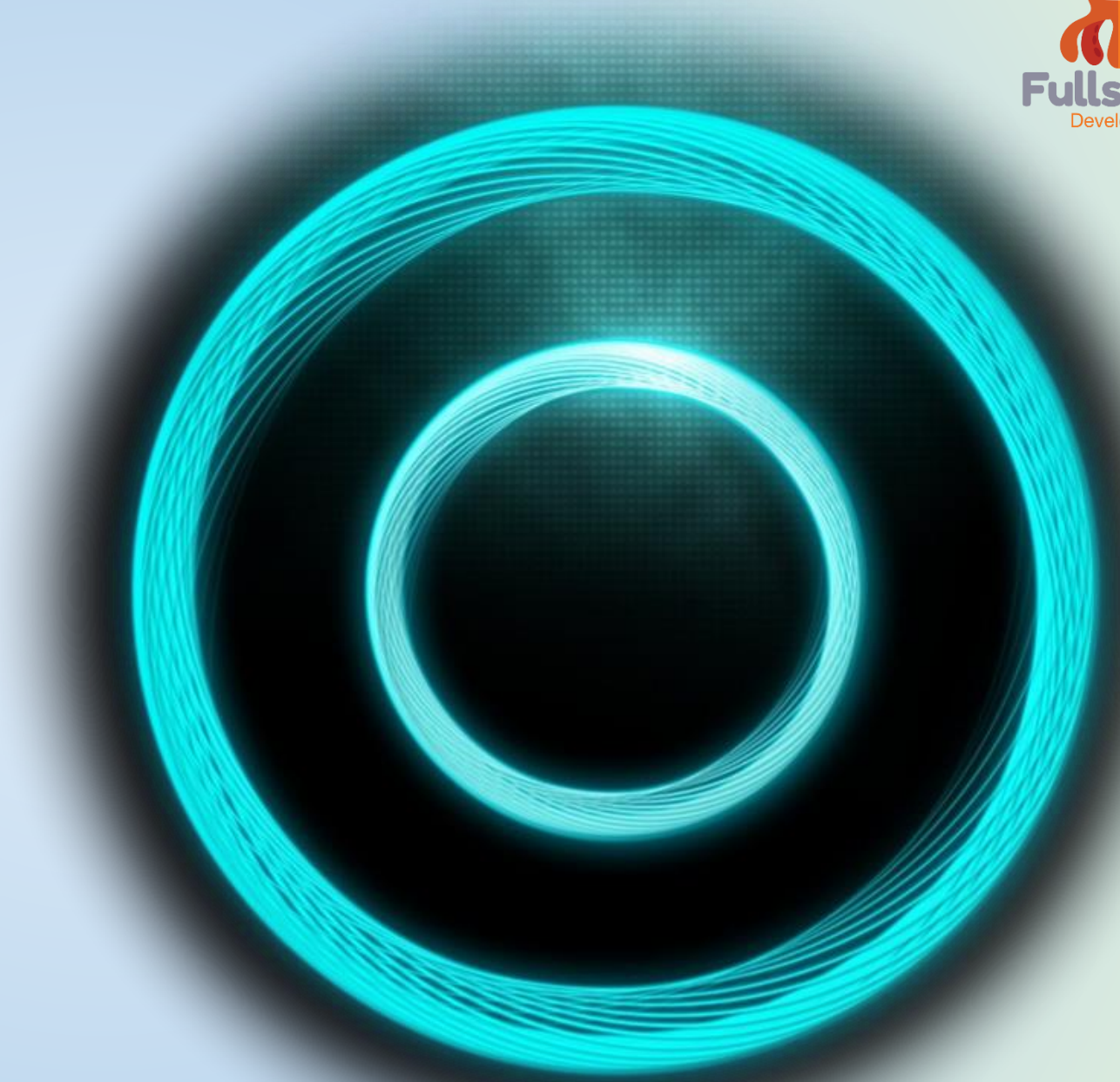
Java 8 Concurrency & Multithreading



Yoav Nordmann
Senior Software Engineer
Tikal Knowledge

yoav.nordmann@tikalk.com

Fullstack Developers

TIKAL

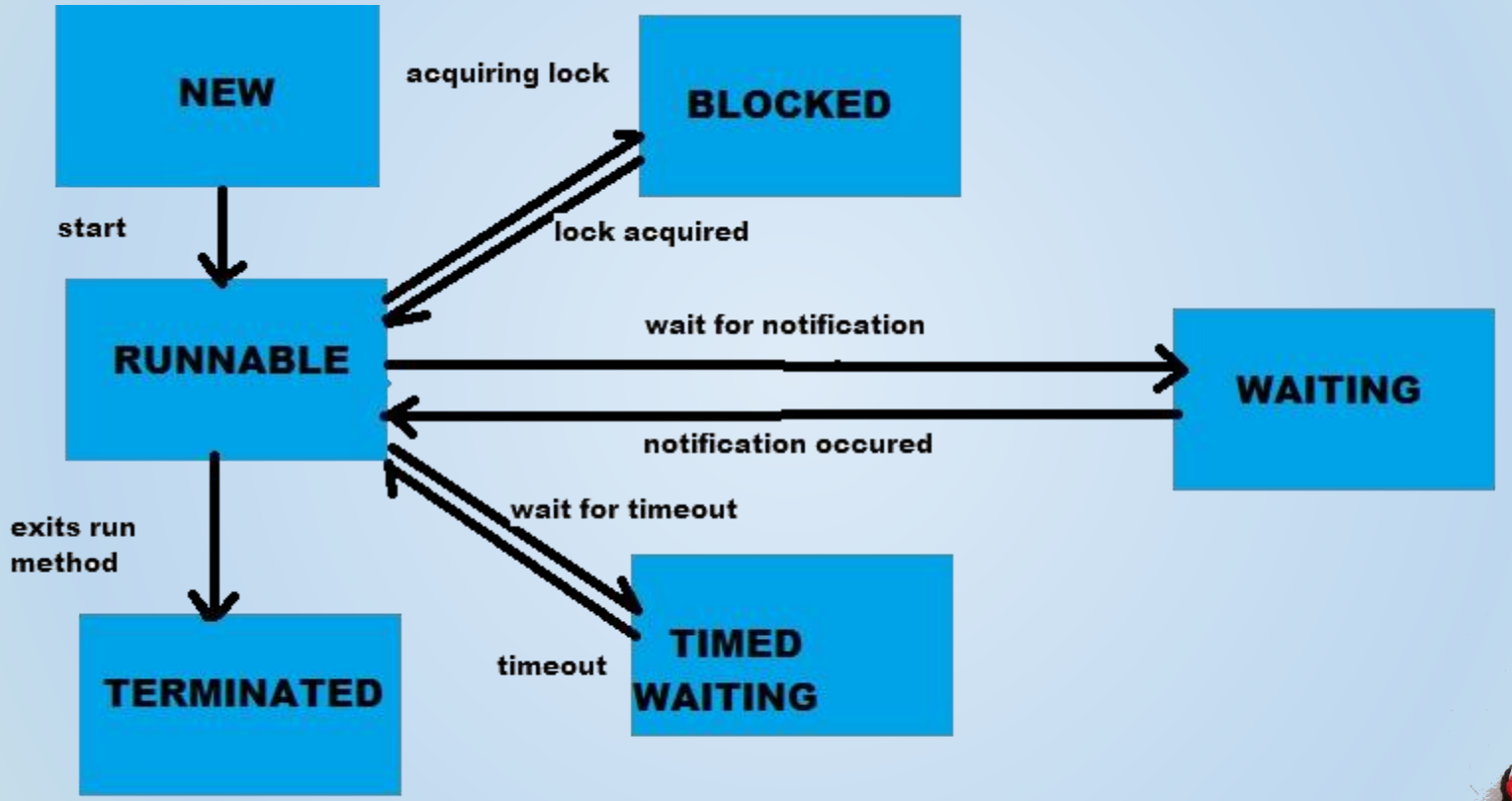
“Learning the art of programming, like most other disciplines, consists of first learning the rules and then learning when to break them.”

