ПОТОКИ ВЫПОЛНЕНИЯ

Класс Thread и интерфейс Runnable

К большинству современных распределенных приложений (Rich Client) и Web-приложений (Thin Client) выдвигаются требования одновременной поддержки многих пользователей, каждому из которых выделяется отдельный поток, а также разделения и параллельной обработки информационных ресурсов. Потоки – средство, которое помогает организовать одновременное выполнение нескольких задач, каждую в независимом потоке. Потоки представляют собой классы, каждый из которых запускается и функционирует самостоятельно, автономно (или относительно автономно) от главного потока выполнения программы. Существуют два способа создания и запуска потока: расширение класса **Thread** или реализация интерфейса **Runnable**.

```
// пример # 1 : расширение класса Thread: Talk.java
package chapt14;

public class Talk extends Thread {
     public void run() {
          for (int i = 0; i < 8; i++) {
               System.out.println("Talking");
               try {
                    // остановка на 400 миллисекунд
                    Thread.sleep(400);
               } catch (InterruptedException e) {
                    System.err.print(e);
               }
          }
     }
}
```

При реализации интерфейса **Runnable** необходимо определить его единственный абстрактный метод **run()**. Например:

```
/* пример # 2 : реализация интерфейса Runnable: Walk.java: WalkTalk.java */
package chapt14;

public class Walk implements Runnable {
     public void run() {
          for (int  i = 0; i < 8; i++) {
               System.out.println("Walking");
               try {
                    Thread.sleep(400);
               } catch (InterruptedException e) {
                    System.err.println(e);
               }
          }
     }
}
package chapt14;

public class WalkTalk {
     public static void main(String[] args) {
          // новые объекты потоков
          Talk talk  = new Talk();
          Thread walk = new Thread(new Walk());
          // запуск потоков
          talk.start();
          walk.start();

          //Walk w = new Walk(); // просто объект, не поток
          // w.run(); //выполнится метод, но поток не запустится!
     }
}
```

Использование двух потоков для объектов классов **Talk** и **Walk** приводит к выводу строк: `Talking Walking`. Порядок вывода, как правило, различен при нескольких запусках приложения.

Жизненный цикл потока

При выполнении программы объект класса **Thread** может быть в одном из четырех основных состояний: “новый”, “работоспособный”, “неработоспособный” и “пассивный”. При создании потока он получает состояние “новый” (**NEW**) и

не выполняется. Для перевода потока из состояния “новый” в состояние “работоспособный” (**RUNNABLE**) следует выполнить метод **start()**, который вызывает метод **run()** – основной метод потока.

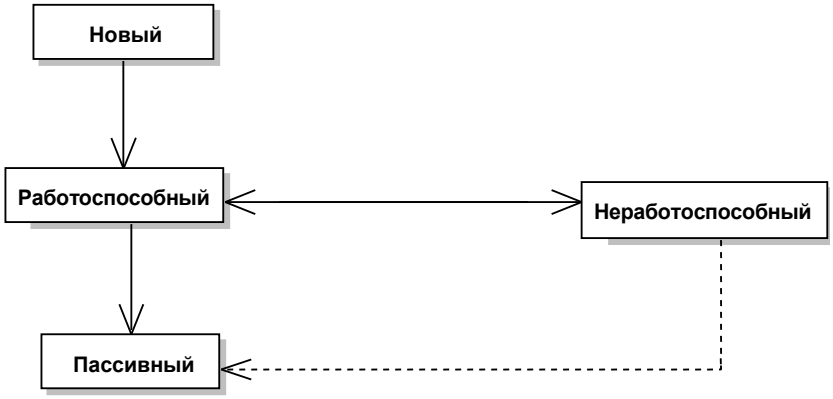

Рис. 1. Состояния потока

Поток может находиться в одном из состояний, соответствующих элементам статически вложенного перечисления **Thread.State**:

NEW – поток создан, но еще не запущен;

RUNNABLE – поток выполняется;

BLOCKED – поток блокирован;

WAITING – поток ждет окончания работы другого потока;

TIMED_WAITING – поток некоторое время ждет окончания другого потока;

TERMINATED — поток завершен.

