

THREADS

1. Выполнение инструкций потоками

Поток выполнения: **последовательность команд**, выполняемых процессором.

Другие названия:

ПОТОК ВЫЧИСЛЕНИЯ, **НИТЬ**, (англ.) **thread**.

Выполняемая программа может иметь **несколько ПОТОКОВ**.

Любую инструкцию (вызов метода, оператор, операция и т.п.) всегда выполняет некоторый **ПОТОК**.

2. Суперкласс потоков выполнения

`java.lang.Thread`

1. Поток - объект

Выражение «**на потоке вызван метод**» следует понимать как вызов метода на объекте соответствующего класса потока.

2. Поток - процесс выполнения команд

Выражение «**поток выполняет метод**» следует понимать как выполнение инструкций метода ПОТОКОМ.

3. Главный поток

Любая программа имеет хотя бы один поток вычислений — **главный поток**.

Точкой входа в главный поток является метод **main**.

```
public class test {  
    public static void main(String[] argv) {  
        // инструкции метода main  
        // выполняет главный поток программы  
    }  
}
```

Главный поток запускает JVM.

Все инструкции метода **main** выполняет главный поток.

4. Статические методы класса Thread

В любом месте программы можно вызывать статические методы класса **Thread**, которые относятся к текущему потоку, т.е. **к тому потоку, который вызывает эти методы.**

Например, ссылку на объект Thread текущего потока можно получить с помощью статического метода

```
Thread.currentThread();
```

5. Имя потока

Любому потоку можно присвоить имя – либо с помощью конструктора, либо с помощью метода `setName`. Имя потока возвращает метод `getName` (оба метода `setName` и `getName` определены в классе `Thread`).

```
public static void main(String[] argv) {  
    Thread.currentThread().setName("main");  
    System.out.println(  
        Thread.currentThread().getName()); // main  
}
```

Замечание. JVM не использует имена потоков, они служат только для удобства. **Двум разным потокам можно присвоить одно и то же имя.**

6. Создание и запуск дочернего потока (**extends Thread**)

Чтобы создать поток нужно расширить класс Thread, перекрыв его метод run

```
class B extends Thread {  
    public void run() {  
        // do something  
    }  
}
```

После создания поток можно запустить на выполнения с помощью метода `start` класса `Thread`.

```
B b = new B();  
b.start();  
// или  
new B().start();
```

Замечание:

run - точка входа в поток

main - точка входа в главный поток

(главный поток запускает JVM).

7. Создание и запуск дочернего потока (implements Runnable)

Другой способ создания потока заключается в реализации интерфейса **Runnable**, который имеет единственный метод `run`.

```
class B implements Runnable {  
    public void run() {  
        // do something  
    }  
}
```

После этого создают новый поток с помощью конструктора `Thread(Runnable target)`.

```
Thread t = new Thread(new B());  
t.start();  
// или  
new Thread(new B()).start();
```

Замечание. Класс `Thread` реализует интерфейс `Runnable`.

```
public class Thread implements Runnable
```

Замечание. В качестве параметра конструктору Thread может быть передан **объект класса, который наследует Thread.**

```
public class Test {  
    public static void main(String[] argv) {  
        Thread t = new Thread(  
            new MyThread());  
        t.start();  
    }  
}
```

```
class MyThread extends Thread {  
    public void run() {}  
}
```

8. Создание и запуск потока в одном классе

Поток можно создать и запустить в одном классе.

```
public class MyThread extends Thread {  
    public void run() {  
        ...  
    }  
  
    public static void main(String[] argv) {  
        // запуск потока  
        new MyThread().start();  
    }  
}
```

9. Запуск потока в конструкторе класса-потока

Поток можно запустить **в конструкторе** потока.

```
class MyThread extends Thread {  
    public MyThread() {  
        this.start();  
    }  
    public void run() {  
        // do something  
    }  
    public static void main(String[] argv) {  
        new MyThread ();  
    }  
}
```

10. Создание и запуск потока с помощью анонимного класса

Поток можно создать и запустить в методе с помощью **анонимного класса**.

```
public class MyThread {  
    public static void main(String[] argv) {  
        new Thread() {  
            public void run() {  
                // do something  
            }  
        }.start();  
    }  
}
```

11. Завершение выполнения потока

Поток начинает свое выполнение, когда на нем вызовут **метод start в родительском потоке**. Метод **start** в свою очередь вызывает метод **run**.

