Конспект лекций по Java. Занятие 11

В.Фесюнов

1. Коллекции объектов Java (продолжение).

1.1. Коллекции-множества (Set)

Продолжим знакомство с коллекциями Java. Рассмотрим коллекции — множества (Set). Два основных класса для работы с множествами в Java — это HashSet и TreeSet.

Класс **HashSet** .

Имеет два основных конструктора (аналогично **ArrayList**)

```
public HashSet()

  Строит пустое множество

public HashSet(Collection c)

  Строит множество из элементов коллекции
```

и следующие методы

```
public Iterator iterator()

public int size()

public boolean isEmpty()

public boolean contains(Object o)
```

```
public boolean add(Object o)

public boolean addAll(Collection c)

public Object[] toArray()

public boolean remove(Object o)

public boolean removeAll(Collection c)

public boolean retainAll(Collection c)

  (retain — сохранить). Выполняет операцию "пересечение множеств".


public void clear()

public Object clone()
```

Они аналогичны рассмотренным ранее методам ArrayList за исключением того, что метод add(Object o) добавляет объект в множество только в том случае, если его там нет. Возвращаемое методом значение — true, если объект добавлен, и false, если нет.

Название Hash... происходит от понятия хэш-функция. Хэш-функция — это функция, сужающая множество значений объекта до некоторого подмножества целых чисел. Класс Object имеет метод hashCode(), который используется классом HashSet для эффективного размещения объектов, заносимых в коллекцию. В классах объектов, заносимых в HashSet, этот метод должен быть переопределен (*override*).

Класс **TreeSet** .

В дополнение к обычным двум конструкторам

```
public TreeSet()

public TreeSet(Collection c)
```

имеет еще один

```
public TreeSet(Comparator c)
```

Наличие этого конструктора связано со следующим свойством TreeSet — он поддерживает упорядоченность множества. Сначала рассмотрим, что дает упорядоченность множества, а потом — средства определения порядка на множестве объектов.

В дополнение к методам, которые есть в HashSet, TreeSet имеет ряд методов, связанных с упорядоченностью множества.

```
public SortedSet subSet(Object fromElement, Object toElement)

  Здесь SortedSet — интерфейс "упорядоченное множество". Метод возвращает
  подмножество в заданном интервале объектов (toElement не входит в это
  подмножество). Результирующее множество обладает тем свойством, что в
  него нельзя включить элемент вне заданного интервала (выдает
  IllegalArgumentException).


public SortedSet headSet(Object toElement)

  Аналогично предыдущему, но с начала исходного множества — до
  элемента toElement, исключая его.


public SortedSet tailSet(Object fromElement)

  Аналогично subset, но начиная с элемента fromElement до конца множества.


public Object first()

  Первый элемент множества


public Object last()

  Последний элемент множества
```

2. Задание порядка в Java

Разберемся с вопросом задания порядка на некотором множестве объектов.

Определение порядка на множестве не такой простой вопрос, если это понятие нужно поддерживать в программе. Обычно он решается путем задания функции сравнения. Т.е. для элементов какого-то множества задается функция, которая позволяет сравнить два элемента этого множества. В качестве результата такая функция возвращает целое число, которое меньше нуля, если первый элемент меньше второго, ноль, если они равны, и больше нуля, если первый элемент больше второго. Иными словами, для задания порядка нам нужно запрограммировать некоторую функцию, возвращающую логическую разность двух объектов множества.

Однако даже при таком подходе остается масса вариантов реализации данного принципа. В Java эта проблема решена стандартными средствами библиотеки языка, а именно — путем задания интерфейсов, используемых при задании порядка. При этом, Java позволяет сравнивать объекты как одного, так и разных классов.

Для задания порядка используются следующие интерфейсы: Comparable и Comparator. Они дают два различных способа задания порядка. Мы рассмотрим оба этих интерфейса, определим их назначение, разберем, как ими пользоваться и чем они отличаются друг от друга.

