Practical Concurrent and Parallel Programming 2

Thomas Dybdahl Ahle
IT University of Copenhagen

Thursday 2019-09-04

Plan for today

- Primitive atomic operations: AtomicLong, ...
- Immutability, **`final`**, and safe publication
- Java monitor pattern
- Standard collection classes not thread-safe
- FutureTask<T> and asynchronous execution
- Building a scalable result cache
- Defensive copying (VehicleTracker)

Based on slides by
Peter Sestoft

Exercises

- Last week's exercises:
  - Too easy?
  - Too hard?
  - Too time-consuming?
  - Too confusing?
  - Any particular problems?

Goetz examples use servlets

```
public class StatelessFactorizer implements Servlet {
  public void service(ServletRequest req, ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    BigInteger[] factors = factor(i);
    encodeIntoResponse(resp, factors);
  }
}
```

Goetz p. 19

- Because a webserver is naturally concurrent
  - So servlets should be thread-safe
- We use similar, simpler examples:

```
class StatelessFactorizer implements Factorizer {
  public long[] getFactors(long p) {
    long[] factors = PrimeFactors.compute(p);
    return factors;
  }
}
```

TestFactorizer.java

A "server" for computing prime factors 2 3 5 7 11 … of a number

- Could replace the example by this

```
interface Factorizer {
  public long[] getFactors(long p);
  public long getCount();
}
```

- Call the server from multiple threads:

```
for (int t=0; t<threadCount; t++) {
  threads[t] =
    new Thread(() -> {
      for (int i=2; i<range; i++) {
        long[] result = factorizer.getFactors(i);
      }
    });
  threads[t].start();
}
```

Stateless objects are thread-safe

```
class StatelessFactorizer implements Factorizer {
  public long[] getFactors(long p) {
    long[] factors = PrimeFactors.compute(p);
    return factors;
  }
  public long getCount() { return 0; }
}
```

Like Goetz p. 18

- Local variables (**`p`**, **`factors`**) are never shared between threads
  - two getFactors calls can execute at the same time

Bad attempt to count calls

```
class UnsafeCountingFactorizer implements Factorizer {
  private long count = 0;
  public long[] getFactors(long p) {
    long[] factors = PrimeFactors.compute(p);
    count++;
    return factors;
  }
  public long getCount() { return count; }
}
```

Like Goetz p. 19

- Not thread-safe
- Q: Why?
- Q: How could we make it thread-safe?

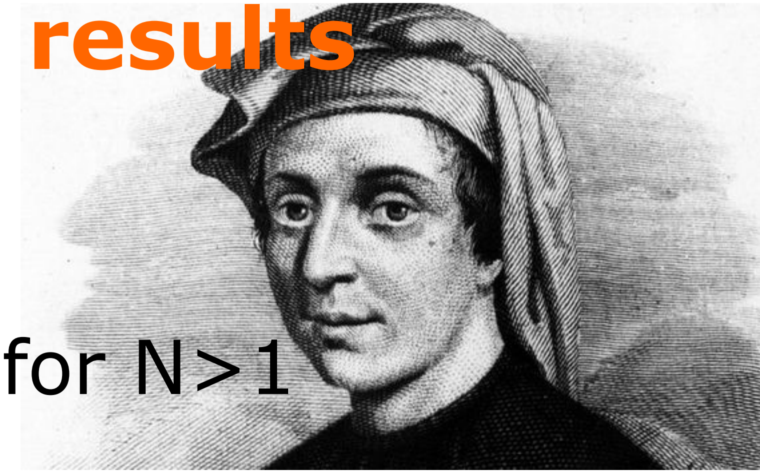
Thread-safe server counting calls

```
class CountingFactorizer impleme
  private final AtomicLong count
  public long[] getFactors(long
    long[] factors = PrimeFactor
    count.incrementAndGet();
    return factors;
  }
  public long getCount() { retur
}
```

Like Goetz p. 23

- java.util.concurrent.atomic
  supports atomic thread-safe arithmetics
- Similar to a thread-safe LongCounter class

Caching computed results



- Fibonacci numbers:
  F(0) = F(1) = 1
  F(N) = F(N-1) + F(N-2) for N>1
- F(N) is exponential
- Naïve recursive implementation:
  F(N) ~ 1.6^N operations
- Iterative / dynamic programming with memoization: O(N) operations
- Serial java 8:
  HashMap.computeIfAbsent(...)

Bad attempt to cache last factorization

```
class TooSynchrCachingFactorizer implements Factorizer {
  private long lastNumber = 1;
  private long[] lastFactors = new long[] { 1 };
  // Invariant: product(lastFactors) == lastNumber

  public synchronized long[] getFactors(long p) {
    if (p == lastNumber)
      return lastFactors.clone();
    else {
      long[] factors = PrimeFactors.compute(p);
      lastNumber = p;
      lastFactors = factors;
      return factors;
} } }
```

cache

Like Goetz p. 26

Without **synchronized** the two fields could be written by different threads

- Bad performance: no parallelism at all
- Q: Why?

