

# Fork-Join.

# Объекты синхронизации

ПРЕПОДАВАТЕЛЬ



# Юрий Костяной

**Java/Kotlin backend-разработчик**

- 3+ года опыта в коммерческой разработке
- 2+ года опыта в преподавании
- Проекты по интеграции сторонних платформ, CRM
- Проблемно-ориентированный подход в преподавании



# ВАЖНО:

- Камера должна быть включена на протяжении всего занятия.
- Если у Вас возник вопрос в процессе занятия, пожалуйста, поднимите руку и дождитесь, пока преподаватель закончит мысль и спросит Вас, также можно задать вопрос в чате или когда преподаватель скажет, что начался блок вопросов.
- Организационные вопросы по обучению решаются с кураторами, а не на тематических занятиях.
- Вести себя уважительно и этично по отношению к остальным участникам занятия.
- Во время занятия будут интерактивные задания, будьте готовы включить камеру или демонстрацию экрана по просьбе преподавателя.

# ПЛАН ЗАНЯТИЯ

1. Повторение
2. Основной блок
3. Вопросы по основному блоку
4. Домашняя работа

1

# ПОВТОРЕНИЕ ИЗУЧЕННОГО

# Повторение

Какой из следующих методов ExecutorService используется для подачи задачи на выполнение и получения Future, представляющего результат выполнения этой задачи?

1. `submit(Callable<T> task)`
2. `execute(Runnable task)`
3. `invokeAll(Collection<? extends Callable<T>> tasks)`
4. `invokeAny(Collection<? extends Callable<T>> tasks)`



# Повторение

Какой из следующих методов ExecutorService используется для подачи задачи на выполнение и получения Future, представляющего результат выполнения этой задачи?

1. `submit(Callable<T> task)`
2. `execute(Runnable task)`
3. `invokeAll(Collection<? extends Callable<T>> tasks)`
4. `invokeAny(Collection<? extends Callable<T>> tasks)`

Метод `submit(Callable<T> task)` подходит для подачи задачи на выполнение и возвращает Future, который можно использовать для получения результата выполнения задачи.

# Повторение

В чем основная разница между интерфейсами Callable и Runnable в Java?

1. Callable не может использоваться в ExecutorService, тогда как Runnable может.
2. Runnable может использоваться для асинхронного выполнения, тогда как Callable только для синхронного.
3. Runnable не может возвращать результат, тогда как Callable может.
4. Они идентичны и могут использоваться взаимозаменяемо.





# Повторение

В чем основная разница между интерфейсами Callable и Runnable в Java?

1. Callable не может использоваться в ExecutorService, тогда как Runnable может.
2. Runnable может использоваться для асинхронного выполнения, тогда как Callable только для синхронного.
3. Runnable не может возвращать результат, тогда как Callable может.
4. Они идентичны и могут использоваться взаимозаменяемо.

Runnable не возвращает результат выполнения задачи, в то время как Callable возвращает результат и может бросать проверяемые исключения.

# Повторение

Какие из утверждений о Lock в Java верны?

1. Lock предоставляет только блокировку на чтение.
2. Lock обеспечивает простейший способ синхронизации многопоточных операций.
3. Lock предоставляет более гибкую и тонкую управляемость блокировкой, чем ключевое слово `synchronized`.
4. Lock является частью стандартной библиотеки Java только начиная с версии 11.

# Повторение

Какие из утверждений о Lock в Java верны?

1. Lock предоставляет только блокировку на чтение.
2. Lock обеспечивает простейший способ синхронизации многопоточных операций.
3. Lock предоставляет более гибкую и тонкую управляемость блокировкой, чем ключевое слово `synchronized`.
4. Lock является частью стандартной библиотеки Java только начиная с версии 11.

Lock предоставляет более сложные возможности управления блокировкой, чем ключевое слово `synchronized`, например, возможность использования условий, неявная поддержка нескольких условий и т.д.

# Повторение

Какие из следующих методов Future предоставляют возможность проверки завершения задачи и получения ее результата?

1. `get()`
2. `isDone()`
3. `cancel()`
4. `await()`



# Повторение

Какие из следующих методов Future предоставляют возможность проверки завершения задачи и получения ее результата?

1. `get()`
2. `isDone()`
3. `cancel()`
4. `await()`

Метод `isDone()` возвращает `true`, если задача завершена, в противном случае - `false`. Метод `get()` используется для получения результата задачи.

# Повторение

Какой метод ReentrantLock используется для освобождения блокировки?

1. `release()`
2. `unlock()`
3. `free()`
4. `open()`



# Повторение

Какой метод ReentrantLock используется для освобождения блокировки?

1. `release()`
2. `unlock()`
3. `free()`
4. `open()`



# Повторение

Зачем используется объект Condition в Java?

- Для ожидания возможности записи в файл.
- Для организации условных переменных и управления потоками.
- Для определения типов данных в программе.
- Для создания новых потоков.





# Повторение

Зачем используется объект Condition в Java?

- Для ожидания возможности записи в файл.
- Для организации условных переменных и управления потоками.
- Для определения типов данных в программе.
- Для создания новых потоков.

Объект Condition используется совместно с ReentrantLock для организации условных переменных, которые позволяют точно управлять потоками и их выполнением в зависимости от определенных условий.

# Повторение

Какой метод `ScheduledExecutorService` используется для планирования выполнения задачи с фиксированной задержкой между завершением предыдущей задачи и началом следующей?

1. `schedule(Callable<V> callable, long delay, TimeUnit unit)`
2. `scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)`
3. `scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)`
4. `execute(Runnable command)`

# Повторение

Какой метод `ScheduledExecutorService` используется для планирования выполнения задачи с фиксированной задержкой между завершением предыдущей задачи и началом следующей?

1. `schedule(Callable<V> callable, long delay, TimeUnit unit)`
2. `scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)`
3. `scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)`
4. `execute(Runnable command)`

# Повторение

Что будет выведено в консоль?

```
public class ThreadLocalExample {  
    private static final ThreadLocal<Integer> threadLocal  
    = new ThreadLocal<>();  
  
    public static void main(String[] args) {  
        threadLocal.set(42);  
        Integer value1 = threadLocal.get();  
        System.out.println("Main Thread - Value: " + value1);  
        Thread thread1 = new Thread(() -> {  
            Integer value = threadLocal.get();  
            System.out.println("Thread 1 - Value: " + value);  
        });  
        Thread thread2 = new Thread(() -> {  
            Integer value = threadLocal.get();  
            System.out.println("Thread 2 - Value: " + value);  
        });  
        thread1.start();  
        thread2.start();  
    }  
}
```

# Повторение

Что будет выведено в консоль?

