Многопоточность

**Многопоточности** – это программирование интерфейсов. Многопоточность незаменима тогда, когда необходимо, чтобы графический интерфейс продолжал отзываться на действия пользователя во время выполнения некоторой обработки информации.

**Процесс** — это совокупность кода и данных, разделяющих общее виртуальное адресное пространство.

Для каждого процесса ОС создает так называемое «виртуальное адресное пространство», к которому процесс имеет прямой доступ. Это пространство принадлежит процессу, содержит только его данные и находится в полном его распоряжении. Операционная система же отвечает за то, как виртуальное пространство процесса проецируется на физическую память.

**Поток** – это одна единица исполнения кода. Каждый поток последовательно выполняет инструкции процесса, которому он принадлежит, параллельно с другими потоками этого процесса.

**Псевдо-параллелизмом** – это процедура при которой ядро процессора переключается с одного потока на другой, чем имитирует их параллельное выполнение.

**Главный поток** – это поток с которого начинается выполнения всего процесса.

Thread

**Thread** – это класс, потомки которого представляют потоки.

Все реализованные интерфейсы: Runnable

Прямые известные подклассы: ForkJoinWorkerThread

extends Object implements Runnable

**Поля:**

**MAX\_PRIORITY -** максимальный приоритет, который может иметь поток.

**MIN\_PRIORITY -** минимальный приоритет, который может иметь поток.

**NORM\_PRIORITY -** приоритет по умолчанию, который присваивается потоку.

**Методы:**

**activeCount()** - возвращает оценку количества потоков платформы live в группе потоков текущего потока и ее подгруппах.

**checkAccess()** - устаревший, для удаления: Этот элемент API подлежит удалению в будущей версии. Этот метод полезен только в сочетании с менеджером безопасности, который устарел и подлежит удалению в будущей версии.

**clone()** - выдает исключение CloneNotSupportedException, поскольку поток не может быть клонирован осмысленно.

**countStackFrames() -** устаревший, для удаления: Этот элемент API подлежит удалению в будущей версии.Этот метод изначально был разработан для подсчета количества фреймов стека, но результаты никогда не были четко определены и зависели от приостановки потока.

**currentThread()** - возвращает объект Thread для текущего потока.

**dumpStack()** - выводит трассировку стека текущего потока в стандартный поток ошибок.

**enumerate(Thread[] tarray) -** копирует в указанный массив каждый поток текущей платформы в группе потоков текущего потока и ее подгруппах.

**getAllStackTraces()** - возвращает карту трассировок стека для всех потоков текущей платформы.

**getContextClassLoader()** - возвращает контекст ClassLoader для этого потока.

**getDefaultUncaughtExceptionHandler()** - возвращает обработчик по умолчанию, вызываемый, когда поток внезапно завершается из-за не перехваченного исключения.

**getId() -** устарел.Этот метод не является окончательным и может быть переопределен для возврата значения, которое не является идентификатором потока.

**getName() -** возвращает имя этого потока.

**getPriority() -** возвращает приоритет этого потока.

**getStackTrace() -** возвращает массив элементов трассировки стека, представляющих дамп стека этого потока.

**getState()** - возвращает состояние этого потока.

**getThreadGroup()** - возвращает группу потоков потока или null если поток завершился.

**getUncaughtExceptionHandler() -** возвращает обработчик, вызываемый, когда этот поток внезапно завершается из-за не перехваченного исключения.

**holdsLock(Object obj) -** возвращает true тогда и только тогда, когда текущий поток удерживает блокировку монитора для указанного объекта.

**interrupt() - прерывает этот поток.**

**interrupted()** - проверяет, был ли прерван текущий поток.

**isAlive() -** проверяет, является ли этот поток активным.

**isDaemon()** - проверяет, является ли этот поток потоком демона.

**isInterrupted()** - проверяет, был ли прерван этот поток.

**isVirtual() -** предварительный просмотр. Возвращает, true если этот поток является виртуальным потоком.

**join()** - ожидает завершения этого потока.

**join(long millis) -** ожидает завершения этого потока не более millis миллисекунд.

**join(long millis, int nanos)** - ожидает завершения этого потока не более millis миллисекунд плюс nanos наносекунды.

**join(Duration duration)** - ожидает завершения этого потока в течение заданной продолжительности ожидания.

**ofPlatform() -** предварительный просмотр. Возвращает конструктор для создания платформы Thread или ThreadFactory который создает потоки платформы.

**ofVirtual()** - предварительный просмотр. Возвращает конструктор для создания виртуального Thread или ThreadFactory, который создает виртуальные потоки.

**onSpinWait()** - указывает, что вызывающий на мгновение не может выполнить выполнение, пока не произойдет одно или более действий со стороны других действий.

**resume() -** устаревший, для удаления: Этот элемент API подлежит удалению в будущей версии. Этот метод существует исключительно для использования с suspend(), который устарел, поскольку подвержен взаимоблокировке.

**run()** - этот метод запускается потоком при его выполнении.

**setContextClassLoader(ClassLoader cl)** - задает контекст ClassLoader для этого потока.

**setDaemon(boolean on)** - помечает этот поток либо как поток демона, либо как поток не-демона.

**setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler ueh)** -установите обработчик по умолчанию, вызываемый, когда поток внезапно завершается из-за не перехваченного исключения, и для этого потока не был определен никакой другой обработчик.

**setName(String name)** - изменяет имя этого потока, чтобы оно было равно аргументу name.

**setPriority(int newPriority) -** изменяет приоритет этого потока.

**setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler ueh) -** установите обработчик, вызываемый, когда этот поток внезапно завершается из-за не перехваченного исключения.

**sleep(long millis) -** переводит текущий выполняющийся поток в спящий режим (временно прекращает выполнение) на указанное количество миллисекунд, в зависимости от точности системных таймеров и планировщиков.

**sleep(long millis, int nanos) -** переводит текущий выполняющийся поток в спящий режим (временно прекращает выполнение) на указанное количество миллисекунд плюс указанное количество наносекунд, в зависимости от точности системных таймеров и планировщиков.

**sleep(Duration duration) -** переводит текущий выполняющийся поток в спящий режим (временно прекращает выполнение) на указанную продолжительность в зависимости от точности системных таймеров и планировщиков.

**start()** - планирует начало выполнения этого потока.

**startVirtualThread(Runnable task)** - предварительный просмотр. Создает виртуальный поток для выполнения задачи и планирует его выполнение.

**stop()** - устаревший, для удаления: Этот элемент API подлежит удалению в будущей версии. Этот метод по своей сути небезопасен.

**suspend()** - устаревший, для удаления: Этот элемент API подлежит удалению в будущей версии. Этот метод устарел, поскольку он по своей сути подвержен взаимоблокировке.

**threadId()** - возвращает идентификатор этого потока.

**toString()** - возвращает строковое представление этого потока.

**yield()** - подсказка планировщику о том, что текущий поток готов предоставить свое текущее использование процессора.

Runnable

**Runnable –** это интерфейс при реализации которого объект можно перевести в отдельный поток.

**Методы:**

**run()** - когда для создания потока используется интерфейс реализации объекта Runnable, запуск потока приводит к вызову run метода объекта в этом отдельно выполняющемся потоке.

Как создать свой поток:

1. **Первый способ** – создать объект типа Thread и передать ему объект, который реализует интерфейс Runnable, он содержит метод run который будет выполнятся в новом потоке.
2. **Второй способ** – Создать класс наследник класса Thread который переопределит метод run. Создать экземпляр этого класса и вызвать наследуемый метод start.

Завершение процесса и демоны

**В Java** процесс завершается тогда, когда завершается последний его поток. Даже если метод main() уже завершился, но еще выполняются порожденные им потоки, система будет ждать их завершения.

**Потоки – демоны** – это потоки к которым не относится вышеописанное правило. Если обычные потоки будут завершены, то все потоки – демоны будут завершены принудительно.

**Методы:**

**setDaemon(true)** – объявляет поток демоном.

**isDaemon()** – проверяет является ли поток демоном.

**Планировщик потоков** – это часть jvm которая решает какой поток должен запустится в какой момент, а какой приостановится.