Atomic operations

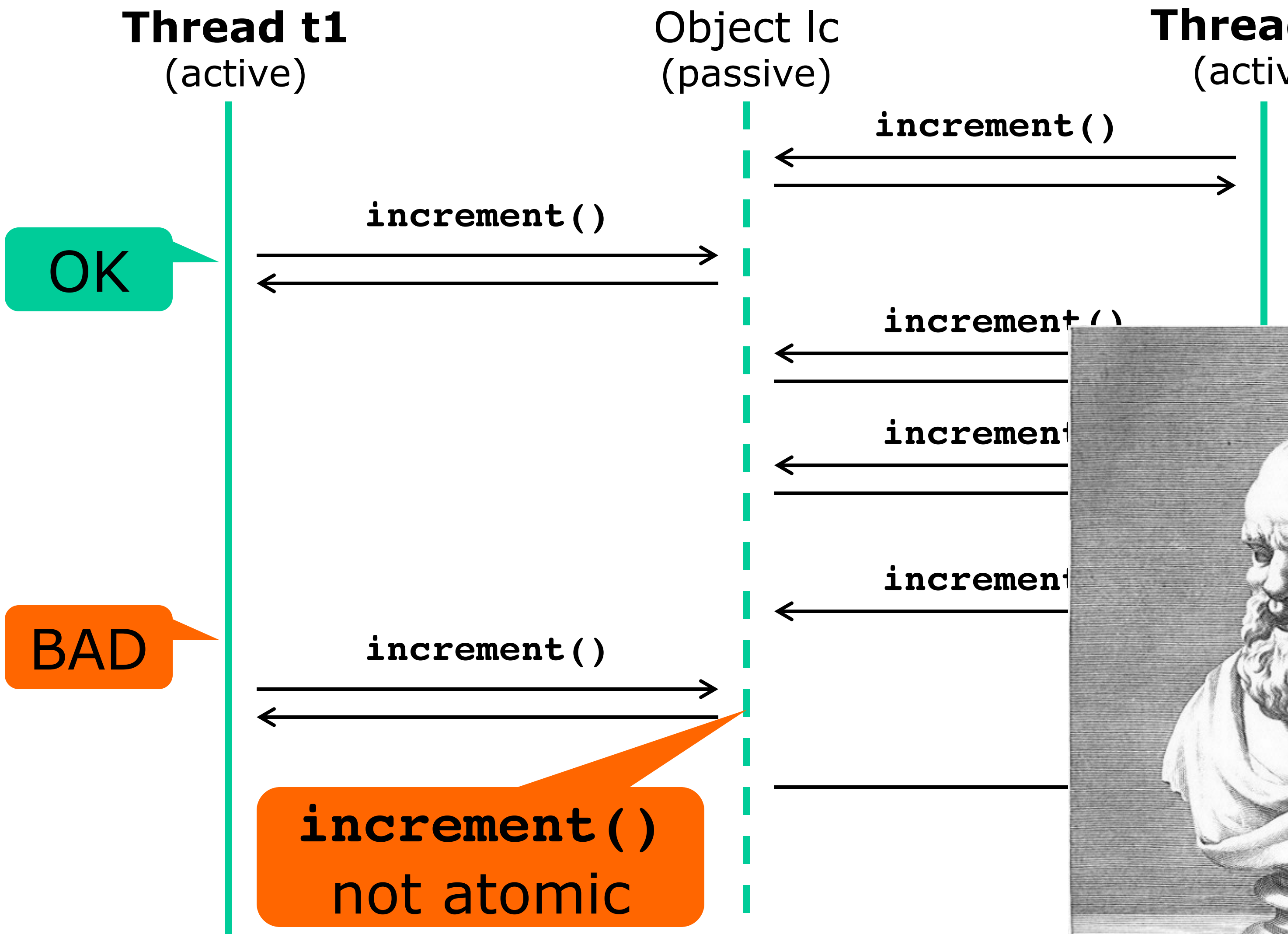
- We want to *atomically* update *both* **`lastNumber`** and **`lastFactors`**