Получить значение состояния потока можно вызовом метода **getState()**.

Поток переходит в состояние “неработоспособный” (**WAITING**) вызовом методов **wait()**, **suspend()** (deprecated-метод) или методов ввода/вывода, которые предполагают задержку. Для задержки потока на некоторое время (в миллисекундах) можно перевести его в режим ожидания (**TIMED_WAITING**) с помощью методов **sleep(long millis)** и **wait(long timeout)**, при выполнении которого может генерироваться прерывание **InterruptedException**. Вернуть потоку работоспособность после вызова метода **suspend()** можно методом **resume()** (deprecated-метод), а после вызова метода **wait()** – методами **notify()** или **notifyAll()**. Поток переходит в “пассивное” состояние (**TERMINATED**), если вызваны методы **interrupt()**, **stop()** (deprecated-метод) или метод **run()** завершил выполнение. После этого, чтобы запустить поток еще раз, необходимо создать новый объект потока. Метод **interrupt()** успешно завершает поток, если он находится в состоянии “работоспособность”. Если же поток неработоспособен, то метод генерирует исключительные ситуации разного типа в зависимости от способа остановки потока.

Интерфейс **Runnable** не имеет метода **start()**, а только единственный метод **run()**. Поэтому для запуска такого потока, как **Walk**, следует создать объект класса **Thread** и передать объект **Walk** его конструктору. Однако при прямом вызове метода **run()** поток не запустится, выполнится только тело самого метода.

Методы **suspend()**, **resume()** и **stop()** являются deprecated-методами и запрещены к использованию, так как они не являются в полной мере “потокобезопасными”.

Управление приоритетами и группы потоков

Потоку можно назначить приоритет от **1** (константа **MIN_PRIORITY**) до **10** (**MAX_PRIORITY**) с помощью метода **setPriority(int prior)**. Получить значение приоритета можно с помощью метода **getPriority()**.

// пример # 3 : демонстрация приоритетов: PriorityRunner.java: PriorThread.java

```
package chapt14;

public class PriorThread extends Thread {
     public PriorThread(String name){
          super(name);
     }
     public void run(){
          for (int i = 0; i < 71; i++){
               System.out.println(getName() + " " + i);
               try {
                    sleep(1);//попробовать sleep(0);
               } catch (InterruptedException e) {
                    System.err.print("Error" + e);
               }
          }
     }
}
```

```
package chapt14;

public class PriorityRunner {
     public static void main(String[] args) {
          PriorThread min = new PriorThread("Min");//1
          PriorThread max = new PriorThread("Max");//10
          PriorThread norm = new PriorThread("Norm");//5
          min.setPriority(Thread.MIN_PRIORITY);
          max.setPriority(Thread.MAX_PRIORITY);
          norm.setPriority(Thread.NORM_PRIORITY);
          min.start();
          norm.start();
          max.start();
     }
}
```

Поток с более высоким приоритетом в данном случае, как правило, монополизирует вывод на консоль.

Потоки объединяются в группы потоков. После создания потока нельзя изменить его принадлежность к группе.

```
ThreadGroup tg = new ThreadGroup("Группа потоков 1");
Thread t0 = new Thread(tg, "поток 0");
```

Все потоки, объединенные группой, имеют одинаковый приоритет. Чтобы определить, к какой группе относится поток, следует вызвать метод **getThreadGroup()**. Если поток до включения в группу имел приоритет выше приоритета группы потоков, то после включения значение его приритета станет равным приоритету группы. Поток же со значением приоритета более низким, чем приоритет группы после включения в оную, значения своего приоритета не изменит.

