**Многопоточность**

64. Что такое процесс? Что такое поток? Состояния потока.

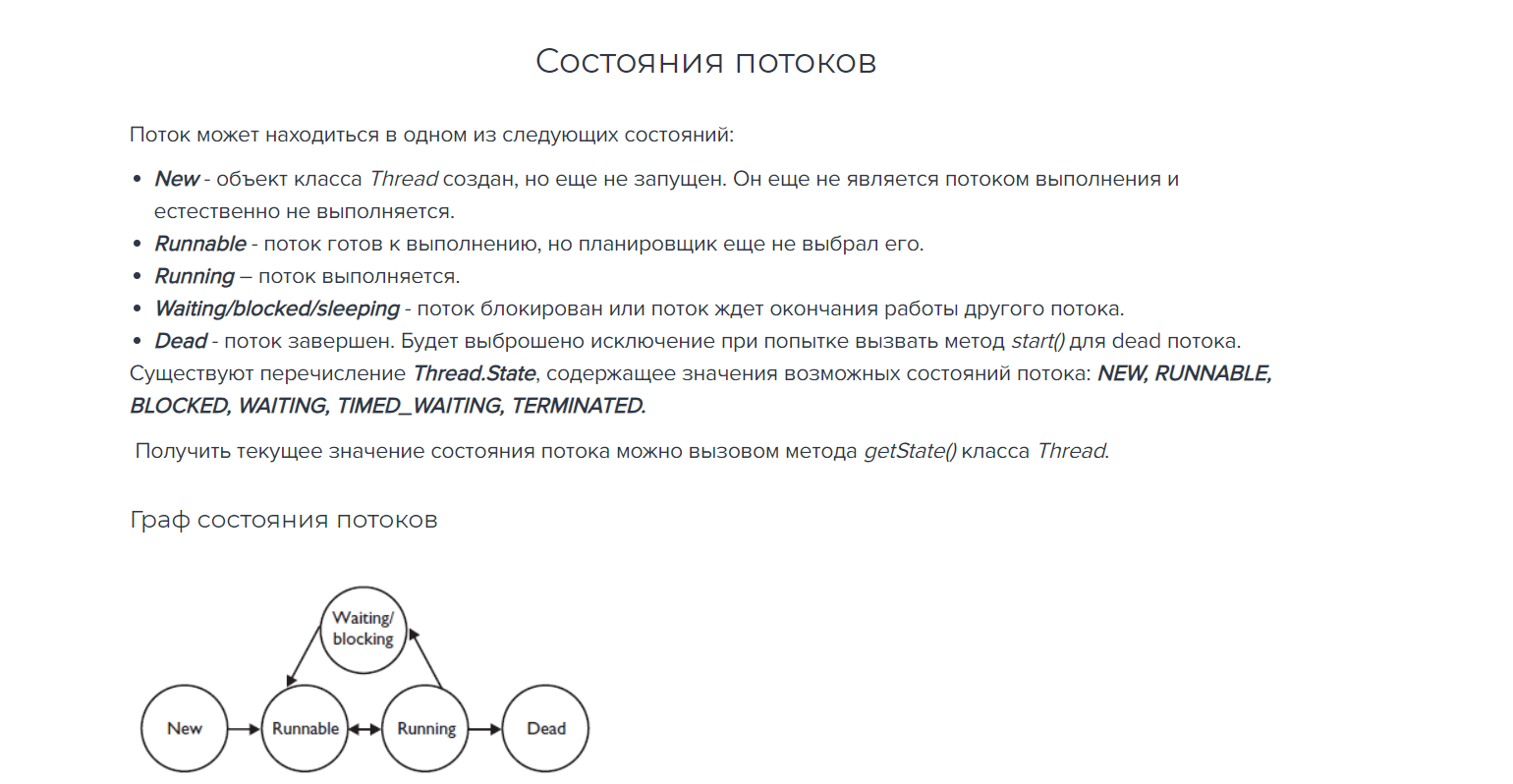
**Процесс**

Процесс (process) - это объект, который создается операционной системой, когда пользователь запускает приложение. Процессу выделяется отдельное адресное пространство, причем это пространство физически недоступно для других процессов. Процесс может работать с файлами или с каналами связи локальной или глобальной сети. Когда вы запускаете текстовый процессор или программу калькулятора, вы создаете новый процесс.

**Поток**

Для каждого процесса операционная система создает один главный поток (thread ), который является потоком выполняющихся по очереди команд центрального процессора. При необходимости главный поток может создавать другие потоки, пользуясь для этого программным интерфейсом операционной системы.