Интерфейс *Comparable* предназначен для определения так называемого естественного порядка (*natural ordering*). Этот интерфейс определен в пакете java.lang. Обратившись к документации мы увидим, что данный интерфейс содержит всего один метод.

public int compareTo(Object o)

Сравнивает объект с другим объектом.

Разберем пример. Пусть мы хотим задать порядок на множестве объектов класса A

```
public class A implements Comparable {

      String name;
```

```
      . . .

}
```

Причем, мы хотим сравнивать объекты по полю name в алфавитном порядке. Тогда мы задаем implements Comparable и определяем в классе метод

```
public int compareTo(Object obj) {
      return name.compareTo(((A)obj).name);
}
```

В результате класс A должен выглядеть так

```
public class A implements Comparable {

      String name;

  public int compareTo(Object obj) {
      return name.compareTo(((A)obj).name);
  }
  . . .
}
```

Как видно из данного примера задать нужный порядок не очень сложно. Разберем только, что же собственно здесь делается.

Метод compareTo(...) будет вызываться тогда, когда требуется сравнить два объекта. Он будет вызываться для некоторого объекта и в качестве параметра ему будет передан другой объект, который требуется сравнить с данным.

В нашем примере реализация метода compareTo(...) опирается на метод compareTo(...) класса String (поле name — класса String). Если мы обратимся к документации по методу compareTo(...) класса String, то увидим, что он сравнивает посимвольно две строки, пока не обнаружит два первых неодинаковых символа. В качестве результата он возвращает 0, если строки совпадают и разность кодов двух первых отличных друг

от друга символов, если нет.

Замечание:

Следует также обратить внимание на то, что параметр obj сначала приводится к классу A (downcasting), из него выбирается поле name и передается в качестве параметра методу compareTo(...) класса String. Если в качестве параметра будет передан объект не класса A, а какого-то другого класса, то возникнет ClassCastException. Если нам требуется сравнивать объекты нескольких различных классов (такая необходимость возникает довольно редко), то это нужно учесть в методе compareTo(...) .

Возвращаясь к TreeSet, нужно указать, что все объекты, заносимые в TreeSet, должны удовлетворять интерфейсу Comparable. Исключение составляет случай использования конструктора

public TreeSet(Comparator c)

который мы рассмотрим позже. Если добавляемый объект не удовлетворяет интерфейсу, то возникнет exception.

Класс TreeSet организует сравнение элементов, используя compareTo(...) и расположит их в соответствии с заданным этим методом порядком.

Как было указано, Java позволяет сравнивать объекты различных классов. Рассмотрим для примера, как нужно переопределить метод compareTo(), чтобы он учитывал возможность сравнения как объектов класса A как друг с другом, так и с объектами некоторого класса B. Для этого метод compareTo(...) должен выглядеть примерно так

```
public int compareTo(Object obj) {
      if ( obj instanceof A )
          return name.compareTo(((A)obj).name);
      return ... здесь сравнение с объектом B
}
```

В свою очередь, в классе B, нужно указать, что он удовлетворяет интерфейсу Comparable и реализовать в нем метод compareTo(), который учитывает возможность сравнения B с A.

Итак, интерфейс Comparable дает нам возможность задавать порядок на множестве объектов одного или нескольких классов. Однако и этого не всегда хватает. Бывают

случаи, когда в одной программе требуется иметь несколько различных вариантов порядка на одном и том же множестве. Например, если у нас есть список сотрудников, то нам может потребоваться порядок по фамилиям и порядок по табельным номерам. В этом случае интерфейс Comparable нас не спасает — он позволяет задать только один порядок.

В описанном случае вместо интерфейса Comparable следует использовать интерфейс Comparator.

Интерфейс **Comparator** .

Определен в пакете java.util. Работа с использованием Comparator отличается от того, что мы рассмотрели по интерфейсу Comparable. Здесь строится отдельный вспомогательный класс (будем называть его компаратор), используемый для сравнения объектов.

Обратимся к документации. Интерфейс Comparator имеет два метода

```
public int compare(Object o1, Object o2)

  и

public boolean equals(Object obj)
```

Второй из методов используется редко — он предназначен для сравнения самих компараторов. Более того, реализовывать данный метод необязательно.