Main Thread - Value: 42

Thread 2 - Value: null

Thread 1 - Value: null

Каждый поток имеет свое собственное значение ThreadLocal, и если значение не было установлено для конкретного потока, оно будет равно null.

```
public class ThreadLocalExample {  
    private static final ThreadLocal<Integer> threadLocal  
    = new ThreadLocal<>();  
  
    public static void main(String[] args) {  
        threadLocal.set(42);  
        Integer value1 = threadLocal.get();  
        System.out.println("Main Thread - Value: " + value1);  
        Thread thread1 = new Thread(() -> {  
            Integer value = threadLocal.get();  
            System.out.println("Thread 1 - Value: " + value);  
        });  
        Thread thread2 = new Thread(() -> {  
            Integer value = threadLocal.get();  
            System.out.println("Thread 2 - Value: " + value);  
        });  
        thread1.start();  
        thread2.start();  
    }  
}
```

# Повторение

Что будет выведено в консоль?

```
public class ThreadLocalExample1 {  
    private static ThreadLocal<Integer> threadLocal =  
    new ThreadLocal<Integer>() {  
        @Override  
        protected Integer initialValue() {  
            return 99;  
        }  
    };  
  
    public static void main(String[] args) {  
        Thread thread1 = new Thread(() -> {  
            System.out.println("Thread 1 - Value: " + threadLocal.get());  
        });  
        Thread thread2 = new Thread(() -> {  
            System.out.println("Thread 2 - Value: " + threadLocal.get());  
        });  
        thread1.start();  
        thread2.start();  
    }  
}
```

# Повторение

Что будет выведено в консоль?

Thread 2 - Value: 99

Thread 1 - Value: 99

Метод `initialValue()` позволяет установить начальное значение `ThreadLocal` для каждого потока. Если значение не установлено явно, будет использовано значение, возвращаемое методом `initialValue()`.

```
public class ThreadLocalExample1 {  
    private static ThreadLocal<Integer> threadLocal =  
    new ThreadLocal<Integer>() {  
        @Override  
        protected Integer initialValue() {  
            return 99;  
        }  
    };  
  
    public static void main(String[] args) {  
        Thread thread1 = new Thread(() -> {  
            System.out.println("Thread 1 - Value: " + threadLocal.get());  
        });  
        Thread thread2 = new Thread(() -> {  
            System.out.println("Thread 2 - Value: " + threadLocal.get());  
        });  
        thread1.start();  
        thread2.start();  
    }  
}
```

# Повторение

В чём прикол мема?



```
Executors.  
newSingleThreadExecutor();
```



```
Executors  
.newFixedThreadPool(1);
```



2

# ОСНОВНОЙ БЛОК

# Введение

- Разделяй и властвуй
- Только время принадлежит нам



# Проблема

Многие алгоритмы класса «разделяй и властвуй» (divide and conquer) - сортировка слиянием, быстрая сортировка и др. - просто идеально подходят для многопоточности.

*Неужели в Java нет подходящей автоматизации для таких алгоритмов?*



Разделяй и властвуй

# Разделяй и властвуй

## Фреймворк `fork/join`

В Java 7 был добавлен [fork/join framework](#). Он предоставляет средства, позволяющие ускорить параллельные процессы с помощью использования всех доступных в системе процессоров.

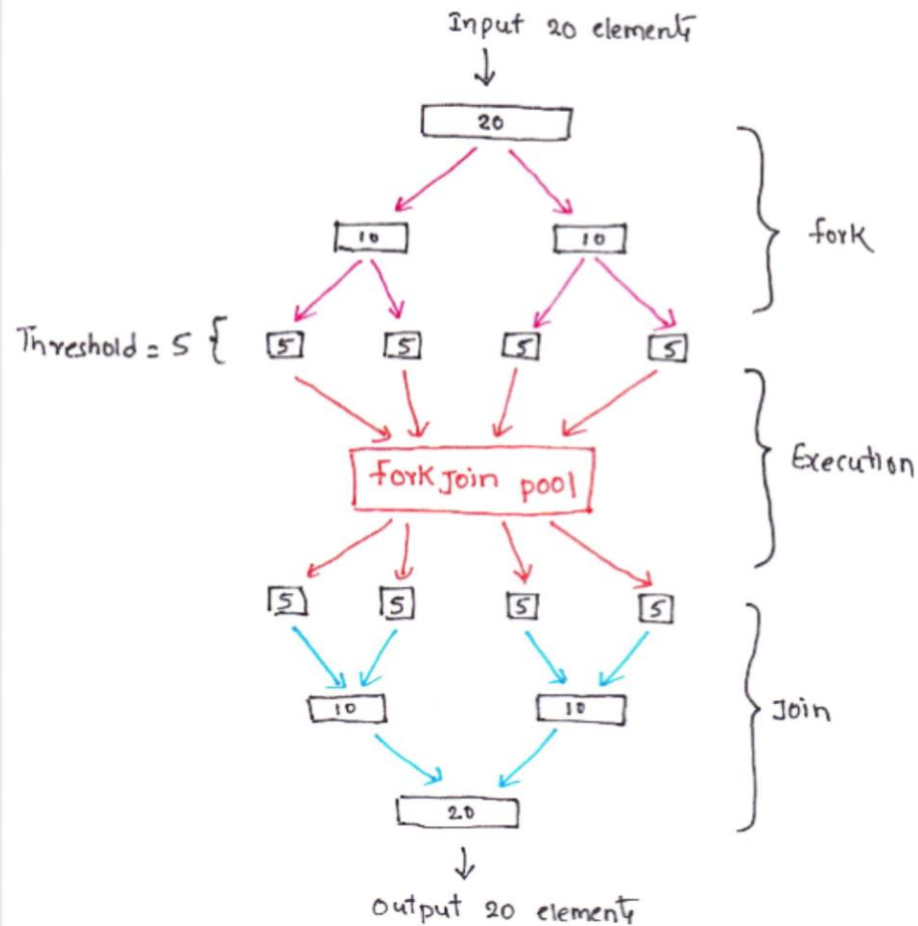


# Разделяй и властвуй

## Фреймворк fork/join

В основе фреймворка лежит подход «разделяй и властвуй».

