ActionListener. Все интерфейсы слушателей событий являются расширениями интерфейса java.util.EventListener (https://docs.oracle.com/javase/7/docs/api/java/util/EventListener.html). В этом интерфейсе не определяется ни один из методов, но он играет роль интерфейса-метки, в котором однозначно определены все слушатели событий как таковые.

В интерфейсе слушателя событий может определяться несколько методов. По установленному соглашению, методам слушателей событий может быть передан один единственный аргумент, являющийся объектом того события, которое соответствует данному слушателю. В этом объекте должна содержаться вся информация, необходимая программе для формирования реакции на данное событие. Кратко рассмотрим некоторые наиболее часто применяемые события.

Событие класса ActionEvent (https://docs.oracle.com/javase/7/docs/api/java/awt/event/ActionEvent.html). Сообщает о действии над компонентом. Интерфейс слушателя: ActionListener (https://docs.oracle.com/javase/7/docs/api/java/awt/event/ActionListener.html). Методы слушателя: void actionPerformed(ActionEvent). Источник события: Компоненты *Swing*, у которых есть ка кое-то «главное» действие (например, у кнопки — нажатие)

Событие класса ChangeEvent (https://docs.oracle.com/javase/7/docs/api/javax/swing/event/ChangeEvent.html) Запускается некоторыми компонентами и моделями для сообщения о своих изменениях. Интерфейс слушателя: ChangeListener (https://docs.oracle.com/javase/7/docs/api/javax/swing/event/ChangeListener.html) Методы слушателя: void stateChanged(ChangeEvent). Источник события: Некоторые компоненты *Swing*. Многие модели используют это событие для связи с *UI*-представителями.

Событие класса PropertyChangeEvent

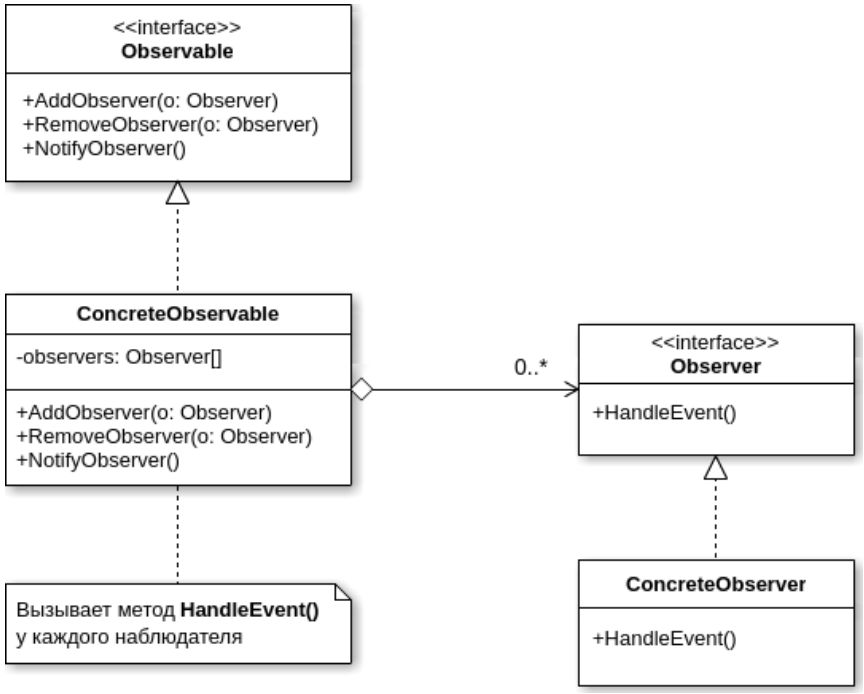
(https://docs.oracle.com/javase/7/docs/api/java/beans/PropertyChangeEvent.html) Обеспечивает работу механизма связанных свойств *JavaBeans*. Интерфейс слушателя: PropertyChangeListener (https://docs.oracle.com/javase/7/docs/api/java/beans/PropertyChangeListener.html). Методы слушателя: void propertyChange (PropertyChangeEvent). Источник события: Практически все графические компоненты *JavaBeans* (в том числе все компоненты *Swing*). Событие PropertyChangeEvent — это основополагающее событие архитектуры *JavaBeans*, оно позволяет следить за тем, как и какие свойства меняются в компоненте.