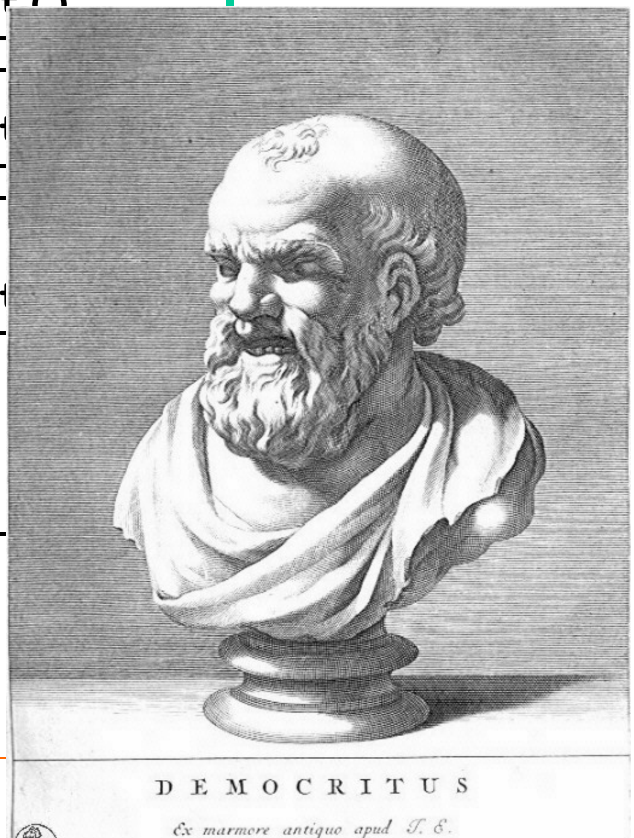
Operations A and B are *atomic* with respect to each other if, from the perspective of a thread executing A, when another thread executes B, either all of B has executed or none of it has.

An *atomic operation* is one that is atomic with respect to all operations (including itself) that operate on the same state.

Goetz p. 22, 25

Lack of atomicity: overlapping reads and writes





Atomic update without excess locking

```
class CachingFactorizer implements Factorizer {
  private long lastNumber = 6;
  private long[] lastFactors = new long[] { 2,3 };
  public long[] getFactors(long p) {
    long[] factors = null;
    synchronized (this) {
      if (p == lastNumber)
        factors = lastFactors.clone();
    }
    if (factors == null) {
      factors = PrimeFactors.compute(p);
      synchronized (this) {
        lastNumber = p;
        lastFactors = factors.clone();
      }
    }
    return factors;
} }
```

Atomic test-then-act

Atomic write of both fields

Like Goetz p. 31

Using locks for atomicity

For each mutable state variable that may be accessed by more than one thread, **all** accesses to that variable must be performed with the **same** lock held. Then the variable is *guarded* by that lock.

For every invariant that involves more than one variable, **all** the variables involved in that invariant must be guarded by the **same** lock.

Goetz p. 28, 29

- Common mis-reading and mis-reasoning:
  - The *purpose* of **`synchronized`** is to get atomicity
  - So **`synchronized`** roughly means "**`atomic`**"
  - True only if **all other** accesses are **`synchronized`**!!!

Wrong

Alternative: Wrap the state in an immutable object

```
class OneValueCache {
  private final long lastNumber;
  private final long[] lastFactors;
  public OneValueCache(long p, long[] factors) {
    this.lastNumber = p;
    this.lastFactors = factors.clone();
  }
  public long[] getFactors(long p) {
    if (lastFactors == null || lastNumber != p)
      return null;
    else
      return lastFactors.clone();
  }
}
```

The fields cannot change between test and return

Like Goetz p. 49

- Immutable, so automatically thread-safe

Make the state a single field, referring to an immutable object

```
class VolatileCachingFactorizer implements Factorizer {
  private volatile OneValueCache cache
    = new OneValueCache(0, null);
  public long[] getFactors(long p) {
    long[] factors = cache.getFactors(p);
    if (factors == null) {
      factors = PrimeFactors.compute(p);
      cache = new OneValueCache(p, factors);
    }
    return factors;
  }
}
```

Single-field state, atomic assignment

Atomic update

Like Goetz p. 50

- Only one mutable field, atomic update
- Easy to implement, easy to see it is correct
- Allocates many OneValueCache objects: Bad?
  - Not a problem with modern garbage collectors

Make the state a single field, referring to an immutable object

```
public class Main {
    class Holder {
        Holder(Main outer) {
            System.out.println(outer.inner);
        }
    }
    Holder inner;
    void run() {
        inner = new Holder(this);
    }
}
```

- What does this program print?
- How can we find out?

Make the state a single field, referring to an immutable object

```
class Main {
    public Holder holder;
    public void initialize() {
        holder = new Holder(42);
    }
}
public class Holder {
  private int n;
  public Holder(int n) { this.n = n; }
  public void assertSanity() {
    if (n != n)
      throw new AssertionError("This statement is false");
  }
}
```

Making holder volatile ensures that " all writes that happen prior to the volatile store are visible to all subsequent threads of the volatile field"

- What's going on here? How can the test ever fail?

Safe publication: visibility

- The **`final`** field modifier has two effects
  - **Non-updatability** can be checked by the compiler
  - **Visibility** from other threads of the fields' values after the constructor returns
- So **`final`** has *visibility effect* like **`volatile`**
- Without **`final`** or synchronization, another thread may not see the given field values

- That was Java. What about C#/.NET?
  - No visibility effect of **`readonly`** field modifier
  - So must be ensured by locking or MemoryBarrier
  - Seems a little dangerous?

Immutability

- OOP: An object has state, held by its fields
  - Fields should be **`private`** for encapsulation
  - It is common to define getters and setters

But mutable state causes lots of problems

- Immutable design:
  - Each object has one state
  - Each state an object

Immutable objects are always thread-safe.

