Конспект лекций по Java. Занятие 9

В.Фесюнов

1. Практические навыки. Шаблон "синглетон" (singleton pattern)

На прошлом занятии мы рассмотрели спецификаторы доступа. Сейчас мы рассмотрим реальный пример использования спецификаторов доступа для построения синглетонов.

Синглетон — это один из многих стандартных шаблонов (**patterns**). Он обозначает класс, для которого можно создать ровно один объект. Если в программе требуется применение подобного класса, то имеет смысл создавать его по данному стандартному образцу.

Пусть нам требуется создать класс-синглетон SomeThing. Тогда это может быть реализовано так.

```
public class SomeThing {

    private static SomeThing instance = null;

    protected SomeThing () {
      . . .
    }

    public static final SomeThing getInstance() {
        if (instance == null) {
            instance = new SomeThing ();
        }
        return instance;
    }
```

```
    . . .
}
```

Здесь в классе описано статическое, недоступное извне (`private`) поле *instance* класса `SomeThing` , конструктор и метод `getInstance` . В классе могут (и должны) быть другие поля и методы.

Конструктор описан как `protected` , что не позволяет создавать объекты данного класса операцией `new` вне класса. Единственной возможностью получить объект данного класса является вызов метода `getInstance` . В свою очередь, этот метод обеспечивает создание единственного экземпляра объекта `SomeThing` .

- Если быть более точным, то конструктор класса можно вызвать из классов пакета и из порожденных классов. Если не предполагается наследовать данный класс, то конструктор можно описать как `private` .

Следует отметить, что работать с таким классом очень легко. В любом месте программы вызов "SomeThing.getInstance()" обеспечит получение ссылки на единственный объект данного класса. Это позволяет, во многих случаях, избежать передачи и хранения перекрестных ссылок объектов друг на друга.

Возьмем данный пример себе на заметку и продолжим знакомиться с возможностями Java.

2. Вложенные классы

Java позволяет создавать классы внутри классов. Т.е. текстуально описание класса может находиться внутри фигурных скобок тела другого класса. Такие классы называются **вложенными (inner)** .

Аппарат вложенных классов в Java имеет много деталей, он хорошо проработан и весьма широко используется при программировании. При помощи вложенных классов строится **framework** (инфраструктура) охватывающего класса. Например, если в классе нужно использовать другой класс и этот класс используется только в этом классе и более ни к кому другому классу отношение не имеет, то имеет смысл этот класс сделать вложенным. Это, правда, не единственный случай, когда имеет смысл строить вложенные классы. Но для того, чтобы понять, для чего они используются

нужно сначала хотя бы немного с ними познакомиться.

Вложенный класс может быть помещен как на уровень описания охватывающего класса, так и внутрь некоторого метода охватывающего класса, а также внутрь некоторого блока.

Пример (файл Outer.java)

```
public class Outer {
    int y;
    Outer(int x) {
         y = x;
    }

    class Inner1 {    // Вложенный класс. Размещен внутри класса Outer
         . . .
    }

    int g(String s) {
       class Inner2 { // Вложенный класс. Размещен внутри метода g класса Outer

          . . .
       }
       . . .
    }
    . . .
}
```

Здесь внутрь класса Outer помещены два других класса — Inner1 и Inner2. Причем, класс Inner1 помещен непосредственно внутрь Outer (его описание находится на том же уровне вложенности, что и методы класса Outer). А класс Inner2 помещен внутрь метода g(...) данного класса.

При компиляции Outer.java будут образованы три class-файла: Outer.class, Outer$Inner1.class и Outer$Inner2.class.

В отличие от обычных классов для вложенных классов имеют смысл описатели `private` и `protected` для самого класса. Вложенный класс с описателем `private` доступен только внутри охватывающего класса, `protected` — внутри

охватывающего класса и классов, порожденных от охватывающего.

Эти описатели имеют смысл только для inner-классов, описанных на уровне класса. Вложенные классы, описанные внутри методов или блоков, все равно доступны только внутри этих методов или блоков.

Объекты inner-классов могут порождаться как внутри охватывающего класса, так и извне его. Для того чтобы можно было явно создать объект inner-класса извне охватывающего класса, он должен быть описан с описателем `public` или вообще без описателя. Кроме того, конструктор вложенного класса также должен быть с описателем `public` или без описателя.