Рассмотрим краткий пример работы с событием ActionEvent на примере кнопки завершения работы приложения:

```
JButton exitButton = new JButton("Exit");
exitButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.exit(1);
    }
});
```

Шаблон Observer

Рссмотрим шаблон проектирование *Observer* (*Наблюдатель*), который широко используется в библиотеке *Java Swing* и в архитертуре *Java Beans* для управления событиями. Хорошая статья на эту тему приведена в *Википедии* (https://uk.wikipedia.org/wiki/%D0%A1%D0%BF%D0%BE%D1%81%D1%82%D0%B5%D1%80%D1%96%D0%B3%D0%B0%D1%87_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D1%94%D0%BA%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)). Данный шаблон относится к поведенческим шаблонам проектирования (англ. *behavioral patterns*). Реализует у класса механизм, который позволяет объекту этого класса получать оповещения об изменении состояния других объектов и тем самым наблюдать за ними. Классы, на события которых другие классы подписываются, называются *субъектами* (*Subjects*), а подписывающиеся классы называются *наблюдателями* (*Observers*).



При реализации шаблона «наблюдатель» обычно используются следующие классы:

- Observable — интерфейс, определяющий методы для добавления, удаления и оповещения наблюдателей;
- Observer — интерфейс, с помощью которого наблюдатель получает оповещение;
- ConcreteObservable — конкретный класс, который реализует интерфейс Observable;
- ConcreteObserver — конкретный класс, который реализует интерфейс Observer.

```
package test_observer;

public interface Observable {
   void registerObserver(Observer o);
    void removeObserver(Observer o);
    void notifyObservers();
}
```

```
package test_observer;

public interface Observer {
   void update (float temperature, float humidity, int pressure);
}
```

```
package test_observer;

public class CurrentConditionsDisplay implements Observer {
```

```
   private float temperature;
    private float humidity;
    private int pressure;

   @Override
   public void update(float temperature, float humidity, int pressure) {
      this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        display();
   }

   public void display() {
        System.out.printf(
"Сейчас значения: %.1f градусов цельсия и %.1f%% влажности. Давление %d мм рт. ст.\n",
temperature, humidity, pressure);
    }

}
```

```
package test_observer;

import java.util.LinkedList;
import java.util.List;

public class WeatherData implements Observable {
   private List<Observer> observers;
    private float temperature;
    private float humidity;
    private int pressure;

    public WeatherData() {
        observers = new LinkedList<>();
    }

   @Override
   public void registerObserver(Observer o) {
      observers.add(o);
   }

   @Override
   public void removeObserver(Observer o) {
      observers.remove(o);
   }

   @Override
   public void notifyObservers() {
      for (Observer observer : observers)
            observer.update(temperature, humidity, pressure);
   }

   public void setMeasurements(float temperature, float humidity, int pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        notifyObservers();
    }

}
```

```
package test_observer;


//В примере описывается получение данных от метеорологической станции (класс
WeatherData, рассылатель событий) и
//использование их для вывода на экран (класс CurrentConditionsDisplay, слушатель
событий).
//Слушатель регистрируется у наблюдателя с помощью метода registerObserver (при этом
слушатель заносится в список observers).
//Регистрация происходит в момент создания объекта currentDisplay, т.к. метод
registerObserver применяется в конструкторе.
//При изменении погодных данных вызывается метод notifyObservers, который в свою
очередь вызывает метод update
//у всех слушателей, передавая им обновлённые данные.

public class WeatherStation {
   public static void main(String[] args) {
        WeatherData weatherData = new WeatherData();

        Observer currentDisplay = new CurrentConditionsDisplay ();

        weatherData.registerObserver(currentDisplay);

        weatherData.setMeasurements(29f, 65f, 745);
        weatherData.setMeasurements(39f, 70f, 760);
        weatherData.setMeasurements(42f, 72f, 763);
    }
}
/*
Сейчас значения: 29,0 градусов цельсия и 65,0% влажности. Давление 745 мм рт. ст.
Сейчас значения: 39,0 градусов цельсия и 70,0% влажности. Давление 760 мм рт. ст.
Сейчас значения: 42,0 градусов цельсия и 72,0% влажности. Давление 763 мм рт. ст.
*/
```

