Конспект лекций по Java. Занятие 10

В.Фесюнов

1. Статические вложенные классы

На прошлом занятии мы познакомились с вложенными классами Java и, в частности, с анонимными вложенными классами. Однако, мы рассмотрели не все, что относится к данной теме. Еще один "подвид" вложенных классов — это статические вложенные классы. Т.е. у inner-класса может быть описатель static.

Пример

```
class Outer2 {
    . . .
    static class Inner2 {
        . . .
    }
    . . .
}
```

Статические inner-классы используются тогда, когда из вложенного класса не требуется обращаться к нестатическим полям и методам охватывающего класса. Дело в том, что объекты статических вложенных классов не содержат ту неявную ссылку на объект охватывающего класса, о которой мы говорили на прошлом занятии. С другой стороны, при порождении объекта статического inner-класса не нужен объект охватывающего класса. Соответственно объект такого класса может быть без проблем порожден из статического метода охватывающего класса или вообще из другого класса.

Если быть более точным, то из статического класса можно обратиться к нестатическим полям и методам охватывающего класса, но не напрямую, а через ссылку на объект

охватывающего класса.

Расширенный пример

```
class Outer3 {
    String name;
    . . .
    static class Inner3 {
        . . .
        public void f(Outer3 obj) {
              System.out.println(obj.name);    // Здесь без obj нельзя
        }
    }
    . . .
    public static Inner3 createInner() {
         return new Inner3();
    }
    . . .
}
```

В данном примере в классе Outer3 описан статический inner-класс Inner3. В этом классе метод f(...) печатает поле name охватывающего класса, но для этого в f(...) передается параметр типа Outer3. В самом классе Outer3 есть статический метод createInner(), который порождает объект класса Inner3.

Приведем также примеры создания объекта класса Inner3 извне класса Outer3.

```
Outer3.Inner3 obj1 = new Outer3.Inner3();   // явное порождение
Outer3.Inner3 obj2 = Outer3.createInner();   // порождение через метод createInner()
```

Во всем остальном статические вложенные классы такие же, как и нестатические.

2. Коллекции объектов Java.

Подробное знакомство со стандартной библиотекой Java мы начнем с *коллекций объектов* . Они реализованы различными классами пакета java.util. Мы рассмотрим не весь этот пакет, а только классы, связанные с коллекциями, что, однако, составляет

значительную его часть.

Что такое *коллекции* объектов. Это очень мощный и исключительно полезный механизм. Простейшей коллекцией является массив. Но массив имеет ряд недостатков. Один из самых существенных - размер массива фиксируется до начала его использования. Т.е. мы должны заранее знать или подсчитать, сколько нам потребуется элементов коллекции до начала работы с ней. Зачастую это неудобно, а в некоторых случаях — невозможно.

Поэтому все современные базовые библиотеки различных языков программирования имеют тот или иной вариант поддержки коллекций объектов. Коллекции обладают одним важным свойством — их размер не ограничен. Выделение необходимых для коллекции ресурсов спрятано внутри соответствующего класса. Работа с коллекциями облегчает и упрощает разработку приложений. Отсутствие же подобного механизма в составе средств разработки вызывает серьезные проблемы, которые приводят либо к различным ограничениям в реализации либо к самостоятельной разработке адекватных средств для хранения и обработки массивов информации заранее неопределенного размера.

В Java средства работы с коллекциями весьма развиты и имеют много приятных и полезных особенностей, облегчающих жизнь разработчикам.

Начнем изучение коллекций в Java с примера.

Пример

Одним из широко используемых классов коллекций является ArrayList. Пример использования этой коллекции приводится ниже.

```
import java.util.*;
import java.io.*;

public class ArrayListTest {
     ArrayList lst = new ArrayList();
     Random generator = new Random();

     void addRandom() {
           lst.add(new Integer(generator.nextInt()));
```

```
    }


    public String toString() {
         return lst.toString();
    }

    public static void main(String args[]) {
        ArrayListTest tst = new ArrayListTest();
        for(int i = 0; i < 100; i++ )
               tst.addRandom();
         System.out.println("Сто случайных чисел: "+tst.toString());
    }
}
```

Рассмотрим данный пример подробнее. Здесь, кроме класса ArrayList, использованы еще ряд классов бибилиотеки Java.