При порождении объекта inner-класса из охватывающего класса синтаксис обычный:

```
     Inner1 i1 = new Inner1();
```

При порождении объекта inner-класса извне охватывающего класса нужно чтобы inner-класс был доступен извне и, кроме того, чтобы существовал объект охватывающего класса.

Пусть t1 — объект класса Outer. Тогда порождение объекта класса Inner1 выглядит так:

```
     Outer.Inner1 i2 = t1.new Inner1();
```

Дело в том, что объект inner-класса имеет свободный доступ ко всем полям и методам объекта охватывающего класса. Это обеспечивается тем, что он содержит в себе неявную ссылку на объект охватывающего класса.

Так в приведенном выше примере класс Inner1 может содержать такой метод:

```
public int f1() {
     return y++;
}
```

Здесь y — поле класса Outer. Оно совершенно свободно доступно изнутри вложенного класса. С другой стороны, объекты классов Outer и Inner1 — это два совершенно разных объекта, размещенных в разных областях памяти. Именно поэтому любой

объект класса Inner1 должен хранить ссылку на объект класса Outer. Эта ссылка может быть использована в программе и явно (это, правда, очень редкий случай). Синтаксис такого использования на примере:

```
Outer ref = Outer.this;
   // Здесь Outer.this — ссылка на объект охватывающего класса.
```

В частности, метод f1() мог быть написан и так

```
public int f1() {
     return Outer.this.y++;
}
```

2.1. Классы, вложенные в методы или блоки

Рассмотрим особенности, связанные с классами, вложенными в методы или блоки. С такими классами связаны дополнительные возможности и ограничения.

Во-первых, как уже отмечалось, такой класс доступен только внутри данного метода или блока.

Во-вторых, он имеет доступ к локальным переменным, а также параметрам данного метода или блока, но с одним ограничением — эти переменные и/или параметры должны быть описаны как final.

Рассмотрим пример.

```
public class Outer {

   int x = 0;

   class Inner1 {
       public int f() {
           return x++;
       }
   }
```

```
   public void g(final int j) {
     final int k = j;
     class Inner2 {

       public int f(int r) {
           return r*k + j;        // Здесь и j и k должны быть final
       }
     }
     Inner2 i1 = new Inner2();
     int s = i1.f(10);
   }
}

class Outside {

   public void g1() {
       Outer t1 = new Outer();
       Outer.Inner1 i2 = t1.new Inner1();
       i2.f();
   }
}
```

Это совершенно абстрактный пример, но он демонстрирует различные синтаксические конструкции с использованием inner-классов.

Здесь описаны два обычных класса — Outer и Outside, и два вложенных — Inner1 и Inner2.

- Класс Outer имеет поле x и метод g(...).
- Класс Outside имеет единственный метод — g1(...).
- Класс Inner1 описан внутри Outer на уровне самого класса и имеет метод f(). Внутри этого метода продемонстрировано обращение к полю x класса Outer.
- Класс Inner2 описан внутри метода g(...) класса Outer. Он имеет метод f(...), в котором идет обращение к переменной k, параметрам r и j. Параметр r является обычным параметром данного метода, но k и j — это внешние по отношению к классу Inner2 переменные. Поэтому они должны быть описаны как final.
- В методе g(...) класса Outer продемонстрировано создание и использование объекта класса Inner2. При этом используется обычный синтаксис создания объекта класса.

Нигде за пределами метода g(...) явно использовать Inner2 нельзя.

- В методе g1() класса Outside демонстрируется внешнее (по отношению к охватывающему классу Outer) использование класса Inner1.

Вложенные классы, как и обычные классы, могут порождаться (наследоваться) от других классов и/или удовлетворять интерфейсам. Также они могут использоваться в качестве базовых классов при построении новых классов (это используется крайне редко).

2.2. Применение inner-классов

Рассмотрим пример применения inner-класса. Предположим, мы пишем приложение, работающее с плоскими геометрическими фигурами. Один из возможных способов реализации состоит в следующем. Мы определяем интерфейс Shape (фигура), который определяет, что мы можем делать с фигурами. Потом мы определяем класс ShapeFactory с методами createCircle(...), createTriangle(...) и т.д. Для создания окружностей и треугольников мы внутри класса ShapeFactory описываем inner-классы Circle и Triangle.