Создание своего события

Часто при создании приложений вполне хватает стандартных компонентов и событий. Однако бывают случаи, когда нужные возможности они обеспечить не могут. В таком случае придется создать собственный компонент, унаследовав его от какого-либо компонента библиотеки или полностью написав самостоятельно. В этом случае бывает не обходимо уведомлять заинтересованные стороны рассылкой событий и предоставить способ обработки этих событий.

Для компонента, соответствующего архитектуре *JavaBeans*, это означает наличие интерфейса слушателя, класса события и пары методов для присоединения и удаления слушателей. При этом следует помнить про соглашение об именах для событий. Интегрированная среда розработки определяет события, генерируемые компонентом *JavaBeans*, при наличии методов включения и удаления обработчиков. Имена классов событий должны оканчиваться суффиком Event, а сами классы должны расширять класс EventObject.

Предположим, что компонент *JavaBeans* генерирует событие типа *Имя_События*Event. В таком случае интерфейс обработчика должен

называться *Имя_События*Listener, а методы включения и удаления обработчиков должны иметь приведенные ниже имена:

public void add*Имя_События*Listener(*Имя_События*Listener e);
public void delete*Имя_События*Listener(*Имя_События*Listener e);

Рассмотрим, для примера, создание собственного события для ранее разработанных компонентов.

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, сформированное по указанному чуть выше правилу:

```
package lectest3;

public class TemperatureChangeEvent extends java.util.EventObject {
   private static final long serialVersionUID = 1L;

   private double theTemperature;

   public TemperatureChangeEvent(Object source, double temparature) {
      super(source);
      this.theTemperature = temparature;
   }

   public double getTemperature() {
      return theTemperature;
   }
}
```

В нашем событии будет храниться информация о температуре, поэтому объявим свойство, доступное только для чнения: закрытое поле и открытый «*геттер*». Необходимо помнить, 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.

Далее нам нужно описать интерфейс слушателя нашего события. Данный интерфейс будут реализовывать программисты - клиенты компонента, заинтересованные в отслеживании температуры. 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:

```
package lectest3;

import java.util.EventListener;

public interface TemperatureChangeListener extends EventListener {
   void temperatureChanged(TemperatureChangeEvent evt);
}
```

В интерфейсе слушателя мы определили всего один метод temperatureChanged(), который и будет вызываться при изменении температуры. В качестве параметра этому методу передается объект события TemperatureChangeEvent, так что заинтересованный в изменении температуры программист, реализовавший интерфейс слушателя, будет знать подробности о событии.

Теперь нам остается включить поддержку события в класс самого компонента. Для этого в нем нужно определить пару методов для присоединения и отсоединения слушателей TemperatureChangeEvent, эти методы должны следовать схеме именования событий *JavaBeans*. В нашем случае методы будут именоваться addTemperatureChangeListener() и removeTemperatureChangeListener(). Слушатели, которых программисты регистрируют в данных методах, будут оповещаться о изменении температуры. 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 (*multicas*t) количества слушателей. В компоненте создается список, в котором и хранятся все присоединяемые к компоненту слушатели. При возникновении события все находящиеся в списке слушатели получают о нем полную информацию.

