ТЕХНОАТОМ

Java

Java Concurrency

Александр Помосов

Reminder

ТЕХНОАТОМ

Отметьтесь на портале

Reminder

ТЕХНОАТОМ

Обновите репозиторий

Agenda

TEXHOATOM

- Multithreading basics
- Concurrency challenges
- java.util.concurrent
- Thread-safety recipes

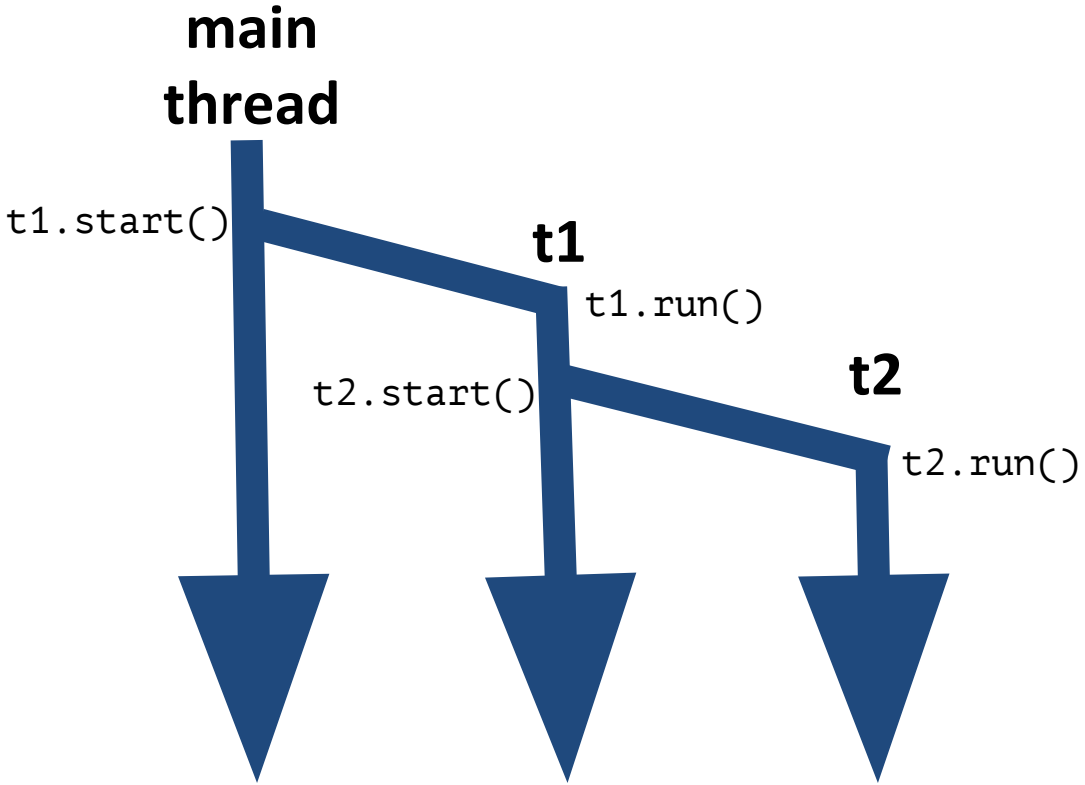
Agenda

TEXHOATOM

- Multithreading basics
- Concurrency challenges
- java.util.concurrent
- Thread-safety recipes

Threads revisited

TEXHOATOM



Operating System role

TEXHOATOM

- Creates threads (clone syscall)
- Schedules threads (context switch)
- Provides api for Thread management