**Остановить поток** – нельзя, он может остановится только сам. Но можно явно указать чтоб поток остановился.

Приоритет потоков

**Приоритет** - существует для контроля важности выполнения потока. По умолчанию поток имеет приоритет 5. Его можно задать от 1 до 10. Задать его можно с помощью метода **setPriority(int).**

Использование памяти

Чем больше потоков создается – тем больше памяти используется. Во многих системах может быть ограничение на количество потоков. Даже если такого ограничения нет, в любом случае имеется естественное ограничение в виде максимальной скорости процессора.

Для каждого потока создается свой собственный стек в памяти. Туда помещаются все локальные переменные и ряд других данных, связанных с выполнением потока.

Возможные ошибки. Deadlock и Race condition

**Deadlock (взаимная блокировка)** – несколько потоков находятся в состоянии ожидания ресурсов, занятых друг другом, и ни один из них не может продолжать выполнение.

**Race Condition (состояние гонки)** – ошибка проектирования многопоточной системы или приложения, при которой работа системы или приложения зависит от того, в каком порядке выполняются части кода.

Синхронизация потоков. Блокировка ресурсов

**Синхронизацией потоков** – это способ при котором доступ к ресурсу ограничивается и выдается последовательно каждому потоку. Для этого используется ключевое слово **synchronized.**

Semaphore

**Semaphore** — это класс который занимается синхронизации доступа к ресурсу. Ограничивает количество потоков, которые могут войти в заданный участок кода использует счетчик потоков, который указывает, сколько потоков одновременно могут получать доступ к общему ресурсу.

**Методы:**

**acquire()** - получает разрешение от этого семафора, блокируя до тех пор, пока один из них не станет доступным или поток не будет прерван.

**acquire(int permits)** - получает заданное количество разрешений из этого семафора, блокируя до тех пор, пока все не будут доступны, или поток не будет прерван.

**acquireUninterruptibly()** - получает разрешение от этого семафора, блокируя до тех пор, пока оно не станет доступным.

**acquireUninterruptibly(int permits)** - получает заданное количество разрешений из этого семафора, блокируя до тех пор, пока не будут доступны все.

**availablePermits()** - возвращает текущее количество разрешений, доступных в этом семафоре.

**drainPermits()** - получает и возвращает все разрешения, которые доступны немедленно, или, если доступны отрицательные разрешения, освобождает их.

**getQueuedThreads()** - возвращает коллекцию, содержащую потоки, которые могут ожидать получения.

**getQueueLength()** - возвращает оценку количества потоков, ожидающих получения.

**hasQueuedThreads()** - запрашивает, ожидают ли какие-либо потоки получения.

**isFair()** - возвращает true, если для этого семафора установлено значение справедливости true.

**reducePermits(int reduction)** - уменьшает количество доступных разрешений на указанное уменьшение.

**release()** - освобождает разрешение, возвращая его семафору.

**release(int permits)** - освобождает заданное количество разрешений, возвращая их в семафор.

**toString()** - возвращает строку, идентифицирующую этот семафор, а также его состояние.

**tryAcquire()** - получает разрешение от этого семафора, только если он доступен во время вызова.

**tryAcquire(int permits)** - получает заданное количество разрешений от этого семафора, только если все они доступны во время вызова.

**tryAcquire(int permits, long timeout, TimeUnit unit) -** получает заданное количество разрешений из этого семафора, если все они становятся доступными в течение заданного времени ожидания и текущий поток не был прерван.

**tryAcquire(long timeout, TimeUnit unit)** - получает разрешение от этого семафора, если он становится доступным в течение заданного времени ожидания и текущий поток не был прерван.

Mutex

**Mutex** — поле для синхронизации потоков. Есть у каждого объекта в Java. Это простейший **Semaphore**, который может находиться в одном из двух состояний: true или false.

Monitor

**Monitor** — это дополнительная надстройка над **Mutex**. Фактически монитор — это «невидимый» для программиста кусок код.

synchronized (obj) {

//логика, которая одновременно доступна только для одного потока

}

В блоке кода, который помечен, словом, **synchronized**, происходит захват мьютекса нашего объекта obj. Защитный механизм создает именно монитор! Компилятор преобразует слово synchronized в несколько специальных кусков кода.

Модификатор volatile

При взаимодействии с переменной каждый поток хранит ее значение в своем стеке. Может возникнуть ситуация, что один поток изменит значение общей переменной, а второй поток будет продолжать работать с ее старым значением из своего кэша.

1. Операции чтения и записи volatile переменной являются атомарными.
2. Переменная не будет помещаться в кэш: результат записи значения в volatile переменную одним потоком будет виден всем другим потокам, которые используют эту переменную для чтения.

Callable

**Callable** – это интерфейс схожий по работе с Runnable. Но в отличие от него он объявляется метод call() который возвращает результат работы потока. Использует дженерик для указания типа возвращаемого объекта.

**Методы:**

**call()** - возвращает результат работы потока.

Future

**Future** – это интерфейс который может хранить результаты асинхронного вычисления.

**Методы:**

**cancel(boolean mayInterruptIfRunning)** - пытается отменить выполнение этой задачи.

**get()** - ожидает, если необходимо, завершения вычисления, а затем извлекает его результат.

**get(long timeout, TimeUnit unit)** - ожидает, если необходимо, не более заданного времени завершения вычисления, а затем извлекает его результат, если таковой имеется.

**isCancelled()** - возвращает, true если эта задача была отменена до ее нормального завершения.

**isDone()** - возвращает true, если эта задача выполнена.

FutureTask

**FutureTask** – это класс который предоставляет базовую реализацию Future с методами для запуска и отмены вычисления, запроса, чтобы увидеть, завершено ли вычисление, и получения результата вычисления. Результат может быть получен только по завершении вычисления; get методы будут заблокированы, если вычисление еще не завершено.

Введите параметры:

V - Тип результата, возвращаемый этой задачей FutureTask get методы All Implemented Все реализованные интерфейсы:

Runnable, Future<V>, RunnableFuture<V>

extends Object implements RunnableFuture<V>

**Методы:**

**cancel(boolean mayInterruptIfRunning)** - пытается отменить выполнение этой задачи.

**done()** - защищенный метод, вызываемый при переходе этой задачи в состояние isDone (как обычно, так и путем отмены).

**get()** - ожидает, если необходимо, завершения вычисления, а затем извлекает его результат.

**get(long timeout, TimeUnit unit)** - ожидает, если необходимо, не более заданного времени завершения вычисления, а затем извлекает его результат, если таковой имеется.

**isCancelled()** - возвращает true если эта задача была отменена до ее нормального завершения.

**isDone()** - возвращает true если эта задача выполнена.

**run()** - устанавливает это будущее в результат своего вычисления, если оно не было отменено.

**runAndReset()** - выполняет вычисление без установки его результата, а затем сбрасывает это будущее в исходное состояние, не в состоянии этого сделать, если вычисление сталкивается с исключением или отменяется.

**set(V v)** - устанавливает результат этого будущего в заданное значение, если только это будущее уже не было установлено или не было отменено.

**setException(Throwable t)** - заставляет это будущее сообщать о ExecutionException с заданным значением throwable в качестве его причины, если только это будущее уже не было установлено или не было отменено.

**toString()** - возвращает строковое представление этой FutureTask.

 Атомарные переменные

**Атомарность** – это операции, которые могут быть выполнены только полностью. Если они не выполняются полностью, они не выполняются вообще, но прервать их невозможно.

Атомарные переменные – это набор классов который оборачивают примитивные типы для поток безопасной работы.

**К ним относятся:**

**AtomicBoolean** – boolean значение, которое может обновляться атомарно.

**AtomicInteger** – int значение, которое может быть обновлено атомарно.

**AtomicIntegerArray** – int массив, элементы в котором могут обновляться атомарно.

**AtomicIntegerFieldUpdater<T>** - Утилита на основе рефлекции, которая позволяет выполнять атомарные обновления для назначенных volatile int поля назначенных классов.

**AtomicLong** – long значение, которое может обновляться атомарно.

**AtomicLongArray** - long массив, элементы в котором могут обновляться атомарно.

**AtomicLongFieldUpdater<T>** - утилита на основе рефлекции, которая позволяет выполнять атомарные обновления для назначенных volatile long поля назначенных классов.

