



Java course

- [Начало Java](#)
- [Проект «Отдел кадров»](#)
- [Курсы](#)
- [Статьи](#)
- [Контакты/Вопросы](#)

- [Введение](#)
- [Установка JDK](#)
- [Основные шаги](#)
- [Данные](#)
- [Порядок операций](#)
- [IDE NetBeans](#)
- [ООП](#)
- [Инкапсуляция](#)
- [Наследование](#)
- [Пакеты](#)
- [Переопределение и перегрузка](#)
- [Полиморфизм](#)
- [Статические свойства и методы](#)
- [Отношения между классами](#)
- [Визуализация робота](#)
- [Пример — очередь объектов](#)
- [Массивы — знакомство](#)
- [Многомерные массивы](#)
- [Абстрактные классы](#)
- [Интерфейсы](#)

Многopotочность — первые шаги

Когда на собеседовании затрагивается вопрос многопоточности, то я себя ощущаю крайне неудобно. И не потому, что плохо знаю этот предмет — полагаю, что я его понимаю достаточно неплохо. Но священный трепет и тайна, которыми окутана эта область знаний, меня раздражают. Потому как хоть это и сложно, но вполне понимаемо. А вот собеседователи могут искать (и находить) совсем иные смыслы и особенности. Можно не сойтись во мнениях :).

Если понять, что именно там вызывает сложность и какие именно инструменты вам предлагаются, то дальше проблема уже не в самой многопоточности, но в сложности управления задачами, для которых она используется. Разобьем наше повествование на несколько частей:

1. Зачем нужна многопоточность ?
2. Как создавать потоки
3. Какие проблемы возникают и почему
4. Инструменты для решения проблем

Вполне допускаю, что мое восприятие субъективно и какие-то моменты и тонкости могут не попасть в это описание. Просто потому, что чем глубже погружаться в эту проблематику, тем большее количество интересных моментов можно обнаружить (на уровне реализации JVM или на уровне архитектуры процессоров). Также необходимо учесть, что характер решаемых задач тоже влияет — ведь именно при

- [Расширенное описание классов](#)
- [Исключения](#)
- [Решения на основе классов](#)
- [Список контактов — начало](#)
- [Коллекции — базовые принципы](#)
- [Коллекции — продолжение](#)
- [Список контактов — GUI приложение](#)
- [Что такое JAR-файлы](#)
- [Многопоточность — первые шаги](#)
- [Многопоточность и синхронизация](#)
- [Работаем с XML](#)
- [Reflection — основы](#)
- [Установка СУБД PostgreSQL](#)
- [Базы данных на Java — первые шаги](#)
- [Возможности JDBC — второй этап](#)
- [JDBC — групповые операции](#)
- [Список контактов — работаем с БД](#)
- [Переезжаем на Maven](#)
- [Потоки ввода-вывода](#)
- [Сетевое взаимодействие](#)
- [С чего начинается Web](#)

решении задач мы наталкиваемся на какие-либо ограничения, особенности реализации и прочие хитрые моменты.