Joshua Bloch, Effective Java Programming Language Guide

What is a Thread

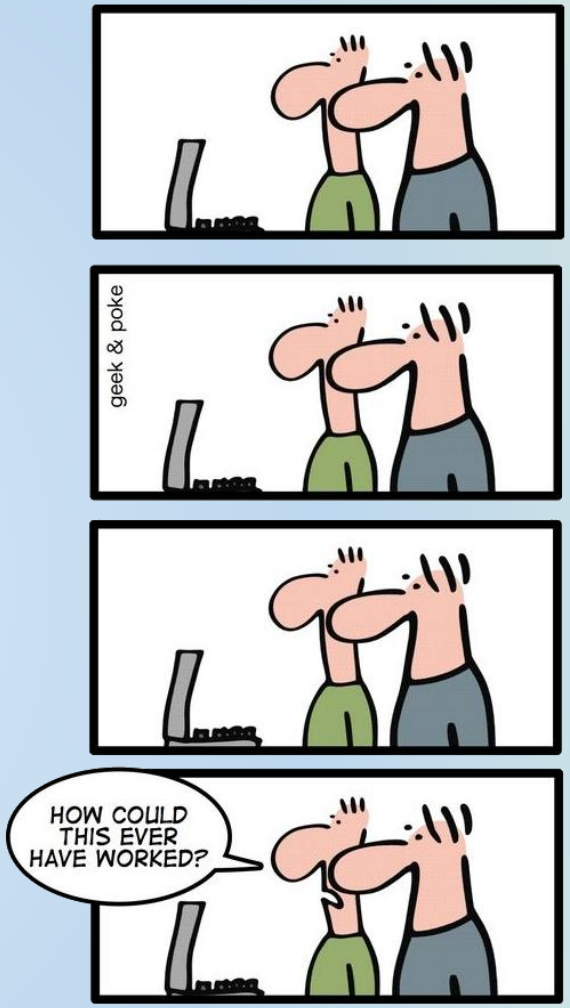
- Thread: single sequential flow of control within a program
- Multitasking allows single processor to run several concurrent threads.
- Different processes do not share memory space.
- A thread can execute concurrently with other threads within a single process.
- All threads managed by the JVM share memory space and can communicate with each other.

Thread States



Concurrency Overview

- Collections
- Synchronizers
- Executors
- Pools
- Tasks
- Results
- Atomics



Synchronized Collections

- BlockingDeque<E>
- BlockingQueue<E>
- ConcurrentLinkedDeque<E>
- ConcurrentLinkedQueue<E>
- ConcurrentMap<K,V>
- ConcurrentHashMap<K,V>
- ConcurrentNavigableMap<K,V>
- ConcurrentSkipListMap<K,V>
- ConcurrentSkipListSet<E>
- CopyOnWriteArrayList<E>
- CopyOnWriteArraySet<E>
- DelayQueue<E extends Delayed>
- LinkedBlockingDeque<E>
- LinkedBlockingQueue<E>
- LinkedTransferQueue<E>
- PriorityBlockingQueue<E>
- SynchronousQueue<E>
- ArrayBlockingQueue<E>
- TransferQueue<E>

Synchronizers

- CountDownLatch
- CyclicBarrier
- Exchanger<V>
- Phaser
- Semaphore
- ReentrantLock
- ReentrantReadWriteLock
- StampedLock
- ThreadLocalRandom

Executors and Pools

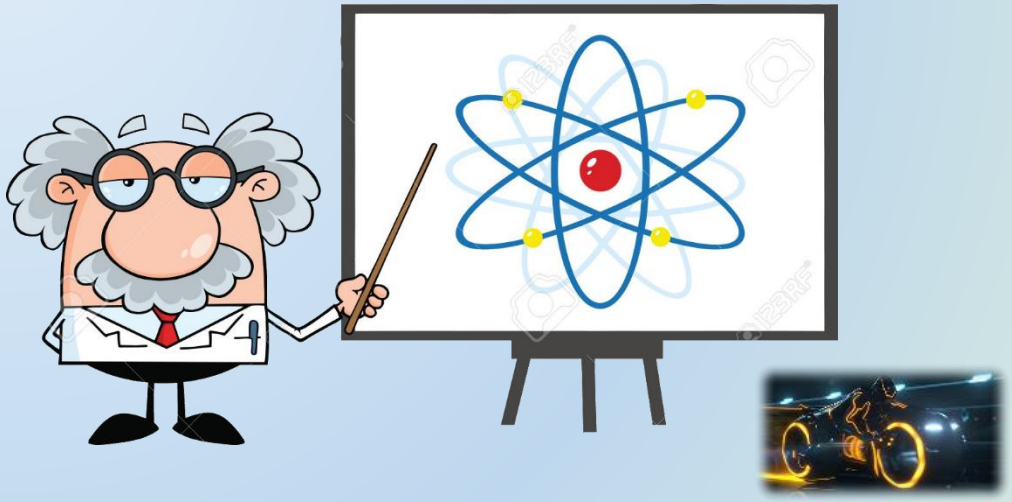
- Executors
- Executor
- ForkJoinPool
- ExecutorService
- ScheduledExecutorService
- AbstractExecutorService
- ExecutorCompletionService<V>
- ScheduledThreadPoolExecutor
- ThreadPoolExecutor

Tasks

- Callable<V>
- ForkJoinTask<V>
- ForkJoinWorkerThread
- FutureTask<V>
- RecursiveAction
- RecursiveTask<V>

