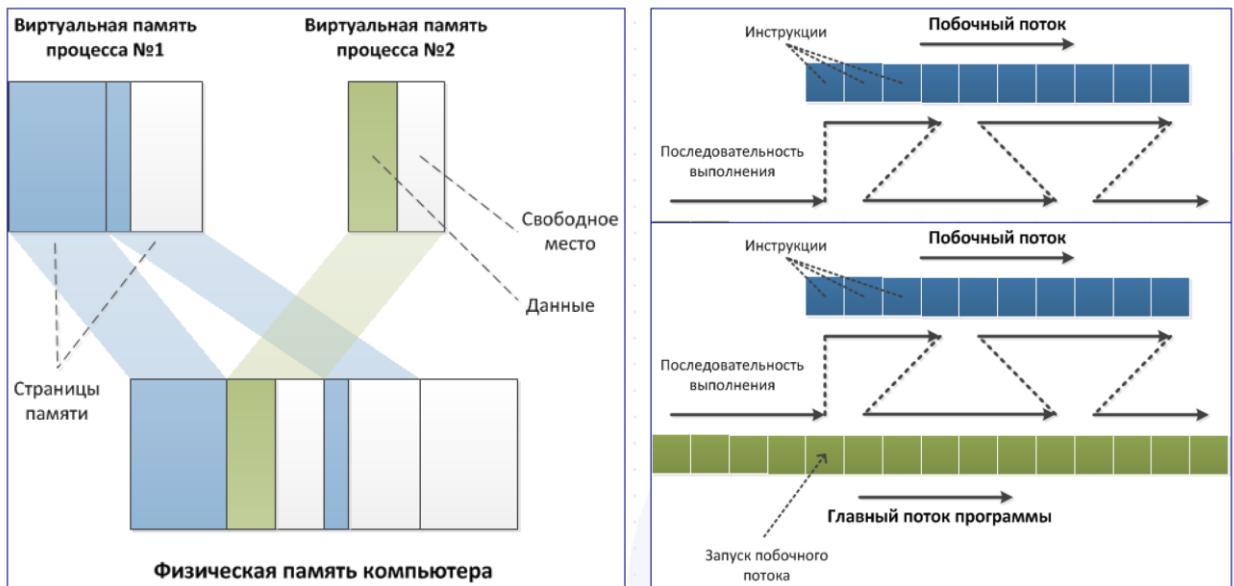


Многопоточность в Java — это одновременное выполнение двух или более потоков для максимального использования центрального процессора (CPU — central processing unit). Каждый поток работает параллельно и не требует отдельной области памяти. К тому же, переключение контекста между потоками занимает меньше времени.



1. Чем процесс отличается от потока?

Процесс — последовательное выполнение программы.

Поток — наименьшее составляющее процесса или легковесный процесс.

Многопоточность — это принцип построения программы, при котором несколько блоков кода могут выполняться одновременно.

Цели:

- **производительность** (разбираем задачу на более мелкие)
- **параллелизм** или concurrency (одновременное выполнение нескольких задач)

На одно ядро процессора, в каждый момент времени, приходится одна единица исполнения. То есть одноядерный процессор может обрабатывать команды только последовательно, по одной за раз (в упрощенном случае).

Однако запуск нескольких параллельных потоков возможен и в системах с одноядерными процессорами. В этом случае система будет периодически переключаться между потоками, поочередно давая выполняться то одному, то другому потоку. Такая схема называется **псевдо-параллелизмом**. Система запоминает состояние (контекст) каждого потока, перед тем как переключиться на другой поток, и восстанавливает его по возвращению к выполнению потока.

Если потоки не синхронизированы, то каждый раз в консоли будет разный результат.

Процессы в Java: определение и функции

- Процесс состоит из кода и данных. Он создается операционной системой при запуске приложения, является достаточно ресурсоемким и обладает собственным виртуальным адресным пространством.
- Процессы работают независимо друг от друга, они не имеют прямого доступа к общим данным в других процессах.
- Операционная система ОС выделяет ресурсы для процесса — **память и время** на выполнение.
- Если один из процессов заблокирован, то ни один другой процесс не может выполняться, пока он не будет разблокирован.

- Для создания нового процесса обычно дублируется родительский процесс.
- Процесс может контролировать дочерние процессы, но не процессы того же уровня.

Что такое потоки

Поток — наименьшее составляющее процесса. Потоки могут выполняться параллельно друг с другом. Их также часто называют легковесными процессами.

Они используют **адресное пространство процесса** и делят его с другими потоками.

Потоки могут контролироваться друг другом и общаться посредством методов Object **wait()**, **notify()**, **notifyAll()**.

При запуске программы операционная система создает **процесс**, загружая в его адресное пространство код и данные программы, а затем запускает главный поток созданного процесса.

2. Чем Thread отличается от Runnable?

Когда нужно использовать Thread, а когда Runnable?

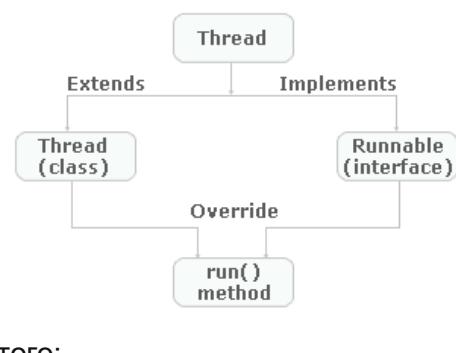
Ответ что тред — это класс, а ранбл интерфейс - считается неполным, нужно рассказать подробно.

Способы запуска потоков

Приложение, создающее **экземпляр класса Thread**, должно предоставить код, который будет работать в этом потоке. Существует два способа, чтобы добиться этого:

Использовать подкласс Thread.

Класс Thread сам реализует Runnable, хотя его метод run не делает ничего. Можно **объявить класс Thread подклассом**, предоставляя собственную реализацию метода run, как в примере справа.



```
public class HelloThread extends Thread {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        (new HelloThread()).start();
    }
}
```

Чтобы породить новый поток нужно:

- 1) Создать объект класса Thread
- 2) Передать в него метод, который нужно выполнить
- 3) Вызвать у созданного объект Thread метод **start()**.

Этот способ больше подходит для простых приложений, но есть условие: **класс задачи должен быть потомком Thread**.

Предоставить реализацию объекта Runnable.

Интерфейс Runnable определяет единственный метод — **run**, который должен содержать код, выполняющийся в потоке. **Объект Runnable передается конструктору Thread**. Например:

```
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }
}
```

Объект может превратить отличный от Thread класс в подкласс.

Этот способ более общий и гибкий и может использоваться для высокоуровневых API управления потоками.

Оба примера вызывают **Thread.start**, чтобы запустить новый поток.

```

    public static void main(String[] args) {
        MyThread1 myThread1 = new MyThread1();
        MyThread1 myThread2 = new MyThread1();
        MyThread2 myThread2 = new MyThread2();
        myThread1.start();
        myThread2.start();
    }
}

class MyThread1 extends Thread{
    public void run(){
        for(int i=1; i<=1000; i++){
            System.out.println(i);
        }
    }
}

class MyThread2 extends Thread{
    public void run(){
        for(int i=1000; i>0; i--){
            System.out.println(i);
        }
    }
}

```

//Создание
class MyThread extends Thread{ public void run() { код } }
//Запуск
new MyThread().start();

//Создание
class MyRunnableImpl implements Runnable{ public void run() { код } }
//Запуск
new Thread(new MyRunnableImpl()).start();

Из за того, что в Java отсутствует множественное наследование,
чаще используют 2-ой вариант.

```

package multithreading;

public class Ex3 {
    public static void main(String[] args) {
        Thread thread1 = new Thread(new MyThread3());
        Thread thread2 = new Thread(new MyThread4());
        thread1.start();
        thread2.start();
    }
}

class MyThread3 implements Runnable {
    public void run() {
        for (int i = 1; i <= 1000; i++) {
            System.out.println(i);
        }
    }
}

class MyThread4 implements Runnable {
    public void run() {
        for (int i = 1000; i > 0; i--) {
            System.out.println(i);
        }
    }
}

public class Ex3 implements Runnable {
    public void run() {
        for (int i = 1; i <= 1000; i++) {
            System.out.println(i);
        }
    }
}

public static void main(String[] args) {
    Thread thread1 = new Thread(new Ex3());
    thread1.start();

    for (int i = 1000; i > 0; i--) {
        System.out.println(i);
    }
}

```

*Как выполнить две задачи параллельно?

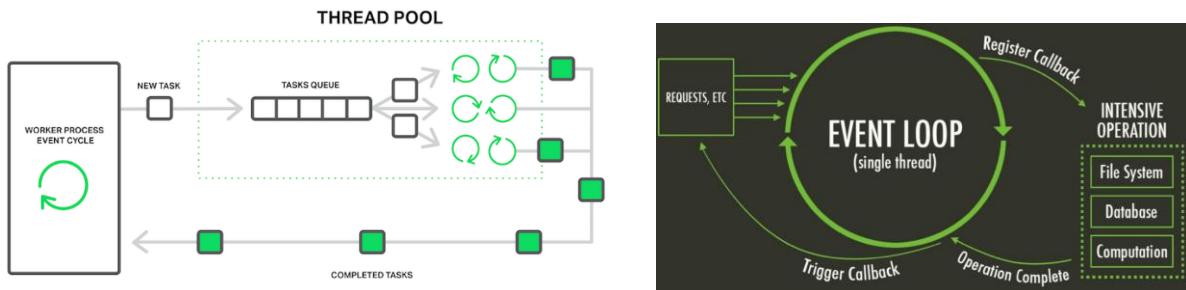
Простейший, путь – явно создать два объекта типа `Thread`, передать им инстансы `Runnable`, с нужными задачами в реализации их методов `run`, и запустить вызвав `thread.start()`. Если в основном потоке нужно дождаться завершения задач, то после `start()` вызывается метод `thread.join()`. Исполнение зависит на вызове этого метода до тех пор, пока тред не закончит свою задачу и не умрет. Вся работа задач с внешними данными должна быть синхронизирована.

Такое ручное создание тредов полезно в учебных целях, но **считается плохой практикой в промышленном коде**: само создание – дорогостоящая операция, а большое количество случайно созданных потоков может приводить к проблеме голодаия потоков (`starvation`).

В качестве **продвинутой альтернативы** используются **пуллы потоков** – реализаций интерфейса `ExecutorService`.

Такие сервисы создаются статическими фабричными методами **класса Executors** (исполнители). Они умеют принимать задачи в виде `Runnable`- или `Callable`-объектов на заранее созданном наборе потоков (собственно, пуле).

Кроме самого пула, экземпляры `ExecutorService` содержат **фабрику потоков** («инструкцию» как создать thread при необходимости), и **коллекцию-очередь задач** на исполнение (см. рисунок).



Thread pool – это множество потоков, каждый из которых предназначен для выполнения той или иной задачи.

В Java с thread pool-ами удобнее всего работать посредством ExecutorService.

Thread pool удобнее всего создавать, используя factory методы класса Executors:
`Executors.newFixedThreadPool(int count)` – создаст pool с 5-ю потоками;
`Executors.newSingleThreadExecutor()` – создаст pool с одним потоком.

Метод `execute` передаёт наше задание (task) в thread pool, где оно выполняется одним из потоков.

После выполнения метода `shutdown` `ExecutorService` понимает, что новых заданий больше не будет и, выполнив поступившие до этого задания, прекращает работу.

Метод `awaitTermination` принуждает поток в котором он вызвался подождать до тех пор, пока не выполнится одно из двух событий: либо `ExecutorService` прекратит свою работу, либо пройдёт время, указанное в параметре метода `awaitTermination`.

```
public class ThreadPoolEx1 {
    public static void main(String[] args) throws InterruptedException {
        ExecutorService executorService = Executors.newFixedThreadPool(5);
        for(int i=0; i<10; i++){
            executorService.execute(new RunnableImpl100());
        }
        executorService.shutdown();
        executorService.awaitTermination(5, TimeUnit.SECONDS);
        System.out.println("Main ends");
    }
}
```

ScheduledExecutorService мы используем тогда, когда хотим установить расписание на запуск потоков из пула.

Данный pool создаётся, используя factory метод класса Executors:

`Executors.newScheduledThreadPool(int count)`

```
scheduledExecutorService.scheduleAtFixedRate(new RunnableImpl200(),
    initialDelay: 3, period: 1, TimeUnit.SECONDS);

Thread.sleep(millis: 20000);
scheduledExecutorService.shutdown();
```

В ответ на передачу на исполнение `Runnable` или `Callable`, сервис возвращает связанный с ним объект типа **Future** – хранилище, которое будет заполнено результатом выполнения задачи в будущем. Даже если никакого результата не ожидается, Future поможет дождаться момента завершения обработки задачи (про это будет отдельный вопрос).

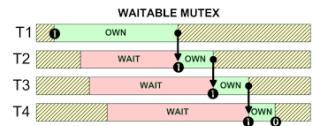
3. Что такое монитор? Как монитор реализован в java?

Монитор – механизм синхронизации потоков, обеспечивающий корректную работу и доступ к неразделяемым ресурсам при синхронизации. Частью монитора является **mutex**, который встроен в класс Object и имеется у каждого объекта.

Важно понимать, что блокировка потока происходит именно на уровне ОБЪЕКТА (mutex).

Удобно представлять mutex как **id захватившего его объекта** (одноместный семафор).

- Если этот id равен 0, то ресурс свободен.
- Если этот id НЕ равен 0, то ресурс занят: встать в очередь и ждать освобождения.



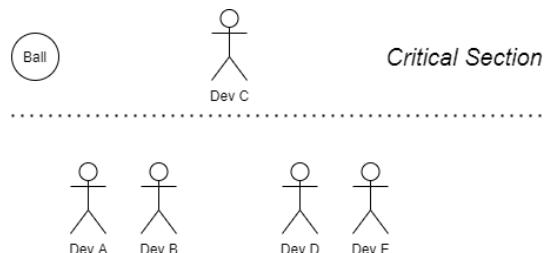
В Java монитор реализован с помощью ключевого слова **synchronized**.

Пример из жизни: <https://www.baeldung.com/cs/what-is-mutex>

В чем проблема, для которой mutexы являются решением?

Представим ежедневную утреннюю встречу Scrum с пятью разработчиками и Scrum Master. Один пьёт кофе, другой хочет закончить работу, трое рады рассказать об успехах и начинают говорить одновременно.

Это заканчивается хаосом, и никто ничего не понимает. Скрим-мастер подходит и дает мяч одному из разработчиков и говорит: «Говорить может только тот, у кого мяч!». С этого момента они поняли друг друга и могли быстро завершить встречу.



4. Что такое синхронизация? Какие способы синхронизации существуют в java?

- ✓ Между синхронизациями по одному объекту установлен полный порядок (total order)
- ✓ Завершение синхронизации (monitorexit) **hb** начало последующей синхронизации по тому же объекту (monitoreenter)

```
public class Ex11 {
    volatile static int counter = 0;
    public static synchronized void increment(){counter++;}
        ← Синхронизация на уровне метода
    public static void main(String[] args) throws InterruptedException {
        Thread thread1 = new Thread(new R());
        Thread thread2 = new Thread(new R());
        thread1.start();
        thread2.start();
        thread1.join();
        thread2.join();

        System.out.println(counter);
    }
}
```

Found duplicate code more... (Ctrl+F1)

Синхронизация только одного блока кода
в параметрах указываем монитор объекта

```
public void doWork1() {
    synchronized(this) {
        Counter2.count++;
        System.out.println(Counter2.count);
    }
}
```

Синхронизация – процесс, который позволяет выполнять **потоки параллельно**.

В Java все объекты имеют блокировку, благодаря которой только один поток одновременно может получить **доступ к критическому коду в объекте**.

Такая синхронизация помогает предотвратить повреждение состояния объекта.

Способы синхронизации в Java:

- Системная синхронизация с использованием **wait()** (ждать), **notify()** (уведомлять)

Поток, который ждет выполнения каких-либо условий, вызывает у этого объекта метод **wait()**, предварительно захватив его монитор. На этом его работа приостанавливается.

Другой поток может вызвать на этом же самом объекте метод **notify()**, предварительно захватив монитор объекта. В результате, **ждущий** на объекте **поток «просыпается»** и продолжает свое выполнение. В обоих случаях монитор надо захватывать в явном виде, через **synchronized-блок**, потому как методы **wait()** и **notify()** не синхронизированы!

- Системная синхронизация с использованием **join()** (присоединяться)

Метод **join()**, вызванный у экземпляра класса **Thread**, позволяет текущему потоку остановиться до того момента, пока другой поток, связанный с этим объектом, закончит свою работу.