Все потоки, созданные процессом, выполняются в адресном пространстве этого процесса и имеют доступ к ресурсам процесса. Однако поток одного процесса не имеет никакого доступа к ресурсам потока другого процесса, так как они работают в разных адресных пространствах. При необходимости организации взаимодействия между процессами или потоками, принадлежащими разным процессам, следует пользоваться системными средствами, специально предназначенными для этого.



65. Как создать поток? Какими способами можно создать поток, запустить его, прервать (завершить, убить)?

Cуществует всего 4 способа создания потока в java:

1. Расширяя java. lang. Thread класс + start()
2. Внедряя java. lang. Runnable интерфейс + run()
3. Используя анонимный внутренний класс
4. Внедрение интерфейса Callable .

66. Как выполнить набор команд в отдельном потоке?

67. Как работают методы wait и notify/notifyAll?

* **wait()**: освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод notify()
* **notify()**: продолжает работу потока, у которого ранее был вызван метод wait()
* **notifyAll()**: возобновляет работу всех потоков, у которых ранее был вызван метод wait()

Все эти методы вызываются только из синхронизированного контекста - синхронизированного блока или метода.

68. Чем отличается работа метода wait() с параметром и без параметра?

Разница методов в следующем:  
  
final void wait() - метод используется в многопоточной среде, может вызываться только потоком, владеющим объектом синхронизации. При этом объект синхронизации освобождается, а текущий поток переходит в режим ожидания сигнала освобождения объекта синхронизации другим потоком путем вызова метода notify() либо notifyAll().

final void wait(long time) - аналогично wait() данный метод используется в многопоточной среде, переходит текущий поток в режим ожидания сигнала освобождения объекта синхронизации другим потоком путем вызова метода notify() либо notifyAll(), или ожидание происходит заданное время time, затем выполнение продолжается безусловно.

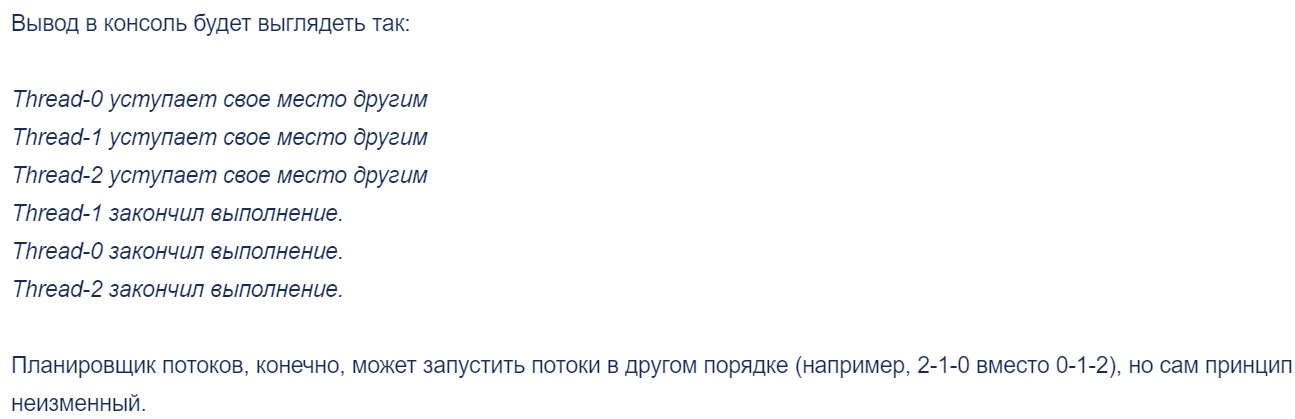
69. Как работает метод yield()? Чем отличаются методы Thread.sleep() и Thread.yield()?

**Метод yield()** с английского переводится как «уступать».

Когда мы вызываем метод yield у потока, он фактически говорит другим потокам: «Так, ребята, я никуда особо не тороплюсь, так что если кому-то из вас важно получить время процессора — берите, мне не срочно».







Thread.yield() VS Thread.sleep():

Thread.yield()**:** это подсказка (не гарантирующая ничего точно) планировщику, что вы сделали достаточно и что какой-то другой поток с тем же приоритетом может запускаться и использовать процессор.

Состояние потока: running -> runnable.

Thread.sleep(): блокирует выполнение конкретного потока в течение определенного времени.

Состояние потока: running -> waiting/sleeping.