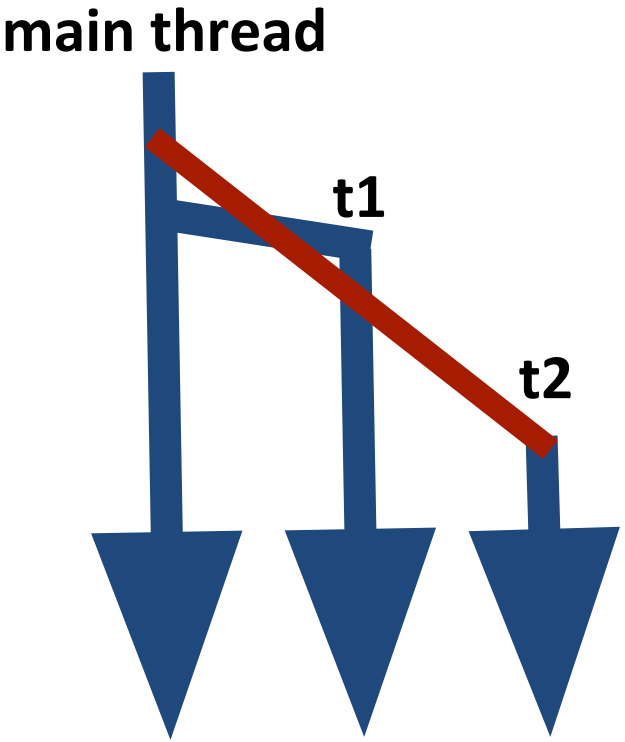
Behaviour of multithreaded program is (inter alia) dependent on **OS scheduling**

Threads start example

TEXHOATOM

The order in which threads start
is not defined
and is dependent on
OS scheduling

```
@see races.RandomRunExample
```



Agenda

TEXHOATOM

- Multithreading basics
- Concurrency challenges
- java.util.concurrent
- Thread-safety recipes

Challenge 1. Race condition

TEXHOATOM

Race condition (состояние гони, гонка)

program behaviour where the output is dependent on the sequence or timing of other uncontrollable events

Parallel programs are racy by nature, some races may be errors.

```
@see races
```

deadly race: https://ru.wikipedia.org/wiki/Therac-25

Challenge 2. Data races

TEXHOATOM

Data race

- two or more threads in a **single process** access the same memory location concurrently, and
- at least one of the accesses is for writing

`@see data_races`

Solution - allow only one thread to access data at a time

Concurrency visualization

TEXHOATOM

Java concurrent visualization

```
cd lecture9/
java -jar javaConcurrentAnimated.jar
```

Mutex

TEXHOATOM

Mutex (mutual exclusion)/lock

mechanism that allow only one thread to enter ‘critical section’ (block of code that must be executed only by one thread at a time). That thread acquires lock

Reentrant lock

lock, that can be acquired by single thread multiple times

@see javaConcurrentAnimated.jar (ReentrantLock)

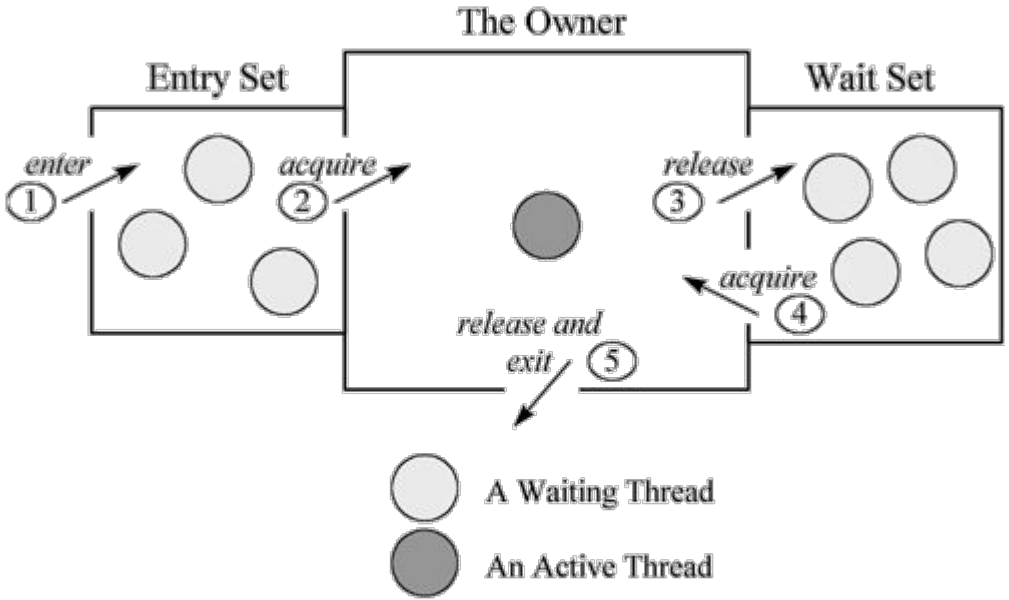
Monitor

TEXHOATOM

Monitor

(mutex + entry set)

Only one thread at a time may own a monitor. Any other threads attempting to lock that monitor are blocked until they can obtain a lock on that monitor.