Использование классов из пакета **java.util.concurrent.locks** - механизмы синхронизации потоков, альтернативы базовым **synchronized** (**wait**, **notify**, **notifyAll**): **Lock**, **Condition**, **ReadWriteLock**.

5. Как работают методы Object wait(), notify() и notifyAll()?

Методы **wait()**, **notify()**, **notifyAll()** должны вызываться только из синхронизированного кода.

Для извещения потоком других потоков о своих действиях часто используются следующие методы:

wait() – тормозит текущий поток на этом объекте (ждите) и **отпускает его монитор** (уступает дорогу).

notify() – НЕ освобождает монитор и **будит поток**, у которого был ранее вызван метод **wait()**;

notifyAll() – НЕ освобождает монитор и **будит ВСЕ** потоки, у которых ранее был вызван метод **wait()**;



Часто этот вопрос формулируется как **задача Producer-consumer***

https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem

Эту задачу и практические задачи на многопоточность вообще при возможности лучше реализовывать на высокоуровневых примитивах синхронизации. Другой подход – воспользоваться также низкоуровневой, но оптимистической блокировкой на compareAndSet. Но обычно использование **notify/wait** (пессимистическая блокировка) – условие этого задания, то есть требуется реализовать уже существую BlockingQueue.

Эти методы вместе с **synchronized** – самый низкий уровень пессимистических блокировок в Java, использующийся внутри реализации примитивов синхронизации.

Еще с Java 5 в непосредственном использовании этих методов нет необходимости, но теоретические знания всё еще часто спрашивают на интервью.

Чтобы вызывать эти методы у объекта, необходимо чтобы был захвачен его монитор (т.е. нужно быть внутри **synchronized-блока** на этом объекте).

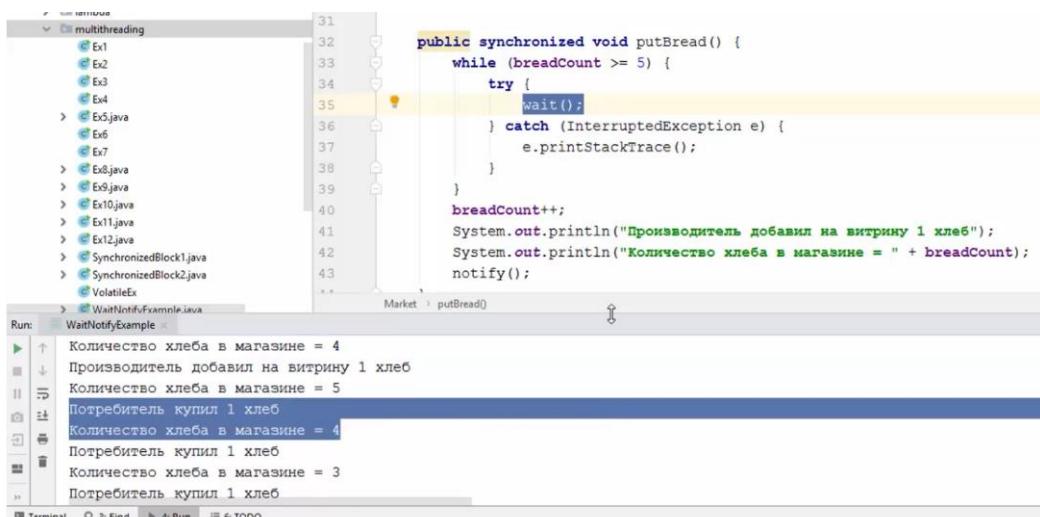
В противном случае будет выброшено `IllegalMonitorStateException`. Так что для полного ответа нужно понимать, как работает **monitor lock** (блок `synchronized`).

Вызов `wait` тормозит текущий поток на ожидание на этом объекте и отпускает его монитор. Исполнение продолжится, когда другой поток вызовет `notify` и отпустит блокировку монитора. Если на объекте ожидают несколько потоков, `notify` разбудит один случайный, `notifyAll` - все сразу.

В теории, ожидание `wait` может быть прервано без вызова `notify`, по желанию JVM (*spurious wakeup*). На практике это бывает крайне редко, но нужно страховаться и после вызова `wait` добавлять дополнительную проверку условия завершения ожидания.

Еще два нештатных случая завершения `wait` – прерывание потока извне и таймаут ожидания. В случае прерывания выбрасывается `InterruptedException`. Для таймаута нужно указать время ожидания параметрами метода `wait`. Значение 0 игнорируется.

```
try {
    wait( timeoutMillis: 1000 );
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

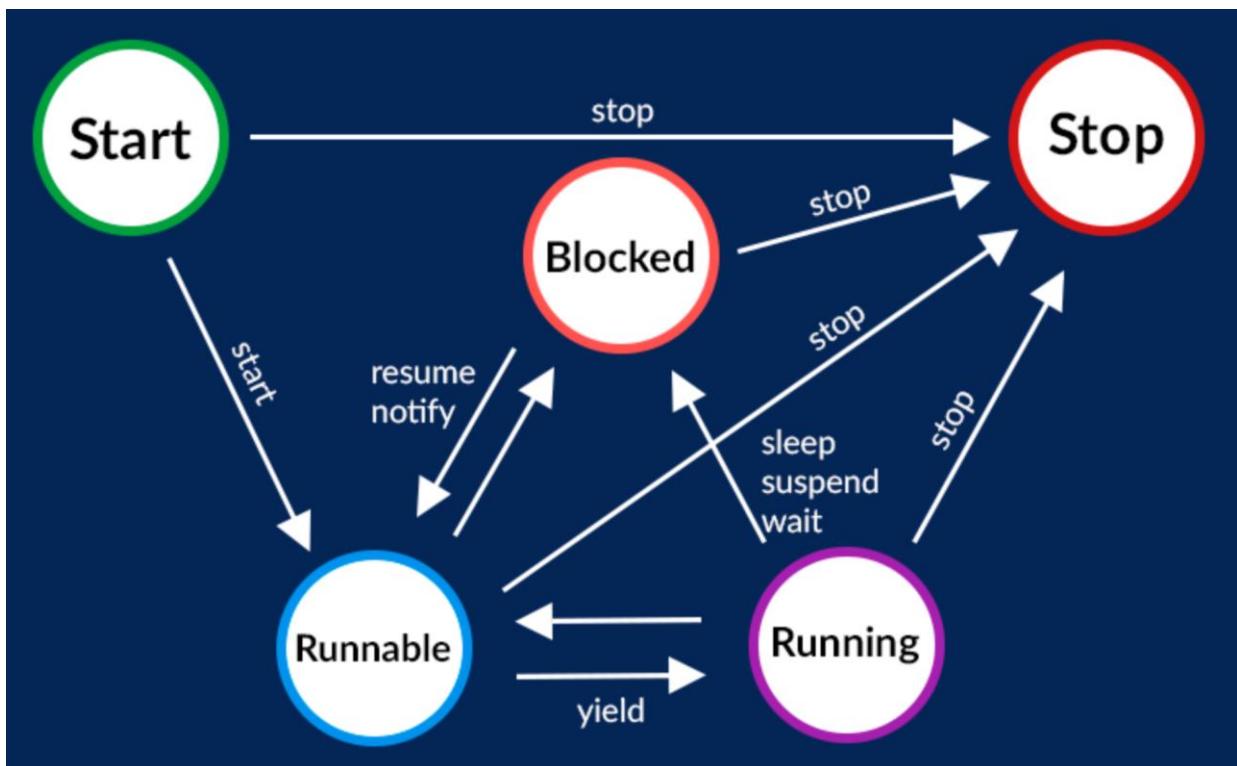


Механизм синхронизации Java-потоков

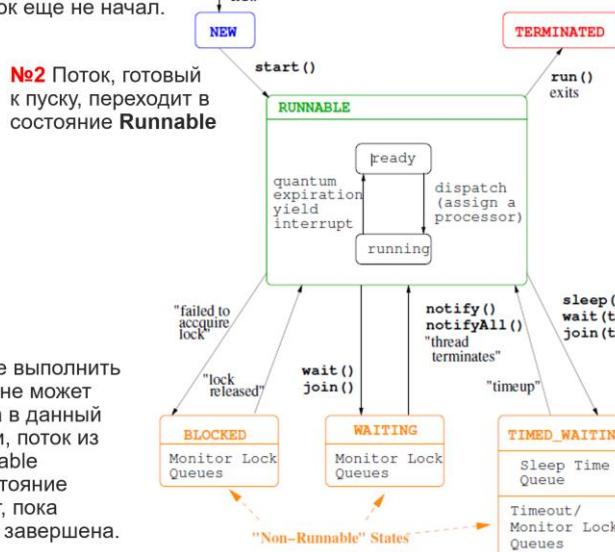


Если НИ ОДИН поток не находится в ожидании на методе `wait()`, то при вызове `notify()` или `notifyAll()` ничего не происходит.

6. В каких состояниях может находиться поток?



№1 Когда создается новый поток, он находится в состоянии **New**, причем запускаться поток еще не начал.



№3 При попытке выполнить задачу, которая не может быть завершена в данный момент времени, поток из состояния **Runnable** переходит в состояние **Blocked**. И ждет, пока задача не будет завершена.

№6 Поток завершается по любой из следующих причин:
1) в обычном режиме (программа выполнена)
2) нештатное событие

№5 Поток находится в состоянии **Runnable**. Теперь он вызывает метод `sleep(t)`, `wait(t)` или `join(t)` с неким промежутком времени в качестве параметра и переходит в состояние **Timed waiting**.

№4 Когда поток находится в состоянии **Waiting**, он ждет другой поток, связанный условием. Когда это условие выполняется, планировщик получает уведомление и вызываются методы `notify()` или `notifyAll()`. В этом случае ожидающий поток переходит в состояние **Runnable**.

Жизненный цикл потока:

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

- **New** – поток создан, но еще не запущен
- **Runnable** – поток выполняется методы: `start()`, принудительное продолжение `notify()`, `notifyAll()`.
- **Blocked** – поток блокирован (состояние возникает при синхронизации: вызов методов помеченных, как `synchronized`)

- **Waiting** – поток ждёт окончания работы другого потока или принудительного продолжения. Методы: `join()`, `wait()`, `suspend()` (`deprecated`).
- **Timed_waiting** – поток некоторое время ждёт окончания другого потока. Методы: `yield()`, `sleep(long mills)`, `join(long timeout)` и `wait(long timeout)`.
- **Terminated** – поток завершён. Методы: `interrupt()`, `stop()` (`deprecated`) или нормальное завершение метода `run()`.
- **Dead** — после того, как поток завершил свое выполнение, его состояние меняется на `dead`, то есть он завершает свой жизненный цикл.

Получить текущее значение состояния потока можно через `getState()`.

***Задача про обедающих философов** — классический пример, используемый в информатике для иллюстрации проблем синхронизации при разработке параллельных алгоритмов и техник решения этих проблем.

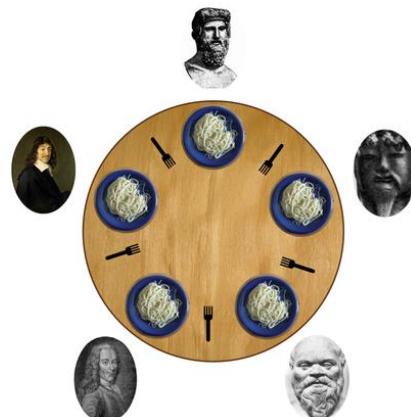
Задача была сформулирована в 1965 году Эдсгером Дейкстрой как экзаменационное упражнение для студентов. В качестве примера был взят конкурирующий доступ к ленточному накопителю. Задача была сформулирована Энтони Хоаром в том виде, в котором она известна сегодня.

Пять безмолвных философов сидят вокруг круглого стола, перед каждым философом стоит тарелка спагетти. Вилки лежат на столе между каждой парой ближайших философов.

Каждый философ может **либо есть, либо размышлять**. Приём пищи не ограничен количеством оставшихся спагетти — подразумевается бесконечный запас. Тем не менее, философ может есть только тогда, когда держит две вилки — взятую справа и слева (альтернативная формулировка проблемы подразумевает миски с рисом и палочки для еды вместо тарелок со спагетти и вилок).

Каждый философ может взять ближайшую вилку (если она доступна) или положить — если он уже держит её. Взятие каждой вилки и возвращение её на стол являются раздельными действиями, которые должны выполняться одно за другим.

Вопрос задачи заключается в том, чтобы разработать модель поведения (параллельный алгоритм), при котором ни один из философов не будет голодать, то есть будет вечно чередовать приём пищи и размышлений.



Проблема: Задача сформулирована таким образом, чтобы иллюстрировать проблему избежания взаимной блокировки (deadlock) — состояния системы, при котором прогресс невозможен.

Варианты решения: официант (семафор), иерархия ресурсов (приоритизация), взаимоисключающая блокировка (мьютекс).

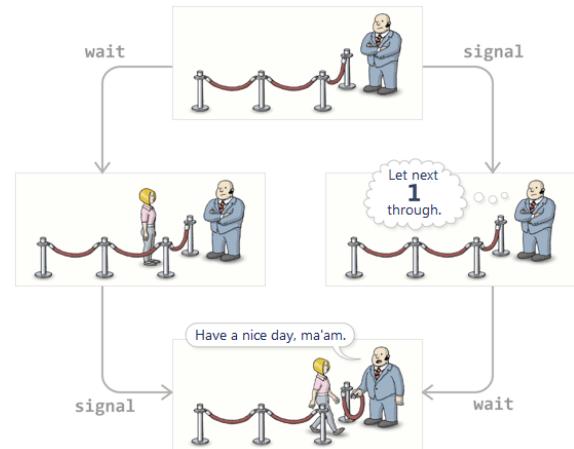
Загуглить или прочитать Wiki =)

7. Что такое семафор? Как он реализован в Java?

Короткий ответ: Semaphore – это новый тип синхронизатора: **семафор со счётчиком**, реализующий шаблон синхронизации Семафор.

Доступ управляется с помощью счётчика:
изначальное значение счетчика задается в конструкторе при создании синхронизатора,
когда поток заходит в заданный блок кода, то
значение счетчика уменьшается на единицу,
когда поток его покидает, то увеличивается.

Если значение счетчика равно нулю, то текущий поток блокируется, пока кто-нибудь не выйдет из защищаемого блока. **Semaphore используется для защиты дорогих ресурсов**, которые доступны в ограниченном количестве, например подключение к базе данных в пуле.



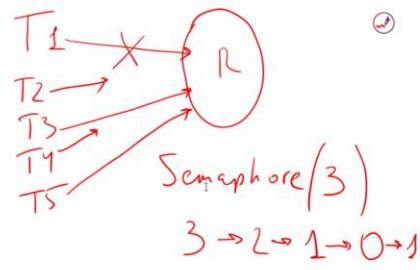
Подробнее. Поток должен ждать. Ждать до тех пор, пока не удастся получить эксклюзивный доступ к ресурсу или пока не появятся задачи для исполнения. Один из механизмов ожидания, при котором поток не ставится на выполнение планировщиком ядра ОС, реализуется при помощи **семафора**.

Семафор – один из старейших примитивов синхронизации. Он был изобретен Эдсгером Дейкстра в 1968 году. По большому счету это счетчик, который можно увеличивать и уменьшать из разных потоков.

Уменьшение до 0 блокирует уменьшающий поток. Состояние, когда счетчик больше нуля называют *сигнальное состояние*, операцию его увеличения – *release* (освобождение) или *signal*, уменьшения – *acquire* (захват) или *wait* (ожидание).

На практике можно представить, что **release** – выделение квоты доступа к критической секции программы, а **acquire** – использование необходимого объема доступной квоты, или ожидание, если её не хватает. Подробнее с деталями работы семафора поможет ознакомиться перевод статьи с картинками на хабре. <https://habr.com/ru/post/261273/>

Semaphore – это синхронизатор, позволяющий ограничить доступ к какому-то ресурсу. В конструктор Semaphore нужно передавать количество потоков, которым Semaphore будет разрешать одновременно использовать этот ресурс.



Есть ресурс R, создадим Семафор. В параметрах указываем число – сколько потоков может использовать этот ресурс. Например, три. Это счётчик. Семафор будет использовать этот счётчик, чтобы гарантировать, что не более трёх потоков одновременно пользуются ресурсом.

Допустим, пять потоков пробуют получить доступ к ресурсу (T1, T2, T3, T4, T5). **А доступа всего три.** Когда T1 запрашивает доступ, семафор смотрит $3 > 0$? Да! Доступ открыт и счётчик = 2. Когда T3 запрашивает доступ, то $2 > 0$? Да! Доступ открыт и т.д. для потока T5. А вот когда T2 или T4 запросят доступ, то $0 > 0$? Нет! Они залочатся и будут ждать, когда счётчик станет > 0 и доступ будет открыт.

В Java семафор реализован классом `Semaphore`. Состоит этот класс в основном из разных форм методов `acquire` (с таймаутом, с игнорированием `InterruptedException`, неблокирующий) и `release`. Методы могут принимать параметр `permits` (разрешения) – тот самый объем квот, которые необходимо освободить/захватить.

Несколько вспомогательных методов позволяют узнать больше о количестве и составе очереди потоков, которые ждут освобождения пермитов. А методы `availablePermits` и `drainPermits` позволяют узнать количество оставшихся пермитов, и захватить их все соответственно. В конструкторе конфигурируются изначальное количество пермитов, и свойство `fair` (аналогичное свойству `ReentrantLock`).

***Вопрос.** Что будет если счётчик Семафора равен 1?

Ответ. Тогда алгоритм работы Семафора будет такой же, как у Lock (открыто/закрыто).

Пример очереди к телефонной будке: 2 кабины, пять человек (Трёхулов 3.):

