EPAM Systems, RD Dep.

Конспект и раздаточный материал

JAVA.SE.07 Multithreading

REVISION HISTORY					
Ver.	**Description of Change**	**Author**	**Date**	**Approved**	
				Name	**Effective Date**
<1.0>	Первая версия	Игорь Блинов	<25.10.2011>		
<2.0>	Вторая версия. Конспект переделан под обновленное содержание материала модуля.	Ольга Смолякова	<17.06.2014>		

СОДЕРЖАНИЕ
JAVA.SE.07 MULTITHREADING

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).

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.

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.

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.

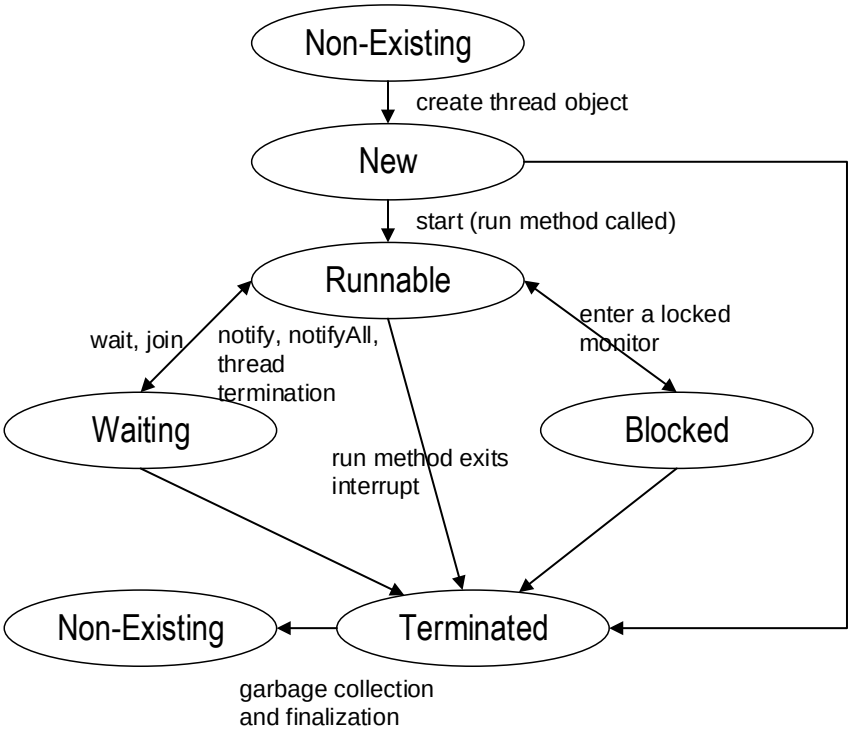
- Однопоточные системы используют подход, называемый *циклом событий с опросом* (event loop with polling).
- Выгода от многопоточности Java заключается в том, что устраняется механизм "главный цикл/опрос". Один поток может делать паузу без остановки других частей программы. Например, время простоя, образующееся, когда поток читает данные из сети или ждет ввод пользователя, может использоваться в другом месте.

Жизненный цикло потока

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.



Создание и выполнение потоков

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**.

```
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);
                    }
                }
            }
}

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 {
                        Thread.sleep(400);
                  } catch (InterruptedException e) {
                        System.err.print(e);
                  }
            }
      }
}

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

Некоторые методы класса Thread

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
getId()	Возвращает идентификатор потока
start()	Запустить поток с помощью вызова его метода run()
getState()	Возвращает константу Thread.State как состояние потока.

getThreadGroup()	Возвращает ссылку на ThreadGroup которой принадлежит поток.
interrupt()	Прерывает поток
isDaemon()	Определяет, является ли поток демоном
isInterrupted()	Проверяет, был ли прерван поток
setDaemon()	Устанавливает состояние потока: поток-демон или пользовательский поток
setName()	Устанавливает имя потока
setPriority()	Устанавливает приоритет потока
currentThread()	Возвращает ссылку на текущий исполняемый потоковый объект
interrupted()	Проверяет, был ли прерван текущий поток
yield()	Сообщает планировщику, что текущий поток может уступить свое текущее пользование процессором
sleep ()	Приостановить поток на определенный период времени

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 (рекомендация). Когда главный поток останавливается, программа завершается.

Хотя главный поток создается автоматически после запуска программы, он может управляться через Thread-объект. Для организации управления нужно получить ссылку на него, вызывая метод currentThread.

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.