- Random — класс из java.util. Расширяет возможности класса Math по генерации случайных чисел (см. документацию).
- Integer — так называемый wrapper-класс (класс-обертка) для целых (int). Он использован потому, что в коллекцию нельзя занести данные элементарных типов, а только объекты классов.

Класс ArrayListTest имеет два поля — поле lst класса ArrayList и поле generator класса Random, используемое для генерации случайных чисел. Метод addRandom() генерирует и заносит в коллекцию очередное случайное число. Метод toString() просто обращается к методу toString() класса ArrayList, который обеспечивает формирование представления списка в виде строки.

Метод main(...) создает объект класса ArrayListTest и организует цикл порождени 100 случайных чисел с занесением их в коллекцию, вызыва метод addRandom(). После этого он печатает результат.

Оттранслируем и запустим данную программу.

Этот пример не демонстрирует особых преимуществ коллекций, а лишь технику их использования. Из него видно, что добавить элемент в коллекцию можно методом add(...) класса ArrayList и при этом мы нигде не указываем размер коллекции.

Рассмотрим коллекции более систематично.

В Java коллекции объектов разбиты на три больших категории: List (список), Set (множество) и Map (отображение).

Посмотрим документацию. В java.util определены интерфейсы:

```
Collection
       |
       +------- List
       |
       +------- Set

 Map
```

Эти интерфейсы составляют основу для построения классов коллекций Java.

- List — это список объектов. Объекты можно добавлять в список (метод add()), заменять в списке (метод set()), удалять из списка (метод remove()), извлекать (метод get()). Существует также возможность организации прохода по списку при помощи итератора.
- Set — множество объектов. Те же возможности, что и у List, но объект может входить в множество только один раз. Т.е. двойное добавление одного и того же объекта в множество не изменяет само множество.
- Map — отображение или ассоциативный массив. В Map мы добавляем не отдельные объекты, а пары объектов (*ключ, значение*). Соответственно есть операции поиска *значения* по *ключу* . Добавление пары с уже существующим в Map ключем приводит к замене, а не к добавлению. Из отображения (Map) можно получить множество (Set) ключей и список (List) значений.

2.1. Коллекции — это наборы произвольных объектов

Общим свойством коллекций является то, что они работают с Object (базовый класс для всех классов Java). Это означает, что мы можем добавлять в коллекцию любые объекты Java, т.к. Object является базовым классом для всех классов Java. При извлечении из коллекции мы тоже получаем Object. Зачастую это приводит к необходимости преобразования полученного из коллекции объекта к нужному классу

(downcasting).

Это очень удобно для реализации самих коллекций, но не всегда хорошо при программировании. Если мы составляем коллекцию из объектов класса A и в результате ошибки поместим туда объект класса B, то при выполнении программы мы получим ClassCastException при попытке преобразования извлеченного из коллекции элемента к классу A (а потом будем долго искать, где же ошибка).

Идеальный вариант это, во-первых, получить сообщение об ошибке в точке ее возникновения, а не в точке ее проявления, а, во-вторых, во время трансляции программы, а не при ее выполнении.

Поэтому во многих случаях лучше построить на основе класса поддержки коллекции свой класс, который обеспечит нужный контроль типов во время трансляции. Например, пусть нам нужен список объектов класса MyType и мы решили использовать для его построения стандартный класс ArrayList (см. документацию). Тогда можно создать вспомогательный класс MyList такого вида:

```
public class MyList {
     private ArrayList v = new ArrayList();
     public void add(MyType obj) {
           v.add(obj);
     }
     public MyType get(int index) {
           return (MyType)v.get(index);
     }
     public int size() {
           return v.size();
     }
}
```

Класс MyList хорош тем, что в такой список нельзя добавить ничего, кроме объектов типа MyType, а извлеченные из списка объекты сразу приводятся к типу MyType внутри метода get().

Отступление . Этот пример интересен еще одним. Он демонстрирует случай, когда вместо наследования следует применять агрегацию. Действительно, если бы мы

наследовали от ArrayList, то остались бы доступными методы ArrayList, оперирующие с Object, а не с MyType и мы бы не получили желаемого эффекта.

