# Practical Concurrent and Parallel Programming III

IT University of Copenhagen

Friday 2020-09-11

**Kasper Østerbye**

## Plan for today

### Design of synchronized datastructures

### Building blocks

#### Synchronizers

### An example

## Hand-out of material

- Readings will be available at latest Thursday morning.
- Exercises will be available Thursday morning.
- Slides will be available Friday morning (yes, I edit them Thursday evening)
    - And regret I cannot update the exercises 😀.

## Hand-out of material

- Readings will be available at latest Thursday morning.
- Exercises will be available Thursday morning.
- Slides will be available Friday morning (yes, I edit them Thursday evening)
  - And regret I cannot update the exercises 😀.

Keep those questions coming in the forum!

## Selected issues

**Best practice for publishing objects.**

- Avoid this-escape. If the newly created object need to go into a data structure (even a global variable), use a private constructor and a *factory method*.
- Publish through a thread-safe datastructure or a final/volatile static member.
- Thread localization - sometimes it is possible to ensure only one thread generates objects of a specific type. For example a thread that produce objects for other to process.

## Selected issues

**Best practice for publishing objects.**

- Avoid this-escape. If the newly created object need to go into a data structure (even a global variable), use a private constructor and a *factory method*.
- Publish through a thread-safe datastructure or a final/volatile static member.
- Thread localization - sometimes it is possible to ensure only one thread generates objects of a specific type. For example a thread that produce objects for other to process.

**Static holding state shared between all instances of a class**

I saw many different solutions to the Person with unique ID.

## Person with ID 1

```java
class Person {
  private static long ID = 0;
  private final long id;
  // other fields
  Person (String name,...){
    id = ID++; // NOT THREAD SAFE
    // other initialization
  }
  // ...
}
```

# Person with ID 2

```java
class Person {
  private static long ID = 0;
  private final long id;
  // other fields
  Person (String name,...){
    synchronized(Person.class){
      id = ID++; // Thread safe
    }
    // other initialization
  }
  // ...
}
```

## Person with ID 3

```
class Person {
  private static AtomicLong ID = new AtomicLong(0);
  private final long id;
  // other fields
  Person (String name,...){
    id = ID.getAndIncrement();
    // other initialization
  }
  // ...
}
```

## Person with ID 4

Some software engineers things static variables is bad-taste, and is just confusing. Instead, use two classes:

```
class Person {}
@Singleton class PersonManager{
  public static final instance = new PersonManager();
}
```

```
class Person {
  Person (String name,...){
    id = PersonManager.instance.getNextID();
    // other initialization
  }
}



---
# Designing synchronized datatstructures
Beware that the static and non-static part are two objects, not one. Hence they ea

--
### The role of the representation invariant
A *representation invariant* is a logical statement of the relationship between th

The invariant must be true after the constructor has finished, and after each (pub
```

# Person with ID 4

Some software engineers things static variables is bad-taste, and is just confusing. Instead, use two classes:

```
class Person {}
@Singleton class PersonManager{
  public static final instance = new PersonManager();
}
```

```
class Person {
  Person (String name,...){
    id = PersonManager.instance.getNextID();
    // other initialization
  }
}



---
# Designing synchronized datatstructures
Beware that the static and non-static part are two objects, not one. Hence they ea

--
### The role of the representation invariant
A *representation invariant* is a logical statement of the relationship between th

The invariant must be true after the constructor has finished, and after each (pub
```

## Building blocks

## Concurrent collections

- Normal collections are in the `java.util` library. In 90% of the cases one use `List` or `Map`.

- The concurrent collections are in `java.util.concurrent`. There are many, and they are mostly specialized.

In particular:

- java.util.concurrent.ConcurrentHashMap
- But **no** java.util.concurrent.ConcurrentArrayList
- However, `Collections.synchronizedList(new ArrayList())` wraps an ArrayList

## No concurrent ArrayList

Consider these three operations on List:

- `set(index, elem)`
- `get(index)`
- `add(elem)`

The set/get can be solved by an substituting `ArrayList<E>` with `ArrayList< AtomicReference<E> >`.

