java.util.concurrent.*

What you may not know

Boredom (and maybe a beer) brought to you by: Jan Skrabal

What is java.util.concurrent?

- 80+ classes
- Designed for easier concurrency handling
- Since Java 1.5
- Still updated

Overview

- Atomic types and alternatives
- Locks
- Executors, Executor Services
- Futures
- Barriers
- Other Classes

Atomic Types

Package java.util.concurrent.atomic.*

- Atomic<*Boolean*|*Integer*|*Long*|*Reference*>
- Atomic<*Integer*|*Long*|*Reference*>Array
- Atomic<*Integer*|*Long*|*Reference*>FieldUpdater
- <*Double*|*Long*>Adder
- <*Double*|*Long*>Accumulator
- *AtomicMarkableReference* and *AtomicStampedReference*

Atomic Updater

- Alternative to AtomicType
- Better from memory perspective
- Comparable performance

```
class Pojo {

    private static final AtomicLongFieldUpdater<Pojo> LONG_UPDATER = AtomicLongFieldUpdater.newUpdater(
            Pojo.class, fieldName: "longVar");
    private volatile long longVar = 0;

    void setLongVar(long l) { LONG_UPDATER.set(this, l); }

    void incrLongVar(long l) { LONG_UPDATER.addAndGet( obj: this, l); }
}
```