3. Итераторы

В коллекциях широко используются *итераторы* . В Java итератор — это вспомогательный объект, используемый для прохода по коллекции объектов. Как и сами коллекции, итераторы базируются на интерфейсе. Это интерфейс *Iterator* , определенный в пакете java.util (см.документацию). Т.е. любой итератор, как бы он не был устроен, имеет следующие три метода:

- boolean hasNext() — проверяет есть ли еще элементы в коллекции
- Object next() — выдает очередной элемент коллекции
- void remove() — удаляет последний выбранный элемент из коллекции.

Кроме Iterator есть еще ListIterator — это расширенный вариант итератора с доп. возможностями и Enumerator — это устаревший вариант, оставленный для совместимости с предыдущими версиями.

В свою очередь интерфейс Collection имеет метод

Iterator iterator();

Это обязывает все классы коллекций создавать поддержку итераторов (обычно реализованы с использованием inner-классов, удовлетворяющих интерфейсу Iterator).

Рассмотрим теперь интерфейс List. В дополнение к методу iterator() он имеет метод

ListIterator listIterator();

Соответственно все коллекции-списки реализуют List-итераторы.

Вернемся к примеру с коллекцией из 100 случайных чисел в начале занятия. В нем плохо то, что строка, изображающая случайные числа, не форматирована и результат не выглядит удобочитаемым. Поставим себе задачу разбить эту строку на подстроки так, чтобы каждая из них содержала не более 6 чисел.

Для этого нам достаточно переписать метод toString().

```
public String toString() {
```

```
        String res = "";
        Iterator iter = lst.iterator();
        for(int i = 0; iter.hasNext(); i++) {
            if( i%6 == 0 )
                res += "\n";
            res += " " + iter.next().toString();   // !!!
        }
        return res;
    }
```

Внесем эти изменения, оттранслируем и запустим программу. Теперь числа выводятся в более удобочитаемом виде.

Разберем, как реализован метод toString(). Метод iterator() из ArrayList возвращает объект, ссылку на который мы запоминаем в переменной iter. Этот объект не интерфейса Iterator (нельзя построить объект интерфейса). Это объект некоторого класса, определенного внутри ArrayList, удовлетворяющего интерфейсу Iterator. Этот класс имеет свое имя, поля, возможно, какие-то private-методы. Но нас это не интересует. Мы просто приводим его к типу Iterator и используем как итератор.

- Чаще всего подобные классы строятся с использованием механизма вложенных классов.
- Этот пример, кроме всего прочего, демонстрирует случай, когда не требуется выполнять приведение типа после извлечения объекта из коллекции.

Рассмотрим строку:

res += " " + iter.next().toString();

Извлеченный из коллекции методом next() объект мы не приводим к типу Integer, а сразу применяем метод toString(). Здесь используется то свойство, что все классы имеют метод toString() (т.к. он есть в Object).

Метод remove()

В интерфейсе Iterator определен метод remove(). Он позволяет удалить из коллекции последний извлеченный из нее объект. В то же время, в интерфейсе Collection (а это значит — во всех классах-коллекциях) есть методы remove(Object o) и removeAll().

Здесь нет двойственности возможностей. Дело в том, что удаление из коллекции

объектов и вообще модификация коллекции параллельно с проходом по коллекции через итератор запрещено (выдается ConcurrentModificationException). Единственное, что разрешено — это удалить объект методом remove() из Iterator (причем, только один - последний извлеченный).

Интерфейс ListIterator расширяет эти возможности (см. метод set(Object o) интерфейса ListIterator).

4. Классы реализации коллекций

Рассмотрим конкретные реализации коллекций, т.е. классы, которые обеспечивают описанную выше функциональность. Мы будем рассматривать их в том же разрезе типов коллекций — List, Set, Map.

4.1. Коллекции-списки (List)

Реализованы в следующий трех вариантах ArrayList, LinkedList, Vector.

- Класс Vector — это устаревший вариант, оставлен для совместимости с предыдущими версиями.