# No concurrent ArrayList

Consider these three operations on List:

- `set(index, elem)`
- `get(index)`
- `add(elem)`

The set/get can be solved by an substituting `ArrayList<E>` with `ArrayList< AtomicReference<E> >`.

However, the add operation might occasionally need to copy the whole underlying array. To make this robust `add` needs to:

- close the list for updates and reads, waits for all currently ongoing operations to finish, and then updates.

?? There is a good Q/A on this on stackoverflow: [https://stackoverflow.com/questions/6916385/is-there-a-concurrent-list-in-javas-jdk](https://stackoverflow.com/questions/6916385/is-there-a-concurrent-list-in-javas-jdk)

## Concurrent iterators

A general decision one must make with iterators of collections is their semantics:

- Snapshot - they give the elements in the collection when the iterator was generated
- Unstable - the elements returned by the iterator are only guaranteed if the underlying collection is not changed while iterating.

Notice - both are useful. In particular the unstable can be done very efficient. If you can guarantee its premise, this is the one to go for.

## Concurrent iterators

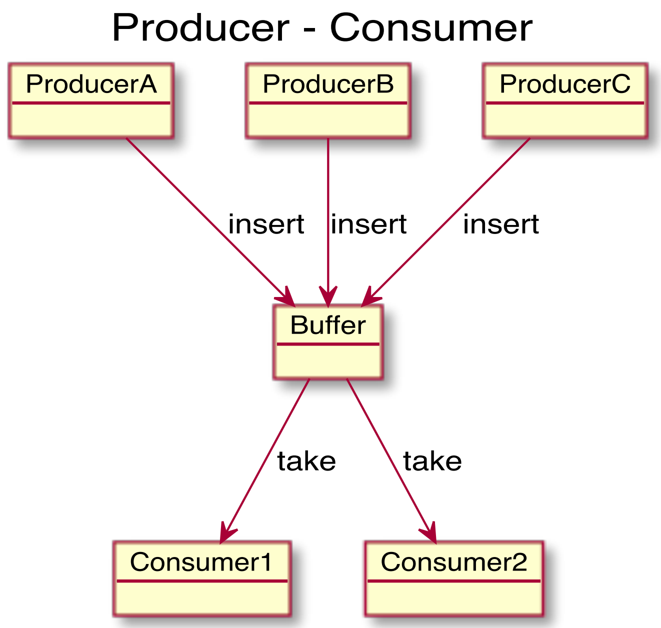
A general decision one must make with iterators of collections is their semantics:

- Snapshot - they give the elements in the collection when the iterator was generated
- Unstable - the elements returned by the iterator are only guaranteed if the underlying collection is not changed while iterating.

Notice - both are useful. In particular the unstable can be done very efficient. If you can guarantee its premise, this is the one to go for.

There is a `java.util.concurrent.CopyOnWriteArrayList<E>` which has snapshot semantics. It is highly inefficient with lots of modifications, but cool for lots of readings.