```
class Person extends Thread{
    String name;
    private Semaphore callBox;
    public Person(String name, Semaphore callBox){
        this.name=name;
        this.callBox=callBox;
    }
    public void run(){
        try {
            System.out.println(name + " ждёт...");
            callBox.acquire();
            System.out.println(name + " пользуется телефоном");
            sleep(2000);
            System.out.println(name + " завёшил(а) звонок");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        finally {
            callBox.release();
        }
    }
}
```

Viktor ждёт...
Elena ждёт...
Zaur ждёт...
Zaur пользуется телефоном
Viktor пользуется телефоном
[Zaur завёшил(а) звонок]
Elena пользуется телефоном
Viktor завёшил(а) звонок
Marina пользуется телефоном
Elena завёшил(а) звонок
Oleg пользуется телефоном
Marina завёшил(а) звонок
Oleg завёшил(а) звонок



8. Что обозначает ключевое слово `volatile`? Почему операции над `volatile` переменными не атомарны?

Процессор считывает данные из памяти, изменяет и записывает их обратно в память. Чтобы ускорить работу процессора в него встроили свою «быструю» память – кэш.

Чтобы ускорить свою работу, процессор копирует самые часто используемые переменные из области памяти в свой кэш и все изменения с ними производит в этой быстрой памяти. А после – копирует обратно в «медленную» память. Медленная память все это время содержит старые(!) (неизмененные) значения переменных.

И тогда может возникнуть проблема. Одна нить меняет переменную, а вторая нить «не видит» этого изменения, т.к. оно было совершено в быстрой памяти. Это следствие того, что нити не имеют доступа к кэшу друг друга. (Процессор часто содержит несколько независимых ядер и нити физически могут исполняться на разных ядрах.)

Было придумано специальное ключевое слово `volatile`. Помещение его перед определением переменной принудительно всегда читало и писало эту переменную только в обычную (медленную) память.

Переменная `volatile` находится в хипе, а не в кэше стека!

`Volatile` (eng. изменчивый, непостоянный). При создании многопоточных приложений мы можем столкнуться с двумя серьезными проблемами. Обе проблемы решаются с помощью всего одного ключевого слова — `volatile`.

Что за проблемы?

Во-первых, 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.

См. дальше Java Memory Model (JMM)

Во-вторых, в Java операции чтения и записи полей всех типов, кроме long и double, являются атомарными (состоят из одной части).

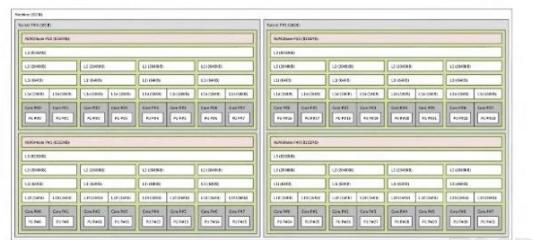
✓ Операция атомарна, если невозможно наблюдать частичный результат её выполнения. Любой наблюдатель видит либо состояние системы до атомарной операции, либо после.

✓ В Java:

- ✓ Запись в поле типа boolean, byte, short, char, int, Object всегда атомарна
- ✓ Запись в поле типа long/double: атомарна запись старших и младших 32 бит

✓ Запись в поле типа long/double, объявленное volatile, атомарна

Иерархия кэшей (AMD Bulldozer)



```
public class Main {
    public volatile long x = 2222222222222222L;
    public static void main(String[] args) {
    }
}
```

Например, если в одном потоке меняешь значение переменной int, а в другом потоке читаешь значение этой переменной, то получишь либо ее старое значение, либо новое — то, которое получилось после изменения в потоке 1. Никаких «промежуточных вариантов» там появиться не может. Однако с **long и double** это **не работает (64 бита)**.

Почему? Из-за кроссплатформенности! <https://youtu.be/ShzQJUFzq58?t=3014>

long и double — самые «тяжеловесные» примитивы в Java: они весят по **64 бита**. И в некоторых 32-битных платформах просто не реализована атомарность чтения и записи 64-битных переменных.

Такие переменные читаются и записываются в две операции. Сначала в переменную записываются первые 32 бита, потом еще 32. Соответственно, в этих случаях может возникнуть проблема.

Один поток записывает какое-то 64-битное значение в переменную X, и делает он это «в два захода». В то же время второй поток пытается прочитать значение этой переменной, причем делает это как раз посередине, когда первые 32 бита уже записаны, а вторые — еще нет. В результате он читает промежуточное, некорректное значение, и получается ошибка.

Неатомарная запись

✓ `this.obj = new Foo();` // obj = 0x0000_0000_0000_0000

0123_4567 89AB_CDEF

✓ А в другом потоке на другом ядре: `this.obj.bar = 5;`

Уже было в L1:

0000_0000

0123_4567 0000_0000

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

- Переменная всегда будет атомарно читаться и записываться (даже если это 64-битные double или long)
- Java-машина НЕ будет помещать ее в кэш. Так что ситуация, когда 10 потоков работают со своими локальными копиями исключена.

Для синхронизации значения переменной между потоками ключевое слово **volatile** используется тогда, когда **только ОДИН поток может изменять значение этой переменной, а остальные потоки могут его только читать**.

*Расскажи про правила «happens-before» и Java Memory Model (JMM).

В Java основную часть работы по выделению времени и ресурсов потокам для выполнения их задач выполняет **планировщик потоков**, т.к. потоки выполняются в произвольном порядке, и чаще всего предсказать порядок невозможno.

Видимость (visibility)

- ✓ Результат операции write X, выполненной в потоке A, виден в операции read X, выполненной в потоке B
- ✓ Видимость определена только для конкретных потоков A и B, нет «глобальной видимости».

Порядок (ordering)

- ✓ A happens before B (A hb B), если все записи, выполненные до точки A (включительно), видны в любой операции чтения после точки B (включительно)
- ✓ A hb B, B hb C \Rightarrow A hb C

Простые правила happens before

Tagir Valeev

- ✓ Для двух операций A и B в одном потоке **A hb B**, если A раньше B в тексте программы (program order).
- ✓ Завершение конструктора объекта X **hb** начало finalize X
- ✓ Вызов thread.start() **hb** первое действие в потоке thread
- ✓ Последнее действие в потоке thread **hb** thread.join()
- ✓ Инициализация объекта по умолчанию **hb** любое другое действие

Java Memory Model

- ✓ Часть спецификации языка Java (JLS 17.4)
- ✓ Описывает взаимодействие приложения с памятью
- ✓ Даёт определённые гарантии относительно того, какие записи в память когда и как могут быть видимы
- ✓ Не зависит от реализации JVM, операционной системы и железа

Описывает, **КАК потоки должны взаимодействовать через общую память**, определяет набор действий меж поточного взаимодействия, например:

- чтение и запись переменной
- захват и освобождение монитора
- чтение и запись volatile переменной
- запуск нового потока.

JMM определяет отношение между этими действиями "happens-before" - абстракцией обозначающей, что если операция X связана отношением happens-before с операцией Y, то весь код следуемый за операцией Y, выполняемый в одном потоке, видит все изменения, сделанные другим потоком, до операции X.

<https://habr.com/ru/company/golovachcourses/blog/221133/>

Гибкость является особенностью дизайна — давая компилятору, среде исполнения и аппаратному обеспечению **гибкость выполнять операции в оптимальном порядке в рамках модели памяти**, мы можем достичь более высокой производительности.

Можно выделить несколько основных областей, имеющих отношение к модели памяти:

Synchronized (синхронизация)

При входе в synchronized метод или блок поток **обновляет содержимое локальной памяти**, а при выходе из synchronized метода или блока **поток записывает изменения**, сделанные в локальной памяти, в главную. Такое поведение synchronized методов и блоков следует из правил для отношения «happens-before».

Visibility (видимость)

Один поток может **временно сохранить** значения некоторых полей не в основную память, а в **регистры или локальный кэш** процессора, таким образом **второй поток**, читая из основной памяти, **может не увидеть** последних **изменений** поля. И наоборот, если поток на протяжении какого-то времени работает с регистрами и локальными кэшами, читая данные оттуда, он может сразу не увидеть изменений, сделанных другим потоком в основную память.

К вопросу видимости имеют отношение следующие ключевые слова языка Java: **synchronized**, **volatile**, **final**.

С точки зрения Java **все переменные** (за исключением локальных переменных, объявленных внутри метода) **хранятся в heap памяти, которая доступна всем потокам**. Кроме этого, каждый поток имеет локальную (рабочую) память, где он хранит копии переменных, с которыми он работает, и при выполнении программы поток работает только с этими копиями.

Volatile (волатильность, изменчивость)

Запись volatile-переменных производится в основную память, минуя локальную. Чтение volatile переменной производится также из основной памяти, то есть значение переменной не может сохраняться в регистрах или локальной памяти потока, и **операция чтения** этой переменной **гарантированно вернёт** последнее записанное в ней **значение**.

Final (финализация)

После того как **объект был корректно создан**, любой **поток** может **видеть значения его final полей без дополнительной синхронизации**. «Корректно создан» означает, что ссылка на создающийся объект не должна использоваться до тех пор, пока не завершился конструктор объекта.

Рекомендуется изменять final поля объекта только внутри конструктора, в противном случае поведение не специфицировано.

Reordering (переупорядочивание)

Для увеличения производительности процессор/компилятор могут переставлять местами некоторые инструкции/операции. **Процессор может решить поменять порядок выполнения операций**, если, например, сочтет что такая последовательность выполнится быстрее. Эффект может наблюдаться, когда один поток кладет результаты первой операции в регистр или локальный кэш, а результат второй операции попадает непосредственно в основную память. Тогда второй поток, обращаясь к основной памяти может сначала увидеть результат второй операции, и только потом первой, когда все регистры или кэши синхронизируются с основной памятью.

Также регулируется набором правил «happens-before»: операции чтения и записи volatile переменных не могут быть переупорядочены с операциями чтения и записи других volatile и не-volatile переменных.

9. Для чего нужны Atomic типы данных? Чем отличаются от volatile?

Начнем с того, что такое атомики и зачем нужны.

Atomic* – семейство классов из `java.util.concurrent`. Они предоставляют набор атомарных операций для соответствующих типов.

Например, с помощью методов `getAndIncrement` и `incrementAndGet` класса `AtomicInteger` можно делать ЦЕЛОСТНЫМ неатомарный в обычных условиях инкремент

The screenshot shows the IntelliJ IDEA interface with the code editor open. The code is as follows:

```
package multithreading;
import java.util.concurrent.atomic.AtomicInteger;
public class AtomicIntegerEx {
    // static int counter = 0;
    static AtomicInteger counter = new AtomicInteger( initialValue: 0 );
    public static void increment() {
        // counter++;
        counter.incrementAndGet();
    }
}
```

The code editor highlights the `increment` method. Below the code editor is the run configuration panel:

Run:	AtomicIntegerEx
"C:\Program Files\Java\jdk-11.0.3\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2020.3.3\lib\idea_rt.jar=5309,C:\Program Files\JetBrains\IntelliJ IDEA 2020.3.3\bin"	
200	
Process finished with exit code 0	

Условно можно разделить подходы реализации большинства atomic-методов на две группы: **compare-and-set** и **set-and-get**.

Методы категории **compare-and-set** принимают старое значение и новое. *Если переданное старое значение совпало с текущим, устанавливается новое.*

Обычно делегируют вызов в методы класса `Unsafe`, которые заменяются нативными реализациями виртуальной машины. Виртуальная машина в большинстве случаев использует атомарную операцию процессора [compare-and-swap](#) (CAS).

Поэтому атомики обычно более эффективны чем стандартная дорогостоящая блокировка. **Все операции, которые нужны, происходят атомарно.**

В случае **set-and-get** старое значение неизвестно. Поэтому нужен небольшой трюк: программа сначала считывает текущее значение, а затем записывает новое, тоже с помощью CAS, потому что запись могла успеть поменяться даже за этот шаг. Эта попытка чтения + записи повторяется в цикле, пока старое значение не совпадет и переменная не будет успешно записана.

Этот трюк называется [double-checked или optimistic locking](#), и может быть использован и в пользовательском коде с любым способом синхронизации. Оптимистичность заключается в том, что мы надеемся что [сстояния гонки](#) нет, прибегая к синхронизации только если гонка всё же случилась. Реализация оптимистичной блокировки может быть дана как отдельная задача.

AtomicInteger

AtomicInteger – это класс, который предоставляет возможность работать с целочисленным значением int, используя атомарные операции.

incrementAndGet

getAndIncrement

addAndGet

getAndAdd

decrementAndGet

getAndDecrement

```

1 public class Adder extends Thread {
2
3     static Integer SHARED_SUB_TOTAL=0;
4
5     private Integer valueToAdd;
6
7     public Adder(Integer value) {
8         valueToAdd=value;
9     }
10
11    @Override
12    public void run() {
13        Integer previousTotal = SHARED_SUB_TOTAL;
14        Integer newTotal = previousTotal + valueToAdd;
15        SHARED_SUB_TOTAL=newTotal;
16    }
17
18 }
19
20 Adder thread1 = new Adder(10);
21 Adder thread2 = new Adder(4);
22
23 thread1.start();
24 thread2.start();

```

ШАГИ 1-3

Стартуем с ПРОМЕЖУТОЧНОГО ИТОГО, равного 0, которое читают оба потока.

ШАГ 4

Поток 2 устанавливает значение SUBTOTAL равным 4 после добавления своего вклада.

ШАГ 5:

Поток 1, не зная об изменении потока 2, записывает обратно значение 10 после прибавления своего вклада к исходному значению, равному нулю. Он теряет изменения, сделанные потоком 2.

1 First construct two threads

2 Call start() on both threads

3 Both threads read the SUBTOTAL of 0

4 Thread 2 sets the value of SUBTOTAL to 0+4

5 Thread 1 *incorrectly* sets the value of SUBTOTAL to 0 + 10

6 The main thread only reads the last value set

Thread 1**Thread 2****Main Thread****ШАГ 6:**

ПРОМЕЖУТОЧНЫЙ ИТОГ теперь установлен на 10 и является неправильным! Итого должно быть 14.

<https://openclassrooms.com/en/courses/5684021-scale-up-your-code-with-java-concurrency/6600431-combat-shared-mutability-using-atomic-variables>

Семейство Atomics

AtomicBoolean, AtomicInteger, AtomicLong:

- Операции с этими классами работают быстрее, чем если синхронизироваться через synchronized/volatile;
- Существуют методы для атомарного добавления на заданную величину, а также инкремент/декремент.

AtomicInteger, **AtomicLong** – классы с массивами

AtomicReference – класс для атомарных операций со ссылкой на объект.

AtomicMarkableReference – класс для атомарных операций со следующей парой полей: ссылка на объект и битовый флаг true/false.

AtomicStampedReference – класс для атомарных операций со следующей парой полей: ссылка на объект и int значение.

AtomicReferenceArray – массив ссылок на объекты, которые могут атомарно обновляться.

Примеры и устройство Atomics:

```

public class th3_01_Atomics {

    private AtomicInteger atomicCount = new AtomicInteger(0);