Собственно сравнение объектов выполняет первый метод. Рассмотрим интерфейс Comparator на примере.

Пример

```
public class Employer {    // служащий
     int tabnom;         // табельный номер
     String name;       // ФИО
     static NameComparator nameComparator = new  NameComparator();
     static TabComparator tabComparator = new  TabComparator();
```

```
     public static Comparator getNameComparator() {
           return nameComparator;
     }

     public static Comparator getTabComparator() {
           return tabComparator;
     }

     static class NameComparator implements Comparator {

          public int compare(Object o1, Object o2) {
                return ((Employer)o1).name.compareTo(((Employer)o2).name);
          }
// метод equals() реализовывать не обязательно
     }

     static class TabComparator  implements Comparator {
          public int compare(Object o1, Object o2) {
                return ((Employer)o1).tabnom —  ((Employer)o2).tabnom;
          }
     }
   . . .
}
```

В классе Employer заданы два статических вложенных класса NameComparator и TabComparator, а также созданы объекты этих классов. Для получения конкретного компаратора созданы методы getNameComparator() и getTabComparator().

Имея такой класс, мы можем сравнивать объекты класса при помощи одного или другого компаратора. Например, рассмотрим, как построить TreeSet, упорядоченый по ФИО служащего.

TreeSet reestr = new TreeSet(Employer.getNameComparator());

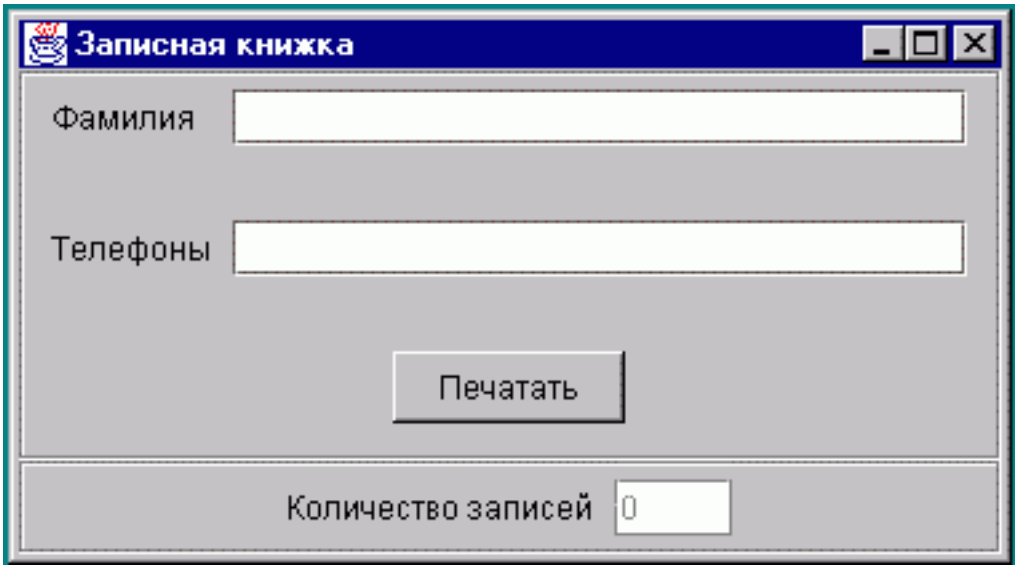
Далее мы можем добавлять в reestr объекты класса Employer. Они будут автоматически упорядочиваться по ФИО.

На этом мы завершим рассмотрение множеств и рассмотрим практический пример использования множеств. Попутно углубим наши познания по изучению

возможностей создания визуальных программ на Java.

3. Практическая работа

Решим следующую задачу. Надо создать "Записную книжку" очень простой структуры. В нее заносится фамилия и список телефонов. Наше приложение будет очень простым и не будет "страдать от избытка функциональности", его внешний вид примерно следующий.



Здесь видны два поля ввода — для фамилии и списка телефонов, кнопка "Печатать" и поле для индикации количества записей в нашей книжке. Конечно, в таком приложении должны быть еще средства сохранения/восстановления информации, поиска и проч. Но это пока за рамками наших возможностей. Для этого нужно просто больше знать о Java.

