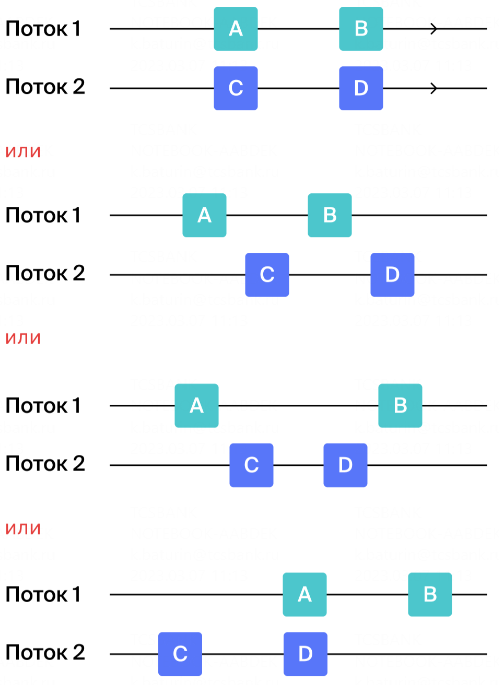
**Поток** – это инструмент, который позволяет выполнять части программы одновременно. Потоки представлены классом ***Thread***.

В рамках одного потока, операции выполняются последовательно. Между двумя потоками операции могут выполняться в любом порядке.



Поток может быть:

* **выполняющимся**
* **готовым к выполнению**, как только он получит время и ресурсы ЦП
* **приостановленным**, т.е. временно не выполняющимся
* **возобновленным** в дальнейшем
* **заблокированным** в ожидании ресурсов для своего выполнения
* **завершенным**, когда его выполнение закончено и не может быть возобновлено

Создать поток можно двумя способами: либо с помощью реализации интерфейса *Runnable*, либо путем создания подкласса класса *Thread*. Один и тот же объект типа *Runnable* можно передавать нескольким потокам.

*Thread thread = new Thread(() -> {* набор действий *}); // с помощью лямбды*

*Runnable runnable = () -> {* набор действий *}; // явно создав Runnable*

*Thread thread = new Thread(runnable);*

*Runnable runnable = () -> { // создав анонимный класс*

*@Override*

*Public void run() {* набор действий *} };*

*Thread thread = new Thread(runnable);*

*Thread thread = new Thread() { // унаследовавшись от класса Thread и переопределив метод run*

*@Override*

*public void run() {* набор действий *} };*

*public class MyLogic implements Runnable { // создав класс реализующий Runnable*

*@Override*

*public void run() {* набор действий *} };*

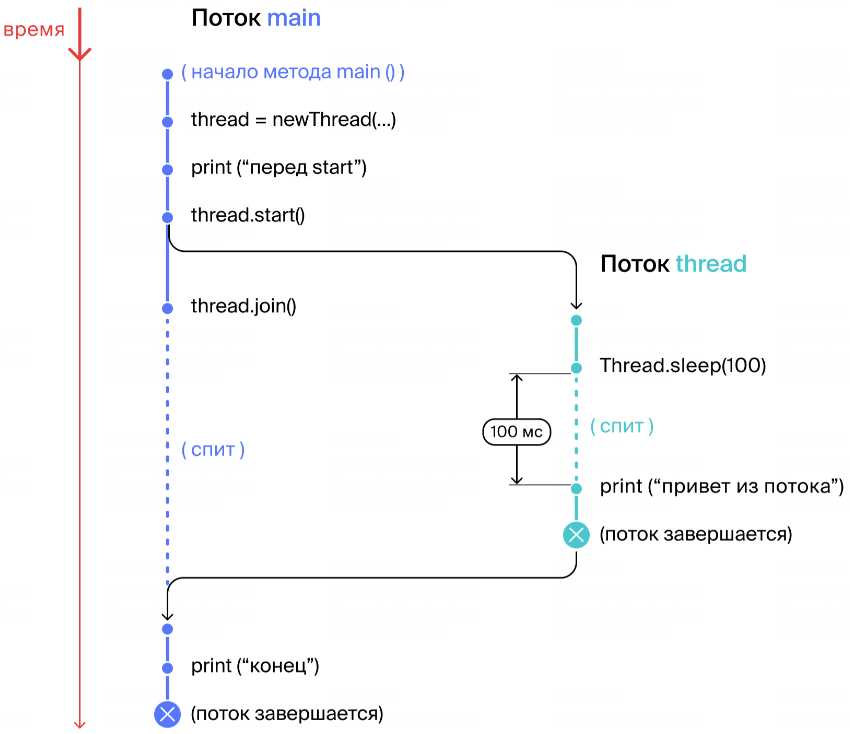
*Thread thread = new Thread(new Mylogic());*

В каждом процессе существует ***main* (основной)** поток. Все остальные подчиненные потоки могут быть порождены им. Программа завершается, когда завершаются все её потоки.

Метод ***thread.start()*** запускает выполнение потока.