**AtomicMarkableReference<V>** - AtomicMarkableReferenceAn поддерживает ссылку на объект вместе с битом метки, который может быть обновлен атомарно.

**AtomicReference<V>** - ссылка на объект, которая может обновляться атомарно.

**AtomicReferenceArray<E>** - массив ссылок на объекты, в которых элементы могут обновляться атомарно.

**AtomicReferenceFieldUpdater<T,V>** - утилита на основе отражения, которая позволяет выполнять атомарные обновления для назначенных volatile ссылочные поля назначенных классов.

**AtomicStampedReference<V>** - AtomicStampedReferenceAn поддерживает ссылку на объект вместе с целочисленным "штампом", который может быть обновлен атомарно.

**DoubleAccumulator** - одна или несколько переменных, которые вместе поддерживают выполнение double значение обновлено с помощью предоставленной функции.

**DoubleAdder** - одна или несколько переменных, которые вместе поддерживают изначально нулевое значение double сумма.

**LongAccumulator** - одна или несколько переменных, которые вместе поддерживают выполнение long значение обновлено с помощью предоставленной функции.

**LongAdder** - одна или несколько переменных, которые вместе поддерживают изначально нулевое значение long сумма.

**Их методы можно посмотреть на:** <https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/concurrent/atomic/package-summary.html>

Lock

**Lock** – это интерфейс который обеспечивают более обширные операции блокировки, чем могут быть получены с помощью synchronized методов и инструкций.

**Методы:**

**lock()** - получает блокировку.

**lockInterruptibly()** - получает блокировку до тех пор, пока текущий поток не будет прерван.

**newCondition()** - возвращает новый Condition экземпляр, который привязан к этому Lock экземпляру.

**tryLock()** - получает блокировку, только если она свободна во время вызова.

**tryLock(long time, TimeUnit unit)** - получает блокировку, если она свободна в течение заданного времени ожидания и текущий поток не был прерван.

**unlock()** - снимает блокировку.

ReentrantLock

**ReentrantLock** – это класс который реализует интерфейс Lock и обеспечивает блокировку потоков.

**Методы:**

**getHoldCount()** - запрашивает количество удержаний для этой блокировки текущим потоком.

**getOwner()** - возвращает поток, которому в данный момент принадлежит эта блокировка, или null если он не принадлежит.

**getQueuedThreads()** - возвращает коллекцию, содержащую потоки, которые могут ожидать получения этой блокировки.

**getQueueLength()** - возвращает оценку количества потоков, ожидающих получения этой блокировки.

**getWaitingThreads(Condition condition)** - возвращает коллекцию, содержащую те потоки, которые могут ожидать выполнения данного условия, связанного с этой блокировкой.

**getWaitQueueLength(Condition condition)** - возвращает оценку количества потоков, ожидающих выполнения данного условия, связанного с этой блокировкой.

**hasQueuedThread(Thread thread)** - запрашивает, ожидает ли данный поток получения этой блокировки.

**hasQueuedThreads()** - запрашивает, ожидают ли какие-либо потоки получения этой блокировки.

**hasWaiters(Condition condition)** - запрашивает, ожидают ли какие-либо потоки выполнения данного условия, связанного с этой блокировкой.

**isFair()** - возвращает true если для этой блокировки установлена справедливость true.

**isHeldByCurrentThread()** - запрашивает, удерживается ли эта блокировка текущим потоком.

**isLocked()** - запрашивает, удерживается ли эта блокировка каким-либо потоком.

**lock()** - получает блокировку.

**lockInterruptibly()** - получает блокировку, если текущий поток не прерванный.

**newCondition()** - возвращает Condition экземпляр для использования с этим Lock экземпляр.

**toString()** - возвращает строку, идентифицирующую эту блокировку, а также ее состояние блокировки.

**tryLock()** - получает блокировку, только если она не удерживается другим потоком во время вызова.

**tryLock(long timeout, TimeUnit unit)** - получает блокировку, если она не удерживается другим потоком в течение заданного времени ожидания и текущий поток не был прерван.

**unlock()** - пытается снять эту блокировку.

Condition

**Condition** – это интерфейс который позволяет выставить ожидания в нескольких потоках до момента пока другой поток его не запустит.

**Методы:**

**await()** - заставляет текущий поток ждать, пока не будет подан сигнал или прерванный.

**await(long time, TimeUnit unit)** - заставляет текущий поток ждать, пока не будет подан сигнал или прерван, или не истечет указанное время ожидания.

**awaitNanos(long nanosTimeout)** - заставляет текущий поток ждать, пока не будет подан сигнал или прерван, или не истечет указанное время ожидания.

**awaitUninterruptibly()** - заставляет текущий поток ждать, пока не будет подан сигнал.

**awaitUntil(Date deadline)** - заставляет текущий поток ждать, пока не будет подан сигнал или прерван, или не истечет указанный крайний срок.

**signal()** -пробуждает один ожидающий поток.

**signalAll()** - пробуждает все ожидающие потоки.

 Executors

**Executors** – это интерфейс для работы с пулом потоков. И для переиспользования потоков.

**Методы:**

**callable(Runnable task)** - возвращает Callable объект, который при вызове выполняет заданную задачу и возвращает null.

**callable(Runnable task, T result)** - возвращает Callable объект, который при вызове выполняет заданную задачу и возвращает заданный результат.

**callable(PrivilegedAction<?> action)** - возвращает Callable объект, который при вызове выполняет данное привилегированное действие и возвращает его результат.

**callable(PrivilegedExceptionAction<?> action**) - возвращает Callable объект, который при вызове выполняет данное привилегированное действие исключения и возвращает его результат.

**defaultThreadFactory()** - возвращает фабрику потоков по умолчанию, используемую для создания новых потоков.

**newCachedThreadPool()** - создает пул потоков, который создает новые потоки по мере необходимости, но будет повторно использовать ранее созданные потоки, когда они будут доступны.

**newCachedThreadPool(ThreadFactory threadFactory)** - создает пул потоков, который создает новые потоки по мере необходимости, но будет повторно использовать ранее созданные потоки, когда они будут доступны, и использует предоставленный ThreadFactory для создания новых потоков, когда это необходимо.

**newFixedThreadPool(int nThreads)** - создает пул потоков, который повторно использует фиксированное количество потоков, работающих из общей неограниченной очереди.

**newFixedThreadPool(int nThreads, ThreadFactory threadFactory**) - создает пул потоков, который повторно использует фиксированное количество потоков, работающих в общей неограниченной очереди, используя предоставленный ThreadFactory для создания новых потоков при необходимости.

**newScheduledThreadPool(int corePoolSize)** - создает пул потоков, который может планировать выполнение команд после заданной задержки или выполнять периодически.

**newScheduledThreadPool(int corePoolSize, ThreadFactory threadFactory)** - создает пул потоков, который может планировать выполнение команд после заданной задержки или выполнять периодически.

**newSingleThreadExecutor()** - создает исполнителя, который использует один рабочий поток, работающий с неограниченной очередью.

**newSingleThreadExecutor(ThreadFactory threadFactory)** - создает исполнителя, который использует один рабочий поток, работающий с неограниченной очередью, и использует предоставленный ThreadFactory для создания нового потока, когда это необходимо.

**newSingleThreadScheduledExecutor()** - создает однопоточный исполнитель, который может планировать выполнение команд после заданной задержки или выполнять периодически.

**newSingleThreadScheduledExecutor(ThreadFactory threadFactory)** - создает однопоточный исполнитель, который может планировать выполнение команд после заданной задержки или выполнять периодически.

**newWorkStealingPool()** - создает пул потоков, забирающий работу, используя количество доступных процессоров в качестве целевого уровня параллелизма.

**newWorkStealingPool(int parallelism)** - создает пул потоков, который поддерживает достаточное количество потоков для поддержки заданного уровня параллелизма и может использовать несколько очередей для уменьшения конкуренции.

**privilegedCallable(Callable<T> callable)** - устаревший, для удаления: Этот элемент API подлежит удалению в будущей версии. Этот метод полезен только в сочетании с Security Manager, который устарел и подлежит удалению в будущей версии.