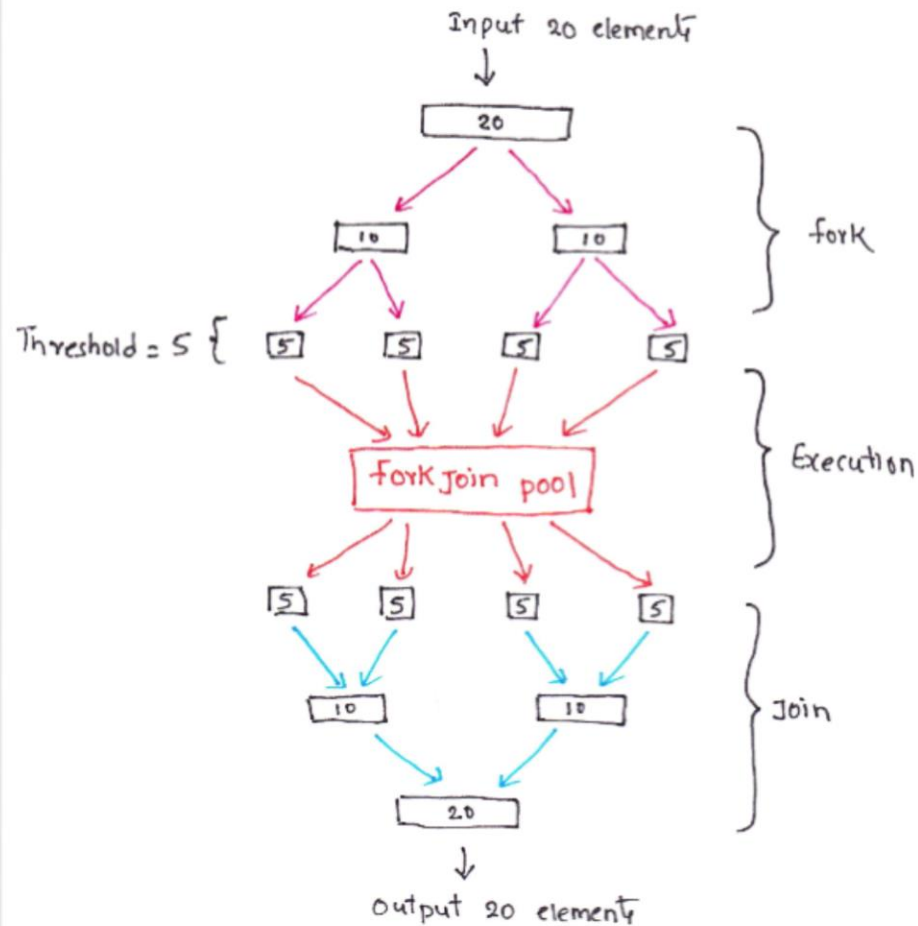
На практике фреймворк сначала разделяет большую задачу на набор подзадач (*fork*). Это повторяется рекурсивно до тех пор, пока задача не станет примитивной, т.е. достаточно простой для выполнения асинхронно.



# Разделяй и властвуй

## Фреймворк fork/join

Затем начинается часть рекурсивного объединения промежуточных результатов в конечный результат (*join*). Когда результат задачи не может быть представлен в виде значения (метод возвращает *void*) фреймворк просто ждёт, пока завершится выполнение всех подзадач.



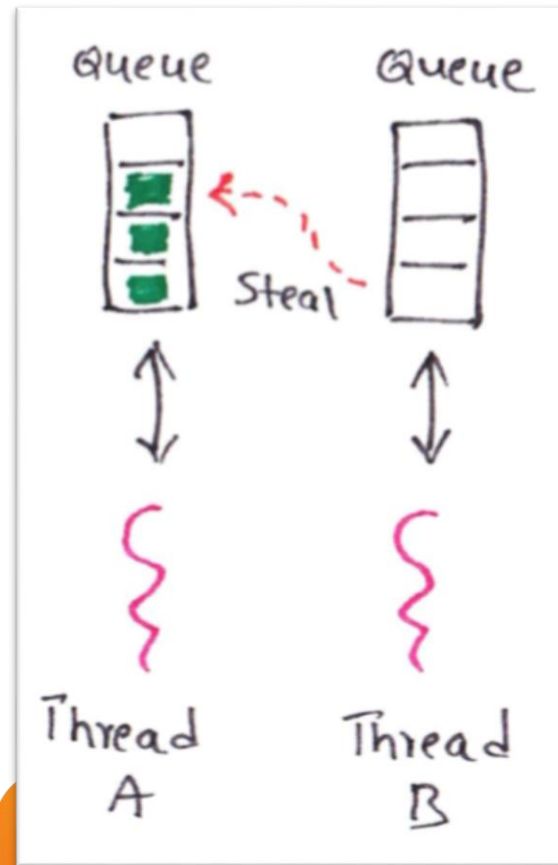
# Разделяй и властвуй

## ForkJoinPool

Чтобы обеспечить эффективное параллельное выполнение fork/join framework использует пул потоков *ForkJoinPool*.

**ForkJoinPool** – имплементация интерфейса *ExecutorService*, управляющая потоками *ForkJoinWorkerThread* и позволяющая получать информацию о состоянии потока и производительности.

Поток *ForkJoinWorkerThread* может выполнять лишь одну задачу за раз, но *ForkJoinPool* не создаёт отдельный поток для каждой из подзадач. Вместо этого каждый поток в пуле имеет свою двунаправленную очередь (deque), которая хранит задачи. Эта архитектура позволяет балансировать нагрузку между потоками благодаря алгоритму *кражи работы* (work-stealing algorithm).



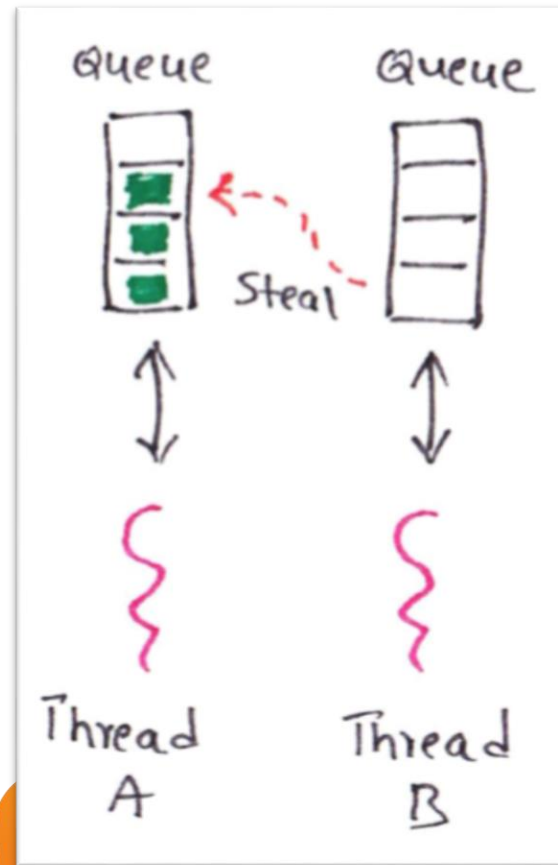
# Разделяй и властвуй

## ForkJoinPool

Чтобы обеспечить эффективное параллельное выполнение fork/join framework использует пул потоков *ForkJoinPool*.