An object is *immutable* if:

- Its state cannot be modified after construction
- All its fields are **`final`**
- It is properly constructed (**`this`** does not escape)

Goetz p. 46, 47

Bloch: Effective Java, item 15

Item 15: Minimize mutability

An immutable class is simply a class whose instances cannot be modified. All of the information contained in each instance is provided when it is created and is fixed for the lifetime of the object. The Java platform libraries contain many immutable classes, including `String`, the boxed primitive classes, and `BigInteger` and `BigDecimal`. There are many good reasons for this: Immutable classes are easier to design, implement, and use than mutable classes. They are less prone to error and are more secure.

To make a class immutable, follow these five rules:

1. **Don't provide any methods that modify the object's state** (known as *mutators*).
2. **Ensure that the class can't be extended.** This prevents careless or malicious subclasses from compromising the immutable behavior of the class by behaving as if the object's state has changed. Preventing subclassing is generally ac-

Classes should be immutable unless there's a very good reason to make them mutable. Immutable classes provide many advantages, and their only disadvan-

forced by the system. Also, it is necessary to ensure correct behavior if a reference to a newly created instance is passed from one thread to another without synchronization, as spelled out in the *memory model* [JLS, 17.5; Goetz06 16].

4. **Make all fields private.** This prevents clients from obtaining access to muta-

Josh Bloch designed the Java collection classes

A serious Java (or C#) developer should own and use this book.

Good for beating people in the head.

Bloch p. 73

Remember our Cache object

```
class OneValueCache {
  private final long lastNumber;
  private final long[] lastFactors;
  public OneValueCache(long p, long[] factors) {
    this.lastNumber = p;
    this.lastFactors = factors.clone();
  }
  public long[] getFactors(long p) {
    if (lastFactors == null || lastNumber != p)
      return null;
    else
      return lastFactors.clone();
  }
}
```

The fields cannot change between test and return

Like Goetz p. 49

- Immutable, so automatically thread-safe

Why .clone() in the factorizers?

```
public long[] getFactors(long p) {
  ...
  factors = lastFactors.clone();
  ...
  lastFactors = factors.clone();
  ...
}
```

- Because Java array elements are mutable
- So unsafe to share an array with just anybody
- Must *defensively clone* the array when passing a reference to other parts of the program
- This is a problem in sequential code too, but much worse in concurrent code
  - Minimize Mutability!

The classic collection classes are not threadsafe

```
final Collection<Integer> coll = new HashSet<Integer>();
final int itemCount = 100_000;
Thread addEven = new Thread(new Runnable() { public void run() {
  for (int i=0; i<itemCount; i++)
    coll.add(2 * i);
}});
Thread addOdd = new Thread(new Runnable() { public void run() {
  for (int i=0; i<itemCount; i++)
    coll.add(2 * i + 1);
}});
```

TestCollection.java

- May give wrong results or obscure exceptions:

```
There are 169563 items, should be 200000
```

```
"Thread-0" ClassCastException: java.util.HashMap$Node cannot be
cast to java.util.HashMap$TreeNode
```

- Wrap as synchronized coll. for thread safety

```
final Collection<Integer> coll
  = Collections.synchronizedCollection(new HashSet<Integer>());
```

Collections in a concurrent context

- Preferably use a modern concurrent collection class from java.util.concurrent.*
  - Operations **`get`**, **`put`**, **`remove`** ... are thread-safe
  - But iterators and **`for`** are only *weakly consistent:*
    - they may proceed concurrently with other operations
    - they will never throw `ConcurrentModificationException`
    - they are guaranteed to traverse elements as they existed upon construction exactly once, and may (but are not guaranteed to) reflect any modifications subsequent to construction.
- Or else wrap collection as synchronized
- Or synchronize accesses yourself
- Or make a thread-local copy of the collection and iterate over that

Java 8 class library documentation

Part 2

- About functional programming?

Callable<T> versus Runnable

- A Runnable is one method that returns nothing

```
public interface Runnable {
  public void run();
}
```

`unit -> unit`

- A java.util.concurrent.Callable<T> returns a T:

```
public interface Callable<T> {
  public T call() throws Exception;
}
```

`unit -> T`

```
Callable<String> getWiki = new Callable<String>() {
  public String call() throws Exception {
    return getContents("http://www.wikipedia.org/", 10);
}};
// Call the Callable, block till it returns:
try { String homepage = getWiki.call(); ... }
catch (Exception exn) { throw new RuntimeException(exn); }
```

TestCallable.java

Synchronous FutureTask<T>

```
Callable<String> getWiki = new Callable<String>() {
  public String call() throws Exception {
    return getContents("http://www.wikipedia.org/", 10);
}};
FutureTask<String> fut = new FutureTask<String>(getWiki);
fut.run();
try {
  String homepage = fut.get();
  System.out.println(homepage);
}
catch (Exception exn) { throw new RuntimeException(exn); }
```