Java Object internal monitors

TEXHOATOM

In java every Object has internal monitor.
That is, every Object can act as a lock.

@see javaConcurrentAnimated.jar (synchronized)

@see synchronized_example

Let’s fix data_races example - make increment a critical section
@see data_races **(2 balls)**

Challenge 3. Deadlock

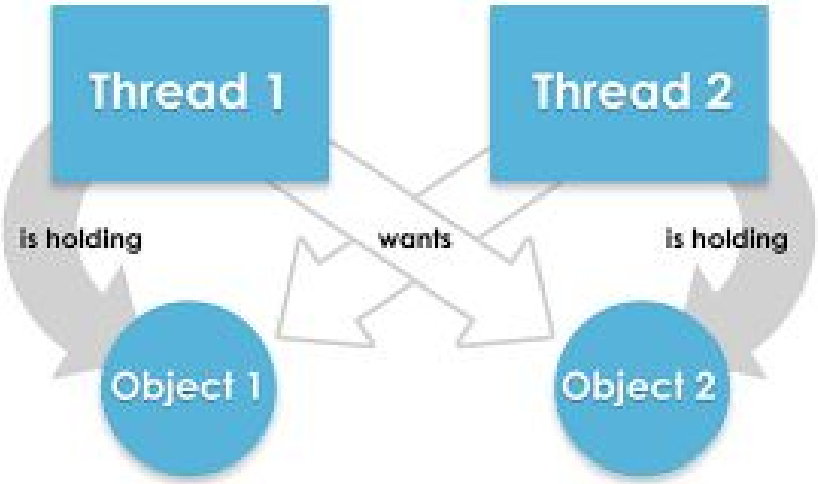
TEXHOATOM

Deadlock

a state in which each member of a group of actions, is waiting for some other member to release a lock

(Do not forget **starvation** and **livelock**)

@see deadlock



Java Object internal monitors

TEXHOATOM

What if I want to control monitor - to control internal monitor from program?

Internal Object monitor have '**wait set**' and methods for controlling waiting threads:

Object.wait(), Object.wait(timeout) - current thread releases the monitor and enters wait set

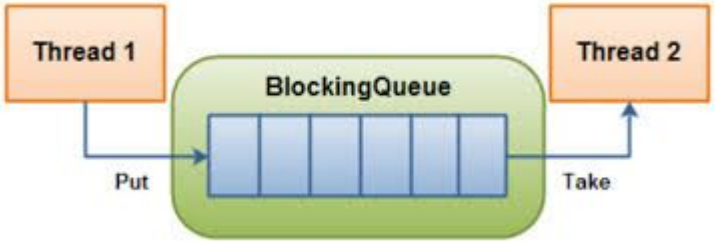
Object.notify(), Object.notifyAll() - removes random (all) threads from wait set into blocking set

```
@see java.lang.Object
```

Blocking queue example

TEXHOATOM

Blocking queue blocks reading thread, when there is nothing to read. And blocks writing thread, when it is full




```
@see javaConcurrentAnimated.jar
(BlockingQueue)
@see blocking_queue
```

Inter-thread communication

TEXHOATOM

Threads (unlike processes) can communicate via **shared memory** (shared variables/shared mutable state)

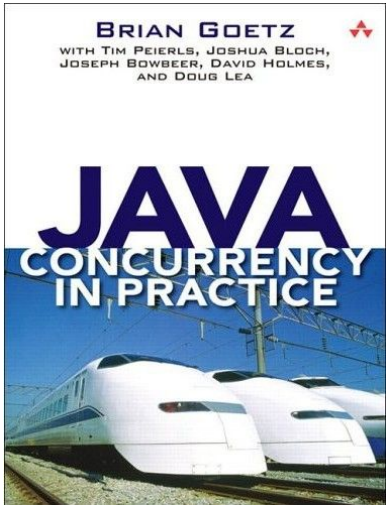
If we have reads and writes to shared mutable variables, that affect each other, we call it **concurrent access**

Oracle guide to concurrency in Java
https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

Thread-safety

TEXHOATOM

A class is **thread-safe** if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code



(from JCiP)
https://www.amazon.com/Java-Concurrency-Practice-Brian-Goetz/dp/0321349601

No state - no problem

TEXHOATOM

It is **data** that must be protected from concurrent access. Even thought '**synchronized'** about code.
Every access to protected data must be synchronized (every read and write), else program is not properly synchronized.