**ForkJoinPool** – имплементация интерфейса *ExecutorService*, управляющая потоками *ForkJoinWorkerThread* и позволяющая получать информацию о состоянии потока и производительности.

Поток *ForkJoinWorkerThread* может выполнять лишь одну задачу за раз, но *ForkJoinPool* не создаёт отдельный поток для каждой из подзадач. Вместо этого каждый поток в пуле имеет свою двунаправленную очередь (*deque*), которая хранит задачи. Эта архитектура позволяет балансировать нагрузку между потоками благодаря алгоритму *кражи работы* (work-stealing algorithm).





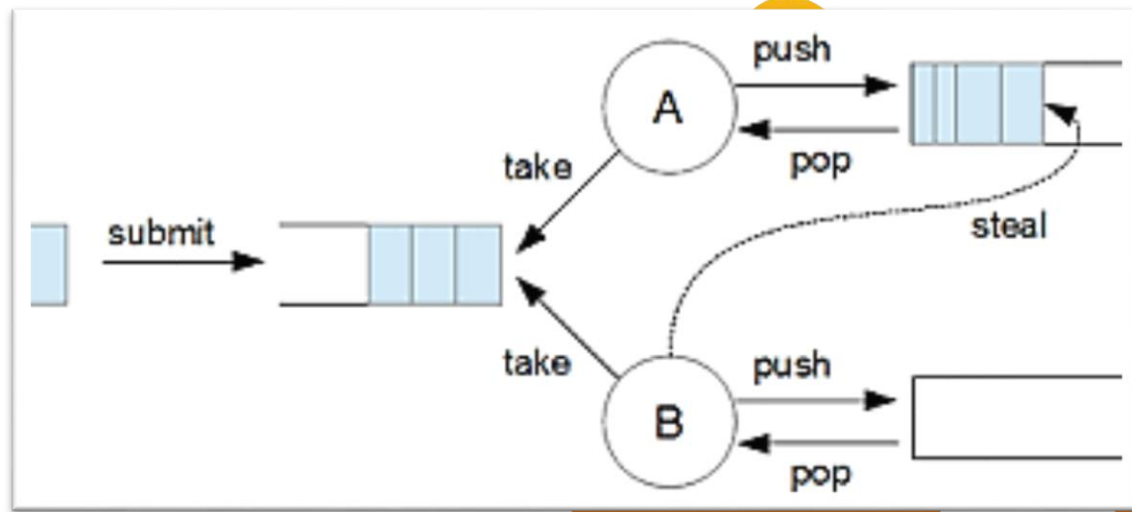
# Разделяй и властвуй

## Алгоритм кражи работы

Каждый освободившийся поток, не найдя задач в своём дэке, пытается взять такую задачу в дэке другого потока. Поток всегда берёт задачи из головы своего дэка или из хвоста чужого дека, либо из глобальной (общей) очереди задач. При этом легко определить, где больше всего задач находится в текущий момент времени.

Такой подход

- минимизирует вероятность борьбы потоков за задачи;
- сокращает время поиска задачи освободившимся потоком, т.к. всегда задача берётся из той очереди, где задач больше всего.



Разделяй и властвуй

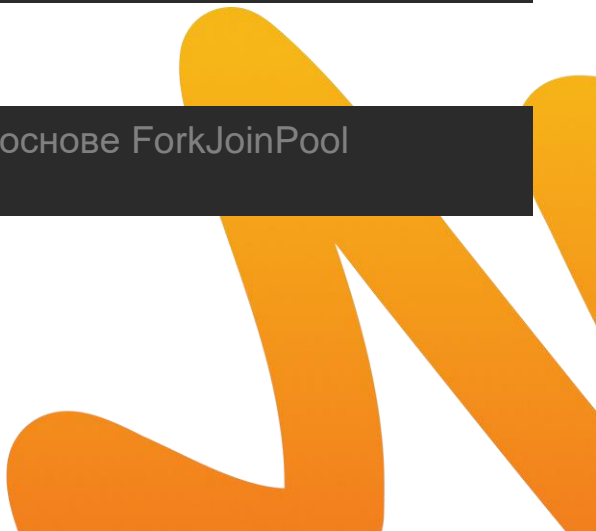
# Создание ForkJoinPool

```
// Рекомендуемый способ создания объекта ForkJoinPool  
ForkJoinPool commonPool = ForkJoinPool.commonPool();
```

```
// Если требуется создать настраиваемый ForkJoinPool, можно воспользоваться конструктором  
// Он позволяет задать уровень распараллеливания (количества ядер), фабрику потоков и  
// обработчика исключений
```

```
ForkJoinPool forkJoinPool = new ForkJoinPool(2); // используем 2 ядра процессора
```

```
// В Java 8 появилась возможность создавать сервис выполнения на основе ForkJoinPool  
ExecutorService service = Executors.newWorkStealingPool(10);
```



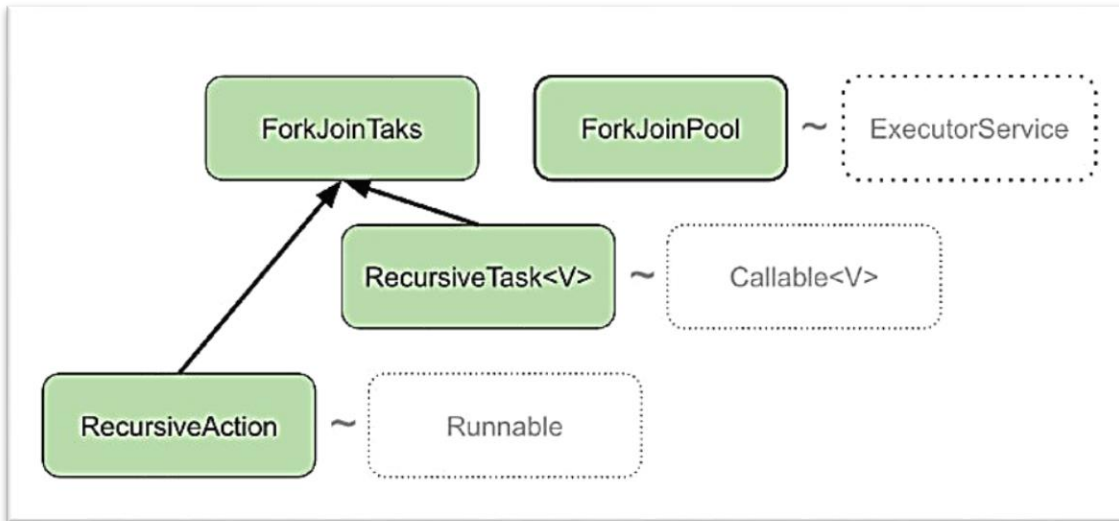
# Разделяй и властвуй

## ForkJoinTask

**ForkJoinTask<V>** - это основной тип задачи, выполняемой в *ForkJoinPool*. Для создания задачи необходимо расширить одного из наследников ForkJoinTask:

- **RecursiveAction** – расширяем его для выполнения задач, не возвращающих результат (void).
- **RecursiveTask<V>** – расширяем его для задач, возвращающих результат.