70. Чем отличаются методы Thread.sleep и wait?

wait():

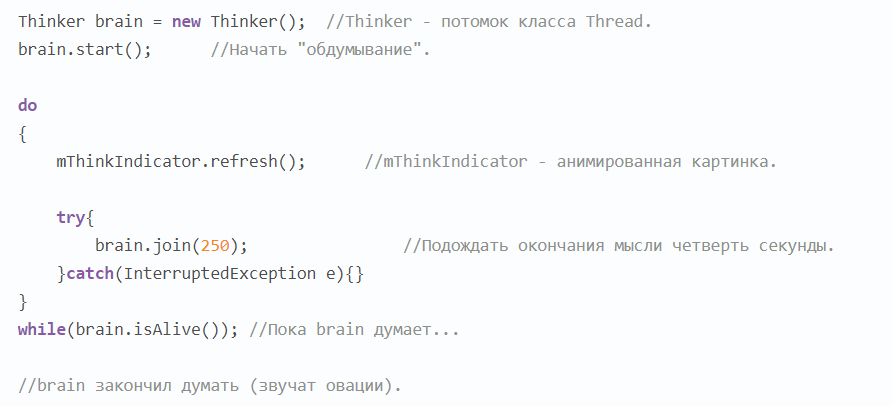
* Метод wait () снимает блокировку.
* wait () — это метод класса Object.
* wait () — нестатический метод — public final void wait () генерирует InterruptedException {//…}
* wait () должен быть уведомлен методами notify () или notifyAll ().
* Метод wait () должен вызываться из цикла, чтобы справиться с ложной тревогой.
* Метод wait () должен вызываться из синхронизированного контекста (т. е. синхронизированного метода или блока), в противном случае он вызовет исключение IllegalMonitorStateException

sleep():

* Метод sleep () не снимает блокировку.
* sleep () — это метод класса java.lang.
* sleep () — это статический метод — public static void sleep (long millis, int nanos) выдает InterruptedException {//…}
* по истечении указанного времени сон () завершен.
* sleep () лучше не вызывать из цикла (т.е. см. код ниже).
* sleep () может вызываться из любого места. нет особых требований.

71. Как работает метод join()?

В Java предусмотрен механизм, позволяющий одному потоку ждать завершения выполнения другого. Для этого используется метод join(). Например, чтобы главный поток подождал завершения побочного потока myThready, необходимо выполнить инструкцию myThready.join() в главном потоке. Как только поток myThready завершится, метод join() вернет управление, и главный поток сможет продолжить выполнение.  
  
Метод join() имеет перегруженную версию, которая получает в качестве параметра время ожидания. В этом случае join() возвращает управление либо когда завершится ожидаемый поток, либо когда закончится время ожидания. Подобно методу Thread.sleep() метод join может ждать в течение миллисекунд и наносекунд – аргументы те же.  
  
С помощью задания времени ожидания потока можно, например, выполнять обновление анимированной картинки пока главный (или любой другой) поток ждёт завершения побочного потока, выполняющего ресурсоёмкие операции:



В этом примере поток brain (мозг) думает над чем-то, и предполагается, что это занимает у него длительное время. Главный поток ждет его четверть секунды и, в случае, если этого времени на раздумье не хватило, обновляет «индикатор раздумий» (некоторая анимированная картинка). В итоге, во время раздумий, пользователь наблюдает на экране индикатор мыслительного процесса, что дает ему знать, что электронные мозги чем-то заняты.

72. Как правильно завершить работу потока? (Иногда говорят, убить поток).

73. Что такое синхронизация? Зачем она нужна? Для чего нужно ключевое слово synсhronized? Какие методы синхронизации вы знаете? Какими средствами достигается?

74. Отличия работы synсhronized от Lock?

Разница между synсhronized и Lock:

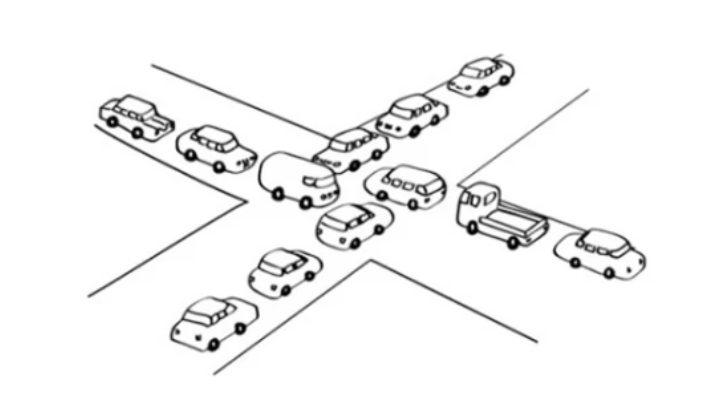
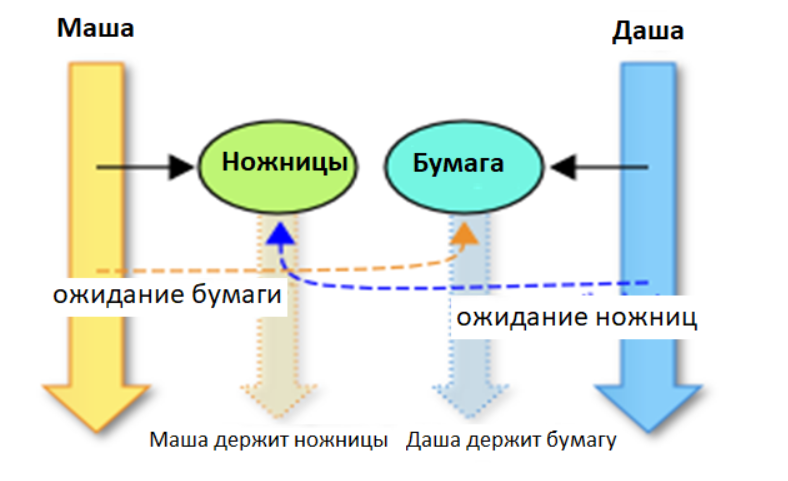
* Lock — это интерфейс, а synchronized — ключевое слово.
* synсhronized снимает блокировку автоматически, а Lock должна снять блокировку вручную.
* Lock может прервать ответ потока, ожидающего блокировки, но synchronized не будет, и поток будет ждать все время.
* Lock позволяет узнать, есть ли у потока блокировка, а synсhronized нет.
* Lock может повысить эффективность нескольких потоков.
* synсhronized блокирует классы, методы и кодовые блоки, в то время как Lock — для всего блока.

75. Есть ли у Lock механизм, аналогичный механизму wait\notify у synchronized?

76. Что такое deadlock? Нарисовать схему, как это происходит.

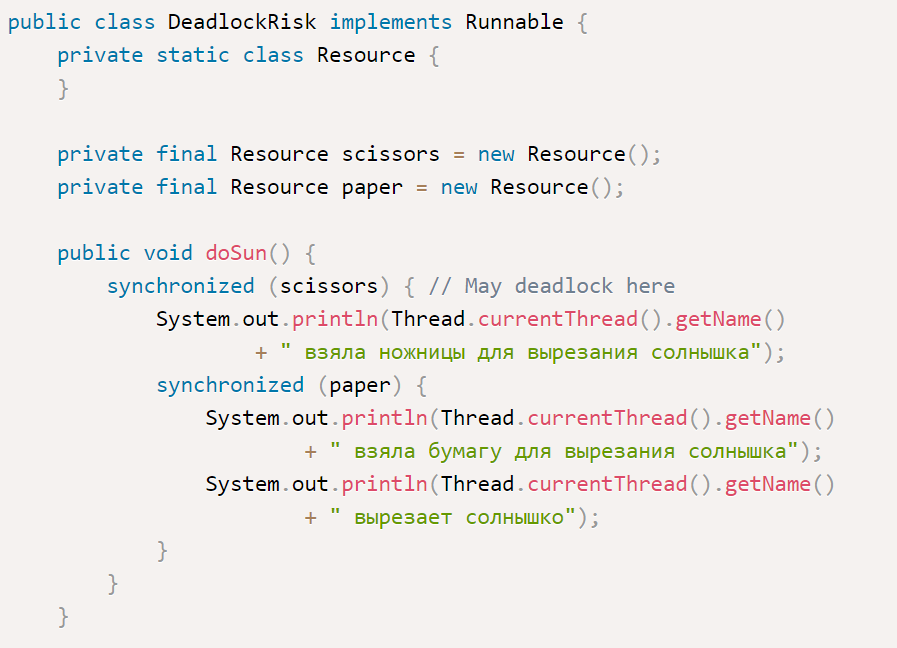
Deadlock или взаимная блокировка — это ошибка, которая происходит когда нити имеют циклическую зависимость от пары синхронизированных объектов. Представьте, что одна нить входит в монитор объекта x, а другая — объекта y. Если нить в объекте x пытается вызвать любой синхронизированный метод объекта y, а объект y в то же самое время пытается вызвать любой синхронизированный метод объекта x, то нити застрянут в процессе ожидания.

