Инструкция synchronized

Синхронизировать объект можно не только при помощи методов с соответствующим модификатором, но и при помощи синхронизированного блока кода. В этом случае происходит блокировка объекта, указанного в инструкции **synchronized**, и он становится недоступным для других синхронизированных методов и блоков. Обычные методы на синхронизацию внимания не обращают, поэтому ответственность за грамотную блокировку объектов ложится на программиста.

```
/* пример # 9 : блокировка объекта потоком: TwoThread.java */
package chapt14;
public class TwoThread {
     public static void main(String args[]) {
          final StringBuffer s = new StringBuffer();
          new Thread() {
               public void run() {
                    int i = 0;
                    synchronized (s) {
                         while (i++ < 3) {
                              s.append("A");
                          try {
                                   sleep(100);
                          } catch (InterruptedException e) {
                                   System.err.print(e);
                              }
                              System.out.println(s);
                         }
                    }//конец synchronized
               }
          }.start();
          new Thread() {
               public void run() {
                    int j = 0;
                    synchronized (s) {
                         while (j++ < 3) {
                              s.append("B");
                         System.out.println(s);
                    }
                   }//конец synchronized
                   }
          }.start();
     }
}
```

В результате компиляции и запуска будет, скорее всего (так как и второй поток может заблокировать объект первым), выведено:

```
A
AA
AAA
AAAB
AAABB
AAABBB
```

Один из потоков блокирует объект, и до тех пор, пока он не закончит выполнение блока синхронизации, в котором производится изменение значения объекта, ни один другой поток не может вызвать синхронизированный блок для этого объекта.

Если в коде убрать синхронизацию объекта **s**, то вывод будет другим, так как другой поток сможет получить доступ к объекту и изменить его раньше, чем первый закончит выполнение цикла.

В следующем примере рассмотрено взаимодействие методов **wait()** и **notify()** при освобождении и возврате блокировки в **synchronized** блоке. Эти методы используются для управления потоками в ситуации, когда необходимо задать определенную последовательность действий без повторного запуска потоков.

Метод **wait()**, вызванный внутри синхронизированного блока или метода, останавливает выполнение текущего потока и освобождает от блокировки захваченный объект, в частности объект **lock**. Возвратить блокировку объекта потоку можно вызовом метода **notify()** для конкретного потока или **notifyAll()** для всех потоков. Вызов может быть осуществлен только из другого потока, заблокировавшего, в свою очередь, указанный объект.

```
/* пример # 10 : взаимодействие wait() и notify(): Blocked.java: Runner.java */
package chapt14;

public class Blocked {
      private int i = 1000;

      public int getI() {
            return i;
      }
      public void setI(int i) {
            this.i = i;
      }
      public synchronized void doWait() {
                  try {
                        System.out.print("Не ");
                        this.wait(); /* остановка потока и
                                        освобождение блокировки*/
      System.out.print("сущностей "); // после возврата блокировки
                        Thread.sleep(50);
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
                  for (int j = 0; j < 5; j++) i/=5;
                  System.out.print("сверх ");
            }
      }
package chapt14;

public class Runner {
```

```
    public static void main(String[] args) {
        Blocked lock = new Blocked();
        new Thread() {
            public void run() {
                lock.doWait();
            }}.start();
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        synchronized (lock) {// 1
            lock.setI(lock.getI() + 2);
            System.out.print("преумножай ");
            lock.notify(); // возврат блокировки
        }
        synchronized (lock) {// 2
            lock.setI(lock.getI() + 3);
            //блокировка после doWait()
            System.out.print("необходимого. ");
            try {
                lock.wait(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.print("=" + lock.getI());
    }
}
```

В результате компиляции и запуска будет выведено следующее сообщение:

```
Не преумножай сущностей сверх необходимого. =3
```

Задержки потоков методом **sleep()** используются для точной демонстрации последовательности действий, выполняемых потоками. Если же в коде приложения убрать все блоки синхронизации, а также вызовы методов **wait()** и **notify()**, то вывод может быть следующим:

```
Не сущностей преумножай необходимого. =1005сверх
```

Состояния потока

В классе **Thread** объявлено внутреннее перечисление **State**, простейшее применение элементов которого призвано помочь в отслеживании состояний потока в процессе функционирования приложения и, как следствие, в улучшении управления им.

```
/* пример # 11 : состояния NEW, RUNNABLE, TIMED_WAITING, TERMINATED :
ThreadTimedWaitingStateTest.java */
package chapt14;
public class ThreadTimedWaitingStateTest extends Thread {
```