При этом у обоих классов должен быть переопределён метод *compute()* (аналог run или call).



Метод *join()* у *RecursiveAction* возвращает *null*, у for *RecursiveTask<V>* - результат.

# Разделяй и властвуй

## Метод `compute()`

Общий шаблон реализации метода `compute()`:

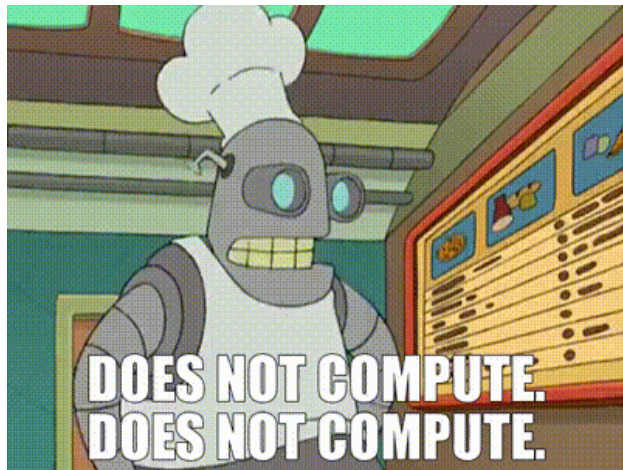
```
If(problemSize < threshold) {  
    // решаем задачу напрямую (без параллелизма)  
} else {  
    // задача разбивается на подзадачи по критерию threshold  
    // рекурсивно решаем каждую подзадачу  
    // объединяем результаты подзадач  
}
```

Для вычисления *threshold* можно воспользоваться формулой:

$$threshold = N / (L * CPU),$$

где N – размер задачи, CPU – количество доступных ядер процессора, L – load factor (количество задач на поток). L обычно принимают 8 или 16.

Хорошая практика, когда  $100 < threshold < 10\_000$



# Разделяй и властвуй

## Передача задач в ForkJoinPool

Передать задачи в *ForkJoinPool* можно несколькими способами:

- методы *submit()* или *execute()*

```
forkJoinPool.execute(customRecursiveTask);  
int result = customRecursiveTask.join();
```

- метод *invoke()* разбивает задачу и ждёт результата, не требуя вызова метода *join()*.

```
int result = forkJoinPool.invoke(customRecursiveTask);
```

- метод *invokeAll()* позволяет направить набор задач (две и более или коллекцию), делит задачу на подзадачи и затем возвращает результат в виде коллекции объектов *Future* в порядке их появления.
- методы *fork()* и *join()*. *fork()* отправляет задачу в пул, но выполнение начинается лишь в момент вызова *join()*.

```
customRecursiveTaskFirst.fork();  
int result = customRecursiveTaskLast.join();
```



Разделяй и властвуй

# Пример использования ForkJoinPool



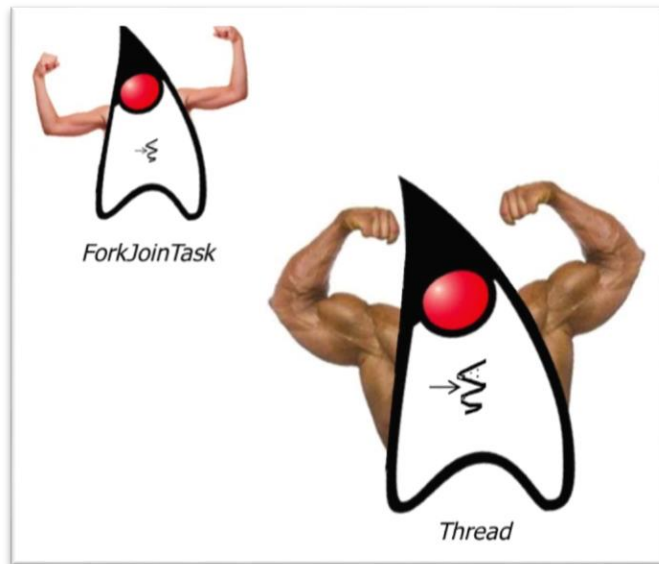
ThreadForkJoinExample.zip



## Разделяй и властвуй

# Особенности применения ForkJoinPool

- *ForkJoinTask* более легковесны, чем классические потоки, т.е. требуют меньше ресурсов в момент запуска.
- Метод *compute()* можно вызвать для второй подзадачи *ForkJoinTask* вместо *fork()*, чтобы использовать текущий поток. Для *ForkJoinAction* лучше вызвать *invokeAll()*.
- *ForkJoinPool* сложная и тяжёлая структура, поэтому их количество в приложении должно быть минимально. В идеале – один.
- Старайтесь использовать стандартны пул, получаемы с помощью *ForkJoinPool.commonPool()*.
- Постарайтесь определить оптимальный критерий разбиения задачи на подзадачи.
- Избегайте блокировок в *ForkJoinTasks*.



# Задание

Представьте, что у вас есть сеть магазинов, и вы хотите рассчитать общую выручку за определенный период времени. Каждый магазин предоставляет данные о продажах в виде массива. Вам нужно эффективно подсчитать общую сумму продаж для всех магазинов с использованием параллельных вычислений.

Шаги:

1. Создайте класс, расширяющий `RecursiveTask<Long>`, представляющий задачу подсчета выручки для определенного магазина.
2. В конструкторе класса укажите массив продаж и границы массива для текущего магазина.
3. В методе `compute()` вычислите общую сумму продаж для текущего магазина. Если количество продаж невелико, вычислите сумму напрямую.
4. Если количество продаж больше порогового значения, разделите задачу на две подзадачи для параллельного выполнения.
5. Дождитесь завершения подзадач и объедините их результаты для получения общей выручки.





