Java.SE.07

MULTITHREADING

Author: **Ihar Blinou**
Oracle Certified Java Instructor
ihar_blinou@epam.com

Содержание

1. Понятие многопоточности
2. Работа с потоками
3. Синхронизация
4. Concurrent

ПОНЯТИЕ МНОГОПОТОЧНОСТИ

Понятие многопоточности

Java обеспечивает встроенную поддержку для *многопоточного программирования.*

Многопоточная программа содержит две и более частей, которые могут выполняться одновременно, конкурируя друг с другом.

Каждая часть такой программы называется *потоком,* а каждый поток определяет отдельный путь выполнения (в последовательности операторов программы).

Многопоточность — это специализированная форма многозадачности.

Понятие многопоточности

Многозадачные потоки требуют меньших накладных расходов по сравнению с многозадачными процессами. Процессы — это тяжеловесные задачи, которым требуются отдельные адресные пространства. Связи между процессами ограничены и стоят не дешево. Переключение контекста от одного процесса к другому также весьма дорогостоящая задача.

С другой стороны, потоки достаточно легковесны. Они совместно используют одно и то же адресное пространство и кооперативно оперируют с одним и тем же тяжеловесным процессом, межпоточные связи недороги, а переключение контекста от одного потока к другому имеет низкую стоимость.

Понятие многопоточности

Многопоточность дает возможность писать очень эффективные программы, которые максимально используют CPU, потому что время его простоя можно свести к минимуму.

Это особенно важно для интерактивной сетевой среды, в которой работает Java, потому что время простоя является общим.

Скорость передачи данных по сети намного меньше, чем скорость, с которой компьютер может их обрабатывать.

В традиционной однопоточной среде ваша программа должна ждать окончания каждой своей задачи, прежде чем она сможет перейти к следующей (даже при том, что большую часть времени CPU простаивает).

Многопоточность позволяет получить доступ к этому времени простоя и лучше его использовать.

Понятие многопоточности

Исполнительная система Java во многом зависит от потоков, и все библиотеки классов разработаны с учетом многопоточности.

Java использует потоки для обеспечения асинхронности во всей среде.

Ценность многопоточной среды лучше понимается по контрасту с ее аналогом.

- Однопоточные системы используют подход, называемый *циклом событий с опросом* (event loop with polling). В этой модели, единственный поток управления выполняется в бесконечном цикле, опрашивая единственную очередь событий, чтобы решить, что делать дальше. Как только этот механизм опроса возвращает сигнал готовности сетевого файла готов для чтения, цикл событий передает управление соответствующему обработчику событий. До возврата из этого обработчика в системе ничего больше случиться не может.

Понятие многопоточности

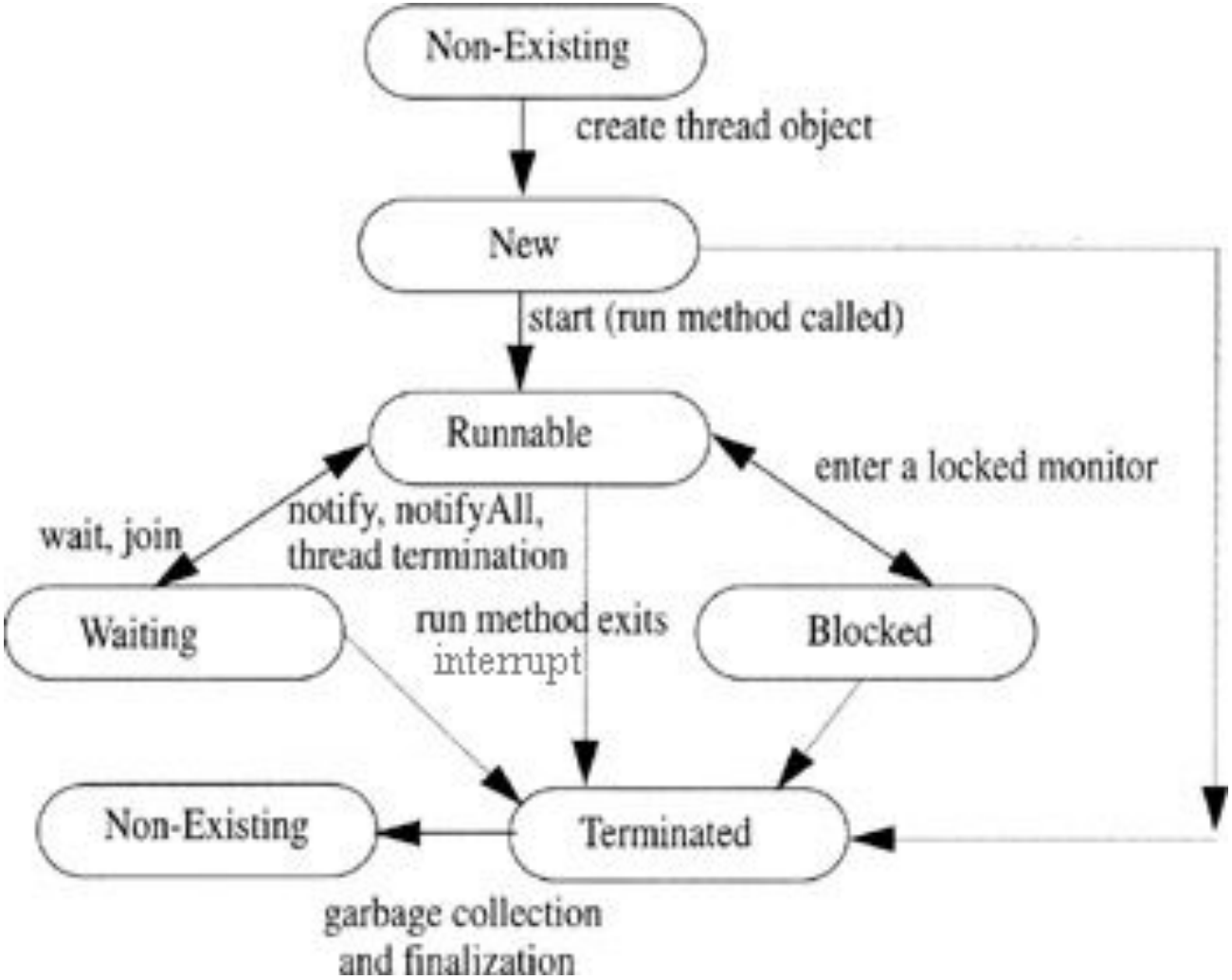
- Выгода от многопоточности Java заключается в том, что устраняется механизм "главный цикл/опрос". Один поток может делать паузу без остановки других частей программы. Например, время простоя, образующееся, когда поток читает данные из сети или ждет ввод пользователя, может использоваться в другом месте.