```
package _java._se._07.startthread;
public class CurrentThreadDemo {
      public static void main(String args[]) {
            Thread t = Thread.currentThread();
            System.out.println("Текущий поток: " + t);
            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");
            }
      }
}
```

Существуют два способа определения, закончился ли поток:

- **isAlive()**
- **getState()**

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

final boolean isAlive()

Метод **getState**() возвращает одну из констант перечисления Thread.State, определяющую состояние потока.

Thread.State getState()

Сравнение **getState()** и **isAlive()** класса **Thread.**

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

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.

join(long millis)
join(long millis, int nanos)

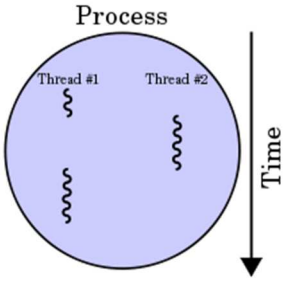
```
package _java._se._07.startthread;
public class DemoJoin {
      public static void main(String args[]) {
            NewThread ob1 = new NewThread("One");
            NewThread ob2 = new NewThread("Two");
            NewThread ob3 = new NewThread("Three");
            System.out.println("Thread 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());
		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.");
	}
}

class NewThread implements Runnable {
	String name;
	Thread t;
	NewThread(String threadname) {
		name = threadname;
		t = new Thread(this, name);
		System.out.println("New thread: " + t);
		t.start();
	}

	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.");
	}
}


package _java._se._07.startthread;
public class GetStateDemo implements Runnable{
	public void run() {
		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 {
			Thread.sleep(1000);
		} catch (Exception e) {
			System.out.println(e);
		}
		Thread.State state = th1.getState();
		System.out.println(th1.getName() + " " + state);
```

```
        }
}
```

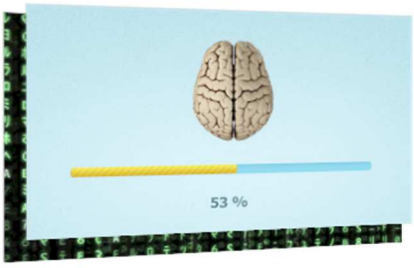
Приоритеты потоков

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



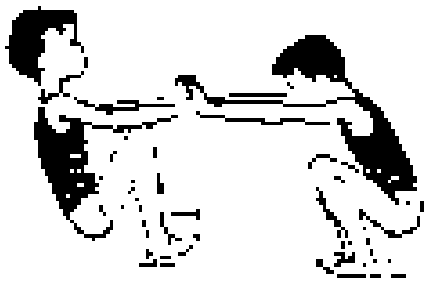
Теоретически высокоприоритетные потоки получают больше времени CPU, чем низкоприоритетные.

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.)



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.

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.

Для установки приоритета потока используется метод **setPriority()**, который является членом класса Thread:

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.

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

final int getPriority()

```
package _java._se._07.startthread;
public class PriorityDemo {
	public static void main(String args[]) {
		Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
		Clicker hi = new Clicker();
		Clicker lo = new Clicker();

		hi.setPriority(Thread.NORM_PRIORITY + 2);
		lo.setPriority(Thread.NORM_PRIORITY - 2);

		lo.start();
		hi.start();

		try {
			Thread.sleep(50);
		} catch (InterruptedException e) {
			System.out.println("Main thread interrupted.");
		}

		lo.stopClick();
		hi.stopClick();

		try {
			hi.join();
			lo.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);
	}
}

class Clicker extends Thread {
	int click = 0;
	private volatile boolean running = true;

	public Clicker() {
	}

	public void run() {
		while (running) {
			click++;
		}
	}

	public void stopClick() {
		running = false;
	}

}
```

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.

- Программа Java завершает работу, когда завершили работу все ее потоки, но это не совсем так. А как же скрытые системные потоки, такие как поток сбора мусора и другие служебные потоки, созданные JVM? Мы не можем их остановить. Если эти потоки продолжают работать, как же программа Java вообще завершит работу?
- Эти системные потоки называются демонами. На самом деле программа Java завершает работу, если завершены все ее потоки, не являющиеся демонами.

