В пакете **java.util.concurrent** определяются следующие средства параллелизма:

* Синхронизаторы
* Исполнители (Executors)
* Параллельные коллекции
* Каркас Fork/Join Framework

**СИНХРОНИЗАТОРЫ**

**Синхронизаторы** предоставляют высокоуровневые способы синхронизации взаимодействия нескольких потоков. Основные из них:

**Semaphore –** реализует классический семафор. Управляет доступом к общему ресурсу с помощью счетчика.

Когда потоку нужен доступ к ресурсу, он проверяет счетчик семафора. Если счетчик больше 0, то поток получит разрешение, а значение счетчика уменьшиться на 1. В противном случае поток будет заблокирован, пока счетчик будет равен 0. Если потоку больше не нужен доступ к ресурсу, он освобождает его, в результате чего счетчик семафора увеличивается на 1.

**Конструторы:**



Первый параметр **значение счетчика**. Второй определяет, **в каком порядке будет предоставляться доступ**. По умолчанию false – доступ предоставляется в неопределенном порядке. Если true – потокам предоставляется доступ в том порядке, в котором они его запрашивали.

**Методы:**

* **sem.acquire() –** запросить одно разрешение (уменьшить счетчик на 1)
* **sem.acquire(int число) –** запросить несколько разрешений
* **sem.release() –** освободить одно разрешение
* **sem.release(int число) –** освободить несколько разрешений

**CountDownLatch –** ожидает до тех пор, пока не произойдет определенное количество событий. Объект этого класса создается с количеством событий, которые должны произойти до того момента, как будет снята самоблокировка.

**Конструктор:**



**Методы:**

* **cdl.await() –** вызывает блокировку потока, пока счетчик CountDownLatch не достигнет нуля.
* **cdl.await(long ожидание, TimeUnit единица\_времени) –** блокировка пока счетчик не достигнет 0, либо пока не истечет таймаут. Возвращает false если достигнут таймаут, либо true если счетчик достиг 0.
* **cdl.countDown() –** метод оповещения о событии. Уменьшает счетчик на 1.

**CyclicBarrier –** позволяет группе потоков исполнения войти в режим ожидания в предварительно заданной точке выполнения. То есть все потоки останавливаются в определенном месте до тех пор, пока все потоки не достигнут этой точки (барьера).

**Конструторы:**



Первый параметр – количество потоков, которые должны достичь барьера. Второй параметр – поток, который будет исполнен по достижении барьера.

Барьер в каждом потоке определяется вызовом метода **await()** объекта CyclicBarrier**.** В итоге исполнение потока будет приостановлено до тех пор , пока **await()** не будет вызван во всех потоках. Если указать второй параметр конструктора, то будет также запущен этот поток.

Этот класс можно использовать повторно. То есть после того как барьер будет достигнут, выполнение продолжиться, а счетчик потоков вернется в начальное значение.

**Методы:**

* **await() –** ожидать, пока указанное число потоков не достигнет барьерной точки
* **await(long ожидание, TimeUnit единица\_времени) –** ожидание будет происходить пока не истечет таймаут. В обеих формах возвращается порядок, в котором потоки будут достигать барьерной точки. Первый поток вернет значение, равное количеству ожидаемых потоков минус 1, а последний поток возвращает нулевое значение.

**Exchanger –** упрощает обмен данными между двумя потоками исполнения.

Он ожидает до тех пор, пока два отдельных потока вызовут его метод **exchange()**. После этого, Exchanger произведет обмен данными, предоставляемыми обоими потоками.

Exchanger является обобщенным, где обобщение указывает тип обмениваемых данных.

**Методы:**

* **exchange(V буфер) –** буфер это ссылка на обмениваемые данные. Возвращаются данные, полученные из другого потока исполнения.
* **exchange(V буфер, long ожидание, TimeUnit единица времени) –** как и первая только с таймаутом.

Главная особенность метода exchange состоит в том, что он не завершается успешно до тех пор, пока не будет вызван для одного и того же объекта Exchanger из двух отдельных потоков.

**Phaser –** синхронизирует потоки исполнения, которые представляют одну или несколько стадий (фаз) выполнения действия.

Обычно сначала создается новый экземпляр класса Phaser. Затем синхронизатор фаз регистрирует одну или несколько сторон, вызывая метод **register()** или указывая нужное количество сторон в конструкторе. Синхронизатор фаз ожидает до тех пор, пока все зарегистрированные стороны не завершат фазу.

Стороны извещают об этом, вызывая например методы **arrive()** или **arriveAndAwaitAdvance().** Как только все стороны достигнут данной фазы, синхронизатор фаз может перейти к следующей фазе (если она имеется) или завершить свою работу.

**Методы:**

* **register() –** зарегистрировать сторону в фазе. Возвращается номер регистрируемой фазы.
* **arrive() –** сторона вызывает его, чтобы сообщить о завершении фазы. Возвращает текущий номер фазы. Если работа синхронизатора фаз завершена, возвращает отрицательное значение. Этот метод не приостанавливает выполнения потока (то есть он не ждет пока другие потоки завершат фазу)
* **arriveAndAwaitAdvance()** – как и arrive(), но поток будет ожидать пока другие потоки завершат фазу.
* **arriveAndDeregister()** – сообщить о завершении фазы и отменить регистрацию в синхронизаторе.

**EXECUTERS**

**Executors (исполнители)** управляют исполнением потоков. На вершине иерархии находится интерфейс **Executor**, предназначенный для запуска потока исполнения.

В Executor определяется метод **execute(Runnable поток).** В результате вызова этого метода запускается указанный поток.

Интерфейс **ExecutorService** расширяет интерфейс Executor и предоставляет методы, управляющие исполнением.

Метод **shutdown()** останавливает останавливает все потоки исполнения, находящиеся в данный момент под управлением данного ExecutorService.

Имеет три реализации в к классах **ThreadPoolExecutor, ScheduledThreadPoolExecutor** и **ForkJoinPool.**

**EXECUTORS**

**Executors –** служебный класс, содержащий несколько статических методов, упрощающих создание различных исполнителей.

**Методы:**

* **newCachedThreadPool() –** создает пул потоков, который не только вводит потоки в исполнение по мере необходимости, но и по возможности повторно использует их.
* **newFixedThreadPool(int количесвто\_потоков) –** пул потоков фиксированной длины
* **newScheduleThreadPool(int количесвто\_потоков) –** пул в котором можно осуществлять планирование потоков исполнения.

**FUTURE**

Интерфейс **Future<V>** содержит значение, возвращаемое потоком после исполнения. V определяет тип возвращаемого результата. Таким образом это значение определяется “на будущее”, когда поток завершит свое исполнение.

**Методы:**

* **get() –** получить значение из Future. Поток блокируется, пока не завершится соответствующий Callable.
* **get(long ожидание, TimeUnit единица времени) –** аналогичен предыдущему, но ждет только в течении указанного времени

**CALLABLE**

Интерфейс **Callable<V>** определяет поток исполнения, возвращающий значение. **V –** обозначает тип данных, возвращаемых потоком исполнения.

В нем определяется единственный метод **call().** Внутри него определяется код, который нужно выполнить. Если результат нельзя вычислить, метод call() генерирует исключение.

Запускается Callable вызовом метода **submit(Callable<T> callable)** определенного в интерфейсе **ExecutorService.** Результат возвращается через объект типа Future.

**ATOMICS**

Средства пакета java.util.concurrent.atomic упрощают присваивание переменным без применения блокировок.