Stateless objects are always thread-safe

So let's avoid state! **(no, impossible in practice)**

Shared mutable state

TEXHOATOM

Concurrency appear when shared mutable state is accessed from several threads

Let's avoid concurrency!
(really good idea!)

Immutable and unshared objects are always **thread-safe**

```
@see shared_mutable_state
```

Use immutable state (final)

TEXHOATOM

‘final’ guarantees that after construction reference will be always read properly

- **final** only guarantees immutability for single reference to make object fully-immutable you must mark every reference final, not only root object
- if you change final fields via **reflection** - you lose guarantees

Make final as much shared variables as possible

Unshared state (ThreadLocal)

TEXHOATOM

```
ThreadLocal<Object> locals = new ThreadLocal<Object>();
```

As with final - ThreadLocal only guarantees, that the reference, that is accessed via ThreadLocal variable (‘locals’ in example) is thread local, no in-depth thread locality.

`@see thread_local`

What happen when we have SMS?

TEXHOATOM

This is defined by **Java Memory Model (JMM)**

Java Language Specification (JLS)
Chapter 17. Threads and locks
https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html
(do not read! first look at https://shipilev.net/#jmm)

JMM is tricky to understand and is hard to use directly.

Java Memory Model (JMM)

TEXHOATOM

JMM specifies what can be read by particular read action in program.

More precisely it defines **guarantees** on read/write atomicity, write visibility and instruction ordering.

JMM. Why so complex?

TEXHOATOM

JVM is highly optimized. It is possible because of relatively **weak guarantees** of JMM.

JMM was created as a **trade-off between performance, complexity of JVM and abilities of hardware.**
JMM considered to be one of the most successful memory models.

Recently Introduced C++ Memory Model is highly based on JMM.

Challenge 4. Atomicity

TEXHOATOM

Some operations that are expected to be atomic - are not:

- i++;
- double/long reads and writes on 32 bit systems
- check then act actions:

```
if (!map.containsKey(key)) {
    map.put(key, value);
}
```

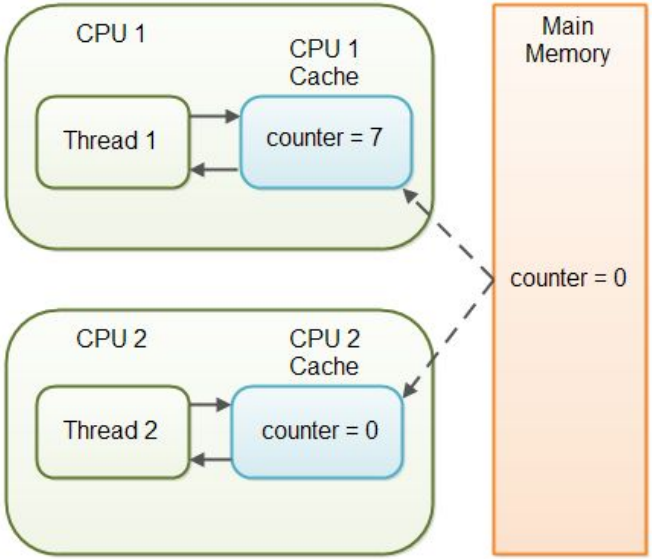
@see data_races

Challenge 5. Visibility

TEXHOATOM

Modern processors have multi-level caches. Thus threads running on different processors may not see changes made by other threads.
It actually depends on cache coherence protocol.
https://en.wikipedia.org/wiki/Cache_coherence
Most modern processors provide coherent caches, so visibility problems are rare

But WORA! `@see visibility`



Challenge 6. Ordering

TEXHOATOM

In sake of performance javac, jit and JVM may change your code whenever it is accepted by Java Memory Model, that is reorder instructions.
After all, processor reorders instructions by himself.
JMM restrict some reorderings.