**privilegedCallableUsingCurrentClassLoader(Callable<T> callable)** - устаревший, для удаления: Этот элемент API подлежит удалению в будущей версии. Этот метод полезен только в сочетании с Security Manager, который устарел и подлежит удалению в будущей версии.

**privilegedThreadFactory()** - устаревший, для удаления: Этот элемент API подлежит удалению в будущей версии. Этот метод полезен только в сочетании с Security Manager, который устарел и подлежит удалению в будущей версии.

**unconfigurableExecutorService(ExecutorService executor)** - возвращает объект, который делегирует все определенные ExecutorService методы данному исполнителю, но не любые другие методы, которые в противном случае могли бы быть доступны с помощью приведений.

**unconfigurableScheduledExecutorService(ScheduledExecutorService executor)** - возвращает объект, который делегирует все определенные ScheduledExecutorService методы данному исполнителю, но не любые другие методы, которые в противном случае могли бы быть доступны с помощью приведений.

ExecutorService

**ExecutorService** – это интерфейс который расширяет возможности Execute.

**Методы:**

**awaitTermination(long timeout, TimeUnit unit)** - блокирует до тех пор, пока все задачи не завершат выполнение после запроса на завершение работы, или не наступит тайм-аут, или текущий поток не будет прерван, в зависимости от того, что произойдет раньше.

**invokeAll(Collection<? extends Callable<T>> tasks**) - выполняет заданные задачи, возвращая список фьючерсов с указанием их статуса и результатов, когда все они завершены.

**invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)** - выполняет заданные задачи, возвращая список фьючерсов с сохранением их статуса и результатов, когда все завершено или истекает время ожидания, в зависимости от того, что произойдет раньше.

**invokeAny(Collection<? extends Callable<T>> tasks)** - выполняет заданные задачи, возвращая результат той, которая успешно завершена (т. Е. без создания исключения), если таковые имеются.

**invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)** - выполняет заданные задачи, возвращая результат одной из них, которая успешно завершена (т. Е. Без создания исключения), если какие-либо из них выполняются до истечения заданного времени ожидания.

**isShutdown()** - возвращает, true если этот исполнитель был закрыт.

**isTerminated()** - возвращает, true если все задачи завершены после завершения работы.

**shutdown()** - инициирует упорядоченное завершение работы, при котором выполняются ранее отправленные задачи, но новые задачи не будут приняты.

**shutdownNow()** - пытается остановить все активно выполняемые задачи, останавливает обработку ожидающих задач и возвращает список задач, ожидавших выполнения.

**submit(Runnable task)** - отправляет выполняемую задачу на выполнение и возвращает будущее, представляющее эту задачу.

**submit(Runnable task, T result)** - отправляет выполняемую задачу на выполнение и возвращает будущее, представляющее эту задачу.

**submit(Callable<T> task)** - отправляет задачу, возвращающую значение, на выполнение и возвращает будущее, представляющее ожидающие результаты задачи.

ScheduledExecutorService

**ScheduledExecutorService** – это интерфейс который позволяет планировать выполнение команд после заданной задержки или выполнять периодически.

**Методы:**

**schedule(Runnable command, long delay, TimeUnit unit)** - отправляет одноразовую задачу, которая становится разрешенной после заданной задержки.

**schedule(Callable<V> callable, long delay, TimeUnit unit**) - отправляет одноразовую задачу, возвращающую значение, которая становится разрешенной после заданной задержки.

**scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)** - отправляет периодическое действие, которое становится разрешенным сначала после заданной начальной задержки, а затем с заданным периодом; то есть выполнение начнется после initialDelay, затем initialDelay + period, затем initialDelay + 2 \* period и так далее.

**scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)** - отправляет периодическое действие, которое становится разрешенным сначала после заданной начальной задержки, а затем с заданной задержкой между завершением одного выполнения и началом следующего.

CountDownLatch

**CountDownLatch** –это класс который позволяет блокировать методы о тех пор, пока текущее значение счетчика не достигнет нуля из-за вызовов countDown().

**Методы:**

**await()** - заставляет текущий поток ждать, пока время обратного отсчета не уменьшится до нуля, если только поток не будет прерван.

**await(long timeout, TimeUnit unit)** - заставляет текущий поток ожидать, пока время обратного отсчета не уменьшится до нуля, если только поток не будет прерван или не истечет указанное время ожидания.

**countDown()** - уменьшает количество защелок, освобождая все ожидающие потоки, если количество достигает нуля.

**getCount()** - возвращает текущее количество.

**toString()** - возвращает строку, идентифицирующую эту блокировку, а также ее состояние.

Exchanger

**Exchanger** – это класс который может передавать данные между потоками.

**Методы:**

**exchange​(V x)** - ожидает прибытия другого потока в эту точку обмена (если только текущий поток не прерван), а затем передает ему данный объект, получая взамен свой объект.

**exchange​(V x, long timeout, TimeUnit unit)** - ожидает прибытия другого потока в эту точку обмена (если только текущий поток не будет прерван или не истечет указанное время ожидания), а затем передает ему данный объект, получая взамен свой объект.

CyclicBarrier

**CyclicBarrier** – это класс который позволяет набору потоков ожидать друг друга, чтобы достичь общей точки барьера.

**Метод:**

**await()**  - ожидает, пока все стороны не вызовут await этот барьер.

**await​(long timeout, TimeUnit unit)** - ожидает, пока все стороны не вызовут await этот барьер или не истечет указанное время ожидания.

**getNumberWaiting()** - возвращает количество участников, ожидающих в данный момент у барьера.

**getParties()** - возвращает количество сторон, необходимых для отключения этого барьера.

**isBroken()** - запрашивает, находится ли этот барьер в нерабочем состоянии.

**reset()** - возвращает барьер в исходное состояние.

Phaser

**Phaser** – это класс который позволяет контролировать несколько потоков чтоб они выполняли действия синхронно.

**Методы:**

**arrive()** - оступает на этот фазер, не дожидаясь прибытия других.

**arriveAndAwaitAdvance()** - поступает на этот фазер и ожидает других.

**arriveAndDeregister()** - поступает на этот phaser и отменяет регистрацию в нем, не дожидаясь прибытия других.

**awaitAdvance(int phase)** - ожидает перехода фазы этого фазера от заданного значения фазы, немедленно возвращаясь, если текущая фаза не равна заданному значению фазы или этот фазер завершен.

**awaitAdvanceInterruptibly(int phase)** - ожидает, что фаза этого фазера перейдет от заданного значения фазы, отбрасывая, InterruptedException если прервано во время ожидания, или немедленно возвращается, если текущая фаза не равна заданному значению фазы или этот фазер завершен.

**awaitAdvanceInterruptibly(int phase, long timeout, TimeUnit unit)** - ожидает, что фаза этого фазера перейдет от заданного значения фазы или истечет заданный тайм-аут, выбрасывая, InterruptedException если прервано во время ожидания, или немедленно возвращается, если текущая фаза не равна заданному значению фазы или этот фазер завершен.

**bulkRegister(int parties)** - добавляет заданное количество новых неотправленных сторон к этому phaser.

**forceTermination()** - переводит этот фазер в состояние завершения.

**getArrivedParties()** - возвращает количество зарегистрированных сторон, которые достигли текущей фазы этого этапа.

**getParent()** - возвращает родительский элемент этого фазера, или null если его нет.

**getPhase()** - возвращает номер текущей фазы.

**getRegisteredParties()** - возвращает количество сторон, зарегистрированных на этом фазере.

**getRoot()** - возвращает корневого предка этого фазера, который совпадает с этим фазером, если у него нет родительского элемента.

**getUnarrivedParties()** - возвращает количество зарегистрированных сторон, которые еще не достигли текущей фазы этого этапа.

**isTerminated()** - возвращает, true если этот фазер был завершен.

**onAdvance(int phase, int registeredParties)** - переопределяемый метод для выполнения действия при надвигающемся продвижении фазы и для управления завершением.

**register()** - добавляет в этот фазер новую, еще не подготовленную сторону.

**toString()** - возвращает строку, идентифицирующую этот фазер, а также его состояние.

BlockingQueue

**BlockingQueue** – это интерфейс который позволяет блокировать доступ к коллекции потокам который хотят взять элемент из очереди если в очередь еще нечего не положили. Нужен во избежание ошибок.