Поток завершает свое выполнение после выполнения последней инструкции метода **run**. Возможен выход из потоков в связи с выбросом исключений.

Аналогия для главного потока – главный поток завершает свое выполнение после выполнения последней инструкции метода **main**.

Замечание. Существуют т.н. «**ПОТОКИ-ДЕМОНЫ**», которые предназначены для обслуживания других потоков.

Если в программе запущенными остаются только потоки-демоны, то JVM принудительно прекращает их работу и завершает выполнение приложения.

Чтобы сделать поток «**ДЕМОНОМ**» необходимо *перед его запуском* вызвать на нем метод **setDaemon**, передав значение **true**.


```
public class test extends Thread {  
    public void run() { while(true); } // бесконечный цикл  
    public static void main(String[] argv)  
        throws InterruptedException {  
        // главный поток создает поток test  
        test t = new test();  
        t.setDaemon(true); // делает его «демоном»  
        t.start(); // запускает  
        // чтобы дочерний поток успел запуститься:  
        Thread.sleep(1000);  
    } // в данном месте главный поток завершает свое  
        // выполнение и JVM завершает работу  
        // приложения прерывая бесконечный цикл  
}
```

12. Метод sleep класса Thread

Класс Thread содержит **статический** метод **sleep**, который делает паузу в выполнении **текущего потока** на заданное число миллисекунд.

```
public static void sleep(long millis)  
    throws InterruptedException
```

Метод выбросит исключение **InterruptedException**, если на потоке для которого он делает паузу вызван метод **interrupt**.

```
public static void main(String[] argv) {  
    for (int j = 0; j < 10; j++) {  
        System.out.println(j);  
        // пауза главного потока  
        // примерно на 1 секунду  
        try {  
            Thread.sleep(1000);  
        }  
        catch (Exception e) {e.printStackTrace();}  
    }  
}
```

```
public class test extends Thread {  
    public void run() {  
        for (int j = 0; j < 10; j++) {  
            System.out.println(j);  
            try {Thread.sleep(200);}  
            catch (Exception e) {e.printStackTrace();}  
        }  
    }  
    public static void main(String[] args) throws  
    InterruptedException {  
        test t = new test(); t.start();  
        Thread.sleep(1000);  
        t.interrupt(); // через секунду будет выброс  
        // исключения методом sleep если он  
        // выполняется в данный момент  
    }  
}
```

Замечание. Метод `sleep` перегружен. Точность времени паузы не гарантируется.

// пауза на ms миллисекунд
`public static void sleep(long ms)`

// пауза на ms миллисекунд и ns наносекунд
`public static void sleep(long ms, int ns)`

Замечание. `InterruptedException` наследуется напрямую от `Exception`, поэтому необходимо предусмотреть обработку этого исключения.

13. Метод `alive` класса `Thread`

Метод `isAlive` класса `Thread` возвращает `true`, если поток, на котором он вызван, **запущен** и еще **не прекратил свое выполнение**.

```
public class test extends Thread {  
    public static void main(String[] argv) {  
        // гл. поток создает и запускает новый поток  
        test t = new test();  
        t.start();  
        // главный поток вызывает метод  
        // isAlive на запущенном потоке:  
        while (boolean isAlive = t.isAlive()) {  
            // будет выводиться true пока  
            // выполняется запущенный поток:  
            System.out.println(isAlive);  
        } // число циклов заранее неизвестно  
    } // зависит от того, как долго будет  
        // выполняться метод run  
    public void run() {}  
}
```

14. Пример запуска двух потоков, которые выводят разные сообщения с разной периодичностью

```
public class test extends Thread {  
    private String mes;  
    private int ms;  
  
    // конструктор потока  
    public test(String mes, int ms) {  
        this.mes = mes;  
        this.ms = ms;  
        // запуск потока в конструкторе  
        this.start();  
    }  
}
```



```
public void run() {  
    while (true) { // бесконечный цикл  
        // вывод сообщения  
        System.out.println(this.mes);  
        // пауза на ms миллисекунд  
        try {sleep(this.ms);}   
        catch (Exception e) {}  
    }  
}  
// главный поток запускает три потока  
public static void main(String[] argv) {  
    new test("A", 500); // первый  
    new test("B", 700); // второй  
    new test("C", 1000); // третий  
}
```

15. Единственность способа запуска потока.