Challenge 7. Performance

TEXHOATOM

Reasoning about performance of concurrent programs is tricky

@see https://shipilev.net/

Solution - volatile

TEXHOATOM

‘volatile’ means:

- atomic **reads and writes to reference** (not all operations on object)
- reads to volatile variables always return right value
- reads and writes of volatile variables can not be reordered
- ‘happens-before’ relation

(If you make all references in your program ‘volatile’, there will be no data races)

`@see volatile_example`

Agenda

TEXHOATOM

- Multithreading basics
- Concurrency challenges
- java.util.concurrent
- Thread-safety recipes

java.util.concurrent

TEXHOATOM

It is hard to reason low-level JMM categories, but there are a number of high-level constructions in JDK

Atomics

TEXHOATOM

Atomics provide non-blocking operations on common objects. Also provides methods for atomic **‘check then act’** operations (compareAndSet, incrementAndGet)

```
@see data_races/Stopper.java
@see javaConcurrentAnimated.jar
(AtomicInteger)
```

Concurrent Collections

TEXHOATOM

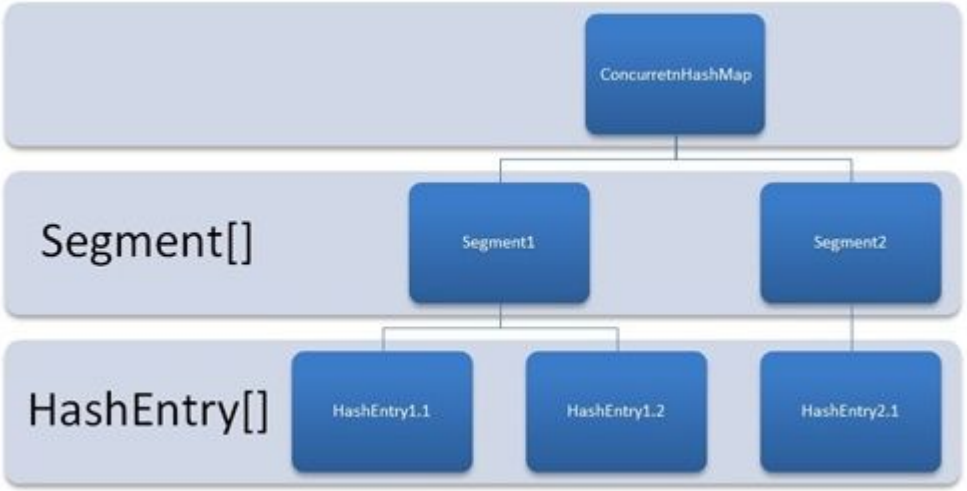
ConcurrentHashMap
BlockingQueue

```
@see javaConcurrentAnimated.jar
(ConcurrentHashMap, BlockingQueue)
```

ConcurrentHashMap

TEXHOATOM

A hash table supporting **full concurrency** of retrievals and **high expected concurrency** for updates.