Наше приложение должно работать так. При вводе информации во второе поле (Телефоны) информация (фамилия и телефон) будет запоминаться в некотором внутреннем хранилище программы, (мы будем для этого использовать TreeSet). При этом поле-индикатор количества записей изменяется, а поля ввода очищаются. При нажатии на кнопку "Печатать" весь накопленный нами список записей выводится на печать (в поток System.out) в некотором удобочитаемом формате.

Обратим внимание на внешний вид приложения.

- На экране явно видны две панели, причем с границами (border). Мы должны рассмотреть, как формировать границы панелей.
- Верхняя панель поделена на три горизонтальные части, в каждой из которых расположены различные визуальные компоненты (метки, поля и одна кнопка). Для формирования такого внешнего вида нужно познакомиться с менеджерами компоновки (Layout Manager). В частности, для нашего приложения понадобится менеджер GridLayout.

Разберем эти вопросы, прежде чем приступим к реализации.

3.1. Границы панелей.

В библиотеке Swing есть много средств для создания различного рода границ. Мы рассмотрим сейчас только то, что нужно непосредственно для нашего приложения. Граница, представленная на рисунке, носит название Etched Border. Ее можно создать так.

```
JPanel pnl = new JPanel();
pnl.setBorder(BorderFactory.createEtchedBorder());
```

Здесь pnl — панель, для которой мы задаем границу типа Etched Border. Для установки границ используется метод setBorder(...) класса JPanel. Сами границы представляются объектами класса Border и могут быть порождены методами класса BorderFactory (пакет javax.swing). Посмотрим подробнее это в документации.

3.2. Менеджеры компоновки.

На занятии 7 мы уже познакомились с одним менеджером компоновки - BorderLayout. Как уже отмечалось, этот менеджер является менеджером по умолчанию для рабочей части (Content Pane) окна JFrame. Домашнее задание прошлого занятия связано с неявным использованием другого менеджера компоновки — FlowLayout. Для правильного решения домашнего задания нужно было создать панели (JPanel) для полей ввода и вывода. А панель JPanel по умолчанию использует FlowLayout как менеджер компоновки.

Это пригодится нам и в данном задании, поэтому рассмотрим FlowLayout несколько подробнее. Данный менеджер размещает внутренние подкомпоненты слева направо в строке, выделяя им, по возможности, их предпочтительный размер (preferred size). Если очередная компонента не помещается в текущей строке, то она переносится на следующую строку, и так далее.

Как видим, это довольно простой менеджер компоновки. Он не обеспечивает нужной гибкости в разбиении окна приложения. Но он обладает одним очень важным свойством (как мы убедимся позже) — он "не растягивает" компоненты, как некоторые другие менеджеры. Поэтому один из возможных принципов построения нужной структуры окна приложения состоит в том, что мы разбиваем окно при помощи других менеджеров компоновки (не FlowLayout) на элементарные части или боки. В эти блоки мы вставляем не непосредственно элементарные визуальные компоненты (метки, поля, кнопки и т.п.) а панели JPanel, а уже в панели — элементарные компоненты. Т.е. FlowLayout следует применять на самом нижнем уровне вложенности вспомогательных панелей.

В нашем приложении нам понадобится другой менеджер компоновки — **GridLayout** . Этот менеджер позволяет разбить область на ячейки одинакового размера в виде сетки. Он не является менеджером по умолчанию и может быть установлен только явно. Поэтому сначала разберем, как установить менеджер компоновки. Обратимся к документации по JPanel и обратим внимание на конструкторы и метод **setLayout(...)** , который унаследован классом JPanel от класса **Container** (пакет java.awt).

public JPanel(LayoutManager layout)

Создает панель с указанным менеджером компоновки

public void setLayout(LayoutManager mgr)

Устанавливает менеджер компоновки для данного контейнера

Это два основных способа установки менеджера компоновки для панели. Т.е. мы можем установить менеджер компоновки либо сразу при создании панели (в конструкторе), либо позже, используя setLayout(...).