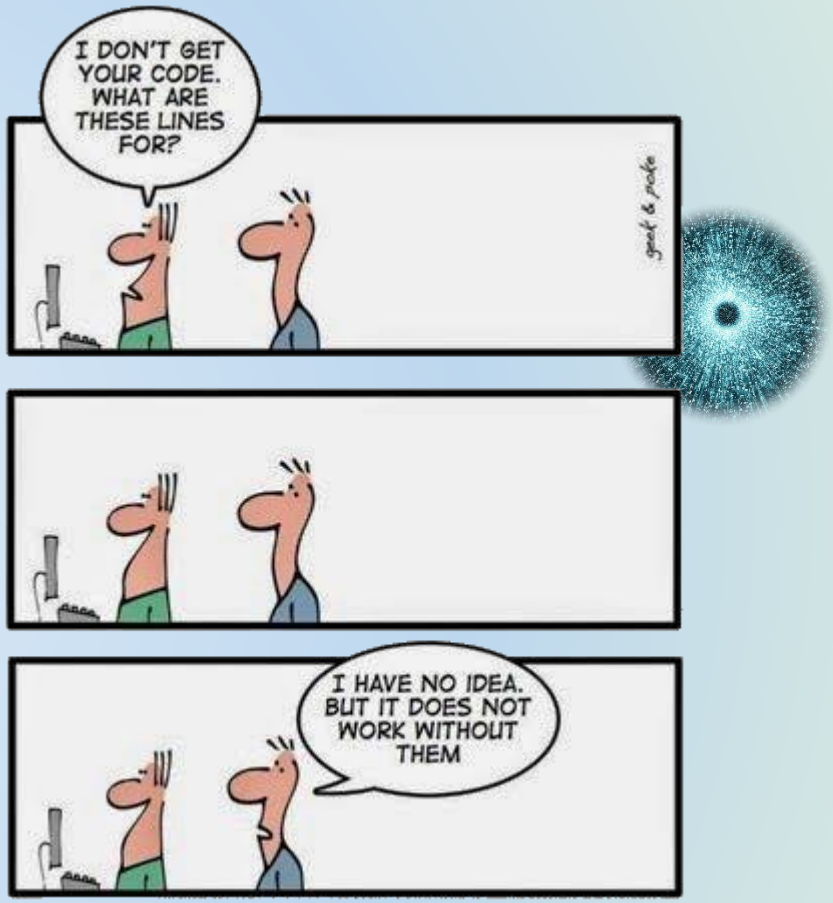
Atomics

- AtomicBoolean
- AtomicInteger
- AtomicIntegerArray
- AtomicIntegerFieldUpdater<T>
- AtomicLong
- AtomicLongArray
- AtomicLongFieldUpdater<T>
- AtomicMarkableReference<V>
- AtomicReference<V>
- AtomicReferenceArray<E>
- AtomicReferenceFieldUpdater<T,V>
- AtomicStampedReference<V>



Task Results

- Future<V>
- RunnableFuture<V>
- RunnableScheduledFuture<V>
- ScheduledFuture<V>
- CompletableFuture<T>
  - CompletionService<V>
  - CompletionStage<T>



What's New in Java 8

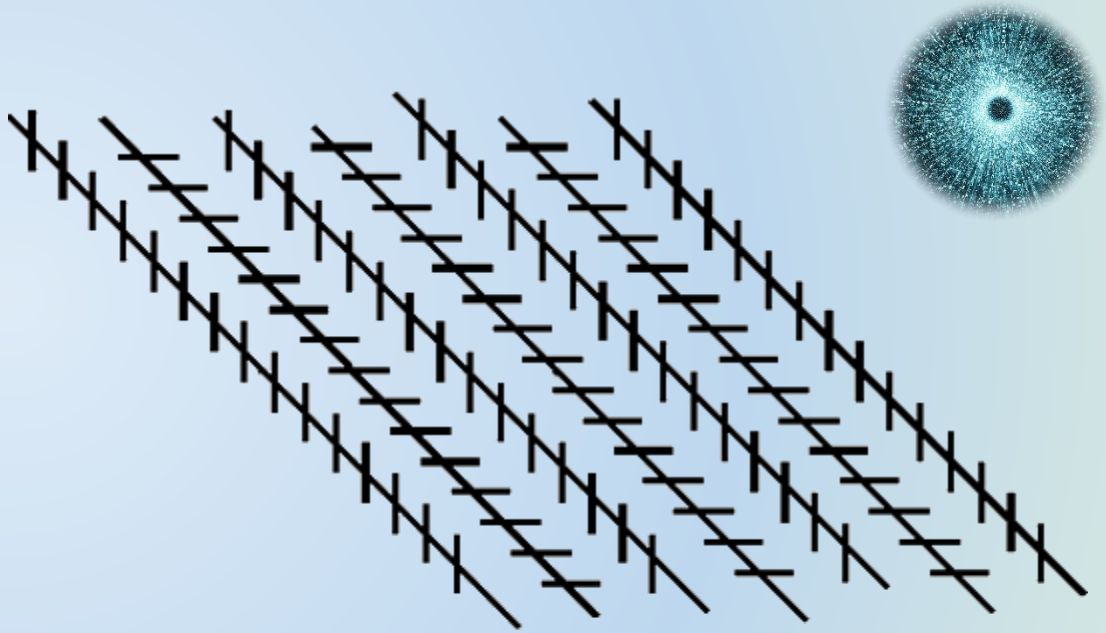
- Common ForkJoin Pool
- Parallel Streams
- CompletableFuture
- StampedLock
- Adders & Accumulators
- ConcurrentHashMap changes
- @Contended

Common ForkJoin Pool

- ForkJoinPool.commonPool()
- Single instance for common use
- Lazy initialized
- Default size: java.lang.Runtime.availableProcessors() - 1

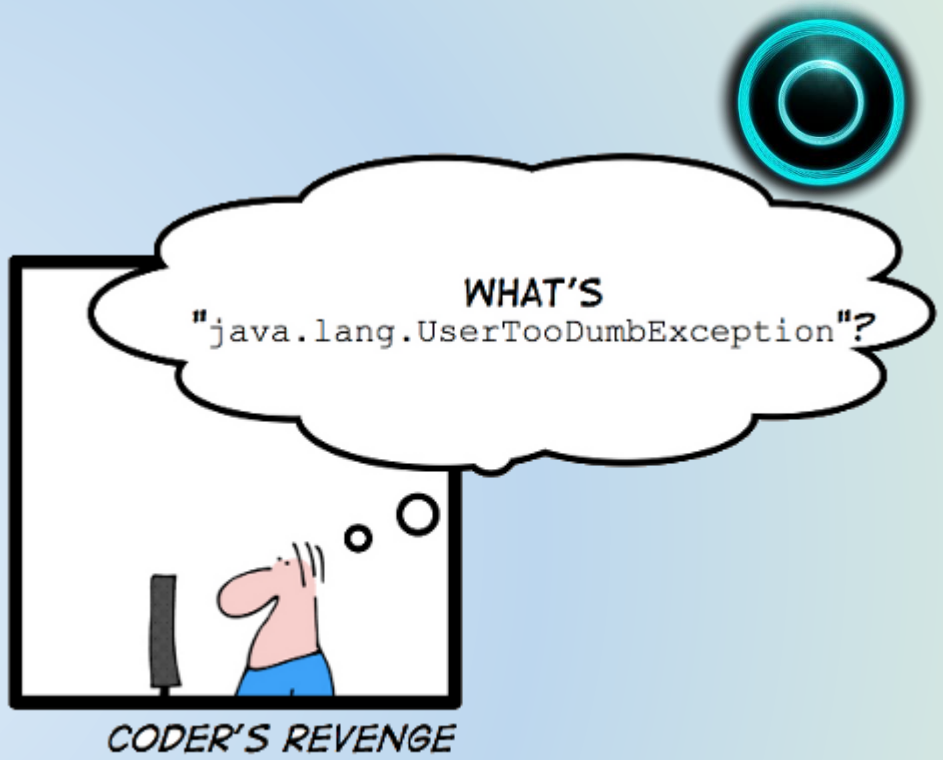
Parallel Streams

- `List<T>.parallelStream()`
- `Stream<T>.parallel()`
- `Stream<T>.sequential()`
- `Stream<T>.isParallel()`