Файл Shape.java

```
public interface Shape {
   . . .
}
```

Файл ShapeFactory.java

```
public class ShapeFactory {

    private class Triangle implements Shape {
           . . .
          Triangle(float a, float b, float c) {
             . . .
           }
           . . .
     }
```

```
     private class Circle implements Shape {
           . . .
          Circle (float r) {
             . . .
           }
           . . .
     }

      public Shape createTriangle(float a, float b, float c) {
            return new Triangle(a, b, c);
      }

      public Shape createCircle(float r) {
            return new Circle(r);
      }
      . . .
}
```

Что мы видим, так сказать, извне класса ShapeFactory?

Классы Triangle и Circle объявлены как private и потому недоступны. Т.е. доступными для внешнего использования являются только методы createTriangle(...) и createCircle(...). Эти методы порождают объекты Triangle и Circle, но типом возвращаемого значения у них вляется Shape (при выполнении return выполняется upcasting). Такое возможно благодаря тому, что и Triangle и Circle удовлетворяют интерфейсу Shape.

Таким образом, мы можем получить как треугольники, так и окружности, но дальше мы можем работать с ними только как с абстрактными фигурами, используя методы, определенные в интерфейсе Shape. Например.

```
ShapeFactory f = new ShapeFactory();  // Создаем объект ShapeFactory
Shape f1 = f.createCircle(10);
Shape f2 = f.createTriangle(2, 3, 4);
f1.g(6);   // Здесь подразумевается, что g(int) метод интерфейса Shape
```

В свою очередь классы Triangle и Circle должны содержать реализацию всех методов интерфейса Shape.

2.3. Анонимные вложенные классы

Анонимные классы — это классы без имени. Они являются подмножеством вложенных классов. Т.е. анонимный класс — это всегда вложенный класс. Анонимные классы используются весьма широко, возможно, даже чаще чем обычные вложенные классы. Дело в том, что они дают удобную возможность, позволяющую сократить размер кода программы. Собственно говоря, в этом их основное преимущество.

Вместо того, чтобы сначала описывать класс, а потом создавать объект этого класса, мы с помощью анонимных классов можем сделать это одновременно (два в одном ;-)).

Рассмотрим это на примере. Пусть есть класс A и мы хотим создать на его базе новый класс, а также создать объект этого класса. В обычном случае это выглядит примерно так. Мы где-то описываем класс B, порожденный от A, а потом в нужной точке программы создаем объект класса B:

```
// Описание класса B
class B extends A {
   . . . описание полей и методов порожденного класса . . .
}


// Использование класса B
A a = new B();
```

Применяя аппарат анонимных классов это можно записать гораздо короче и, что самое главное, одним фрагментом кода:

```
A a = new A() {
        . . . описание полей и методов порожденного класса . . .
        };
```

Естественно с анонимными классами связано ряд ограничений. Так анонимный класс не может иметь конструктора (конструктор по определению — это метод с именем, совпадающим с именем класса, а анонимный класс имени не имеет). Из-за этого возникают две проблемы. Первая — где производить инициализирующие действия,

вторая — как вызвать не конструктор по умолчанию базового класса, а конструктор с параметрами.

Решение первой проблемы состоит в том, что все инициализирующие действия можно выполнить либо при помощи явной инициализации, либо в специальном блоке инициализации.

Расширим предыдущий пример.

```
A a = new A() {
          int x = 0;
          String name;

          {       // Это блок инициализации
              name = ...;
           }
          . . . описание полей и методов порожденного класса . . .
          };
```

- Блок инициализации допустим не только в анонимных, но и в обычных классах. Но там он не особенно нужен, поскольку есть конструктор.

Вторая проблема решается при помощи указания параметров в операции new:

```
A a = new A(10) {    // Вызывается конструктор базового класса A с одним параметром
          . . .
          . . . описание полей и методов порожденного класса . . .
          . . .
          };
```

Еще одно ограничение связано с тем, что анонимный вложенный класс всегда вложен в некоторый метод. Следовательно, он может обращаться к переменным, доступным в этой точке (в точке его описания), но тогда эти переменные должны быть final.