```
package _java._se._07._daemon;
public class DaemonThread extends Thread{

	public void run(){
		for(int i=0;i<10; i++){
			System.out.print(i + " ");
			try {
				Thread.sleep(1000);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}
}

package _java._se._07._daemon;
public class DaemonInspector {

	public static void main(String[] args) {
		System.out.println("Start main thread.");

		DaemonThread daemon = new DaemonThread();
		daemon.setDaemon(true);
		daemon.start();

		try {
			Thread.sleep(5000);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}

		System.out.println("End main thread.");
	}

}
```

Группы потоков

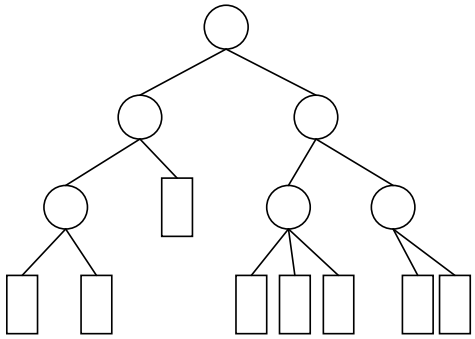
Для того, чтобы отдельный поток не мог начать останавливать и прерывать все потоки подряд, введено понятие группы.

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.

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 поддерживает следующие конструкторы и методы:

- **public ThreadGroup(String name)** - создает новый объект класса ThreadGroup, принадлежащий той группе потоков, к которой относится и поток-"родитель". Как и в случае объектов потоков, имена групп не используются исполняющей системой непосредственно, но в качестве параметра name имени группы может быть передано значение null.
- **public ThreadGroup(ThreadGroup parent, String name)** - создает новый объект класса ThreadGroup с указанным именем name в составе "родительской" группы потоков parent. Если в качестве parent передано значение null, выбрасывается исключение типа nullPointerException.

Метод	Значение
activeCount()	Возвращает количество активных потоков для группы и ее подгрупп
activeGroupCount()	Возвращает количество активных групп
checkAccess()	выбрасывает исключение типа SecurityException, если текущему потоку не позволено воздействовать на параметры группы потоков; в противном случае просто возвращает управление
destroy()	Уничтожает группу потоков и ее подгруппы. Группа не должна содержать потоков, иначе метод выбрасывает исключение типа IllegalThreadStateException. Если в составе группы

	имеются другие группы, они также не должны содержать потоков. Не уничтожает объекты потоков, принадлежащих группе.
enumerate(Thread[] list), enumerate(Thread[] list, boolean recurse)	Создает массив активных потоков из потоков принадлежащих группе [и подгруппам группы]
enumerate(ThreadGroup[] list), enumerate(ThreadGroup[] list, boolean recurse)	Создает массив активных подгрупп потоков для текущей группы и ее подгрупп.
getMaxPriority()	Максимально возможный приоритет группы потоков
getName()	Имя группы потоков
getParent()	Возвращает ссылку на объект "родительской" группы потоков либо null, если такового нет (последнее возможно только для группы потоков верхнего уровня иерархии).
interrupt()	Прерывает все потоки, входящие в группу

Обращение к методу interrupt объекта группы приводит к вызову методов interrupt для каждого потока в группе, включая и те, которые принадлежат вложенным группам.

Метод	**Значение**
isDaemon()	Определяет, является ли группа потоков демоном
isDestroyed()	Определяет, была ли данная группа потоков уничтожена
list()	Печатает информацию о группе потоков в стандартных поток вывода
parentOf(ThreadGroup g)	проверяет, является ли текущая группа "родительской" по отношению к группе g, либо совпадает с группой g
setDaemon()	Устанавливает группу потоков как группу-демон
setMaxPriority()	Устанавливает максимальный приоритет группе потоков
uncaughtException(Thread t, Throwable e)	Вызывается, когда поток t в текущей группе генерирует исключение e, которое далее не обрабатывается

- Группа потоков может быть *группой-демоном* (daemon group). "Демонический" объект ThreadGroup автоматически уничтожается, если он становится пустым.
- Задание признака принадлежности объекта ThreadGroup к категории групп-демонов не имеет отношения к тому, является ли любой из потоков, принадлежащих группе, потоком-демоном.

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.