Запустить дочерний поток можно ТОЛЬКО с помощью метода **start**.

Если на объекте потока вызвать метод **run**, то это приведет лишь к одноразовому выполнению тела данного метода в том потоке, который вызвал run.

```
public class test extends Thread {
    public void run() {
        Thread t = Thread.currentThread();
        System.out.println(t.getName());
    }
    public static void main(String[] argv) {
        test t = new test();
        t.run();    // main
        t.start(); // thread-0
    }
}
```

Замечание. JVM задает каждому потоку некоторое системное имя при его создании

```
test t = new test();
```

Это имя возвращает метод `getName`, если его вызывать на соответствующем объекте-потоке.

```
t.getName();
```

Если требуется получить имя потока, который выполняет некоторый метод, достаточно в код метода вставить следующую инструкцию:

```
String threadName = Thread.currentThread().getName();
```

16. Проблема параллельного выполнения одного кода разными потоками

Два разных потока могут выполнять один и тот же код, это может привести к нежелательным последствиям.

```
public class test {  
    private boolean flag;  
    // выполняют два потока параллельно  
    public void m(boolean flag) {  
        this.flag = flag;  
        try {Thread.sleep(200);} // ждем  
        catch (InterruptedException e)  
    {e.printStackTrace();}  
        System.out.println(this.flag + " == " + flag);  
    } // true == false
```

```
public test() { // конструктор создает два потока
    new Thread() { // создание первого потока
        public void run() {
            while (true) m(true);
        }
    }.start(); // запуск первого потока
    new Thread() { // создание второго потока
        public void run() {
            while (true) m(false);
        }
    }.start(); // запуск второго потока
}
public static void main(String[] argv) {
    new test();
}
```

17. Синхронизация

Для разрешения проблемы параллельного доступа разных потоков к одному и тому же коду применяется синхронизация.

Синхронизации может быть подвергнут метод

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

статический метод

```
public static synchronized void m() {...}
```

участок кода метода.

```
public void m() {  
    synchronized(o) {...}
```

```
}
```

Синхронизация осуществляется с помощью специального **объекта-монитора**, который *связан с этой синхронизацией*.

Когда поток начинает выполнять синхронизированный код, говорят, что он **блокирует соответствующий монитор** (или владеет монитором).

Если потоку нужно выполнить синхронизированный код, а **монитор заблокирован другим потоком**, то он **находится в заблокированном состоянии**, пока с монитора не будет снята **блокировка**.

18. Монитор синхронизации

Монитором всегда является объект, который **зависит от того, какой код синхронизируется.**

<i>синхронизируемый код</i>	<i>монитор</i>
нестатический метод	объект this
статический метод	объект типа Class, соответствующий классу, в котором определен метод
участок кода	произвольный объект o - параметр инструкции synchronized

Замечание. Монитором не может быть **null**.

Замечание. Т.к. JVM создает уникальный объект типа Class, для каждого загруженного класса, то для **статических синхронизированных** методов существует **один и только один монитор**.

Для **нестатического синхронизированного** метода может существовать **несколько мониторов** – по числу созданных объектов класса, в котором определен этот метод.

```
public class test extends Thread {  
    public synchronized void m() {  
        while (true) System.out.println(  
            Thread.currentThread().getName());  
    }  
}
```

```
public test() {this.start();}  
public void run() {this.m();}
```

```
public static void main(String[] argv) {  
    test treadA = new test(); // запуск потока А  
    test threadB = new test(); // запуск потока В  
}  
// в процессе выполнения будут выводиться  
// имена потоков threadA и threadB вперемешку
```

В предыдущем примере два потока А и В выполняют один и тот же метод одновременно, т.к. они блокируют разные мониторы: **threadA** и **threadB** соответственно (для этого случая синхронизация не работает).

Если метод `m` сделать статическим, то на экран сообщения будет выводить только один поток – А. Второй поток В будет бесконечно долго ждать, пока первый **не разблокирует монитор**, который в данном случае один: объект класса `Class`, соответствующий классу `test` (**`test.class`**).

19. Инструкция `synchronized`

Инструкция `synchronized` позволяет синхронизировать произвольный участок кода метода.

```
public class test {  
    // объект-монитор  
    private Object lockA = new Object();  
    public void m() {  
        synchronized (lockA) { ... }  
    }  
}
```

Замечание. Параметром инструкции `synchronized` всегда является *объект*.

