# Конспект лекций по Java. Занятие 23

В.Фесюнов

## 1. Множественные нити выполнения (Multiple threads). Продолжение

Рассмотрим ряд возможностей и деталей реализации нитей в Java.

### 1.1. Саморегистрация нитей

Как уже отмечалось, каждая нить связана с объектом некоторого класса, унаследованного от Thread. В примере ThreadExample2.java мы порождали нить при помощи фрагмента программы:

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

Здесь в `sth` заносится ссылка на этот объект. Наличие ссылки гарантирует, что объект не будет уничтожен.

Однако, для нитей это необязательно. Ссылки на объекты, соответствующие нитям, можно не хранить. При старте нити соответствующий объект регистрируется JVM и сборщик мусора не будет уничтожать этот объект пока нить выполняется даже в том случае, если на него нет ни одной ссылки.

Т.е., если ссылка на объект нити не нужна для каких-то других целей, то ее можно не хранить. В частности, в примере ThreadExample2.java можно было бы порождение нити организовать так

```
    (new SimpleThread()).start();
```

### 1.2. Завершение и останов нити

Как уже указывалось, нормальное завершение нити происходит при выходе из метода `run` . Кроме того, иногда в приложении требуется выполнить временный останов нити, с тем, чтобы потом возобновить ее работу. Например, в `ThreadExample4a.java` , который мы частично рассмотрели в конце прошлого занятия, часы можно запустить, остановить, потом опять запустить, и т.д.

В классе `Thread` есть методы `stop` (завершить), `suspend` (приостановить) и `resume` (возобновить), но все они объявлены **deprecated** (устаревшими) и их использование не рекомендовано. О причинах можно почитать подробнее в документации по Java и мы их здесь рассматривать не будем.

Как же поступить в случае, если нужно приостановить выполнение процесса? Для этого нужно не останавливать процесс, а обеспечить такой алгоритм, при котором он "работает в холостую". Типичный пример представлен в `ThreadExample4a.java` .

Рассмотрим его подробнее. Вложенный класс `SimpleThread` содержит поле-флажок

```
    private boolean runFlag = false;
```

которое используется для приостановки процесса выдачи текущего времени. В цикле метода `run` это поле проверяется и если флажек установлен, то время вычисляется и выводится на экран, если нет, то на экран выводится пустая строка.

Дополнительно в классе реализованы методы `switchOnOff` и `isOn` , которые позволяют производить внешние (по отношеню к классу) манипуляции с данным флажком.

## 1.3. Интерфейс Runable

На прошлом занятии мы познакомились с одним из двух способов создания нитей - при помощи создания класса наследника `Thread` .

Второй способ состоит в реализации интерфейса `Runnable` . Этот вариант полностью эквивалентен первому, но иногда более удобен — можно не создавать отдельный класс. Вместо построения нового класса мы в некотором классе объявляем `implements Runnable` в заголовке класса и реализуем в нем метод `run()` . После этого для порождения новой нити используется собственно класс `Thread` . А именно

— порождается объект класса `Thread` , но при этом используется конструктор

`public Thread(Runnable target)`

Рассмотрим схематический пример.

```
public class A implements Runnable {

    public void run() {
      . . .
    }
    . . .
    public int f() {
// Запускаем нить
        Thread th = new Thread(this);
        th.start();
        . . .
    }
    . . .
}
```

Здесь при создании нити в качестве параметра передается `this` . В результате при старте нити будет вызван метод `run` класса `A` .

Переделаем наш пример на использование интерфейса `Runnable` . Результат представлен в листинге `ThreadExample5a.java` :

```
// ThreadExample5a.java

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

public class ThreadExample5a extends JFrame implements Runnable {

  JTextField txt1 = new JTextField(10);
  JTextField txt2 = new JTextField(10);
  JTextField txtTime = new JTextField(19);
  int randValue = 0;
```

```
long numbOfRand = 0;
Thread sth = null;
JButton sbtn;
private boolean runFlag = false;

ThreadExample5a() {
   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 Thread(ThreadExample5a.this);
             sth.start();
           }
           switchOnOff();
           sbtn.setText( 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) {
      }
    }
  }

  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 ThreadExample5a();
  }
}
```

Здесь исключен класс `SimpleThread` , а наш основной класс ( `ThreadExample5a` ) описан, как реализующий интерфейс `Runnable` . Все методы класса `SimpleThread` перенесены в класс `ThreadExample5a` . Переменная `sth` изменила свой тип на Thread, а запуск нити реализован так:

```
            sth = new Thread(ThreadExample5a.this);
            sth.start();
```

Т.е. это действительно альтернативный способ создания нитей и ничего более. Использование того или иного способа зависит от конкретной ситуации и предпочтений разработчика.