Atomic Updater - Performance

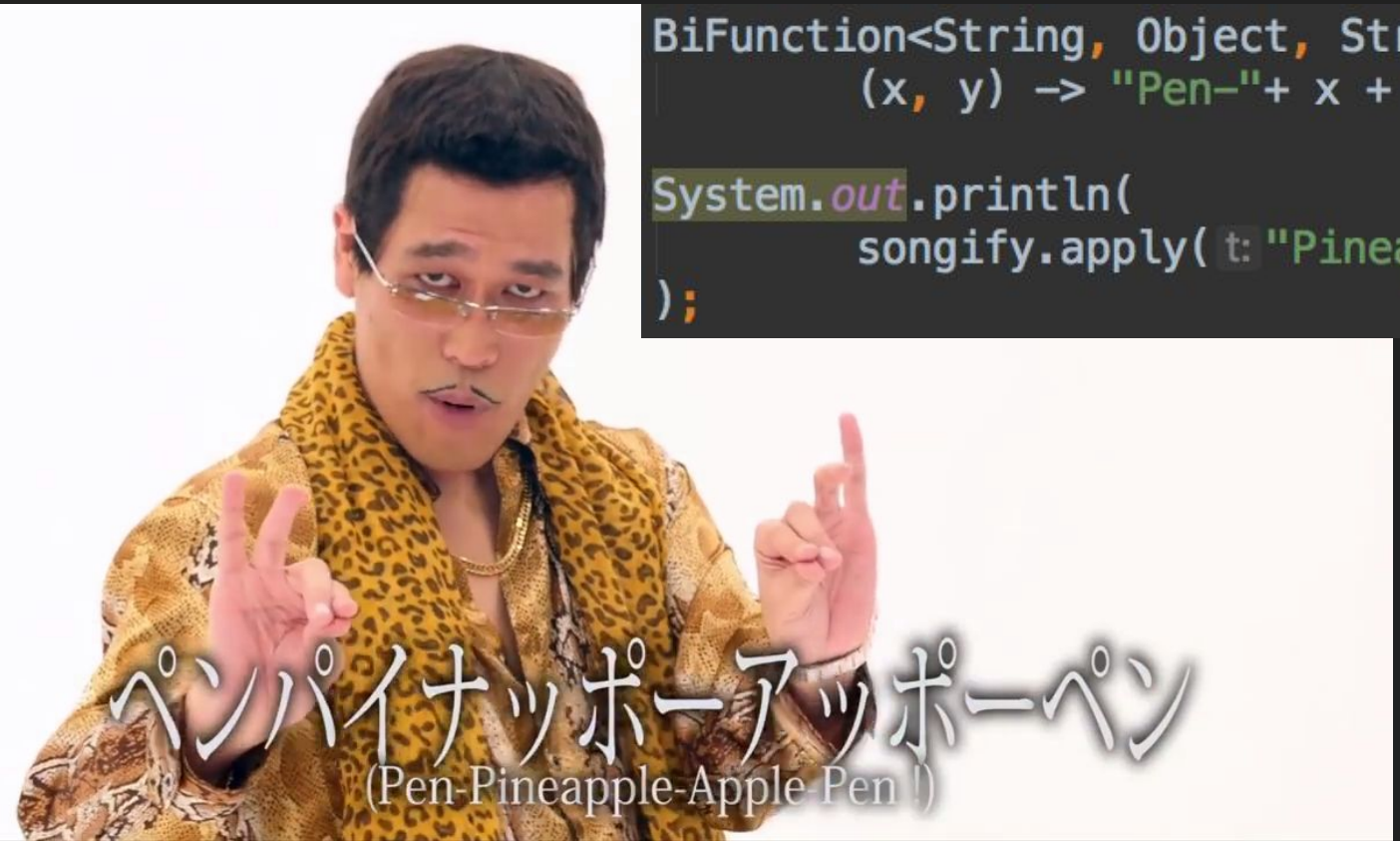
10,000,000 Iterations

Type	Set [ms]	Get [ms]	Increment [ms]	Total [ms]
AtomicLong	184	27	238	449
AtomicLongUpdater	260	40	140	440
Double as AtomicReference	**456**	56	**468**	**980**
Double as AtomicLong	291	**73**	204	568

Intemezzo - BiFunction

```
BiFunction<String, Object, String> songify =
        (x, y) -> "Pen-"+ x + "-" + y + "-Pen!";

System.out.println(
        songify.apply( t: "Pineapple", new Apple())
);
```



Accumulator and Adder

- Specialised alternative to AtomicType
- Better performance under high contention
- *new LongAdder()* is equivalent to new *LongAccumulator((x, y) -> x + y, 0L)*

```
private static final LongAccumulator ACCUMULATOR =
        new LongAccumulator((x, y) -> x * y,  identity: 1);

public static void main(String[] args) {
    ACCUMULATOR.accumulate( x: 10);
    System.out.println(ACCUMULATOR.get());
    ACCUMULATOR.accumulate( x: 10);
    System.out.println(ACCUMULATOR.get());
    ACCUMULATOR.accumulate( x: 10);
    System.out.println(ACCUMULATOR.get());
}
```

Output

```
10
100
1000
```

Other types of references

AtomicMarkableReference

- Boolean and AtomicReference
- Special case of AtomicStampedReference

AtomicStampedReference

- Long and AtomicReference
- Can solve A-B-A problem

ABA Problem

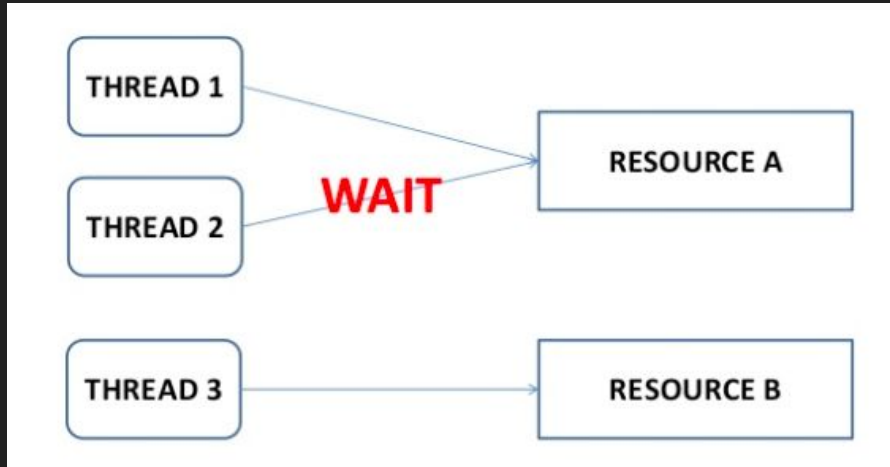
- Thread A reads value *x*
- Thread B changes value *x* to *y*
- Thread C changes value back to *x*
- Thread A uses CAS (compare and set) which succeeds

StampedAtomicReference for the rescue!