Run **call()** on "main" thread

Get result of **call()**

Similar to .NET System.Threading.Tasks.Task<T>

- A FutureTask<T>
  - Produces a T
  - Is created from a Callable<T>
  - Above we run it synchronously on the main thread
  - More useful to run asynchronously on other thread

Asynchronous FutureTask<T>

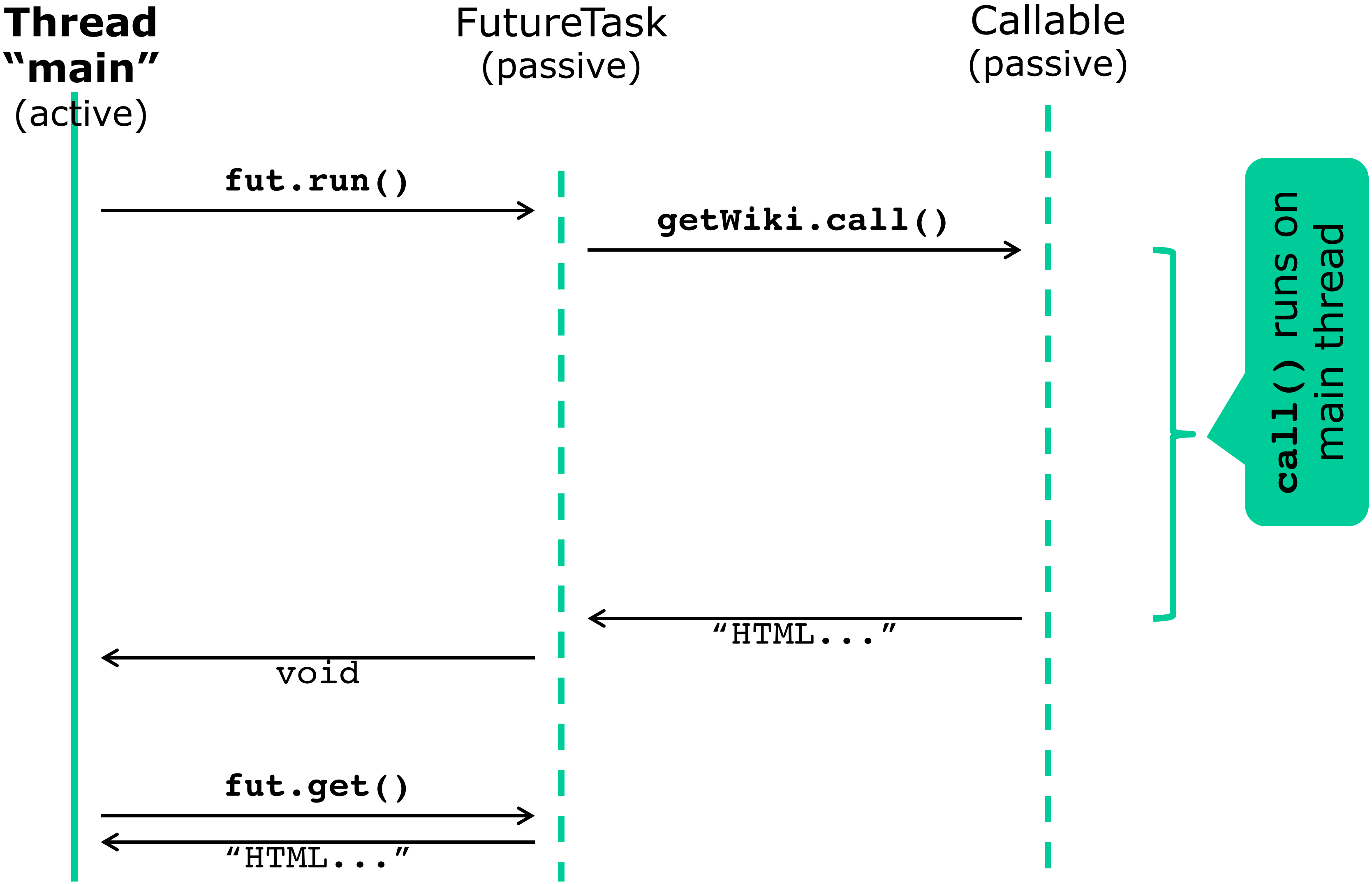
```
Callable<String> getWiki = new Callable<String>() {
  public String call() throws Exception {
    return getContents("http://www.wikipedia.org/", 10);
}};
FutureTask<String> fut = new FutureTask<String>(getWiki);
Thread t = new Thread(fut);
t.start();
try {
  String homepage = fut.get();
  System.out.println(homepage);
}
catch (Exception exn) { throw new RuntimeException(exn); }
```