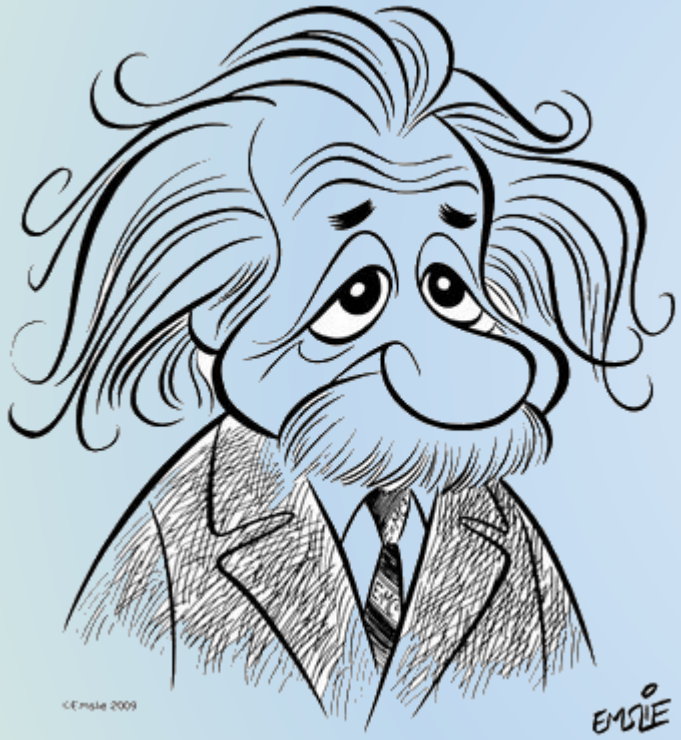


Parallel Stream Pitfalls

- Java 8 streams cannot be reused.
  - As soon as you call any terminal operation the stream is closed.
- Parallel Stream might eat your resources



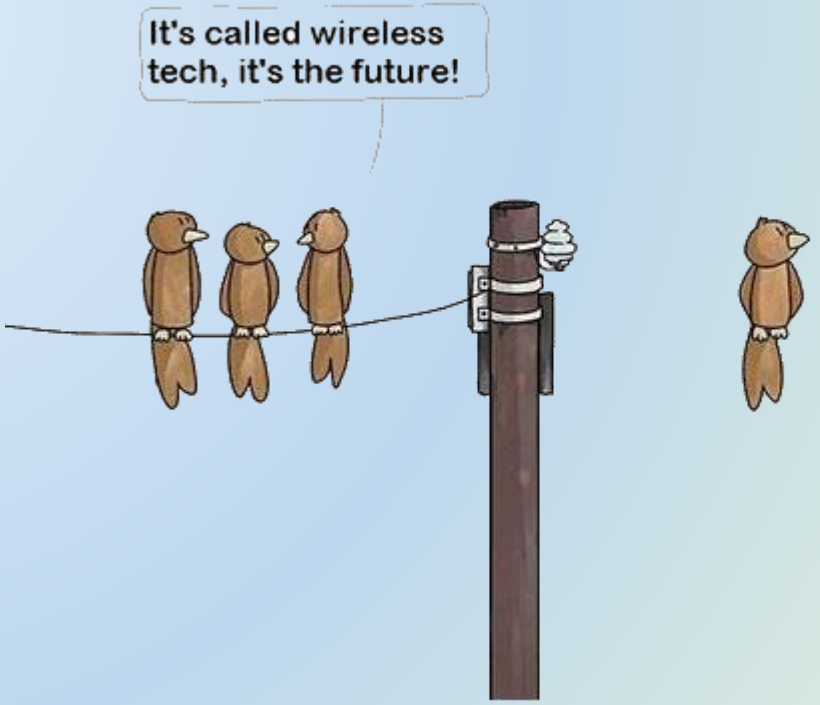
Parallel Streams



Code Examples

CompletableFuture

- Extension to the classic ”Future”
- Improved Functionality
- Decoupling Task Knowledge from Task Invoker
- “Reactive Programming”
- Non-blocking



Classic Future

```
ExecutorService executor = Executors.newFixedThreadPool(1);
Future<Integer> future = executor.submit(() -> getStockInfo("ILS"));

System.out.println("future done? " + future.isDone());
Integer result = null;
try {
        result = future.get();
} catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
{
System.out.println("Value: " + result);
```

CompletableFuture

```
CompletableFuture<Integer> completableFuture =
                CompletableFuture.supplyAsync(() -> getStockInfo("ILS"));

completableFuture.thenAccept(
                result -> System.out.println("Value: " + result));
```

```
cFuture.cancel(true); //???
```

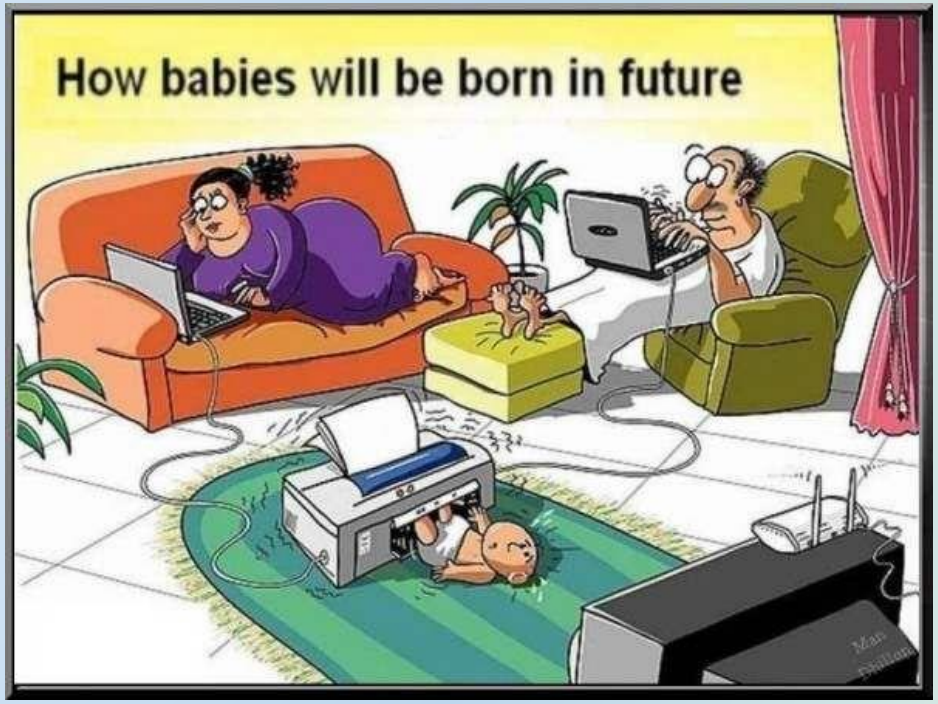
CompletableFuture

```
public class CompletableFuture<T> implements
Future<T>, CompletionStage<T>
```

```
//Supplier Interface
CompletableFuture.supplyAsync(() -> getStockInfo("ILS"));

//Runnable interface
CompletableFuture.runAsync(() -> getStockInfo("ILS"));
```

CompletableFuture - Reactions

```
//Runnable Interface
cFuture.thenRun(...);

//Function Interface
cFuture.thenApply(...);

//Consumer Interface
cFuture.thenAccept(...);
```