Зачем нужна многопоточность

Ответ на этот вопрос не является проблемой. На сегодня количество процессоров в серверных системах больше одного, да и один процессор имеет несколько ядер. Предагаю продемонстрировать это на примере. Будьте внимательны — класс находится в пакете **edu.javacourse.threads**

```
1 package edu.javacourse.threads;
2
3 import java.math.BigInteger;
4 import java.security.SecureRandom;
5 import java.util.ArrayList;
6 import java.util.List;
7 import java.util.concurrent.Callable;
8 import java.util.concurrent.ExecutionException;
9 import java.util.concurrent.ExecutorService;
10 import java.util.concurrent.Executors;
11 import java.util.concurrent.Future;
12
13 public class Processor {
14
15     public static final int STR_COUNT = 100;
16     public static final int TASK_COUNT = 10000;
17
18     public static void main(String[] args) {
19         {
```

```

20         BigTaskOneThread bt = new BigTaskOneThread();
21         long d1 = System.currentTimeMillis();
22         Long result = bt.startTask();
23         long d2 = System.currentTimeMillis();
24         System.out.println(result + ", Time 1:" + (d2 - d1));
25     }
26     {
27         BigTaskManyThreads bt = new BigTaskManyThreads();
28         long d1 = System.currentTimeMillis();
29         Long result = bt.startTask();
30         long d2 = System.currentTimeMillis();
31         System.out.println(result + ", Time 2:" + (d2 - d1));
32     }
33 }
34
35 public Long process() {
36     Long summa = 0L;
37
38     SecureRandom random = new SecureRandom();
39     for (int i = 0; i < Processor.TASK_COUNT; i++) {
40         String g = new BigInteger(500, random).toString(32);
41         for (char c : g.toCharArray()) {
42             summa += c;
43         }
44     }
45     return summa;
46 }
47
48
49 class BigTaskOneThread {
50
51     public Long startTask() {
52         Long summa = 0L;
53         for (int i = 0; i < Processor.STR_COUNT; i++) {
54             Processor p = new Processor();
55             summa += p.process();
56         }
57         return summa;
58     }
59
60 }
61
62 class BigTaskManyThreads {
63
64     public Long startTask() {
65         int ap = Runtime.getRuntime().availableProcessors();
66         ExecutorService es = Executors.newFixedThreadPool(ap);

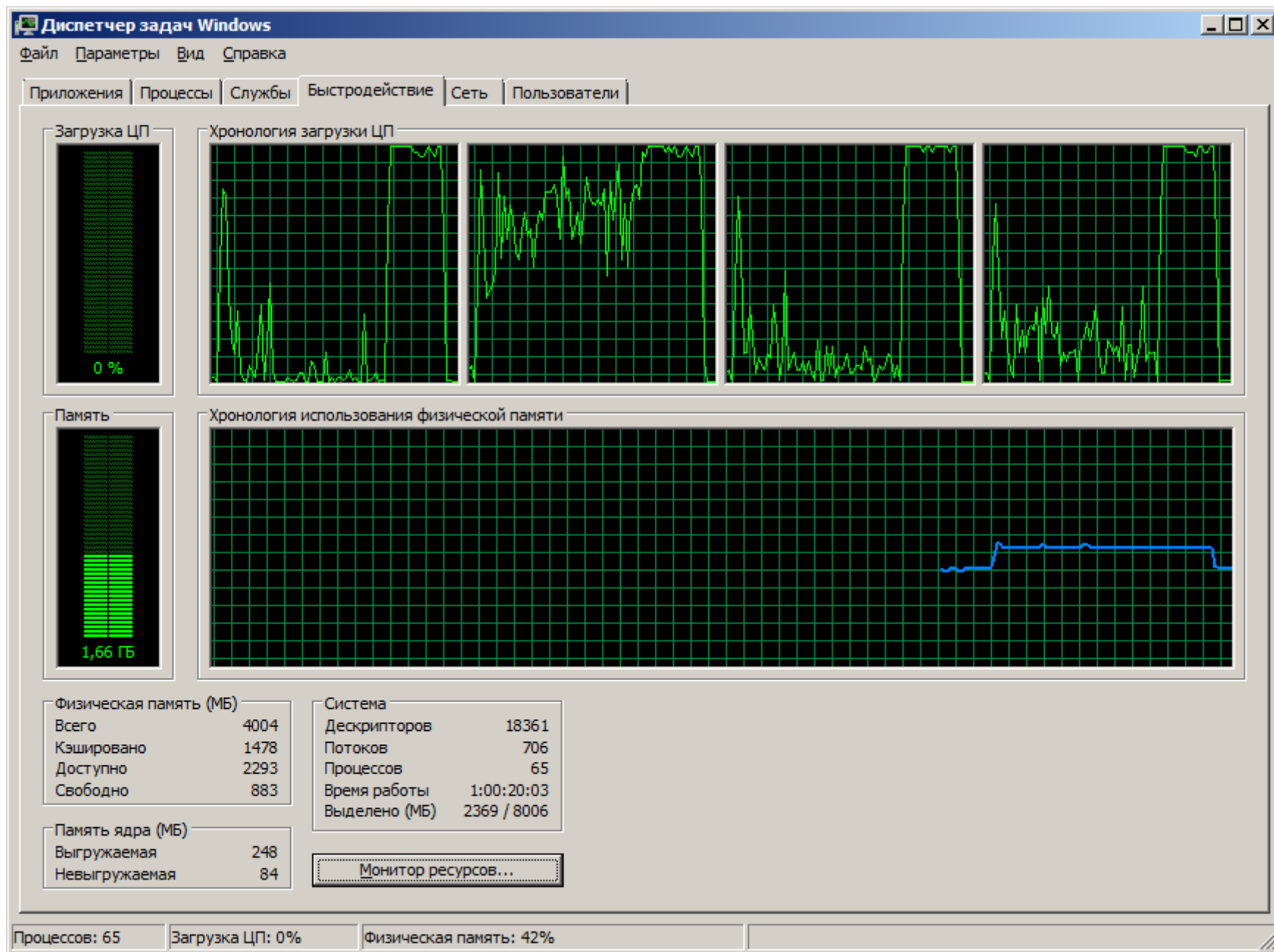
```

```

67
68     Long summa = 0L;
69     try {
70         List<MyCallable> threads = new ArrayList<MyCallable>();
71         for (int i = 0; i < Processor.STR_COUNT; i++) {
72             threads.add(new MyCallable());
73         }
74         List<Future<Long>> result = es.invokeAll(threads);
75
76         for (Future<Long> f : result) {
77             summa += f.get();
78         }
79         es.shutdown();
80     } catch (InterruptedException | ExecutionException ex) {
81         ex.printStackTrace(System.out);
82     }
83     return summa;
84 }
85 }
86
87 class MyCallable implements Callable<Long> {
88
89     @Override
90     public Long call() throws Exception {
91         Processor p = new Processor();
92         return p.process();
93     }
94 }

```

Пока не вдавайтесь в подробности — просто запустите этот замечательный пример, но перед этим советую запустить диспетчер задач, чтобы посмотреть загрузку процессоров. Вот какая картинка наблюдается у меня.



Сами видите, что до определенного времени загрузка была так себе — некоторая часть процессорного времени была свободна (у каждого ядра). Что явно не в наших интересах. У нас «сложная вычислительная задача», которая могла бы забрать себе все время CPU и выполниться быстрее. Но это не так. А вот вторая часть задачи уже показывает 100%-ую загрузку всех ядер процессора — и это хорошо, мы эффективно используем все

вычислительные ресурсы. Да и время отработки у нас стало гораздо меньше — у меня вторая часть выполнялась в три раза быстрее. В чем же причина ? Причина в нескольких потоках — мы загрузили каждое ядро отдельной задачей (потоком). Многопоточность подобна многозадачности — у вас может быть запущено сразу несколько программ — редактор, браузер, текстовый редактор — и вы можете переключаться между ними. Многопоточность по сути та же многозадачность только внутри одной задачи (одного запущенного приложения/программы). И т.к. она внутри одного процесса, то переключение между потоками проще и дешевле для процессора.