Create and start thread running **call()**

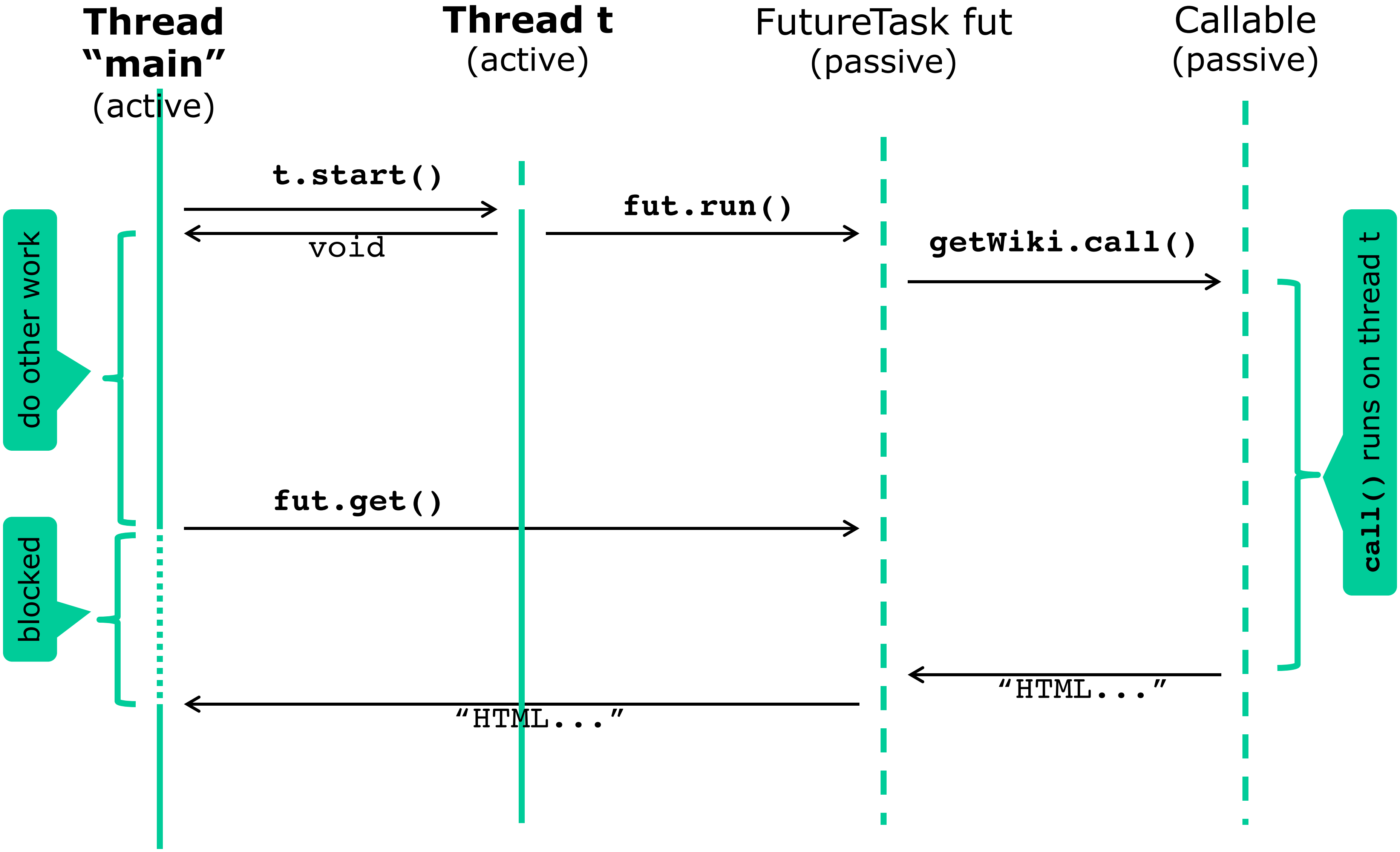
Block until **call()** completes

- The "main" thread can do other work between **t.start()** and **fut.get()**
- FutureTask can also be run as a *task*, week 5

Synchronous FutureTask



Asynchronous FutureTask



Those @$%&!!! checked exceptions

- Our exception han … s:

If **call()** throws **exn**, then **get** … p a throws ExecutionException(**ex** … that

```
try { String homepage = f
catch (Exception exn) { t                      n); }
```

- Goetz has a bett … ach:

```
try { S
catch (                 exn) {
  Throw                 tCause();
  if (c                 xception)
    thr                 se;
  else
    thr                 (cause);
}
```

"When you go through college and you're doing assignments, they just ask you to code up the one true path [of execution where failure isn't a consideration]. I certainly never experienced a college course where error handling was at all discussed. You come out of college and the only stuff you've had to deal with is the one true path."