Locks - Reentrant Locks

- ReentrantLock
  - Alternative to synchronized keyword
  - Single lock
  - One thread can lock multiple times

- ReentrantReadWriteLock
  - Pair of locks (ReadLock, WriteLock)
  - ReadLock and WriteLock affects each other



ReentrantLock

```
public class ReentrantLockExample {
    private static final ReentrantLock REENTRANT_LOCK = new ReentrantLock();

    public static void main(String[] args) {
        new Thread(new Task(), name: "T2").start();
        new Thread(new Task(), name: "T1").start();
    }

    public static class Task implements Runnable {
        @Override
        public void run() { recursiveFunction( nestIndex: 0); }
        void recursiveFunction(int nestIndex) {
            if(nestIndex != 5)  {
                REENTRANT_LOCK.lock();
                try{
                    System.out.println(Thread.currentThread().getName() + " - " +
                            REENTRANT_LOCK.getHoldCount());
                    recursiveFunction( nestIndex: nestIndex + 1);
                } finally {
                    REENTRANT_LOCK.unlock();
                }
            }
        }
    }
}
```

Possible Outputs:

```
T1 - 1
T1 - 2
T1 - 3
T1 - 4
T1 - 5
T2 - 1
T2 - 2
T2 - 3
T2 - 4
T2 - 5
```

```
T2 - 1
T2 - 2
T2 - 3
T2 - 4
T2 - 5
T1 - 1
T1 - 2
T1 - 3
T1 - 4
T1 - 5
```

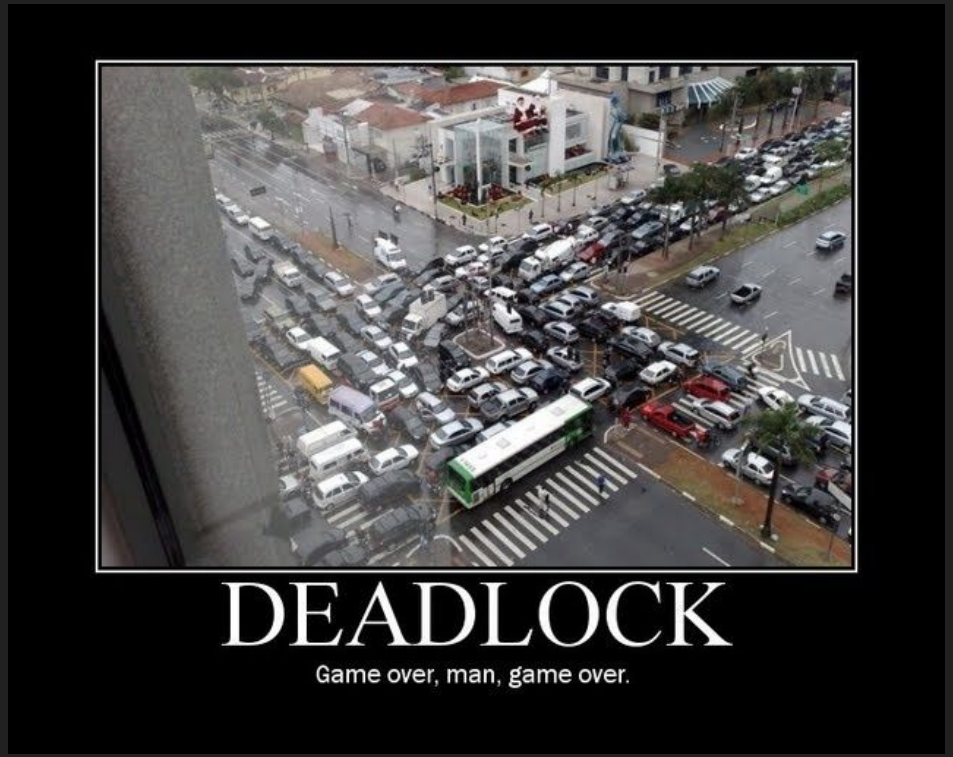
Locks - StampedLock

- Pair of locks (ReadLock, WriteLock)
- Returns stamp used for lock validation
- Has optimistic locking mode implemented
- **Doesn’t implement reentrant policy**
  - You can easily shoot yourself in the foot

