**THREADS**

Create and extend class **Thread**, then override method **run()**, where all functions start

Create new exemplar of this class and run method **start();** to create thread

**Thread.sleap(…)** – static function to stop current thread (put in try/catch with **InterraptedExeption**)

**Thread.*yield*();** - throw current thread from the running to the runnable

**.join()** – connect this thread to the queue, that’s mean ask him to wait

**Synchronized(this** or variable**){…}** – change current function

**.wait()** – ask to stop thread in this variable (connect with function on the top)

**.notify()** – start all threads in this synchronized line (connect with function on the top)

**Synchronized** – add it before name of the function to ask the function to finish before let another thread make it

**Volatile** – keyword is used to mark a Java variable as "being stored in main memory". More precisely that means, that every read of a volatile variable will be read from the computer's main memory, and not from the CPU cache, and that every write to a volatile variable will be written to main memory, and not just to the CPU cache.

**… .setPrioryty(1-10 of Thread. …)** – set priority

**ThreadLocalRandom.*current*().next...()** – random value in treads

There are special variables for threads begins with **Atomic…** for example **AtomicInteger** … = new **AtomicInteger(…)** – to create value and it has lots of methods for threads, without any bugs (synchronized)

To close the thread

**Lock** lock **= new ReentrantLock();** - create lock variable

lock**.lock();** - lock thread

lock**.unlock();** - unlock thread

lock**.tryLock()** – checks if it is possible to lock

**ReadWriteLock** readWriteLock **= new ReentrantReadWriteLock();  
Lock** read **= readWriteLock.readLock();** - lock reading from variables **Lock** write **= readWriteLock.writeLock();** - lock writing

**Condition** *condition***=** *lock***.newCondition();** - create condition variable

*condition***.await();** - pause this thread

*condition***.signal();** - continue paused thread

to control threads better

**ExecutorService** executorService **= Executors.*newFixedThreadPool*(…);** - create variable, which can control threads better then common caseexecutorService**.submit(…);** - start thread insede the classexecutorService**.shutdown();** - end threads after all functionsexecutorService**.shutdownNow()** – end threads now

To start thread after some time

**ScheduledExecutorService** scheduledExecutorService **= Executors.*newSingleThreadScheduledExecutor*();** - create scheduled variable, which can start thread after some time scheduledExecutorService**.schedule(**new MyRunnable()**, …, TimeUnit.***SECONDS***);** - start thread after some time

(Starting function, time, kind of time)

To create variable which limits number of threads

**Semaphore … = new Semaphore(**num of threads**);** - creating value with limited number of threads

**… .acquire();** - requiring one free box in this thread if it is or waiting for it

**… .release();** - putting of the box

Pause code until count iterator wont be equals 0

**CountDownLatch** countDownLatch **= new CountDownLatch(…);** - creation variable with limited number

countDownLatch**.countDown();** - decrease number in variable

countDownLatch**.await();** - this works as a door, which means that all code after the stars only when number, which has been placed in constructor, will be equals to 0;

if you want to swap the values between threads

**Exchanger<…> … = new Exchanger<>();** - create variable, which can swap variables between treads

**… .exchange(…)** – method, which swaps values with another thread, it will be waiting till the same variable will call the same method to swap with it

If you want to start thread after something

**CyclicBarrier … = new CyclicBarrier(**number**,** start thread**)** – creating variable which runs method when its iterator = 0

**… .await()** – tells variable to decrease its iterator, when it is equals 0, the thread in constructor will start

If you need to wait till another method rich some part of the code

**Phaser … = new Phaser(…);** - create variable and include count of variables, which has to call method in the bottom

**… .arriveAndAwaitAdvance();** - stop function and wait till another treads push this method

**ForkJoinPool** pool **= new ForkJoinPool(***number of cores***);** - create variable with number of cores

pool**.invoke(…);** - start thread in this variable ( this thread has to extend **RecursiveTask<…>** )

**… .fork();** - connect created thread to the others (call from thread’s variable)