Rethrow "expected" **call()** exceptions

Turn others into unchecked exceptions

Like Goetz p. 97

Goetz's scalable result cache

- Wrapping a computation so that it caches results and reuses them
  - Example: Given URL, computation fetches webpage
  - If URL is requested again, cache returns webpage

- Versions of Goetz's result cache ("Memoizer")
  - M1: lock-based, not scalable
  - M2: ConcurrentMap, large risk of computing twice
  - M3: use FutureTask, small risk of computing twice
  - M4: use putIfAbsent, no risk of computing twice
  - M5: use computeIfAbsent (Java 8), no risk of ...
    - See also Exercise 2.4.7

Goetz's scalable result cache

Goetz p. 103

- Interface representing functions from A to V

```
interface Computable <A, V> {
  V compute(A arg) throws InterruptedException;
}
```

A -> V

- Example 1: Our prime factorizer

```
class Factorizer implements Computable<Long, long[]> {
  public long[] compute(Long wrappedP) {
    long p = wrappedP;
    ...
} }
```

TestCache.java

- Example 2: Fetching a web page

```
class FetchWebpage implements Computable<String, String> {
  public String compute(String url) {
    ... create Http connection, fetch webpage ...
} }
```

Thread-safe but non-scalable cache

M1

```
class Memoizer1<A, V> implements Computable<A, V> {
  private final Map<A, V> cache = new HashMap<A, V>();
  private final Computable<A, V> c;

  public Memoizer1(Computable<A, V> c) { this.c = c; }

  public synchronized V compute(A arg) throws InterruptedEx... {
    V result = cache.get(arg);
    if (result == null) {
      result = c.compute(arg);
      cache.put(arg, result);
    }
    return result;
  }
}
```

If not in cache, compute and put

Goetz p. 103

```
Computable<Long, long[]> factorizer = new Factorizer(),
  cachingFactorizer = new Memoizer1<Long,long[]>(factorizer);
long[] factors = cachingFactorizer.compute(7182763656381322L);
```

- Not scalable as we have seen before.

Thread-safe scalable cache, using concurrent hashmap

M2

```
class Memoizer2<A, V> implements Computable<A, V> {
  private final Map<A, V> cache = new ConcurrentHashMap<A, V>();
  private final Computable<A, V> c;

  public Memoizer2(Computable<A, V> c) { this.c = c; }

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

Goetz p. 105

- But large risk of computing same thing twice
  - Argument put in cache only after computing result
    - so cache may be updated long after **`compute(arg)`** call

How Memoizer2 can duplicate work

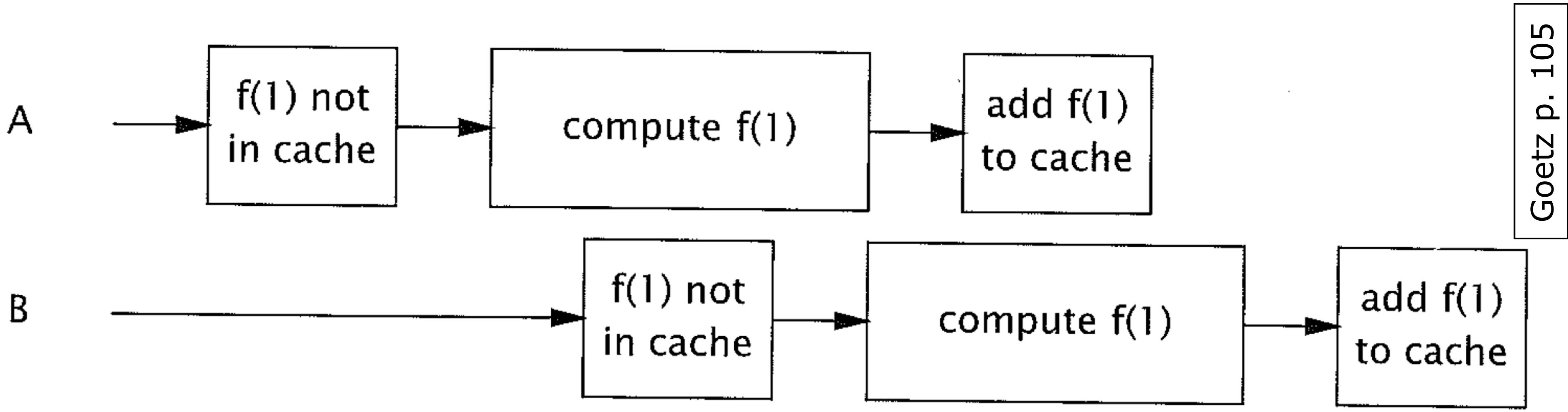


FIGURE 5.3. Two threads computing the same value when using Memoizer2.

- Better approach, Memoizer3:
  - Create a FutureTask for **`arg`**
  - Add the FutureTask to cache immediately at **`arg`**
  - Run the future on the calling thread
  - Return **`fut.get()`**

M3

Thread-safe scalable cache using FutureTask<V>, v. 3

Goetz p. 106