# The producer-consumer pattern



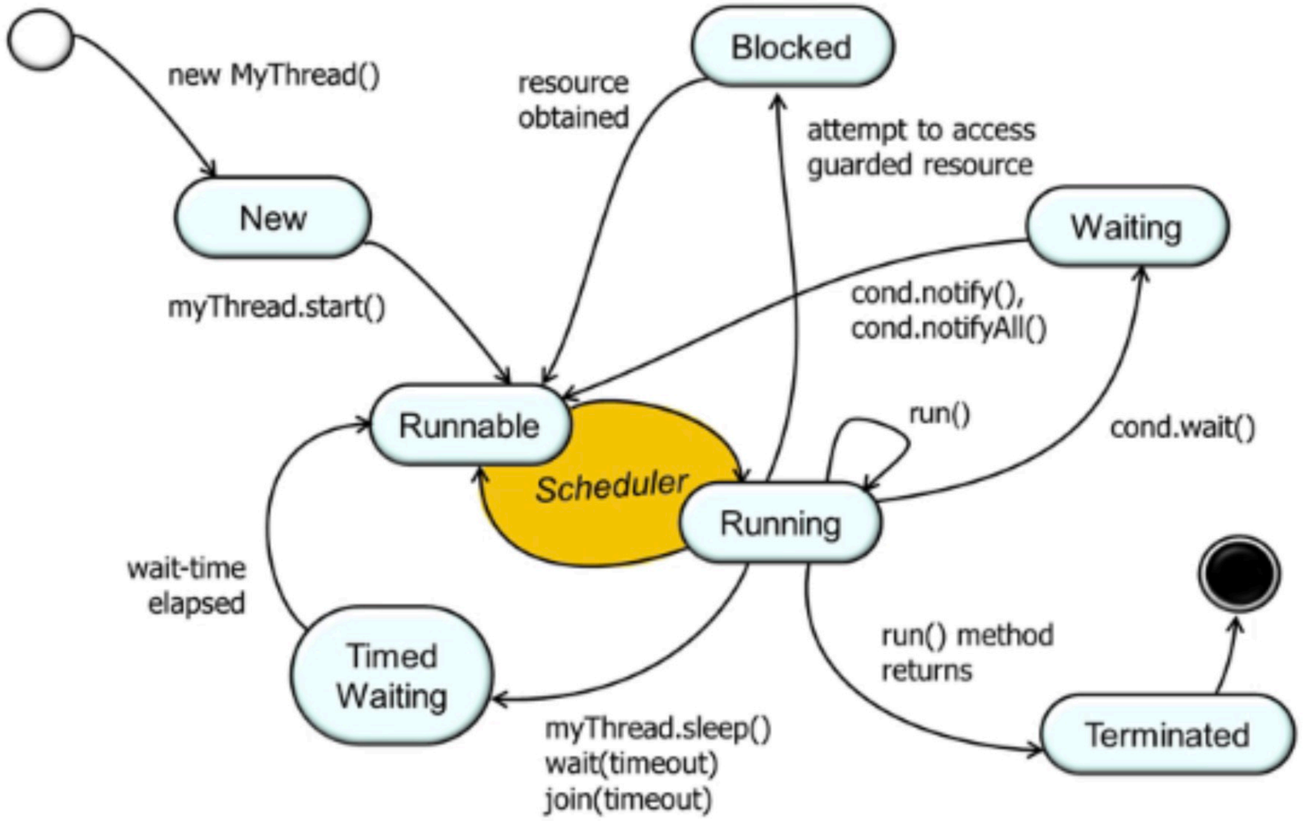
The idea is that a number of Producer threads generate items which the Consumer threads then take and do something with them.

Typically, the buffer is given a fixed size.

If the buffer is full, trying to insert will block the inserting Producer.

If the buffer is empty, trying to take will block the taking Consumer

# Blocking



Think of the states "Blocked" and "Waiting" as the same.

The thread is in a state where it cannot *-by itself-* get back to "Runnable"

# Interrupting a thread

```
otherThread.interrupt()
```

If this thread is blocked in an invocation of the

- wait(), wait(long), or wait(long, int) methods of the Object class, or of the
- join(), join(long), join(long, int), sleep(long), or sleep(long, int), methods of Thread,

then it will receive an InterruptedException.

```
try{
  Thread.sleep(2000);
  } catch (InterruptedException knock) {
    ...
  }
}
```

Notice: If `otherThread` is not currently blocked, all that will happen is that a flag is set, and `otherThread` can then examine if it was interrupted - and maybe do something special in that occasion.

## Interrupt

Should not be used for synchronization, use one of the library classes intended for this