Find 5 Differences!

```
StampedLock lock = new StampedLock();

lock.writeLock();
System.out.println("Locked for write");
lock.writeLock();
System.out.println("Unreachable print");
```



StampedLock Optimistic Locking

```
StampedLock lock = new StampedLock();
executor.submit(() -> {
    long stamp = lock.tryOptimisticRead();
    try {
        System.out.println("Optimistic Lock Valid: " + lock.validate(stamp));
        ExampleUtils.sleep( millis: 1);
        System.out.println("Optimistic Lock Valid: " + lock.validate(stamp));
        ExampleUtils.sleep( millis: 2);
        System.out.println("Optimistic Lock Valid: " + lock.validate(stamp));
    } finally {
        lock.unlock(stamp);
    }
});

executor.submit(() -> {
    long stamp = lock.writeLock();
    try {
        System.out.println("Write Lock acquired");
        ExampleUtils.sleep( millis: 2);
    } finally {
        lock.unlock(stamp);
        System.out.println("Write done");
    }
});
```

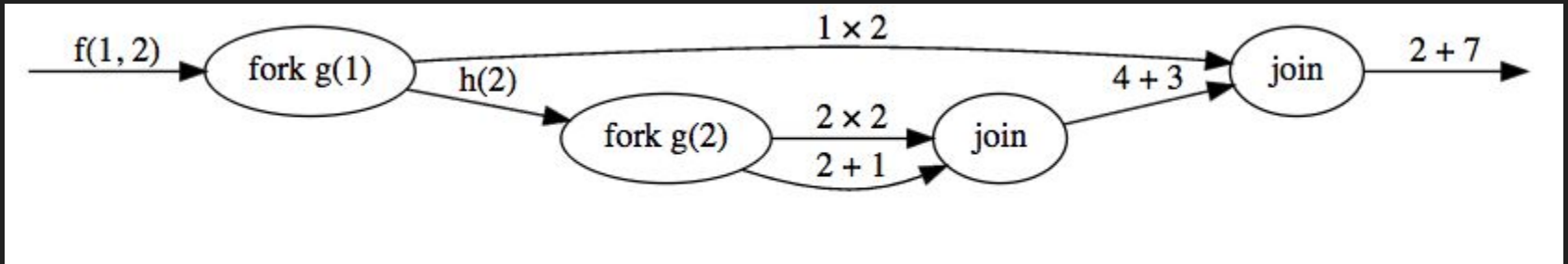
Output:

```
Optimistic Lock Valid: true
Write Lock acquired
Optimistic Lock Valid: false
Write done
Optimistic Lock Valid: false
```

Executors, Executor Services

java.util.concurrent.Executors.*

- ThreadPoolExecutor
  - Given number of threads
- CachedPoolExecutor
  - Dynamically spawns / kills threads
- ScheduledExecutor
  - Can execute task with delay
- WorkStealingExecutor



Intermezzo2 - Future & Callable

```
public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
    V get() throws InterruptedException, ExecutionException;
    V get(long timeout, TimeUnit unit) throws
            InterruptedException, ExecutionException, TimeoutException;
}
```

```
@FunctionalInterface
public interface Callable<V> {
    V call() throws Exception;
}
```

ExecutorService & (back to the) Future

```
executorService.submit
```

```
submit(Runnable task)                    Future<?>
submit(Callable<T> task)                 Future<T>
submit(Runnable task, T result)          Future<T>
```

FutureTask

- Implementation of Future
- Wrapper for Callable
- But do not have to wrap Callable!

```
Possible state transitions:
NEW -> COMPLETING -> NORMAL
NEW -> COMPLETING -> EXCEPTIONAL
NEW -> CANCELLED
NEW -> INTERRUPTING -> INTERRUPTED
```

