Конспект лекций по Java. Занятие 22

В.Фесюнов

1. Идентификация типа во время выполнения. Рефлексия

Это наиболее сложная, но, в то же время, наиболее редкоиспользуемая часть RTTI.

Рефлексия (`reflection`) редко используется при разработке обычных приложений. В то же время, она является важным свойством Java, позволяющим строить очень гибкие и высокоинтеллектуальные технологии. Так, рефлексия является одним из базовых механизмов, лежащем в основе реализации технологии **JavaBeans** , которая позволяет разрабатывать многократно-используемые программные компоненты.

Можно назвать много реальных и возможных областей использования рефлексии (например, те же Интеллектуальные Пакеты Прикладных Программ **ИППП**). Но на данном этапе знакомства с Java, по моему мнению, не столько важно научиться использовать рефлексию, сколько определить для себя, что это такое, какие возможности она имеет. Т.е. представить себе общую картину. Это, во-первых, поможет, при необходимости, изучить ее глубже, а, во-вторых, это просто необходимо для понимания того, как работает технология JavaBeans, которую мы будем рассматривать далее.

1.1. Что такое рефлексия

Это набор средств для получения полной и исчерпывающей информации о каком-либо классе во время выполнения программы.

Она дает возможность в некоторой программе работать с классами, которые не были известны разработчику этой программы в момент ее написания.

Чаще всего, рефлексия применяется совместно с некоторыми соглашениями. Например, мы уже говорили о том, что поля класса не принято делать открытыми (`public`), а для доступа к ним рекомендуется создавать в классе get- и set-методы, сигнатура которых (название, параметры, возвращаемые значения) строится по определенным правилам. Поля, удовлетворяющие этим соглашениям, называют *атрибутами класса* . Используя рефлексию можно получить весь список методов некоторого класса, выбрать из них get- и set-методы и, таким образом, получить список всех атрибутов класса. В свою очередь, это может быть использовано для визуального представления атрибутов класса или для задания/получения значений атрибутов экземпляра этого класса.

1.2. Знакомство с механизмом рефлексии

Механизмы рефлексии собраны в одном пакете `java.lang.reflect` . Но исходная информация для рефлексии сосредоточена в классе `Class` .

Посмотрим документацию по `Class` . Обратим внимание на методы `getConstructors` и `getMethods` . Первый из них возвращает массив объектов класса `java.lang.reflect.Constructor` , по одному объекту для каждого конструктора класса. Второй — аналогичный массив объектов класса `java.lang.reflect.Method` , по одному — для каждого метода класса.

Во-первых, нужно отметить, что данные методы возвращают информацию только о общедоступных (`public`) конструкторах или методах. Во-вторых, (это относится толко к `getMethods` , т.к. конструкторы не наследуются) в массив входят не только методы, объявленные в классе, но и унаследованные от суперклассов данного класса.

Есть два других метода, похожих на описанные. Это — `getDeclaredConstructors` и `getDeclaredMethods` . Они отличаются тем, что выдают информацию не только по `public` -методам или конструкторам, а по всем, вне зависимости от их спецификаторов доступа. Кроме того, `getDeclaredMethods` выдает информацию только о тех методах, которые описаны непосредственно в данном классе, методы, унаследованные от суперклассов, в результат не входят.

Далее, есть методы для получения информации о полях класса — `getFields` и `getDeclaredFields` , о суперклассах данного класса — `getClasses` и

`getDeclaringClass` , о реализуемых данным классом интерфейсах — `getInterfaces` .

Все указанные методы, кроме `getDeclaringClass` , возвращают массив объектов, в частности, об этом говорит множественное число в названии этих методов. Но есть аналогичные методы, позволяющие получить информацию по конкретному конструктору, методу, полю. Эти методы не имеют множественного в своем названии (например, `getMethod` или `getDeclaredConstructor`) и имеют параметры, позволяющие уточнить, какой конкретно конструктор или метод нас интересует.