Еще раз напомним, что современные процессоры — многоядерные. По сути у вас не один процессор, а много. Шаги нашей программы, которые написаны последовательно одна команда за другой, выполняются тоже последовательно. По сути процессор не может загрузить все ядра одновременно — задача ведь выполняется последовательно. Параллельных вычислений в ней нет. (хоть JVM, которая выполняет нашу программу, пытается это сделать). Попытку мы видим — ядра загружены работой только частично, но все. Одно больше других.

Если проводить аналогии, то представьте, что одну траншею копают 4 человека, у которых одна лопата. Один копает, а другие в какой-то части могут ему подсказать советом или ногами откинуть землю. Но в основном работает один. В конце концов кто-то может забрать у товарища лопату, чтобы тот отдохнул. Но тем не менее лопата одна и использовать ее может только один. Тоже самое происходит и с нашей программой — у нас одна последовательность команд, которая выполняется тоже последовательно. И загрузить сразу все ядра мы не можем.

Даже если у вас одно ядро, все равно много потоков могут быть полезны. Допустим, что у нас есть какое-нибудь приложение, которое принимает запросы пользователей. Представьте себе, что запрос требует обращения к внешней системе из нашего приложения. Т.е. запрос получили и сразу передали дальше. Система отвечает не очень быстро и на время, пока она «думает» мы могли бы «отвлечь» процессор на другие запросы. Допустим, что пришло 10 запросов, каждый из них передается во внешнюю систему, которая думает около секунды. Предположим, что затраты на то, чтобы сформировать запрос к этой системе, крошечные. И чтобы отдать пришедший ответ пользователю затраты тоже минимальны. Но если запросы будут обрабатываться последовательно, то 10 запросов мы обработаем за 10 секунд. Если же мы сможем сделать так, чтобы как только мы отправили запрос во внешнюю систему, мы могли бы переключиться на обработку другого запроса и отправку его опять во внешнюю систему, то все 10 запросов могут быть отправлены туда друг за другом очень быстро. Отправка ответов системе обратно пользователю тоже происходит мгновенно. Значит задержка для всех 10-ти пользователей составит чуть больше секунды. Это же здорово. 10 запросов за секунду вместо 10 и заметим, что процессор одноядерный. Нам просто дали возможность использовать «простаивающий» процессор.

Потоки позволяют в ручном режиме запускать несколько задач в рамках одного приложения. В самом простом варианте внутри одного приложения всегда есть один основной поток. Всегда. А вот дополнительные потоки мы должны запускать сами. И запуск — не самое сложное. Самое сложное заставить их работать совместно. Но об этом мы поговорим позже. А пока давайте просто научимся запускать потоки.

Как создавать потоки

Для запуска потоков (можете улыбнуться) существует специальный класс — **Thread** (поток). Также есть еще два интерфейса, но как с ними работать, мы разберем несколько позже. Напишем самое простое приложение и разберем его в подробностях.

```
1 public class SimpleThreadManager
2 {
3     public static void main(String[] arg) {
4         for(int i=0; i<10; i++) {
5             Thread first = new SimpleThread();
```

```

6         first.start();
7     }
8 }
9 }
10
11 class SimpleThread extends Thread
12 {
13     @Override
14     public void run() {
15         try {
16             long pause = Math.round(Math.random()*2000);
17             Thread.sleep(pause);
18             System.out.println("Simple Thread - pause="+pause);
19         } catch (InterruptedException i_ex) {
20
21         }
22     }
23 }

```

Мы написали два класса. Один — **SimpleThreadManager** занимается тем, что в цикле запускает 10 потоков. При каждом выполнении цикла он создает объект класса **SimpleThread** (который наследуется от **Thread**). Глядя на присваивание сразу вспоминаем полиморфизм 😊 У класса **Thread** есть метод **start**, который и запускает поток на выполнение. Будьте внимательны — именно метод **start**, а не **run**. Второй класс — **SimpleThread**. В первую очередь отметим, что он наследуется от класса **Thread**. Во-вторых — мы переопределяем метод **run**. Именно этот метод и выполняется в отдельном потоке. Обратите внимание на этот факт: поток запускается методом **start**, а вся работа происходит в методе **run**.

Сам метод **run** делает достаточно несложную работу — он вычисляет случайное время задержки. **Math.random()** возвращает случайное число от 0 до 1, мы его умножаем на 2000 и округляем с помощью функции **Math.round()**. В итоге получаем случайное число от 0 до 2000. Потом вызов статического метода **Thread.sleep()** ставит на паузу текущий поток, потом печатается сообщение, в котором указывается на сколько миллисекунд поток «засыпал». Также обратите внимание, что вызов **Thread.sleep()** может порождать исключение **InterruptedException**, которое мы обрабатываем (ничего не делаем)

Можно использовать анонимный класс и тогда код будет выглядеть вот так:

```

1 public class SimpleThreadManager {
2
3     public static void main(String[] arg) {
4         for (int i = 0; i < 10; i++) {
5             Thread first = new Thread() {
6                 @Override
7                 public void run() {
8                     try {
9                         long pause = Math.round(Math.random() * 2000);

```

```

10         Thread.sleep(pause);
11         System.out.println("Simple Thread - pause=" + pause);
12     } catch (InterruptedException i_ex) {
13     }
14 }
15 };
16 first.start();
17 }
18 }
19 }

```

Как видите, ничего сложного нет — создаем класс-наследник от **Thread**, в нем переопределяем метод **run**. Этот метод делает всю нужную работу. И держим в голове мысль, что этот метод будет выполняться как-бы параллельно относительно других потоков. Потом где в нужном месте кода создаем наш класс и вызываем у него метод **start**. Вуаля. Все достаточно просто. На первый взгляд.