    // целостная ссылка на что-то
    private final AtomicReference<Object> init = new AtomicReference<>();
    private Object someObject=new Object();

    void run() {
        Object o1=new Object();
        if (!init.compareAndSet(someObject, o1)) {
            throw new IllegalStateException("Already initialized");
        }
        Object prev=init.getAndSet(o1);
        init.set(someObject);
        int prevCount= atomicCount.getAndAdd(
    }
}

public final int getAndAdd(int delta) {
    return unsafe.getAndAddInt(this, valueOffset, delta);
}

public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        var5 = this.getIntVolatile(var1, var2);
    } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));

    return var5;
}

```

10. Что такое потоки демоны? Для чего они нужны? Как создать поток-демон?

В Java процесс завершается тогда, когда завершаются все его основные и дочерние потоки.

Потоки-демоны — это низкоприоритетные потоки, работающие в фоновом режиме для выполнения таких задач, как **сбор «мусора»** (освобождают память неиспользованных объектов и очищают кэш).

Большинство потоков JVM (Java Virtual Machine) являются потоками-демонами.

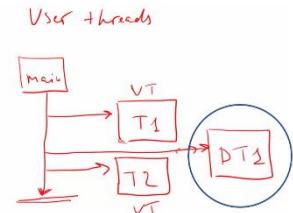
Свойства потоков-демонов:

- Не влияют на закрытие JVM, когда все пользовательские потоки завершили свое исполнение;
- JVM сама закрывается, когда все пользовательские потоки перестают выполняться;
- Если JVM обнаружит работающий поток-демон, она завершит его, после чего закроется. JVM не учитывает, работает поток или нет.

Чтобы установить, является ли поток демоном, используется метод **boolean isDaemon()**. Если да, то он возвращает значение true, если нет, то — то значение false.

Daemon потоки предназначены для выполнение фоновых задач и оказания различных сервисов User потокам. При завершении работы последнего User потока программа завершает своё выполнение, не дожидаясь окончания Daemon потоков.

Потоки не-демоны называются пользовательскими (user thread).



Метод **setDaemon()** вызывается **ДО запуска потока**, до того, как поток будет запущен. В параметры передаём значение true. Так объявляем поток Демон потоком.

Если мы запустили поток и только потом создаём Демона, то будет выброшено исключение **IllegalThreadStateException**.

Метод **isDaemon()** – проверяем, является ли поток Демон-потоком.

Демон в широком значении – фоновая программа. В Java потоки-демоны имеют схожий смысл: это потоки для фоновых действий по обслуживанию основных потоков.

Тред создается демоном, если его родитель демон. Свойство Java-треда **isDaemon** можно переключать в любой момент до старта потока.

По сравнению с пользовательскими потоками демоны имеют **меньший приоритет выполнения**.

Когда все *пользовательские* треды завершились, JVM завершает работу. Демоны не выполняют самостоятельных задач, поэтому не препятствуют остановке, программа завершается, не дожидаясь окончания их работы.

Daemon thread может быть полезен для таких действий, как инвалидация кэша, периодическая актуализация значений из внешних источников, освобождение неиспользуемых пользовательских ресурсов.

[11. Что такое приоритет потока? На что он влияет? Какой приоритет у потоков по умолчанию?](#)

Каждому потоку исполнения в Java присваивается свой приоритет, который определяет поведение данного потока по отношению к другим потокам.

Приоритеты потоков исполнения задаются целыми числами (обычно от 1 до 10), определяющими относительный приоритет одного потока над другими.

Приоритет потока исполнения используется для принятия решения при переходе от одного потока исполнения к другому. Это так называемое переключение контекста. Задается с помощью метода **public final void setPriority(int newPriority)**.

По умолчанию приоритет потока = 5.

```
public class DaemonExample {
    public static void main(String[] args) {
        System.out.println("Main thread starts");
        UserThread userThread = new UserThread();
        userThread.setName("user_thread");
        DaemonThread daemonThread = new DaemonThread();
        daemonThread.setName("daemon_thread");
        daemonThread.setDaemon(true);
        userThread.start();
        daemonThread.start();
        System.out.println("Main thread ends");
    }
}
```

```

package multithreading;

public class Ex5 {
    public static void main(String[] args) {
        MyThread5 myThread5 = new MyThread5();
        System.out.println("Name of myThread5 = " + myThread5.getName() +
                           " Priority of myThread5 = " + myThread5.getPriority());
        MyThread5 myThread6 = new MyThread5();
        System.out.println("Name of myThread6 = " + myThread6.getName() +
                           " Priority of myThread6 = " + myThread6.getPriority());
    }
}

```

Run: Ex5

```

"C:\Program Files\Java\jdk-11.0.3\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2018.3.3\lib\idea_rt.jar=5334,C:\Program Files\JetBrains\IntelliJ IDEA 2018.3.3"
Name of myThread5 = Thread-0 Priority of myThread5 = 5 ← Приоритет по дефолту
Name of myThread6 = Thread-1 Priority of myThread6 = 5

Process finished with exit code 0

```

Существуют следующие константы для определения приоритета потока:

- Thread.MIN_PRIORITY (1)
- Thread.NORM_PRIORITY (5)
- Thread.MAX_PRIORITY (10)

```

myThread5.setPriority(Thread.MIN_PRIORITY); I
System.out.println("Name of myThread5 = " + myThread5.getName() + " Priority of myThread5 = " + myThread5.getPriority());

```

НЕ полагайтесь на приоритет потоков при проектировании многопоточных приложений!
Скорее всего планировщик потоков будет использовать приоритеты при выборе следующего потока на выполнение, но это **НЕ гарантируется**.

```

package multithreading;

public class Ex6 implements Runnable {
    public void run() {
        System.out.println("Method run. Thread name = " +
                           Thread.currentThread().getName());
    }
}

public static void main(String[] args) {
    Thread thread = new Thread(new Ex6()); создали...
    thread.start();     и запустили поток
    System.out.println("Method main. Thread name = " +
                       Thread.currentThread().getName());
}

```

Run: Ex6

```

"C:\Program Files\Java\jdk-11.0.3\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2018.3.3\lib\idea_rt.jar=5334,C:\Program Files\JetBrains\IntelliJ IDEA 2018.3.3"
Method main. Thread name = main
Method run. Thread name = Thread-0

```

При запуске вызывается метод run(), где мы выводим инфо о текущем потоке в методе currentThread().
И Java назвала этот поток Thread-0

12. Как работает Thread.join()? Для чего он нужен?

Если в основном потоке нужно дождаться завершения задач – после `start()`, вызывается метод `thread.join()` (присоединиться). Исполнение зависнет на вызове этого метода до тех пор, пока тред не закончит свою задачу и не умрет.

Одна нить может вызывать метод `join()` у объекта второй нити. В результате первая нить (которая вызвала метод) приостанавливает свою работу до окончания работы второй нити (у объекта которой был вызван метод).

Тут стоит различать две вещи: есть, собственно, нить – отдельный процесс выполнения команд, а есть объект этой нити (объект Thread).

```
for (Thread thread : threads) {
    thread.join(); // дождаться завершения thread
}
```

```
public class Ex9 {
    public static void main(String[] args) throws InterruptedException {
        System.out.println("Method main begins");
        Thread thread = new Thread(new Worker());
        thread.start();
        thread.join( millis: 1500 );
        System.out.println("Method main ends");
    }
}
```

Метод можно использовать с параметрами (пример выше), принимает мили секунды.

Тогда поток, в котором вызван метод **join()** (здесь в main потоке), будет ждать пока тред (**thread.join**) не завершит свою работу или не пройдёт период 1,5 сек. Какое из этих событий случится первым, то и подстегнёт поток main продолжить свою работу.

Как работает Join:

```
public class Threads_04_Join {
    static class JoinThread extends Thread {
        public JoinThread (String name) {
            super(name);
        }
        public void run() {
            String nameT = getName();
            long timeout = 0;
            System.out.println("Старт потока " + nameT);
            try {
                switch (nameT) {
                    case "First":
                        timeout = 5_000;
                        break;
                    case "Second":
                        timeout = 1_000;
                }
                Thread.sleep(timeout);
                System.out.println("завершение потока " + nameT);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public static void main(String[ ] args) {
    System.out.println("Старт потока main");
    JoinThread t1 = new JoinThread("First");
    JoinThread t2 = new JoinThread("Second");
    t1.start();
    t2.start();
    try {
        t1.join(); // поток main остановлен до окончания работы потока t1
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println(Thread.currentThread().getName()); // имя текущего потока
}
```

D:\Java\jdk1.8.0_73\bin\java ...
 Старт потока main
 Старт потока First
 Старт потока Second
 завершение потока Second
 завершение потока First
 main
 Process finished with exit code 0

13. Чем отличаются методы yield() и sleep()?

Короткий ответ:

- **sleep(500)** – текущая нить «засыпает», то есть приостанавливает свою работу на 500 миллисекунд – 0.5 секунды
- **yield()** – текущая нить «пропускает свой ход». Из состояния running переходит в состояние ready, а Java-машина приступает к выполнению следующей нити.

Код	Описание
public static void main(String[] args){	Программа запустится.
Thread.sleep(2000); }	Затем замрет на 2 секунды (2 000 миллисекунд) Затем завершится.

Подробнее. Метод **sleep()** останавливает работу потока на заданное количество времени.

В этом примере **почему слово «конец» выведено вначале?** Потому что при запуске новых потоков, поток `main` их запустил. Далее эти потоки «ответвились» от него (работают независимо), а `main` продолжает свою работу. Он запустил потоки и работает дальше – выводит слово «конец» в консоль. Он не спит, поэтому работает быстро. А через секунду оба потока начинают поочерёдно выводить цифры в консоль и продолжают дальше с интервалом в 1 секунду.

Пока эти потоки не прекратят работу, программа не завершится. Исключение **InterruptedException** нужно обработать в `try-catch` блоке, т.к. поток может ПРЕРВАТЬ другой поток. Если поток просит остановиться, а мы в этот момент находимся в «спячке», то будет выброшен **InterruptedException** Exc.

```

public class Ex8 extends Thread {
    public void run() {
        for (int i = 1; i <= 10; i++) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName() + " " + i);
        }
    }
}

```

Ex8

"C:\Program Files\Java\jdk-11.0.3\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2018.3.3\lib\idea_rt.jar=5334,C:\Program Files\JetBrains\IntelliJ IDEA 2018.3.3\bin" -Dfile.encoding=UTF-8

Konec!

Thread-0 1

Thread-1 1

Thread-1 2

Thread-0 2

Thread-1 3

Thread-0 3

Когда **это** время истекает – процессор переключается на другую нить и начинает выполнять ее команды.

Вызов **метода Thread.yield()** позволяет досрочно завершить квант времени текущей нити или, другими словами, переключает процессор на следующую нить.

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».

Поток, реализующий интерфейс Runnable

```

static class MyRunnable implements Runnable {
    String name;
    public MyRunnable(String name) {
        this.name = name;
        //а здесь суперклассом будет Object а не Thread
        //поэтому делаем свое поле для имени потока
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(name+" is run. loop:"+i);
            try {
                //методы sleep и yield из предыдущего примера у запускаемого
                //экземпляра отсутствуют и их нельзя вызвать по имени
                //НО! sleep и yield – статические в классе Thread,
                //поэтому их можно все равно вызвать
                Thread.sleep(7);
                Thread.yield();
            } catch (InterruptedException e) {
                System.err.println(e);
            }
        }
    }
}

```



```

//это обычный основной поток приложения
public static void main(String[] args) throws InterruptedException {
    //создадим объекты, готовые к запуску в потоке
    MyRunnable one=new MyRunnable("one");
    MyRunnable two=new MyRunnable("--two");
    //one.run(); // ВЫПОЛНИТСЯ МЕТОД, НО ТАК ПОТОК НЕ ЗАПУСТИТСЯ!

    //создаем потоки
    Thread th1 = new Thread(one);
    Thread th2 = new Thread(two);
    System.out.println("one:" + th1.getState() + " two:" + th2.getState());

    //запускаем потоки
    th1.start();
    th2.start();
    System.out.println("one:" + th1.getState() + " two:" + th2.getState());

    //приостановка главного потока, пока не завершатся два созданных
    th2.join(); //порядок приостановки неважен
    th1.join(); //потому что, см. ниже
    System.out.println("one:" + th1.getState() + " two:" + th2.getState());
    System.out.println("Все потоки завершены");
    th1.join(); //методы join доступны даже после завершения потока
    th2.join(); //т.к. экземпляры завершенных объектов еще есть в памяти
               //но они мгновенно отдают управление
    System.out.println("Конец");
}

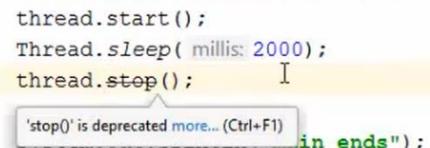
```

D:\Java\jdk1.8.0_73\bin\java ...
one:NEW two:NEW
one:RUNNABLE two:RUNNABLE
one is run. loop:0
--two is run. loop:0
--two is run. loop:1
one is run. loop:1
--two is run. loop:2
one is run. loop:2
--two is run. loop:3
one is run. loop:3
one is run. loop:4
--two is run. loop:4
--two is run. loop:5
one is run. loop:5
one is run. loop:6
--two is run. loop:6
--two is run. loop:7
one is run. loop:7
one is run. loop:8
--two is run. loop:8
one is run. loop:9
--two is run. loop:9
one:TERMINATED two:TERMINATED
Все потоки завершены
Конец

14. Как правильно остановить поток? Для чего нужны методы .stop(), .interrupt(), .interrupted(), .isInterrupted().

ТАК неправильно!

Метод **stop()** прерывает поток грубо, в неоконченном состоянии. Метод считается устаревшим.



Если хотим прервать поток, то должны позаботиться об исключении, чтобы вовремя узнать о непредвиденной ситуации:

```

public static void main(String[] args) throws InterruptedException {
    System.out.println("main starts");
    InterruptedThread thread = new InterruptedThread();
    thread.start();
    Thread.sleep( millis: 2000 );
    thread.interrupt();

    thread.join();
    System.out.println("main ends");
}

```

Есть возможность послать сигнал потоку, что мы хотим его прервать **interrupt()**.
Есть возможность в самом потоке проверить, хотят ли его прервать **isInterrupted()**.
Что делать, если проверка показала, что поток хотят прервать, должен решать сам программист.

```

public void run() {
    for (int i = 1; i <= 1000000000; i++) {
        if(isInterrupted()){
            System.out.println("Potok хотят прервать");
            System.out.println("Mi ubedilis, chto sosotyanie" +
                " vsekh obyektov normalnoe i reshili zavershit rabotu potoka");
            return;
        }
        sqrtSum += Math.sqrt(i);
        try {
            sleep( millis: 10000 );
        } catch [InterruptedException e] {
            System.out.println("Potok хотят прервать во время сна." +
                "Davayte zavershim ego rabotu");
        }
    }
    System.out.println(sqrtSum);
}

```

Метод interrupt

В классе Thread существует переменная `isInterrupted` и метод остановки `interrupt()`.

Никто не гарантирует, что нить можно остановить. Она может остановиться только сама.

Код	Описание
<pre> class Clock implements Runnable { public void run() { Thread current = Thread.currentThread(); while (!current.isInterrupted()) { try { Thread.sleep(1000); } catch (InterruptedException e) { e.printStackTrace(); current.interrupt(); } System.out.println("Tik"); } } } public static void main(String[] args) throws Exception { Clock clock = new Clock(); Thread clockThread = new Thread(clock); clockThread.start(); Thread.sleep(10000); clockThread.interrupt(); } </pre>	<p>Т.к. много нитей могут вызывать метод <code>run</code> одного объекта, то объект <code>Clock</code> в своем методе <code>run</code> получает объект вызвавшей его нити («текущей нити»).</p> <p>Класс <code>Clock</code> (часы) будет писать в консоль раз в секунду слово «<code>Tik</code>», пока переменная <code>isInterrupted</code> текущей нити равна <code>false</code>.</p> <p>Когда переменная <code>isInterrupted</code> станет равной <code>true</code>, метод <code>run</code> завершится.</p>
	<p>Главная нить, запускает дочернюю нить – часы, которая должна работать вечно.</p> <p>Ждет 10 секунд и отменяет задание, вызовом метода <code>interrupt</code>. Главная нить завершает свою работу. Нить часов завершает свою работу.</p>

15. Чем Runnable отличается от Callable?

Коротко: Метод `Runnable.run()` не возвращает никакого значения, `Callable.call()` возвращает объект Future, который может содержать результат вычислений; Метод `run()` не может выбрасывать проверяемые исключения, в то время как метод `call()` может.

Разница между интерфейсами Runnable и Callable:

Интерфейс	Runnable	Callable – новый аналог
Доступны	с java 1.1	добавлен после 1.5
Метод	метод запуска <code>run()</code>	метод вызова <code>call()</code>
Выбрасывает исключения?	НЕТ	ДА
Тип возвращаемого значения	не может возвращать значения (<code>void</code>)	использует Generic'и для определения типа возвращаемого объекта (возвращает <code>Future</code>)
Присоединитесь к пулу потоков для выполнения	использует метод <code>execute()</code> интерфейса <code>ExecutorService</code>	использует метод <code>submit()</code>