# Проблема

Синхронизация в Java реализуется использованием зарезервированного слова *synchronized*. Когда какой-либо поток начинает использовать общий для всех потоков объект, то он проверяет монитор (мьютекс) этого объекта. Если мьютекс свободен, то поток блокирует его, помечая как занятый, и приступает к использованию данного ресурса. После завершения работы, поток разблокирует мьютекс (помечает свободным). Если же поток обнаруживает, что объект заблокирован, то он «засыпает» в ожидании освобождения мьютекса. При освобождении мьютекса ожидающий поток тут же заблокирует его и приступит к работе.

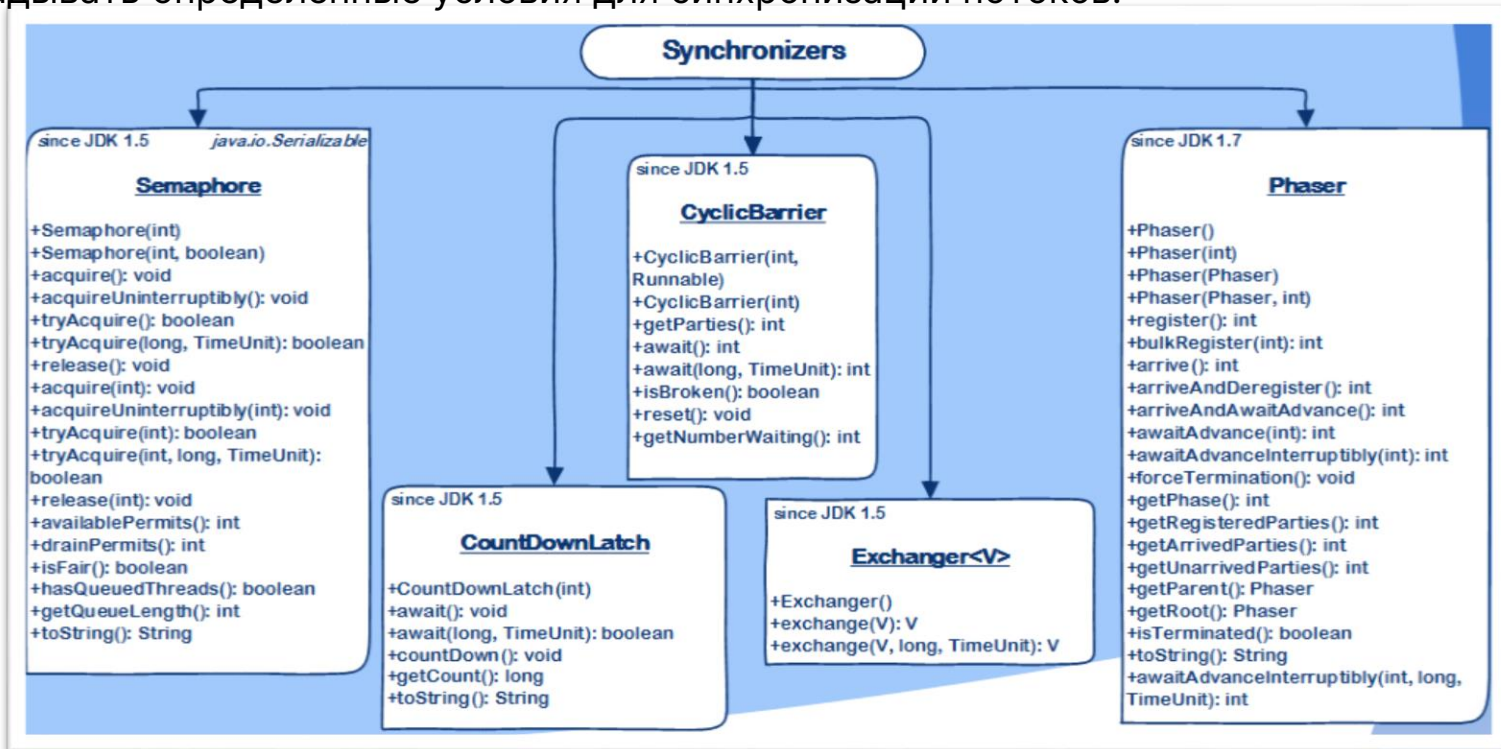
*Как быть, если несколько потоков ожидают освобождения мьютекса? Должен ли следующим получить ресурс тот поток, который пришёл за ним первым? Как это организовать?*



# Только время принадлежит нам

# Synchronizers

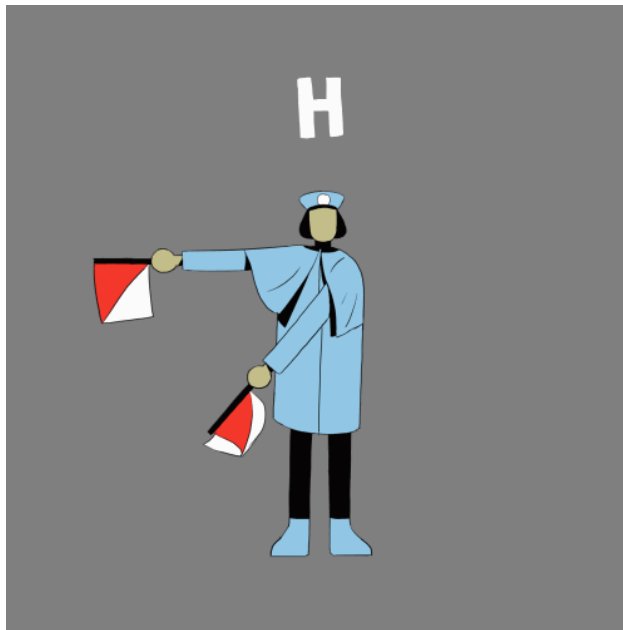
Пакет *java.util.concurrent* содержит пять объектов синхронизации, позволяющих накладывать определенные условия для синхронизации потоков.



# Только время принадлежит нам

# Semaphore

**Semaphore** – объект синхронизации, ограничивающий количество потоков, которые могут «войти» в заданный участок кода, иначе говоря, к общему ресурсу, в качестве которого могут выступать программные/аппаратные ресурсы или файловая система.