## Synchronizers

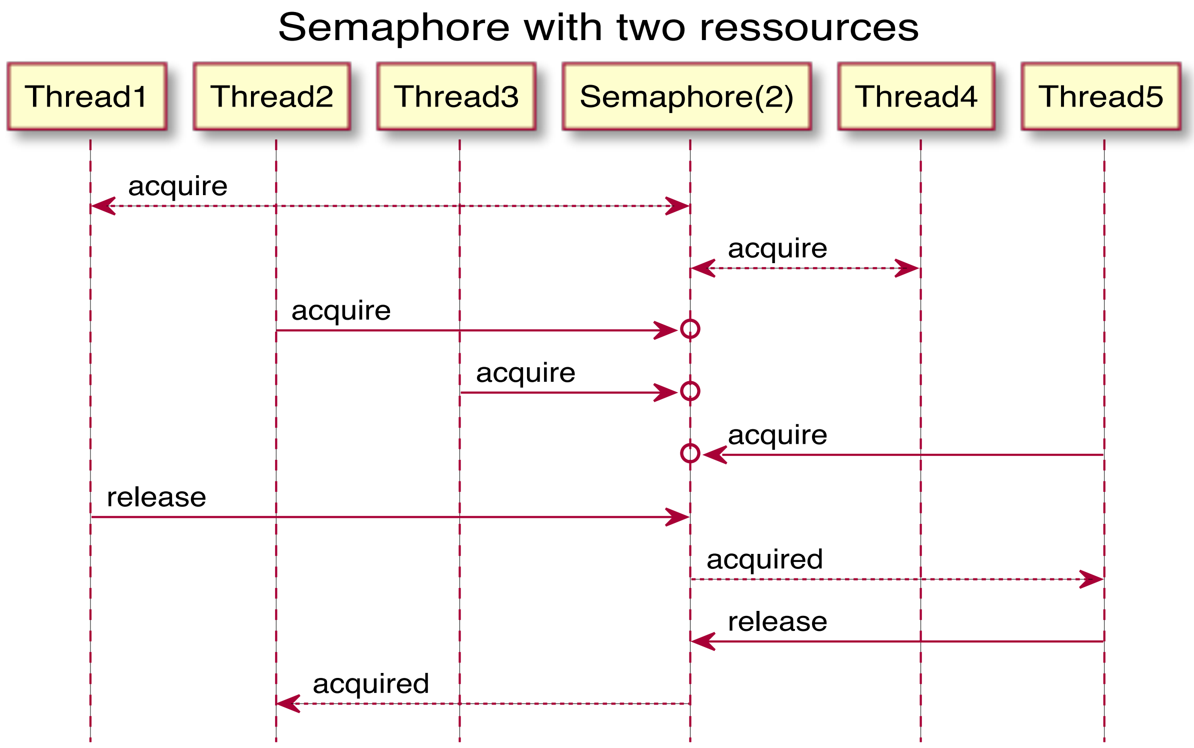
The java.util.concurrent has several kinds of objects which implements ways to synchronize threads for different scenarios.

## Semaphore

A semaphore is is used for guarding a ressource (or a number of resources)

When the train arrives at the semaphore, the train driver looks at the semaphore. If it is closed, they wait. If it is open, the train driver jumps out, lowers the semaphore (which is connected by a cable to a semaphore on the other end of the line), drives through, and raises the semaphore at the other end.