Интересный пример создания множества нитей приводится в Thinking in Java. Bruce Eckel . (Глава 14, файл Counter4.java). Данный пример может работать и как приложение и как аплет. В нем создается N нитей-счетчиков, которые можно запустить/остановить каждую в отдельности.

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

class Ticker extends Thread {
  private JButton b = new JButton("Toggle");
  private JTextField t = new JTextField(10);
  private int count = 0;
  private boolean runFlag = true;
  public Ticker(Container c) {
    b.addActionListener(new ToggleL());
    JPanel p = new JPanel();
    p.add(t);
    p.add(b);
    c.add(p);
  }
  class ToggleL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      runFlag = !runFlag;
    }
  }
  public void run() {
    while (true) {
      if (runFlag)
        t.setText(Integer.toString(count++));
      try {
        sleep(100);
      } catch (InterruptedException e) {}
    }
  }
}

public class Counter4 extends JApplet {
  private JButton start = new JButton("Start");
  private boolean started = false;
  private Ticker[] s;
  private boolean isApplet = true;
  private int size;
  public void init() {
```

```
  Container cp = getContentPane();
  cp.setLayout(new FlowLayout());
  // Get parameter "size" from Web page:
  if (isApplet)
    size =
      Integer.parseInt(getParameter("size"));
  s = new Ticker[size];
  for (int i = 0; i < s.length; i++)
    s[i] = new Ticker(cp);
  start.addActionListener(new StartL());
  cp.add(start);
}
class StartL implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    if(!started) {
      started = true;
      for (int i = 0; i < s.length; i++)
        s[i].start();
    }
  }
}
public static void main(String[] args) {
  Counter4 applet = new Counter4();
  // This isn't an applet, so set the flag and
  // produce the parameter values from args:
  applet.isApplet = false;
  applet.size =
    (args.length == 0 ? 5 :
      Integer.parseInt(args[0]));
  JFrame frame = new JFrame("Counter4");
  frame.addWindowListener(
    new WindowAdapter() {
      public void windowClosing(WindowEvent e){
        System.exit(0);
      }
    });
  frame.getContentPane().add(applet);
  frame.setSize(200, applet.size * 50 + 30);
  applet.init();
  applet.start();
```

```
    frame.setVisible(true);
  }
}
```

## 1.4. Нити — демоны (Daemons)

Кроме обычных нитей существуют еще нити-демоны. Они отличаются от обычных нитей только тем, что при завершении всех нитей, кроме нитей-демонов, программа завершается. В нашем демонстрационном примере этого не требовалось, поскольку программа завершается вызовом метода `System.exit(0)` , который принудительно прекращает работу всех нитей программы.

Для обеспечения "демоничности" нитей служат два метода класса `Thread` :

```
public final boolean isDaemon()
```

Позволяет проверить, является ли нить демоном.

```
public final void setDaemon(boolean on)
```

Устанавливает/снимает признак нити-демона.

## 1.5. Приоритеты нитей

Для оптимизации параллельной работы нитей в Java имеется возможность устанавливать приоритеты нитей. Нити с большим приоритетом имеют преимущество в получении времени процессора перед нитями с более низким приоритетом.

Работа с приоритетами обеспечивается следующими методами класса Thread:

```
public final void setPriority(int newPriority)
```

Устанавливает приоритет нити.