Рассмотрим еще один способ создания класса для запуска в отдельном потоке. В этот раз мы используем интерфейс — **Runnable**. Работа с ним очень похожа на вариант с классом **Thread** с небольшими отличиями. Давайте сразу посмотрим пример:

```

1 public class SimpleThreadManager
2 {
3     public static void main(String[] arg) {
4         for(int i=0; i<10; i++) {
5             Runnable first = new SimpleRunnable();
6             Thread t = new Thread(first);
7             t.start();
8         }
9     }
10 }
11
12 class SimpleRunnable implements Runnable
13 {
14     @Override
15     public void run() {
16         try {
17             long pause = Math.round(Math.random()*2000);
18             Thread.sleep(pause);
19             System.out.println("Simple Thread - pause="+pause);
20         } catch (InterruptedException i_ex) {
21         }
22     }
23 }
24 }

```


Сначала предлагаю посмотреть на код класса **SimpleRunnable**. Как видим он не сильно отличается от предыдущего случая — отличие состоит в том, что мы не наследуемся от класса **Thread**, а реализуем интерфейс **Runnable**. Переопределяемый метод точно такой же, как и был раньше — **run**. отличия минимальны.

Теперь обратим внимание на класс **SimpleThreadManager**. Как видим здесь мы используем несколько иной алгоритм запуска потока. Сначала создаем объект класса **SimpleRunnable**, потом создаем объект класса **Thread**, которому в конструкторе передаем наш объект. И уже после этого вызываем метод **start**. В момент вызова **start** класс **Thread** проверяет, не передавали ли ему в конструкторе объект, реализующий интерфейс **Runnable**. Если да — то запускается метод **run** переданного объекта. Если нет — объект **Thread** будет выполнять свой метод **run**. В общем тоже достаточно несложно.

Вы можете задуматься — а зачем создали интерфейс, когда есть класс **Thread** ? Ответ в общем-то очевиден — в чрезвычайной гибкости. Ведь в этом случае запускаться в отдельном потоке может какой угодно класс. Т.е. мы можем создать класс для сложного расчета — например, для вычисления прибыльности работ/закупок/и т.д. по договору. Это и хождения в базу данных, и какие-то файлы, возможно внешние запросы в другие системы. В общем — долго. И таких договоров штук 40. Теперь мы можем запускать расчет нескольких договоров параллельно и при этом этот сложный класс совсем не должен быть наследником **Thread**.

Передача параметров

Рассмотрим пример многопоточного приложения, коорый рисует на экране форму с часами. Наши часы будут обновляться отдельным потоком. Приведем код этого примера и потом прокомментируем.

```
1 package edu.javacourse.clock;
2
3 import java.awt.Font;
4 import java.text.SimpleDateFormat;
5 import java.util.Calendar;
6 import javax.swing.JFrame;
7 import javax.swing.JLabel;
8 import javax.swing.SwingConstants;
9 import javax.swing.SwingUtilities;
10
11 public class SimpleClock extends JFrame
12 {
13     private JLabel clockLabel = new JLabel();
14
15     public SimpleClock() {
16         // Установить заголовок
17         setTitle("ClockThread");
18
19         // Выравнивать метку по горизонтали - есть такой метод у Label
20         clockLabel.setHorizontalAlignment(SwingConstants.CENTER);
21     }
```

```

22 // Установить размер шрифта для метки - есть такой метод у Label
23 // Для эт ого создаем шрифт и сразу его отдаем методу setFont
24 Font f = new Font("Default", Font.BOLD + Font.ITALIC, 24);
25 clockLabel.setFont(f);
26
27 // Добавить метку на основную панель окна
28 getContentPane().add(clockLabel);
29
30 // Установить размеры окна
31 setBounds(400, 300, 300, 100);
32
33 // ОБРАТИТЬ ВНИМАНИЕ !!!
34 // Создаем отдельный поток, который должен обновлять значение метки
35 Thread thr = new MyThread(this);
36 thr.start();
37 }
38
39 public void setTime() {
40     // Создаем объект для форматирования даты
41     SimpleDateFormat df = new SimpleDateFormat("dd.MM.yyyy HH:mm:ss");
42     // Устанавливаем новое значение для метки - сразу форматируем дату в строку и устанавливаем новый текст
43     clockLabel.setText(df.format(Calendar.getInstance().getTime()));
44 }
45
46 public static void main(String[] args) {
47     SimpleClock cl = new SimpleClock();
48     cl.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
49     cl.setVisible(true);
50 }
51 }
52
53
54 class MyThread extends Thread {
55
56     private SimpleClock clock;
57
58     public MyThread(SimpleClock clock) {
59         this.clock = clock;
60     }
61
62     @Override
63     public void run() {
64         while (true) {
65             clock.setTime();
66             try {
67                 Thread.sleep(500);
68             } catch (Exception e) {

```

```
69 |         }
70 |     }
71 | }
72 | }
```

Решение достаточно простое — мы создаем форму, на которую кладем текстовый компонент класса **JLabel**. Подготовка необходимых параметров происходит в конструкторе класса **SimpleClock** (строки 17-31). Здесь мы устанавливаем заголовок окна, потом создаем шрифт и устанавливаем его у текстового компонента **clockLabel**. Потом кладем компонент на форму и устанавливаем уже размеры формы. В самом конце (строки 35-36) создаем класс для потока **MyThread** и запускаем его. Также класс **SimpleClock** имеет метод **setTime**, который получает текущее время путем вызова **Calendar.getInstance().getTime()** и преобразует его в строку с помощью объекта класса **SimpleDateFormat** (строки 46-50). Этот метод надо будет вызывать из потока.