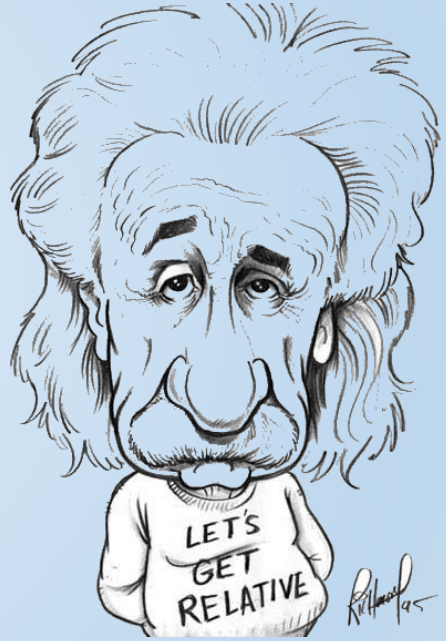


CompletableFuture - Reactions

```
CompletableFuture.supplyAsync(() -> getStockInfo("ILS"))
.whenComplete((result, exc) ->
                    System.out.println("Value: " + result))
.thenApply(() -> getMinRate("ILS"))
.thenAccept(...)
.thenRun(...);
```

CompletableFuture

Code Examples



CompletableFuture - Combine

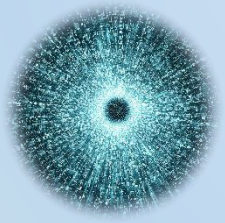
```
CompletableFuture<Integer> ils =
      CompletableFuture.supplyAsync(() -> getStockInfo("ILS"));
CompletableFuture<Integer> usd =
      CompletableFuture.supplyAsync(() -> getStockInfo("USD"));

usd.thenCombine(ils, (a,b) ->
            Double.valueOf(a) / Double.valueOf(b))
.thenAccept(div -> System.out.println("Ration: " + div));
```

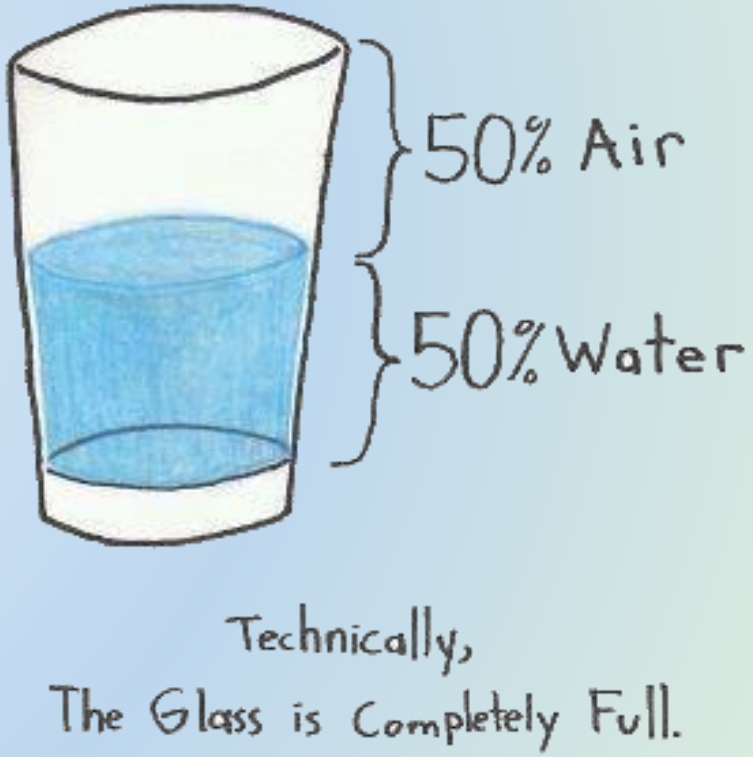
CompletableFuture - Supply

```
public static CompletionStage<Integer> getStockInfo(String str){
        CompletableFuture<Integer> future = new CompletableFuture<Integer>();
        Runnable task = new Runnable() {
                public void run() {
                        try{
                                Integer result = doSomethingReallyNasty();
                                future.complete(result);
                        }catch(Exception exception){
                                future.completeExceptionally(exception);
                        {
                }
        };
        ForkJoinPool.commonPool().submit(task);
        return future;
}
```

StampedLock

- A better version of the ReadWriteLock
- No “Reentrant” capabilities.
- Optimistic vs Pessimistic Locking





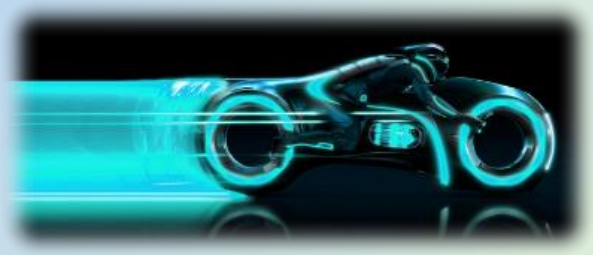
StampedLock

```
long stamp = lock.writeLock();
try {
      map.put("foo", "bar");
} finally {
      lock.unlockWrite(stamp);
{
```

```
long stamp = lock.readLock();
try {
      map.get("foo");
} finally {
      lock.unlockRead(stamp);
{
```

StampedLock – Optimistic Read

```
long stamp = lock.tryOptimisticRead();
try {
     //do your reading here
     lock.validate(stamp));
} finally {
     lock.unlock(stamp);
{
```



StampedLock – Optimistic Read

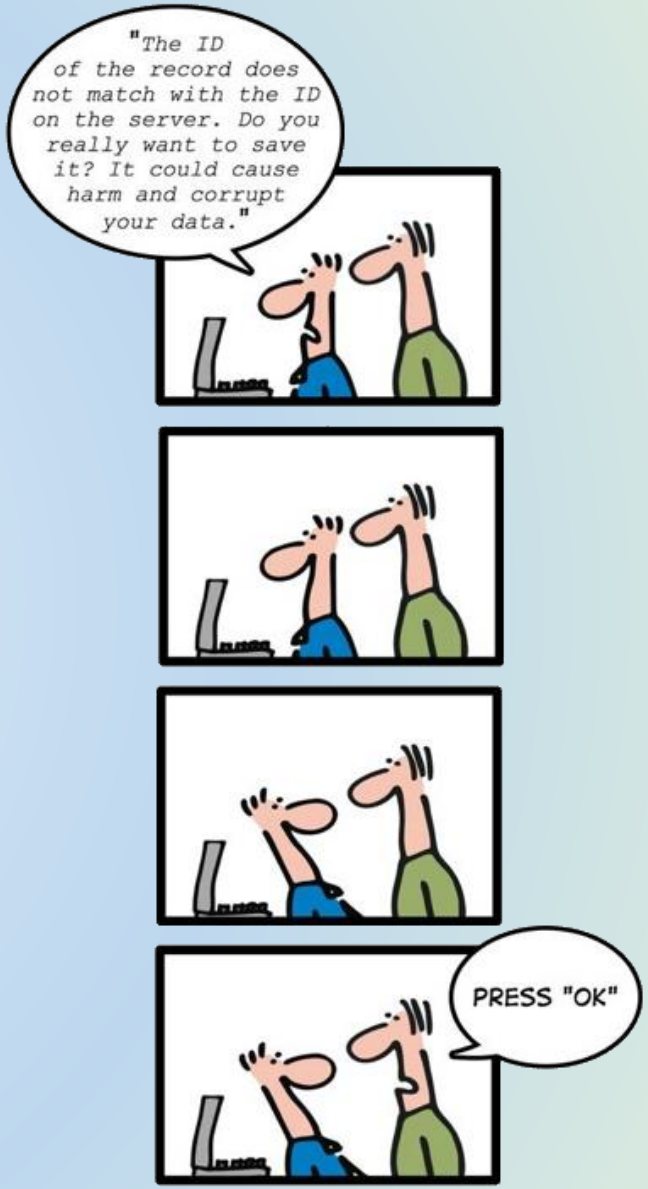
```
long stamp = lock.tryOptimisticRead(); // non blocking
read();
if (!lock.validate(stamp)) {
// write occurred, try again with a read lock
     stamp = lock.readLock();
     try {
          read();
     } finally {
          lock.unlock(stamp);
     }
}
```