В 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. Реализуем в компоненте список слушателей, так что выберем второй способ:

```
package lectest3;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Iterator;

public class TempBean {
   private static final long serialVersionUID = 1L;

   private double temperature;
   private double[] delta;

   public TempBean() {
      this.temperature = 0;
      this.delta = new double[2];
      delta[0] = -2;
      delta[1] = 8;
   }

   public double getTemperature() {
      return temperature;
   }

   public void setTemperature(double temperature) {
      this.temperature = temperature;
      fireTemperatureChange(); //!!!
   }

   public double[] getDelta() {
      return delta;
   }

   public void setDelta(double[] delta) {
      this.delta = delta;
   }

   public double getDelta(int i) {
      return delta[i];
   }

   public void setDelta(int i, double delta) {
      this.delta[i] = delta;
   }

   @Override
   public String toString() {
      return "TempBean [temperature=" + temperature + ", delta="
            + Arrays.toString(delta) + "]";
   }

   //
   private ArrayList<TemperatureChangeListener> listen =
         new ArrayList<TemperatureChangeListener>();

   public synchronized void addTemperatureChangeListener(TemperatureChangeListener l) {
      if(!listen.contains(l)) {
         listen.add(l);
      }
   }
```

```
   public synchronized void removeTemperatureChangeListener(TemperatureChangeListener
l) {
      if(listen.contains(l)) {
         listen.remove(l);
      }
   }

   protected void fireTemperatureChange() {
      Iterator <TemperatureChangeListener> i = listen.iterator();
      while( i.hasNext() ) {
         (i.next()).temperatureChanged(new TemperatureChangeEvent(this, temperature));
      }
   }
   //
}
```

При изменении температуры будет вызван метод fireTemperatureChange(), обязанностью которого является оповещение слушателей о событии. Кстати, название вида fireXXX() (или fireXXXEvent()) является неофициальным стандартом для методов, «запускающих» высокоуровневые события. Слушатели TemperatureChangeListener будут храниться в списке ArrayList, адаптированном только под хранение объектов TemperatureChangeListener. Благодаря большим возможностям стандартного списка ArrayList методы для присоединения и отсоединения слушателей реализовать очень просто: им нужно лишь использовать соответствующие возможности списка. Также несложно выполнить и оповещение слушателей о событиях: для каждого элемента списка мы вызываем метод temperatureChanged(), передавая ему в качестве параметра созданный объект-событие.

В нашем примере 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: можно просто объявить методы для присоединения и отсоединения слушателей как synchronized. Аналогично можно изменить сигнатуру метода fireTemperatureChange() (он тоже работает со списком, а список не должен изменяться при отсылке событий).

Есть и еще один способ включить для компонента поддержку многозадачного окружения: с помощью класса java.util.Collections и статического метода synchronizedList(). Этот метод вернет версию списка ArrayList со встроенной поддержкой многозадачности. Но, мы на предыдущих практиках уже кратко говорили, о том, что работать с компонентами *Swing* из нескольких потоков без специальных усилий нельзя, так что смысла в добавлении слушателей из нескольких потоков, как правило, нет.

Допишем наше приложение и проверим работу нашего нового компонента.

```
package lectest3;

import java.beans.Beans;
import java.io.IOException;

public class Thermometer implements TemperatureChangeListener {

   private TempBean bean;

   //public Thermometer() throws ClassNotFoundException, IOException {
      //bean = (TempBean) Beans.instantiate(null, "lectest3.TempBean");
   public Thermometer() {
      bean = new TempBean();
      bean.addTemperatureChangeListener(this);
   }

   @Override
   public void temperatureChanged(TemperatureChangeEvent evt) {
      System.out.println("Currrent temperature: " + evt.getTemperature());
   }

   public void changeTemp() {
      for(int i = 0; i < 10; i++) {
         bean.setTemperature(i*0.5);
      }
   }

   public static void main(String[] args) throws ClassNotFoundException, IOException {
      new Thermometer().changeTemp();
   }
/*
Currrent temperature: 0.0
Currrent temperature: 0.5
Currrent temperature: 1.0
Currrent temperature: 1.5
Currrent temperature: 2.0
Currrent temperature: 2.5
Currrent temperature: 3.0
Currrent temperature: 3.5
Currrent temperature: 4.0
Currrent temperature: 4.5
 */
}
```