```
class Factorial implements Runnable {
    int f;

    public Factorial(int f) {
        this.f = f;
    }

    @Override
    public void run() {
        if (f<=0) {
            System.out.println("Vi vveli nevernoe chislo");
            return;
        }
        int result = 1;
        for(int i = 1; i<=f; i++){
            result*=i;
        }
        RunnableFactorial.factorialResult = result;
    }
}

class Factorial2 implements Callable<Integer> {
    int f;

    public Factorial2(int f) {
        this.f=f;
    }

    @Override
    public Integer call() throws Exception {
        if(f<=0){
            throw new Exception("Vi vveli nevernoe chislo");
        }
        int result = 1;
        for(int i = 1; i<=f; i++){
            result *=i;
        }
        return result;
    }
}
```

16. Что такое FutureTask?

FutureTask представляет собой **отменяемое асинхронное вычисление** в параллельном потоке. Этот класс предоставляет базовую реализацию Future, с методами для запуска и остановки вычисления, методами для запроса состояния вычисления и извлечения результатов.

java.util.concurrent
Interface Future<V>

Type Parameters:

V - The result type returned by this Future's get method

All Known Subinterfaces:

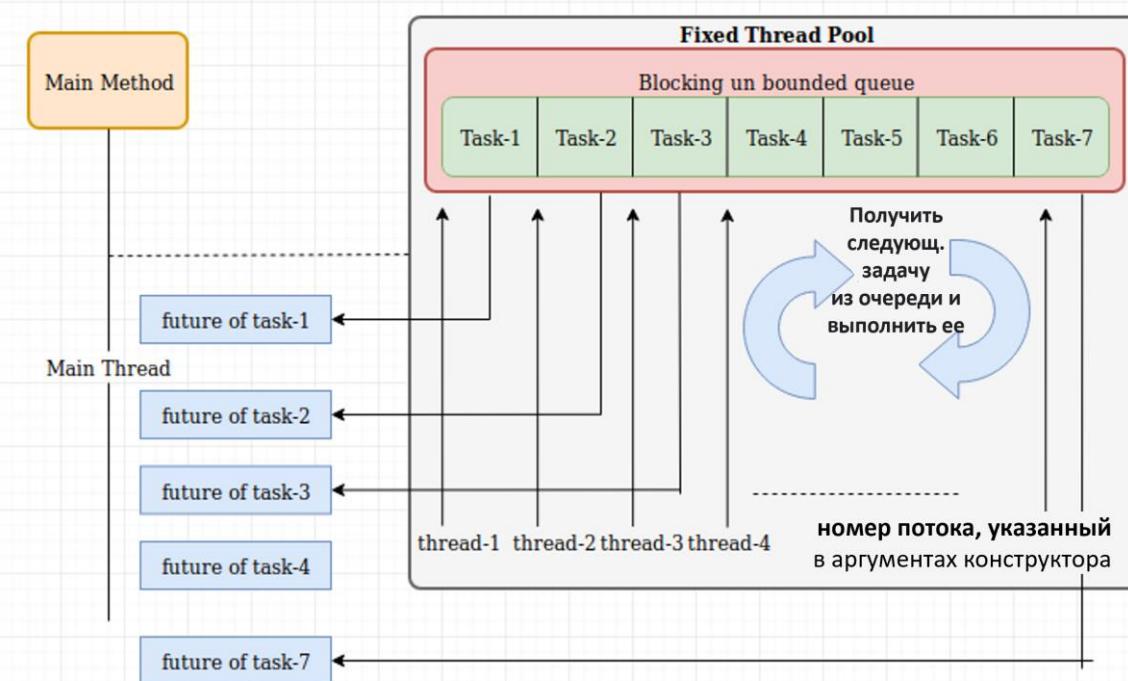
Response<T>, RunnableFuture<V>, RunnableScheduledFuture<V>, ScheduledFuture<V>

All Known Implementing Classes:

ForkJoinTask, FutureTask, RecursiveAction, RecursiveTask, SwingWorker

Результат может быть получен только когда вычисление завершено, метод получения будет заблокирован, если вычисление ещё не завершено.

Объекты FutureTask могут быть использованы для обёртки объектов Callable и Runnable. Так как FutureTask помимо Future реализует Runnable, его можно передать в Executor на выполнение.



Вызываемое (callable) и будущее выполнение в фиксированном пуле потоков

Callable, так же как и Runnable, представляет собой **определенное задание, которое выполняется потоком**. В отличии от Runnable, Callable:

- имеет тип возвращаемого значения (НЕ void)
- может выбрасывать исключение

Метод **submit()** передаёт задание (task) в пул потоков для выполнения одним из потоков и **возвращает тип Future**, в котором и хранится результат выполнения задания.

Метод **get()** позволяет получить результат из объекта Future.

```
public static void main(String[] args) {
    ExecutorService executorService = Executors.newSingleThreadExecutor();
    Factorial2 factorial2 = new Factorial2( f: 6 );
    Future<Integer> future = executorService.submit(factorial2);
    try { Результат еще нет, он будет в будущем!
        System.out.println("Xotim poluchit rezultat");
        factorialResult = future.get();
        System.out.println("Poluchili rezultat");
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ExecutionException e) {
        System.out.println(e.getCause());
    }
    finally {
        executorService.shutdown();
    }
    System.out.println(factorialResult);
}
```

Когда с помощью метода **get()** мы пытаемся достать результат, то поток **main еще не закончил работу**. Поэтому поток будет заблокирован до тех пор, пока task не завершится.

То есть пока факториал не будет найден и **Future не вернёт нам результат**.

17. Что такое deadlock? <https://www.baeldung.com/cs/deadlock-livelock-starvation>

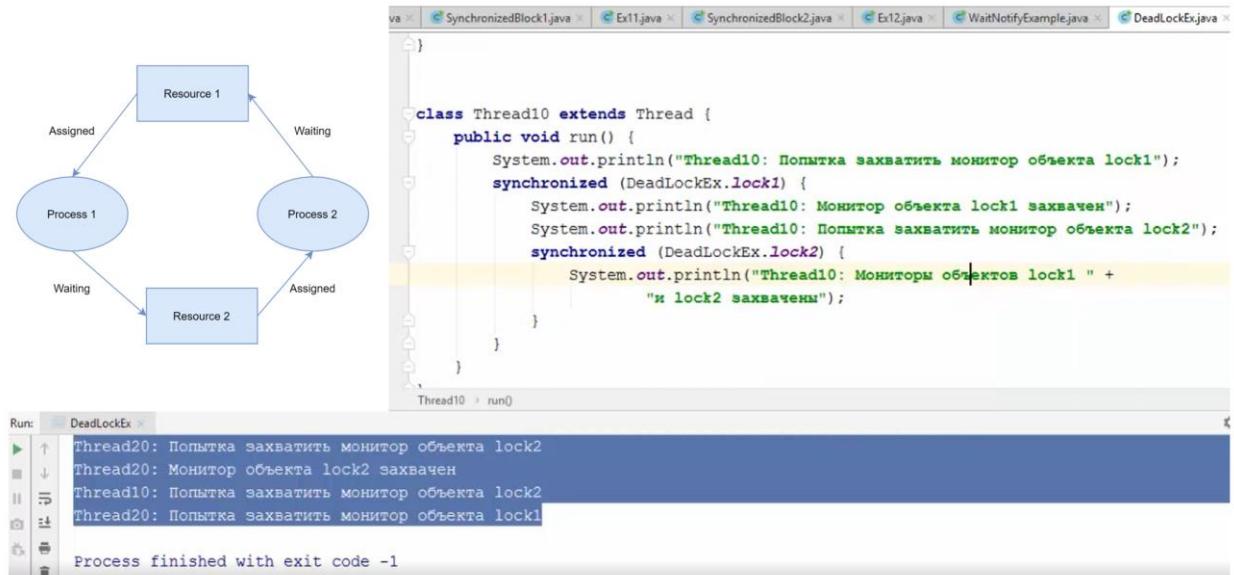
Deadlock – это взаимная блокировка, ситуация, когда два или более потока «наступают друг-другу на хвост» – зависают в вечном ожидании ресурсов, захваченных друг другом.

Deadlock – это ситуация в многопоточном программировании, когда два или более потока започены навсегда, ожидают друг друга и ничего не делают.

Такая ситуация может возникнуть, когда несколько потоков используют синхронизацию на нескольких объектах **не в одинаковом порядке**.

В мультипрограммной среде более одного процесса могут конкурировать за конечный набор ресурсов. Если процесс запрашивает ресурс, а ресурс в данный момент недоступен, то процесс ожидает его. Иногда этому ожидающему процессу никогда не удастся получить доступ к ресурсу.

На рисунке **сценарий взаимоблокировки** между процессами 1 и 2. Оба процесса удерживают один ресурс и ждут другого ресурса, удерживаемого другим процессом. Это тупиковая ситуация, поскольку ни процесс 1, ни процесс 2 не могут двигаться вперед, пока один из процессов не откажется от своего ресурса (пример кода из урока Заура Трегулова).



Выход: синхронизироваться в одинаковом порядке для разных методов.

Циклическое ожидание: это условие можно сделать ложным, установив общий порядок всех типов ресурсов и гарантируя, что каждый процесс запрашивает ресурсы **в возрастающем порядке перечисления**. Например, если есть набор **n** ресурсов **R1, R2..... Rn** и процессу требуется ресурс **R1, R2**, то для успешного выполнения задачи ему нужно сначала запросить **R1**, а затем **R2**.

Стандартный подход к обеспечению гарантии защиты от дедлока – установка строгого порядка взятия блокировок. Если для мониторов А и В соблюдается всеобщий порядок захвата АВ (и соответственно отпускания ВА), то ни с одним потоком не случится попасть на ожидание В, успешно при этом захватив А.

Из этого можно догадаться, простой способ гарантировать возможность дедлока – явно нарушить это условие.

Нарушение условия даст дедлок «скорее всего когда-нибудь». Чтобы получить его **точно и с первого раза**, нужно гарантировать, что оба потока окажутся на этапе

Дедлок (DeadLock) — взаимная блокировка возникает, когда:

- А. Каждой нити в процессе работы нужно захватить оба мютекса.
- Б. Первая нить захватила первый мютекс и ждет освобождения второго.
- С. Вторая нить захватила второй мютекс и ждет освобождения первого.

между захватами одного и другого ресурса в одно время. Это можно сделать множеством способов, в примере ниже использован [CyclicBarrier](#).

Вопрос дедлоков – одна из краеугольных тем [параллельных вычислений](#), уходящая далеко за рамки этого вопроса. Для дальнейшего изучения рекомендуются статьи на википедии [про дедлеки](#), про [задачу об обедающих философах](#) как классическая иллюстрация проблемы, и глава 10.1 книги [Java Concurrency in Practice](#).

```
Object a = new Object(), b = new Object(); // мониторы
CyclicBarrier barrier = new CyclicBarrier( parties: 2);

void deadlock() {
    new Thread(() -> {
        synchronized (a) { // забираем А
            try { barrier.await(); } catch (Exception e) { } // синхронизируемся
            synchronized (b) { } // пытаемся забрать В
        }
    }).start();

    new Thread(() -> {
        synchronized (b) { // забираем В
            try { barrier.await(); } catch (Exception e) { } // синхронизируемся
            synchronized (a) { } // пытаемся забрать А
        }
    }).start();
}
```

Взаимная блокировка (deadlock) - явление, при котором все потоки находятся в режиме ожидания и своё состояние не меняют. Происходит, когда достигаются состояния:

- **взаимного исключения:** по крайней мере один ресурс занят в режиме неделимости и, следовательно, только один поток может использовать ресурс в данный момент времени.
- **удержания и ожидания:** поток удерживает как минимум один ресурс и запрашивает дополнительные ресурсы, которые удерживаются другими потоками.
- **отсутствия пред очистки:** операционная система не переназначает ресурсы: если они уже заняты, они должны отдаваться удерживающим потокам сразу же.
- **циклического ожидания:** поток ждет освобождения ресурса другим потоком, который в свою очередь ждет освобождения ресурса, заблокированного первым потоком.

Простейший способ избежать взаимной блокировки – [не допускать циклического ожидания](#).

Этого можно достичь, получая мониторы разделяемых ресурсов в определенном порядке и освобождая их в обратном порядке.

18. Что такое livelock?

Livelock – похожая на DeadLock проблема, с тем лишь отличием, что потоки не останавливаются, а вместо этого зацикливаются, выполняя одни и те же бесполезные действия, ходят по кругу.

LiveLock – это ситуация, когда два или более потоков залочены навсегда, ожидают друг друга, проделывают какую-то работу, но без какого-либо прогресса.

Работа есть, прогресса нет.

На рисунке показан пример livelock. И «процесс 1», и «процесс 2» нуждаются в общем ресурсе. Каждый процесс проверяет, находится ли другой процесс в активном состоянии. Если да, то он передает ресурс другому процессу.

Однако, поскольку оба процесса находятся в неактивном состоянии, оба продолжают передавать ресурсы друг другу на неопределенный срок.

livelock – тип взаимной блокировки, при котором несколько потоков выполняют бесполезную работу, попадая в зацикленность при попытке получения каких-либо ресурсов.

При этом их состояния постоянно изменяются в зависимости друг от друга. Фактической ошибки не возникает, но КПД системы падает до 0. Часто возникает в результате попыток предотвращения deadlock.

Реальный пример livelock, – когда два человека встречаются в узком коридоре и каждый, пытаясь быть вежливым, отходит в сторону, и так они бесконечно двигаются из стороны в сторону, абсолютно не продвигаясь в нужном им направлении.

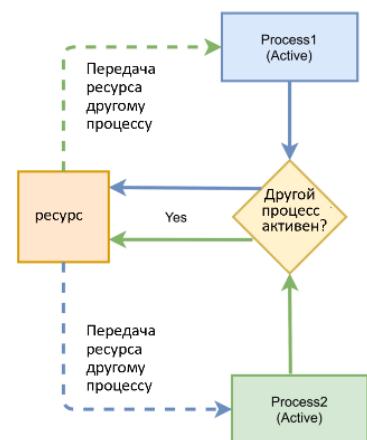
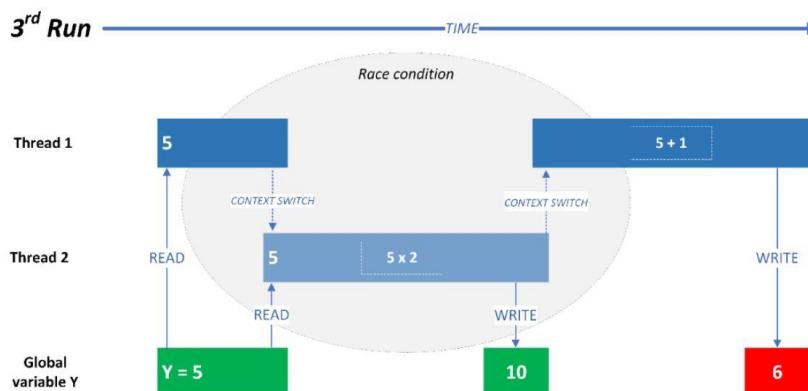
* Что такое lock starvation?

Это ситуация, когда **менее приоритетные потоки** ждут долгое время для того, чтобы могли запуститься.

19. Что такое race condition?

Состояние гонки Race Condition – ошибка проектирования многопоточной системы или приложения, при которой работа зависит от того, в каком порядке выполняются потоки.

Состояние гонки возникает, когда поток, который должен исполнится в начале, проиграл гонку и первым исполняется другой поток: поведение кода изменяется, из-за чего возникают недетерминированные (неопределенные) ошибки.



DataRace – это свойство выполнения программы. Согласно JMM, выполнение считается содержащим гонку данных, если оно содержит по крайней мере два конфликтующих доступа (чтение или запись в одну и ту же переменную), которые не упорядочены отношениями «happens before».

Starvation – голодание, т.е. потоки не заблокированы, но есть нехватка ресурсов из-за чего потоки ничего не делают.

Самый простой способ решения — копирование переменной в локальную переменную. Или просто синхронизация потоков методами и sync-блоками.

20. Что такое Фреймворк fork/join? Для чего он нужен?

Фреймворк Fork/Join, представленный в JDK 7 – это набор классов и интерфейсов, позволяющих использовать преимущества многопроцессорной архитектуры современных компьютеров.

Он разработан для выполнения задач, которые можно рекурсивно **разбить на маленькие подзадачи**, которые можно **решать параллельно**.

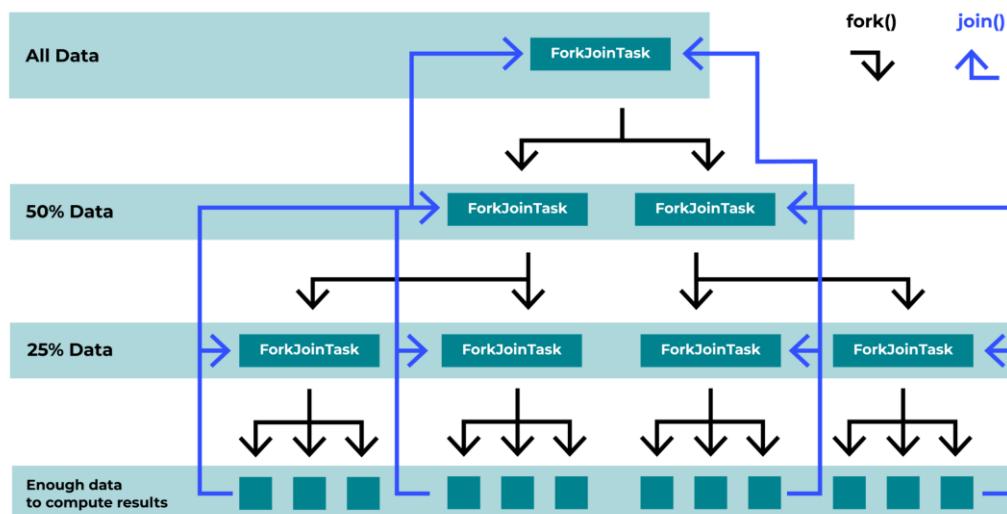
Этап Fork: большая задача разделяется на несколько меньших подзадач, которые в свою очередь также разбиваются на меньшие. И так до тех пор, пока задача не становится **тривиальной и решаемой последовательным способом**.

Этап Join: далее (опционально) идёт процесс «свёртки», т.е. решения подзадач некоторым образом объединяются пока не получится решение всей задачи.

Решение всех подзадач (в т.ч. и само разбиение на подзадачи) происходит параллельно.

fork() to deconstruct and join() to compose the result

fork() для деконструкции и join() для составления результата



Стрелки **fork()** показывают, как **разбить проблему на задачи**, которые сосредотачиваются на более и более конкретных частях данных и выполняются асинхронно.

Стрелка **join()** показывает, как **результаты всплывают** от самых мелких задач к верхней задаче. Здесь вы найдете окончательное решение в методе **compute()**.

Статья: <https://openclassrooms.com/en/courses/5684021-scale-up-your-code-with-java-concurrency/6660941-create-a-recursive-solution-using-fork-join>

Для решения некоторых задач этап Join не требуется. Например, для параллельного QuickSort массив рекурсивно делится на всё меньшие и меньшие диапазоны, пока не вырождается в тривиальный случай из одного элемента. Хотя в некотором смысле Join будет необходим и тут, т.к. всё равно остаётся необходимость дождаться пока не закончится выполнение всех подзадач.

Ещё одно преимущество этого фреймворка заключается в том, что он использует **work-stealing алгоритм**: потоки, которые завершили выполнение собственных подзадач, могут «украсть» подзадачи у других потоков, которые все ещё заняты.

47 – 40 = 7 секунд