20. Метод join класса Thread

Класс Thread содержит метод join, предназначенный для перевода потока, который вызвал этот метод в режим паузы до тех пор, пока не закончит свою работу поток, на котором этот метод был вызван.

Замечание. Метод выбрасывает исключение **InterruptedException**, если в потоке, который его вызвал установить т.н. внутренний флаг прерывания потока (вызвать на этом же объекте потока метод **interrupt**).

Замечание. Поток, выполняющий метод **join** не снимает блокировки с мониторов, которые он заблокировал.

```
class test extends Thread {  
    public void run() {  
        try {sleep(2000);}  
        catch (Exception e) {e.printStackTrace();}  
    }  
    public static void main(String[] argv)  
        throws Exception{  
        // создание и запуск дочернего потока:  
        test t = new test();  
        t.start();  
        // главный поток ждет завершения  
        // дочернего потока:  
        t.join();  
        System.out.println("pause has expired");  
    }  
}
```


21. Метод wait класса Object

Метод wait переводит поток в режим ожидания.
Метод имеет три варианта.

wait()	бесконечное ожидание
wait(long ms)	ожидание ms миллисекунд
wait(long ms, int ns)	ожидание ms миллисекунд и ns наносекунд

Замечание. wait(0) эквивалентно wait()

Метод wait должен вызываться

- (1) только из синхронизированного кода
- (2) на объекте мониторе, связанным с данным синхронизированным кодом.
- (3) **поток**, который вызывает эти методы, **должен владеть монитором**.

wait может выбросить исключение **InterruptedException**.

Замечание. При вызове метода wait на мониторе, блокировка с этого монитора снимается и поток переходит в режим ожидания на этом мониторе.

Замечание. Метод wait определен в классе Object. Монитором может быть любой объект.

```
public class test extends Thread {
```

```
    // точка входа в поток:
```

```
    public void run() {
```

```
        // синхронизированный блок (монитор = this):
```

```
        synchronized (this) {
```

```
            // перевод this в режим ожидания:
```

```
            try {wait();}
```

```
            catch (Exception e) {
```

```
                // вывести сообщение после
```

```
                // прерывания:
```

```
                System.out.println(  
                    "thread has been interrupted");
```

```
            }
```

```
        }
```

```
    }
```

```
public static void main(String[] argv)
```

```
throws Exception {
```

```
// создание и запуск потока:
```

```
test t = new test();
```

```
t.start();
```

```
// пауза в главном потоке, чтобы успел
```

```
// начать свое выполнение метод wait:
```

```
Thread.sleep(500);
```

```
// прервать запущенный поток:
```

```
t.interrupt();
```

```
}
```

```
}
```

Замечание. Когда поток выполняет метод `wait` возможны т.н. случайные пробуждения. Поэтому выполнение метода `wait` следует заключать в цикл с проверкой условия.

Контракт класса `Object`:

As in the one argument version, **interrupts and spurious wakeups are possible**, and this method should always be used in a loop:

```
synchronized (obj) {  
    while (<condition does not hold>)  
        obj.wait(timeout, nanos);  
    ... // Perform action appropriate to condition  
}
```

22. Методы notify и notifyAll класса Object

Для выхода потока из режима ожидания применяют методы notify и notifyAll, которые должны:

- (1) *вызываться на объекте-мониторе*
- (2) *только из синхронизированного кода.*
- (3) *поток, который вызывает эти методы, должен владеть монитором.*

На одном и том же мониторе может находиться несколько потоков в режиме ожидания.

Метод `notify` пробуждает только один **случайно выбранный** поток; `notifyAll` пробуждает все потоки, которые ожидают на этом мониторе.

Замечание. Методы `notify` и `notifyAll` определены в классе `Object`, т.к. монитором потенциально может являться любой объект.

Замечание. Методы `notify` и `notifyAll` выбрасывают `IllegalMonitorStateException` с сообщением «`current thread not owner`», если поток, который вызывает эти методы, не является владельцем монитора.