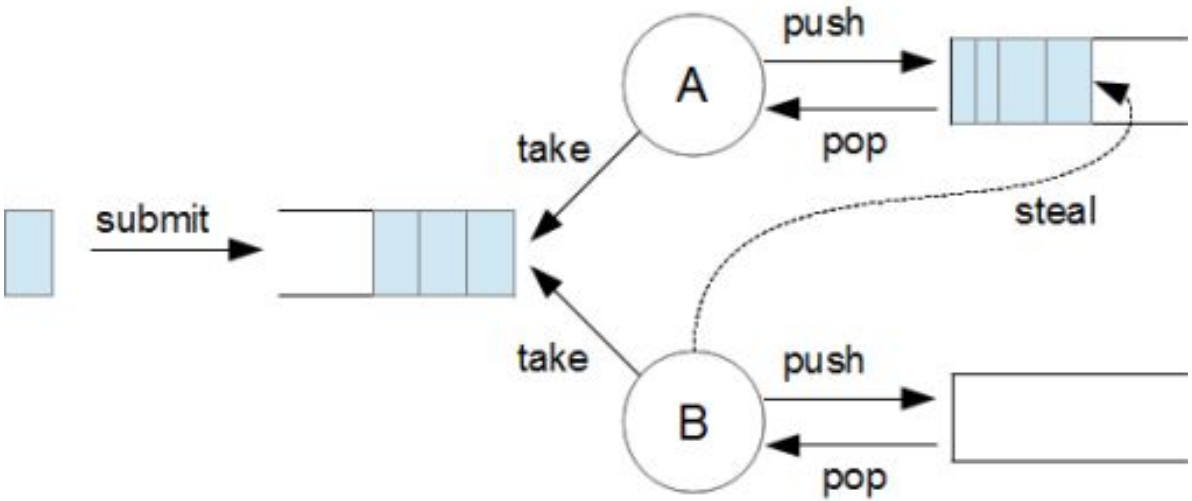


Iteration over **copy** of collection at some point

https://habrahabr.ru/post/132884/
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html

ForkJoinPool

TEXHOATOM




```
@see javaConcurrentAnimated.jar
```

Futures

TEXHOATOM

Future is implementation of **‘promises’**.
A Future represents the result of an asynchronous computation
We can block on **get()** until the result is ready.

```
@see javaConcurrentAnimated.jar (Future)
```

Synchronizers

TEXHOATOM

It is hard to reason low-level JMM categories, but there are a number of high-level constructions in JDK

```
@see javaConcurrentAnimated.jar
(CyclicBarrier, Phaser)
```

Good manual with visualization
https://habrahabr.ru/post/277669/

Agenda

TEXHOATOM

- Multithreading basics
- Concurrency challenges
- java.util.concurrent
- Thread-safety recipes

Recipe 1. Unshared immutable state

TEXHOATOM

- Don't share the state variable across threads
- Make the state variable immutable
- Use synchronization whenever accessing the state variable

Recipe 2. Safe initialization

TEXHOATOM

Do not share ‘this’ from constructor

Bad example ->

```
public class SomeClass{
    private Object v1;
    private final Object v2;
    SomeClass(Object v1, Object v2){
        //’this’ leaks
        //initalization is not completed
        StaticRegistry.register(this);
        this.v1 = v1;
        this.v2 = v2;
    }
}
```

Recipe 3. Safe publication

TEXHOATOM

To publish an object safely, **both the reference to the object and the object's state** must be made visible to other threads at the same time. A properly constructed object can be safely published by:

- Initializing an object reference from a **static initializer**
- Storing a reference to it into a **volatile field** or **AtomicReference**
- Storing a reference to it into a **final field** of a properly constructed object
- Storing a reference to it into a **field that is properly guarded by a lock**

Recipe 4. Use JDK constructions

TEXHOATOM

That to use in real life situations (by priority)

1. concurrent collections/synchronizers/ForkJoinPool
2. synchronized/volatile for reference reads/writes
3. atomics
4. wait/notify
5. volatile ‘happens-before’ magic

(If you are doing 4 or 5 for simple task, maybe you are doing something wrong)

References

TEXHOATOM

java concurrency in practice (signature book for Java Developer)
https://www.amazon.com/Java-Concurrency-Practice-Brian-Goetz/dp/0321349601
Shipilev blog (JMM, concurrency, performance, benchmarks for people, JDK contributor)
https://shipilev.net/
Doug Lea-s home page (java.util.concurrent father and famous spec in concurrency and allocators)
http://g.oswego.edu/
Java Memory Model Pragmatics (best explanation of JMM - available in russian)
https://shipilev.net/#jmm
JMM Under the hood (deep explanation of JMM)
http://gvsmirnov.ru/blog/tech/2014/02/10/jmm-under-the-hood.html
What Every Dev Must Know About Multithreaded Apps (Common knowledge)
https://lyle.smu.edu/~coyle/cse8313/handouts.fall06/s04.msdn.multithreading.pdf
Most active russian community on java, concurrency and related topics
http://razbor-poletov.com/ (podcast)
https://gitter.im/razbor-poletov/razbor-poletov.github.com (chat)

Reasoning about concurrent program

TEXHOATOM

Bugs in concurrent programs are hard to reproduce
Hopefully we have toolchain for analysis of multithreaded programs

jcstress
http://openjdk.java.net/projects/code-tools/jcstress/
(requires JDK9)

ТЕХНОАТОМ

Спасибо за внимание!

Александр Помосов

alpieex@gmail.com