Есть и другие полезные методы класса `Class` , которые мы опустим, чтобы не загромождать изложение.

Если разбираться с возможностями рефлексии дальше, то следует перейти к рассмотрению классов `Constructor` , `Method` , `Field` пакета `java.lang.reflect` .

Обратимся, к примеру, к документации по классу `Method` . Здесь можно увидеть, что объект такого класса позволяет узнать много дополнительной информации по конкретному методу класса. Например, мы можем узнать его имя (метод `getName`), тип возвращаемого значения (метод `getReturnType`) или список параметров (метод `getParameterTypes`).

Мы даже можем вызвать этот метод при помощи метода `invoke` . При этом мы в качестве параметров должны указать объект, для которого мы вызываем данный метод и массив параметров вызова данного метода. Причем, объект должен быть объектом того класса, к которому принадлежит данный метод, а массив параметров должен содержать столько элементов, сколько параметров требуется данному метод и они должны соответствовать по типам параметрам данного метода.

1.3. Демонстрационный пример

Рассмотрим пример, демонстрирующий возможности рефлексии. Этот пример не очень сложен. В нем не используются развитые возможности рефлексии, заложенные в классах пакета `java.lang.reflect` . Данный пример позволит просто вывести информацию о методах того или иного класса на экран с помощью метода `toString` класса `Method` . Пример можно легко расширить, чтобы он выдавал также

информацию о конструкторах и полях класса, применяя метод `toString` соответствующих классов.

Файл **ReflectDemo.java** .

```
// ReflectDemo.java
// Демонстрация работы рефлексии

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

public class ReflectDemo extends JFrame {

    private JTextField fld = new JTextField(30);
    private JTextArea  msg = new JTextArea(8, 40);

    ReflectDemo() {
     super("Демонстрация работы рефлексии");

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

     setSize(600, 350);
     Container c = getContentPane();
     JPanel pn = new JPanel();
     c.add(pn, BorderLayout.NORTH);
     pn.add(new Label("Имя класса"));
     pn.add(fld);
     fld.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent evt) {
             reflectInfo();
         }
     });
     JScrollPane pane = new JScrollPane(msg);
     c.add(pane, BorderLayout.CENTER);
```

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

     setVisible(true);
    }

    void reflectInfo() {
        String className = fld.getText();
        try {
            Class cls = Class.forName(className);
            Method[] m = cls.getMethods();
            int lastLine = msg.getLineCount()-1;
            if ( lastLine >= 0 )
                msg.replaceRange("", 0, msg.getLineEndOffset(lastLine));
            msg.append("Методы класса "+className+":\n");
            for(int i = 0; i < m.length; i++)
                msg.append(m[i].toString()+"\n");
        } catch (Exception ex) {
            msg.append("Ошибка при поиске класса "+className+":\n");
            msg.append(ex.toString()+"\n");
        }
    }