```
package _java._se._07._threadgroup;
public class MyThread extends Thread {

	public MyThread(String threadname, ThreadGroup tgOb) {
		super(tgOb, threadname);
		System.out.println("New thread: " + this);
		start();
	}

	public void run() {
		try {
			for (int i = 5; i > 0; i--) {
				System.out.println(getName() + ": " + i);
				Thread.sleep(1000);
			}
		} catch (Exception e) {
			System.out.println("Exception in " + getName());
		}
		System.out.println(getName() + " exiting.");
	}

}
```

```
package _java._se._07._threadgroup;
public class ThreadGroupDemo {
	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);

		try {
			Thread.sleep(2500);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
```

```
            groupA.interrupt();
      }
}

package _java._se._07._threadgroup;
public class ThreadGroupListDemo {

      public static void main(String[] args) {
            ThreadGroup mainGroup = new ThreadGroup("mainGroup");
            ThreadGroup groupA = new ThreadGroup(mainGroup, "Group A");
            ThreadGroup groupB = new ThreadGroup(mainGroup, "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);

            groupA.list();
            groupB.list();

            mainGroup.interrupt();

      }
}
```

Обработка исключений

- Поток, породивший другой поток, не обрабатывает исключения дочернего потока.
- Если основной поток прекратит свое существование из-за необработанного исключения, это не скажется на работоспособности порожденного им потока.

Потоки Java предлагают следующие способы обработки неотловленных исключений:

- *Класс Thread*
  - **setDefaultUncaughtExceptionHandler**(**Thread.UncaughtExceptionHandler** eh)
  - **setUncaughtExceptionHandler**(**Thread.UncaughtExceptionHandler** eh)
- *Класс ThreadGroup*
  - **uncaughtException**(**Thread** t, **Throwable** e)

```
package _java._se._07._exception;
public class ThreadUncaughtExceptionDemo {

      public static void main(String[] args) {
            Thread t = new Thread(new SimpleThread());
            t.setUncaughtExceptionHandler(new
Thread.UncaughtExceptionHandler() {

                  public void uncaughtException(Thread t, Throwable e) {
                        System.out.println(t + " throws exception: " + e);
                  }
            });
            t.start();
      }
```

```
}

class SimpleThread implements Runnable {

	public void run() {
		throw new RuntimeException("It is a greate exception.");
	}
}



package _java._se._07._exception;
public class ThreadDefaultUncaughtExceptionDemo {
	public static void main(String[] args) {

		Thread.setDefaultUncaughtExceptionHandler(new
Thread.UncaughtExceptionHandler() {
			public void uncaughtException(Thread t, Throwable e) {
				System.out.println(t + " (default handler)throws
exception: " + e);
			}
		});

		Thread t1 = new Thread(new MyThread());
		Thread t2 = new Thread(new MyThread());

		t2.setUncaughtExceptionHandler(new
Thread.UncaughtExceptionHandler() {

			public void uncaughtException(Thread t, Throwable e) {
				System.out.println(t + " throws exception: " + e);
			}
		});

		t1.start();
		t2.start();
	}
}

class MyThread implements Runnable {
	public void run() {
		throw new RuntimeException();
	}
}


package _java._se._07._exception;
public class ThreadGroupUncaughtExceptionDemo {

	public static void main(String[] args) {
		NewThreadGroup g = new NewThreadGroup("one");

		ThreadD t1 = new ThreadD("1", g);
		ThreadD t2 = new ThreadD("2", g);
		ThreadD t3 = new ThreadD("3", g);
```

```
            t3.setUncaughtExceptionHandler(new
Thread.UncaughtExceptionHandler() {

                  public void uncaughtException(Thread t, Throwable e) {
                        System.out.println(t + " throws exception: " + e);
                  }
            });

            t1.start();
            t2.start();
            t3.start();

      }

}

class NewThreadGroup extends ThreadGroup {

    NewThreadGroup(String n) {
       super(n);
    }

    NewThreadGroup(ThreadGroup parent, String n) {
       super(parent, n);
    }

    public void uncaughtException(Thread t, Throwable e) {
       System.out.println(t + " has unhandled exception:" + e);
    }
}

class ThreadD extends Thread {

      public ThreadD(String threadname, ThreadGroup tgOb) {
            super(tgOb, threadname);
      }

      public void run() {
            throw new RuntimeException("Oy, exception!!!");
      }
}
```

Синхронизация

Поскольку многопоточность обеспечивает *асинхронное* поведение программ, необходимо правильно синхронизировать приложение, когда в этом возникает необходимость.

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

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***.

Только один поток может захватить и держать монитор в заданный момент.

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

Говорят, что другие потоки *ожидают* монитор.

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

Базовая синхронизация в Java возможна при использовании

- синхронизированных методов и
- синхронизированных блоков.

Оба подхода используют ключевое слово **synchronized**.