Понятие многопоточности

Потоки существуют в нескольких состояниях.

- Поток может быть в состоянии ***выполнения***.
- Может находиться в состоянии ***готовности к выполнению***, как только он получит время CPU.
- Выполняющийся поток может быть ***приостановлен***, что временно притормаживает его действие.
- Затем приостановленный поток может быть ***продолжен*** (возобновлен) с того места, где он был остановлен.
- Поток может быть ***блокирован*** в ожидании ресурса. В любой момент выполнение потока может быть завершено, что немедленно останавливает его выполнение.

Понятие многопоточности



РАБОТА С ПОТОКАМИ

Работа с потоками

Многопоточная система Java построена на классе **Thread**, его методах и связанном с ним интерфейсе **Runnable**.

Thread инкапсулирует поток выполнения. Так как вы не можете непосредственно обращаться к внутреннему состоянию потока выполнения, то будете иметь с ним дело через его полномочного представителя — экземпляр (объект) класса Thread, который его породил.

Чтобы создать новый поток, ваша программа должна будет или расширять класс **Thread** или реализовывать интерфейс **Runnable**.

Работа с потоками. Example 01

```
package _java._se._07.startthread;

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);
                  }
            }
      }
}
```

Работа с потоками. Example 01

```
package _java._se._07.startthread;

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);
                  }
            }
      }
}
```

Работа с потоками. Example 01

```
package _java._se._07.startthread;
public class DemoThead {
      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!
            }
}
```

Работа с потоками. Example 01

Результат:

```
Talking
Walking
Walking
Talking
Walking
Talking
Talking
Walking
Walking
Talking
Walking
Talking
Walking
Talking
Walking
Talking
```

Работа с потоками

Некоторые методы класса **Thread**

Метод	Значение
getName ()	Получить имя потока
getPriority ()	Получить приоритет потока
isAlive ()	Определить, выполняется ли еще поток
join ()	Ждать завершения потока
run ()	Указать точку входа в поток
sleep ()	Приостановить поток на определенный период времени
start ()	Запустить поток с помощью вызова его метода run ()

Работа с потоками

Когда Java-программа запускается, один поток начинает выполняться немедленно.

Он обычно называется *главным потоком*.

Главный поток важен по двум причинам:

- Это поток, из которого будут порождены все другие "дочерние" потоки.
- Это должен быть последний поток, в котором заканчивается выполнение. Когда главный поток останавливается, программа завершается.

Работа с потоками

Хотя главный поток создается автоматически после запуска программы, он может управляться через Thread-объект. Для организации управления нужно получить ссылку на него, вызывая метод Currentrhread (), который является public static членом класса Thread.

static Thread currentThread()

Этот метод возвращает ссылку на поток, в котором он вызывается. Как только вы получаете ссылку на главный поток, то можете управлять им точно так же, как любым другим потоком.

Работа с потоками. Example 02

```
package _java._se._07.startthread;
public class CurrentThreadDemo {
	public static void main(String args[]) {
		Thread t = Thread.currentThread();
		System.out.println("Текущий поток: " + t);
		// изменить имя потока
		t.setName("My Thread");
		System.out.println("После изменения имени: " + t);
		try {
			for (int n = 5; n > 0; n--) {
				System.out.println(n);
				Thread.sleep(1000);
			}
		} catch (InterruptedException e) {
			System.out.println("Главный поток завершен");
		}
	}
}
```

Работа с потоками. Example 02

Результат:

```
Текущий поток: Thread[main,5,main]
После изменения имени: Thread[My Thread,5,main]
5
4
3
2
1
```

Работа с потоками

Существуют два способа определения, закончился ли поток.

Один из них позволяет вызывать метод **isAlive()** на потоке. Этот метод определен в Thread и его общая форма выглядит так:

final boolean isAlive()

Метод **isAlive**() возвращает true, если поток, на котором он вызывается — все еще выполняется. В противном случае возвращается false.

Работа с потоками

В то время как **isAlive**() полезен только иногда, чаще для ожидания завершения потока вызывается метод **join**() следующего формата:

final void join() throws InterruptedException

Этот метод ждет завершения потока, на котором он вызван. Его имя происходит из концепции перевода потока в состояние ожидания, пока указанный поток не присоединит его.

Дополнительные формы **join**() позволяют определять максимальное время ожидания завершения указанного потока.

Работа с потоками

Состояние потока возвращается методами **int getState()** и **boolean isAlive()** класса **Thread**

getState()	isAlive()
NEW	
RUNNABLE	+
BLOCKED	+
WAITING	+
TIMED_WAITING	+
TERMINATED	

Работа с потоками. Example 03

```
package _java._se._07.startthread;
public class DemoJoin {
	public static void main(String args[]) {
		NewThread ob1 = new NewThread("One");
		NewThread ob2 = new NewThread("Two");
		NewThread ob3 = new NewThread("Three");
		System.out.println("Thread One is alive: " + ob1.t.isAlive());
		System.out.println("Thread Two is alive: " + ob2.t.isAlive());
		System.out.println("Thread Three is alive: " + ob3.t.isAlive());
		// wait for threads to finish
		try {
			System.out.println("Waiting for threads to finish.");
			ob1.t.join();
			ob2.t.join();
			ob3.t.join();
		} catch (InterruptedException e) {
			System.out.println("Main thread Interrupted");
		}
		System.out.println("Thread One is alive: " + ob1.t.isAlive());
		System.out.println("Thread Two is alive: " + ob2.t.isAlive());
		System.out.println("Thread Three is alive: " + ob3.t.isAlive());
		System.out.println("Main thread exiting.");
	}
}
```