```
public final int getPriority()
```

Позволяет узнать приоритет нити.

Значение параметра в методе `setPriority` не может произвольным. Оно должно

находиться в пределах от `MIN_PRIORITY` до `MAX_PRIORITY` . При своем создании нить имеет приоритет `NORM_PRIORITY` .

## 1.6. Практическая работа

Модифицируйте пример Counter4.java так, чтобы разные нити имели разный приоритет. Запустите это приложение и посмотрите, какой это дает эффект.

## 1.7. Диспетчеризация нитей

Параллельная работа нитей на одном процессоре обеспечивается операционной системой (ОС). Для этого ОС должна периодически прерывать выполнение нитей, чтобы обеспечить переключение между ними. Различные ОС имеют разные стратегии обеспечения мультипрограммного режима работы. Одни из них лучше, другие хуже.

Класс `Thread` имеет средства, способствующие созданию программ, хорошо работающих при любых стратегиях. Но эти средства являются явными. Т.е., если разработчик предусмотрел их использование в нити, то нить будет работать нормально под любой ОС, если нет, то она может заблокировать все остальные нити программы, монопольно захватив процессор. Основное правило состоит в том, что нить должна периодически отдавать управление ОС. Для этого в классе `Thread` есть методы `sleep` и `yield` .

Метод `sleep` приостанавливает выполнение данной нити на заданное количество миллисекунд. Все остальные нити и задачи при этом не ждут истечения интервала времени, а продолжают работать. Во всех приведенных приложениях внутри основного цикла метода `run` находится вызов метода `sleep` .

Но бывают случаи, когда в основном цикле нити не нужно делать никакую задержку. Т.е. нить должна работать беспрерывно. Тогда вместо `sleep` нужно использовать `yield` . Этот метод не приостанавливает работу нити, а просто дает возможность ОС выполнить, при необходимости, переключение на другую нить или задачу.

- В любом методе `run` нужно использовать либо `sleep` , либо `yield` , либо какой-то другой их эквивалент ( `wait` или методы чтения из файла). В противном случае можно получить эффект монопольного захвата ресурсов машины одной

нитью.

## 1.8. Имена нитей

Каждая нить имеет имя. Имя нити — это строка. Оно может быть использовано при выдаче отладочной информации, при печати сообщений об ошибках и т.п. Если оно не установлено явно, JVM формирует его сама при создании нити.

Имя нити может быть задано либо в параметре конструктора `Thread` , либо методом `setName` . Извлечь имя можно при помощи метода `getName` .

## 1.9. Конкурентный доступ к ресурсам при многопоточной обработке

Как уже указывалось, нити, в отличие от задач, работают на одном пространстве памяти. Соответственно, при использовании нитей иногда возникает проблема конкурентного доступа к одним и тем же ресурсам из разных нитей.

Иногда это не существенно, например, если ресурс используется только на чтение. Но иногда это может приводить к печальным последствиям.

Типичный пример — это приложение, выполняющее списание средств со счета. Такое приложение перед списанием должно проверить наличие средств на счете, а потом выполнить собственно списание. Если может выполняться несколько экземпляров такого алгоритма в разных нитях, то они могут проверить наличие средств и определить, что средства для каждого отдельного списания на счете есть. После чего обе нити выполнят списание, хотя общая сумма может превышать сальдо счета.

Второй возможный сценарий на ту же тему следующий. Нить сначала считывает сальдо счета, потом уменьшает считанное значение на указанную сумму, а потом записывает на счет. Тогда, при параллельном выполнении двух таких нитей, обе они могут считать одну и ту же сумму, изменить ее (каждая по своему) и записать обратно. При этом последняя (по порядку выполнения) записанная сумма перекроет первую.

### 1.9.1. Демонстрационный пример

Продемонстрируем это на примере. Возьмем наш пример со случайными числами и

временем и модифицируем его. Во-первых, уберем с экрана кнопку "Показать число", поскольку мы будем показывать случайные числа в цикле из процессов. Во-вторых, добавим на экран поле "Контрольная сумма". Контрольная сумма изначально у нас будет равна 0. При показе текущего случайного числа мы дополнительно будем вычислять эту контрольную сумму, прибавляя и отнимая от нее текущее случайное число. Если все в порядке, то контрольная сумма будет равна 0.

Текст программы приведен ниже

```
// ThreadExample6.java

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

public class ThreadExample6 extends JFrame implements Runnable {

  JTextField txt1 = new JTextField(10);
  JTextField txt2 = new JTextField(10);
  JTextField txtTime = new JTextField(19);
  JTextField txt3 = new JTextField(10);
  int randValue = 0;
  long numbOfRand = 0;
  Thread sth = null;
  JButton sbtn;
  int checkSum = 0;
  private boolean runFlag = false;

  ThreadExample6() {
     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));
    JPanel pn3 = new JPanel(new FlowLayout(FlowLayout.CENTER, 5, 5));
    pnm.add(pn1);
    pnm.add(pn2);
    pnm.add(pn3);
    pn1.add(new JLabel("Номер числа        "));
    pn1.add(txt1);
    txt1.setEnabled(false);
    pn2.add(new JLabel("Случайное число"));
    pn2.add(txt2);
    txt2.setEnabled(false);
    pn3.add(new JLabel("Контрольная сумма"));
    pn3.add(txt3);
    txt3.setEnabled(false);
    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 Thread(ThreadExample6.this);
             sth.start();
           }
           switchOnOff();
           sbtn.setText( isOn() ? "Остановить часы" : "Показать
время" );
        }
    });