Управление потоками

Приостановить (задержать) выполнение потока можно с помощью метода **sleep(**время задержки**)** класса **Thread**. Менее надежный альтернативный способ состоит в вызове метода **yield()**, который может сделать некоторую паузу и позволяет другим потокам начать выполнение своей задачи. Метод **join()** блокирует работу потока, в котором он вызван, до тех пор, пока не будет закончено выполнение вызывающего метод потока.

// пример # 4 : задержка потока: JoinRunner.java

```
package chapt14;

class Th extends Thread {
     public Th(String str) {
          super();
          setName(str);
     }
     public void run() {
          String nameT = getName();
          System.out.println("Старт потока " + nameT);
          if ("First".equals(nameT)) {
               try {
                    sleep(5000);
               } catch (InterruptedException e) {
                    e.printStackTrace();
               }
               System.out.println("завершение потока "
                                                  + nameT);
          } else if ("Second".equals(nameT)) {
               try {
                    sleep(1000);
               } catch (InterruptedException e) {
                    e.printStackTrace();
               }
              System.out.println("завершение потока "
                                                  + nameT);
          }
     }
}
public class JoinRunner {
     public static void main(String[] args) {
          Th tr1 = new Th("First");
          Th tr2 = new Th("Second");
          tr1.start();
```

```
          tr2.start();
          try {
             tr1.join();
             System.out.println("завершение main");
          } catch (InterruptedException e){
                  e.printStackTrace();
          }
          /* join() не дает работать потоку main до окончания выполнения потока tr1 */
     }
}
```

Возможно, будет выведено:

```
Старт потока First
Старт потока Second
завершение потока Second
завершение потока First
завершение main
```

Несмотря на вызов метода **join()** для потока **tr1**, поток **tr2** будет работать, в отличие от потока **main**, который сможет продолжить свое выполнение только по завершении потока **tr1.**

Вызов метода **yield()** для исполняемого потока должен приводить к приостановке потока на некоторый квант времени, для того чтобы другие потоки могли выполнять свои действия. Однако если требуется надежная остановка потока, то следует использовать его крайне осторожно или вообще применить другой способ.

// пример # 5 : задержка потока: YieldRunner.java

```
package chapt14;

public class YieldRunner {
     public static void main(String[] args) {
          new Thread() {
             public void run() {
                   System.out.println("старт потока 1");
                   Thread.yield();
                   System.out.println("завершение 1");
               }
          }.start();
          new Thread() {
             public void run() {
                   System.out.println("старт потока 2");
                   System.out.println("завершение 2");
               }
          }.start();
     }
}
```

В результате может быть выведено:

```
старт потока 1
старт потока 2
завершение 2
завершение 1
```

Активизация метода **yield()** в коде метода **run()** первого объекта потока приведет к тому, что, скорее всего, первый поток будет остановлен на некоторый квант времени, что даст возможность другому потоку запуститься и выполнить свой код.

Потоки-демоны

Потоки-демоны работают в фоновом режиме вместе с программой, но не являются неотъемлемой частью программы. Если какой-либо процесс может выполняться на фоне работы основных потоков выполнения и его деятельность заключается в обслуживании основных потоков приложения, то такой процесс может быть запущен как поток-демон. С помощью метода **setDaemon(boolean value)**, вызванного вновь созданным потоком до его запуска, можно определить поток-демон. Метод **boolean isDaemon()** позволяет определить, является ли указанный поток демоном или нет.