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

}
```

1.3.1. Пояснения к примеру

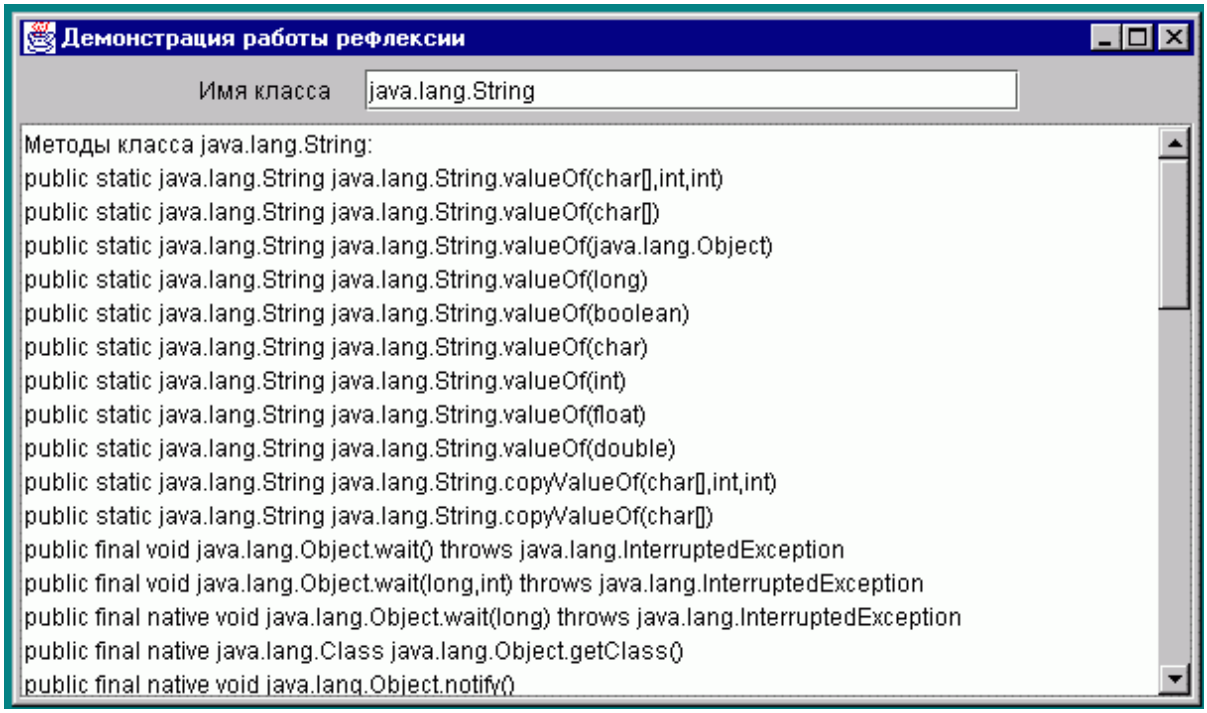
В данном примере экран формируется точно также, как и в примере **NewInstanceDemo.java** , который мы рассматривали на прошлом занятии. Вместо метода `createObject` в данном примере в слушателе поля ввода вызывается метод `reflectInfo` .

Все содержательные действия программы выполняются данным методом. Сначала он выбирает имя класса из текстового поля в переменную `className` , потом вызывает метод `forName` для получения объекта класса `Class` . Получив этот объект, программа вызывает для него метод `getMethods` , который возвращает массив всех `public` -методов данного класса. Строки