В классе **MyThread** есть несколько моментов, на которые я хочу обратить ваше внимание.

1. Наш поток должен вызывать метод **setTime** у объекта класса **SimpleClock**. Для этого поток должен иметь ссылку на этот объект — я описывал это в статье [Отношения между классами](#). Но как передать этот объект в метод **run** — там же нет параметров. Наиболее удобным способом (на мой взгляд) является создание полей в классе-потоке, для хранения ссылок на нужные объекты и инициализация их путем передачи параметров. Я сделал это в виде параметра для конструктора, но точно также можно сделать это через вызовы сеттеров (предлагаю вам самим реализовать такой вариант — это несложно). Теперь внутри метода **run** мы имеем доступ к нужному нам объекту через поле **clock**.

2. Наш метод **run** делает бесконечный цикл — условие вечно истинное. Т.е. наш поток может исполняться теоретически бесконечно. Как его правильно останавливать — узнаем чуть позже. А пока обратим внимание, что внутри нашего бесконечного цикла мы делаем очень простую работу — вызываем метод для обновления времени и потом засыпаем на секунды.

Если быть более точным, то наш метод **setTime** обновляет графический компонент достаточно некорректно. На самом деле мы должны использовать специальный поток (да, JVM запускает отдельный поток именно для графики). Вот более корректный вариант — пока не буду его комментировать. Просто примите как данность — графика не просто так должна обновляться, а с помощью специального вызова — создается расширяющий интерфейс **Runnable** класс и он отдается классу **SwingUtilities**, который вызовет его в нужное время. Больше не будут углубляться — если захотите, сами почитаете. Это может быть интересно.

```
1 package edu.javacourse.clock;
2
3 import java.awt.Font;
4 import java.text.SimpleDateFormat;
5 import java.util.Calendar;
6 import javax.swing.JFrame;
7 import javax.swing.JLabel;
8 import javax.swing.SwingConstants;
9 import javax.swing.SwingUtilities;
10
11 public class SimpleClock extends JFrame
12 {
13     private JLabel clockLabel = new JLabel();
```

```
14
15 public SimpleClock() {
16     // Установить заголовок
17     setTitle("ClockThread");
18
19     // Выравнивать метку по горизонтали - есть такой метод у Label
20     clockLabel.setHorizontalAlignment(SwingConstants.CENTER);
21
22     // Установить размер шрифта для метки - есть такой метод у Label
23     // Для эт ого создаем шрифт и сразу его отдаем методу setFont
24     Font f = new Font("Default", Font.BOLD + Font.ITALIC, 24);
25     clockLabel.setFont(f);
26
27     // Добавить метку на основную панель окна
28     getContentPane().add(clockLabel);
29
30     // Установить размеры окна
31     setBounds(400, 300, 300, 100);
32
33     // ОБРАТИТЬ ВНИМАНИЕ !!!
34     // Создаем отдельный поток, который должен обновлять значение метки
35     Thread thr = new MyThread(this);
36     thr.start();
37 }
38
39 public void setTime() {
40     // Более корректный вызов в отдельном потоке, который отвечает за графику
41     SwingUtilities.invokeLater(new Runnable() {
42         public void run() {
43             SimpleDateFormat df = new SimpleDateFormat("dd.MM.yyyy HH:mm:ss");
44             clockLabel.setText(df.format(Calendar.getInstance().getTime()));
45         }
46     });
47 }
48
49 public static void main(String[] args) {
50     SimpleClock cl = new SimpleClock();
51     cl.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
52     cl.setVisible(true);
53 }
54 }
55
56
57 class MyThread extends Thread {
58
59     private SimpleClock clock;
60
```

```

61     public MyThread(SimpleClock clock) {
62         this.clock = clock;
63     }
64
65     @Override
66     public void run() {
67         while (true) {
68             clock.setTime();
69             try {
70                 Thread.sleep(500);
71             } catch (Exception e) {
72             }
73         }
74     }
75 }

```

Остановка потока

Полагаю, что вы уже догадались, что поток исполняется до тех пор, пока работает метод **run**. В нашем предыдущем примере мы сделали его бесконечным. В каких-то задачах поток может закончиться сам — просто потому что закончилось какое-то вычисление. Но что делать, когда поток надо остановить извне.

Особенность этой ситуации в том, что надо очень хорошо себе представить правильное поведение двух участников процесса — того, кто хочет, чтобы поток закончился (назовем его инициатор окончания) и самого потока.

Так вот — ПРАВИЛЬНЫМ поведением считается такое, когда инициатор посылает сообщение самому потоку с просьбой прекратить свою работу и уже сам поток решает, когда это надо сделать. До сих пор у потока существует метод **destroy()**, который позволяет прекратить поток, но его вызов считается крайне плохим решением.

