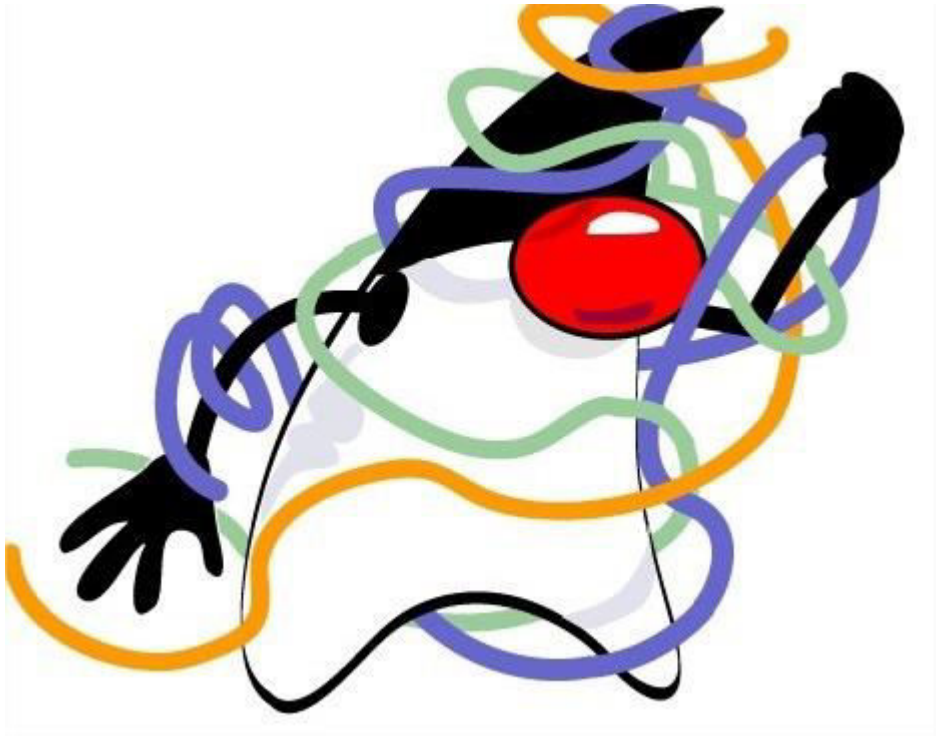
java.util.concurrent.*

Rikhard Goldenberg
SberTech JavaSchool, Mar.2018

CONCURRENCY IN JAVA

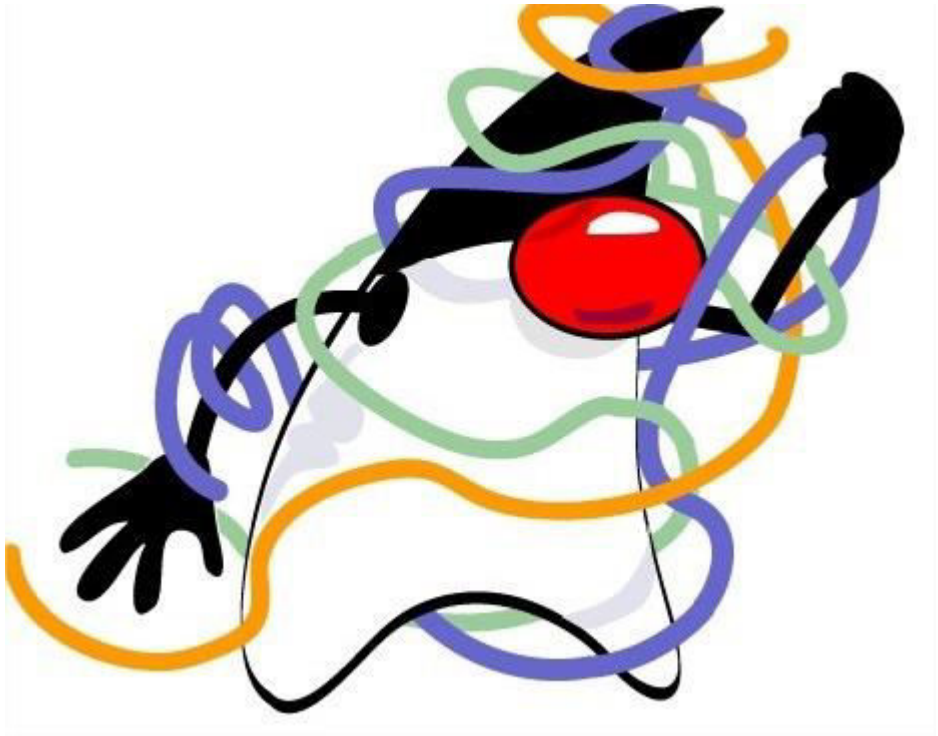


low-level toolbox

- *synchronized*
- *volatile*
- *wait()/notify()/notifyAll()*
- *Thread & Runnable*

  etc.