## Semaphore II

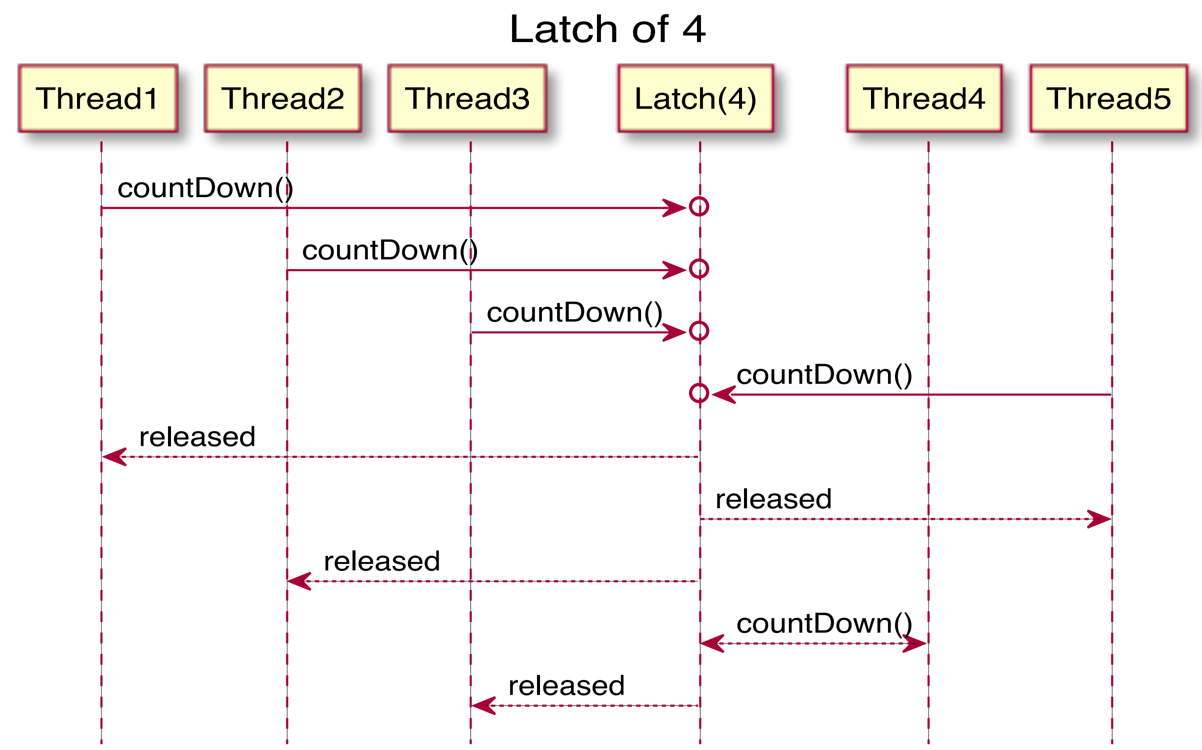


## Latch (CountDownLatch)

```java
class Driver { // ...
   void main() throws InterruptedException {
     CountDownLatch startSignal = new CountDownLatch(1),
     doneSignal = new CountDownLatch(N);
     for (int i = 0; i < N; ++i) // create and start threads
       new Thread(new Worker(startSignal, doneSignal)).start();

     doSomethingElse();            // don't let run yet
     startSignal.countDown();      // let all threads proceed
     doSomethingElse();
     doneSignal.await();           // wait for all to finish
}}
class Worker implements Runnable {
   private final CountDownLatch startSignal, doneSignal;
   Worker(CountDownLatch startSignal, CountDownLatch doneSignal) {
      this.startSignal = startSignal; this.doneSignal = doneSignal;
   }
   public void run() {
      try {
        startSignal.await();
        doWork();
        doneSignal.countDown();
      } catch (InterruptedException ex) {} // return;
   }
   void doWork() { ... }
}
```

## Latch (trapdoor?)



I think trapdoor because once all are present you fall through all at the same time (all threads are moved from blocked to runnable state).

## Not mentioned

Barriers (Goetz talk about them), Exchanger (not mentioned in Goetz).