Работа с потоками. Example 03

```
class NewThread implements Runnable {
      String name; // name of thread
      Thread t;
      NewThread(String threadname) {
            name = threadname;
            t = new Thread(this, name);
            System.out.println("New thread: " + t);
            t.start(); // Start the thread
      }
      // This is the entry point for thread.
      public void run() {
            try {
                  for (int i = 5; i > 0; i--) {
                        System.out.println(name + ": " + i);
                        Thread.sleep(1000);
                  }
            } catch (InterruptedException e) {
                  System.out.println(name + " interrupted.");
            }
            System.out.println(name + " exiting.");
      }
}
```

Работа с потоками. Example 03

Результат:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
One: 5
New thread: Thread[Three,5,main]
Two: 5
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
Three: 5
Two: 4
One: 4
Three: 4
One: 3
Three: 3
Two: 3
Three: 2
Two: 2
One: 2
Three: 1
One: 1
Two: 1
Three exiting.
One exiting.
Two exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.
```

Работа с потоками. Example 04

```
package _java._se._07.startthread;
public class GetStateDemo implements Runnable{
	public void run() {
		// Returns the state of this thread.
		Thread.State state = Thread.currentThread().getState();
		System.out.println(Thread.currentThread().getName() + " " + state);
	}
	public static void main(String args[]) {
		Thread th1 = new Thread(new GetStateDemo());
		th1.start();
		try {
			th1.sleep(1000);
		} catch (Exception e) {
			System.out.println(e);
		}
		// Returns the state of the thread.
		Thread.State state = th1.getState();
		System.out.println(th1.getName() + " " + state);
	}
}
```

Работа с потоками. Example 04

Результат:

```
Thread-0 RUNNABLE
Thread-0 TERMINATED
```

Работа с потоками

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.

```
class MyThread3 extends Thread {
      public void run() {
            for (int i = 0; i < 10; i++) {
                  System.out.println("child thread");
                  Thread.yield();
            }
      }
}
```

Приоритеты потоков

Планировщик потоков использует их приоритеты для принятия решений о том, когда нужно разрешать выполнение тому или иному потоку.

Теоретически высокоприоритетные потоки получают больше времени CPU, чем низкоприоритетные.

На практике, однако, количество времени CPU, которое поток получает, часто зависит от нескольких факторов помимо его приоритета. (Например, относительная доступность времени CPU может зависеть от того, как операционная система реализует многозадачный режим.)

Работа с потоками

Высокоприоритетный поток может также упреждать низкоприоритетный (т. е. перехватывать у него управление процессором).

Скажем, когда низкоприоритетный поток выполняется, а высокоприоритетный поток возобновляется (от ожидания на вводе/выводе, к примеру), высокоприоритетный поток будет упреждать низкоприоритетный.

Теоретически, потоки равного приоритета должны получить равный доступ к CPU.

Работа с потоками

Для безопасности потоки, которые совместно используют один и тот же приоритет, должны время от времени уступать друг другу управление.

Это гарантирует, что все потоки имеют шанс выполниться под неприоритетной операционной системой.

Практически, даже в неприоритетных средах, большинство потоков все еще получают шанс выполняться, потому что большинство из них неизбежно сталкивается с некоторыми блокирующими ситуациями, типа ожидания ввода/вывода.

Когда это случается, блокированный поток приостанавливается, а другие могут продолжаться.

Работа с потоками

Для установки приоритета потока используйте метод **setPriority()**, который является членом класса Thread. Вот его общая форма:

final void setPriority(int *level)*

где ***level*** определяет новую установку приоритета для вызывающего потока.

- Значение параметра ***level*** должно быть в пределах диапазона **MIN_PRIORITY** и **MAX_PRIORITY**. В настоящее время эти значения равны 1 и 10, соответственно.
- Чтобы вернуть потоку приоритет, заданный по умолчанию, определите **NORM_PRIORITY**, который в настоящее время равен 5.
- Эти приоритеты определены в Thread как final-переменные.

Работа с потоками

Вы можете получить текущую установку приоритета, вызывая метод **getPriority()** класса **Thread**, чей формат имеет следующий вид:

final int getPriority()

Реализации Java могут иметь радикально различное поведение, когда они переходят к планированию.

Работа с потоками. Example 05

```
package _java._se._07.startthread;
public class PriorityDemo {
      public static void main(String args[]) {
            Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
            Clicker hi = new Clicker(Thread.NORM_PRIORITY + 2);
            Clicker lo = new Clicker(Thread.NORM_PRIORITY - 2);
            lo.start();
            hi.start();
            try {
                  Thread.sleep(100);
            } catch (InterruptedException e) {
                  System.out.println("Main thread interrupted.");
            }
            lo.stop();
            hi.stop();
            // Wait for child threads to terminate.
            try {
                  hi.t.join();
                  lo.t.join();
            } catch (InterruptedException e) {
                  System.out.println("InterruptedException caught");
            }
            System.out.println("Low-priority thread: " + lo.click);
            System.out.println("High-priority thread: " + hi.click);
      }
}
```

Работа с потоками. Example 05

```
class Clicker implements Runnable {
      int click = 0;
      Thread t;
      private volatile boolean running = true;
      public Clicker(int p) {
            t = new Thread(this);
            t.setPriority(p);
      }
      public void run() {
            while (running) {
                  click++;
            }
      }
      public void stop() {
            running = false;
      }
      public void start() {
            t.start();
      }
}
```

Работа с потоками. Example 05

Результат:

```
Low-priority thread: 35443425
High-priority thread: 36428949
```

Группы потоков

Для того, чтобы отдельный поток не мог начать останавливать и прерывать все потоки подряд, введено понятие группы.

Поток может оказывать влияние только на потоки, которые находятся в одной с ним группе.

Группу потоков представляет класс **ThreadGroup**.

Такая организация позволяет защитить потоки от нежелательного внешнего воздействия.

Работа с потоками

Группа потоков может содержать другие группы, что позволяет организовать все потоки и группы в иерархическое дерево, в котором каждый объект **ThreadGroup**, за исключением корневого, имеет родителя.

Класс **ThreadGroup** обладает методами для изменения свойств всех входящих в него потоков, таких, как приоритет, daemon и т. д.

Метод **list()** позволяет получить список потоков.

Работа с потоками

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.

Работа с потоками. Example 06

```
package _java._se._07.startthread;
class MyThread extends Thread {
	boolean suspended;
	MyThread(String threadname, ThreadGroup tgOb) {
		super(tgOb, threadname);
		System.out.println("New thread: " + this);
		suspended = false;
		start(); // Start the thread
	}
	public void run() {
		try {
			for (int i = 5; i > 0; i--) {
				System.out.println(getName() + ": " + i);
				Thread.sleep(1000);
				synchronized (this) {
					while (suspended) {
						wait();
					}
				}
			}
		} catch (Exception e) {
			System.out.println("Exception in " + getName());
		}
		System.out.println(getName() + " exiting.");
	}
```

Работа с потоками. Example 06

```
        void suspendMe() {
                suspended = true;
        }
        synchronized void resumeMe() {
                suspended = false;
                notify();
        }
}
```

Работа с потоками. Example 06

```
public class ThreadCroupDemo {

      public static void main(String[] args){

          ThreadGroup groupA = new ThreadGroup("Group A");
          ThreadGroup groupB = new ThreadGroup("Group B");
          MyThread ob1 = new MyThread("One", groupA);
          MyThread ob2 = new MyThread("Two", groupA);
          MyThread ob3 = new MyThread("Three", groupB);
          MyThread ob4 = new MyThread("Four", groupB);
          System.out.println("\nHere is output from list():");
          groupA.list();
          groupB.list();
          System.out.println("Suspending Group A");
          Thread tga[] = new Thread[groupA.activeCount()];
          groupA.enumerate(tga); // get threads in group

          for (int i = 0; i < tga.length; i++) {
            ((MyThread) tga[i]).suspendMe(); // suspend each thread
          }
```

Работа с потоками. Example 06

```
        try {
          System.out.println("Waiting for threads to finish.");
          ob1.join();
          ob2.join();
          ob3.join();
          ob4.join();
        } catch (Exception e) {
          System.out.println("Exception in Main thread");
        }
        System.out.println("Main thread exiting.");
      }
}
```

Работа с потоками. Example 06

Результат:

```
New thread: Thread[One,5,Group A]
New thread: Thread[Two,5,Group A]
One: 5
New thread: Thread[Three,5,Group B]
Two: 5
New thread: Thread[Four,5,Group B]
Three: 5
Here is output from list():
java.lang.ThreadGroup[name=Group A,maxpri=10]
    Thread[One,5,Group A]
    Thread[Two,5,Group A]
Four: 5
java.lang.ThreadGroup[name=Group B,maxpri=10]
    Thread[Three,5,Group B]
    Thread[Four,5,Group B]
Suspending Group A
Three: 4
Four: 4
Resuming Group A
Two: 4
```

```
Waiting for threads to finish.
One: 4
Three: 3
One: 3
Two: 3
Four: 3
Three: 2
Two: 2
One: 2
Four: 2
Three: 1
One: 1
Two: 1
Four: 1
Three exiting.
Two exiting.
One exiting.
Four exiting.
Main thread exiting.
```

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.

Работа с потоками. Example 07

```
package _java._se._07.startthread;
class T extends Thread {
public void run() {
      try {
            if (isDaemon()) {
                  System.out.println("старт потока-демона");
                  sleep(1000); // заменить параметр на 1
            } else {
                  System.out.println("старт обычного потока");
                  sleep(100);
            }
      } catch (InterruptedException e) {
            System.err.print("Error" + e);
      } finally {
            if (!isDaemon()){
                  System.out.println("завершение обычного потока");
            }
            else
                  System.out.println("завершение потока-демона");
      }
}
}
```

Работа с потоками. Example 07