/ пример # 6 : запуск и выполнение потока-демона: DemoDaemonThread.java */*

```
package chapt14;

class T extends Thread {
     public void run() {
          try {
               if (isDaemon()){
                System.out.println("старт потока-демона");
```

```
                sleep(10000); // заменить параметр на 1
               } else {
               System.out.println("старт обычного потока");
               }
          } catch (InterruptedException e) {
               System.err.print("Error" + e);
          } finally {
               if (!isDaemon())
                    System.out.println(
                      "завершение обычного потока");
               else
                    System.out.println(
                      "завершение потока-демона");
          }
     }
}
package chapt14;

public class DemoDaemonThread {
     public static void main(String[] args) {
          T usual = new T();
          T daemon = new T();
          daemon.setDaemon(true);
          daemon.start();
          usual.start();
          System.out.println(
                    "последний оператор main");
     }
}
```

В результате компиляции и запуска, возможно, будет выведено:

```
последний оператор main
старт потока-демона
старт обычного потока
завершение обычного потока
```

Поток-демон (из-за вызова метода **sleep(10000)**) не успел завершить выполнение своего кода до завершения основного потока приложения, связанного с методом **main()**. Базовое свойство потоков-демонов заключается в возможности основного потока приложения завершить выполнение потока-демона (в отличие от обычных потоков) с окончанием кода метода **main()**, не обращая внимания на то, что поток-демон еще работает. Если уменьшать время задержки потока-демона, то он может успеть завершить свое выполнение до окончания работы основного потока.

Потоки в графических приложениях

Добавить анимацию в апплет можно при использовании потоков. Поток, ассоциированный с апплетом, следует запускать тогда, когда апплет становится видимым, и останавливать при сворачивании браузера. В этом случае метод **repaint()** обновляет экран, в то время как программа выполняется. Поток создает анимационный эффект повторением вызова метода **paint()** и отображением графики в новой позиции.

```
/* пример # 7 : освобождение ресурсов апплетом: GraphicThreadsDemo.java */
package chapt14;
import java.awt.Color;
import java.awt.Container;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class GraphicThreadsDemo extends JFrame {
      JPanel panel = new JPanel();
      Graphics g;
      JButton btn = new JButton("Добавить шарик");
      int i;

      public GraphicThreadsDemo() {
            setBounds(100, 200, 270, 350);
            Container contentPane = getContentPane();
            contentPane.setLayout(null);
```

```
            btn.setBounds(50, 10, 160, 20);
            contentPane.add(btn);
            panel.setBounds(30, 40, 200, 200);
            panel.setBackground(Color.WHITE);
            contentPane.add(panel);
            btn.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ev) {
                        new BallThread(panel).start();
                        i++;
                        repaint();
                  }
            });
      }
      public static void main(String[] args) {
            GraphicThreadsDemo frame =
                        new GraphicThreadsDemo();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setVisible(true);
      }
      public void paint(Graphics g){
            super.paint(g);
         g.drawString("Количество шариков: " + i, 65, 300);
      }
}
class BallThread extends Thread {
      JPanel panel;
      private int posX, posY;
      private final int BALL_SIZE = 10;
      private double alpha;
      private int SPEED = 4;

      BallThread(JPanel p) {
            this.panel = p;
            //задание начальной позиции и направления шарика
            posX = (int)((panel.getWidth() - BALL_SIZE)
                                    * Math.random());
            posY = (int)((panel.getHeight() - BALL_SIZE)
                                    * Math.random());
            alpha = Math.random() * 10;
      }
      public void run() {
            while(true) {
                  posX += (int)(SPEED * Math.cos(alpha));
                  posY += (int)(SPEED * Math.sin(alpha));
            //вычисление угла отражения
            if( posX >= panel.getWidth() - BALL_SIZE )
                  alpha = alpha + Math.PI - 2 * alpha;
            else if( posX <= 0 )
                        alpha = Math.PI - alpha;
            if( posY >= panel.getHeight() - BALL_SIZE )
                        alpha = -alpha;
            else if( posY <= 0 )
                        alpha = -alpha;
                  paint(panel.getGraphics());
            }
      }
      public void paint(Graphics g) {
      //прорисовка шарика
      g.setColor(Color.BLACK);
      g.fillArc(posX, posY, BALL_SIZE, BALL_SIZE, 0, 360);
      g.setColor(Color.WHITE);
      g.drawArc(posX + 1, posY + 1, BALL_SIZE,
                              BALL_SIZE, 120, 30);
            try {
                  sleep(30);
            } catch(InterruptedException e) {
                  e.printStackTrace();
```

```
            }
        //удаление шарика
        g.setColor(panel.getBackground());
        g.fillArc(posX, posY, BALL_SIZE, BALL_SIZE, 0, 360);
        }
}
```

