# Конспект лекций по Java. Занятие 24

В.Фесюнов

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

### 1.1. Блокировки нитей

Продолжим рассмотрение конкурентного доступа нитей к ресурсам и сосредоточим свое внимание на блокировках нитей.

Когда одна из нитей входит в критический участок ( `synchronized` метод или блок), связанный с блокировкой какого-то объекта, то остальные нити не могут войти в критические участки, связанные с блокировкой того-же объекта, пока захватившая объект нить не выйдет из критического участка. Т.е. блокировка объекта может вызывать блокировку ряда нитей.

Кроме этого варианта, мы рассмотрели еще один способ блокировки процесса на некоторое время — это метод `sleep` , который блокирует нить на некоторое время.

В Java есть еще две возможности заблокировать нить — это вызвать операцию ввода/вывода и вызвать метод `wait` .

- Есть еще один вариант — это метод `suspend` , но мы не будем его рассматривать, поскольку он является небезопасным и устаревшим.

**Заблокированные** нити не участвуют в конкурентоной борьбе за время процессора. Планировщик ( **sheduler** ), выделяющий кванты времени нитям, просто игнорирует их. В отличии от заблокированных нитей, нити, которые участвуют в конкурентной борьбе за процессор, мы будем называть **активными** — **runable** (более точный перевод — готовые к выполнению). Ну и для полноты терминологии, ту нить (или нити, если на машине несколько процессоров), которая выпоняется в настоящее время

мы будем называть **выполняющейся** .

Суммируя, можно сказать, что нить может быть заблокирована в следующих случаях.

- вызван метод `sleep` . Нить блокируется на время. По истечению этого времени нить переводится в разряд активных.
- нить пытается войти в критический участок, но соответствующий ресурс заблокирован какой-то другой нитью. Данная нить блокируется до тех пор, пока не будет разблокирован этот ресурс (если нет других нитей, ожидающих тот же ресурс).
- нить выполняет операцию ввода/вывода. Нить блокируется до окончания этой операции.
- вызван метод `wait` . Нить блокируется до окончания временного интервала или вызова метода `notify` или `notifyAll` .

## 1.2. Метод wait

Методы `wait` , `notify` , `notifyAll` — наиболее сложный для понимания инструмент управления доступом к разделяемым ресурсам.

Во-первых, в отличии от метода `sleep` , который в чем-то похож на один из двух вариантов метода `wait` , последний является методом класса `Object` , а не `Thread` . То же самое относится и к методам `notify` и `notifyAll` . Они являются методами класса `Object` .

Дело в том, что метод `wait` блокирует выполнение нити, из которой он вызван, и одновременно разблокирует объект, для которого он вызван. Поскольку блокировать можно любой объект, то и метод `wait` реализован в `Object` .

Далее. Метод `wait` может быть вызван только из критического участка, связанного с тем объектом, для которого вызывается `wait` . Если это не так, то выдается `IllegalMonitorStateException` — недопустимое состояние монитора.

Это требует небольшого пояснения.

- Для обеспечения блокировки в каждом объекте создается специальное системное поле, в которое прописывается состояние блокировки объекта - заблокирован/разблокирован. Это поле называется **монитор** .

Такой механизм работы метода `wait` позволяет нам на время отменить действие критического участка и дать возможность какой-то другой нити, ожидающей освобождения данного ресурса, в свою очередь войти в критический участок.

Метод `wait` имеет две модификации — с параметром и без параметра. В первой модификации в качестве параметра задается время ожидания. Выход из метода `wait` происходит либо по завершению указанного интервала времени, либо при вызове из другой нити метода `notify` или `notifyAll` для объекта, по которому мы производим синхронизацию. Во второй модификации время ожидания не задается, что равносильно указанию нуля в качестве времени ожидания, и означает бесконечное ожидание, которое может быть прервано только вызовом `notify` .

Встает вопрос. Что же будет, если при выходе из `wait` , который обязан находится внутри критического участка, ресурс, по которому мы производим синхронизацию, окажется заблокированнным другой нитью? Ответ состоит в том, что нить будет ждать окончания блокировки этого ресурса точно так же, как она ждала бы при попытке входа в критический участок. Обычно так и происходит. Т.е. выход из метода `wait` вызывает смену одного типа блокировки нити (блокировка по `wait` ) на другой — блокировка по монитору. Когда монитор объекта будет освобожден (это произойдет, когда заблокировавшая ресурс нить выйдет из критического участка или вызовет метод `wait` ), наша нить захватит монитор и продолжит выполнение критического участка.