```
            int lastLine = msg.getLineCount()-1;
            if ( lastLine >= 0 )
                msg.replaceRange("", 0, msg.getLineEndOffset(lastLine));
```

предназначены для очистки текстовой области от предыдущих результатов. После этого в цикле для всех методов класса вызывается метод `toString` и его результаты добавляются в конец текстовой области методом `append` .

Оттранслируем и запустим данный пример. Ниже на рис. представлены результаты работы нашего приложения. В качестве имени класса здесь было введено "java.lang.String".



1.4. Заключение

Данная тема носила обзорный характер. Поэтому мы не будем решать какие-либо примеры по рефлексии.

Мы познакомились с основными возможностями рефлексии и определили, что можно сделать, используя рефлексию, и, примерно, какими способами это делать. На этом мы завершим рассмотрение рефлексии.

2. Множественные нити выполнения (Multiple threads)

Все современные операционные системы (ОС) работают в мультипрограммном режиме, когда параллельно на одном компьютере выполняются много различных программ. ОС организует переключение между ними, деля время процессора (или процессоров) между различными программами. Все это происходит, так сказать, за сценой. А для пользователя это выглядит как параллельное выполнение нескольких программ одновременно.

Существует два под-варианта реализации мультипрограммного режима работы — при помощи задач (многозадачная или мультизадачная обработка) и при помощи нитей или потоков (мультипотоковая обработка). При этом они могут сосуществовать параллельно. Не вдаваясь в подробности, можно сказать, что мультизадачная обработка более громоздкая и сложная, но имеет больше возможностей. Каждой задаче выделяется свое пространство памяти. В отличии от задач, нити более легковесны и работают все на одном пространстве памяти.

- Термин *thread* можно перевести и как поток, и как нить. Но в английском есть термин *stream* , который тоже переводится, как поток и применяется тогда, когда речь идет о потоках ввода/вывода. Поэтому, для однозначности, для *thread* мы будем использовать перевод *нить* . В то же время, термин *мульти-какая-то-там-нитяная обработка* в русском языке сформировать нельзя. Поэтому мы будем использовать *мультипотоковая обработка* . Неоднозначность здесь не возникает, т.к. к потокам ввода/вывода приставка *мульти-* обычно не применяется.

Каждый из этих режимов имеет свое применение. В рамках конкретной ОС

существуют те или иные верхнеуровневые механизмы создания задач. Так, запуская программу в Windows, мы тем самым запускаем задачу. Параллельно мы можем запустить другие программы. Для запуска нитей также существуют механизмы операционной системы. Но нити — это параллельные процессы в рамках одной программы. Соответственно, никаких верхнеуровневых (доступных обычному пользователю) средств для запуска нитей нет и быть не может. Эти механизмы могут быть доступны только в рамках той или иной системы программирования.

В Java реализована полная поддержка нитей. Более того, сама стандартная библиотека Java во многом опирается на мультипотоковую обработку. И многие программы, даже не использующие явно средства организации нитей, тем не менее, работают в мультипотоковой среде. Это, в частности, относится ко всем диалоговым программам. Как пакет AWT, так и пакет Swing существенно используют мультипотоковую обработку.

2.1. Демонстрационный пример

Рассмотрим демонстрационную программу. В ней мы увидим нити в действии. Этот пример показывает, что обычная диалоговая программа уже работает в мультипотоковой среде.

```
// ThreadExample1.java

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

public class ThreadExample1 extends JFrame {

  JTextField txt1 = new JTextField(10);
  JTextField txt2 = new JTextField(10);
  int randValue = 0;
  long numbOfRand = 0;

  ThreadExample1() {
     super("Знакомство с нитями");
```

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

setSize(300, 200);
Container c = getContentPane();
JPanel pnm = new JPanel(new GridLayout(2, 1, 5, 5));
c.add(pnm, BorderLayout.CENTER);
JPanel pn1 = new JPanel(new FlowLayout(FlowLayout.CENTER, 5, 5));
JPanel pn2 = new JPanel(new FlowLayout(FlowLayout.CENTER, 5, 5));
pnm.add(pn1);
pnm.add(pn2);
pn1.add(new JLabel("Номер числа        "));
pn1.add(txt1);
txt1.setEnabled(false);
pn2.add(new JLabel("Случайное число"));
pn2.add(txt2);
txt2.setEnabled(false);
JPanel pnb = new JPanel();
JButton btn = new JButton("Показать число");
pnb.add(btn);
c.add(pnb, BorderLayout.SOUTH);
btn.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      txt2.setText(String.valueOf(randValue));
      txt1.setText(String.valueOf(numbOfRand));
    }
});

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

setVisible(true);
```

```
    Random rnd = new Random();
    for(;;numbOfRand++) {
      randValue = rnd.nextInt();
      try {
         Thread.currentThread().sleep(1000);
      } catch(InterruptedException e) {
      }
    }
  }

  public static void main(String[] args) {
    new ThreadExample1();
  }
}
```

Оттранслируем и запустим этот пример. Нажимая на кнопку "Показать число" с неравными интервалами времени мы увидим, что "за кадром" идет независимый процесс порождения случайных чисел.

Теперь разберемся с самой программой. В тексте программы не использовано никаких специальных средств порождения нитей.

Это обычная диалоговая программа. В конструкторе формируется внешний вид экрана — создаются и компонуются панели, метки, поля, кнопки. Для кнопки `btn` создается слушатель, который выводит в поля `txt1` и `txt2` текущие значения переменных `randValue` и `numbOfRand` . Потом создается слушатель для закрытия окна и, наконец, вызывается setVisible(true), который визуализирует экран нашего приложения. Далее идет цикл генерации случайных чисел с интервалом в одну секунду.