StampedLock – Convert Write Lock

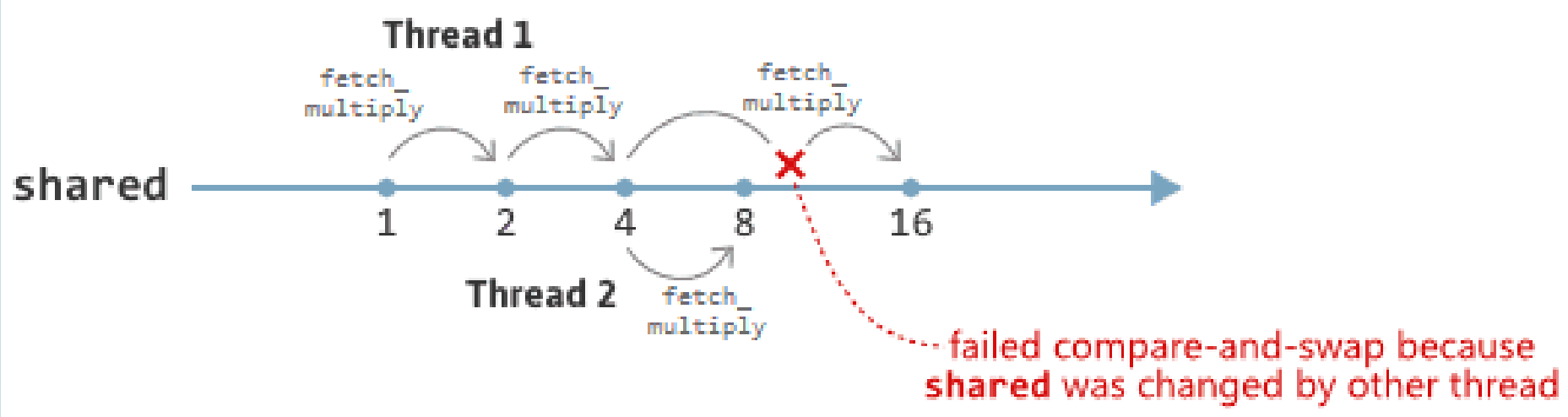
```
stamp = lock.tryConvertToWriteLock(stamp);
if (stamp == 0L) {
      //no success
      stamp = lock.writeLock();
{
```

Adders & Accumulators

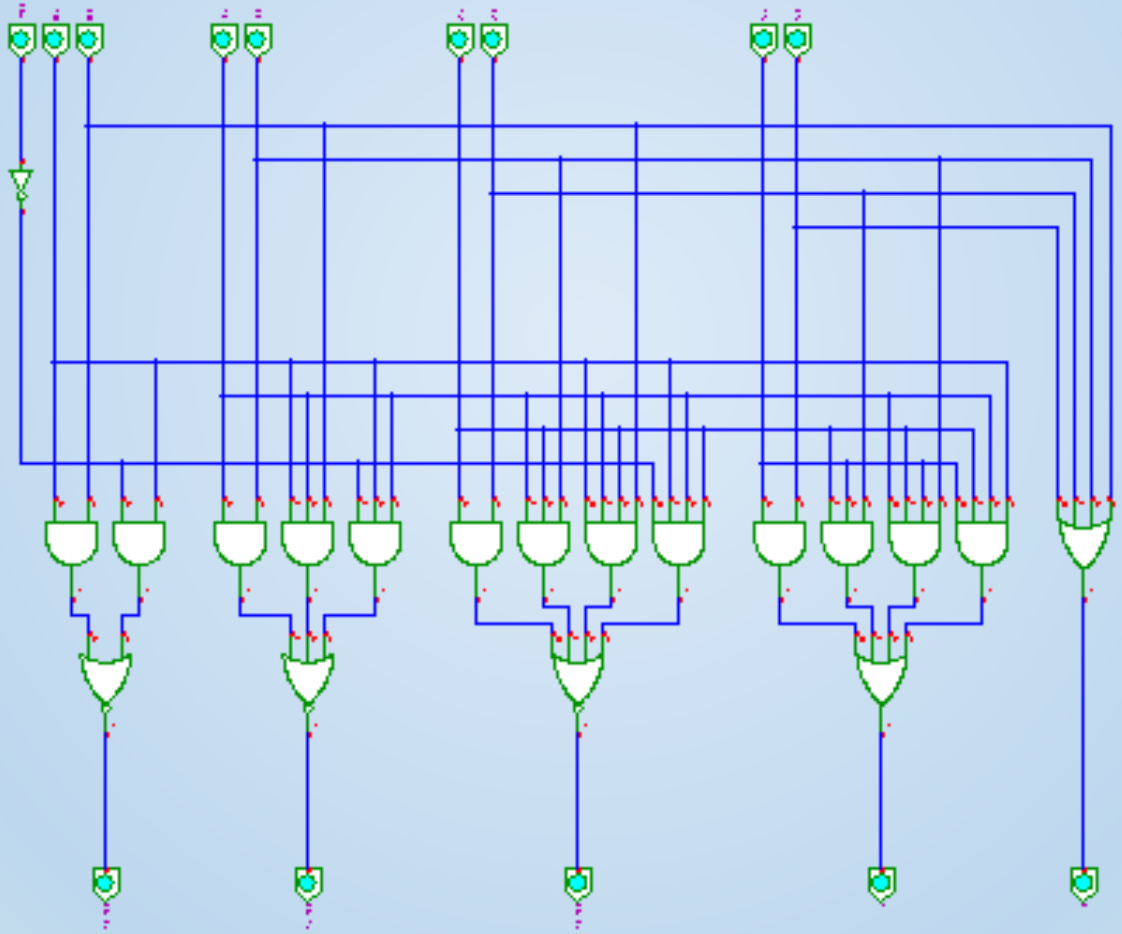
- Replace the “Atomic” classes
- Much more scalable
- Avoid “compare & swap” operation cycles
- Multiple Atomic “cells”
- Accumulator is a generalization of the Adders



Compare and Swap



Multiple Atomic “cells”