Примеры из жизни:

Например, две девочки Маша и Даша в детском саду делают аппликацию. Для работы каждой нужны ножницы и цветная бумага. Предположим Маша взяла ножницы (поток Маша вошла в монитор объекта ножницы), а Даша бумагу (поток Даша вошла в монитор объекта бумага). Каждая из них ждет другой предмет и не хочет делиться тем, что взяла. Они не могут продолжить свою работу и будут ждать вечно.

Пример кода:





77. Semaphore, CyclicBarrier, CountDownLatch. Чем похожи на Lock и чем от него отличаются?

78. Написать deadlock, придумать примеры с использованием synchronized, AtomicInteger

79. По каким объектам синхронизируются статические и нестатические методы?

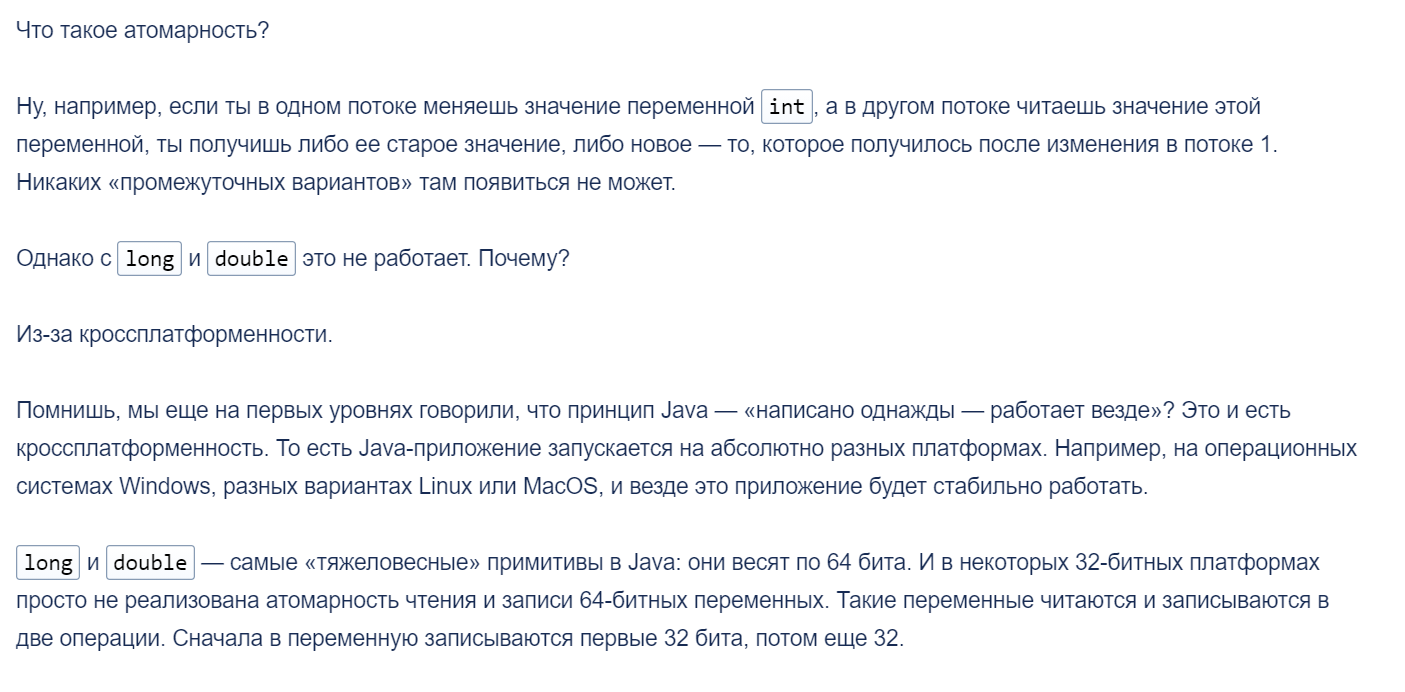
80. Для чего применяется volatile? Пакет java.util.concurrent.atomic.

Имеем 2 проблемы:

**1. В процессе работы многопоточного приложения разные потоки могут кэшировать значения переменных.** Возможна ситуация, когда один поток изменил значение переменной, а второй не увидел этого изменения, потому что работал со своей, кэшированной копией переменной.