Подобная цепочка действий повторяется для любого нового события JavaBeans: вы описываете класс события и интерфейс его слушателя, следуя хорошо известным правилам, и добавляете в компонент пару методов для регистрации слушателей и их отсоединения. Для хранения слушателей используется подходящий список, все хранящиеся в нем слушатели оповещаются о возникновении события.

Обратите внимание на то, что хранение слушателей в простых коллекциях данных, таких как список ArrayList, накладывает на слушателя определенные ограничения. Дело в том, что слушатель вызывается в момент

перечисления элементов списка, и удалить себя или другого слушателя в этот момент не сможет, так как список на это не рассчитан. Модификацию списка слушателей лучше проводить не из кода слушателя.

Приведем еще один пример создания собственных событий для другого, ранее созданного компонента *JavaBean*.

```
package example3;

public class MoneyWarningEvent extends java.util.EventObject {

    private static final long serialVersionUID = 1L;

    private double money;

    public MoneyWarningEvent(Object source, double money) {
        super(source);
        this.money = money;
    }

    public double getMoney() {
        return money;
    }
}
```

```
package example3;

public interface MoneyWarningListener extends java.util.EventListener {
    void moneyDangerousChanged(MoneyWarningEvent moneyChangeEvent);
}
```

```
package example3;

import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Iterator;

public class BankAccountBean implements java.io.Externalizable {
    private String user;
    private double money;
    private double withdraw;
    private double[] limits;

    ////////////////////////////////////////////////////////////////////////
    private ArrayList<MoneyWarningListener> listen =
            new ArrayList<MoneyWarningListener>();

    public synchronized void addMoneyWarningListener(MoneyWarningListener listener) {
        if(!listen.contains(listener)) {
            listen.add(listener);
        }
    }

    public synchronized void removeMoneyWarningListener(MoneyWarningListener listener)
{
```

```
    if(listen.contains(listener)) {
        listen.remove(listener);
    }
}

protected void fireMoneyWarning() {
    Iterator<MoneyWarningListener> i = listen.iterator();
    while( i.hasNext() ) {
        (i.next()).moneyDangerousChanged(new MoneyWarningEvent(this, this.money));
    }
}
////////////////////////////////////////////////////////////////////////////////


public BankAccountBean() {
    this.user = "NoName";
    this.money = 0.0;
    this.withdraw = 1000;
    this.limits = new double[2];
    limits[0] = 1000;
    limits[1] = 10000;
}

public String getUser() {
    return user;
}

public void setUser(String user) {
    this.user = user;
}

public double getMoney() {
    return money;
}

public void setMoney(double money) {
    this.money = money;
    if ((this.money <= this.limits[0]) || (this.money >= this.limits[1]))
        fireMoneyWarning(); //!!!
}

public double getWithdraw() {
    return withdraw;
}

public void setWithdraw(double withdraw) {
    this.withdraw = withdraw;
}

public double[] getLimits() {
    return limits;
}

public void setLimits(double[] limits) {
    this.limits = limits;
}

public double getLimits(int i) {
    return this.limits[i];
}

public void setLimits(int i, double limit) {
    this.limits[i] = limit;
}
```

```
    @Override
    public void writeExternal(ObjectOutput objectOutput) throws IOException {
        //objectOutput.writeObject(this.user);
        objectOutput.writeObject(new StringBuilder(this.user).reverse().toString());
        objectOutput.writeDouble(this.money);
        objectOutput.writeDouble(this.withdraw);
        objectOutput.writeInt(this.limits.length);
        for(double lim : this.limits)
            objectOutput.writeDouble(lim);
        objectOutput.writeInt(this.listen.size());
        for (MoneyWarningListener listener : this.listen)
            objectOutput.writeObject(listener);
    }