Давайте попробуем рассмотреть все это на примере. Предположим, нам надо создать приложение «Часы» (подобное предыдущему), которое должно управлять отображением времени — добавим две кнопки, одна из которых будет запускать наши часы, а другая — останавливать.

```

1  package edu.javacourse.clock;
2
3  import java.awt.BorderLayout;
4  import java.awt.Font;
5  import java.awt.event.ActionEvent;
6  import java.awt.event.ActionListener;
7  import java.text.SimpleDateFormat;
8  import java.util.Calendar;
9  import javax.swing.JButton;
10 import javax.swing.JFrame;
11 import javax.swing.JLabel;
12 import javax.swing.SwingConstants;

```

```
13 import javax.swing.SwingUtilities;
14
15 public class StartStopClock extends JFrame implements ActionListener
16 {
17     private static final String START = "START";
18     private static final String STOP = "STOP";
19
20     private final JLabel clockLabel = new JLabel();
21     private ClockThread clockThread = null;
22
23     public StartStopClock() {
24         // Установить заголовок
25         setTitle("ClockThread");
26
27         // Выравнивать метку по горизонтали - есть такой метод у Label
28         clockLabel.setHorizontalAlignment(SwingConstants.CENTER);
29
30         // Установить размер шрифта для метки - есть такой метод у Label
31         // Для эт ого создаем шрифт и сразу его отдаем методу setFont
32         Font f = new Font("Default", Font.BOLD + Font.ITALIC, 24);
33         clockLabel.setFont(f);
34
35         // Добавить метку на основную панель окна
36         add(clockLabel);
37
38         // Добавить кнопку для старта
39         JButton start = new JButton(START);
40         start.setActionCommand(START);
41         start.addActionListener(this);
42         add(start, BorderLayout.NORTH);
43
44         // Добавить кнопку для становки
45         JButton stop = new JButton(STOP);
46         stop.setActionCommand(STOP);
47         stop.addActionListener(this);
48         add(stop, BorderLayout.SOUTH);
49
50         // Установить размеры окна
51         setBounds(400, 300, 300, 200);
52
53     }
54
55     public void setTime() {
56         // Более корректный вызов в отдельном потоке, который отвечает за графику
57         SwingUtilities.invokeLater(new Runnable() {
58             @Override
59             public void run() {
```

```

60         SimpleDateFormat df = new SimpleDateFormat("dd.MM.yyyy HH:mm:ss");
61         clockLabel.setText(df.format(Calendar.getInstance().getTime()));
62     }
63 });
64 }
65
66 @Override
67 public void actionPerformed(ActionEvent ae) {
68     if (START.equals(ae.getActionCommand())) {
69         if (clockThread == null) {
70             clockThread = new ClockThread(this);
71             clockThread.start();
72         }
73     }
74     if (STOP.equals(ae.getActionCommand())) {
75         if (clockThread != null) {
76             clockThread.stopClock();
77             clockThread = null;
78         }
79     }
80 }
81
82 public static void main(String[] args) {
83     StartStopClock cl = new StartStopClock();
84     cl.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
85     cl.setVisible(true);
86 }
87 }
88
89
90 class ClockThread extends Thread {
91
92     private StartStopClock clock;
93     private volatile boolean isRunning = true;
94
95     public ClockThread(StartStopClock clock) {
96         this.clock = clock;
97     }
98
99     public void stopClock() {
100         isRunning = false;
101     }
102
103     @Override
104     public void run() {
105         while (isRunning) {
106             clock.setTime();

```

```

107         try {
108             Thread.sleep(500);
109         } catch (Exception e) {
110         }
111     }
112 }
113 }

```

Итак, за счет чего наша программа так замечательно работает — можно запустить и посмотреть ее в действии.

Начнем с метода **main**. В нем мы создаем нашу форму, выставляем ей специальное свойство, которое прекращает работу приложения при закрытии формы и делает нашу форму видимой (строки 84-86).

Далее рассмотрим конструктор (строки 24-52). В нем самое важное — это строки 39-48). Здесь мы создаем две кнопки START и STOP. Работа с ними практически одинаковая, поэтому рассмотрим только кнопку START. Сначала мы создаем кнопку. Вызов **start.setActionCommand(START)** устанавливает уникальный идентификатор команды, который будет отправлен слушателю. Следующий вызов **start.addActionListener(this)** регистрирует нашу форму **StartStopClock** в качестве слушателя у кнопки (мы раньше такое видели), но чтобы не бегать по статьям еще раз опишем — кнопка при нажатии на нее проверяет список своих слушателей и каждому посылает сообщение типа **ActionEvent**. Они могут быть какого угодно класса, но каждый из них должен реализовать интерфейс **ActionListener**, что наша форма и делает — у нее есть метод **actionPerformed**. Можно сказать, что кнопка собрала у всех, кто ее хотел слушать телефоны и как только ее нажали, она всем позвонила. Единственное требование к тому, кому надо позвонить — у него должен быть телефон — метод **actionPerformed**. И заключительным шагом мы «кладем» нашу кнопку на форму методом **add**, в котором передаем кнопку и вторым параметром указываем ее местоположение на форме — для кнопки START это север (NORTH). Наша форма готова и кнопки при нажатии вызовут еще один важный элемент нашей программы — метод **actionPerformed**. Смотрим его реализацию — строки 68-81).