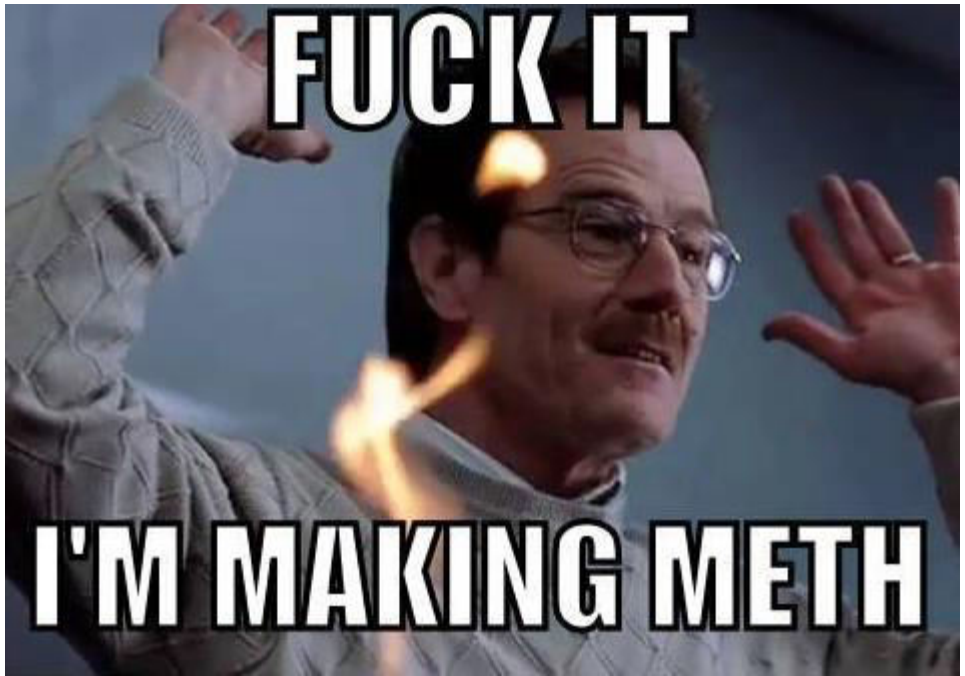
CONCURRENCY IN JAVA



low-level toolbox

- *synchronized*
- *volatile*
- *wait()/notify()/notifyAll()*
- *Thread & Runnable*

etc.