Использование `wait` позволяет организовать более изощренную обработку, но, соответственно, его использование усложняет программу. Поэтому не стоит его применять без особых на то оснований. Если программу можно написать используя `synchronized` — методы и блоки, то этим стоит и ограничится и не применять `wait` .

Для демонстрационных целей мы, однако, рассмотрим применение `wait` в новой версии нашей программы ( `ThreadExample8.java` ), хотя в ней можно обойтись и без него. На этом примере мы посмотрим некоторые элементы техники применения `wait` .

```
// ThreadExample8.java
```

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

public class ThreadExample8 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;

  ThreadExample8() {
     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(ThreadExample8.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();
    synchronized(this) {
      for(;;numbOfRand++) {
          randValue = rnd.nextInt();
          try {
             wait(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 ThreadExample8();
  }
}
```

Данный пример лишь немного отлчается от предыдущего ( `ThreadExample7.java` ). В нем изменен фрагмент, выполняющий порождение случайных чисел. Предыдущая версия выглядела так

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

Новая редакция выглядит следующим образом

```
    Random rnd = new Random();
    synchronized(this) {
      for(;;numbOfRand++) {
```

```
            randValue = rnd.nextInt();
            try {
               wait(100);
            } catch(InterruptedException e) {
            }
        }
    }
```

Здесь весь цикл помещен внутрь `synchronized` блока, но внутри цикла вместо `sleep` стоит `wait` . И, хотя весь цикл порождения случайных чисел находится внутри критического участка, он не блокирует полностью работу остальных нитей, поскольку `wait` не только организует временную блокировку данного процесса (на 100 миллисекунд за цикл), но и разблокирует на это время синхронизирующий объект, давая тем самым возможность остальным процессам поработать с ним.

Если говорить о технике программирования, то, при использовании `synchronized` методов и блоков без `wait` , внутри них не должно быть, как правило, каких-либо циклов. И, наоборот, при использовании `wait` , обычно, тот критический участок, в котором он вызывается, содержит в себе цикл, который периодически прерывается вызовами метода `wait` .

## 1.3. Применение wait с notify и notifyAll

Иногда требуется, чтобы нить ждала наступления какого-либо события, после чего она может продолжить свою работу. Именно для этих целей применяется `wait` совместно с `notify` и/или `notifyAll` .

Разберемся как этого достичь. Прежде всего нужно отметить, что у нас должен быть объект, по которому все задействованные в данном сценарии нити будут осуществлять синхронизацию. И нужно очень внимательно следить за тем, чтобы все они синхронизировались именно по выбранному объекту, т.к. использование разными нитями различных синхронизирующих объектов — типичная ошибка при построении такого рода программ.

Пусть в качестве синхронизирующего объекта выступает объект, на который ссылается ссылка `ref` . Пусть нить **A** в некоторой точке должна дождаться события, которое должно произойти в некоторой точке в нити **B** и ссылка `ref` доступна как в **A**

, так и в **B** . Тогда в **A** соответствующий фрагмент программы мог бы выглядеть так

```
synchronized(ref) {
  . . .
  try {
     ref.wait();
  } catch(InterruptedException e) {
  }
  . . .
}
```

А в **B** так

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

Нить **A** дойдет до точки вызова метода `wait` и вызовет его. При этом она заблокируется по `wait` и одновременно освободит синхронизирующий объект, что даст возможность нити **B** войти в критический участок, выполнить необходимые действия и вызвать `notify` . Вызов `notify` "разбудит" нить **A** . Однако, нить **B** все еще находится в критическом участке, поэтому **A** все еще будет заблокирована, но не по `wait` , а по ожиданию освобождения синхронизирующего объекта. Как только **B** выйдет из критического участка, **A** опять захватит синхронизирующий объект и продолжит свою работу.

Это простейший случай, в котором задействованы всего две нити. Но, как мы видим, даже в этом случае все очень не просто. Кроме того, мы сделали еще одно неявное допущение, а именно, мы предположили, что **A** войдет в критический участок раньше, чем **B** . А для нитей гарантировать такое можно только в очень редких случаях. Поэтому тут явно чего-то еще не хватает.

Обычно, в реальных программах событие, которое должно произойти связывают не с вызовом `notify` , а с чем-то более существенным. Например, факт наступления данного события можно было бы отмечать в некоторой логической переменной или поле класса. Пусть в нашем случае это будет `eventHappened` . Тогда фрагмент нити

**A** должен иметь такой вид