Когда кнопка вызывает этот метод, она передает ему специальный объект (**ActionEvent**), который описывает событие (его параметры). Нас в этом событии интересует только идентификатор команды (мы его установили методом **setActionCommand**. Если нажали кнопку START, то мы создаем поток, как и раньше это делали — ничего особенного. Проверка на **null** позволяет нам избежать двойного запуска (попробуйте убрать эту проверку и понажимайте кнопку START несколько раз подряд — потом попробуйте остановить наши часы кнопкой STOP).

Кнопка STOP позволяет нам обратиться к нашему классу потока и вызвать у него метод **stopClock()**, который как раз и предназначен для того, чтобы мы могли сообщить потоку, что хотим его остановить/прекратить.

И наконец самый главный участник нашего приложения — класс **ClockThread**.

Как уже мы видели раньше, ему передается ссылка на форму с часами и в методе **run** он в цикле периодически вызывает метод **setTime()**. Как вы уже видите, мы добавили специальную переменную **isRunning**, которая и определяет, продолжать ли наш цикл или нет. У нашего класса есть простой метод **stopClock()**, который устанавливает значение переменной **isRunning** в **false** и тем самым прекращает цикл. Все очень просто и элегантно — мы попросили поток остановиться и он, завершив очередной шаг, спокойно это сделал.

Это очень важно — МЫ ДАЛИ потоку ВОЗМОЖНОСТЬ сделать все необходимое, чтобы завершиться корректно — ведь в процессе работы поток (для примера) мог накапливать данные и периодически их куда-то записывать. Представим себе, что мы его грубо остановили и он бы не успел записать очередную порцию важных данных. Это плохо. Поэтому и предлагается именно наш вариант.

Осталось заключительное замечание. Вы наверняка обратили внимание на слово **volatile** в строке описания нашей переменной **isRunning** (его можно перевести как «непостоянный», «изменчивый»). Это слово используется, чтобы подсказать JVM — обращение к этой переменной может производиться из разных потоков, что у нас и получается — один поток меняет показания часов, а другой вызывается при нажатии на кнопку. Оба обращаются к этой переменной. Но зачем это надо JVM ?

Дало в том, что когда приложение выполняется, JVM постоянно пытается оптимизировать его исполнение. И иногда для скорости, в разных потоках может возникнуть по сути две копии переменной **isRunning**. Это не всегда происходит, но случается. Таким образом может возникнуть ситуация, что поток от кнопки изменит одну копию, а поток, который меняет часы и только читает переменную, будет использовать другую копию — и никогда не остановится.

Чтобы такого абсолютно точно не происходило, мы подсказываем JVM, чтобы она «обратила внимание» на нашу переменную и была осторожна при оптимизации. Вот собственно и все.

В качестве домашнего задания попробуйте написать форму, которая «бегает» по экрану слева направо (или сверху вниз) и при достижении границы начинает двигаться в другую сторону.

Теперь мы можем перейти к следующей части повествования о многопоточности, которая введет вас в мир леденящих кровь ужасов, возникающих при совместной работе нескольких потоков :).

И теперь нас ждет следующая статья: [Многопоточность и синхронизация](#).

23 comments to *Многопоточность — первые шаги*



Июль 24, 2015 at 23:37

[Владимир](#) says:

Замечательная и понятная статья. Спасибо!

[Reply](#)



Июль 25, 2015 at 09:49

admin says:

Спасибо. Я дополнил статью — теперь она содержит то, что я хотел. Но многопоточность на этой статье не закончится — там еще много интересного.

[Reply](#)



Июль 29, 2015 at 14:38

egor says:

Таким образом, процессор сам выбирает как распараллеливать процессы в многопоточной среде?

[Reply.](#)



Июль 29, 2015 at 14:47

admin says:

Там все очень сложно и я сам не все точно могу рассказать 😊 По сути потоки — это объекты операционной системы (Windows, Linux). JVM создает потоки, как объекты операционной системы и ими в какой-то степени управляет. Но сама ОС это тоже делает и именно она «загружает» процессор задачами и потоками.

Если честно — я не настолько глубоко погружался в JVM, чтобы ответить более подробно. Вполне допускаю, что даже в этих строчках я что-то напутал.

[Reply.](#)



Июль 29, 2015 at 19:07

egor says:

спасибо

[Reply.](#)



Ноябрь 22, 2015 at 12:19

Vitaly says:

Здравствуйте, а можете, пожалуйста, пояснить именно 1-е приложение где вы сравниваете работу вычислений в 1 поток и в многопоточности. В особенности класс BigTaskManyThreads.

[Reply](#)



o

Ноябрь 23, 2015 at 11:53

admin says:

В комментариях к примеру написано «Пока не вдавайтесь в подробности – просто запустите этот замечательный пример, но перед этим советую запустить диспетчер задач, чтобы посмотреть загрузку процессоров.». Данный пример не предполагается разбирать на части, пока не будут описаны основные конструкции многопоточности.

[Reply](#)



•

Август 19, 2016 at 21:41

Grif says:

Интересная статья, в принципе как и весь материал автора.

Единственно запускал весь код статьи на Ubuntu и там наблюдается весьма занятная картина для первого примера, который должен показать, что многопоточность быстрее — у меня вышло все наоборот, однопоточная задача выполнялась немного быстрее, зато у многопоточной всегда число немного больше.

Когда я посмотрел на индикаторы загрузки ЦП, выглядело все достаточно забавно — на первом этапе выполнения загрузка ядер ЦП (у моего ноута 4 ядра) была весьма хаотична, но загруженность ядер достигала почти 100% ... во второй части загрузка всех ядер была равномерной, но не превысила 50%.