Т.е. никаких нитей мы явно не создаем и, тем не менее работа приложения показывает, что в программе выполняются параллельные нити — приложение нормально реагирует на нажатие кнопки пользователем, и одновременно работает цикл генерации случайных чисел.

Секрет в данном случае черезвычайно прост. Для реакции на действия пользователя, перерисовки экрана и других внутренних действий создается отдельная нить. Она называется *Нить Диспетчеризации Событий* , по английски — *Event Dispatching*

Thread . Вторая нить — это основная нить нашей программы. Она была инициирована JVM и в ней был вызван метод `main` нашего класса.

Если мы вернемся к примерам диалоговых программ, рассмотренным ранее, и внимательно их проанализируем, то увидим, что во всех этих программах основная нить выполнения заканчивает свою работу после построения и визуализации экранной формы. Тем не менее программа продолжает свою работу.

- Программа на Java продолжает работу пока активна хотя бы одна ее нить. Исключение составляют нити-демоны (daemons). Если все нити, кроме демонов, завершились, программа завершается.

2.2. Как организовать нить

Простейшим способом создания нити является создание класса производного от класса `Thread` . В этом классе нужно переопределить метод `run` , потом породить объект созданного класса и вызвать его метод `start` . С этого момента начнет свое существование новая нить. Она будет выполняться параллельно с другими нитями, конкурируя с ними за время процессора. Метод `start` выполняет системные действия по созданию нити, после чего вызывает переопределенный нами метод `run` . Когда произойдет выход из метода `run` нить завершит свою работу.

Расширим наш пример. Добавим в верхний правый угол экрана часы (поле, отображающее текущее время).

```
// ThreadExample2.java:

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

public class ThreadExample2 extends JFrame {

  JTextField txt1 = new JTextField(10);
  JTextField txt2 = new JTextField(10);
  JTextField txtTime = new JTextField(18);
  int randValue = 0;
```

```
long numbOfRand = 0;

ThreadExample2() {
   super("Знакомство с нитями (часы)");

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

  setSize(400, 300);
  Container c = getContentPane();
  JPanel pnm = new JPanel(new GridLayout(2, 1, 5, 5));
  c.add(pnm, BorderLayout.CENTER);
  JPanel pn1 = new JPanel(new FlowLayout(FlowLayout.CENTER, 5, 5));
  JPanel pn2 = new JPanel(new FlowLayout(FlowLayout.CENTER, 5, 5));
  pnm.add(pn1);
  pnm.add(pn2);
  pn1.add(new JLabel("Номер числа        "));
  pn1.add(txt1);
  txt1.setEnabled(false);
  pn2.add(new JLabel("Случайное число"));
  pn2.add(txt2);
  txt2.setEnabled(false);
  JPanel pnb = new JPanel();
  JButton btn = new JButton("Показать число");
  pnb.add(btn);
  c.add(pnb, BorderLayout.SOUTH);
  btn.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        txt1.setText(String.valueOf(numbOfRand));
        txt2.setText(String.valueOf(randValue));
      }
  });
  JPanel pntop = new JPanel(new FlowLayout(FlowLayout.RIGHT, 5, 5));
  pntop.add(txtTime);
  txtTime.setEnabled(false);
  txtTime.setEditable(false);
  c.add(pntop, BorderLayout.NORTH);
```

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

    setVisible(true);

    class SimpleThread extends Thread {
      public void run() {
        while ( true ) {
          Date dt = new Date();
          txtTime.setText(dt.toString());
          try {
             Thread.sleep(200);
          } catch(InterruptedException e) {
          }
        }
      }
    }

    SimpleThread sth = new SimpleThread();
    sth.start();