СБЕРБАНК ТЕХНОЛОГИИ

CONCURRENCY ~~IN JAVA~~

OPTIONS

СБЕРБАНК ТЕХНОЛОГИИ

CONCURRENCY ~~IN JAVA~~ IS HARD

OPTION #1

СБЕРБАНК ТЕХНОЛОГИИ

CONCURRENCY ~~IN JAVA~~ IS HARD

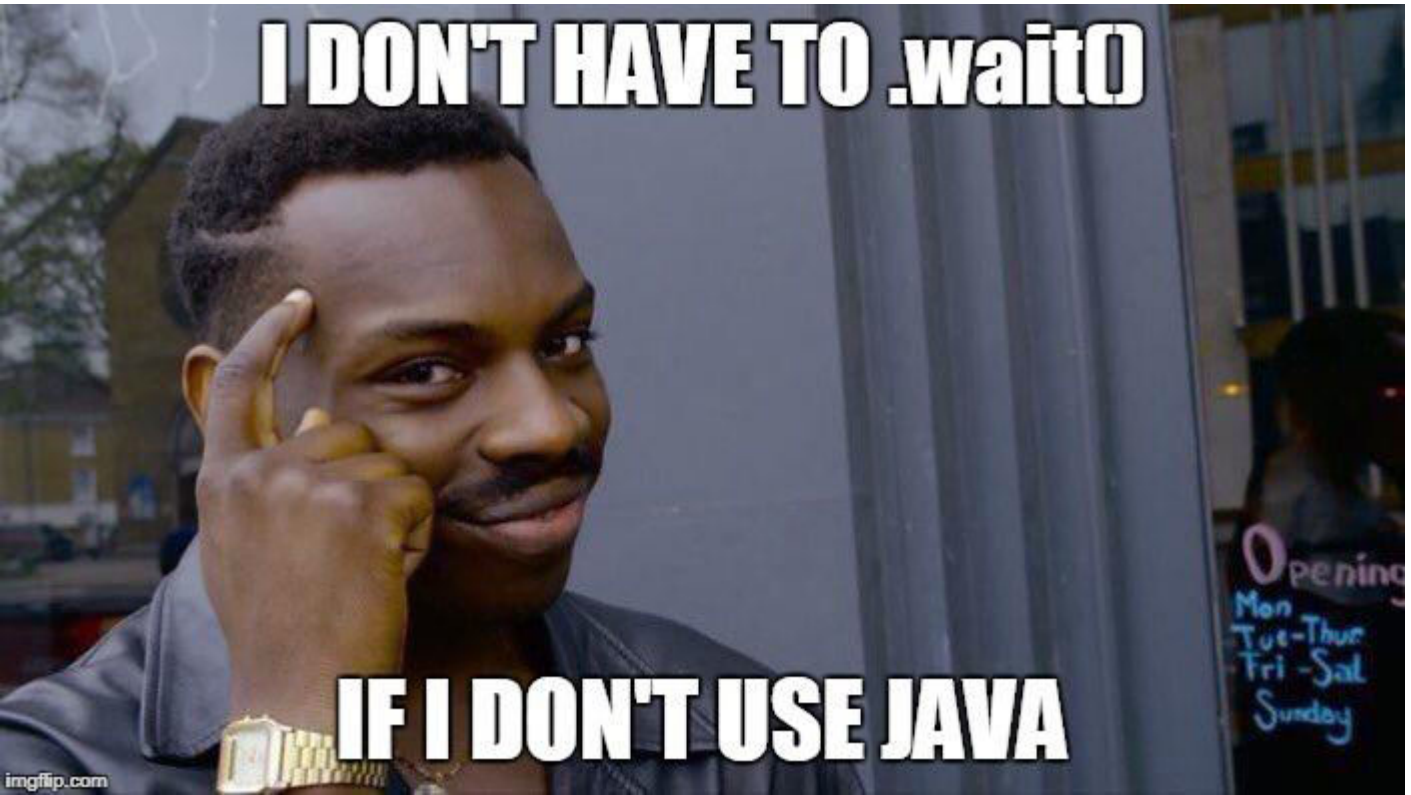
~~OPTION #1~~

BUT WHAT IF

CONCURRENCY *IN JAVA* IS HARD

OPTION #2

Scala

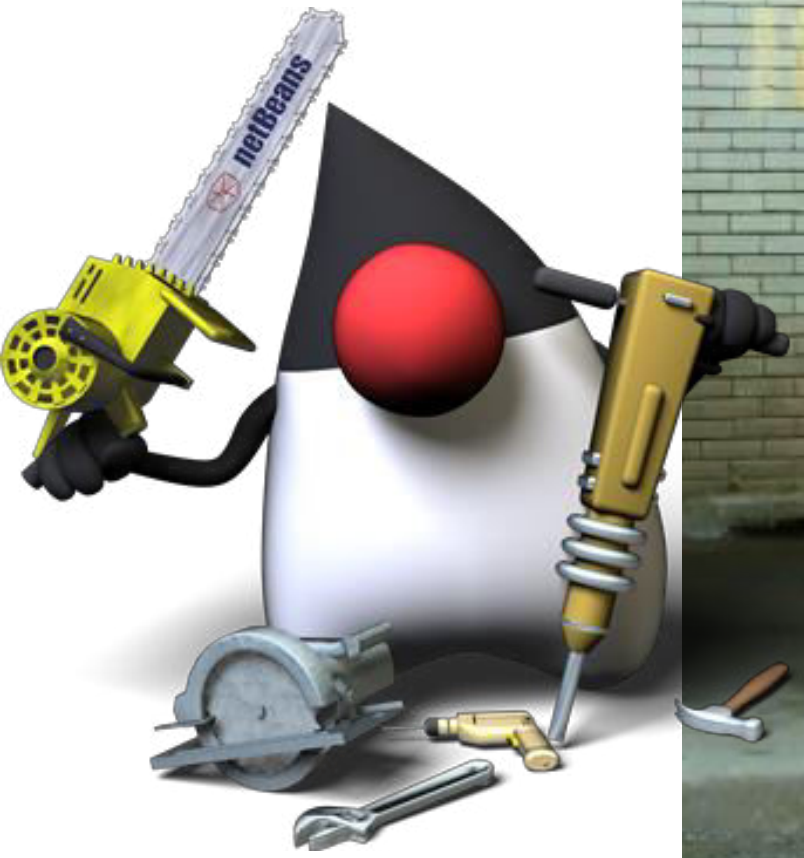


Clojure

СБЕРБАНК ТЕХНОЛОГИИ

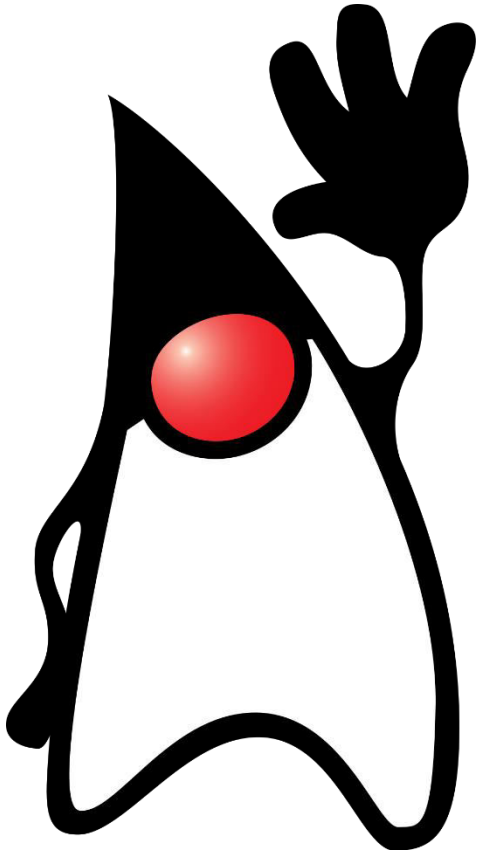
CONCURRENCY *IN JAVA* IS HARD

~~OPTION #2~~



CONCURRENCY *IN JAVA* IS HARD