**Методы:**

**add(E e)** - вставляет указанный элемент в эту очередь, если это возможно сделать немедленно, не нарушая ограничений емкости, возвращаясь true при успешном выполнении и выдавая сообщение, IllegalStateException если в данный момент нет свободного места.

**contains(Object o)** - возвращает, true если эта очередь содержит указанный элемент.

**drainTo(Collection<? super E> c)** - удаляет все доступные элементы из этой очереди и добавляет их в заданную коллекцию.

**drainTo(Collection<? super E> c, int maxElements)** - удаляет не более заданного количества доступных элементов из этой очереди и добавляет их в заданную коллекцию.

**offer(E e)** - вставляет указанный элемент в эту очередь, если это возможно сделать немедленно, не нарушая ограничений емкости, возвращаясь true при успешном выполнении и false если в данный момент нет свободного места.

**offer(E e, long timeout, TimeUnit unit)** - вставляет указанный элемент в эту очередь, ожидая до указанного времени ожидания, если необходимо, чтобы освободилось место.

**poll(long timeout, TimeUnit unit)** - извлекает и удаляет начало этой очереди, ожидая до указанного времени ожидания, если необходимо, чтобы элемент стал доступным.

**put(E e)** - вставляет указанный элемент в эту очередь, ожидая, если необходимо, чтобы освободилось место.

**remainingCapacity()** - возвращает количество дополнительных элементов, которые эта очередь в идеале может (при отсутствии ограничений памяти или ресурсов) принимать без блокировки, или Integer.MAX\_VALUE если нет внутреннего ограничения.

**remove(Object o)** - удаляет единственный экземпляр указанного элемента из этой очереди, если он присутствует.

**take()** - извлекает и удаляет начало этой очереди, ожидая, если необходимо, пока элемент не станет доступным.

ThreadFactory

**ThreadFactory** – это интерфейс который позволяет создавать новые потоки по требованию.

**Методы:**

**newThread(Runnable r) -** cоздает новый Thread.

 ThreadLocalRandom

**ThreadLocalRandom –** это класс для генерации случайных чисел, изолированных для текущего потока.

**Методы:**

**current()** - возвращает ThreadLocalRandom объект текущего потока.

**doubles()** - возвращает фактически неограниченный поток псевдослучайных double значений, каждое из которых находится между нулем (включительно) и единицей (исключительно).

**doubles(double randomNumberOrigin, double randomNumberBound)** - возвращает фактически неограниченный поток псевдослучайных double значений, каждое из которых соответствует заданному источнику (включительно) и привязке (исключительно).

**doubles(long streamSize)** - возвращает поток, генерирующий заданное streamSize количество псевдослучайных double значений, каждое из которых находится между нулем (включительно) и единицей (исключительное).

**doubles(long streamSize, double randomNumberOrigin, double randomNumberBound)** - возвращает поток, генерирующий заданное streamSize количество псевдослучайных double значений, каждое из которых соответствует заданному источнику (включительно) и привязке (исключительно).

**ints()** - возвращает фактически неограниченный поток псевдослучайных int значений.

**ints(int randomNumberOrigin, int randomNumberBound)** - возвращает фактически неограниченный поток псевдослучайных int значений, каждое из которых соответствует заданному источнику (включительно) и привязке (исключительно).

**ints(long streamSize)** - возвращает поток, генерирующий заданное streamSize количество псевдослучайных int значений.

**ints(long streamSize, int randomNumberOrigin, int randomNumberBound)** - возвращает поток, генерирующий заданное streamSize количество псевдослучайных int значений, каждое из которых соответствует заданному источнику (включительно) и привязке (исключительно).

**longs()** - возвращает фактически неограниченный поток псевдослучайных long значений.

**longs(long streamSize)** - возвращает поток, генерирующий заданное streamSize количество псевдослучайных long значений.

**longs(long randomNumberOrigin, long randomNumberBound)** - возвращает фактически неограниченный поток псевдослучайных long значений, каждое из которых соответствует заданному источнику (включительно) и привязке (исключительно).

**longs(long streamSize, long randomNumberOrigin, long randomNumberBound) -** возвращает поток, генерирующий заданное streamSize количество псевдослучайных long значений, каждое из которых соответствует заданному источнику (включительно) и привязке (исключительно).

**next(int bits)** - генерирует псевдослучайное число с указанным количеством младших битов.

**nextDouble(double bound)** - возвращает псевдослучайно выбранное double значение между нулем (включительно) и указанной границей (исключительной).

**nextDouble(double origin, double bound)** - возвращает псевдослучайно выбранное double значение между указанным источником (включительно) и указанной границей (исключительно).

**nextFloat(float bound)** - возвращает псевдослучайно выбранное float значение между нулем (включительно) и указанной границей (исключительной).

**nextFloat(float origin, float bound)** - возвращает псевдослучайно выбранное float значение между указанным источником (включительно) и указанной границей (исключительно).

**nextInt(int bound)** - возвращает псевдослучайное, равномерно распределенное int значение между 0 (включительно) и указанным значением (исключительным), полученное из последовательности этого генератора случайных чисел.

**nextInt(int origin, int bound)** - возвращает псевдослучайно выбранное int значение между указанным источником (включительно) и указанной границей (исключительно).

**nextLong(long bound)** - возвращает псевдослучайно выбранное long значение между нулем (включительно) и указанной границей (исключительной).

**nextLong(long origin, long bound**) - возвращает псевдослучайно выбранное long значение между указанным источником (включительно) и указанной границей (исключительно).

**setSeed(long seed)** - выдает ошибку UnsupportedOperationException.

TimeUnit

**TimeUnit –** это класс перечисление который позволяет указать с какими единицами времени мы сейчас работаем.

**Констант перечисления:**

**DAYS -** единица времени, представляющая двадцать четыре часа.

**HOURS -** единица измерения времени, представляющая шестьдесят минут.

**MICROSECONDS -** единица измерения времени, представляющая одну тысячную миллисекунды.

**MILLISECONDS -** единица измерения времени, представляющая одну тысячную секунды.

**MINUTES -** единица измерения времени, представляющая шестьдесят секунд.

**NANOSECONDS -** единица измерения времени, представляющая одну тысячную долю микросекунды.

**SECONDS -** единица измерения времени, представляющая одну секунду.

**Методы:**

**convert(long sourceDuration, TimeUnit sourceUnit) -** преобразует заданную продолжительность времени в заданной единице в эту единицу.

**convert(Duration duration) -** преобразует заданную продолжительность времени в эту единицу.

**of(ChronoUnit chronoUnit) -** преобразует ChronoUnit к эквивалентному TimeUnit.

**sleep(long timeout) -** выполняет Thread.sleep используя эту единицу времени.

**timedJoin(Thread thread, long timeout) -** выполняет синхронизированный Thread.join используя эту единицу времени.

**timedWait(Object obj, long timeout) -** выполняет синхронизацию Object.wait с использованием этой единицы измерения времени.

**toChronoUnit() -** преобразует это TimeUnit к эквивалентному ChronoUnit.

**toDays(long duration) -** эквивалентно DAYS.convert(duration, this).

**toHours(long duration) -** эквивалентно HOURS.convert(duration, this).

**toMicros(long duration) -** эквивалентно MICROSECONDS.convert(duration, this).

**toMillis(long duration) -** эквивалентно MILLISECONDS.convert(duration, this).

**toMinutes(long duration) -** эквивалентно MINUTES.convert(duration, this).

**toNanos(long duration) -** эквивалентно NANOSECONDS.convert(duration, this).

**toSeconds(long duration) -** эквивалентно SECONDS.convert(duration, this).

**valueOf(String name) -** возвращает константу enum этого класса с указанным именем.

**values() -** возвращает массив, содержащий константы этого класса enum, в порядке их объявления.

CompletableFuture

**CompletableFuture** – это класс для асинхронной работы. Он дает возможность комбинировать шаги обработки результата, соединяя их в цепочку. Как и Future, использует дженерики для указания типа возвращаемого объекта.

**Методы:**

**acceptEither(CompletionStage<? extends T> other, Consumer<? super T> action)** - возвращает новый CompletionStage, который, когда этот или другой данный этап завершается нормально, выполняется с соответствующим результатом в качестве аргумента предоставленного действия.