Рис.2.Потоки в апплетах

При вызове метода **stop()** апплета поток перестает существовать, так как ссылка на него устанавливается в **null** и освобождает ресурсы. Для следующего запуска потока необходимо вновь инициализировать ссылку и вызвать метод **start()** потока.

Методы synchronized

Очень часто возникает ситуация, когда несколько потоков, обращающихся к некоторому общему ресурсу, начинают мешать друг другу; более того, они могут повредить этот общий ресурс. Например, когда два потока записывают информацию в файл/объект/поток. Для предотвращения такой ситуации может использоваться ключевое слово **synchronized**. Синхронизации не требуют только атомарные процессы по записи/чтению, не превышающие по объему 32 бит.

В качестве примера будет рассмотрен процесс записи информации в файл двумя конкурирующими потоками. В методе **main()** класса **SynchroThreads** создаются два потока. В этом же методе создается экземпляр класса **Synchro**, содержащий поле типа **FileWriter**, связанное с файлом на диске. Экземпляр **Synchro** передается в качестве параметра обоим потокам. Первый поток записывает строку методом **writing()** в экземпляр класса **Synchro**. Второй поток также пытается сделать запись строки в тот же самый объект **Synchro**. Для избежания одновременной записи такие методы объявляются как **synchronized**. Синхронизированный метод изолирует объект, после чего объект становится недоступным для других потоков. Изоляция снимается, когда поток полностью выполнит соответствующий метод. Другой способ снятия изоляции – вызов метода **wait()** из изолированного метода.

В примере продемонстрирован вариант синхронизации файла для защиты от одновременной записи информации в файл двумя различными потоками.

```
/* пример # 8 : синхронизация записи информации в файл : MyThread.java : Synchro.java : SynchroThreads.java */
package chapt14;
import java.io.*;

public class Synchro {
      private FileWriter fileWriter;

      public Synchro(String file) throws IOException {
            fileWriter = new FileWriter(file, true);
            }
      public void close() {
            try {
                  fileWriter.close();
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }
      public synchronized void writing(String str, int i) {
                  try {
```

```
                        System.out.print(str + i);
                        fileWriter.append(str + i);
                Thread.sleep((long)(Math.random() * 50));
                        System.out.print("->" + i + " ");
                        fileWriter.append("->" + i + " ");
                    } catch (IOException e) {
                        System.err.print("ошибка файла");
                        e.printStackTrace();
                    } catch (InterruptedException e) {
                        System.err.print("ошибка потока");
                        e.printStackTrace();
                    }
        }
}
package chapt14;

public class MyThread extends Thread {
        private Synchro s;

        public MyThread(String str, Synchro s) {
                super(str);
                this.s = s;
        }
        public void run() {
                for (int i = 0; i < 5; i++) {
                        s.writing(getName(), i);
                }
        }
}
package chapt14;
import java.io.*;

public class SynchroThreads {
        public static void main(String[] args) {
                try {
                Synchro s = new Synchro("c:\\temp\\data.txt");

                MyThread t1 = new MyThread("First", s);
                MyThread t2 = new MyThread("Second", s);
                t1.start();
                t2.start();
                t1.join();
                t2.join();
                        s.close();
                } catch (IOException e) {
                        System.err.print("ошибка файла");
                        e.printStackTrace();
                } catch (InterruptedException e) {
                        System.err.print("ошибка потока");
                        e.printStackTrace();
                }
        }
}
```

В результате в файл будет выведено:

First0->0 Second0->0 First1->1 Second1->1 First2->2
Second2->2 First3->3 Second3->3 First4->4 Second4->4

Код построен таким образом, что при отключении синхронизации метода **writing()** при его вызове одним потоком другой поток может вклиниться и произвести запись своей информации, несмотря на то, что метод не завершил запись, инициированную первым потоком.