Теперь рассмотрим GridLayout. Два его основных конструктора это

public GridLayout(int rows, int cols)

Служит для разбивки области на указанное количество строк и столбцов. Если в качестве одного из параметров указать 0, то количество элементов по данному измерению будет произвольным. Но один из параметров обязан быть ненулевым.

public GridLayout(int rows, int cols, int hgap, int vgap)

Аналогичен предыдущему конструктору, но дополнительно позволяет задать расстояние между ячейками.

У класса GridLayout есть множество методов, но они используются редко и указанных конструкторов нам будет вполне достаточно.

В качестве отправной точки рассмотрим следующий вариант класса PhoneNotes - основного класса нашего приложения

```
// PhoneNotes.java: Записная книжка

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class PhoneNotes extends JFrame {

    JTextField fldFio = new JTextField(25);
    JTextField fldPhone = new JTextField(25);
    JTextField fldCnt = new JTextField(4);

    public PhoneNotes() {
     super("Записная книжка");

     try  {
       UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
     }
     catch(Exception e) {
     }

    setSize(400, 250);
    Container c = getContentPane();
```

```
// Центральная панель
    JPanel centerPanel = new JPanel(new GridLayout(3, 2, 5, 5));
    centerPanel.setBorder(BorderFactory.createEtchedBorder());
    JLabel aLabel = new JLabel("Фамилия ");
    centerPanel.add(aLabel);
    centerPanel.add(fldFio);
    aLabel = new JLabel("Телефон ");
    centerPanel.add(aLabel);
    centerPanel.add(fldPhone);
    JButton btn = new JButton("Печатать");
    centerPanel.add(btn);
    c.add(centerPanel, BorderLayout.CENTER);
// Нижняя панель
    JPanel statusPanel = new JPanel();
    statusPanel.setBorder(BorderFactory.createEtchedBorder());
    aLabel = new JLabel("Количество записей ");
    statusPanel.add(aLabel);
    fldCnt.setEnabled(false);
    statusPanel.add(fldCnt);
    c.add(statusPanel, BorderLayout.SOUTH);
// Listener'ы полей и кнопок
    fldPhone.addActionListener(new ActionListener(){
          public void actionPerformed(ActionEvent e) {
//   ???
          }
        });
    btn.addActionListener(new ActionListener(){
          public void actionPerformed(ActionEvent e) {
//   ???
          }
        });
    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);

    setVisible(true);
  }
```

```
  public static void main(String[] args) {
    new PhoneNotes();
  }
}
```

Оттранслируем и запустим на выполнение данное приложение. Мы увидим, что оно еще далеко от совершенства в плане внешнего вида и, что в нем не реализована требуемая нам функциональность.

4. Задание

- **1.** Улучшим внешний вид приложения. Изменим сетку с 3*2 на 3*1 и в каждую ячейку сетки добавим панель, а уже в эти панели будем добавлять метки, поля, кнопки.
- **2.** Рассмотрим класс (бизнес класс) для хранения одной записи записной книжки. Это класс Person.

```
// Person.java

import java.io.*;

public class Person implements Serializable {

    private String name;
    private String phone;

    Person(String name, String phone) {
        this.name = name;
        this.phone = phone;
    }

    Person() {
        this(null, null);
    }

    String getName() {
        return name;
```

```
    }

    String getPhone() {
        return phone;
    }

}
```

Этот класс предназначен для хранения информации по одной персоне.

- **3.** В слушателе (listener) поля fldPhone реализуем создание объекта этого класса на основе информации из полей fldFio и fldPhone. Кроме того, реализуем очистку этих полей после создания данного объекта.
- **4.** Добавим в класс PhoneNotes поле для коллекции записей (поле класса TreeSet) и реализуем занесение информации (объектов Person) в эту коллекцию.
- **5.** В классе Person реализуем интерфейс Comparable, в частности, определим метод compareTo(...) для сравнения двух персон .
- **6.** В классе Person определим также метод toString() для формирования строки из объекта. Используем этот метод для организации печати списка по кнопке "Печатать".