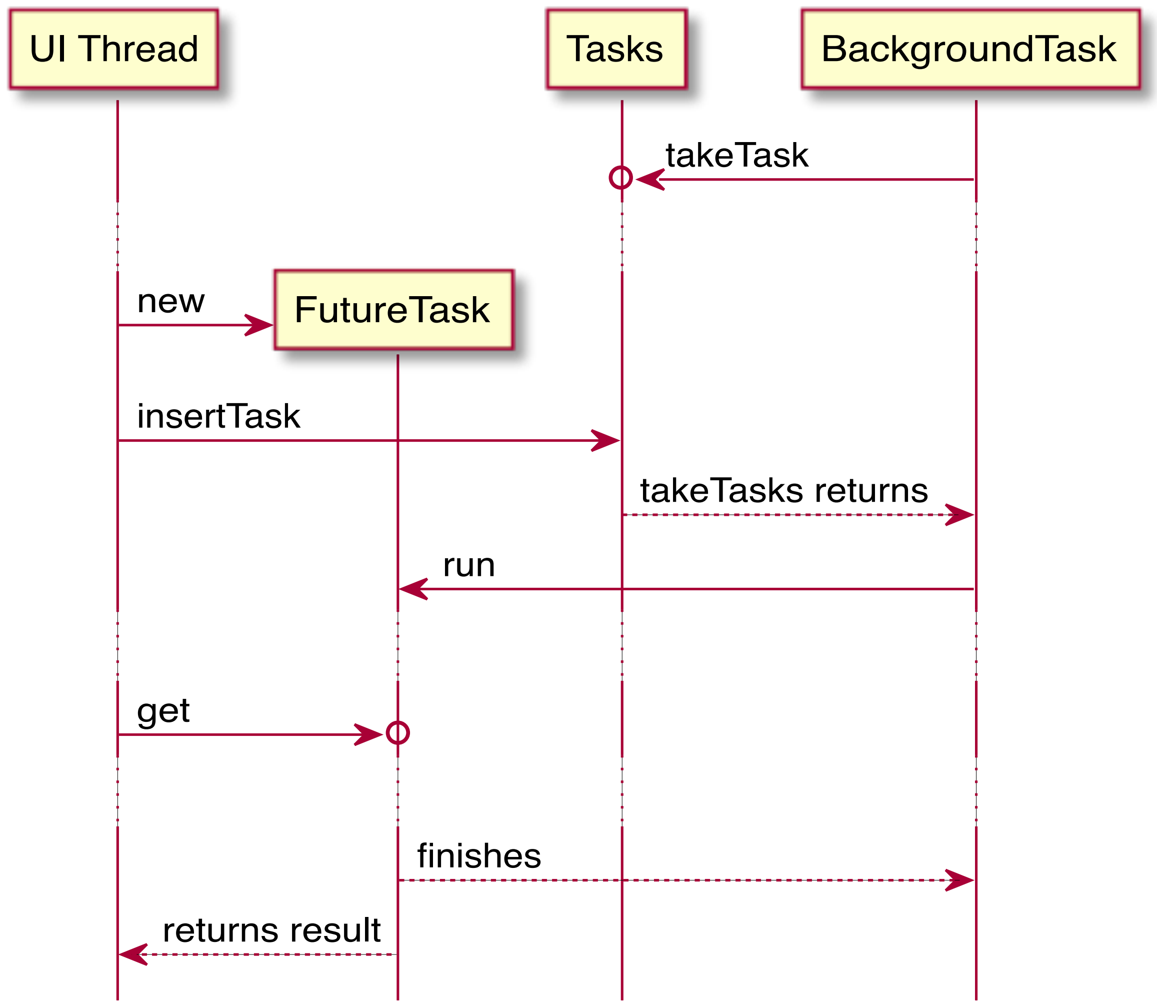
## FutureTask

I recently ordered a pair of kayak-shoes on the German Amazon. I did not expect to get the shoes delivered right when I ordered them, but I got a *tracking number*, which allow me to follow how the order is progressing.

`FutureTask` is a bit the same - you order a computation in an other thread (or group of threads).
The result of such an order is a `Future`, an object which at some point later in time will hold the result. A future is somewhat similar to the tracking number.

At any point in time I can ask if my order has been delivered, or I can go to the post-box and just wait. Same in Java. You can ask a `Future` is is has been computed - using the `boolean isDone()` method. Or just wait for the result (or get immediately if is has been finished) - using the `V get()` method.

# FutureTask - sequence diagram



## Building an efficient, scalable result cache

If you study this example, and get 90% of it, you are good for Part 1!

The example is advanced, and shows how much better one can do than *just* making all methods synchronized.

## Servlet using the cache

```java
public class Factorizer implements Servlet {
    private final Function<BigInteger, BigInteger[]> cache
            = new Memoizer<BigInteger, BigInteger[]>(c);

    public void service(ServletRequest req, ServletResponse resp) {
        try {
            BigInteger i = extractFromRequest(req);
            BigInteger result = cache.compute( i );
            encodeIntoResponse(resp, cache.compute(i));
        } catch (InterruptedException e) {
            encodeError(resp, "factorization interrupted");
        }
    }

    //...
}
```