Для анонимных вложенных классов, как и для обычных вложенных классов, при трансляции java-файла порождается отдельный class-файл. Имена этих class-файлов строятся при помощи последовательной нумерации. Так, если предыдущий фрагмент с порождением класса на базе класса A находился внутри класса X, то создался бы файл

X$1.class.

3. Практическая работа

3.1. Применение анонимных вложенных классов в рассмотренных ранее примерах.

Анонимные вложенные классы очень часто используются при построении Listener'ов (слушателей). Мы будем подробнее рассматривать событийную модель позже, но пример ее использования нам уже встречался в программах Dialog1 - Dialog3. Сейчас мы вернемся к этим примерам и продвинемся дальше в понимании тех моментов, которые мы изначально оставили за рамками рассмотрения.

```
// Dialog3.java
// Визульное приложения с текстовой областью.

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

public class Dialog3 extends JFrame {

  JTextArea txt;

  Dialog3() {
     super("Визульное приложения с текстовой областью");

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

    setSize(400, 200);
    Container c = getContentPane();
    c.add(new JLabel("Hello, привет"), BorderLayout.NORTH);
    txt = new JTextArea(5, 30);
```

```
    JScrollPane pane = new JScrollPane(txt);
    c.add(pane, BorderLayout.CENTER);

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);

    setVisible(true);
  }

  public void test() {
    txt.append("Первая строка\n");
    txt.append("Вторая строка\n");
  }

  public static void main(String[] args) {
    Dialog3 d = new Dialog3();
    d.test();
  }
}
```

Это демонстрационная программа Dialog3, которую мы рассматривали на 7-м занятии. Интересующий нас фрагмент это

```
    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);
```

Здесь порождается анонимный класс на базе класса *WindowAdapter* и создается объект (*wndCloser*) этого класса. Потом этот объект передается в метод *addWindowListener(...)* в качестве параметра. Пока еще остается непонятным, что это дает и как все это работает, но формально уже ясно, что здесь используется механизм

анонимных классов.

3.2. Слушатели (listeners) полей и кнопок.

Изучение inner-классов и, в частности, анонимных классов дает нам возможность продвинуться дальше в знакомстве с возможностями визуальной среды Java — с пакетом Swing.

Рассмотрим такие элементы библиотеки **Swing** как класс **JButton** и класс **JTextField** .

Класс **JButton** предназначен для создания кнопок. Его возможности и, так сказать, широта использования весьма значительны. Но сейчас мы рассмотрим только стандартные средства его использования по принципу "делай как я".

Посмотрим документацию по классу `JButton` и обратим внимание на его конструктор

public JButton(String text)

Создает кнопку с надписью. Текст надписи задан параметром.

А также на метод

public void addActionListener(ActionListener l)

Добавляет слушателя (listener) типа `ActionListener` к данной кнопке

Этот метод унаследован классом `JButton` от класса **AbstractButton** .

Если, в свою очередь, посмотреть на описание **ActionListener** , то можно увидеть, что это интерфейс из пакета **java.awt.event** и что этот интерфейс имеет единственный метод

public void actionPerformed(ActionEvent e)

Т.е. для того, чтобы добавить слушателя, мы должны создать класс, удовлетворяющий интерфейсу ActionListener, и создать объект этого класса (тут нам как раз пригодятся анонимные классы).

Смысл всего этого в том, что при нажатии на кнопку будет вызван метод **actionPerformed(...)** , где мы можем запрограммировать все необходимые нам

действия.

Это все, что нам сейчас нужно по классу JButton.

Класс **JTextField** используется для создания полей ввода. Наиболее часто используемый конструктор это

public JTextField()

Создает поле ввода

Как и JButton он имеет метод **addActionListener(...)** . Отличие от JButton заключается в том, что метод **actionPerformed(...)** в данном случае вызывается при выходе из поля ввода по клавише Enter.