    @Override
    public void readExternal(ObjectInput objectInput) throws IOException,
ClassNotFoundException {
        //this.user = (String)objectInput.readObject();
        this.user = new StringBuilder((String)
objectInput.readObject()).reverse().toString();
        this.money = objectInput.readDouble();
        this.withdraw = objectInput.readDouble();
        int len = objectInput.readInt();
        this.limits = new double[len];
        for (int i = 0; i < len; i++)
            this.limits[i] = objectInput.readDouble();
        len = objectInput.readInt();
        for (int i = 0; i < len; i++)
            this.listen.add((MoneyWarningListener)objectInput.readObject());
    }

    @Override
    public String toString() {
        return "BankAccountBean{" +
                "user='" + user + '\'' +
                ", money=" + money +
                ", withdraw=" + withdraw +
                ", limits=" + Arrays.toString(limits) +
                '}';
    }
}
```

```
package example3;

public class User implements MoneyWarningListener, java.io.Serializable {

    private BankAccountBean account;

    public User() {
        this.account = new BankAccountBean();
        account.addMoneyWarningListener(this);
    }

    public BankAccountBean getAccount() {
        return account;
    }

    public void setAccount(BankAccountBean account) {
        this.account.removeMoneyWarningListener(this);
        this.account = account;
        this.account.addMoneyWarningListener(this);
```

```
    }

    public void addMoney(double money) {
        account.setMoney(account.getMoney()+money);
    }

    @Override
    public void moneyDangerousChanged(MoneyWarningEvent moneyChangeEvent) {
        String name = ((BankAccountBean) moneyChangeEvent.getSource()).getUser();
        System.out.println("Dear " + name +  "!\n\tWarning!!! Current money: " +
moneyChangeEvent.getMoney());;
    }

    @Override
    public String toString() {
        return "User{" +
                "account=" + account +
                '}';
    }
}
```

```
package example3;

import java.io.*;

public class Main {

    public static void serialize(Object obj, String fileName) {
        try {
            FileOutputStream fos = new FileOutputStream(fileName);
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(obj);
            oos.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static Object deSrialize(String fileName) {
        Object res = null;
        try {
            FileInputStream fis = new FileInputStream(fileName);
            ObjectInputStream ois = new ObjectInputStream(fis);
            res = ois.readObject();
            ois.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        return  res;
    }

    public static void main(String[] args) {
        BankAccountBean account = new BankAccountBean();
        account.setUser("Lohankin");
        account.setMoney(5000);
        account.setWithdraw(5000);
        account.setLimits(0,1000);
```

```
        account.setLimits(1, 80000);
        System.out.println("Account: " + account);
        User vasisyalij = new User();
        vasisyalij.setAccount(account);
        System.out.println("User: " + vasisyalij);
        for(int i = 0; i < 5; i++)
            vasisyalij.addMoney(-1000);
        Main.serialize(vasisyalij, "wuser.ser");
        User user = (User) Main.deSrialize("wuser.ser");
        System.out.println("Serializable version: " + user);
        for(int i = 0; i < 8; i++)
            vasisyalij.addMoney(10000);
    }
}
/*
Account: BankAccountBean{user='Lohankin', money=5000.0, withdraw=5000.0,
limits=[1000.0, 80000.0]}
User: User{account=BankAccountBean{user='Lohankin', money=5000.0, withdraw=5000.0,
limits=[1000.0, 80000.0]}}
Dear Lohankin!
        Warning!!! Current money: 1000.0
Dear Lohankin!
        Warning!!! Current money: 0.0
Serializable version: User{account=BankAccountBean{user='Lohankin', money=0.0,
withdraw=5000.0, limits=[1000.0, 80000.0]}}
Dear Lohankin!
        Warning!!! Current money: 80000.0
*/
```