Замечание. Нежелательно использовать метод `notify`, т.к. неизвестно какой именно поток получит уведомление.

```
public class test extends Thread {  
    public void run() {  
        // монитором является this /*1*/  
        synchronized (this) {  
            try {  
                // перевод в режим ожидания:  
                wait();  
                // вывод сообщения при  
                // пробуждении:  
                System.out.println(  
                    "thread has been notified");  
            } catch (Exception e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

```
public static void main(String[] argv)
```

```
    throws Exception {
```

```
        test t = new test();
```

```
        t.start();
```

```
        Thread.sleep(500);
```

```
        // главный поток входит в монитор t
```

```
        // (= this в строке /*1*/) 
```

```
        synchronized (t) {
```

```
            // и выполняет вызов notify на мониторе
```

```
            // которым владеет главный поток
```

```
            t.notify();
```

```
        }
```

```
    }
```

```
}
```

23. Блокированное состояние пробужденного потока

Если поток находился в состоянии ожидания, выполняя метод `wait` и был пробужден методом `notify/notifyAll` то он находится в блокированном состоянии до тех пор, пока не освободиться монитор и только после этого продолжает выполнять первую инструкцию за методом `wait`.

```
public class test extends Thread {  
    public void run() {  
        // "abc" – уникальный строковый литерал  
        synchronized ("abc") {  
            try {  
                // поток вызывает на мониторе  
                // "abc" метод wait  
                "abc".wait();  
                // вывод на экран после  
                // выполнения строки /*1*/  
                System.out.println(  
                    "thread has been notified");  
            }  
            catch (Exception e) {e.printStackTrace();}  
        }  
    }  
}
```

```
public static void main(String[] argv)
```

```
    throws Exception {
```

```
        new test().start();
```

```
        Thread.sleep(500);
```

```
        synchronized ("abc") {
```

```
            // главный поток вызывает на
```

```
            // мониторе "abc" метод notifyAll
```

```
            "abc".notifyAll();
```

```
            // главный поток переводится в режим
```

```
            // паузы на примерно на 10 с. /*1*/
```

```
            Thread.sleep(10000);
```

```
        }
```

```
    }
```

```
}
```

24. Совместное использование методов `wait` и `notify/notifyAll`

Совместное использование методов `notify/notifyAll` и `wait` дает возможность потокам синхронизировать свое выполнение.

Замечание. Если на мониторе нет ожидающих потоков, то вызов методов `notify/notifyAll` не генерирует никаких исключительных ситуаций.

```
public class test extends Thread {  
    public static Object lock = new Object();  
    private String mes;  
    public test(String mes) {this.mes = mes;}  
    public void run() {  
        synchronized (test.lock) {  
            try {  
                while (true) {  
                    test.lock.wait();  
                    test.lock.notify();  
                    System.out.println(this.mes);  
                }  
            }  
            catch (Exception e) {e.printStackTrace();}  
        }  
    }  
}
```



```
public static void main(String[] argv)
```

```
throws Exception {
```

```
// после запуска оба потока переходят в
```

```
// режим ожидания, выполняя wait
```

```
new test("A").start();    new test("B").start();
```

```
// пауза, чтобы запущенные потоки успели
```

```
// начать выполнение wait
```

```
Thread.sleep(500);
```

```
synchronized (test.lock) {
```

```
// для оповещения всех ждущих на
```

```
// мониторе потоков нужно вызвать на
```

```
// нем метод notifyAll
```

```
test.lock.notifyAll(); //
```

```
}
```

```
}
```

Замечание. Потоки могут синхронизировать свое выполнение без участия третьего (главного) потока.

```
public class test extends Thread {  
    private String mes;  
    public test(String mes) {this.mes = mes;}  
  
    public static void main(String[] argv)  
        throws Exception {  
        new test("A").start();  
        new test("B").start();  
    }  
}
```

```
public void run() {  
    // монитор - объект класса Class,  
    // соответствующий классу test  
    synchronized (test.class) {  
        try {  
            while (true) {  
                test.class.notify();  
                test.class.wait();  
                System.out.println(this.mes);  
            }  
        }  
        catch (Exception e) {e.printStackTrace();}  
    }  
}  
}
```

25. Взаимная блокировка

Потоки могут входить в состояние взаимной блокировки.