```
synchronized(ref) {
  . . .
  if ( !eventHappened ) {
      try {
         ref.wait();
      } catch(InterruptedException e) {
      }
  }
  . . .
}
```

А в **B** так

```
synchronized(ref) {
  . . .
  eventHappened = true;
  ref.notify();
  . . .
}
```

В нити **A** мы проверяем, не произошло-ли уже ожидаемое событие, и если произошло, то не ждем, а продолжаем работу, в притивном случае — вызываем `wait` . В нити **B** мы отмечаем в `eventHappened` , что событие произошло, и вызываем `notify` , чтобы разблокировать **A** , если он был заблокирован по `wait` . При такой схеме можно не беспокоится о порядке выполнения критических участков разных нитей. В каком бы порядке они не выполнялись, мы получим требуемый результат.

Разберемся, чем отличается `notifyAll` от `notify` . Предположим, что в нашем примере нитей **A** может быть несколько. Тогда все зависит от характера события, которое они ожидают. Это может быть событие, сам факт наступления которого означает, что все нити **A** могут продолжить свою работу. А может такое, которое требует активизации только одной из нитей **A** . Приведенный выше фрагмент нити **B** , в котором используется `notify` , будет правильно работать только для последнего варианта. Если же нам нужно активизировать все ожидающие нити, то нужно использовать `notifyAll` вместо `notify` .

Типичная задача, где требуется применять `wait` с `notify` или `notifyAll` , — это

задача генерации/потребления. Представим себе, что у нас есть нить, генерирующая нечто. Например, это могут быть порции данных для обработки. И есть ряд нитей-потребителей, которые обрабатывают то, что генерирует нить-генератор. В этом случае нити-потребители должны быть синхронизированы с нитями-генератором так, чтобы они были активными только тогда, когда есть что потреблять. Это с одной стороны. С другой и нить-генератор, зачастую необходимо синхронизировать с нитями-потребителями, т.к. объем того, что она может сгенерировать, обычно не безграничен. Чаще всего есть буфер некоторого размера, куда генератор помещает то, что он генерирует. И при заполненности буфера генератор нужно остановить (заблокировать), а когда нить-потребитель забрала что-то из буфера, то генератор нужно активизировать.

Для решения поставленной задачи выберем синхронизирующий объект. В качестве такового в данном случае подойдет сам генератор. Далее выделим критические участки. В генераторе это может быть фрагмент, помещающий очередную порцию уже сгенерированных данных в буфер. В потребителях — это фрагмент, извлекающий порцию из буфера генератора. Следующим шагом может быть вывод о том, что оба этих действия удобно и возможно осуществлять при помощи методов генератора. Хотя они будут выполняться в разных нитях, никто не мешает в этих нитях вызывать разные методы одного класса.

Пусть генератор генерирует некоторые объекты класса `Product` . Тогда для занесения в буфер генератора мы можем создать метод

```
private synchronized void put(Product p) {
  . . .
}
```

а для выборки из буфера генератора — метод

```
public synchronized Product get() {
  . . .
}
```

Теперь следует выделить два события — одно для синхронизации потребителей с генератором, другое — генератора с потребителями. Первое из них — это событие "буфер пуст", второе — "буфер заполнен".

Далее нам требуется конкретизировать эти события. Удобнее всего это сделать, выделив буфер в самостоятельный класс. Его можно сделать вложенным классом класса генератора, т.к. никто, кроме самого генератора к этому классу обращаться не должен. В результате мы уже можем набросать примерный вид класса-генератора.