```
package _java._se._07._synchronized;
public class Account {
	private int balance;

	public Account(int balance){
		this.balance = balance;
	}

	public int getBalance(){
		return balance;
	}

	public void deposit(int amount){
		int x = balance + amount;
		try {
			Thread.sleep(15);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		balance = x;
	}

	public void withdraw(int amount){
		int x = balance - amount;
		try {
			Thread.sleep(20);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		balance = x;
	}
}

package _java._se._07._synchronized;
public class OperatorDeposit extends Thread {
	private Account account;

	public OperatorDeposit(Account account){
```

```
		this.account = account;
	}

	public void run(){
		for(int i=0; i<5; i++){
			account.deposit(100);
		}
	}
}

package _java._se._07._synchronized;
public class OperatorWithdraw extends Thread {
	private Account account;

	public OperatorWithdraw(Account account) {
		this.account = account;
	}

	public void run() {
		for (int i = 0; i < 5; i++) {
			account.withdraw(50);
		}
	}
}

package _java._se._07._synchronized;
public class OperationInspector {

	public static void main(String[] args) throws InterruptedException {
		Account account = new Account(200);
		OperatorDeposit opD = new OperatorDeposit(account);
		OperatorWithdraw opW = new OperatorWithdraw(account);

		opD.start();
		opW.start();

		opD.join();
		opW.join();

		System.out.println(account.getBalance());
	}
}
```

В Java каждый объект имеет свой собственный неявный связанный с ним монитор.

Чтобы ввести монитор объекта, просто вызывают метод, который был помечен 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.

```
package _java._se._07._synchronized;
public class Account {
```

```
	private int balance;

	public Account(int balance){
		this.balance = balance;
	}

	public int getBalance(){
		return balance;
	}

	public synchronized void deposit(int amount){
		int x = balance + amount;
		try {
			Thread.sleep(15);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		balance = x;
	}

	public synchronized void withdraw(int amount){
		int x = balance - amount;
		try {
			Thread.sleep(20);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		balance = x;
	}
}
```

Если необходимо синхронизировать доступ к объектам класса, который не был разработан для многопоточного доступа, то есть класс не использует синхронизированные методы, или класс был создан не вами, а третьим лицом, и вы не имеете доступа к исходному коду, то решением синхронизации может быть применение

синхронизованного блока.

Общая форма оператора **synchronized**:

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

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

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

```
package _java._se._07._synchronized;
public class Account {
	private int balance;
	private Object lock = new Object();

	public Account(int balance){
		this.balance = balance;
	}
```

```
	public int getBalance(){
		return balance;
	}

	public void deposit(int amount){
		synchronized (lock) {
			int x = balance + amount;
			try {
				Thread.sleep(15);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
			balance = x;
		}
	}

	public void withdraw(int amount){
		synchronized (lock) {
			int x = balance - amount;
			try {
				Thread.sleep(20);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
			balance = x;
		}
	}
}

package _java._se._07._synchronized;
public class Account {
	private int balance;

	public Account(int balance){
		this.balance = balance;
	}

	public int getBalance(){
		return balance;
	}

	public void deposit(int amount){
		synchronized (this) {
			int x = balance + amount;
			try {
				Thread.sleep(15);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
			balance = x;
		}
	}

	public void withdraw(int amount){
		synchronized (this) {
			int x = balance - amount;
			try {
```

```
                        Thread.sleep(20);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    balance = x;
                }
            }
        }

package _java._se._07._synchronized;
public class OperatorDeposit extends Thread {
        private Account account;

        public OperatorDeposit(Account account){
                this.account = account;
        }

        public void run(){
                for(int i=0; i<5; i++){
                        synchronized (account) {
                                account.deposit(100);
                        }
                }
        }
}

package _java._se._07._synchronized;
public class OperatorWithdraw extends Thread {
        private Account account;

        public OperatorWithdraw(Account account) {
                this.account = account;
        }

        public void run() {
                for (int i = 0; i < 5; i++) {
                        synchronized (account) {
                                account.withdraw(50);
                        }
                }
        }
}
```

Итак, ключевое слово synchronized может применяться либо к блоку, либо к методу.

Оно указывает, что перед входом в блок или метод поток должен получить соответствующую блокировку. Для метода это означает получение блокировки, относящейся к экземпляру объекта (либо блокировки, относящейся к объекту *Class*, - для методов **static synchronized**).