```
public class DaemonDemo {
      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");
      }
}
```

Работа с потоками. Example 07

Результат:

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

СИНХРОНИЗАЦИЯ

Синхронизация

Правила, которые определяют, когда переключение контекста имеет место.

- Поток может *добровольно отказаться от управления.* Это делается явно, переходом в режим ожидания или блокированием на ожидающем вводе/выводе. В этом сценарии просматриваются все потоки, и CPU передается самому высокоприоритетному потоку, который готов к выполнению.
- Поток может быть *приостановлен более приоритетным потоком.* В этом случае занимающий процессор низкоприоритетный поток временно останавливается (независимо от того, что он делает) потоком с более высоким приоритетом. Данный механизм называется *упреждающей многозадачностью* (preemptive multitasking).

Синхронизация

Поскольку многопоточность обеспечивает *асинхронное* поведение ваших программ, должен существовать способ добиться *синхронности,* когда в этом возникает необходимость.

Например, если вы хотите, чтобы два потока взаимодействовали и совместно использовали сложную структуру данных типа связного списка, нужно каким-то образом гарантировать отсутствие между ними конфликтов.

Вы должны удержать один поток от записи данных, пока другой поток находится в процессе их чтения.

Синхронизация

Для этой цели Java эксплуатирует модель синхронизации процессов — **монитор**.

Монитор — это механизм управления связью между процессами. Вы можете представлять монитор, как очень маленький блок, который содержит только один поток. Как только поток входит в монитор, все другие потоки должны ждать, пока данный не выйдет из монитора. Таким образом, монитор можно использовать для защиты совместно используемого (разделяемого) ресурса от управления несколькими потоками одновременно.

Синхронизация

Большинство многопоточных систем создает мониторы как объекты, которые ваша программа должна явно получить и использовать.

В Java-системе нет класса с именем Monitor. Вместо этого, каждый объект имеет свой собственный неявный монитор, который вводится автоматически при вызове одного из методов объекта.

Как только поток оказывается внутри синхронизированного метода, никакой другой поток не может вызывать иной синхронизированный метод того же объекта.

Синхронизация

После того как вы разделите свою программу на отдельные потоки, нужно определить, как они будут взаимодействовать друг с другом.

Java обеспечивает ясный, дешевый путь для взаимного общения двух (или нескольких) потоков через вызовы предопределенных методов, которыми обладают все объекты.

Система передачи сообщений Java позволяет потоку войти в *синхронизированный* метод на объекте и затем ждать там, пока некоторый другой поток явно не уведомит его о выходе.

Синхронизация

Когда несколько потоков нуждаются в доступе к разделяемому ресурсу, им необходим некоторый способ гарантии того, что ресурс будет использоваться одновременно только одним потоком.

Процесс, с помощью которого это достигается, называется ***синхронизацией***.

Ключом к синхронизации является концепция монитора (также называемая ***семафором***).

Монитор — это объект, который используется для взаимоисключающей блокировки (mutually exclusive lock), или ***mutex***.

Синхронизация

Только один поток может иметь *собственный* монитор в заданный момент.

Когда поток получает блокировку, говорят, что он *вошел* в монитор. Все другие потоки пытающиеся вводить блокированный монитор, будут приостановлены, пока первый не вышел из монитора.

Говорят, что другие потоки *ожидают* монитор.

При желании поток, владеющий монитором, может повторно вводить тот же самый монитор.

Синхронизировать код можно двумя способами. Оба используют ключевое слово **synchronized**.

Синхронизация

Синхронизация в Java проста потому, что каждый объект имеет свой собственный неявный связанный с ним монитор.

Чтобы ввести монитор объекта, просто вызывают метод, который был модифицирован ключевым словом **synchronized**.

Пока поток находится внутри синхронизированного метода, все другие потоки, пытающиеся вызвать его (или любой другой синхронизированный метод) на том же самом экземпляре, должны ждать.

Чтобы выйти из монитора и оставить управление объектом следующему ожидающему потоку, владелец монитора просто возвращается из синхронизированного метода.

Синхронизация. Example 08

```
package _java._se._07.synchro;
public class SynchroMethodDemo {
       private String objID;
        public SynchroMethodDemo(String objID) {
          this.objID = objID;
        }
        public synchronized void doStuff(int val) {
          print("entering doStuff()");
          int num = val * 2 + objID.length();
          print("local variable num=" + num);
          try {
            Thread.sleep(2000);
          } catch (InterruptedException x) {
          }
          print("leaving doStuff()");
        }
        public void print(String msg) {
          threadPrint("objID=" + objID + " - " + msg);
        }
        public static void threadPrint(String msg) {
          String threadName = Thread.currentThread().getName();
          System.out.println(threadName + ": " + msg);
        }
```

Синхронизация. Example 08

```
        public static void main(String[] args) {
          final SynchroMethodDemo ooim = new SynchroMethodDemo("obj1");
          Runnable runA = new Runnable() {
            public void run() {
              ooim.doStuff(3);
            }
          };
          Thread threadA = new Thread(runA, "threadA");
          threadA.start();
          try {
            Thread.sleep(200);
          } catch (InterruptedException x) {
          }
          Runnable runB = new Runnable() {
            public void run() {
              ooim.doStuff(7);
            }
          };
          Thread threadB = new Thread(runB, "threadB");
          threadB.start();
        }
}
```

Синхронизация. Example 08

Результат:

```
threadA: objID=obj1 - entering doStuff()
threadA: objID=obj1 - local variable num=10
threadA: objID=obj1 - leaving doStuff()
threadB: objID=obj1 - entering doStuff()
threadB: objID=obj1 - local variable num=18
threadB: objID=obj1 - leaving doStuff()
```

Синхронизация

Хотя определения синхронизированных методов внутри классов — это простые и эффективные средства достижения синхронизации, они не будут работать во всех случаях.

Например, вы хотите синхронизировать доступ к объектам класса, который не был разработан для многопоточного доступа. То есть класс не использует синхронизированные методы.

Кроме того, этот класс был создан не вами, а третьим лицом, и вы не имеете доступа к исходному коду.

Таким образом, вы не можете добавлять спецификатор synchronized к соответствующим методам в классе.

Решение данной проблемы весьма просто. Нужно поместить вызовы методов, определенных этим классом внутрь синхронизированного блока.

Синхронизация

Вот общая форма оператора synchronized:

```
synchronized(object) {
// операторы для синхронизации
}
```

где *object* — ссылка на объект, который нужно синхронизировать.

Если нужно синхронизировать одиночный оператор, то фигурные скобки можно опустить.

Блок гарантирует, что вызов метода, который является членом объекта *object,* происходит только после того, как текущий поток успешно ввел монитор объекта.

Синхронизация. Example 09

```
package _java._se._07.synchro;
public class SynchroBlockDemo {
	public static synchronized void staticA() {
		System.out.println("entering staticA()");
		try {
			Thread.sleep(5000);
		} catch (InterruptedException x) {
		}
		System.out.println("leaving staticA()");
	}
	public static void staticB() {
		synchronized (SynchroBlockDemo.class) {
			System.out.println("in staticB() : inside sync block");
			try {
				Thread.sleep(2000);
			} catch (InterruptedException x) {
			}
		}
	}
```

Синхронизация. Example 09

```
	public static void main(String[] args) {
		Runnable runA = new Runnable() {
			public void run() {
				SynchroBlockDemo.staticA();
			}
		};
		Thread threadA = new Thread(runA, "A");
		threadA.start();
		try {
			Thread.sleep(200);
		} catch (InterruptedException x) {
		}
		Runnable runB = new Runnable() {
			public void run() {
				SynchroBlockDemo.staticB();
			}
		};
		Thread threadB = new Thread(runB, "B");
		threadB.start();
	}
}
```

Синхронизация. Example 09

Результат:

```
entering staticA()
leaving staticA()
in staticB() : inside sync block
```

Синхронизация

Вы можете достичь более тонкого уровня управления через *связь между процессами.*

Многопоточность заменяет программирование цикла событий, делением задач на дискретные и логические модули.

Потоки также обеспечивают и второе преимущество — они отменяют опрос. Опрос обычно реализуется циклом, который используется для повторяющейся проверки некоторого условия.

Как только условие становится истинным, предпринимается соответствующее действие. На этом теряется время CPU.

Синхронизация

Например, рассмотрим классическую проблему организации очереди, где один поток производит некоторые данные, а другой — их потребляет.

Предположим, что, прежде чем генерировать большее количество данных, производитель должен ждать, пока потребитель не закончит свою работу.

В системе же опроса, потребитель тратил бы впустую много циклов CPU на ожидание конца работы производителя. Как только производитель закончил свою работу, он вынужден начать опрос, затрачивая много циклов CPU на ожидание конца работы потребителя. Ясно, что такая ситуация нежелательна.

Чтобы устранить опросы, Java содержит изящный механизм межпроцессовой связи через методы **wait(), notify() и notifyAll()**. Они реализованы как final-методы в классе object, поэтому доступны всем классам

Синхронизация

- **wait ()** сообщает вызывающему потоку, что нужно уступить монитор и переходить в режим ожидания ("спячки"), пока некоторый другой поток не введет тот же монитор и не вызовет notify ();
- **notify ()** "пробуждает" первый поток (который вызвал wait ()) на том же самом объекте;
- **notifyAll()** пробуждает все потоки, которые вызывали wait () на том же самом объекте. Первым будет выполняться самый высокоприоритетный поток.

Эти методы объявляются в классе Object в следующей форме:

- **final void wait() throws InterruptedException**
- **final void notify()**
- **final void notifyAll()**

Синхронизация. Example 10

```
package _java._se._07.synchro;
class MyResource {
	boolean ready = false;
	synchronized void waitFor() throws Exception {
		System.out.println(Thread.currentThread().getName()
				+ " is entering waitFor().");
		while (!ready)
			wait();
		System.out.println(Thread.currentThread().getName()
				+ " resuming execution.");
	}
	synchronized void start() {
		ready = true;
		notify();
	}
}
```

Синхронизация. Example 10

```
class MyThread implements Runnable {
      MyResource myResource;
      MyThread(String name, MyResource so) {
            myResource = so;
            new Thread(this, name).start();
      }
      public void run() {
            try {
                  myResource.waitFor();
            } catch (Exception e) {
                  e.printStackTrace();
            }
      }
}
public class WaitNotifyDemo {
      public static void main(String args[]) throws Exception {
            MyResource sObj = new MyResource();
            new MyThread("MyThread", sObj);
            for (int i = 0; i < 10; i++) {
                  Thread.sleep(50);
                  System.out.print(".");
            }
            sObj.start();
      }
}
```

Синхронизация. Example 10

Результат:

```
MyThread is entering waitFor().
..........MyThread resuming execution.
```

Синхронизация

Специальный тип ошибки, которую вам нужно избегать и которая специально относится к многозадачности, это — (взаимная) *блокировка.*

Она происходит, когда два потока имеют циклическую зависимость от пары синхронизированных объектов.

Например, предположим, что один поток вводит монитор в объект x, а другой поток вводит монитор в объект y. Если поток в x пробует вызвать любой синхронизированный метод объекта y, это приведет к блокировке, как и ожидается.

Однако если поток в y, в свою очередь, пробует вызвать любой синхронизированный метод объекта x, то он будет всегда ждать, т. к. для получения доступа к x, он был бы должен снять свою собственную блокировку с y, чтобы первый поток мог завершиться.

Синхронизация

Взаимоблокировка — трудная ошибка для отладки по двум причинам:

- Вообще говоря, она происходит очень редко, когда интервалы временного квантования двух потоков находятся в определенном соотношении.
- Она может включать больше двух потоков и синхронизированных объектов.

Синхронизация

Приостановка выполнения потока иногда полезна.

Например, отдельные потоки могут использоваться, чтобы отображать время дня. Если пользователь не хочет видеть отображения часов, то их поток может быть приостановлен. В любом случае приостановка потока — простое дело. После приостановки перезапуск потока также не сложен.

Механизмы приостановки, остановки и возобновления потоков различны для Java 2 и более ранних версий Java.

Синхронизация

До Java 2 для приостановки и перезапуска выполнения потока программа использовала методы **suspend ()** и **resume (),** которые определены в классе **Thread**. Они имеют такую форму:

final void suspend()

final void resume()

Класс Thread также определяет метод с именем stop(), который останавливает поток. Его сигнатура имеет следующий вид:

void stop()

Если поток был остановлен, то его нельзя перезапускать с помощью метода **resume().**

Синхронизация

В Java 2 запрещено использовать методы **suspend(), resume()** или **stop()** для управления потоком.

Поток должен быть спроектирован так, чтобы метод **run()** периодически проверял, должен ли этот поток приостанавливать, возобновлять или останавливать свое собственное выполнение.

Это, как правило, выполняется применением флажковой переменной, которая указывает состояние выполнения потока.

Пока флажок установлен на "выполнение", метод **run()** должен продолжать позволять потоку выполняться.

Если эта переменная установлена на "приостановить", поток должен сделать паузу. Если она установлена на "стоп", поток должен завершиться.

CONCURRENT

Concurrent

В Java версии 1.5 был добавлен новый пакет, содержащий много полезных возможностей, касающихся синхронизации и параллелизма: java.util.concurrent.

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.

Concurrent

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);

Concurrent

- 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;

Concurrent

- высокопроизводительный класс **Lock**, поддерживающий ограниченные ожидания снятия блокировки, прерываемые попытки блокировки, очереди блокировки и установку ожидания снятия нескольких блокировок посредством класса **Condition**;

- 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);

Concurrent

- классы атомарных переменных (**AtomicInteger**, **AtomicLong**, **AtomicReference**), а также их высокопроизводительные аналоги **SyncronizedInt** и др.;

- обработка неотловленных прерываний: класс **Thread** теперь поддерживает установку обработчика на неотловленные прерывания (подобное ранее было доступно только в **ThreadGroup**).

Executors

Пакет java.util.concurrent содержит три Executor-интерфейса:

- Executor
- ExecutorService
- ScheduledExecutorService

Также библиотека java.util.concurrent содержит специальный класс, который называют Executors. Объекты данного класса помогаю работать с потока не на прямую, а использовать исполнители. Данное решение бывает очень полезно когда вам необходимо запустить множество потоков, выполняющих одинаковые задачи.

Concurrent. Example 11