Метод ***thread.sleep(100L)*** приостанавливает выполнение потока на 100 милисекунд.

Метод ***thread.join()*** блокирует выполнение потока, из которого был вызван, до завершения указанного потока или ждет указанное время.



*public class EndOfMain {*

*public static void main(String[] args) throws InterruptedException {*

*Thread thread = new Thread(() -> {*

*try { Thread.sleep(100L);*

*} catch (InterruptedException e) {*

*Thread.currentThread().interrupt();*

*return; }*

*System.out.println("Привет из потока"); }); // вторая строка*

*System.out.println("main(): перед thread.start()"); // первая строка*

*thread.start();*

*thread.join();*

*System.out.println("main(): конец"); } } // третья строка*

Метод ***thread.interrupt()*** прерывает выполнение любого блокирующего метода в потоке (например, *sleep()*) и заставляет прерванный метод выбросить исключение ***InterruptedException***. По факту этот метод устанавливает флаг *interrupt* в положение *true*. Блокирующие методы видят этот флаг, прекращают блокировку и устанавливают флаг в положение *false*. После этого поток доходит до логического завершения. Для корректного использования цикла *while* рекомендуется следующая конструкция:

*Thread thread = new Thread(() -> {*

*while (!Thread.currentThread().isInterrupted()) {*

*System.out.println("SPAM"); } });*

Метод ***thread.stop()*** принудительно прерывает выполнение потока. Не рекомендован к использованию, т.к. могут повредиться данные.

С точки зрения процессора операции инкремента/декремента – это три атомарные операции (прочитать, изменить, записать), которые могут перемешиваться в режиме многопоточности. Для работы с многопоточностью используется популярный пакет ***java.util.concurrent***.

Этот пакет содержит классы ***AtomicInteger***, ***AtomicLong***, ***AtomicReference*** и другие классы для корректной работы в режиме многопоточности. Конкретно для инкремента/декремента используются методы ***AtomicInteger.getAndIncremet()*** и ***AtomicInteger.getAndDecremet()***.

Сделать операцию атомарной можно с помощью оператора ***synchronized***. Этот оператор защищает куски кода от одновременного выполнения разными потоками, создавая **однопоточную критическую секцию**. Критические секции могут быть сгруппированы с одним и тем же объектом-монитором. Он не может быть примитивом и гарантирует последовательность выполнения секций с одним и тем же объектом-монитором. Объявить метод синхронизированным можно несколькими способами:

* объявить при описании метода:

*synchronized int sumArray(int nums[]) { … }*

* вызвать метода внутри блока кода, объявленного как синхронизированный:

*synchronized(ссылка\_на\_объект-монитор) { // синхронизируемые инструкции }*

*Synchronized* может значительно замедлить выполнение программы в отличии от *Atomic*, но является универсальным средством.

*public class IncDecThreadsSynchronized {*

*private static final int N = 100\_000\_000;*

*private static int x = 0;*

*public static void main(String[] args) throws InterruptedException {*

*Object monitor = new Object();*

*Thread incThread = new Thread(() -> {*

*for (int i = 0; i < N; i++) {*

*synchronized (monitor) {*

*x++; } } });*

*Thread decThread = new Thread(() -> {*

*for (int i = 0; i < N; i++) {*

*synchronized (monitor) {*

*x--; } } });*

*incThread.start();*

*decThread.start();*

*incThread.join();*

*decThread.join();*

*System.out.println("x = " + x); } } // в результате всегда выведет x = 0*

Обычные реализации коллекций не поддерживают работу с многопоточностью. Вместо них нужно использовать специальные версии из пакета *java.util.concurrent*. Например, ***ConcurrentSkipListSet***, ***ConcurrentHashMap*** и другие.

*public class TwoThreadsAndSet {*

*private static final int N = 100\_000;*

*public static void main(String[] args) throws InterruptedException {*

*// отличие от предыдущей программы в этой строке:*

*Set<Integer> set = new ConcurrentSkipListSet<>();*

*for (int i = 0; i < N; i++) {*

*set.add(i); }*

*Thread oddThread = new Thread(() -> {*

*for (int i = 0; i < N; i++) {*

*if (i % 2 == 1) {*

*set.remove(i); } } });*

*Thread evenThread = new Thread(() -> {*

*for (int i = 0; i < N; i++) {*

*if (i % 2 == 0) {*

*set.remove(i); } } });*

*oddThread.start();*

*evenThread.start();*

*oddThread.join();*

*evenThread.join();*

*System.out.printf("set.size() = %d%n", set.size()); } } // в результате всегда выведет set.size() = 0*

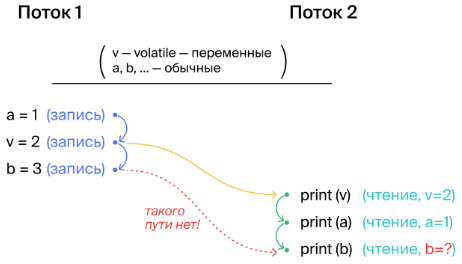
**Взаимная блокировка потоков (*deadlock*)** – это одна из частых причин зависания программ, когда разные потоки вечно ждут друг друга. Другие причины – это бесконечные циклы, нехватка памяти и медленные операции ввода-вывода.