```
package _java._se._07._synchronized;
class StaticSynch{

        public static synchronized void a() throws InterruptedException{
                System.out.println("Line #1 in the method a");
                Thread.sleep(1000);
```

```
            System.out.println("Line #2 in the method a");
      }

      public static synchronized void b() throws InterruptedException{
            System.out.println("Line #1 in the method b");
            Thread.sleep(1000);
            System.out.println("Line #2 in the method b");
      }
}

public class StaticSynchnizedDemo {

      public static void main(String[] args){

            new Thread(){
                  public void run(){
                        for(int i=0; i<5; i++){
                              try {
                                    StaticSynch.a();
                                    Thread.sleep(20);
                              } catch (InterruptedException e) {
                                    e.printStackTrace();
                              }
                        }
                  }
            }.start();

            for(int i=0; i<5; i++){
                  try {
                        StaticSynch.b();
                        Thread.sleep(20);
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
            }
      }
}
```

Wait, notify

Можно 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.

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.

- **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()**

```
package _java._se._07._waitnotify;
import java.util.ArrayList;
import java.util.List;
public class SharedResource {
	private List<Integer> list;

	public SharedResource() {
		list = new ArrayList<Integer>();
	}

	public void setElement(Integer element) {
		list.add(element);
	}

	public Integer getELement() {
		if (list.size() > 0) {
			return list.remove(0);
		}
		return null;
	}

}
```

```
package _java._se._07._waitnotify;
import java.util.Random;

public class UserResourceThread {
	public static void main(String[] args) throws InterruptedException {
		SharedResource res = new SharedResource();
		IntegerSetterGetter t1 = new IntegerSetterGetter("1", res);
		IntegerSetterGetter t2 = new IntegerSetterGetter("2", res);
		IntegerSetterGetter t3 = new IntegerSetterGetter("3", res);
		IntegerSetterGetter t4 = new IntegerSetterGetter("4", res);
```

```
            IntegerSetterGetter t5 = new IntegerSetterGetter("5", res);

            t1.start();
            t2.start();
            t3.start();
            t4.start();
            t5.start();

            Thread.sleep(100);

            t1.stopThread();
            t2.stopThread();
            t3.stopThread();
            t4.stopThread();
            t5.stopThread();

            t1.join();
            t2.join();
            t3.join();
            t4.join();
            t5.join();

            System.out.println("main");
      }
}

class IntegerSetterGetter extends Thread {
      private SharedResource resource;
      private boolean run;

      private Random rand = new Random();

      public IntegerSetterGetter(String name, SharedResource resource) {
            super(name);
            this.resource = resource;
            run = true;
      }

      public void stopThread() {
            run = false;
      }

      public void run() {
            int action;

            try {
                  while (run) {
                        action = rand.nextInt(1000);
                        if (action % 2 == 0) {
                              getIntegersFromResource();
                        } else {
                              setIntegersIntoResource();
                        }
                  }
                  System.out.println("Поток " + getName() + " завершил
работу.");
            } catch (InterruptedException e) {
```

```
                e.printStackTrace();
            }
        }

        private void getIntegersFromResource() throws InterruptedException {
            Integer number;

            synchronized (resource) {
                System.out.println("Поток " + getName()
                        + " хочет извлечь число.");
                number = resource.getELement();
                while (number == null) {
                    System.out.println("Поток " + getName()
                            + " ждет пока очередь заполнится.");
                    resource.wait();
                    System.out
                            .println("Поток " + getName() + "
возобновил работу.");
                    number = resource.getELement();
                }
                System.out
                        .println("Поток " + getName() + " извлек число
" + number);
            }
        }

        private void setIntegersIntoResource() throws InterruptedException {
            Integer number = rand.nextInt(500);
            synchronized (resource) {
                resource.setElement(number);
                System.out.println("Поток " + getName() + " записал число "
                        + number);
                resource.notify();
            }
        }
}
```

Deadlocks

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.

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.