**acceptEitherAsync(CompletionStage<? extends T> other, Consumer<? super T> action)** - возвращает новый CompletionStage, который, когда этот или другой данный этап завершается нормально, выполняется с использованием средства асинхронного выполнения этого этапа по умолчанию, с соответствующим результатом в качестве аргумента для предоставленного действия.

**acceptEitherAsync(CompletionStage<? extends T> other, Consumer<? super T> action, Executor executor)** - возвращает новый CompletionStage, который, когда этот или другой данный этап завершается нормально, выполняется с использованием предоставленного исполнителя с соответствующим результатом в качестве аргумента предоставленного действия.

**allOf(CompletableFuture<?>... cfs)** - возвращает новое завершаемое будущее, которое завершается после завершения всех заданных завершаемых будущих.

**anyOf(CompletableFuture<?>... cfs)** - возвращает новое завершаемое будущее, которое завершается при завершении любого из заданных завершаемых будущих с тем же результатом.

**applyToEither(CompletionStage<? extends T> other, Function<? super T,U> fn)** - возвращает новый CompletionStage, который, когда этот или другой данный этап завершается нормально, выполняется с соответствующим результатом в качестве аргумента предоставленной функции.

**applyToEitherAsync(CompletionStage<? extends T> other, Function<? super T,U> fn)** - возвращает новый CompletionStage, который, когда этот или другой данный этап завершается нормально, выполняется с использованием средства асинхронного выполнения этого этапа по умолчанию, с соответствующим результатом в качестве аргумента предоставленной функции.

**applyToEitherAsync(CompletionStage<? extends T> other, Function<? super T,U> fn, Executor executor)** - возвращает новый CompletionStage, который, когда этот или другой данный этап завершается нормально, выполняется с использованием предоставленного исполнителя с соответствующим результатом в качестве аргумента предоставленной функции.

**cancel(boolean mayInterruptIfRunning)** - если оно еще не завершено, завершите это завершаемое будущее с помощью CancellationException.

**complete(T value)** - если оно еще не завершено, устанавливает значение, возвращаемое get() и связанными с ним методами, в заданное значение.

**completeAsync(Supplier<? extends T> supplier)** - завершает это завершаемое будущее результатом заданной функции поставщика, вызванной из асинхронной задачи с использованием исполнителя по умолчанию.

**completeAsync(Supplier<? extends T> supplier, Executor executor)** - завершает это завершаемое будущее результатом заданной функции поставщика, вызванной из асинхронной задачи с использованием заданного исполнителя.

**completedFuture(U value)** - возвращает новое завершаемое будущее, которое уже завершено с заданным значением.

**completedStage(U value)** - возвращает новый этап завершения, который уже завершен с заданным значением и поддерживает только те методы в интерфейсе CompletionStage.

**completeExceptionally(Throwable ex)** - если он еще не завершен, вызывает вызовы get() и связанных с ними методов для генерирования данного исключения.

**completeOnTimeout(T value, long timeout, TimeUnit unit)** - завершает это завершаемое будущее с заданным значением, если иным образом не завершено до истечения заданного времени ожидания.

**copy()** - возвращает новое завершаемое будущее, которое обычно завершается с тем же значением, что и это завершаемое будущее, когда оно завершается нормально.

**defaultExecutor()** - возвращает исполнителя по умолчанию, используемого для асинхронных методов, которые не указывают исполнителя.

**delayedExecutor(long delay, TimeUnit unit)** - возвращает нового исполнителя, который отправляет задачу исполнителю по умолчанию после заданной задержки (или без задержки, если значение не является положительным).

**delayedExecutor(long delay, TimeUnit unit, Executor executor)** - возвращает нового исполнителя, который отправляет задачу данному базовому исполнителю после заданной задержки (или без задержки, если она не является положительной).

**exceptionally(Function<Throwable,? extends T> fn)** - возвращает новый CompletionStage, который, когда этот этап завершается в исключительных случаях, выполняется с исключением этого этапа в качестве аргумента предоставленной функции.

**failedFuture(Throwable ex**) - возвращает новое завершаемое будущее, которое уже завершено в исключительных случаях с данным исключением.

**failedStage(Throwable ex)** - возвращает новый этап завершения, который уже завершен исключительно с данным исключением и поддерживает только те методы в интерфейсе CompletionStage.

**get()** - ожидает, если необходимо, завершения этого будущего, а затем возвращает его результат.

**get(long timeout, TimeUnit unit)** - ожидает, если необходимо, не более заданного времени для завершения этого future, а затем возвращает его результат, если таковой имеется.

**getNow(T valueIfAbsent)** - возвращает результирующее значение (или выдает любое встреченное исключение), если оно завершено, else возвращает заданное значениеifabsent.

**getNumberOfDependents()** - возвращает предполагаемое количество завершаемых объектов, завершения которых ожидают завершения этого завершаемого объекта.

**handle(BiFunction<? super T,Throwable,? extends U> fn)** - возвращает новый CompletionStage, который, когда этот этап завершается нормально или в исключительных случаях, выполняется с результатом этого этапа и исключением в качестве аргументов предоставленной функции.

**handleAsync(BiFunction<? super T,Throwable,? extends U> fn)** - возвращает новый CompletionStage, который при обычном или исключительном завершении этого этапа выполняется с использованием средства асинхронного выполнения этого этапа по умолчанию, с результатом этого этапа и исключением в качестве аргументов предоставленной функции.

**handleAsync(BiFunction<? super T,Throwable,? extends U> fn, Executor executor)** - возвращает новый CompletionStage, который, когда этот этап завершается нормально или в исключительных случаях, выполняется с использованием предоставленного исполнителя, с результатом этого этапа и исключением в качестве аргументов предоставленной функции.

**isCancelled()** - возвращает, true если это завершаемое будущее было отменено до его нормального завершения.

**isCompletedExceptionally()** - возвращает true, если это завершаемое будущее завершено в исключительных случаях каким-либо образом.

**isDone()** - возвращает true, если завершено любым способом: обычно, в исключительных случаях или путем отмены.

**join()** - возвращает результирующее значение по завершении или выдает (непроверенное) исключение, если выполнено в исключительных случаях.

**minimalCompletionStage()** - возвращает новый CompletionStage, который обычно завершается с тем же значением, что и этот CompletableFuture, когда он завершается нормально, и не может быть независимо завершен или иным образом использован способами, не определенными методами интерфейса CompletionStage.

**newIncompleteFuture()** - возвращает новый неполный CompletableFuture типа, который должен быть возвращен методом CompletionStage.

**obtrudeException(Throwable ex**) - принудительно вызывает последующие вызовы метода get() и связанных методов для генерирования данного исключения, независимо от того, завершено оно уже или нет.

**obtrudeValue(T value)** - принудительно устанавливает или сбрасывает значение, впоследствии возвращаемое методом get() и связанными методами, независимо от того, завершено оно уже или нет.

**orTimeout(long timeout, TimeUnit unit**) - в исключительных случаях завершает это завершаемое будущее с помощью TimeoutException, если иное не завершено до истечения заданного времени ожидания.

**runAfterBoth(CompletionStage<?> other, Runnable action)** - Возвращает новый CompletionStage, который, когда этот и другой заданный этапы завершаются нормально, выполняет указанное действие.

**runAfterBothAsync(CompletionStage<?> other, Runnable action)** - возвращает новый CompletionStage, который, когда этот и другой данный этап завершаются нормально, выполняет данное действие, используя средство асинхронного выполнения этого этапа по умолчанию.

**runAfterBothAsync(CompletionStage<?> other, Runnable action, Executor executor)** - возвращает новый CompletionStage, который, когда этот и другой заданный этапы завершаются нормально, выполняет указанное действие с использованием предоставленного исполнителя.

**runAfterEither(CompletionStage<?> other, Runnable action)** - возвращает новый CompletionStage, который, когда этот или другой данный этап завершается нормально, выполняет указанное действие.

**runAfterEitherAsync(CompletionStage<?> other, Runnable action)** - возвращает новый CompletionStage, который, когда этот или другой данный этап завершается нормально, выполняет данное действие, используя средство асинхронного выполнения этого этапа по умолчанию.