```

public class Main {
    static long numOperations = 10_000_000_000L;
    static int numThreads = Runtime.getRuntime().availableProcessors();
    public static void main(String[] args) throws Exception {
        System.out.println(new Date());
        ForkJoinPool pool = new ForkJoinPool(numThreads);
        System.out.println(pool.invoke(new MyFork(0, numOperations)));
        System.out.println(new Date());
    }
    static class MyFork extends RecursiveTask<Long> {
        long from, to;
        public MyFork(long from, long to) {
            this.from = from;
            this.to = to;
        }
        @Override
        protected Long compute() {
            if(to - from <= numOperations/numThreads) {
                long j = 0;
                for (long i = from; i < to; i++) {
                    j += i;
                }
                return j;
            } else {
                long middle = (to + from)/2;
                MyFork firstHalf = new MyFork(from, middle);
                firstHalf.fork();
                MyFork secondHalf = new MyFork(middle + 1, to);
                long secondValue = secondHalf.compute();
                return firstHalf.join() + secondValue;
            }
        }
    }
}

```

Run Main

```

/usr/lib/jvm/java-8-oracle/bin/java ...
Thu Sep 08 00:45:40 EEST 2016
-5340232226128654848
Thu Sep 08 00:45:47 EEST 2016
Process finished with exit code 0

```

>Main

```

/usr/lib/jvm/java-8-oracle/bin/java ...
Thu Sep 08 00:56:55 EEST 2016
-534023226128654848
Thu Sep 08 00:56:57 EEST 2016

```

57 – 55 = 2 секунды!!! ForkJoinFramework

21. Что означает ключевое слово synchronized? Где и для чего может использоваться?

Код	Описание
<pre> class MyClass{ private String name1 = "Оля"; private String name2 = "Лена"; public void swap(){ synchronized (this){ String s = name1; name1 = name2; name2 = s; } } } </pre>	<p>Метод swap меняет местами значения переменных name1 & name2.</p> <p>Что же будет если его вызывать из двух нитей одновременно?</p>

Когда одна нить заходит внутрь блока кода, помеченного словом synchronized, то Java машина тут же блокирует мьютекс у объекта, который указан в круглых скобках после слова synchronized.

Больше ни одна нить не сможет зайти в этот блок, пока наша нить его не покинет. Как только наша нить выйдет из блока, помеченного `synchronized`, то мьютекс тут же автоматически разблокируется и будет свободен для захвата другой нитью.

Ключевым словом `synchronized` может быть помечен как блок кода, так и метод.

Зарезервированное слово позволяет добиваться синхронизации в помеченных им методах или блоках кода. **Блокировка по ОБЪЕКТУ!**

Нити мешают друг-другу, когда пытаются сообща работать с общими объектами и/или ресурсами. Поэтому был придуман специальный объект – **мьютекс**. Он имеет два состояния: объект свободен или занят (еще называют заблокирован или разблокирован, `lock` или `unlock`, принцип «одиночный семафор»).

Разработчики встроили мьютекс в класс `Object`, так что даже **создавать его не нужно, mutex есть у каждого объекта**.

Ключевое слово `synchronized` можно применять как модификатор метода, и как самостоятельный оператор с блоком кода. Выполняет код при захваченном мониторе объекта. В виде **оператора** объект указывается явно. В виде **модификатора нестатического метода** используется `this`, **статического** – `.class` текущего класса.

Один из основных инструментов обеспечения потокобезопасности. Одновременно выполняется не более одного блока `synchronized` на одном и том же объекте. Такая блокировка называется *intrinsic lock* или *monitor lock*.

Пример блока:

```
static final Object lock = new Object();
public void abc() {
    synchronized(lock) {
```

method body

```
        {   block body   } 
```

method body }

Блок `synchronized` также необходим для использования методов `wait`, `notify`, `notifyAll`.

Блокировка по объекту:

Любой объект Java может служить блокировкой через ключевое слово `synchronized`

```
synchronized (monitor){ // Получение блокировки
    /*...*/
}                                // критическая секция
                                    // Снятие блокировки
```

Снятие блокировки производится автоматически!

Блокировка по методу экземпляра

Метод экземпляра может быть объявлен `synchronized`

```
public synchronized int getValue() {
    ...
}
```

Это практически эквивалентно

```
public int getValue() {
    synchronized (this) {
        ...
    }
}
```

но реализация в байт-коде будет медленнее

Блокировка по методу класса

Метод класса может быть объявлен `synchronized`

```
Class Example {
    public static synchronized int getValue() {
        ...
}
```

Это практически эквивалентно

```
public static int getValue() {
    synchronized (Example.class) {
        ...
}
```

⚠️

❗️

22. Что является монитором у статического синхронизированного класса?

Объект типа Class, соответствующий классу, в котором определен метод.

```
public class SynchronizedBlock2 {
    volatile static int counter = 0;

    public static void increment() {
        synchronized (SynchronizedBlock2.class) {
            counter++;
        }
    }
}
```

23. Что является монитором у НЕ статического синхронизированного класса?

Объект this

24. Способы управления потоками `java.util.concurrent`

Возможности классов этого пакета обеспечивают высокую производительность, масштабируемость, построение потокобезопасных блоков.

Почти все коллекции, с которыми мы работали НЕ были поток- безопасными.
Исключения (устаревшие): Vector, Stack и HashTable.

Synchronized collections получаются из обычных коллекций благодаря ОБЁРТКЕ.

Collections.synchronizedXYZ(коллекция)

Коллекции для работы с многопоточностью

Synchronized collections

Получаются из традиционных коллекций благодаря их обёртыванию

Concurrent collections

Изначально созданы для работы с многопоточностью

Синхронизированные коллекции достигают потокобезопасности благодаря тому, что **используют Lock через synchronized блоки для всех методов**. Это значит, что если несколько потоков захотят добавить или удалить элемент в ArrayList, то доступ в коллекцию будет осуществляться через один поток.

Т.е. первый поток заходит, ставит Lock (замок), отрабатывает и только после этого (когда он закончит работу), будет разблокирован доступ для работы второго потока и т.д.

Операция «дорогая», производительность у таких коллекций при многопоточности низкая.

Чтобы обезопасить приложение от вызова исключения **ConcurrentModificationException** необходимо целиком блокировать коллекцию на время перебора.

Пример слева: выброшено исключение ConcurrentModificationException.

Пример справа: когда идёт перебор элементов и вывод в консоль, то до тех пор, пока этот поток не отработает, метод remove() не будет вызван, т.е. поток 2 будет ждать окончания работы потока 1 и результат получим правильный (консоль).

```

Runnable runnable1 = () -> {
    Iterator<Integer> iterator = synchList.iterator();
    while(iterator.hasNext()) {
        System.out.println(iterator.next());
    }
};
Runnable runnable2 = () ->
    synchList.remove( index: 10 );

Thread thread1 = new Thread(runnable1);
Thread thread2 = new Thread(runnable2);

SynchronizedCollectionEx2 > main() > ()->...

```

```

Runnable runnable1 = () -> {
    synchronized (synchList) {
        Iterator<Integer> iterator = synchList.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
};
Runnable runnable2 = () ->
    synchList.remove( index: 10 );

Thread thread1 = new Thread(runnable1);
Thread thread2 = new Thread(runnable2);
thread1.start();
thread2.start();
thread1.join();
thread2.join();
System.out.println(synchList);