Вывод в этом случае может быть, например, следующим:

First0Second0->0 Second1->0 First1->1 First2->1 Second2->2 First3->3 First4->2 Second3->3 Second4->4 ->4

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

```
            Thread.sleep(50);
        } catch (InterruptedException e) {
            System.err.print("ошибка потока");
        }
    }
    public static void main(String [] args){
        try{
            Thread thread = new ThreadTimedWaitingStateTest();
            // NEW – поток создан, но ещё не запущен
            System.out.println("1: " + thread.getState());
            thread.start();
            // RUNNABLE – поток запущен
            System.out.println("2: " + thread.getState());
            Thread.sleep(10);
            // TIMED_WAITING
            // поток ждет некоторое время окончания работы другого потока
            System.out.println("3: " + thread.getState());
            thread.join();
            // TERMINATED –  поток завершил выполнение
            System.out.println("4: " + thread.getState());
        } catch (InterruptedException e) {
            System.err.print("ошибка потока");
        }
    }
}
```

В результате компиляции и запуска будет выведено:

```
1: NEW
2: RUNNABLE
3: TIMED_WAITING
4: TERMINATED
```

```
/* пример # 12 : состояния BLOCKED, WAITING : ThreadWaitingStateTest.java */
package chapt14;
public class ThreadWaitingStateTest extends Thread {

    public void run() {
        try {
            synchronized (this) {
                wait();
            }
        } catch (InterruptedException e) {
            System.err.print("ошибка потока");
        }
    }

    public static void main(String[] args) {
        try {
            Thread thread = new ThreadWaitingStateTest();
            thread.start();
            synchronized (thread) {
                Thread.sleep(10);
                // BLOCKED – because thread attempting to acquire a lock
                System.out.println("1: " + thread.getState());
            }
            Thread.sleep(10);
            // WAITING – метод wait() внутри synchronized
            // останавил поток и освободил блокировку
            System.out.println("2: " + thread.getState());
            thread.interrupt();
        } catch (InterruptedException e) {
            System.err.print("ошибка потока");
        }
    }
}
```

В результате компиляции и запуска будет выведено:

```
1: BLOCKED
2: WAITING
```

Потоки в J2SE 5

Java всегда предлагала широкие возможности для мультипрограммирования: потоки – это основа языка. Однако очень часто использование таких возможностей становилось ловушкой: правильно написать и отладить многопоточную программу достаточно сложно.

В версии 1.5 языка добавлены пакеты классов **java.util.concurrent.locks**, **java.util.concurrent.atomic**, **java.util.concurrent**, возможности которых обеспечивают более высокую производительность, масштабируемость, построение потокобезопасных блоков параллельных (concurrent) классов, вызов утилит синхронизации, использование семафоров, ключей и atomic-переменных.

Возможности синхронизации существовали и ранее. Практически это означало, что синхронизированные объекты блокировались, хотя необходимо это было далеко не всегда. Например, поток, изменяющий одну часть объекта **Hashtable**, блокировал работу других потоков, которые хотели прочесть (даже не изменить) совсем другую часть этого объекта. Поэтому введение дополнительных возможностей, связанных с синхронизаций потоков и блокировкой ресурсов довольно логично.