Класс ArrayList — используется чаще всего. Имеет два конструктора:

```
public ArrayList()             // Конструктор пустого списка

public ArrayList(Collection c) // Строит список из любой коллекции
```

А также следующие методы:

```
public int size()

  Возвращает количество элементов списка


public boolean isEmpty()

  Проверяет, что список пуст
```

```
public boolean contains(Object elem)

  Проверяет, принадлежит ли заданный объект списку


public int indexOf(Object elem)

  Ищет первое вхождение заданного объекта в список и возвращает
  его индекс.  Возвращает -1, если объект не принадлежит списку.


public int lastIndexOf(Object elem)

  То же самое, но ищет последнее вхождение.


public Object clone()

  Создает "поверхностную" копию списка.


public Object[] toArray()

  Преобразует список в массив.


public Object get(int index)

  Возвращает элемент коллекции с заданным номером.


public Object set(int index,  Object element)

  Заменяет элемент с заданным номером.


public boolean add(Object o)

  Добавляет заданный объект в конец списка.
```

```
public void add(int index, Object element)

  Вставляет элемент в указанную позицию списка.


public Object remove(int index)

  Удаляет заданный элемент списка.


public void clear()

  Полностью очищает список.


public boolean addAll(Collection c)

  Добавляет к списку (в конец) все элементы заданной коллекции.


public boolean addAll(int index, Collection c)

  Вставляет в список с указанной позиции все элементы коллекции.


protected void removeRange(int fromIndex, int toIndex)

  Удаляет из коллекции объекты в заданном интервале индексов
  (исключая toIndex).
```

Класс **LinkedList** .

Он имеет практически ту же функциональность, что и ArrayList, и отличается от него только способом реализации и, как следствие, эффективностью выполнения тех или иных операций. Так добавление в LinkedList осуществляется в среднем быстрее, чем в ArrayList, последовательный проход по списку практически столь же эффективен, как у ArrayList, а произвольное извлечение из списка медленнее, чем у ArrayList.

Знакомство с другими типами коллекций мы продолжим на следующем занятии.

5. Практическая работа

Вернемся к домашнему заданию 9-го занятия. Правильная его реализация выглядит примерно так.

```
// Dialog9Home.java
// Пример визульного приложения на Java. Копирование из полей ввода

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

public class Dialog9Home extends JFrame {

  JTextField fldi = new JTextField(20);
  JTextField fldo = new JTextField(20);

  Dialog9Home() {
     super("Home Work (Lesson 9)");

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

    setSize(400, 150);
    Container c = getContentPane();
    JLabel lblin = new JLabel("Поле    ввода ");
    JLabel lblout= new JLabel("Поле вывода ");
    JButton btn = new JButton("Скопировать");
    JPanel pnlout = new JPanel();
    JPanel pnlin  = new JPanel();
    pnlin.add(lblin);
    pnlin.add(fldi);
    c.add(pnlin, BorderLayout.NORTH);
    pnlout.add(lblout);
```

```
    pnlout.add(fldo);
    c.add(pnlout, BorderLayout.CENTER);
    JPanel pnlBtn = new JPanel();
    pnlBtn.add(btn);
    c.add(pnlBtn, BorderLayout.SOUTH);
// Listener для поля ввода
    fldi.addActionListener(new ActionListener(){
          public void actionPerformed(ActionEvent e) {
              fldo.setText(fldi.getText());
              fldi.setText("");
          }
        });
// Listener для кнопки копирования.
    btn.addActionListener(new ActionListener(){
          public void actionPerformed(ActionEvent e) {
              fldo.setText(fldi.getText());
              fldi.setText("");
              fldi.requestFocus();
          }
        });
    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);

    setVisible(true);
  }

  public static void main(String[] args) {
    new Dialog9Home();
  }
}
```

1. Для размещения визуальных компонент здесь использованы 3 панели — pnlin, pnlout и pnlBtn. Именно это обеспечивает нормальный внешний вид приложения.

2. Вся функциональность данного приложения сосредоточена в двух "слушателях" (listener) — одного для поля ввода, другого для кнопки. Копирование текста

выполняется так

fldo.setText(fldi.getText());

После этого выполняется очистка поля ввода:

fldi.setText("");

6. Задание.

Изменить данное приложение следующим образом.

- **1.** При копировании поля ввода нужно кроме собственно копирования организовать занесение строки из поля ввода во внутренний список (список организовать, используя ArrayList или LinkedList).
- **2.** Добавить вниз экрана еще одну кнопку с надписью "Печатать". По нажатию на эту кнопку весь сохраненный список должен быть выведен на печать (в поток System.out).