```
class Memoizer3<A, V> implements Computable<A, V> {
  private final Map<A, Future<V>> cache
    = new ConcurrentHashMap<A, Future<V>>();
  private final Computable<A, V> c;

  public V compute(final A arg) throws InterruptedException {
    Future<V> f = cache.get(arg);
    if (f == null) {
      Callable<V> eval = () -> c.compute(arg);
      FutureTask<V> ft = new FutureTask<V>(eval);
      cache.put(arg, ft);
      f = ft;
      ft.run();
    }
    try { return f.get(); }
    catch (ExecutionException e) { ... }
  }
```

If arg not in cache ...

... make future, add to cache ...

... run it on calling thread

Block until completed

Memoizer3 can still duplicate work

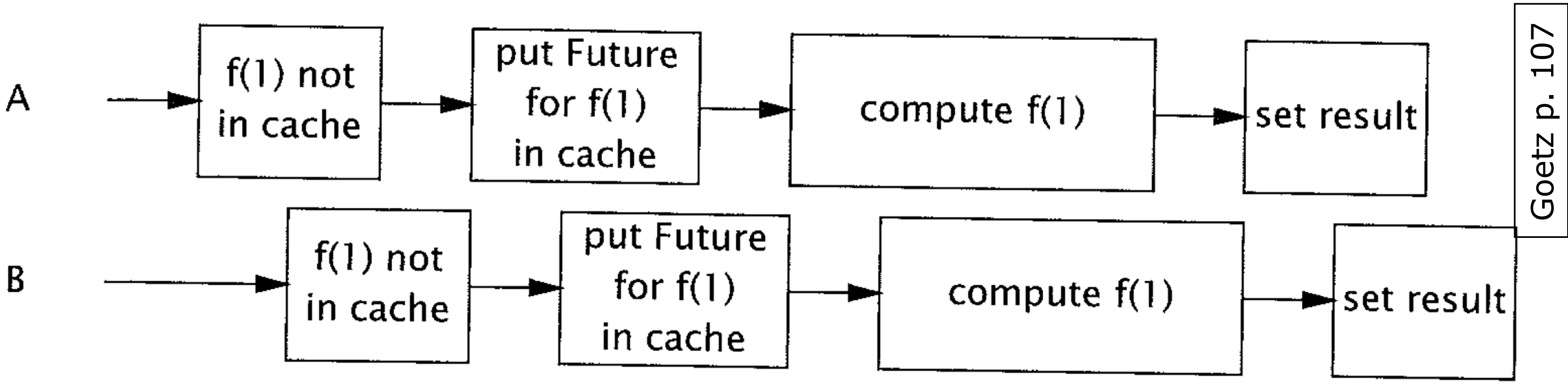


FIGURE 5.4. Unlucky timing that could cause `Memoizer3` to calculate the same value twice.

- Better approach, Memoizer4:
  - Fast initial check for **`arg`** cache
  - If not, create a future for the computation
  - Atomic put-if-absent may add future to cache
  - Run the future on the calling thread
  - Return **`fut.get()`**

Thread-safe scalable cache using FutureTask<V>, v. 4

M4

TestCache.java

```
class Memoizer4<A, V> implements Computable<A, V> {
  private final Map<A, Future<V>> cache
    = new ConcurrentHashMap<A, Future<V>>();
  private final Computable<A, V> c;
  public V compute(final A arg) throws InterruptedException {
    Future<V> f = cache.get(arg);
    if (f == null) {

      Callable<V> eval = () -> c.compute(arg);
      FutureTask<V> ft = new FutureTask<V>();

      f = cache.putIfAbsent(arg, ft);

      if (f == null) {
        f = ft; ft.run();
      }
    }
    try { return f.get(); }
    catch (ExecutionException e) { throw launderThrowable(...); }
  }
```

Fast test: If arg not in cache …

… make future

… atomic put-if-absent

… run on calling thread if not added to cache before

The technique used in Memoizer4

- Suggestion by Bloch item 69:
  - Make a fast (non-atomic) test for arg in cache
  - If not there, create a future object
  - Then atomically put-if-absent (arg, future)
    - If the arg was added in the meantime, do not add
    - Otherwise, add (arg, future) and run the future
- May wastefully create a future, but only rarely
  - The garbage collector will remove it
- Java 8 has computeIfAbsent, can avoid the two-stage test (see next slide)

Thread-safe scalable cache using FutureTask<V>, v. 5 (Java 8)

M5

TestCache.java

```
class Memoizer5<A, V> implements Computable<A, V> {
  private final Map<A, Future<V>> cache
    = new ConcurrentHashMap<A, Future<V>>();
  private final Computable<A, V> c;
  public V compute(final A arg) throws InterruptedException {
    final AtomicReference<FutureTask<V>> ftr = new ...();
    Future<V> f = cache.computeIfAbsent(arg,  argv -> {
         Callable<V> eval = () -> c.compute(argv);
         ftr.set(new FutureTask<V>(eval));
         return ftr.get();
      });
    if (ftr.get() != null)
      ftr.get().run();
    try { return f.get(); }
    catch (ExecutionException e) { ... }
  }
```

make future

... run on calling thread if not already in cache


Book
Review:
Joshua Bloch
Updated for Java 9
Effective Java
Third Edition
Best practices for
...the Java Platform
www.thoughts-on-java.org