CompletableFuture

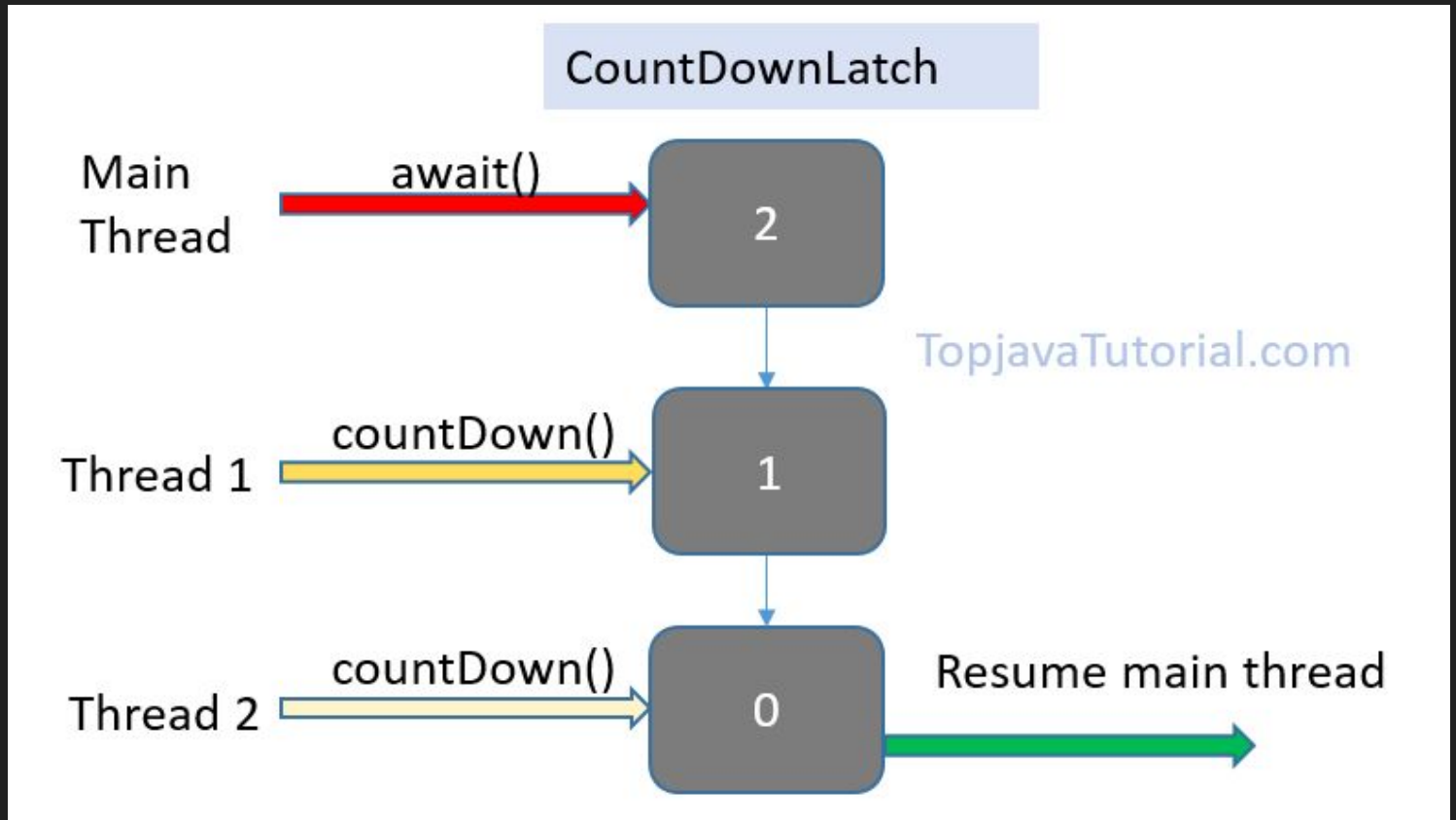
- Async streams / pipelines of tasks
- Pipepine can branch and join again later
- Alternative to barriers, locks, and much more

```
CompletableFuture.supplyAsync(() -> "results.txt")
        .thenApplyAsync(CompletableFutureExample::readFileContent)
        .thenApplyAsync(CompletableFutureExample::parseFileContent)
        .thenApplyAsync(CompletableFutureExample::sum)
        .thenAcceptAsync(result -> System.out.println("Result: " + result))
        .exceptionally(e -> {
            System.out.println("Something went wrong " + e.getMessage());
            return null; //Void
        });
```