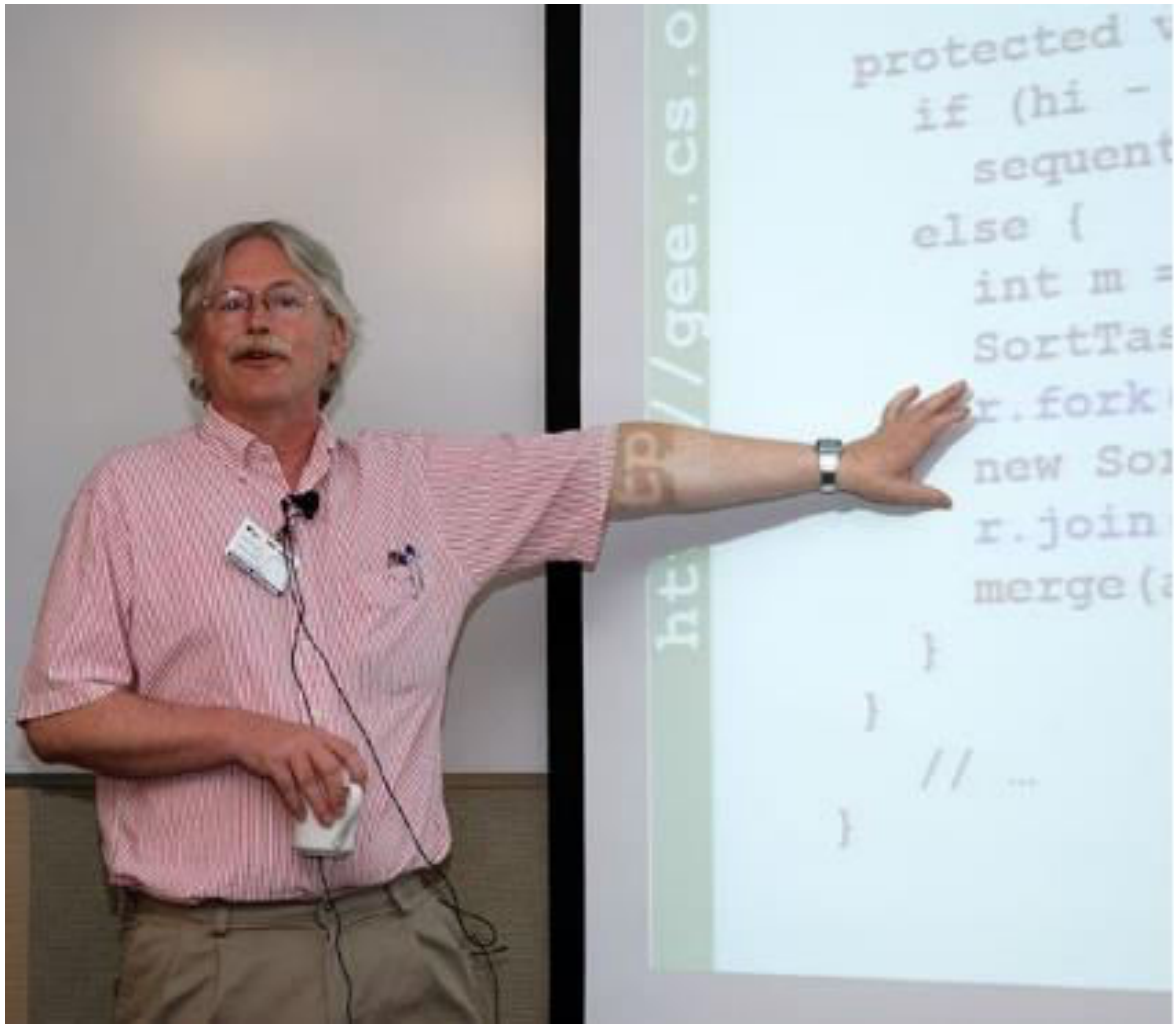
OPTION #3



СБЕРБАНК ТЕХНОЛОГИИ

java.util.concurrent.*

DOUG LEA



СБЕРБАНК ТЕХНОЛОГИИ

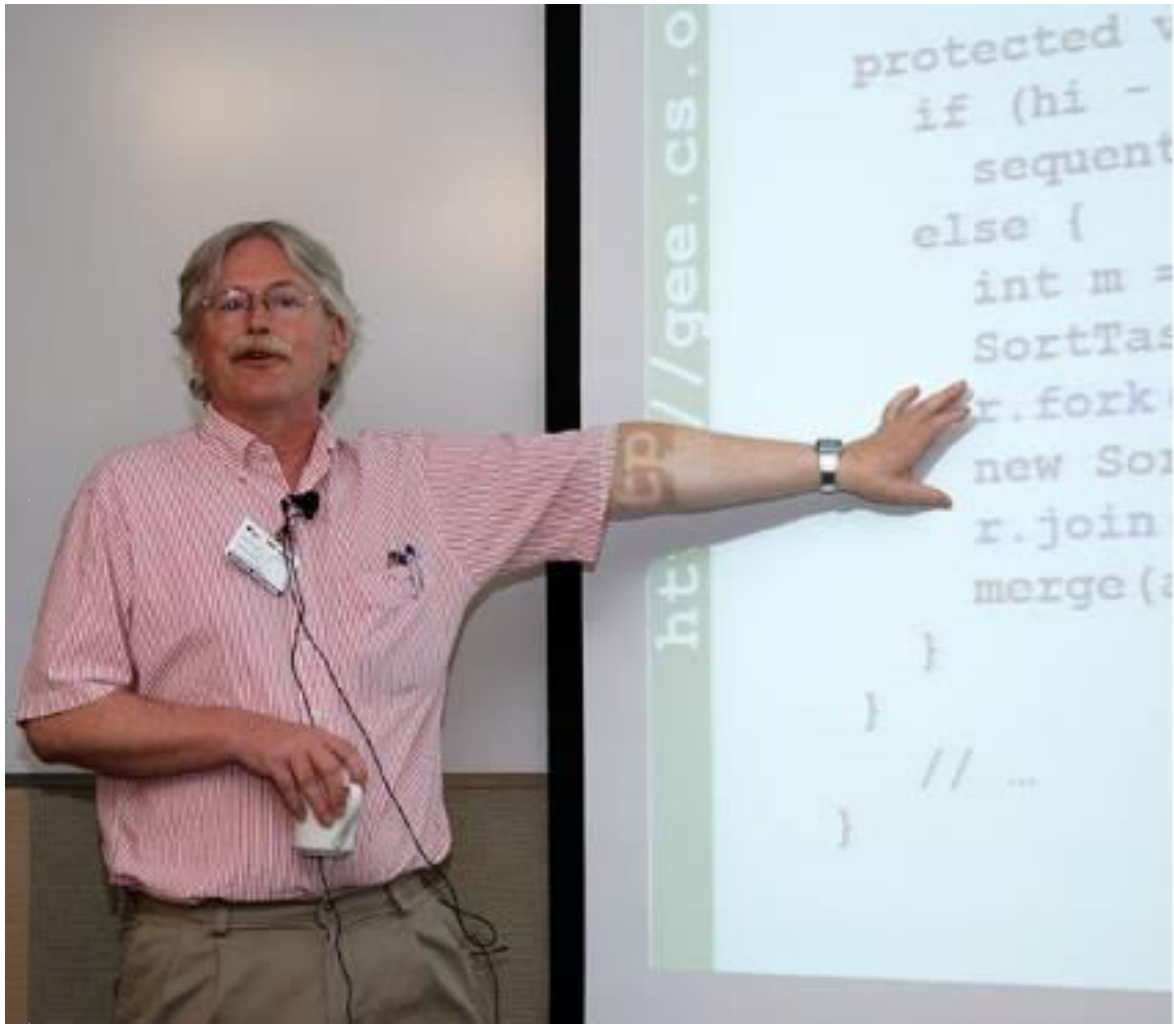
java.util.concurrent.*

concurrency utilities

- since Java 5
- updated in subsequent versions

  - reduced programming effort
  - increased performance
  - increased reliability
  - improved maintainability
  - increased productivity

DOUG LEA



СБЕРБАНК ТЕХНОЛОГИИ

*java.util.concurrent.**

components

- Locks & Atomics
- Concurrent Collections
- Synchronizers
- Executor Framework
- Fork/Join Framework

СБЕРБАНК ТЕХНОЛОГИИ

java.util.concurrent.locks

```
public interface Lock {
    void lock();
    void lockInterruptibly()
                    throws InterruptedException;

    boolean tryLock();
    boolean tryLock(long time, TimeUnit unit)
                    throws InterruptedException;

    void unlock();
    Condition newCondition();
}
```