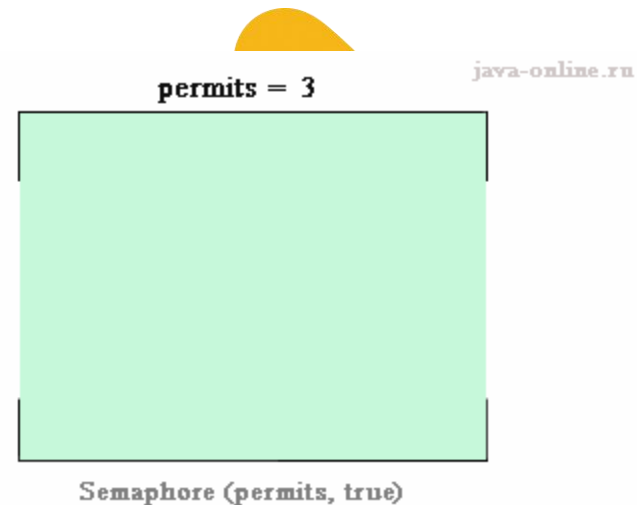


# Только время принадлежит нам

## Semaphore

Для управления доступом к общему ресурсу *Semaphore* использует счетчик. Если значение счетчика больше нуля, то поток исполнения получает разрешение, после чего значение счетчика семафора уменьшается на единицу. При значении счетчика равным нулю очередному потоку исполнения в доступе будет отказано, и он будет заблокирован до тех пор, пока не будут освобождены ресурсы.

Как только один из потоков исполнения освободит ресурсы, т.е. завершит исполнение определенного участка кода, то значение счетчика семафора увеличивается на единицу. Если в это время имеется ожидающий разрешения другой поток исполнения, то он сразу же его получает.



# Только время принадлежит нам

# Конструкторы Semaphore

Конструкторы:

```
Semaphore(int permits);  
Semaphore(int permits, boolean fair);
```

*permits* определяет исходное значение счетчика разрешений, т.е. количество потоков исполнения, которым может быть одновременно предоставлен доступ к общему ресурсу. По умолчанию ожидающим потокам предоставляется разрешение в неопределенном порядке.

Если же использовать второй конструктор и параметру справедливости *fair* присвоить значение *true*, то разрешения будут предоставляться ожидающим потокам исполнения в том порядке, в каком они его запрашивали.

# Только время принадлежит нам

## Методы Semaphore

*Получение разрешения владения ресурсом:*

```
void acquire() throws InterruptedException  
void acquire(int number) throws InterruptedException
```

Первый метод запрашивает одно разрешение, второй – *number* разрешений. Если разрешение не будет получено, то исполнение вызывающего потока будет приостановлено до тех пор, пока не будет получено разрешение, т.е. поток блокируется.

*Освобождение ресурса*

```
void release()  
void release(int number)
```

В первом методе освобождается одно разрешение, а во втором – количество разрешений, обозначенное параметром *number*.

# Только время принадлежит нам

## Пример Semaphore



ThreadSynchronizersExample.zip



# Задание

Есть файловый ресурс, который может обрабатывать ограниченное количество одновременных запросов. Используйте Semaphore, чтобы ограничить количество потоков, которые могут одновременно обращаться к файловому ресурсу.





# Только время принадлежит нам

## CountDownLatch

**CountDownLatch** («защелка с обратным отсчетом») – 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.

Необходимо отметить, что счетчик одноразовый и не может быть инициализирован по-новому.

*Примером CountDownLatch может служить экскурсовод, собирающий группу из заданного количества туристов. Как только группа собрана она отправляется на экскурсию.*



# Только время принадлежит нам

## CountDownLatch

Таким образом, *CountDownLatch* также, как и *Semaphore*, работает со счетчиком, обнуление которого снимает самоблокировки выполняемых потоков. Конструктор *CountDownLatch*:

```
CountDownLatch(int number)
```

*number* определяет количество событий, которые должны произойти до того момента, когда будет снята самоблокировка. Методы:

```
// методы самоблокировки
// ожидание длится до тех пор, пока счетчик CountDownLatch
// не достигнет нуля
void await() throws InterruptedException;
// ожидание длится только в течение определенного периода
// времени, определяемого параметром ожидания wait.
boolean await(long wait, TimeUnit unit) throws
InterruptedException;

// уменьшить счетчик объекта CountDownLatch
void countDown();
```

java-online.ru



CountDownLatch (counter=5)

# Только время принадлежит нам

## Пример CountdownLatch



ThreadSynchronizersExample.zip



# Задание

Есть несколько задач, которые выполняются параллельно, и основной поток должен подождать, пока все эти задачи завершат выполнение. Используйте `CountDownLatch` для синхронизации.



ThreadCountDownLatchTask.zip



# Только время принадлежит нам

## CyclicBarrier

**CyclicBarrier** – объект синхронизации, применяемый, как правило, в распределённых вычислениях. Барьерная синхронизация останавливает участника (исполняемый поток) в определенном месте в ожидании прихода остальных потоков группы. Как только все потоки достигнут барьера, барьер снимается и выполнение потоков продолжается.

Циклический барьер *CyclicBarrier*, также, как и *CountDownLatch*, использует счетчик и похож на него. Отличие связано с тем, что «защелку» нельзя использовать повторно после того, как её счётчик обнулится, а барьер можно использовать (в цикле).



# Только время принадлежит нам

## CyclicBarrier

При барьерной синхронизации алгоритм расчета делят на несколько потоков. С помощью барьера организуют точку сбора частичных результатов вычислений, в которой подводится итог этапа вычислений.

Конструкторы *CyclicBarrier*:

```
// задается количество потоков, которые должны
// достигнуть барьера, чтобы после этого
// одновременно продолжить выполнение кода
CyclicBarrier(int count);
// дополнительно задается реализующий
// интерфейс Runnable класс, который должен быть
// запущен после прихода к барьеру всех потоков.
// Поток запускать самостоятельно НЕ НУЖНО.
// CyclicBarrier это делает автоматически.
CyclicBarrier(int count, Runnable class);
```



java-online.ru

CyclicBarrier (counter=3)

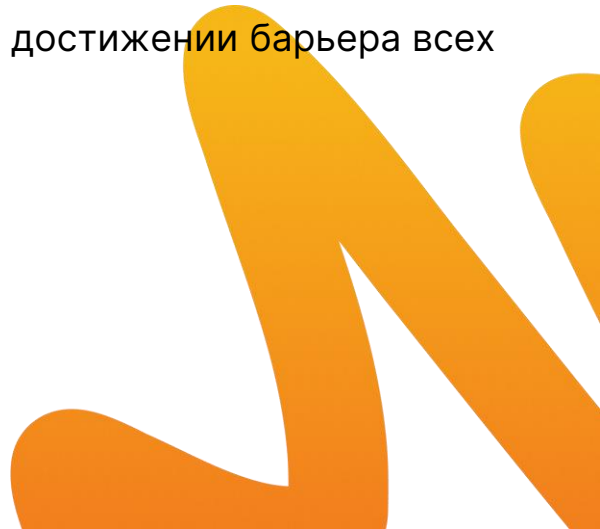
# Только время принадлежит нам

## Методы **CyclicBarrier**

```
void await() throws InterruptedException  
boolean await(long wait, TimeUnit unit) throws InterruptedException;
```

Назначение параметров *wait* и *unit* аналогично *CountDownLatch*.