```
public class ProductGenerator extends Thread {

    class ProductBuffer {
        . . .
        /**
         * Проверят, не пуст ли буфер
         **/
        boolean isEmpty() {
            . . .
        }

        /**
         * Проверят, не заполнен ли буфер до отказа
         **/
        boolean isFull() {
            . . .
        }

        /**
         * Заносит в буфер один элемент
         * @throws IllegalStateException при попытке занести данные
         * в полностью заполненный буфер.
         **/
        void put(Product p) {
            . . .
        }

        /**
         * Извлекает из буфера один элемент
         * @throws IllegalStateException при попытке выбрать данные
         * из пустого буфера.
         **/
        Product get() {
            . . .
```

```
        }

    }

    private ProductBuffer buf = new ProductBuffer();

    /**
     * Метод возвращает сгенерированный объект.
     * Данный метод будет работать в нитях-потребителях.
     **/
    public synchronized Product get() {
        while( buf.isEmpty() ) {
            try {
               wait();
            } catch(InterruptedException e) {
            }
        }
        notifyAll();
        return buf.get();
    }

    /**
     * Метод заносит сгенерированный объект во внутренний буфер.
     * Данный метод будет работать в нити-генераторе.
     **/
    private synchronized void put(Product p) {
        if( buf.isFull() ) {
            try {
               wait();
            } catch(InterruptedException e) {
            }
        }
        notify();
        buf.put(p);
    }

    /**
     * Генерирует один объект. Возвращает его в качестве
     * результата.
     **/
```

```
    private Product generate() {
        ...
    }

    public void run() {
        while( true ) {
           put(generate());
           try {
               sleep(200);
           } catch ( InterruptedException e ) {
           }
        }
    }

}
```

Здесь методы класса `ProductBuffer` не реализованы. Их реализация зависит от того, что фактически будет выбрано в качестве буфера для хранения объектов. Это может быть массив или, например, `java.util.LinkedList` . В классе `ProductGenerator` поле `buf` содежит ссылку на объект класса `ProductBuffer` . Кроме того, класс `ProductGenerator` имеет 4 метода: `get`, `put`, `generate` и `run` . Метод `generate` не реализован, поскольку он зависит от специфики того, что мы генерируем. Метод `run` содержит в себе цикл генерации, в котором с периодичностью в 200 миллисекунд генерируется и заносится в буфер объект класса `Product` .

Давайте подробнее разберемся с методами `get` и `put` класса `ProductGenerator` . В частности, требует пояснения цикл в методе `get` :

```
        while( buf.isEmpty() ) {
            wait();
        }
```

Этот цикл здесь необходим. Дело в том, что при пустом буфере сразу несколько нитей-потребителей могут войти в состояние блокировки по `wait` . Когда генератор сгенерирует очередное значение и вызовет `put` , то он активизирует одну из ожидающих нитей-потребителей (метод `notify` внутри `put` ). В свою очередь, активизированная нить-потребитель продолжит выполнение метода `get` , что

приведет к вызову `notifyAll` . В результате будут активизированы все заблокированные по `wait` нити, но для них в буфере уже может не оказаться ничего, т.к. нить-генератор могла еще не успеть сгенерировать новое значение. Поэтому в `get` необходима проверка на то, что буфер не пуст даже после выхода из `wait` .

В методе `put` такой необходимости нет, т.к. нить генерации единственная и активизировать ее может только нить-потребитель, а это значит, что из буфера хоть что-то извлечено и там заведомо освободилось место.

Решим еще один вопрос. Почему в `get` применен `notifyAll` , а не `notify` и нельзя ли применением `notify` избежать цикла в `get` ? Во-первых, если бы мы применили `notify` , то цикл все равно бы понадобился, поскольку `notify` мог активизировать другую нить-потребитель и для нее опять же могло не оказаться ничего в буфере. Во-вторых, при выполнении `get` мы должны быть уверены в том, что мы активизируем нить-генератор, а это можно сделать только применением `notifyAll` .

Более красивое решение состоит в том, чтобы использовать не один, а два синхронизирующих объекта — один для синхронизации потребителей с генератором, другой — генератора с потребителем. В качестве второго объекта можно было бы выбрать буфер. Эту задачу попробуйте решить сами.

## 1.4. Пример с нитью-генератором и нитями-потребителями

В заключение рассмотрим программу, демонстрирующую применение нитей генераторов/потребителей. Нить-генератор реализована в `RandomGenerator.java` и генерирует случайные числа. Класс `RandomGenerator` построен по образу и подобию `ProductGenerator` с необходимыми модификациями.