Чтобы избежать взаимных блокировок, можно договориться в каком порядке все потоки производят захваты, или не использовать общие объекты-мониторы.

По умолчанию *Java* не гарантирует, что значение, присвоенное в одном потоке, будет прочитано в другом потоке. Для исправления этой проблемы используется ключевое слово ***volatile*** при объявлении переменной. Оно сообщает *Java*, что эта переменная используется несколькими потоками и все потоки должны видеть актуальное значение переменной.

**Правило видимости значений** в упрощенном виде:

* Значения, записанные в рамках одного потока, видны последующим операциям в том же потоке.
* *volatile*-переменные добавляют стрелку от операции записи к последующим операциям чтения, даже если операции выполнены в разных потоках.
* Если от операции записи до операции чтения можно дойти стрелками любого цвета, то операция чтения гарантировано получит записанное значение (*print(a)*).
* Если по стрелкам пройти нельзя, то получение записанного значения не гарантировано.



Если в приложении нет общих изменяемых переменных, то нет и связанных с ними проблем – области видимости и гонки потоков.

**Пул** – это ограниченный набор ресурсов, которые можно использовать многократно. Для работы с пулом потоков в *Java* используется интерфейс ***ExecutorService***. Он сам подбирает свободный поток из пула и передает ему задачу на выполнение. Задачи представлены интерфейсов *Runnable*, как при создании потоков. *ExecutorService* можно создать с помощью статических методов класса ***Executors***.

К задачам, выполняющимся в пуле потоков, применимы те же правила, что и к задачам, запущенным с использованием *Thread* (гонка данных, области видимости и т.д.).

Внутри пула потоков все добавленные задачи сохраняются в очередь и берутся в той же последовательности (первый зашел – первый вышел), но могут выполняться в другом порядке.

*public class ThreadPoolExample {*

*public static void main(String[] args) throws InterruptedException {*

*List<Runnable> tasks = new ArrayList<>();*

*for (int i = 1; i <= 100\_000; i++) {*

*final int copyOfI = i; // нужно для лямбда-выражения*

*Runnable task = () -> {*

*System.out.println("Задача #" + copyOfI); };*

*tasks.add(task); }*

*ExecutorService executor = Executors.newFixedThreadPool(10); // создаем пул из 10 потоков*

*for (Runnable task : tasks) {*

*executor.execute(task); }*

*executor.shutdown();*

*while (!executor.awaitTermination(1, TimeUnit.SECONDS)) { // ежесекундный цикл проверки*

*/\* пусто \*/ } } }*

Для передачи данных между потоками могут использоваться интерфейс многопоточной очереди ***BlockingQueue*** из пакета *java.util.concurrent*. Если очередь переполнена, то ***put()*** будет ждать, пока в очереди появится место. Есть очередь пустая, то ***take()*** будет ждать, пока в очередь что-нибудь добавят. Эти методы блокирующие друг друга.

*public class ProducerAConsumerB {*

*private static final int N = 4;*

*private static final int QUEUE\_SIZE = 2;*

*private static final int POISON\_PILL = -1;*

*public static void main(String[] args) throws InterruptedException {*

*BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(QUEUE\_SIZE);*

*Thread threadA = new Thread(() -> {*

*try {*

*for (int i = 1; i <= N; i++) {*

*queue.put(i);*

*System.out.println("A: добавил " + i + " в очередь"); }*

*queue.put(POISON\_PILL); // объект-флаг о завершении передачи*

*System.out.println("A: добавил " + POISON\_PILL);*

*} catch (InterruptedException e) {*

*Thread.currentThread().interrupt(); } });*

*Thread threadB = new Thread(() -> {*

*try {*

*while (true) {*

*int i = queue.take();*

*System.out.println("B: получил " + i);*

*if (i == POISON\_PILL) {*

*System.out.println("B: конец");*

*break; } }*

*} catch (InterruptedException e) {*

*Thread.currentThread().interrupt(); } });*

*threadA.start();*

*threadB.start();*

*threadA.join();*

*threadB.join();*

*System.out.println("main(): конец"); } }*

Более подробно почитать про многопоточность можно [тут](https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html).

Для координации синхронизированных потоков между собой используются методы ***wait()***, ***notify()*** и ***notifyAll()***. Метод *wait()* переводит поток в режим ожидания до поступления уведомлений или на указанный промежуток времени (рекомендовано использовать в цикле с условием). Методы *notify()* и *notifyAll()* возобновляет выполнение одного или всех ожидающих потоков.

Механизмы приостановки, возобновления и остановки потоков реализуются соответствующими методами класса Thread – ***suspend()***, ***resume()*** и ***stop()***. Однако пользоваться ими напрямую не рекомендуется в связи с возможностью взаимоблокировки всех потоков. Лучше использовать эти методы через логические переменные-флаги.