Рассмотрим абстрактный пример Dialog4.java. В этом примере есть кнопка, поле ввода и метка (`JLabel` , рассматривали ранее), которая используется для отображения результатов наших действий над кнопкой и полем ввода.

```
// Dialog4.java
// Пример визульного приложения на Java.

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

public class Dialog4 extends JFrame {

  JTextField fld = new JTextField();
  JButton btn = new JButton("Нажать нежно");
  JLabel lbl = new JLabel(" ");

  Dialog4() {
     super("Слушатели (listeners) полей и кнопок");

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

```
    setSize(400, 150);
    Container c = getContentPane();
    c.add(lbl, BorderLayout.NORTH);
    c.add(fld, BorderLayout.CENTER);
    c.add(btn, BorderLayout.SOUTH);
    fld.addActionListener(new ActionListener(){
          public void actionPerformed(ActionEvent e) {
              lbl.setText("Введен текст:"+fld.getText());
          }
        });
    btn.addActionListener(new ActionListener(){
          public void actionPerformed(ActionEvent e) {
              lbl.setText("Нажата кнопка");
          }
        });
    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);

    setVisible(true);
  }

  public static void main(String[] args) {
    new Dialog4();
  }
}
```

В этом примере в класс Dialog4 мы поместили три поля fld, btn и lbl, которые предназначены для создания поля ввода, кнопки и метки, соответственно. Строки

```
    c.add(lbl, BorderLayout.NORTH);
    c.add(fld, BorderLayout.CENTER);
    c.add(btn, BorderLayout.SOUTH);
```

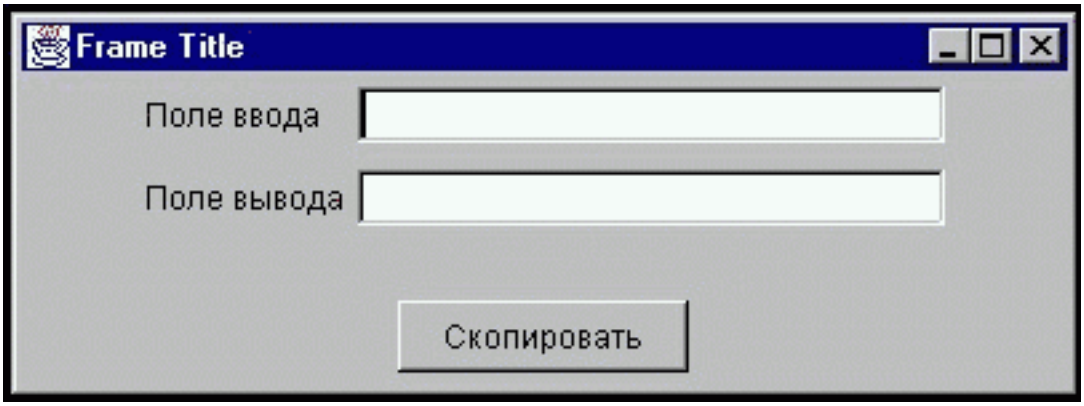
предназначены для размещения указанных визуальных компонент на экране. Далее

следует создание слушателей поля и кнопки. Оба слушателя выводят некоторую информацию в метку lbl, для чего используется метод setText(...) класса JLabel. Кроме того, для выборки текста, введенного в поле fld, в примере использован метод getText() класса JTextField.

Оттранслируем и запустим данную программу.

4. Задания

- **1.** В примере Dialog4 fld, btn и lbl реализованы как поля класса. Разобраться почему и определить какие из них должны быть полями класса, а какие могут быть сделаны локальными переменными конструктора.
- **2.** Поле ввода в примере занимает всю центральную область окна, что не очень хорошо. Улучшить внешний вид можно несколькими способами, но наиболее простой состоит в применении дополнительной панели. Панель — это контейнер визуальных компонент, в свою очередь она тоже является визуальной компонентой. Т.е. мы можем добавить компоненты на контейнер, а потом контейнер добавить на окно приложения. Для построения панелей в Swing имеется класс JPanel.
  Улучшим внешний вид приложения следующим образом. Поместим поле не непосредственно в центр основного окна, а создадим панель JPanel, поместим поле на эту панель, а уже панель — в центр окна. Кроме того, для создания поля нужно использовать конструктор с параметром int columns:
  ```
  JTextField fld = new JTextField(20);
  ```
- **3.** Аналогичные действия произведем с кнопкой btn, поместив ее на отдельную панель.
-



Для поля ввода создается и метка (JLabel) и поле ввода (JTextField), поэтому их нужно сгруппировать вместе в одну визуальную компоненту (используем JPanel). Аналогично и для поля вывода. Для размещения двух панелек и кнопки (JButton) используем BorderLayout.NORTH, BorderLayout.CENTER и BorderLayout.SOUTH, соответственно.