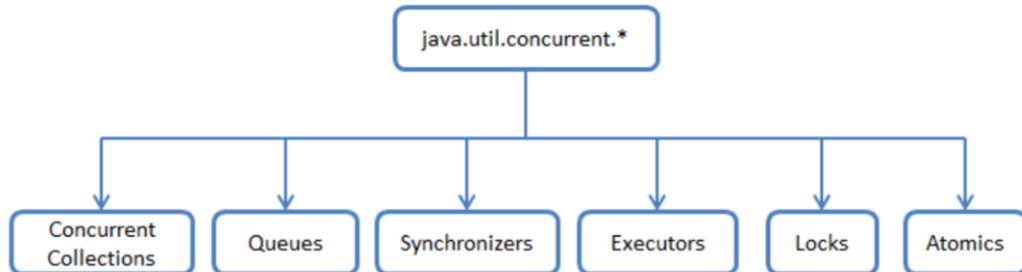
```

SynchronizedCollectionEx2 (1) > 998
999
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,

Process finished with exit code 0

Состав java.util.concurrent

Специально для работы в условиях многопоточности были созданы Concurrent collections: усовершенствован вызов утилит синхронизации, добавлены классы семафоров и блокировок и т.д.



Atomics – набор потокобезопасных переменных и массивов.

Механизм управления заданиями, основанный на интерфейсе Executor:

1. организация запуска пула потоков: **ThreadPoolExecutor**
2. Эффективная реализация пула ExecutorService – **ForkJoinPool**
3. Службы планирования: **ScheduledThreadPoolExecutor**

Параллельные аналоги классов-коллекций

1. Блокирующие очереди **BlockingQueue** и **BlockingDeque**, гарантирующие остановку потока если нет элемента или нет места для вставки)
2. Класс **Exchanger** позволяет двум потокам обмениваться объектами.
3. Классы **CopyOnArrayList** и **CopyOnWriteArrayList**, копирующие своё содержимое при попытке его изменения, причём ранее полученный итератор будет корректно продолжать работать с исходным набором данных.
4. **ConcurrentHashMap**, **ConcurrentLinkedQueue** – эффективные аналоги **HashTable** и **LinkedList**.

Классы – барьеры синхронизации (Synchronizers):

- **CountDownLatch** – заставляет потоки ожидать завершения заданного числа операций, по окончании чего все ожидающие потоки освобождаются.
- **Semaphore** – предлагают потоку ожидать завершения действий в других потоках.
- **CyclicBarrier** – предлагает некоторым потокам ожидать момента, когда они все достигнут какой-либо точки, после чего барьер снимается.
- **Phaser** – улучшенная реализация барьера для синхронизации потоков, совмещает в себе функционал **CyclicBarrier** и **CountDownLatch**. Количество потоков может динамически изменяться. Класс может пере-использоваться.

Неблокирующие коллекции:

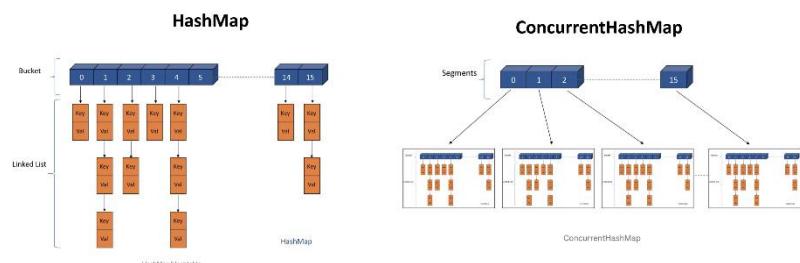
- ConcurrentLinkedQueue (LinkedList)
- ConcurrentLinkedDeque (ArrayDeque)
- CopyOnWriteArrayList (ArrayList)
- CopyOnWriteHashSet (HashSet)
- ConcurrentHashMap (HashMap)
- ConcurrentSkipListMap (TreeMap)
- ConcurrentSkipListSet (TreeSet)

Принципы:

1. Простые операции атомарны
2. Пакетные операции (addAll, removeAll) могут быть не атомарны!
3. Как правило, длина не хранится (isEmpty() !)
4. Не кидает ConcurrentModificationException

*Расскажи про устройство и алгоритм работы ConcurrentHashMap

ConcurrentHashMap создает массив поверх него, и каждый индекс этого массива представляет HashMap.

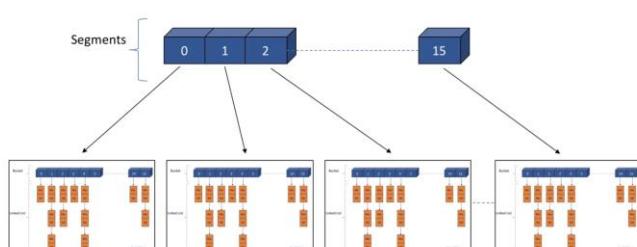


В Java 8 ConcurrentHashMap – это уже древовидная структура вместо связанного списка для дальнейшего повышения производительности.

ConcurrentHashMap implements the interface **ConcurrentMap**, which in turn inherits from the **Map** interface.

In ConcurrentHashMap, any number of threads can read elements without blocking each other.

In ConcurrentHashMap, thanks to its segmentation, when an element is modified, only the bucket it belongs to is locked.



Операция чтения, записи и удаления в ConcurrentHashMap

а) Если посмотреть на приведенную выше диаграмму, становится ясно, что независимо от того, является ли это операцией вставки или чтения, необходимо **сначала определить индекс сегмента**, в котором предполагается выполнение операции вставки/чтения.

б) После того, как это идентифицировано, необходимо определить внутреннюю корзину/массив хэш-карты, чтобы найти точную позицию для вставки/чтения.

с) После определения корзины выполните итерацию по связанному списку, чтобы проверить пару ключ-значение.

- В случае вставки, если ключ совпадает, замените значение новым, в противном случае вставьте ключ со значением в конце связанного списка.
- В случае чтения **всезде**, где совпадают ключи, извлеките значение и верните это значение, а если нет совпадения, верните ноль.
- В случае удаления, если ключ совпадает, удалите ссылку, соответствующую этому ключу.

Смотрите рисунок справа (Hashtable).

Оба потока блокируют коллекцию целиком и это делает Hashtable неэффективным...

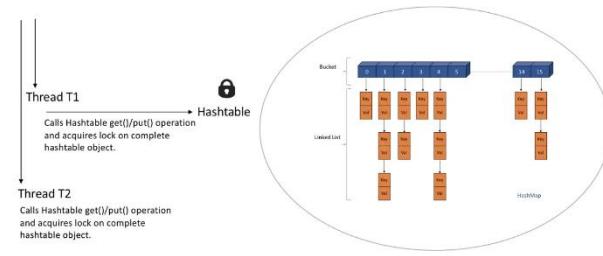
Именно поэтому Java придумала ConcurrentHashMap.

ConcurrentHashMap работает иначе, чем, Hashtable, поскольку **ConcurrentHashMap получает блокировку для каждого сегмента отдельно**.

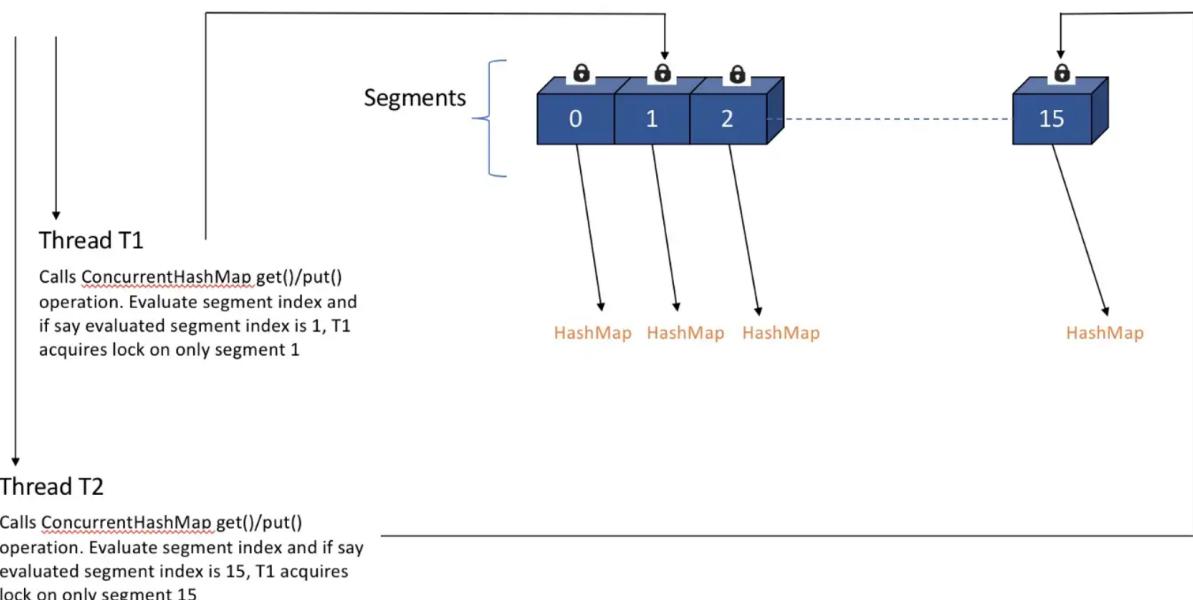
Это означает, что вместо блокировки всей карты он имеет несколько блокировок на уровне сегментов. Таким образом, два потока, работающие с разными сегментами, могут получить блокировку этих сегментов, **не мешая друг другу, и могут работать одновременно**, поскольку они работают с отдельными блокировками сегмента.

Поток T1 вызывает concurrentHashMap.put(key, value), он получает блокировку, скажем, в сегменте 1 и вызывает метод put.

Поток T2 вызывает concurrentHashMap.put(key, value), он получает блокировку, скажем, в сегменте 15 и вызывает метод put, как показано ниже.



Потоки, получающие блокировку на Hashtable



Потоки, получающие блокировку на ConcurrentHashMap

Именно так ConcurrentHashMap повышает производительность и обеспечивает безопасность потоков.

Одновременные операции чтения и записи несколькими потоками в одном или разных сегментах ConcurrentHashMap:

Операция чтения/получения: два потока T1 и T2 могут одновременно считывать данные из одного и того же или разных сегментов ConcurrentHashMap, не блокируя друг друга.

Операция записи/ввода: два потока T1 и T2 могут одновременно записывать данные в разные сегменты, не блокируя друг друга.

Но два потока не могут одновременно записывать данные в одни и те же сегменты. Один должен ждать, пока другой завершит операцию.

Операция чтения-записи: два потока могут одновременно читать и записывать данные в разные сегменты, не блокируя друг друга. Как правило, операции извлечения не блокируются, поэтому могут пересекаться с операциями записи (помещения/удаления). Последнее обновленное значение будет возвращено операцией получения, которая является самым последним обновленным значением операции записи (включая размещение/удаление).

Размер сегментов и уровень параллелизма ConcurrentHashMap

По умолчанию ConcurrentHashMap имеет размер массива сегментов равный 16, поэтому **одновременно 16 потоков** могут помещать данные в карту, учитывая, что каждый поток работает с отдельным индексом массива сегментов.

ConcurrentHashMap имеет конструктор с **3 аргументами**, который помогает настроить размер массива сегментов, определив `concurrencyLevel`.

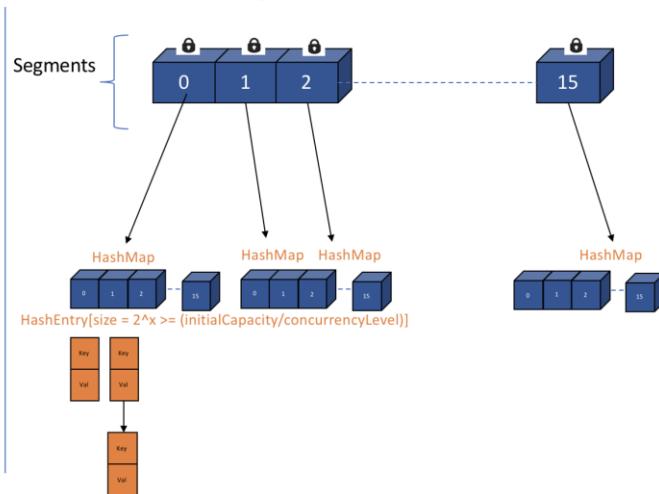
```
ConcurrentHashMap map = new ConcurrentHashMap(int initialCapacity,
float loadFactor, int concurrencyLevel)
```

Например, `ConcurrentHashMap map = new ConcurrentHashMap(100, 0.75f, 10)`

- **Начальная емкость карты составляет 100**, что означает, что ConcurrentHashMap обеспечит наличие места для добавления 100 пар ключ-значение после создания.
- **LoadFactor равен 0,75f**, это означает, что, когда среднее количество элементов на карте превышает 75 (начальная емкость * коэффициент загрузки = $100 * 0.75 = 75$), в это время размер карты будет увеличен, а существующие элементы на карте перефразированы для добавления в новую карту большего размера.
- **Уровень параллелизма равен 10**, это означает, что в любой момент времени размер массива сегментов будет равен 10 или больше 10, так что 10 потоков смогут параллельно записывать на карту.

Но как рассчитать размер массива сегментов?

Segment [size = $2^x \geq \text{concurrencyLevel}$]



Размер массива сегментов = 2 в степени x, где результат должен быть $\geq \text{concurrencyLevel}$ (в нашем случае это 10).

Размер массива сегментов = $2^4 = 16 \geq 10$, что означает, что размер массива сегментов должен быть равен 16.

Другой пример:

$\text{concurrencyLevel} = 8$, тогда размер массива сегментов = ?
найти $2^x \geq 8$
 $2^3 \geq 8$
Размер массива сегментов будет равен 8.

Теперь, когда вы знаете, как рассчитать размер массива сегментов, сразу может возникнуть вопрос, **каков будет размер внутренней хэш-карты в сегменте?**

Как показано на диаграмме выше, каждый индекс массива сегментов представляет собой HashMap, а в HashMap бакет/корзина/массив относится к классу Entry[], а в **ConcurrentHashMap массив относится к классу HashEntry[]** (в Java 8 — древовидная структура, а не связанный список).

Размер массива сегментов = 2 в степени x, где результат должен быть $\geq \text{concurrencyLevel}$ (в нашем случае это 10).
Размер массива сегментов = $2^4 = 16 \geq 10$, что означает, что размер массива сегментов должен быть равен 16.

Таким образом, размер массива HashEntry[] = $2^x \geq (\text{initialCapacity} / \text{concurrencyLevel})$

Например, `ConcurrentHashMap map = new ConcurrentHashMap(64, 0.75f, 8)`

- размер массива HashEntry[] = $2^x \geq 8$ ($64/8$)
- найдите $2^x \geq 8$
- $2^3 \geq 8 \geq$

Размер массива HashEntry[] будет равен 8.

Т.е. каждый сегмент всегда будет иметь емкость 8 пар ключ-значение в ConcurrentHashMap после его создания.

В ConcurrentHashMap **каждый сегмент повторно хешируется отдельно**, поэтому нет конфликта между потоком 1, записывающим в сегмент с индексом 1, и потоком 2, записывающим в сегмент с индексом 4.

Например, если поток 1 помещает данные в индекс массива Segment [] 3 и обнаруживает, что массив HashEntry [] необходимо повторно хешировать из-за превышения емкости коэффициента загрузки, тогда он будет повторно хешировать массив HashEntry [], присутствующий в индексе массива Segment [] только 3.

Массив HashEntry[] в других индексах сегмента останется нетронутым и продолжит обслуживать запросы на размещение и получение параллельно.

<https://itsromiljain.medium.com/curious-case-of-concurrenthashmap-90249632d335>

```
public class ConcurrentHashMapEx {  
    public static void main(String[] args) throws InterruptedException {  
        ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<>();  
        map.put(1, "Zaur");  
        map.put(2, "Oleg");  
        map.put(3, "Sergey");  
        map.put(4, "Ivan");  
        map.put(5, "Igor");  
        System.out.println(map);  
  
        Runnable runnable1 = () -> {  
            Iterator<Integer> iterator = map.keySet().iterator();  
            while (iterator.hasNext()) {  
                try {  
                    Thread.sleep( millis: 100 );  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
                Integer i = iterator.next();  
                System.out.println(i + ":" + map.get(i));  
            }  
        };  
    };
```

*Что знаешь про CopyOnWriteArrayList?

CopyOnWriteArrayList:

- volatile массив внутри
- lock только при модификации списка, поэтому операции чтения очень быстрые
- новая копия массива при модификации
- fail-fast итератор
- модификация через iterator невозможна: **UnsupportedOperationException**

Почему важно, чтобы операции по удалению и вставке элементов были не частыми?

При каждом изменении элементов в данной коллекции, создаётся **КЛОН – копия листа нового вида**.

CopyOnWriteArrayList implements interface List.

CopyOnWriteArrayList следует использовать тогда, когда вам нужно добиться потокобезопасности, у вас небольшое количество операций по изменению элементов и большое количество по их чтению.

Итератор запоминает состояние коллекции на момент итерации и выводит в консоль эти элементы. Что происходит в параллельных потоках уже не важно.

Во втором потоке, когда удалили элемент создалась копия коллекции. А затем добавили новый элемент и создалась ещё одна новая копия коллекции. И только после того, как итератор закончил работу, старая копия уже не нужна и теперь итератор работает с самой новой копией данных.

Элементов коллекции может быть много, а операции затратны. Именно поэтому **стоит использовать CopyOnWriteArrayList тогда**, когда в программе НЕ много операций по изменению элементов, **НО много операций чтения**.

```

CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<>();
list.add("Zaur");
list.add("Oleg");
list.add("Sergey");
list.add("Ivan");
list.add("Igor");
System.out.println(list);

Runnable runnable1 = () -> {
    Iterator<String> iterator = list.iterator();
    while (iterator.hasNext()) {
        try {
            Thread.sleep( millis: 100 );
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
};

Runnable runnable2 = () -> {
    try {
        Thread.sleep( millis: 200 );
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    list.remove( index: 4 );
    list.add("Elena");
};

Thread thread1 = new Thread(runnable1);
Thread thread2 = new Thread(runnable2);
thread1.start();

```

The screenshot shows the Java code for `CopyOnWriteArrayListEx` class. It contains two runnables, `runnable1` and `runnable2`, which are scheduled to run sequentially. `runnable1` sleeps for 100ms and iterates over the list. `runnable2` sleeps for 200ms and removes the element at index 4 and adds "Elena" at index 4. Below the code, a tooltip says "Found duplicate code more... (Ctrl+F1)". On the left, there's a file tree with several collection-related packages like `list_interface`, `map_interface`, etc., and a list of classes including `CopyOnWriteArrayListEx` (which is selected). At the bottom, a run configuration window shows the output of the program, listing the names Zaur, Oleg, Sergey, Ivan, and Elena.

Есть еще класс **CopyOnWriteArrayListSet**, который работает по схожему сценарию.

synchronizers

Объекты синхронизации, позволяющие разработчику управлять и/или ограничивать работу нескольких потоков. Содержит пять объектов синхронизации: semaphore, countDownLatch, cyclicBarrier, exchanger, phaser.

CountDownLatch

объект синхронизации потоков, блокирующий один или несколько потоков до тех пор, пока не будут выполнены определенные условия. Количество условий задается счетчиком. При обнулении счетчика, т.е. при выполнении всех условий, блокировки выполняемых потоков будут сняты, и они продолжат выполнение кода. Одноразовый.

CountDownLatch + Executor (пример кода ниже)

The screenshot shows an IDE interface with two panes. The left pane contains the Java code for `CountDownLatchDemo`. The right pane shows the terminal output of the program's execution.

```

public static void main(String[] args) {
    ExecutorService executors= Executors.newFixedThreadPool(5);
    for (int i = 0; i < 5; i++) {
        executors.execute(new Run("Поток №"+i));
    }
    executors.shutdown();
}

public class CountDownLatchDemo {

    private static final CountDownLatch start = new CountDownLatch(5);
    private static final CountDownLatch finish = new CountDownLatch(5);

    private static class Run implements Runnable {
        private final String name;

        Run(String name) { this.name = name; } //конструктор Run

        @Override
        public void run() {
            System.out.println(name + " создан");
            try {
                start.countDown();
                start.await(); //ждем пока все соберутся
                System.out.println("---" + name + " стартовал");
                Thread.sleep(Math.round(Math.random() * 1000)); //работа
                System.out.println("---" + name + " финишировал");
                finish.countDown();
                finish.await(); //ждем пока все соберутся
                System.out.println("----" + name + " завершился");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

Terminal Output:

```

D:\Java\jdk1.8.0_73\bin\java ...
Поток №0 создан
Поток №1 создан
Поток №2 создан
Поток №3 создан
Поток №4 создан
--Поток №4 стартовал
--Поток №0 стартовал
--Поток №1 стартовал
--Поток №2 стартовал
--Поток №3 стартовал
--Поток №2 финишировал
--Поток №3 финишировал
--Поток №1 финишировал
--Поток №4 финишировал
--Поток №0 финишировал
-----Поток №2 завершился
-----Поток №4 завершился
-----Поток №1 завершился
-----Поток №3 завершился
-----Поток №0 завершился

Process finished with exit code 0

```

CyclicBarrier

Барьерная синхронизация останавливает поток в определенном месте в ожидании прихода остальных потоков группы. Как только все потоки достигнут барьера, барьер снимается и выполнение потоков продолжается. Как и CountDownLatch, использует счетчик и похож на него. Отличие: барьер можно использовать повторно (в цикле).

The screenshot shows an IDE interface with two panes. The left pane contains the Java code for `CyclicBarrierExample`. The right pane shows the terminal output of the program's execution.

```

private static class Task implements Runnable {
    private CyclicBarrier barrier;

    public Task(CyclicBarrier barrier) {
        this.barrier = barrier;
    }

    @Override
    public void run() {
        try {
            System.out.println(Thread.currentThread().getName() + " is waiting on barrier");
            barrier.await();
            System.out.println(Thread.currentThread().getName() + " has crossed the barrier");
        } catch (InterruptedException | BrokenBarrierException ex) {
            ex.printStackTrace();
        }
    }
}

public static void main(String args[]) {
    //creating CyclicBarrier with 3 parties i.e. 3 Threads needs to call await()
    final CyclicBarrier cb = new CyclicBarrier(3, new Runnable() {
        @Override
        public void run() {
            //This task will be executed once all thread reaches barrier
            System.out.println("All parties are arrived at barrier, lets play");
        }
    });

    //starting each of thread
    new Thread(new Task(cb), "Thread 1").start();
    new Thread(new Task(cb), "Thread 2").start();
    new Thread(new Task(cb), "Thread 3").start();
}

```

Terminal Output:

```

Runner (1)  CyclicBarrierExample
H:\Program Files\Java\jdk1.8.0_101\bin\java ...
Thread 1 is waiting on barrier
Thread 2 is waiting on barrier
Thread 3 is waiting on barrier
All parties are arrived at barrier, lets play
Thread 3 has crossed the barrier
Thread 1 has crossed the barrier
Thread 2 has crossed the barrier

Process finished with exit code 0

```

Exchanger

Объект синхронизации, используемый для двустороннего обмена данными между двумя потоками. При обмене данными допускается null значения, что позволяет использовать класс для односторонней передачи объекта или же просто, как синхронизатор двух потоков. Обмен данными выполняется вызовом метода **exchange()**, сопровождаемый самоблокировкой потока.