    Random rnd = new Random();
    for(;;numbOfRand++) {
      randValue = rnd.nextInt();
      try {
         Thread.sleep(1000);
      } catch(InterruptedException e) {
      }
    }
  }

  public static void main(String[] args) {
    new ThreadExample2();
  }
}
```

Здесь построен вложенный класс `SimpleThread` , метод `run` которого выводит в поле `txtTime` текущие дату и время 5 раз в секунду. После описания класса стоит создание объекта этого класса и запуск нити при помощи метода `start` .

2.3. С нитями выполнения нужно быть осторожным

Программирование с использованием нитей требует большего внимания и осторожности. Так, если в приведенном выше примере поменять местами создание и запуск нити и цикл генерации случайных чисел, то дата и время не будут выводиться, т.к. до фрагмента кода создания новой нити управление никогда не дойдет. Правда, простая перестановка указанных фрагментов не пройдет - компилятор Java "достаточно умен" для того, чтобы обнаружить эту ошибку. Но, если вынести фрагмент генерации случайных чисел в отдельный метод, то тут компилятор уже ошибку не обнаружит.

Для того, чтобы убедится, что работая с нитями легко допустить ошибку, рассмотрим следующий пример.

```
// ThreadExample3a.java

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

public class ThreadExample3a extends JFrame {

  JTextField txt1 = new JTextField(10);
  JTextField txt2 = new JTextField(10);
  JTextField txtTime = new JTextField(18);
  int randValue = 0;
  long numbOfRand = 0;

  ThreadExample3a() {
     super("Знакомство с нитями (часы)");

     try  {
```

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

    setSize(400, 300);
    Container c = getContentPane();
    JPanel pnm = new JPanel(new GridLayout(2, 1, 5, 5));
    c.add(pnm, BorderLayout.CENTER);
    JPanel pn1 = new JPanel(new FlowLayout(FlowLayout.CENTER, 5, 5));
    JPanel pn2 = new JPanel(new FlowLayout(FlowLayout.CENTER, 5, 5));
    pnm.add(pn1);
    pnm.add(pn2);
    pn1.add(new JLabel("Номер числа        "));
    pn1.add(txt1);
    txt1.setEnabled(false);
    pn2.add(new JLabel("Случайное число"));
    pn2.add(txt2);
    txt2.setEnabled(false);
    JPanel pnb = new JPanel();
    JButton btn = new JButton("Показать число");
    pnb.add(btn);
    c.add(pnb, BorderLayout.SOUTH);
    btn.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          txt1.setText(String.valueOf(numbOfRand));
          txt2.setText(String.valueOf(randValue));
        }
    });
    JPanel pntop = new JPanel(new FlowLayout(FlowLayout.RIGHT, 5, 5));
    JButton sbtn = new JButton("Показать время");
    pntop.add(sbtn);
    pntop.add(txtTime);
    txtTime.setEnabled(false);
    txtTime.setEditable(false);
    c.add(pntop, BorderLayout.NORTH);
    sbtn.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          while ( true ) {
            Date dt = new Date();
```

```
            txtTime.setText(dt.toString());
            try {
               Thread.sleep(200);
            } catch(InterruptedException ex) {
            }
          }
        }
    });

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

    setVisible(true);

    Random rnd = new Random();
    for(;;numbOfRand++) {
      randValue = rnd.nextInt();
      try {
         Thread.sleep(1000);
      } catch(InterruptedException e) {
      }
    }
  }

  public static void main(String[] args) {
    new ThreadExample3a();
  }
}
```

Здесь предпринята попытка реализовать контролируемый пользователем запуск часов. Добавлена кнопка *Показать время* и к ней подключен слушатель. В нем запускается бесконечный цикл, в котором 5 раз в секунду выводится на экран текущее время. Оттранслировав и запустив эту программу мы с удивлением обнаружим, что при нажатии на кнопку программа "зависает" и не реагирует на наши действия. Снять ее можно только из окна консоли комбинацией клавиш `Ctrl/C` .