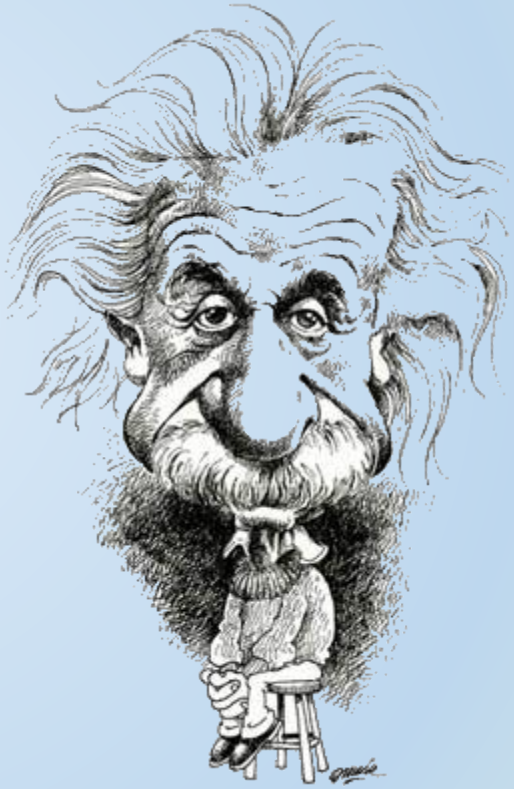
Adders & Accumulators

- `LongAdder longAdder = new LongAdder();`
- `longAdder.add(4);`
- `longAdder.increment();`
- `longAdder.sum();`
- `longAdder.reset();`

- `LongAccumulator maxAccumulator = new LongAccumulator(Long:max, Long.MIN_VALUE);`
- maxAccumulator.accumulate(100);

Adders & Accumulators

Code Example



Adders & Accumulators

- Machine Specs:
  - SPARC T4
  - Oracle Solaris 11.1 SPARC
  - 8-core (64 virtual)
  - Sparc V9 2.85GHz
  - 128 GB RAM

Results

Threads	Atomic Long ops/ms	LongAdder ops/ms
8	3	204
16	3	367
32	2	680
64	1	1087
128	1	1101

ConcurrentHashMap changes

- Buckets are now Trees and not Lists (keys must be Comparable)
- Guaranteed O(log(n)) performance
- The ConcurrentHashMap class introduces over 30 new methods in this release
- forEach, forEachKey, forEachValue, and forEachEntry
- search, searchKeys, searchValues, and searchEntries
- reduce, reduceToDouble, reduceToLong

ConcurrentHashMap changes

- `default V compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction);`

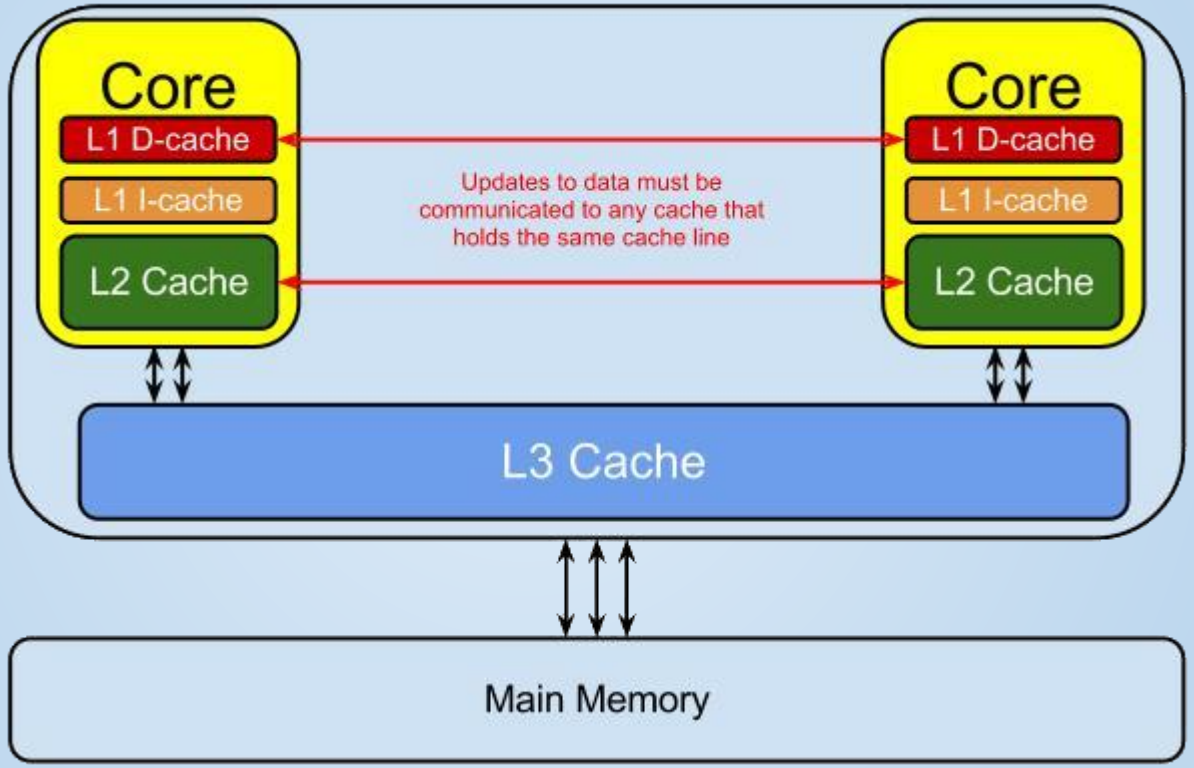
- `default V computeIfPresent(K key,BiFunction<? super K, ? super V, ? extends V> remappingFunction);`
-
- `default V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction);`

@Contended

- Know as “The Silent Performance Killer”
- Multiple Processors / Multi Core Processors
- Solve the Problem of false Sharing

- What is False Sharing ?

Memory


Core
L1 D-cache
L1 I-cache
L2 Cache
Core
L1 D-cache
L1 I-cache
L2 Cache
Updates to data must be communicated to any cache that holds the same cache line
L3 Cache
Main Memory

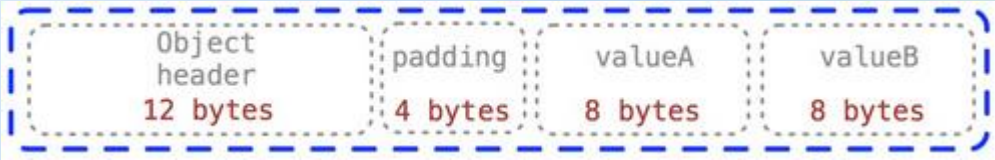
Memory Price

Latency from CPU to...	Approx. number of CPU cycles	Approx. time in nanoseconds
Main memory		~60-80ns
QPI transit (between sockets, not drawn)		~20ns
L3 cache	~40-45 cycles,	~15ns
L2 cache	~10 cycles,	~3ns
L1 cache	~3-4 cycles,	~1ns
Register	1 cycle	