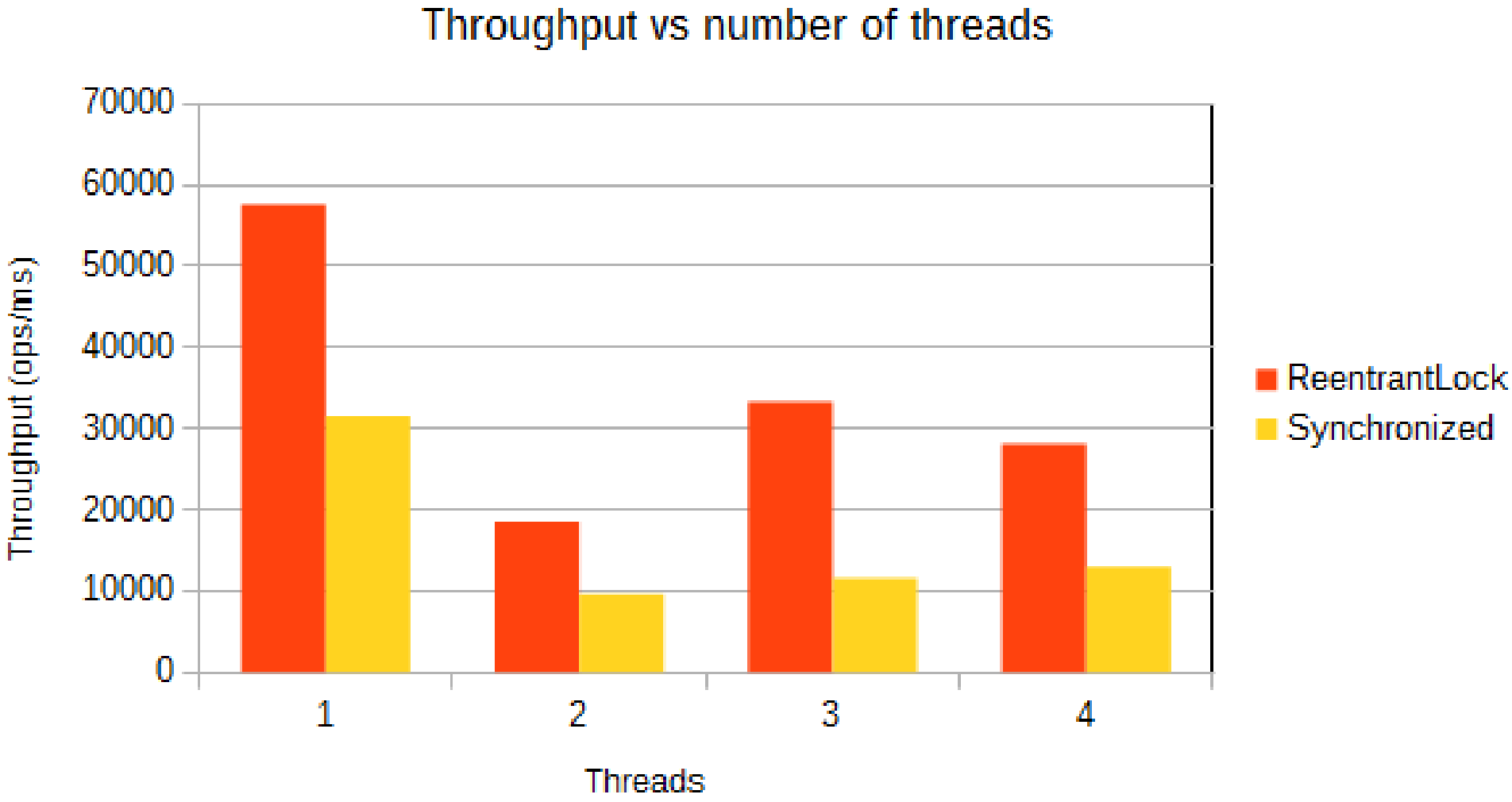
ReentrantLock

```
public interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}
```

ReentrantReadWriteLock

СБЕРБАНК ТЕХНОЛОГИИ

java.util.concurrent.locks



java.util.concurrent.atomic

- AtomicBoolean
- AtomicInteger
- AtomicLong
- AtomicIntegerArray
- AtomicLongArray
- AtomicReference<V>
- AtomicReferenceArray<E>

…and many more!

СБЕРБАНК ТЕХНОЛОГИИ

java.util.concurrent.atomic

method examples

```
AtomicLong {
    boolean compareAndSet(long expect, long update);
    long increment/decrementAndGet();
    long getAndIncrement/Decrement();
}
AtomicLongArray {
    boolean compareAndSet(int index, long expect, long update);
    long increment/decrementAndGet(int index);
    long getAndIncrement/Decrement(int index);
}
AtomicReference<T> {
    T updateAndGet(UnaryOperator<T> updateFunction);
}
```

СБЕРБАНК ТЕХНОЛОГИИ

Concurrent Collections

- `CopyOnWriteArrayList<E>`
- `CopyOnWriteArraySet<E>`

no read overhead,
all mutative operations make a fresh copy of the array

significantly faster in 'high read/low write' usecases

safe iterator, no `ConcurrentModificationException`**,**
but no modification supported

Concurrent Collections

- `CopyOnWriteArrayList<E>`
- `CopyOnWriteArraySet<E>`

all mutative operations make a fresh copy of the array

significantly faster in 'high read/low write' usecases

weakly consistent **iterator/spliterator:**
iterates over a snapshot,
no `ConcurrentModificationException`**,**
but no modification allowed

Concurrent Collections

- `ConcurrentSkipListMap<K,V>`
- `ConcurrentSkipListSet<E>`

average O(log n) time cost for the contains, add, and remove
entries are in sorted order
***weakly consistent* iterator/spliterator**
does *not* permit the use of `null` keys or values