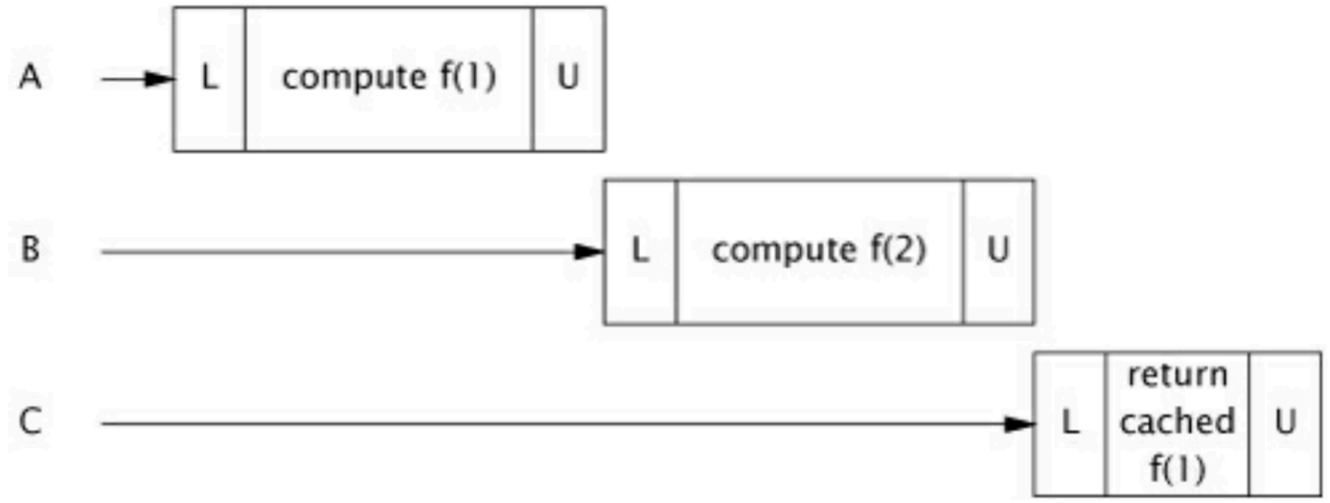
## Memoizer1

```
import java.util.function.Function;
public class Memoizer1 <Arg, Value> implements Function<A, Value> {
    private final Map<Arg, Value> cache = new HashMap<Arg, Value>();
    private final Function<Arg, Value> c;

    public Memoizer1(Function<Arg, Value> c) {
        this.c = c;
    }

    public synchronized Value compute(Arg arg) throws InterruptedException {
        Value result = cache.get(arg);
        if (result == null) {
            result = c.compute(arg);
            cache.put(arg, result);
        }
        return result;
    }
}
```

## Memoizer1 behaviour



*L is short for "Lock", and U for "Unlock"*

Thread A and B asks for two different results, while C asks for a result which has already been cached.

## Memoizer 2

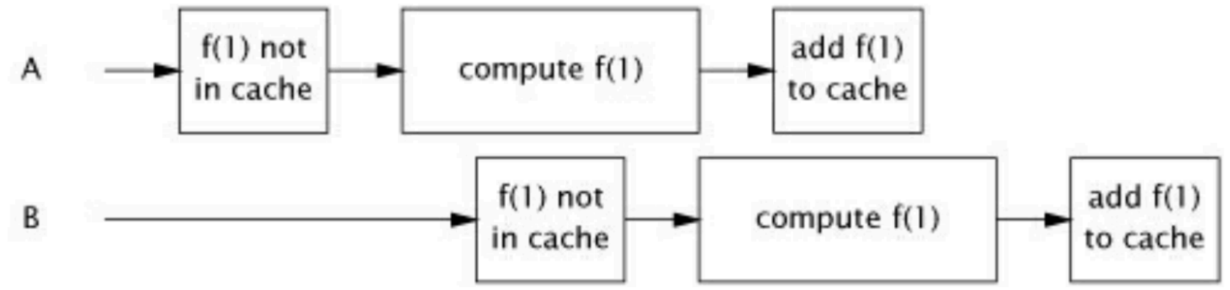
```
public class Memoizer2 <Arg, Value> implements Function<Arg, Value> {
    private final Map<Arg, Value> cache = new ConcurrentHashMap<Arg, Value>();
    private final Computable<Arg, Value> c;

    public Memoizer2(Computable<Arg, Value> c) {
        this.c = c;
    }

    public Value compute(Arg arg) throws InterruptedException {
        Value result = cache.get(arg);
        if (result == null) {
            result = c.compute(arg);
            cache.put(arg, result);
        }
        return result;
    }
}
```