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

```
    }
  };
  addWindowListener(wndCloser);

  setVisible(true);

  class ShowRandom extends Thread {
      public void run() {
          while( true ) {
              checkSum += randValue;
              txt1.setText(String.valueOf(numbOfRand));
              txt2.setText(String.valueOf(randValue));
              checkSum -= randValue;
              txt3.setText(String.valueOf(checkSum));
              try {
                 Thread.sleep(200);
              } catch(InterruptedException e) {
              }
          }
      }
  }
  Thread[] th = new Thread[10];
  for ( int i = 0; i < 10; i++) {
      th[i] = new ShowRandom();
      th[i].start();
  }

  Random rnd = new Random();
  for(;;numbOfRand++) {
    randValue = rnd.nextInt();
    try {
       Thread.sleep(100);
    } catch(InterruptedException e) {
    }
  }
}

public void run() {
  while ( true ) {
    if ( runFlag ) {
```

```
        Date dt = new Date();
        txtTime.setText(dt.toString());
      } else {
        txtTime.setText("");
      }
      try {
         Thread.sleep(500);
      } catch(InterruptedException e) {
      }
    }
  }

  public boolean isOn() {
    return runFlag;
  }

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

  public static void main(String[] args) {
    new ThreadExample6();
  }
}
```

В программе описан класс-нить `ShowRandom` , который выполняет все действия по визуализации текущего случайного числа, а также вычисляет и отображает контрольную сумму. Программа порождает десять объектов этого класса и запускает 10 нитей. Если оттранслировать и запустить эту программу, то через некоторое время ее работы мы увидим, что контрольная сумма стала ненулевой.

Причина такого эффекта в том, что запущенные нити конкурируют друг с другом за время процессора. В этой "конкурентной борьбе" участвуют *Нить Диспетчеризации Событий* , основная нить программы, в которой выполняется генерация случайных чисел, и 10 нитей `ShowRandom` . Кроме того, мы можем запустить нить для показа текущего времени. Каждая из нитей `ShowRandom` может быть приостановлена ОС в любой момент. При этом ОС передаст управление какой-то другой нити. В частности, она может быть прервана после выполнения

```
                checkSum += randValue;
```

и перед выполнением

```
                checkSum -= randValue;
```

Если при этом управление будет передано нити, генерирующей случайные числа, то прибавлено к `checkSum` будет одно число, а вычтено другое.

Как можно увидеть исходя из поведения программы, вероятность такого события не очень велика, но она все же сущестует.

### 1.9.2. Средства синхронизации нитей в Java

Проанализируем эту проблему с другой точки зрения. У нас есть некоторый ресурс, в данном случае — случайное число (переменная `randValue` ), доступ к которому нужно упорядочить. Т.е. нужно заблокировать доступ к этому ресурсу для всех нитей, кроме одной, пока эта нить не выполнит над ним необходимые действия.

В Java есть возможности по синхронизации нитей, построенные на этом принципе. Т.е. определяется некоторый ресурс, который может быть заблокирован. Таким ресурсом может быть любой объект (но не данное элементарного типа). Далее определяются критические участки программы. При входе в такой участок, ресурс блокируется и становится недоступным для всех других нитей (с некоторой оговоркой, которую мы рассмотрим позже). При выходе из критического участка выполняется разблокировка ресурса.

В качестве критического участка программы в Java может быть определен некоторый нестатический метод или блок. При вызове метода блокируется объект, для которого вызван данный метод ( **this** ). Для блока блокируемый объект указывается явно. Синтаксис определения критических участков следующий. Для метода мы просто при описании метода указываем описатель **synchronized** , например,

```
public void synchronized f() {
   ...
}
```

Для блока мы непосредственно перед блоком ставим конструкцию `synchronized(` *объект* ), где *объект* — это ссылка на блокируемый объект. Например,

```
synchronized(ref) {
   ...
}
```

Здесь в качестве критического участка определяется блок, а в качестве блокируемого объекта — объект, на который ссылается `ref` .

### 1.9.3. За все приходится платить

Теперь обещанная оговорка о недоступности заблокированных объектов для других нитей. Другая нить будет приостановлена только в том случае, если в ней также определен критический участок (метод или блок). При входе в такой метод или блок будет выполнена проверка на то, что объект заблокирован, и, если это так, то нить приостановит свое выполнение, пока заблокировавшая объект нить не выйдет из критического учаска. Если объект не заблокирован, то нить заблокирует его и начнет выполнение действий, запрограммированных в критическом участке. При выходе из критического участка объект будет разблокирован. При этом если есть нити, которые ждут разблокировки данного объекта, то одна из них будет активизирована, а остальные продолжат ожидание.