```
package _java._se._07._deadlock;
public class Account {
	private int balance;

	public Account(int balance) {
		this.balance = balance;
	}

	public int getBalance() {
		return balance;
	}

	public void deposit(int amount) {
		balance = balance + amount;
	}

	public void withdraw(int amount) {
		balance = balance - amount;
	}
}
```

```
package _java._se._07._deadlock;
public class Operator extends Thread {
	private Account account1;
	private Account account2;

	public Operator(Account account1, Account account2){
		this.account1 = account1;
		this.account2 = account2;
	}

	public void run(){
		for(int i=0; i<3; i++){
			operationDeposit(10);
		}
	}

	private void operationDeposit(int depositSum){
		synchronized (account1) {
			System.out.println("Заблокирован первый счет.");
			synchronized (account2) {
				System.out.println("Заблокирован второй счет.");
				account1.deposit(depositSum);
				account2.withdraw(depositSum);
			}
		}
	}
}
```

```
package _java._se._07._deadlock;
public class OperatorDemo {

	public static void main(String[] args) {
		Account acc1 = new Account(200);
		Account acc2 = new Account(300);
```

```
            Operator op1 = new Operator(acc1, acc2);
            Operator op2 = new Operator(acc2, acc1);

            op1.start();
            op2.start();

      }
}
```

```
Administrator: C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Olga_Smolyakova@epam.com>jps
4128 OperatorDemo
7604 Program
6168 Jps

C:\Users\Olga_Smolyakova@epam.com>jstack 4128
2014-06-16 12:36:41
Full thread dump Java HotSpot(TM) 64-Bit Server VM (24.45-b08 mixed mode):

"DestroyJavaVM" prio=6 tid=0x0000000001b6d800 nid=0xe04 waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
```

```
Found one Java-level deadlock:
=============================
"Thread-1":
  waiting to lock monitor 0x00000000083dfd18 (object 0x00000000eb652a60, a _java._se._07._deadlock.Account),
  which is held by "Thread-0"
"Thread-0":
  waiting to lock monitor 0x00000000083de718 (object 0x00000000eb652a70, a _java._se._07._deadlock.Account),
  which is held by "Thread-1"

Java stack information for the threads listed above:
===================================================
"Thread-1":
        at _java._se._07._deadlock.Operator.operationDeposit(Operator.java:21)
        - waiting to lock <0x00000000eb652a60> (a _java._se._07._deadlock.Account)
        - locked <0x00000000eb652a70> (a _java._se._07._deadlock.Account)
        at _java._se._07._deadlock.Operator.run(Operator.java:14)
"Thread-0":
        at _java._se._07._deadlock.Operator.operationDeposit(Operator.java:21)
        - waiting to lock <0x00000000eb652a70> (a _java._se._07._deadlock.Account)
        - locked <0x00000000eb652a60> (a _java._se._07._deadlock.Account)
        at _java._se._07._deadlock.Operator.run(Operator.java:14)

Found 1 deadlock.
```

Один из способов, позволяющих не попадать во взаимные блокировки, предполагает всегда получать блокировки для каждого потока в одном и том же порядке.

```
      private void operationDeposit2(int depositSum){
            int hashAcc1 = account1.hashCode();
            int hashAcc2 = account2.hashCode();

            Account acc1=null, acc2=null;

            if (hashAcc1 < hashAcc2){
                  acc1 = account1;
                  acc2 = account2;
```

```
		} else{
			acc1 = account2;
			acc2 = account1;
		}

		synchronized (acc1) {
			System.out.println("Заблокирован первый счет.");
			synchronized (acc2) {
				System.out.println("Заблокирован второй счет.");
				account1.deposit(depositSum);
				account2.withdraw(depositSum);
			}
		}
		System.out.println("Счета разблокированы.");
	}
```

Volatile

Поле **volatile** регулируется следующими правилами:

- Любое значение, видимое потоком, всегда повторно считывается из основной памяти
- Любое значение, записываемое потоком, всегда пробрасывается в основную память до того, как завершится выполнение инструкции.

Переменная **volatile** не вызывает никаких блокировок, т.е. “не предоставляет возможность” попасть во взаимную блокировку.

При обеспечении истинной безопасности потоков переменная **volatile** должна использоваться только для моделирования такой переменной, запись в которую не зависит от текущего (считываемого) состояния.

```
package _java._se._07._volatile;
public class VolatileDemo {

	public static void main(String[] args) throws InterruptedException {
		Clicker click1 = new Clicker();
		click1.start();
		Thread.sleep(50);

		click1.stopClick();
		click1.join();

		System.out.println("Последний оператор метода main()");
	}

}

class Clicker extends Thread {
	private int click = 0;
	private volatile boolean running = true;

	public Clicker() {
	}

	public void run() {
		while (running) {
```

```
                click++;
            }
        }

        public void stopClick() {
            running = false;
        }
}
```

Приостановка/возобновление работы потока

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.

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().**

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.