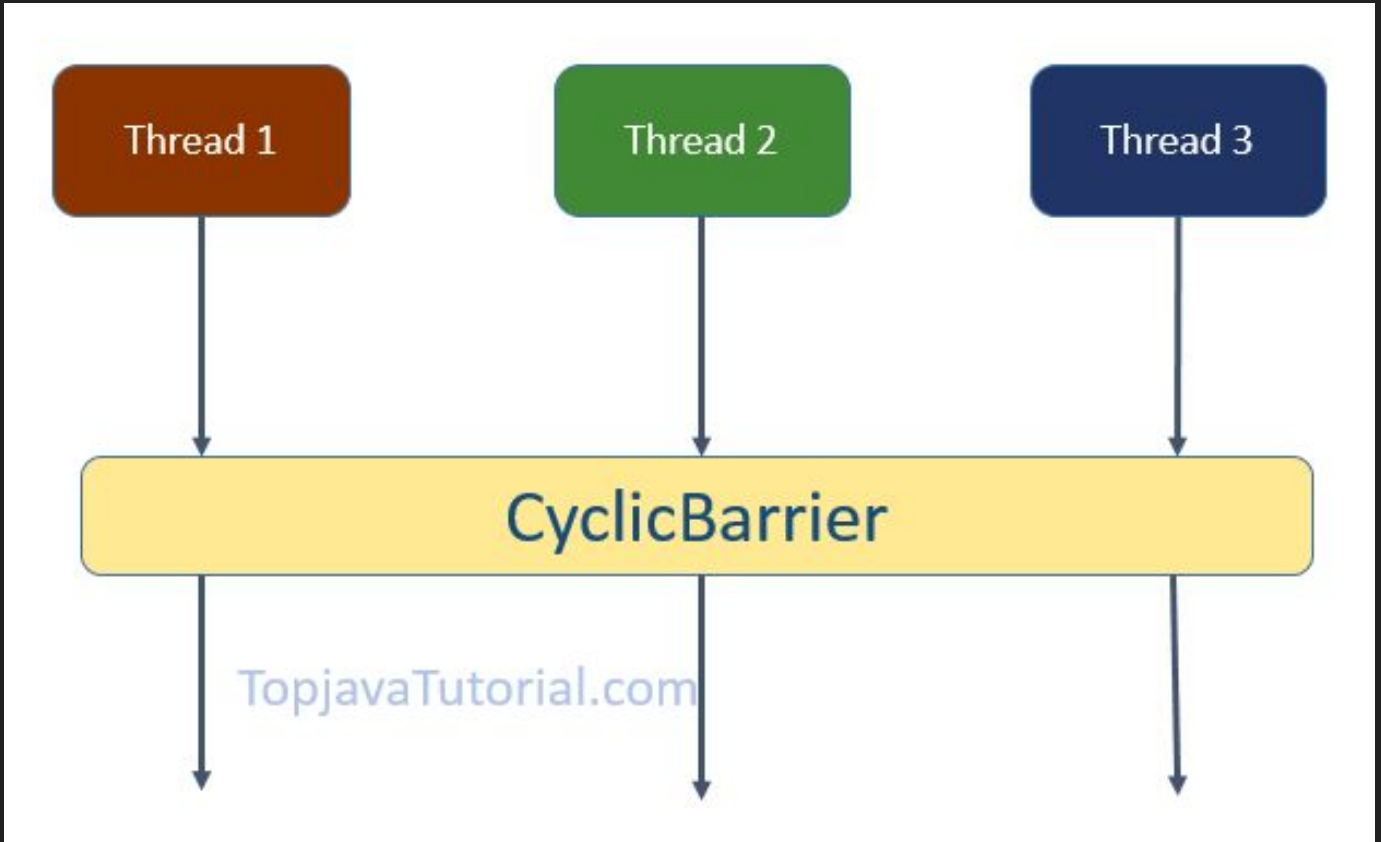
Barriers

- CountDownLatch
  - Help you wait until other threads finish their work
- CyclicBarrier
  - Wait for the group of threads to assemble, then let go
- Phaser
  - More generic CyclicBarrier