```
package executor;
public class MyThread implements Runnable {
     public int count = 0;
     public void run() {
          for (int i = 0; i < 1000000; i++) {
               count++;
          }
          System.out.println(count);
     }
}
```

```
package executor;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class Solution {
     public static void main(String[] args) {
          ExecutorService ex = Executors.newCachedThreadPool();
          ex.execute(new MyThread());
          ex.execute(new MyThread());
     }
}
```

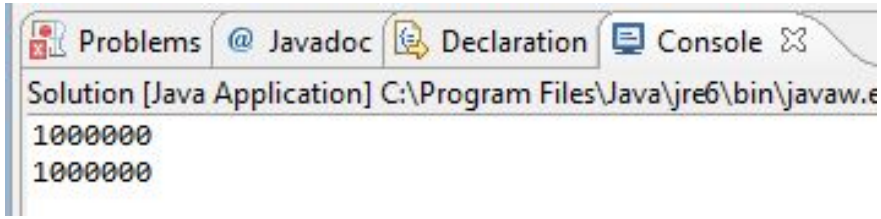
Concurrent

Сначала создается объект класса ExecutorService. После чего вызывается метод execute, которому в качестве параметра необходимо передать объект, созданного нами класса, который мы хотим передать исполнителю.

После передачи потока, исполнитель автоматически запускает его.

Исполнитель позволяет нам экономить время на создание отдельных потоков и на их запуск.

Результат:



Concurrent

В примере объекту «ex» присваивается специальная реализация **Executors.newCachedThreadPool()**. Данная реализация применяется в тех случаях, когда вы заранее неизвестно, какое количество потоков будет передаваться исполнителю.

Если же количество потоков заранее известно необходимо использовать реализацию **newFixedThreadPool(int)** в качестве параметра ей нужно передать число потоков, которое мы будем использовать, в нашем случае 2. Это дает большой выигрыш в быстродействии, так как все потоки создаются сразу.

Если же необходимо передавать исполнителю только один объект класса, то для таких целей можно использовать реализацию **newSingleThreadExecutor()**. Если при использовании данной реализации исполнителю передается несколько потоков, то они попадут в очередь, и каждый из них будет запускаться только после завершения работы предыдущего.

ExecutorService

Данный интерфейс является расширением интерфейса Executor и добавляет следующие полезные возможности:

- Возможность остановить выполняемый процесс
- Возможность выполнения не только Runnable объектов, но и java.util.concurrent.Callable. Основное их отличие от Runnable объектов – возможность возвращать значение потоку, из которого делался вызов.
- Возможность возвращать вызывавшему потоку объект java.util.concurrent.Future, который содержит среди прочего и возвращаемое значение.

Возврат значений из задач. Интерфейс Callable

Очень часто нам необходимо, чтобы поток после выполнения своей работы возвращал нам некоторое значение, в таких ситуациях нам необходимо использовать интерфейс Callable при создании класса. Он очень похож на Runnable, но имеет несколько отличий.

- В первую очередь после объявления данного интерфейса необходимо указать тип параметра, который должен вернуть поток.
- Вместо метода run() необходимо использовать метод call().

Concurrent. Example 12

```
package executorservice;
import java.util.concurrent.Callable;
public class MyThread implements Callable<Integer> {
     public int count = 0;
     public Integer call() {
          for (int i = 0; i < 1000000; i++) {
               count++;
          }

          try {
               Thread.sleep(10000);
          } catch (InterruptedException e) {
               e.printStackTrace();
          }
          return count;
     }
}
```

В данном примере метод call() вернет нам число после завершения операции.

Concurrent

Теперь рассмотрим способ получения полученного значения, используя исполнители.

Для передачи объекта, созданного нами класса исполнителя, используется метод «submit».

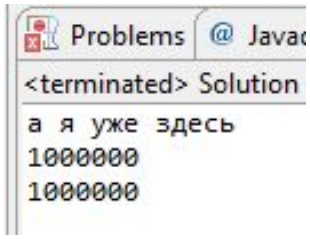
При вызове данного метода создается объект типа «Future» параметризованный по типу результата возвращаемого Callable. В нашем случае «Future<Integer>». В свою очередь из этого объекта мы уже можем получить нужный нам результат, используя метод get(). Данный метод всегда необходимо оборачивать в блок try-catch , так как поток еще может не закончить свою работу, а метод get() уже будет вызван.

Для проверки завершенности потока используется метод isDone(), он возвращает логическое значение.

Concurrent. Example 13

```
package executorservice;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
public class Solution {
  public static void main(String[] args) {
     ExecutorService ex = Executors.newCachedThreadPool();
     Future<Integer> s = ex.submit(new MyThread());
     Future<Integer> s1 = ex.submit(new MyThread());
     try {
          System.out.println("а я уже здесь");
          System.out.println(s.get());
          System.out.println(s1.get());
     } catch (InterruptedException e) {  e.printStackTrace();
     } catch (ExecutionException e) {e.printStackTrace();    }
  }
}
```

Результат:



Управление потоками. Ожидание

Существует несколько методов управления потоками. Давайте рассмотрим метод переводящий поток в состояние ожидания.

Для этого у класса «TimeUnit» выберем метод отвечающий за размерность времени, например «TimeUnit.MICROSECONDS», у этого метода есть метод «sleep», которому в качестве параметра нужно передать число, отвечающее за величину второго параметра(время проведенное в ожидании).

При реализации данного метода его необходимо поместить в блок try-catch.

Concurrent. Example 14