```
package _java._se._07._suspend;
import java.text.SimpleDateFormat;
import java.util.Date;
public class SuspendResumeDemo {

    public static void main(String[] args) throws InterruptedException {
        ConsoleClock clock = new ConsoleClock();
```

```
            clock.start();

            Thread.sleep(3000);

            clock.suspend();

            Thread.sleep(3000);

            clock.resume();

      }
}

class ConsoleClock extends Thread{

      public void run(){
            for (int i=0; i<10; i++){
                  System.out.println(i+ " - " + time());
                  try {
                        Thread.sleep(1000);
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
            }

      }

      private String time(){
            Date d = new Date();
            SimpleDateFormat s = new SimpleDateFormat("hh/mm/ss");
            return s.format(d);
      }
}
```

Concurrent, обзор

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.

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 **Executors**, 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 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);
- 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

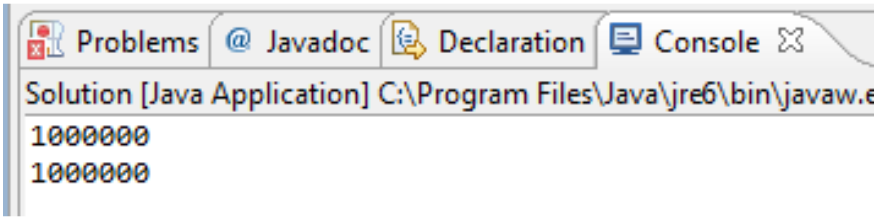
Также библиотека java.util.concurrent содержит специальный класс, который называют Executors. Объекты данного класса помогают работать с потока не на прямую, а использовать исполнители. Данное решение бывает очень полезно когда вам необходимо запустить множество потоков, выполняющих одинаковые задачи.

```
package _java._se._07._concurrent._executors;
public class SimpleThread implements Runnable{
	public int count = 0;
	public void run() {
		for (int i = 0; i < 1000000; i++) {
			count++;
		}
		System.out.println(count);
	}

}
```

```
package _java._se._07._concurrent._executors;
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 SimpleThread());
            ex.execute(new SimpleThread());
            ex.shutdown();
        }
}
```

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:



Executors.newCachedThreadPool() - данная реализация применяется в тех случаях, когда вы заранее неизвестно, какое количество потоков будет передаваться исполнителю.

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

Executors.newSingleThreadExecutor() - 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

Бывает необходимо, чтобы поток после выполнения своей работы возвращал некоторое значение, в таких ситуациях 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().

```
package _java._se._07._concurrent._executors;
import java.util.concurrent.Callable;
public class CallableThread 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 число после завершения операции.

```
package _java._se._07._concurrent._executors;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
public class CollableSolution {
	public static void main(String[] args) {
		ExecutorService ex = Executors.newCachedThreadPool();
		Future<Integer> s = ex.submit(new CallableThread());
		Future<Integer> s1 = ex.submit(new CallableThread());
		try {
			System.out.println("а я уже здесь");
			System.out.println(s.isDone());
			System.out.println(s.get());
			System.out.println(s1.get());
			System.out.println(s.isDone());
		} catch (InterruptedException e) {
			e.printStackTrace();
		} catch (ExecutionException e) {
			e.printStackTrace();
		}
	}
}
```

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.

TimeUnit, ожидание

```
package _java._se._07._concurrent._timeunit;
import java.util.concurrent.TimeUnit;
public class TimeUnitThread {
	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;
	}
}
```

Lock

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.

```
package _java._se._07._concurrent._lock;
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();
      }
}
```

Atomic

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.

```
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);
      }
}
```

Использование атомарных классов: AtomicInteger, Atomic Long ,AtomicReference и т.д. Данный класс гарантирует атомарное выполнение операций.

```
package _java._se._07._concurrent._atomic;
import java.util.concurrent.atomic.AtomicInteger;
public class AtomicThread {
      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);
      }
}
```

Синхронизированные коллекции

```
package _java._se._07._concurrent._collection;
import java.util.Random;
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;
	}
}

package _java._se._07._concurrent._collection;
import java.util.concurrent.PriorityBlockingQueue;
public 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;
	}
}

package _java._se._07._concurrent._collection;
public 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());
		}
	}
}

package _java._se._07._concurrent._collection;
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();
                }
          }
    }
```