Barriers - CountDownLatch



CyclicBarrier

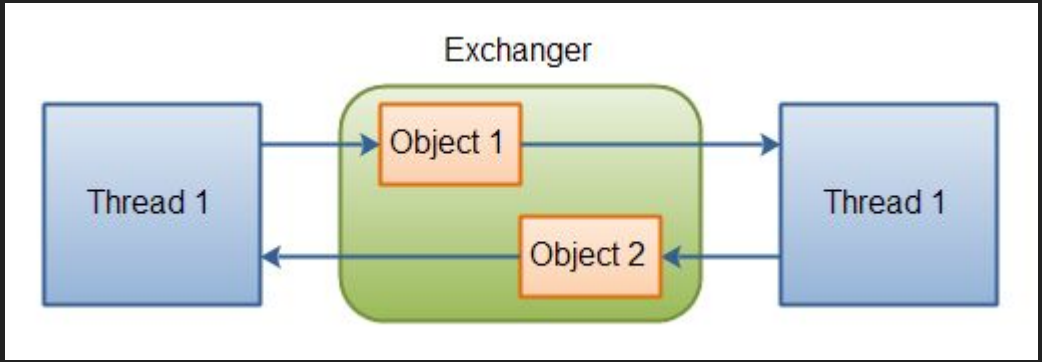


Phaser

- Similar to CyclicBarrier
- Dynamic adding / removing ‘parties’
- Better deadlock avoidance
- Termination API
- Phase counting
- Tiering - tree hierarchy

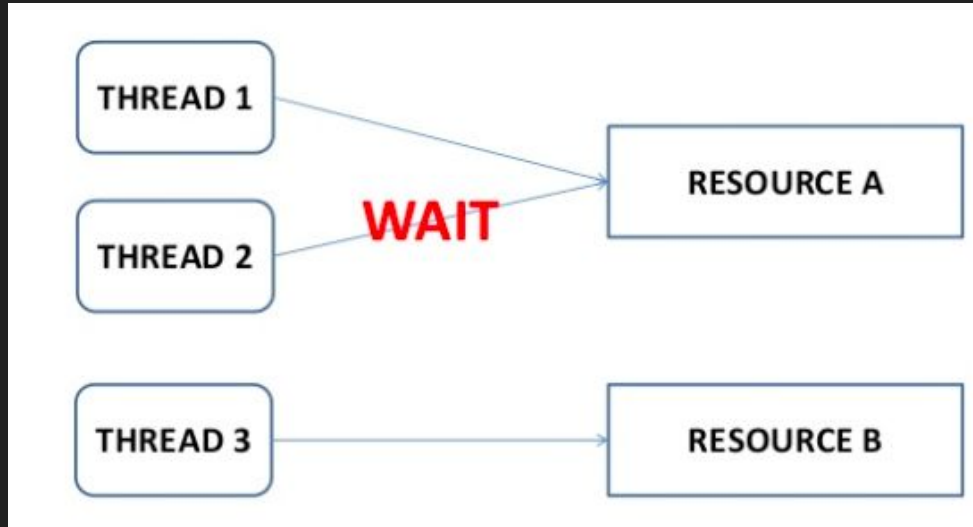
Other classes - Exchanger

- Helps two threads to exchange object
- *exchanger.exchange(object)* blocks
- has an *exchange(object, timeout, unit)* variant



Other Classes - Semaphore

- Limits number of threads accessing the same resource
- Like a bouncer at the club



And much, much more

- Collections and data structures
  - CopyOnWrite collections
  - Concurrent collections
  - Blocking, Synchronous Queues
  - Deques (pronounced “deck”) - double ended queue
- TimeUnit enum
- Lots of interfaces
- Lots of abstract classes

Summary

What you should know now:

- Atomic types
  - And that there are some specialized alternatives
- Locks
  - You should avoid them in general, but they might come handy
- There is more than *Executors.newFixedThreadPool(<the more the merrier>)*
- Futures are awsome tool how to model (sync and) asyc pipelines
- Java has Barriers
  - These are expecially usefull in async systems

Questions?

Any  ?

Thank you!

All examples at https://github.com/Skamaniak/JavaUtilConcurrentOverview