Секрет опять же заключается в том, как работает *Нить Диспетчеризации Событий* , о которой говорилось выше. Дело в том, что эта нить несет ответственность за перерисовку окна приложения, в ней обрабатываются события от клавиатуры и мыши. В частности, все слушатели визуальных компонент вызываются в этой нити. И пока слушатель не отработает, никакие другие действия, которые должна выполнять данная нить, не выполняются. Т.е. в нашем примере слушатель захватил управление и тем самым заблокировал выполнение основной нити управления диалогом. Отсюда мораль.

- Слушатели событий от визуальных компенент должны содержать код, который быстро выполняется, иначе он будет блокировать или задерживать диалог с пользователем.

Какой же выход из создавшейся ситуации. С одной стороны мы не можем в слушателе реализовывать долгосрочные вычисления, а с другой — нам нужно запустить бесконечный цикл отображения времени.

Ответ на этот вопрос состоит в использовании возможностей нитей. Т.е. нам нужно из слушателя запустить отдельную нить и на этом завершить работу слушателя.

Это реализовано в следующем примере.

```
// ThreadExample4a.java

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

public class ThreadExample4a extends JFrame {

  JTextField txt1 = new JTextField(10);
  JTextField txt2 = new JTextField(10);
  JTextField txtTime = new JTextField(18);
  int randValue = 0;
  long numbOfRand = 0;
  SimpleThread sth = null;
  JButton sbtn;
```

```
ThreadExample4a() {
   super("Знакомство с нитями (часы)");

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

  setSize(400, 300);
  Container c = getContentPane();
  JPanel pnm = new JPanel(new GridLayout(2, 1, 5, 5));
  c.add(pnm, BorderLayout.CENTER);
  JPanel pn1 = new JPanel(new FlowLayout(FlowLayout.CENTER, 5, 5));
  JPanel pn2 = new JPanel(new FlowLayout(FlowLayout.CENTER, 5, 5));
  pnm.add(pn1);
  pnm.add(pn2);
  pn1.add(new JLabel("Номер числа        "));
  pn1.add(txt1);
  txt1.setEnabled(false);
  pn2.add(new JLabel("Случайное число"));
  pn2.add(txt2);
  txt2.setEnabled(false);
  JPanel pnb = new JPanel();
  JButton btn = new JButton("Показать число");
  pnb.add(btn);
  c.add(pnb, BorderLayout.SOUTH);
  btn.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        txt1.setText(String.valueOf(numbOfRand));
        txt2.setText(String.valueOf(randValue));
      }
  });
  JPanel pntop = new JPanel(new FlowLayout(FlowLayout.RIGHT, 5, 5));
  sbtn = new JButton("Показать время");
  pntop.add(sbtn);
  pntop.add(txtTime);
  txtTime.setEnabled(false);
  txtTime.setEditable(false);
  c.add(pntop, BorderLayout.NORTH);
```

```
    sbtn.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
           if ( sth == null ) {
             sth = new SimpleThread();
             sth.start();
           }
           sth.switchOnOff();
           sbtn.setText( sth.isOn() ? "Остановить часы" : "Показать
время" );
        }
    });

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

    setVisible(true);

    Random rnd = new Random();
    for(;;numbOfRand++) {
      randValue = rnd.nextInt();
      try {
         Thread.sleep(1000);
      } catch(InterruptedException e) {
      }
    }
  }

  class SimpleThread extends Thread {

    private boolean runFlag = false;

    public void run() {
      while ( true ) {
        if ( runFlag ) {
          Date dt = new Date();
          txtTime.setText(dt.toString());
```

```
        } else {
          txtTime.setText("");
        }
        try {
           Thread.sleep(200);
        } catch(InterruptedException e) {
        }
      }
    }

    public boolean isOn() {
      return runFlag;
    }

    public void switchOnOff() {
      runFlag = !runFlag;
    }

  }

  public static void main(String[] args) {
    new ThreadExample4a();
  }
}
```