```
import java.util.*;

public class RandomGenerator extends Thread {

    class RandomBuffer {

        /**
         * буфер
```

```
     **/
    private int buf[] = new int[100];

    /**
     * Индекс первого заполненного элемента
     **/
    private int beg = 0;

    /**
     * Количество элементов в буфере
     **/
    private int items = 0;

    /**
     * Проверят, не пуст ли буфер
     **/
    boolean isEmpty() {
        return items == 0;
    }

    /**
     * Проверят, не заполнен ли буфер до отказа
     **/
    boolean isFull() {
        return items == 100;
    }

    /**
     * Заносит в буфер один элемент
     * @throws IllegalStateException при попытке занести данные
     * в полностью заполненный буфер.
     **/
    void put(int p) {
        if ( isFull() )
            throw new IllegalStateException("Буфер полон");
        int end = (beg+items++) % 100;
        buf[end] = p;
    }

    /**
```

```
         * Извлекает из буфера один элемент
         * @throws IllegalStateException при попытке выбрать данные
         * из пустого буфера.
         **/
        int get() {
            if ( isEmpty() )
                throw new IllegalStateException("Буфер пуст");
            int v = buf[beg++];
            beg %= 100;
            items--;
            return v;
        }

        public int bufLen() {
            return items;
        }

    }

    private RandomBuffer buf = new RandomBuffer();
    private Random rnd = new Random();

    /**
     * Метод возвращает сгенерированный объект.
     * Данный метод будет работать в нитях-потребителях.
     **/
    public synchronized int get() {
        while( buf.isEmpty() ) {
            try {
               wait();
            } catch(InterruptedException e) {
            }
        }
        notifyAll();
        return buf.get();
    }

    /**
     * Метод заносит сгенерированный объект во внутренний буфер.
     * Данный метод будет работать в нити-генераторе.
```

```
     **/
    private synchronized void put(int p) {
        if( buf.isFull() ) {
            try {
               wait();
            } catch(InterruptedException e) {
            }
        }
        notify();
        buf.put(p);
    }

    /**
     * Генерирует один объект. Возвращает его в качестве
     * результата.
     **/
    private int generate() {
        return rnd.nextInt();
    }

    public void run() {
        while( true ) {
           put(generate());
           try {
               sleep(100);
           } catch ( InterruptedException e ) {
           }
        }
    }

    public int bufLen() {
        return buf.bufLen();
    }

}
```

В качестве буфера физически используется массив из 100 целых, а для управления им два целых поля, содержащих номер первого заполненного элемента буфера и количество елементов в буфере.

Основной класс `ThreadExample9` переделан из класса `ThreadExample8` . Убрано

поле контрольной суммы, исключены все элементы синхронизации внутри `ThreadExample9` , т.к. синхронизация выполняется самим классом-генератором. Исключено поле `randValue` и цикл генерации случайных значений. В качестве нитей-потребителей выступают нити `ShowRandom` , но в них значение случайного числа выбирается не из `randValue` , как ранее, а запрашивается очередное случайное число от генератора.

Для контроля над процессом порождения случайных чисел на экран добавлено поле, которое показывает текущее количество элементов в буфере. Количество нитей, и время ожидания внутри каждого из них подобрано так, чтобы буфер не оставался пустым, но и не заполнялся слишком быстро.

```
// ThreadExample9.java

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

public class ThreadExample9 extends JFrame implements Runnable {

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

  ThreadExample9() {
     super("Знакомство с нитями");

     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("Элементов в буфере"));
    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(ThreadExample9.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 ) {
              txt1.setText(String.valueOf(++numbOfRand));
              txt3.setText(String.valueOf(generator.bufLen()));
              txt2.setText(String.valueOf(generator.get()));
              try {
                 Thread.sleep(700);
              } catch(InterruptedException e) {
              }
          }
      }
  }

  generator.start();
  Thread[] th = new Thread[5];
  for ( int i = 0; i < 5; i++) {
      th[i] = new ShowRandom();
      th[i].start();
  }

}

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 ThreadExample9();
  }
}
```