**(**Естественно, последствия могут быть серьезными. Представь, что это не просто какая-то «переменная», а, например, баланс твоей банковской карты, который вдруг начал рандомно скакать туда-сюда :)**).**

**2. В Java операции чтения и записи полей всех типов, кроме long и double, являются атомарными.**

****

Если мы объявляем в нашей программе какую-то переменную, со словом **volatile** это означает, что:

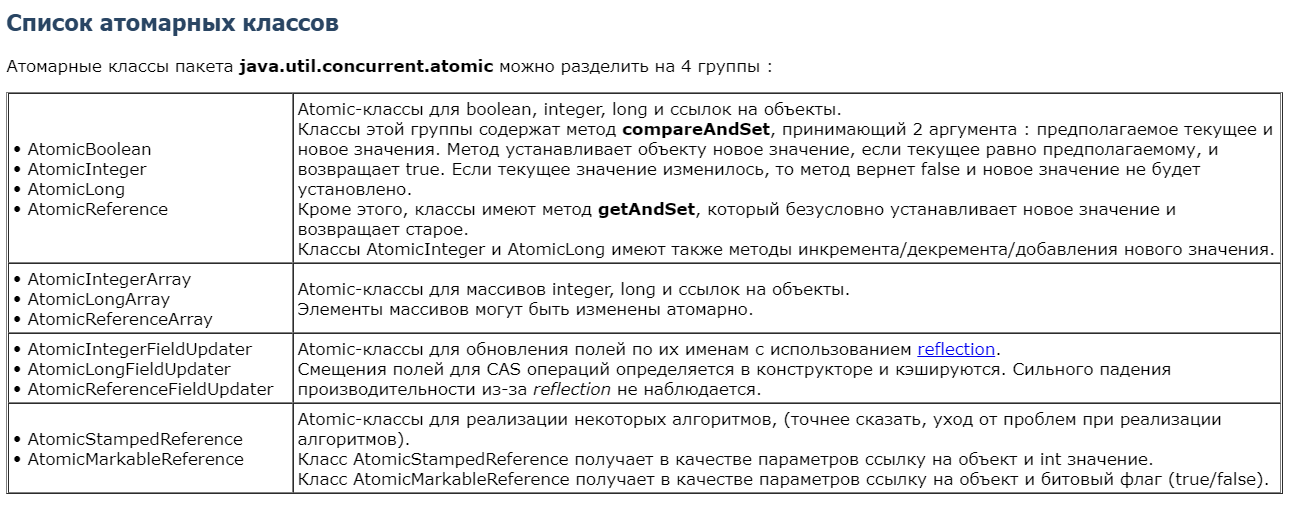
1. Она всегда будет атомарно читаться и записываться. Даже если это 64-битные double или long.
2. Java-машина не будет помещать ее в кэш. Так что ситуация, когда 10 потоков работают со своими локальными копиями исключена. (т.е. гарантирует выполнение отношения happens-before).

Пакет **java.util.concurrent.atomic** содержит классы для выполнения атомарных операций. Операция называется атомарной, если её можно безопасно выполнять при параллельных вычислениях в нескольких потоках, не используя при этом ни блокировок, ни синхронизацию [synchronized](http://java-online.ru/java-thread.xhtml#synchronized). Прежде, чем перейти к рассмотрению атомарных классов, рассмотрим выполнение наипростейших операций инкремента и декремента целочисленных значений.

Блокировка подразумевает **пессимистический** подход, разрешая только одному потоку выполнять определенный код, связанный с изменением значения некоторой «общей» переменной. Таким образом, никакой другой поток не имеет доступа к определенным переменным. Но можно использовать и **оптимистический** подход. В этом случае блокировки не происходит, и если поток обнаруживает, что значение переменной изменилось другим потоком, то он повторяет операцию снова, но уже с новым значением переменной. Так работают атомарные классы.

*Метод****compareAndSet****реализует механизм оптимистической блокировки.*

Основная выгода от атомарных (CAS) операций появляется только при условии, когда переключать контекст процессора с потока на поток становится менее выгодно, чем немного покрутиться в цикле while, выполняя метод *boolean compareAndSwap(oldValue, newValue)*. Если время, потраченное в этом цикле, превышает 1 квант потока, то, с точки зрения производительности, может быть невыгодно использовать атомарные переменные.



81. Есть массив из N-ти элементов. Создать N потоков, которые принимают по числу из массива, обрабатывают и возвращают обратно. Собрать все обработанные числа обратно в массив.