```
package timeunit;
import java.util.concurrent.Callable;
import java.util.concurrent.TimeUnit;
public class MyThread implements Callable<Integer> {
     public int count = 0;
     public Integer call() {
          for (int i = 0; i < 1000000; i++) {
               count++;
               try {
                    TimeUnit.MICROSECONDS.sleep(100);
               } catch (InterruptedException e) {
                    e.printStackTrace();
               }
          }
          return count;
     }
}
```

Механизм управления мьютексами Lock

Lock является явным механизмом управления мьютексами. Он находиться в библиотеке java.util.concurrent.

Объект класса Lock можно явно создать в программе и установить или снять блокировку с помощью его методов.

К плюсам данного объекта можно отнести возможность отлавливания исключений и другие.

Рассмотрим реализацию Lock для класса MyTread.

Concurrent. Example 15

```
package _java._se._07._concurrent;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class LockDemo implements Runnable{
	public static int count;
	private static Lock lock = new ReentrantLock();

	public void run() {
		for (int i = 0; i < 10000000; i++) {
			lock.lock();
			count++;
			lock.unlock();
		}
		System.out.println(count);
	}

	public static void main(String[] args){
		LockDemo lock1 = new LockDemo();
		LockDemo lock2 = new LockDemo();
		Thread th1 = new Thread(lock1);
		Thread th2 = new Thread(lock2);

		th1.start();
		th2.start();
	}
}
```

Атомарные операции. Volatile

Атомарные операции- это операции, которые не могут быть прерваны планировщиком потоков. Чтение и запись примитивных переменных кроме double и long являются атомарными. Даже если операция является атомарной, значение переменной может хранится в кэше ядра, и быть не видным другому потоку, поэтому для обеспечение видимости внутри приложения существует ключевое слово volatile. Но данное ключевое слово не обеспечивает атомарности операциям, не смотря на то что после записи, значение поле будет отображено сразу при всех операциях чтения. Так что приведенный код все еще содержит ошибку некорректного доступа.

```
public class MyThread implements Runnable {
     public static volatile int count;
     public void run() {
          for (int i = 0; i < 10000000; i++) {
               count++;
          }
          System.out.println(count);
     }
}
```

Concurrent. Example 16

Атомарные классы

Попробуем разрешить конфликт с помощью атомарных классов: AtomicInteger, Atomic Long ,AtomicReference и т.д. Данный класс гарантирует атомарное выполнение операций.

```
package atomic;
import java.util.concurrent.atomic.AtomicInteger;
public class MyThread implements Runnable {
     public static AtomicInteger count = new AtomicInteger(0);
     public void run() {
          for (int i = 0; i < 10000000; i++) {
               // count.incrementAndGet();
               // count.addAndGet(1);
               count.getAndAdd(1);
          }
          System.out.println(count);
     }
}
```

Concurrent. Example 17

Синхронизованные коллекции

```
package _java._se._07._concurrent;
import java.util.Random;
import java.util.concurrent.PriorityBlockingQueue;
class Task implements Comparable<Task> {
	private int taskNumer;
	public Task(int num) {
		taskNumer = num;
	}
	public int getTaskNumer() {
		return taskNumer;
	}
	public void setTaskNumer(int taskNumer) {
		this.taskNumer = taskNumer;
	}
	@Override
	public int compareTo(Task o) {
		Random rand = new Random();
		int comp = rand.nextInt(2000);
		if (comp % 2 == 0) return 1;
		if (comp % 3 == 0) return -1;
		else return 0;
	}
}
```

Concurrent. Example 17

```
class Manager implements Runnable {
      private QueueTask pbQ;
      private String name;
      public Manager(QueueTask q, String n) {
            pbQ = q;
            name = n;
      }
      public void run() {
            Task task;
            while ((task = pbQ.getTask()) != null) {
                  System.out.println(name + " get task number " +
task.getTaskNumer());
            }
      }
}
```

Concurrent. Example 17

```
class QueueTask{
      private PriorityBlockingQueue<Task> queue = new PriorityBlockingQueue<Task>();
      public Task getTask() {
            return queue.poll();
      }

      public void setTask(Task task) {
            queue.add(task);
      }
      public PriorityBlockingQueue<Task> getQueue() {
            return queue;
      }
}
```

Concurrent. Example 17

```
public class PriorityBlockingQueueDemo {
	public static void main(String[] args) {
		QueueTask pbQueue = new QueueTask();
		for (int i = 0; i < 10; i++) {
			pbQueue.setTask(new Task(i));
		}

		Manager manager1 = new Manager(pbQueue, "Jonh");
		Manager manager2 = new Manager(pbQueue, "Pol");

		Thread th1 = new Thread(manager1);
		Thread th2 = new Thread(manager2);

		th1.start();
		th2.start();

		try {
			th1.join();
			th2.join();
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
	}
}
```

Concurrent. Example 17

Результат:

```
Jonh get task number 2
Jonh get task number 8
Jonh get task number 7
Pol get task number 3
Jonh get task number 0
Pol get task number 5
Pol get task number 6
Pol get task number 4
Pol get task number 9
Jonh get task number 1
```

Concurrent. Example 17

А вот возможный результат использования в этом примере **PriorityQueue**

```
Jonh get task number 6
Pol get task number 6
Pol get task number 7
Pol get task number 2
Pol get task number 0
Pol get task number 4
Pol get task number 1
Pol get task number 5
Pol get task number 9
Jonh get task number 3
```

СПАСИБО ЗА ВНИМАНИЕ!

ВОПРОСЫ?

Java.SE.07

MultiThreading

Author: **Ihar Blinou, PhD**
Oracle Certified Java Instructor
Ihar_blinou@epam.com