С точки зрения API циклический барьер *CyclicBarrier* имеет только метод самоблокировки *await* и не имеет метода декрементации счетчика, а также позволяет подключить и автоматически запускать дополнительный потоковый класс при достижении барьера всех исполняемых потоков.



# Только время принадлежит нам

## Пример CyclicBarrier



ThreadSynchronizersExample.zip





# Задание

В многопользовательской игре у каждого игрового персонажа есть различные этапы подготовки (например, загрузка ресурсов, инициализация). Используйте CyclicBarrier, чтобы синхронизировать старт игровых персонажей после завершения всех этапов подготовки.



ThreadCyclicBarrierTask.zip



# Только время принадлежит нам

## Phaser

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



# Только время принадлежит нам

## Примеры задач для Phaser



- 1. Многопоточный процесс обработки заказов, состоящий из трех стадий. На первой стадии отдельные потоки исполнения проверяют сведения о клиенте, наличие товара на складе и их стоимость. На второй стадии вычисляется стоимость заказа и стоимость доставки. На заключительной стадии подтверждается оплата и определяется ориентировочное время доставки.*
- 2. Несколько потоков реализуют перевозку пассажиров городским транспортом. Пассажиры ожидают транспорт на разных остановках. Транспорт, останавливаясь на остановках, одних пассажиров «сажает», других «высаживает».*

Общим является то, что один объект синхронизации *Phaser*, исполняющий роль заказа/транспорта, играет главную роль, а другие потоки вступают в работу при определенном состоянии *Phaser*.

# Только время принадлежит нам

## Особенности Phaser

- *Phaser* может иметь несколько фаз (барьеров). Если количество фаз равно 1, то он вырождается в *CyclicBarrier*.
- Каждая фаза (цикл синхронизации) имеет свой номер.
- Количество участников-потоков для каждой фазы жестко не задано и может меняться. Исполнительный поток может регистрироваться в качестве участника и отменять свое участие;
- Исполнительный поток не обязан ожидать, пока все остальные участники соберутся у барьера. Достаточно только сообщить о своем прибытии.

java-online.ru



Phaser

# Только время принадлежит нам

## Конструкторы Phaser

*parties* определяет количество участников, которые должны пройти все фазы. Первый конструктор создает объект *Phaser* без каких-либо участников.

```
Phaser();  
Phaser(int parties);  
Phaser(Phaser parent);  
Phaser(Phaser parent, int parties);
```

Второй – регистрирует передаваемое в конструктор количество участников.

Третий и четвертый конструкторы дополнительно устанавливают родительский объект *Phaser*.

При создании экземпляр класса *Phaser* находится в нулевой фазе. В очередном состоянии (фазе) синхронизатор находится в ожидании до тех пор, пока все зарегистрированные потоки не завершат данную фазу. Потоки извещают об этом, вызывая один из методов *arrive()* или *arriveAndAwaitAdvance()*.

# Только время принадлежит нам

## Методы Phaser

Метод	Описание
<code>int register()</code>	Метод регистрирует участника и возвращает номер текущей фазы.
<code>int arrive()</code>	Метод указывает на завершение выполнения текущей фазы и возвращает номер фазы. Если же работа <i>Phaser</i> закончена, то метод вернет отрицательное число. При вызове метода <i>arrive</i> поток не приостанавливается, а продолжает выполняться.
<code>int arriveAndAwaitAdvance()</code>	Метод вызывается потоком-участником, чтобы указать, что он завершил текущую фазу. Это аналог метода <i>CyclicBarrier.await()</i> , сообщающего о прибытии к барьеру.
<code>int arriveAndDeregister()</code>	Метод сообщает о завершении всех фаз участником и снимается с регистрации. Данный метод возвращает номер текущей фазы или отрицательное число, если <i>Phaser</i> завершил свою работу
<code>int getPhase()</code>	Получение номера текущей фазы.

# Только время принадлежит нам

## Пример Phaser



ThreadSynchronizersExample.zip



# Задание

В стартапе есть несколько этапов разработки, и команды работают параллельно над различными аспектами проекта. Используйте Phaser, чтобы синхронизировать выполнение команд на различных этапах.





# Только время принадлежит нам

## Exchanger

**Exchanger<V>** – параметризированный 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.

```
V exchange(V buffer) throws InterruptedException;  
V exchange(V buffer, long wait, TimeUnit unit)  
throws InterruptedException;
```

*buffer* является ссылкой на обмениваемые данные.

*wait* и *unit* определяют время ожидания.

java-online.ru



Exchanger <V>

# Только время принадлежит нам

## Пример Exchanger



ThreadSynchronizersExample.zip



# Задание

Два почтальона из пунктов А и Б отправляются в соседние поселки В и Г доставить письма. Каждый из почтальонов должен доставить по письму в каждый из поселков. Чтобы не делать лишний круг, они встречаются в промежуточном поселке Д и обмениваются одним письмом. В результате этого каждому из почтальонов придется доставить письма только в один поселок. Все «шаги» почтальонов укажите выводом соответствующих сообщений в консоль.



3

# Домашнее задание

# Домашнее задание

1 Дан массив строк на 10\_000 элементов. Отсортируйте строки по длине, используя сортировку слиянием.

Сортировка должна быть выполнена в многопоточном режиме с применением ForkJoinPool.

2 Решите минимум две любые задачи:

2.1 В компании есть база данных, и несколько потоков пытаются одновременно записать данные в эту базу. Чтобы избежать коллизий при одновременной записи, используйте Semaphore, чтобы ограничить доступ к базе одновременно только нескольким потокам.

2.2 В гонке участвуют несколько участников. Все они должны быть подготовлены к гонке, и только после этого стартовать. Используйте CountdownLatch для ожидания подготовки всех участников.

2.3 На производственной линии роботы выполняют несколько этапов обработки продукции. Используйте CyclicBarrier, чтобы синхронизировать работу роботов и переходить к следующему этапу после завершения предыдущего.

2.4 Есть два потока, один из которых генерирует данные, а другой их обрабатывает. Используйте Exchanger, чтобы потоки могли обмениваться данными.

2.5 Музыкальный оркестр состоит из различных инструментальных секций, и каждая секция должна быть готова к выступлению. Используйте Phaser, чтобы координировать работу участников оркестра на разных этапах подготовки.

# Полезные ссылки

- [CountedCompleter](#) – реализация ForkJoinTask, появившаяся в Java 8.
- [ManagedBlocker](#) – интерфейс, позволяющий управлять заблокированными потоками в ForkJoinPool.

# ЗАКЛЮЧЕНИЕ