Такой механизм блокировки разработан потому, что блокировка/разблокировка требует накладных расходов и замедляет выполнение программы. Альтернативой было бы выполнение проверок на блокировку объекта при каждой операции доступа к объекту. Данный механизм является компромиссом, который, с одной стороны, дает возможность выполнить синхронизацию доступа к разделяемым ресурсам, а с другой — имеет приемлемый уровень временных затрат на поддержку такой синхронизации.

Тем не менее, нужно помнить, что критические участки замедляют выполнение программы. Так, вызов синхронизированного метода в 4 раза медленнее, чем вызов обычного. Поэтому синхронизацию доступа нужно делать только тогда, когда это необходимо.

Кроме того, критические участки программы нужно делать как можно меньше, помня, что пока выполняется данный критический участок все остальные процессы не могут выполнять критические участки, связанные с блокировкой того же ресурса.

### 1.9.4. Исправленный пример

Для того, чтобы рассмотренный выше пример `ThreadExample6.java` работал правильно, нам необходимо синхронизировать доступ к полю `randValue` . Но это поле является элементарным данным, поэтому использовать его в качестве блокируемого ресурса нельзя. В таких случаях блокируют сам объект, которому принадлежит поле. У нас это объект основного класса приложения.

Далее нужно выделить критические участки. Один из них — это порождение нового случайного числа.

```
      randValue = rnd.nextInt();
```

Этот участок мы заключим в блок.

Второй — это фрагмент, в котором вычисляется контрольная сумма с использованием `randValue` . Исключительно ради разнообразия мы выделим его в отдельный метод с именем `check` . Правда при этом нужно учесть, что для метода блокируемый объект — это объект того класса, которому принадлежит данный метод. Поэтому метод `check` должен быть обязательно методом нашего основного класса, а не методом вложенного класса `ShowRandom` .

Листинг исправленной программы `ThreadExample7.java` приведен ниже

```
// ThreadExample7.java

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

public class ThreadExample7 extends JFrame implements Runnable {

  JTextField txt1 = new JTextField(10);
  JTextField txt2 = new JTextField(10);
  JTextField txtTime = new JTextField(19);
  JTextField txt3 = new JTextField(10);
  int randValue = 0;
```

```
long numbOfRand = 0;
Thread sth = null;
JButton sbtn;
int checkSum = 0;
private boolean runFlag = false;

ThreadExample7() {
   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));
  JPanel pn3 = new JPanel(new FlowLayout(FlowLayout.CENTER, 5, 5));
  pnm.add(pn1);
  pnm.add(pn2);
  pnm.add(pn3);
  pn1.add(new JLabel("Номер числа        "));
  pn1.add(txt1);
  txt1.setEnabled(false);
  pn2.add(new JLabel("Случайное число"));
  pn2.add(txt2);
  txt2.setEnabled(false);
  pn3.add(new JLabel("Контрольная сумма"));
  pn3.add(txt3);
  txt3.setEnabled(false);
  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 Thread(ThreadExample7.this);
             sth.start();
           }
           switchOnOff();
           sbtn.setText( isOn() ? "Остановить часы" : "Показать
время" );
        }
    });

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

    setVisible(true);

    class ShowRandom extends Thread {

        public void run() {
            while( true ) {
                check();
                txt3.setText(String.valueOf(checkSum));
                try {
                   Thread.sleep(200);
                } catch(InterruptedException e) {
                }
            }
        }
    }
    Thread[] th = new Thread[10];
    for ( int i = 0; i < 10; i++) {
        th[i] = new ShowRandom();
        th[i].start();
    }
```

```
  Random rnd = new Random();
  for(;;numbOfRand++) {
    synchronized(this) {
        randValue = rnd.nextInt();
    }
    try {
       Thread.sleep(100);
    } catch(InterruptedException e) {
    }
  }
}

public void run() {
  while ( true ) {
    if ( runFlag ) {
      Date dt = new Date();
      txtTime.setText(dt.toString());
    } else {
      txtTime.setText("");
    }
    try {
       Thread.sleep(500);
    } catch(InterruptedException e) {
    }
  }
}

public boolean isOn() {
  return runFlag;
}

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

public synchronized void check() {
    checkSum += randValue;
    txt1.setText(String.valueOf(numbOfRand));
    txt2.setText(String.valueOf(randValue));
```

```
      checkSum -= randValue;
  }

  public static void main(String[] args) {
    new ThreadExample7();
  }
}
```

Запустив этот пример, мы обнаружим, что сколько бы программа не работала, поле контрольной суммы останется нулевым.