Как только второй поток вызовет метод `exchange`, то синхронизатор `Exchanger` выполнит обмен данными между потоками.

Пример Трегулова: камень, ножницы, бумага (код справа).

```
import java.util.List;
import java.util.concurrent.Exchanger;

public class ExchangerEx {
    public static void main(String[] args) {
        Exchanger<Action> exchanger = new Exchanger<>();
        List<Action> friend1Action = new ArrayList<>();
        friend1Action.add(Action.NOJNICI);
        friend1Action.add(Action.BUMAGA);
        friend1Action.add(Action.NOJNICI);
        List<Action> friend2Action = new ArrayList<>();
        friend2Action.add(Action.BUMAGA);
        friend2Action.add(Action.KAMEN);
        friend2Action.add(Action.KAMEN);

        new BestFriend( name: "Vanya", friend1Action, exchanger);
        new BestFriend( name: "Fetya", friend2Action, exchanger);
    }
}
```

Phaser

Объект синхронизации типа «Барьер», но, в отличие от `CyclicBarrier`, может иметь несколько барьеров (фаз), и количество участников на каждой фазе может быть разным.

Atomic

Набор атомарных классов для выполнения атомарных операций. Операция является атомарной, если её можно безопасно выполнять при параллельных вычислениях в нескольких потоках, не используя при этом ни блокировок, ни синхронизацию `synchronized`.

Queues

Содержит классы формирования неблокирующих и блокирующих очередей для многопоточных приложений. Неблокирующие очереди «заточены» на скорость выполнения, блокирующие очереди приостанавливают потоки при работе с очередью. Работает правило FIFO.

```
package collection.thread_safe;

import java.util.concurrent.ArrayBlockingQueue;

public class ArrayBlockingQueueEx1 {
    public static void main(String[] args) {
        ArrayBlockingQueue<Integer> queue = new ArrayBlockingQueue<>( capacity: 4 );
        queue.add(1);
        queue.add(2);
        queue.add(3);
        queue.add(4);
        System.out.println(queue);
    }
}
```

Если попытаемся добавить пятый элемент при `capacity = 4`, то будет выброшено исключение `queue full` (очередь заполнена). А если использовать метод `offer()`, то исключение не будет выброшено, но элемент не добавится.

ArrayBlockingQueue

ArrayBlockingQueue – потокобезопасная очередь с ограниченным размером (capacity restricted).

Обычно один или несколько потоков добавляют элементы в конец очереди, а другой или другие потоки забирают элементы из начала очереди.

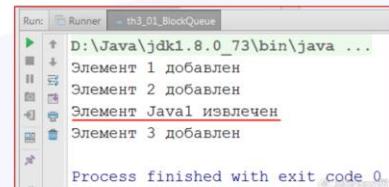
```
public class BlockingQueueDemo {  
    public static void main(String[] args) {  
        //указание что элементов в очереди будет только два  
        final BlockingQueue<String> queue = new ArrayBlockingQueue<>( capacity: 2);  
        //поток на добавление (Java 7)  
        new Thread() {  
            public void run() {  
                for (int i = 1; i < 4; i++) {  
                    try {  
                        queue.put( e: "Java" + i); // добавление 3-х элементов  
                        System.out.println("Элемент " + i + " добавлен");  
                    }catch (InterruptedException e) {  
                        e.printStackTrace();  
                    }  
                }  
            }.start();  
  
        //поток на извлечение (Java 8+)  
        new Thread(() -> {  
            try {  
                Thread.sleep( millis: 1_000);  
                // извлечение одного  
                System.out.println("Элемент " + queue.take() + " извлечен");  
            }catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }).start();  
    }  
}
```

Так работают методы:

put() продюсер

take() потребитель

```
Producer добавил: 8 [5, 6, 7, 8]  
Consumer забрал: 5 [6, 7, 8]  
Producer добавил: 9 [6, 7, 8, 9]  
Consumer забрал: 6 [7, 8, 9]  
Producer добавил: 10 [7, 8, 9, 10]
```



Locks

Механизмы синхронизации потоков, альтернативы базовым synchronized, wait, notify, notifyAll: Lock, Condition, ReadWriteLock.

Lock — базовый интерфейс, предоставляющий более гибкий подход при ограничении доступа к ресурсам/блокам по сравнению с использованием synchronized. Так, при использовании нескольких блокировок, порядок их освобождения может быть произвольный. Имеется возможность перехода к альтернативному сценарию, если блокировка уже захвачена.

Lock и ReentrantLock

Lock – интерфейс, который implementируется классом ReentrantLock.

Также как ключевое слово synchronized, Lock нужен для достижения синхронизации между потоками.

lock()

unlock()

tryLock()

```
class Call {
    private Lock lock = new ReentrantLock();

    void mobileCall() {
        lock.lock();
        try {
            System.out.println("Mobile call starts");
            Thread.sleep( millis: 3000 );
            System.out.println("Mobile call ends");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
}
```

```
class MyCounter implements Runnable {

    public static final Lock lock = new ReentrantLock();

    @Override
    public void run() {
        try {
            //получаем lock
            while (!lock.tryLock( time: 1000, TimeUnit.MILLISECONDS )) {
                System.out.println("Just wait one second in "
                    + Thread.currentThread().getName());
            }

            //полезная работа
            Thread.sleep( millis: 1500 );
            LockDemo.counter++;
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        } finally {
            //отпускаем lock
            lock.unlock();
        }
    }
}
```

```
public class LockDemo {

    static int counter = 0;

    public static void main(String[] args) throws InterruptedException {
        ExecutorService service = Executors.newFixedThreadPool( nThreads: 4 );
        for (int i = 0; i < 5; i++) {
            service.submit(new MyCounter());
        }
        service.shutdown();
        if (service.awaitTermination( timeout: 10, TimeUnit.DAYS )) {
            System.out.println(counter);
        }
    }
}
```

Condition — интерфейсное условие в сочетании с блокировкой Lock позволяет заменить методы монитора/мьютекса (wait, notify и notifyAll) объектом, управляющим ожиданием событий. Объект с условием чаще всего получается из блокировок с использованием метода lock.newCondition(). Таким образом можно получить несколько комплектов wait/notify для одного объекта. Блокировка Lock заменяет использование synchronized, а Condition — объектные методы монитора.

ReadWriteLock — интерфейс создания read/write блокировок, который реализует один единственный класс **ReentrantReadWriteLock**. Блокировку чтение-запись следует использовать при длительных и частых операциях чтения и редких операциях записи. Тогда при доступе к защищенному ресурсу используются разные методы блокировки, как показано ниже:

```
ReadWriteLock rwl = new ReentrantReadWriteLock();
Lock readLock = rwl.readLock();
Lock writeLock = rwl.writeLock();
```

Executors - включает средства, называемые сервисами исполнения, позволяющие управлять потоковыми задачами с возможностью получения результатов через интерфейсы Future и Callable.

```
> public class FutureTaskDemo {  
  
>     public static void main(String[] args) throws ExecutionException, InterruptedException {  
        // тут подготовим задачу  
        Callable<Integer> callTh = () -> { sleep( millis: 500); return 500; };  
  
        // это контейнер для ответа  
        FutureTask<Integer> task = new FutureTask<>(callTh);  
  
        // запускаем задачу в контейнере для одного потока  
        ExecutorService service = Executors.newSingleThreadExecutor();  
        service.submit(task);  
        service.shutdown();  
  
        //тут поток будет ожидать готовности ответа  
        System.out.print("Result:" + task.get());  
    }  
}  
  
FutureTaskDemo ✘  
/Users/akhmelev/Library/Java/JavaVirtualMachines/graalvm-ce-17/Contents/Home/bin/java ...  
Result:500  
Process finished with exit code 0
```

ExecutorService служит альтернативой классу **Thread**, предназначенному для управления потоками.

В основу сервиса исполнения положен интерфейс **Executor**, в котором определен один метод **void execute(Runnable thread)**;

При вызове метода **execute** исполняется поток **thread**.

Интерфейс **ExecutorService**:

Метод	Описание
boolean awaitTermination (long timeout, TimeUnit unit)	Блокировка до тех пор, пока все задачи не завершат выполнение после запроса на завершение работы или пока не наступит таймаут или не будет прерван текущий поток
List<Future<T>> invokeAll (Collection<? extends Callable<T>> tasks)	Выполнение задач с возвращением списка задач с их статусом и результатами завершения
List<Future<T>> invokeAll (Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)	Выполнение задач с возвращением списка задач с их статусом и результатами завершения в течение заданного времени
T invokeAny (Collection<? extends Callable<T>> tasks)	Выполнение задач с возвращением результата успешно выполненной задачи (т. е. без создания исключения), если таковые имеются
T invokeAny (Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)	Выполнение задач в течение заданного времени с возвращением результата успешно выполненной задачи (т. е. без создания исключения), если таковые имеются
boolean isShutdown()	Возвращает true, если исполнитель сервиса остановлен (shutdown)
boolean isTerminated()	Возвращает true, если все задачи исполнителя сервиса завершены по команде остановки (shutdown)
void shutdown()	Упорядоченное завершение работы, при котором ранее отправленные задачи выполняются, а новые задачи не принимаются
List<Runnable> shutdownNow()	Остановка всех активно выполняемых задач, остановка обработки ожидающих задач, возвращение списка задач, ожидающих выполнения
Future<T> submit(Callable<T> task)	Завершение выполнения задачи, возвращающей результат в виде объекта Future
Future<?> submit(Runnable task)	Завершение выполнения задачи, возвращающей объект Future, представляющий данную задачу - то же void execute(Runnable task)
Future<T> submit(Runnable task, T result)	Завершение выполнения задачи, возвращающей объект Future, представляющий данную задачу

25. Stream API & ForkJoinPool Как связаны, что это такое.

В Stream API есть простой способ распараллеливания потока методом **parallel()** или **parallelStream()**, чтобы получить выигрыш в производительности на многоядерных машинах.

По умолчанию parallel stream используют ForkJoinPool.commonPool. Этот пул создается статически и живет пока не будет вызван System::exit. Если задачам не указывать конкретный пул, то они будут исполняться в рамках commonPool.

По умолчанию, размер пула равен на 1 меньше, чем количество доступных ядер.

Когда некий tread отправляет задачу в common pool, то пул может использовать вызывающий tread (caller-thread) в качестве исполнителя. ForkJoinPool пытается загрузить своими задачами и вызывающий tread.

Сортировка слиянием на FJP (ForkJoinPool)

```
class MergeSort extends RecursiveTask<int[]> {
    private final int[] array;
    public MergeSort(int[] array) { this.array = array; }

    @Override
    protected int[] compute() {
        if (array.length > 1) {
            //split
            int mid = array.length / 2;
            int[] left = Arrays.copyOfRange(array, 0, mid);
            int[] right = Arrays.copyOfRange(array, mid, array.length);
            //recursion
            MergeSort leftTask = new MergeSort(left);
            MergeSort rightTask = new MergeSort(right);
            leftTask.fork(); //у FJP внутри LIFO
            rightTask.fork(); //поэтому
            right = rightTask.join(); //join нужен после fork
            left = leftTask.join(); //применяется в обратном порядке
            //merge
            for (int i = 0, il = 0, jr = 0; i < array.length; i++) {
                array[i] = jr == right.length || (il < left.length && left[il] < right[jr])
                    ? left[il++]
                    : right[jr++];
            }
        }
        return array;
    } //compute
}

public static void main(String[] args) {
    int maxSize = 1000;
    int low = 0;
    int max = 400;
    int[] arr = new Random()
        .ints(low, max)
        .limit(maxSize)
        .toArray();
    ForkJoinPool forkJoinPool = new ForkJoinPool();
    MergeSort mergeSort = new MergeSort(arr);
    arr = forkJoinPool.invoke(mergeSort);
    System.out.println(Arrays.toString(arr));
}
```

Run: ForkJoinPoolDemo <
E:\Java\graalvm-ce-java11\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2018.3.5\lib\idea_rt.jar
[0, 0, 1, 1, 1, 2, 2, 3, 3, 3, 3, 4, 5, 5, 5, 5, 6, 6, 6, 6, 6, 7, 7, 8, 8, 9, 9, 9, 9, 10, 10, 10, 11, 11, 12, 12, 13,
Process finished with exit code 0

26. *Зачем используются thread local переменные?

Класс **ThreadLocal** представляет хранилище тред-локальных переменных. По способу использования он похож на обычную обертку над значением, с методами **get()**, **set()** и **remove()** для доступа к нему, и дополнительным фабричным методом **ThreadLocal.withInitial()**, устанавливающим значение по-умолчанию.

Отличие тред-локальной переменной от обычной в том, что **ThreadLocal** хранит отдельную независимую копию значения для каждого ее использующего потока.

Работа с такой переменной поток- безопасна.

Проще говоря, объект класса **ThreadLocal** хранит внутри не одно значение, а как бы хэш-таблицу поток → значение, и при использовании обращается к значению для текущего потока.

Первый, самый очевидный вариант использования – данные, относящиеся

непосредственно к треду, определенный пользователем «контекст потока». На скриншоте ниже пример такого использования: `ThreadId.get()` вернет порядковый номер текущего треда.

Другой случай, с которым локальная переменная потока может помочь – кэширование `read-only` данных в многопоточной среде без дорогостоящей синхронизации.

Помимо обычного `ThreadLocal`, в стандартной библиотеке присутствует его расширение `InheritableThreadLocal`. Этот класс «наследует» значение – изначально берет его для потока, являющегося родителем текущего.

```
class ThreadId {  
    // AtomicInteger счетчик, хранящий ID для следующего потока.  
    static AtomicInteger nextId = new AtomicInteger( initialValue: 0 );  
    // Локальная переменная потока, хранящая ID каждого потока.  
    static ThreadLocal<Integer> threadId = ThreadLocal.withInitial(nextId::getAndIncrement);  
  
    // Возвращает уникальный ID текущего потока.  
    static int get() {  
        return threadId.get();  
    }  
}
```

*Как реализовать двусторонний обмен данными между потоками?

Вопрос, который зачастую дается в виде практической задачи. Конечно, результата можно добиться разными способами: парой [атомарных переменных, критическими секциями, потокобезопасными коллекциями](#).

Но полезно знать, что специально для этого случая в стандартной библиотеке `java.util.concurrent` есть простой класс `Exchanger`.

Класс содержит единственный метод `V exchange(V x)`. Один поток передает в него данные, и встает в ожидание. Ожидание завершается, когда второй поток также приходит в метод `exchange` со своей порцией информации. В качестве результата вызова потоки получают данные друг друга.

На основе класса `Exchanger` удобно создавать *пайплайны обработки данных*. Первый поток выполняет свою часть обработки, и складывает результаты в буфер. В качестве буфера может работать любой многоразовый объект-контейнер. Когда он заполняется, следующий поток обменивает его на второй, пустой буфер. Таким образом два буфера используются поочередно, не выделяется лишний раз память и не нагружается [GC](#). Далее из попарно обменивающихся буферами потоков может строиться длинная многопоточная цепочка обработки.

*Какими коллекциями пользоваться в многопоточной среде?

Первый вариант – превратить в синхронизированную обычную коллекцию, вызвав соответствующий ее типу метод `Collections.synchronized*()`. Самый общий и самый примитивный способ, создает обертку с синхронизацией всех операций с помощью [synchronized](#).

Если работа с коллекцией состоит в основном из чтения, [лучшая в плане производительности](#) альтернатива – `CopyOnWriteArrayList`, и содержащий его в реализации `CopyOnWriteArraySet`.

Поток безопасности достигается копированием внутреннего массива при любой модификации, оригинальный массив остается immutable.

Program order достигается модификатором `volatile` на внутреннем массиве.

Третий вариант – использование [Concurrent-коллекций](#):

- Неблокирующие хэш таблицы [ConcurrentSkipListMap](#), [ConcurrentHashMap](#) и [ConcurrentSkipListSet](#) (хэш-таблица в основе реализации)
- Неблокирующие очереди [ConcurrentLinkedQueue](#) и [ConcurrentLinkedDeque](#)
- Большой набор различных блокирующих очередей