**runAfterEitherAsync(CompletionStage<?> other, Runnable action, Executor executor)** - возвращает новый CompletionStage, который, когда этот или другой данный этап завершается нормально, выполняет указанное действие с использованием предоставленного исполнителя.

**runAsync(Runnable runnable)** - возвращает новое завершаемое будущее, которое асинхронно завершается задачей, запущенной в ForkJoinPool.commonPool(), после выполнения данного действия.

**runAsync(Runnable runnable, Executor executor)** - возвращает новое завершаемое будущее, которое асинхронно завершается задачей, запущенной в данном исполнителе, после выполнения данного действия.

**supplyAsync(Supplier<U> supplier)** - возвращает новое завершаемое будущее, которое асинхронно завершается задачей, выполняемой в ForkJoinPool.commonPool(), со значением, полученным при вызове данного поставщика.

**supplyAsync(Supplier<U> supplier, Executor executor)** - возвращает новое завершаемое будущее, которое асинхронно завершается задачей, запущенной в данном исполнителе, со значением, полученным при вызове данного поставщика.

**thenAccept(Consumer<? super T> action)** - возвращает новый CompletionStage, который, когда этот этап завершается нормально, выполняется с результатом этого этапа в качестве аргумента предоставленного действия.

**thenAcceptAsync(Consumer<? super T> action)** - возвращает новый CompletionStage, который, когда этот этап завершается нормально, выполняется с использованием средства асинхронного выполнения этого этапа по умолчанию, с результатом этого этапа в качестве аргумента для предоставленного действия.

**thenAcceptAsync(Consumer<? super T> action, Executor executor)** - возвращает новый CompletionStage, который, когда этот этап завершается нормально, выполняется с использованием предоставленного исполнителя, с результатом этого этапа в качестве аргумента для предоставленного действия.

**thenAcceptBoth(CompletionStage<? extends U> other, BiConsumer<? super T,? super U> action)** - возвращает новый CompletionStage, который, когда этот и другой заданный этапы завершаются нормально, выполняется с двумя результатами в качестве аргументов предоставленного действия.

**thenAcceptBothAsync(CompletionStage<? extends U> other, BiConsumer<? super T,? super U> action)** - возвращает новый CompletionStage, который, когда этот и другой заданный этап завершаются нормально, выполняется с использованием средства асинхронного выполнения этого этапа по умолчанию, с двумя результатами в качестве аргументов для предоставленного действия.

**thenAcceptBothAsync(CompletionStage<? extends U> other, BiConsumer<? super T,? super U> action, Executor executor)** - возвращает новый CompletionStage, который, когда этот и другой заданный этапы завершаются нормально, выполняется с использованием предоставленного исполнителя с двумя результатами в качестве аргументов для предоставленного действия.

**thenApply(Function<? super T,? extends U> fn)** - возвращает новый CompletionStage, который, когда этот этап завершается нормально, выполняется с результатом этого этапа в качестве аргумента предоставленной функции.

**thenApplyAsync(Function<? super T,? extends U> fn)** - возвращает новый CompletionStage, который, когда этот этап завершается нормально, выполняется с использованием средства асинхронного выполнения этого этапа по умолчанию, с результатом этого этапа в качестве аргумента предоставленной функции.

**thenApplyAsync(Function<? super T,? extends U> fn, Executor executor)** - возвращает новый CompletionStage, который, когда этот этап завершается нормально, выполняется с использованием предоставленного исполнителя, с результатом этого этапа в качестве аргумента предоставленной функции.

**thenCombine(CompletionStage<? extends U> other, BiFunction<? super T,? super U,? extends V> fn)** - возвращает новый CompletionStage, который, когда этот и другой заданный этапы завершаются нормально, выполняется с двумя результатами в качестве аргументов предоставленной функции.

**thenCombineAsync(CompletionStage<? extends U> other, BiFunction<? super T,? super U,? extends V> fn)** - возвращает новый CompletionStage, который, когда этот и другой заданный этап завершаются нормально, выполняется с использованием средства асинхронного выполнения этого этапа по умолчанию, с двумя результатами в качестве аргументов предоставленной функции.

**thenCombineAsync(CompletionStage<? extends U> other, BiFunction<? super T,? super U,? extends V> fn, Executor executor)** - возвращает новый CompletionStage, который, когда этот и другой заданный этапы завершаются нормально, выполняется с использованием предоставленного исполнителя с двумя результатами в качестве аргументов предоставленной функции.

**thenCompose(Function<? super T,? extends CompletionStage<U>> fn)** - Возвращает новый CompletionStage, который завершается с тем же значением, что и CompletionStage, возвращаемый данной функцией.

**thenComposeAsync(Function<? super T,? extends CompletionStage<U>> fn)** - возвращает новый этап завершения, который завершается с тем же значением, что и этап завершения, возвращаемый данной функцией, выполняемый с использованием средства асинхронного выполнения этого этапа по умолчанию.

**thenComposeAsync(Function<? super T,? extends CompletionStage<U>> fn, Executor executor)** - возвращает новый CompletionStage, который завершается с тем же значением, что и CompletionStage, возвращаемый данной функцией, выполняемой с использованием предоставленного исполнителя.

**thenRun(Runnable action)** - возвращает новый CompletionStage, который, когда этот этап завершается нормально, выполняет указанное действие.

**thenRunAsync(Runnable action)** - возвращает новый CompletionStage, который, когда этот этап завершается нормально, выполняет данное действие, используя средство асинхронного выполнения этого этапа по умолчанию.

**thenRunAsync(Runnable action, Executor executor)** - возвращает новый CompletionStage, который, когда этот этап завершается нормально, выполняет данное действие с использованием предоставленного исполнителя.

**toCompletableFuture()** - возвращает это завершаемое будущее.

**toString()** - возвращает строку, идентифицирующую это завершаемое будущее, а также состояние его завершения.

**whenComplete(BiConsumer<? super T,? super Throwable> action)** - возвращает новый CompletionStage с тем же результатом или исключением, что и этот этап, который выполняет указанное действие по завершении этого этапа.

**whenCompleteAsync(BiConsumer<? super T,? super Throwable> action)** - возвращает новый CompletionStage с тем же результатом или исключением, что и этот этап, который выполняет данное действие, используя средство асинхронного выполнения этого этапа по умолчанию, когда этот этап завершается.

**whenCompleteAsync(BiConsumer<? super T,? super Throwable> action, Executor executor)** - возвращает новый CompletionStage с тем же результатом или исключением, что и этот этап, который выполняет данное действие с использованием предоставленного исполнителя по завершении этого этапа.

CompletionStage

**CompletionStage** - это интерфейс который представляет этап возможного асинхронного вычисления, который выполняет действие или вычисляет значение при завершении другого этапа завершения.

**Методы:**

**acceptEither(CompletionStage<? extends T> other, Consumer<? super T> action)** - возвращает новый CompletionStage, который, когда этот или другой данный этап завершается нормально, выполняется с соответствующим результатом в качестве аргумента предоставленного действия.

**acceptEitherAsync(CompletionStage<? extends T> other, Consumer<? super T> action) -** возвращает новый CompletionStage, который, когда этот или другой данный этап завершается нормально, выполняется с использованием средства асинхронного выполнения этого этапа по умолчанию, с соответствующим результатом в качестве аргумента для предоставленного действия.

**acceptEitherAsync(CompletionStage<? extends T> other, Consumer<? super T> action, Executor executor) -** возвращает новый CompletionStage, который, когда этот или другой данный этап завершается нормально, выполняется с использованием предоставленного исполнителя с соответствующим результатом в качестве аргумента предоставленной функции.

**applyToEither(CompletionStage<? extends T> other, Function<? super T,U> fn) -** возвращает новый CompletionStage, который, когда этот или другой данный этап завершается нормально, выполняется с соответствующим результатом в качестве аргумента предоставленной функции.

**applyToEitherAsync(CompletionStage<? extends T> other, Function<? super T,U> fn) -** возвращает новый CompletionStage, который при нормальном завершении этого или другого данного этапа выполняется с использованием средства асинхронного выполнения этого этапа по умолчанию с соответствующим результатом в качестве аргумента предоставленной функции.