```
public class test extends Thread {  
    public static Object lockA = new Object();  
    public static Object lockB = new Object();  
  
    public void run() {  
        try {  
            synchronized (test.lockA) {  
                // после паузы поток ждет, пока  
                // будет снята блокировка с  
                // монитора test.lockB  
                sleep(500);  
                synchronized (test.lockB) {}  
            }  
        } catch (Exception e) {e.printStackTrace();}  
    }  
}
```

```
public static void main(String[] argv)
    throws Exception {
    new test().start();
    synchronized (test.lockB) {
        // после паузы поток ждет, пока будет
        // снята блокировка с монитора
        // test.lockA
        sleep(300);
        synchronized (test.lockA) {}
    }
}
}
```

26. Состояния потоков

Класс Thread содержит статический enum State, элементы которого представляют уникальные состояния потока. Состояние потока возвращает `Thread#getState`

- *NEW*
- *RUNNABLE*
- *BLOCKED*
- *TERMINATED*
- *WAITING*
- *TIMED_WAITING*

Состояние "новый":

NEW

Поток создан, но еще не запущен.

Состояние "работоспособный":

RUNNABLE

Поток выполняется в JVM.

Состояние "блокированный":

BLOCKED

Поток заблокирован на мониторе.

Состояние "остановленный":

TERMINATED

Поток завершил выполнение.

Состояние "ожидающий":

WAITING

Поток выполняет **wait/join** (без параметров).

Состояние "Ожидающий установленное время":

TIMED_WAITING

Поток выполняет **wait/join/sleep** (с параметрами)

28. Метод `interrupt` класса `Thread`

Если поток находится в состоянии *WAITING/TIMED_WAITING* выполняя методы `sleep/join/wait`, а другой поток вызывает на этом потоке метод `interrupt`, то соответствующие методы прекращают свое выполнение и выбрасывают исключение `InterruptedException`.

Замечание. Внутренний флаг прерывания потока в данном случае установлен не будет.

```
public class test extends Thread {  
    public void run() {  
        try {  
            sleep(10000); // InterruptedException  
            this.join(); // InterruptedException  
            synchronized (test.class) {  
                // InterruptedException:  
                test.class.wait();  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
public static void main(String[] argv)
    throws Exception {
    test t = new test();
    t.start();
    t.interrupt();
}
}
```

29. Методы `interrupted` и `isInterrupted` класса `Thread`

Потоки имеют внутренний флаг, который определяет, был ли прерван поток методом `interrupt`. Для проверки этого флага применяются методы `interrupted` и `isInterrupted`.

Замечание. Если во время вызова метода `interrupt` поток выполняет метод `sleep/join/wait`, то соответствующий метод выбросит исключение `InterruptedException`, при этом флаг прерывания будет сброшен в `false`.

```
public static boolean interrupted()
```

проверяет был ли **текущий** поток прерван и сбрасывает внутренний флаг в false.

```
public boolean isInterrupted()
```

проверяет был ли поток (на котором данный метод вызван) прерван, внутренний флаг при этом остается без изменений.

```
Thread t = Thread.currentThread();  
t.interrupt();
```

```
// true  
System.out.println(t.isInterrupted());
```

```
// true  
System.out.println(t.isInterrupted());
```

```
// true  
System.out.println(Thread.interrupted());
```

```
// false  
System.out.println(Thread.interrupted());
```

30. Изменение значения аргумента блока синхронизации

Присваивание аргументу блока синхронизации кода нового значения не приводит к смене монитора в нем.

Если в качестве аргумента синхронизированных блоков предполагается использовать специально созданное для этих целей поле, то целесообразно объявлять его с модификатором **final**.


```
public class test extends Thread {  
  
    private Object mon = "123";  
  
    public void m() throws Exception {  
        synchronized (mon) {  
            System.out.println("mon1=" + mon);  
  
            mon = "abc";  
            Thread.sleep(5000);  
            System.out.println("mon2=" + mon);  
        }  
    }  
}
```

```
public void run() {  
    try {m();}  
    catch (Exception ex) {}  
}
```

```
public static void main(String[] argv)  
    throws Exception {  
    test t = new test();  
    t.start();  
    Thread.sleep(1000);  
    synchronized ("123") {  
        System.out.println("mon3=" + t.mon);  
    }  
}
```

Программа выведет в стандартный поток вывода следующие строки (с паузой после первой строки в 5 секунд):

```
mon1=123
```

```
mon2=abc
```

```
mon3=abc
```

т.е., блокировка с объекта "123" не снимается.