[Reply](#)



o

Август 22, 2016 at 15:23

admin says:

Работа с потоками в конечном итоге контролируется операционной системой и Linux это делает иначе, нежели Windows. Так что такие варианты возможны — Linux старается не давать одному процессу загрузить все CPU очень сильно.

[Reply](#)



Август 25, 2016 at 14:42

Oleh says:

» сделал это в виде параметра для конструктора, но точно также можно сделать это через вызовы сеттеров (предлагаю вам самим реализовать такой вариант — это несложно).»

Несложно, но для новичка делать обязательно без подсказок, отличное задание для углубления понимания и закрепления ранее пройденного материала.

[Reply](#)



Сентябрь 5, 2016 at 15:38

Grif says:

Доброго времени суток уважаемый автор. Хочу попросить помощи относительно рассказанного вами материала, немного не к этой статье но в нужной комментарии писать нельзя. Подскажите пожалуйста как настроить Идею в следующей ситуации:

ВАЖНО !!! При работе с локализацией есть один момент — если вы собираетесь использовать русские буквы, то вы не сможете писать из напрямую.

Во-первых тот же NetBeans отслеживает такую ситуацию автоматически. IDEA тоже — правда ей надо об этом сказать в настройках.

[Reply](#)



o

Сентябрь 5, 2016 at 17:38

admin says:

Ну видимо Гугл вас не понял — вот можно посмотреть: <https://www.jetbrains.com/help/idea/2016.2/configuring-encoding-for-properties-files.html>

[Reply](#)



Октябрь 27, 2016 at 13:40

Валера says:

как многопоточность связана с заданием про форму которая «бегает» по экрану ?

я просто сделал в цикле изменение координат и метод
`form.setBounds(x1, y1, x2, y2);` без потоков

[Reply](#)



Ноябрь 2, 2016 at 16:10

admin says:

Тут вылезает особенность реализации рисования графики — она делается в отдельном потоке. Задача в общем просто позволяет попробовать потоки, а не решать задачу иначе.

[Reply](#)



Июнь 11, 2017 at 16:15

Я says:

Супер! Даже для не начинающих!

[Reply](#)



Июнь 13, 2017 at 22:46

Сергей says:

```
@Override
public void run() {
    while (true) {
        clock.setTime();
        try {
            Thread.sleep(500);
        } catch (Exception e) {
        }
    }
}
```

ПОЧЕМУ Thread.sleep(500);? а не Thread.sleep(1000);?? Разве нам не нужна секундная пауза?

[Reply](#)



o

Июнь 14, 2017 at 09:49

admin says:

Если будет секунда, то есть вероятность, что мы можем проскочить целую секунду и смена будет например с 43 секунды на 45 секунд. Гарантий, что тред включится ровно через секунду нет. Т.е. можно быть на паузе немного больше. Вот и получится «провал».

[Reply](#)



•

Апрель 5, 2018 at 13:59

Данила says:

А почему бы не показать вариант для java 8, лямбду вместо анонимного Runnable? Намного лаконичнее, и приобщает к прекрасному и неизведанному.

```
SwingUtilities.invokeLater(() -> {
    SimpleDateFormat df = new SimpleDateFormat("dd.MM.yyyy HH:mm:ss");
    clockLabel.setText(df.format(Calendar.getInstance().getTime()));
});
```

С большим удовольствием занимаюсь по Вашим материалам. Очень нравится направленность от потребности к теории, а также то, что результат выглядит добротно, а не оказывается слеппен из глины и проволоки, как во многих учебных проектах, где практически игнорируется вопрос архитектуры проекта.

Спасибо Вам большое.

[Reply](#)



o

Апрель 6, 2018 at 11:35

admin says:

Возможно, но статьи начинались давно, еще до 8-й версии. Может быть когда-нибудь я смогу выпустить новую редакцию, но тут уж как получится.

[Reply](#)



•

Сентябрь 6, 2018 at 13:15

andy says:

мне кажется, это не самая лучшая идея показывать новые концепции на основе кода, основанного на свинге, который по сути не изучался. лучше брать более общий код, не связанный с конкретной технологией.

[Reply](#)



o

Сентябрь 6, 2018 at 15:50

admin says:

Мнений может быть много. Свинг удобен тем, что очень наглядно можно посмотреть результат. Он не абстрактный, а очень конкретный, видимый.

А усилий больших не требуется.

[Reply](#)



Сентябрь 6, 2018 at 14:03

andy says:

`clockThread = null;` — это перестраховка?

[Reply](#)



Сентябрь 6, 2018 at 15:49

admin says:

Нет, это нужно, чтобы при нажатии кнопки можно было точно определить, что потока нет. Остановка потока не гарантирует, что указатель сразу станет равен NULL.

[Reply](#)

Leave a reply


Comment

You may use these HTML tags and attributes: ` <abbr title=""> <acronym title=""> <blockquote cite=""> <cite> <code class="" title="" data-url=""> <del datetime=""> <i> <q cite=""> <s> <strike> <pre class="" title="" data-url=""> `

Имя *

E-mail *

Сайт

семь - 1 = 

Add comment

Copyright © 2018 [Java Course](#)

Designed by [Blog templates](#), thanks to: [Free WordPress themes for photographers](#), [LizardThemes.com](#) and [Free WordPress real estate themes](#)