**applyToEitherAsync(CompletionStage<? extends T> other, Function<? super T,U> fn, Executor executor) -** возвращает новый CompletionStage, который, когда этот или другой данный этап завершается нормально, выполняется с использованием предоставленного исполнителя с соответствующим результатом в качестве аргумента предоставленной функции.

**exceptionally(Function<Throwable,? extends T> fn) -** возвращает новый CompletionStage, который, когда этот этап завершается в исключительных случаях, выполняется с исключением этого этапа в качестве аргумента предоставленной функции.

**handle(BiFunction<? super T,Throwable,? extends U> fn) -** возвращает новый CompletionStage, который при обычном или исключительном завершении этого этапа выполняется с результатом этого этапа и исключением в качестве аргументов предоставленной функции.

**handleAsync(BiFunction<? super T,Throwable,? extends U> fn) -** возвращает новый CompletionStage, который при обычном или исключительном завершении этого этапа выполняется с использованием средства асинхронного выполнения этого этапа по умолчанию, с результатом этого этапа и исключением в качестве аргументов предоставленной функции.

**handleAsync(BiFunction<? super T,Throwable,? extends U> fn, Executor executor) -** возвращает новый CompletionStage, который при обычном или исключительном завершении этого этапа выполняется с использованием предоставленного исполнителя, с результатом этого этапа и исключением в качестве аргументов предоставленной функции.

**runAfterBoth(CompletionStage<?> other, Runnable action) -** возвращает новый этап завершения, который, когда этот и другой заданный этапы завершаются нормально, выполняет указанное действие.

**runAfterBothAsync(CompletionStage<?> other, Runnable action) -** возвращает новый CompletionStage, который, когда этот и другой данный этап завершаются нормально, выполняет данное действие, используя средство асинхронного выполнения этого этапа по умолчанию.

**runAfterBothAsync(CompletionStage<?> other, Runnable action, Executor executor) -** возвращает новый CompletionStage, который, когда этот и другой заданный этапы завершаются нормально, выполняет указанное действие с использованием предоставленного исполнителя.

**runAfterEither(CompletionStage<?> other, Runnable action) -** возвращает новый CompletionStage, который, когда этот или другой данный этап завершается нормально, выполняет указанное действие.

**runAfterEitherAsync(CompletionStage<?> other, Runnable action) -** возвращает новый CompletionStage, который, когда этот или другой данный этап завершается нормально, выполняет указанное действие, используя средство асинхронного выполнения этого этапа по умолчанию.

**runAfterEitherAsync(CompletionStage<?> other, Runnable action, Executor executor) -** возвращает новый CompletionStage, который, когда этот или другой данный этап завершается нормально, выполняет указанное действие с использованием предоставленного исполнителя.

**thenAccept(Consumer<? super T> action) -** возвращает новый CompletionStage, который, когда этот этап завершается нормально, выполняется с результатом этого этапа в качестве аргумента предоставленного действия.

**thenAcceptAsync(Consumer<? super T> action) -** возвращает новый CompletionStage, который, когда этот этап завершается нормально, выполняется с использованием средства асинхронного выполнения этого этапа по умолчанию, с результатом этого этапа в качестве аргумента для предоставленного действия.

**thenAcceptAsync(Consumer<? super T> action, Executor executor) -** возвращает новый CompletionStage, который, когда этот этап завершается нормально, выполняется с использованием предоставленного исполнителя, с результатом этого этапа в качестве аргумента для предоставленного действия.

**thenAcceptBoth(CompletionStage<? extends U> other, BiConsumer<? super T,? super U> action) -** возвращает новый CompletionStage, который, когда этот и другой заданный этапы завершаются нормально, выполняется с двумя результатами в качестве аргументов предоставленного действия.

**thenAcceptBothAsync(CompletionStage<? extends U> other, BiConsumer<? super T,? super U> action) -** возвращает новый CompletionStage, который, когда этот и другой заданный этап завершаются нормально, выполняется с использованием средства асинхронного выполнения этого этапа по умолчанию, с двумя результатами в качестве аргументов для предоставленного действия.

**thenAcceptBothAsync(CompletionStage<? extends U> other, BiConsumer<? super T,? super U> action, Executor executor) -** возвращает новый CompletionStage, который, когда этот и другой заданный этапы завершаются нормально, выполняется с использованием предоставленного исполнителя, с двумя результатами в качестве аргументов предоставленной функции.

**thenApply(Function<? super T,? extends U> fn) -** возвращает новый CompletionStage, который, когда этот этап завершается нормально, выполняется с результатом этого этапа в качестве аргумента предоставленной функции.

**thenApplyAsync(Function<? super T,? extends U> fn) -** возвращает новый CompletionStage, который, когда этот этап завершается нормально, выполняется с использованием средства асинхронного выполнения этого этапа по умолчанию, с результатом этого этапа в качестве аргумента предоставленной функции.

**thenApplyAsync(Function<? super T,? extends U> fn, Executor executor) -** возвращает новый CompletionStage, который, когда этот этап завершается нормально, выполняется с использованием предоставленного исполнителя, с результатом этого этапа в качестве аргумента предоставленной функции.

**thenCombine(CompletionStage<? extends U> other, BiFunction<? super T,? super U,? extends V> fn) -** возвращает новый CompletionStage, который, когда этот и другой заданный этапы завершаются нормально, выполняется с двумя результатами в качестве аргументов предоставленной функции.

**thenCombineAsync(CompletionStage<? extends U> other, BiFunction<? super T,? super U,? extends V> fn) -** возвращает новый CompletionStage, который, когда этот и другой заданный этап завершаются нормально, выполняется с использованием средства асинхронного выполнения этого этапа по умолчанию, с двумя результатами в качестве аргументов предоставленной функции.

**thenCombineAsync(CompletionStage<? extends U> other, BiFunction<? super T,? super U,? extends V> fn, Executor executor) -** возвращает новый CompletionStage, который, когда этот и другой заданный этапы завершаются нормально, выполняется с использованием предоставленного исполнителя, с двумя результатами в качестве аргументов предоставленной функции.

**thenCompose(Function<? super T,? extends CompletionStage<U>> fn) -** возвращает новый CompletionStage, который, когда этот этап завершается нормально, выполняется с использованием этого этапа в качестве аргумента предоставленной функции.

**thenComposeAsync(Function<? super T,? extends CompletionStage<U>> fn) -** возвращает новый CompletionStage, который, когда этот этап завершается нормально, выполняется с использованием средства асинхронного выполнения этого этапа по умолчанию, с this stage в качестве аргумента предоставленной функции.

**thenComposeAsync(Function<? super T,? extends CompletionStage<U>> fn, Executor executor) -** возвращает новый CompletionStage, который, когда этот этап завершается нормально, выполняется с использованием предоставленного исполнителя, с результатом этого этапа в качестве аргумента предоставленной функции.

**thenRun(Runnable action) -** возвращает новый этап завершения, который, когда этот этап завершается нормально, выполняет указанное действие.

**thenRunAsync(Runnable action) -** возвращает новый CompletionStage, который, когда этот этап завершается нормально, выполняет данное действие, используя средство асинхронного выполнения этого этапа по умолчанию.

**thenRunAsync(Runnable action, Executor executor) -** возвращает новый CompletionStage, который, когда этот этап завершается нормально, выполняет данное действие с использованием предоставленного исполнителя.

**toCompletableFuture() -** возвращает a CompletableFuture с сохранением тех же свойств завершения, что и на этом этапе.

**whenComplete(BiConsumer<? super T,? super Throwable> action) -** возвращает новый CompletionStage с тем же результатом или исключением, что и этот этап, который выполняет указанное действие по завершении этого этапа.

**whenCompleteAsync(BiConsumer<? super T,? super Throwable> action) -** возвращает новый CompletionStage с тем же результатом или исключением, что и на этом этапе, который выполняет данное действие, используя средство асинхронного выполнения по умолчанию на этом этапе, когда этот этап завершается.

**whenCompleteAsync(BiConsumer<? super T,? super Throwable> action, Executor executor) -** возвращает новый CompletionStage с тем же результатом или исключением, что и этот этап, который выполняет данное действие с использованием предоставленного исполнителя по завершении этого этапа.