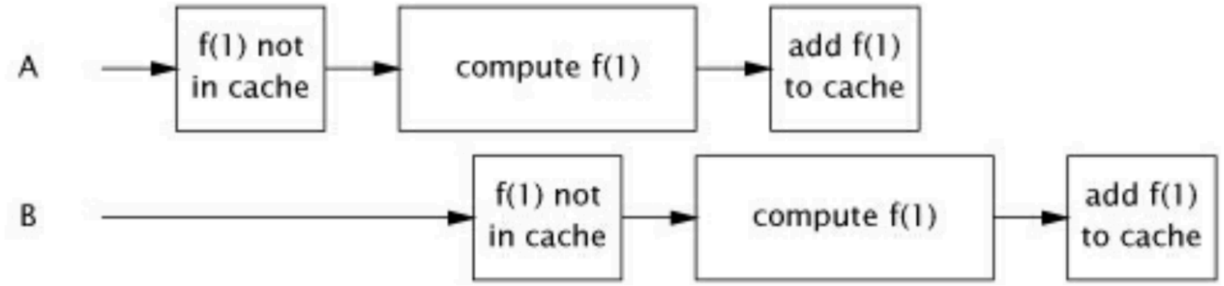
That looks good, not blocking others while I am computing the value…

## However, Memoizer2 behaviour (misbehaviour)



If there is a special interest in getting the factorization of 7774728793298178912098310 28 (or 1 in the figure)

## However, Memoizer2 behaviour (misbehaviour)



If there is a special interest in getting the factorization of 777472879329817891209831028 (or 1 in the figure)

We (might) end up computing the same value more than once.

## Final version, using FutureTask

```
public class Memoizer <Arg, Value> implements Function<Arg, Value> {
    private final ConcurrentMap<Arg, Future<Value> > cache
            = new ConcurrentHashMap<Arg, Future<Value> >();
    private final Function<Arg, Value> func;

    public Memoizer(Function<Arg, Value> func) {
        this.func = func;
    }

    public Value apply(final Arg arg) throws InterruptedException {
        while (true) {
            Future<Value> f = cache.get(arg);
            if (f == null) {
                FutureTask<Value> ft
                  = new FutureTask<Value>( () -> { func.compute(arg);} );
                f = cache.putIfAbsent(arg, ft);
                if (f == null) {
                    f = ft;
                    ft.run();
                }
            }
            try { return f.get(); }
            catch (Exception e) { /*... Misserable death */ }
        }
    }
}
```

## Goetz - summary chapter 1-5

- It's the mutable state that is the issue

## Goetz - summary chapter 1-5

- It's the mutable state that is the issue
- Make fields final unless they need to be mutable.

## Goetz - summary chapter 1-5

- It's the mutable state that is the issue
- Make fields final unless they need to be mutable.
- Immutable objects are automatically thread-safe.

## Goetz - summary chapter 1-5

- It's the mutable state that is the issue
- Make fields final unless they need to be mutable.
- Immutable objects are automatically thread-safe.
- Immutable objects simplify concurrent programming tremendously.

## Goetz - summary chapter 1-5

- It's the mutable state that is the issue
- Make fields final unless they need to be mutable.
- Immutable objects are automatically thread-safe.
- Immutable objects simplify concurrent programming tremendously.
- Encapsulation makes it practical to manage the complexity.

# Goetz - summary chapter 1-5

- It's the mutable state that is the issue
- Make fields final unless they need to be mutable.
- Immutable objects are automatically thread-safe.
- Immutable objects simplify concurrent programming tremendously.
- Encapsulation makes it practical to manage the complexity.
- Guard all variables in an invariant with the same lock.
- Hold locks for the duration of compound actions (Kasper - and no longer)
- A program that accesses a mutable variable from multiple threads without synchronization is a broken program.
- Don't rely on clever reasoning about why you don't need to synchronize.
- Include thread safety in the design process—or explicitly document that your class is not thread-safe.
- Document your synchronization policy.

*Java Concurrency in Practice (p. 194). Pearson Education. Kindle Edition.*