Caching on Modern CPUs

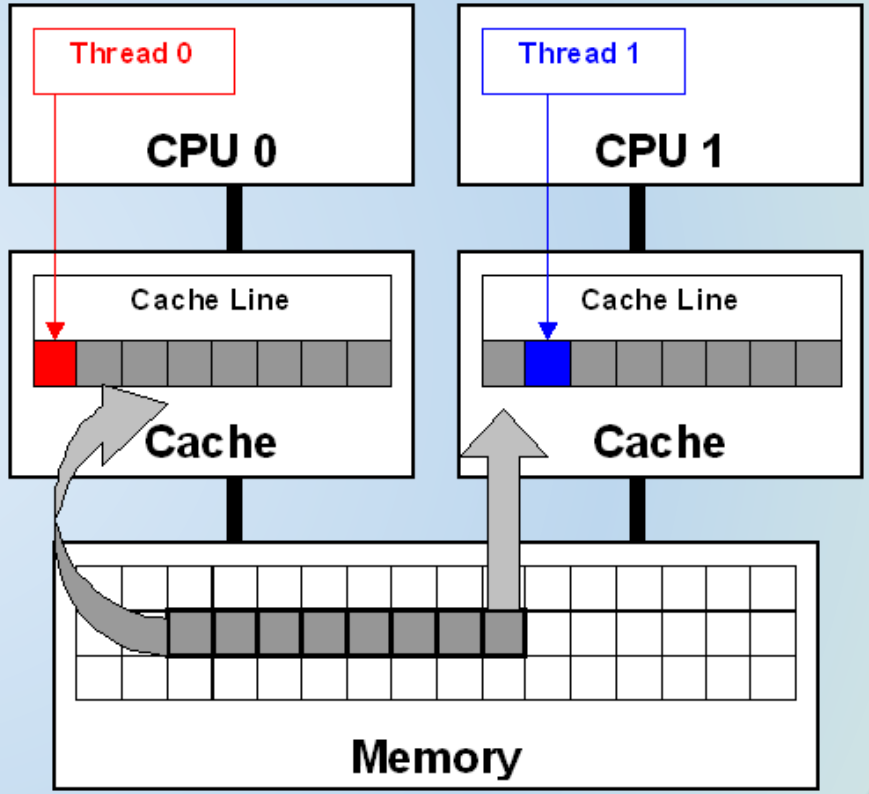
- To efficiently operate on data, it is fetched from main memory into a CPU cache at the granularity of a cache line
- A cache line is a block of memory in the CPU cache, say 64 bytes.

```
public class FalseSharing {
        public volatile long valueA;
        public volatile long valueB;
}
```



What really happens

- CPU_0 knows that it does not own the cache line "n" exclusively, so it has to broadcast an "Invalidate" message across the bus after changing the cache line to notify other cores that their caches are stale.
- CPU_1 is listening on the bus and invalidates the corresponding cache line.
- Consequently, this produces a lot of unnecessary bus traffic although both cores operate on different fields.



This phenomenon is known as false sharing.

@Padding

- A core can execute hundreds of instructions in the time taken to fetch a single cache line.
- If a core has to wait for a cache line to be reloaded, the core will run out of things to do, this is called a stall.
- Stalls can be avoided by reducing false sharing, one technique to reduce false sharing is to **pad out data structure** so that threads working on independent variables fall in separate cache lines.

@Contended

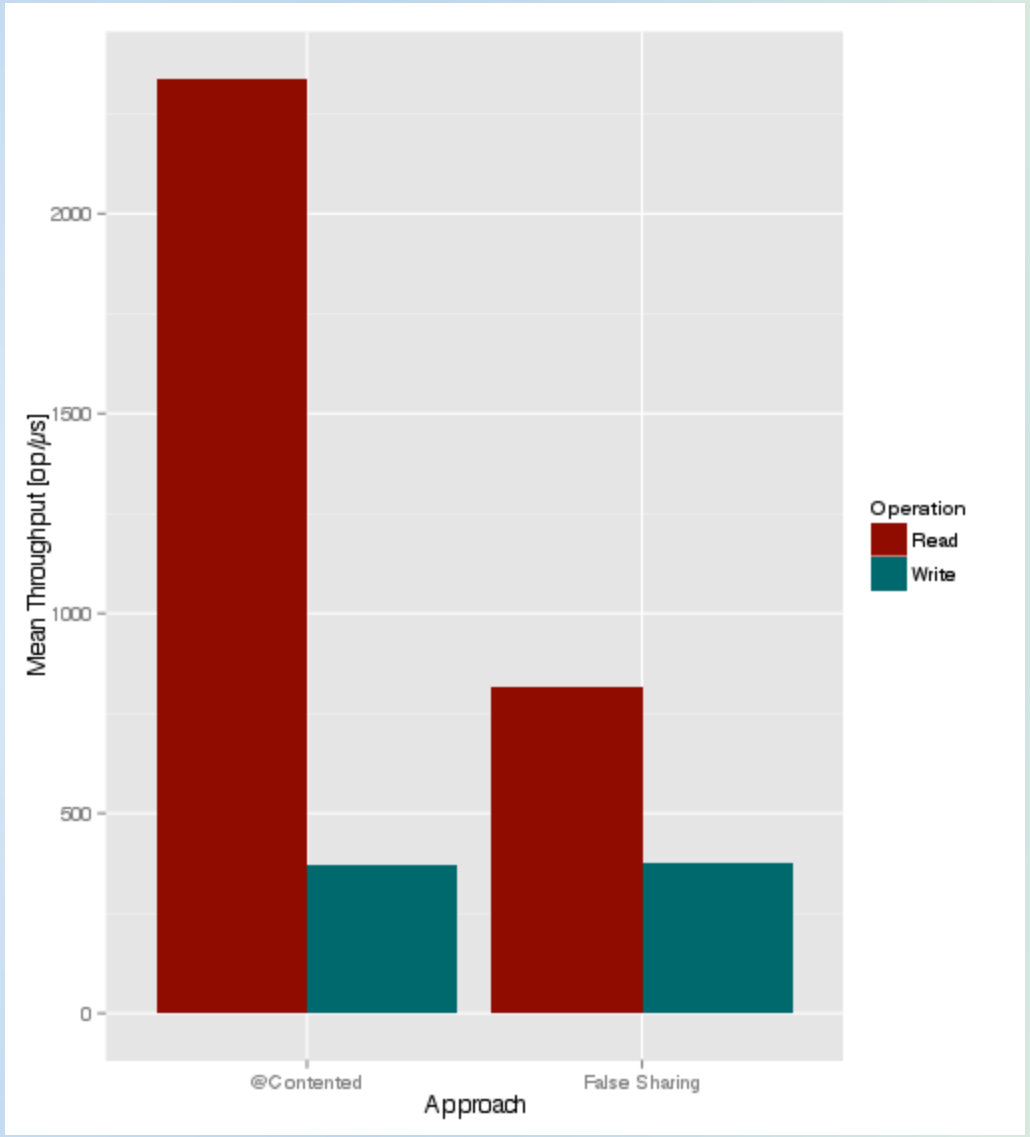
```
import sun.misc.Contended;

public class FalseSharing {
    @Contended
    public volatile int f1;
    public volatile int f2;
}
```

object header (12 bytes)	f2 (4 bytes)	padding due to @Contended (128 bytes)	f1 (4 bytes)	padding (4 bytes)

Access restriction: The type 'Contended' is not API

@Contended - Padding

- The current OpenJDK implementation will then pad the field appropriately, inserting a 128 byte padding after each annotated field.
- 128 bytes is twice the typical cache line size.



https://github.com/YoavNordmann/java-8-concurrency

yoav.nordmann@tikalk.com