СБЕРБАНК ТЕХНОЛОГИИ

Concurrent Collections

- `ConcurrentSkipListMap<K,V>`
- `ConcurrentSkipListSet<E>`

average O(log n) time cost for the contains, add, and remove
entries are in sorted order
***weakly consistent* iterator/spliterator**
does *not* permit the use of `null` keys or values

`ConcurrentHashMap<K,V>`

expected concurrency tuning
no null allowed
***mostly* non-blocking reads**
no time cost guarantees

`ConcurrentMap<K,V>`
`ConcurrentNavigableMap<K,V>`

Queues

`ConcurrentLinkedQueue<E>`

- **non-blocking, no nulls, weakly consistent iteration**
- **don't use .size()!**

`ConcurrentLinkedDeque<E>`

- **supports LIFO**
- **40% slower compared to the above**

СБЕРБАНК ТЕХНОЛОГИИ

Queues

`ConcurrentLinkedQueue<E>`

- **non-blocking, no nulls, weakly consistent iteration**
- **don't use .size()!**

`ConcurrentLinkedDeque<E>`

- **supports LIFO**
- **40% slower compared to the above**

`BlockingQueue<E>`
`BlockingDeque<E>`
`ArrayBlockingQueue<E>`
`LinkedBlockingQueue<E>`
`LinkedBlockingDeque<E>`
`SynchronousQueue<E>`