Ограниченно потокобезопасные (thread safe) коллекции и вспомогательные классы управления потоками сосредоточены в пакете **java.util.concurrent**. Среди них можно отметить:

- параллельные классы очередей **ArrayBlockingQueue** (FIFO очередь с фиксированой длиной), **PriorityBlockingQueue** (очередь с приоритетом) и **ConcurrentLinkedQueue** (FIFO очередь с нефиксированой длиной);
- параллельные аналоги существующих синхронизированых классов-коллекций **ConcurrentHashMap** (аналог **Hashtable**) и **CopyOnWriteArrayList** (реализация **List**, оптимизированная для случая, когда количество итераций во много раз превосходит количество вставок и удалений);
- механизм управления заданиями, основанный на возможностях класса **Executor**, включающий пул потоков и службу их планирования;
- высокопроизводительный класс **Lock**, поддерживающий ограниченные ожидания снятия блокировки, прерываемые попытки блокировки, очереди блокировки и установку ожидания снятия нескольких блокировок посредством класса **Condition**;
- классы атомарных переменных (**AtomicInteger**, **AtomicLong**, **AtomicReference**), а также их высокопроизводительные аналоги **SyncronizedInt** и др.;
- классы синхронизации общего назначения, такие как **Semaphore**, **CountDownLatch** (позволяет потоку ожидать завершения нескольких операций в других потоках), **CyclicBarrier** (позволяет нескольким потокам ожидать момента, когда они все достигнут какой-либо точки) и **Exchanger** (позволяет потокам синхронизироваться и обмениваться информацией);
- обработка неотловленных прерываний: класс **Thread** теперь поддерживает установку обработчика на неотловленные прерывания (подобное ранее было доступно только в **ThreadGroup**).

Хорошим примером новых возможностей является синхронизация коллекции типа **Hashtable**. Объект **Hashtable** синхронизируется целиком, и если один поток занял кэш остальные потоки вынуждены ожидать освобождения объекта. В случае же использования нового класса **ConcurrentHashMap** практически все операции чтения могут проходить одновременно, операции чтения и записи практически не задерживают друг друга, и только одновременные попытки записи могут блокироваться. В случае же использования данного класса как кэша (спецификой которого является большое количество операций чтения и малое – записи), блокироваться многопоточный код практически не будет.

В таблице приведено время выполнения (в миллисекундах) программы, использовавшей в качестве кэша **ConcurrentHashMap** и **Hashtable**. Тесты проводились на двухпроцессорном сервере под управлением Linux. Количество данных для большего количества потоков увеличивалось.

Количество потоков	ConcurrentHashMap	Hashtable
1	1.00	1.03
2	2.59	32.40
4	5.58	78.23
8	13.21	163.48
16	27.58	341.21
32	57.27	778.41

```
// пример # 13 : применение семафора: Sort.java :  ArraySort.java
package chapt14;
import java.util.concurrent.*;

public class Sort {
     public static final int ITEMS_COUNT = 15;
     public static double items[];
     // семафор, контролирующий разрешение на доступ к элементам массива
     public static Semaphore sortSemaphore =
```

```
                                        new Semaphore(0, true);
    public static void main(String[] args) {
        items = new double[ITEMS_COUNT];
        for(int i = 0 ; i < items.length ; ++i)
            items[i] = Math.random();
        new Thread(new ArraySort(items)).start();
        for(int i = 0 ; i < items.length ; ++i) {
            /*
             * при проверке доступности элемента сортируемого
             * массива происходит блокировка главного потока
             * до освобождения семафора
             */
            sortSemaphore.acquireUninterruptibly();
            System.out.println(items[i]);
        }
    }
}
class ArraySort implements Runnable {
    private double items[];

    public ArraySort(double items[]) {
           this.items = items;
    }
    public void run(){
        for(int i = 0 ; i < items.length - 1 ; ++i) {
            for(int j = i + 1 ; j < items.length ; ++j) {
                if( items[i] < items[j] ) {
                    double tmp = items[i];
                    items[i] = items[j];
                    items[j] = tmp;
                }
            }
            // освобождение семафора
            Sort.sortSemaphore.release();
            try {
                Thread.sleep(71);
            } catch (InterruptedException e) {
                System.err.print(e);
            }
        }
        Sort.sortSemaphore.release();
    }
}
```