СБЕРБАНК ТЕХНОЛОГИИ

Synchronizers

- Semaphore
- CountDownLatch
- CyclicBarrier
- Exchanger<V>

…and *Phaser*

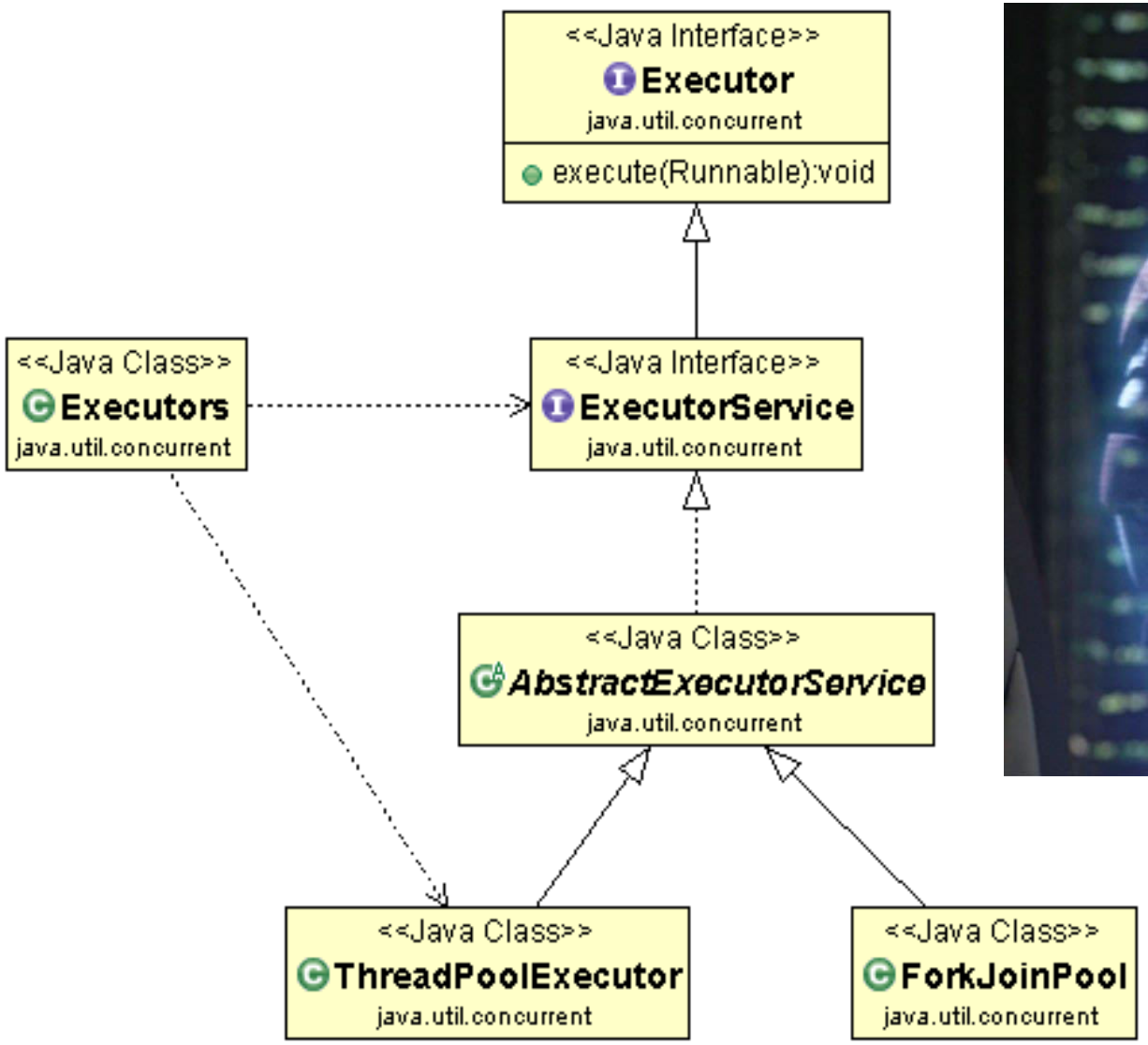
Synchronizers

- Semaphore
- CountDownLatch
- CyclicBarrier
- Exchanger<V>
- Phaser

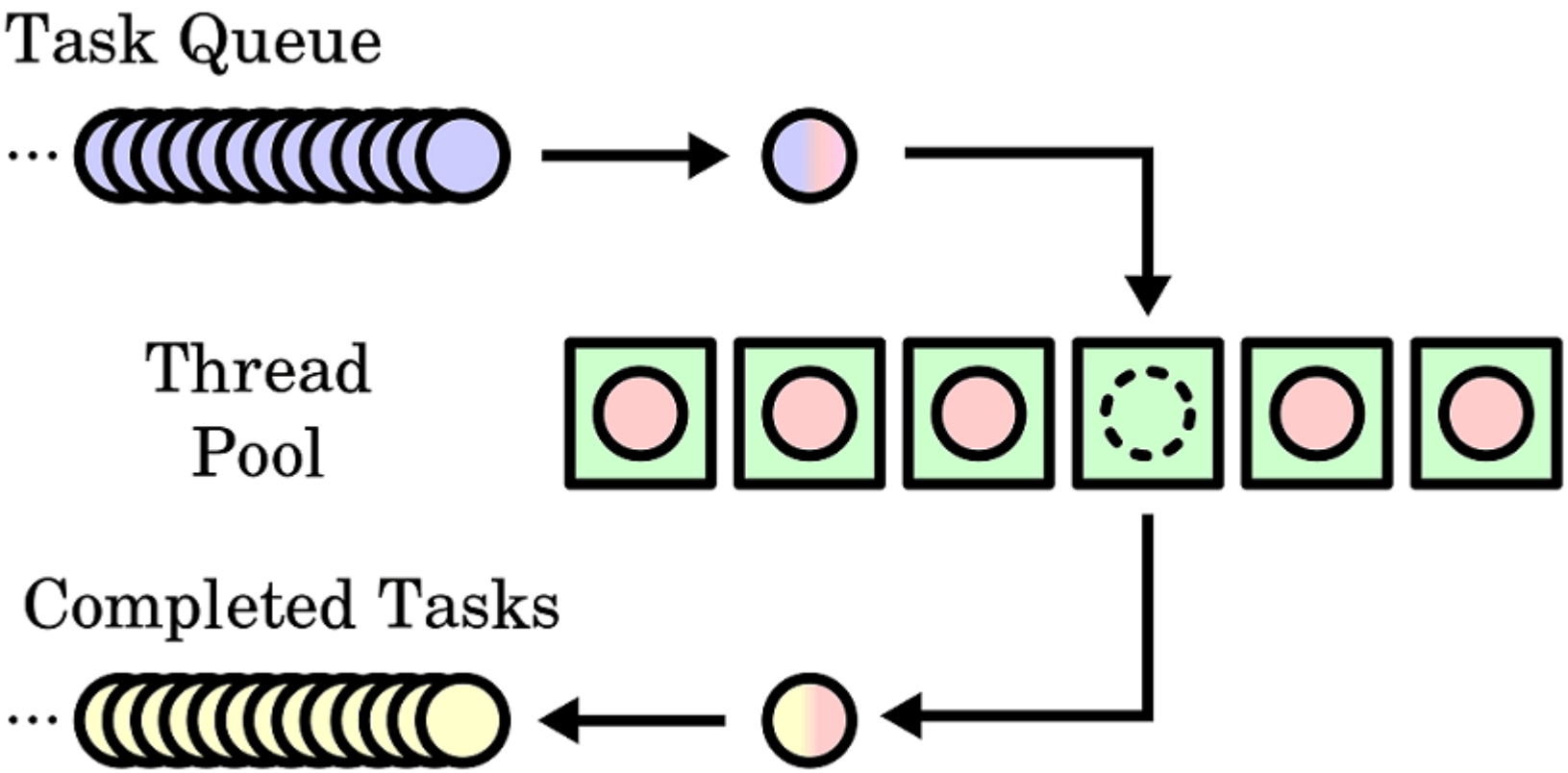
Item 81: Prefer concurrency utilities to *wait* and *notify*

"Given the difficulty of using wait and notify correctly, you should use the higher-level concurrency utilities instead."

Executor framework



Executor framework



СБЕРБАНК ТЕХНОЛОГИИ

Executor framework

Item 80: Prefer executors, tasks, and streams to threads

"…you should generally refrain from working directly with threads. When you work directly with threads, a Thread serves as both a unit of work and the mechanism for executing it.
In the executor framework, the unit of work and the execution mechanism are separate."

"…the Executor Framework does for execution what the Collections Framework did for aggregation."