Practical Concurrent and Parallel Programming 2

Peter Sestoft
IT University of Copenhagen

Friday 2015-09-04

Plan for today

- java.util.concurrent.atomic.AtomicLong
- Safe publication
- Thread and stack confinement
- Immutability
- Java monitor pattern
- Defensive copying, VehicleTracker
- Standard collection classes not thread-safe
- Extending collection classes
- ConcurrentModificationException
- FutureTask<T> and asynchronous execution
- (Silly complications of checked exceptions)
- Building a scalable result cache

Exercises

- Hand-ins this week:
  - Must put yourself into a group, maybe 1-person
  - Your hand-in will automatically count for the group
  - Sorry about last week's hand-in mess!

- 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 are never shared btw 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 repair the code?

Thread-safe server counting calls

```
class CountingFactorizer implements Factorizer {
  private final AtomicLong count = new AtomicLong(0);
  public long[] getFactors(long p) {
    long[] factors = PrimeFactors.compute(p);
    count.incrementAndGet();
    return factors;
  }
  public long getCount() { return count.get(); }
}
```

Like Goetz p. 23

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

Bad attempt to cache last factorization

```
class TooSynchronizedCachingFactorizer implements Factorizer
  private long lastNumber = 1;
  private long[] lastFactors = new long[0];
  // 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

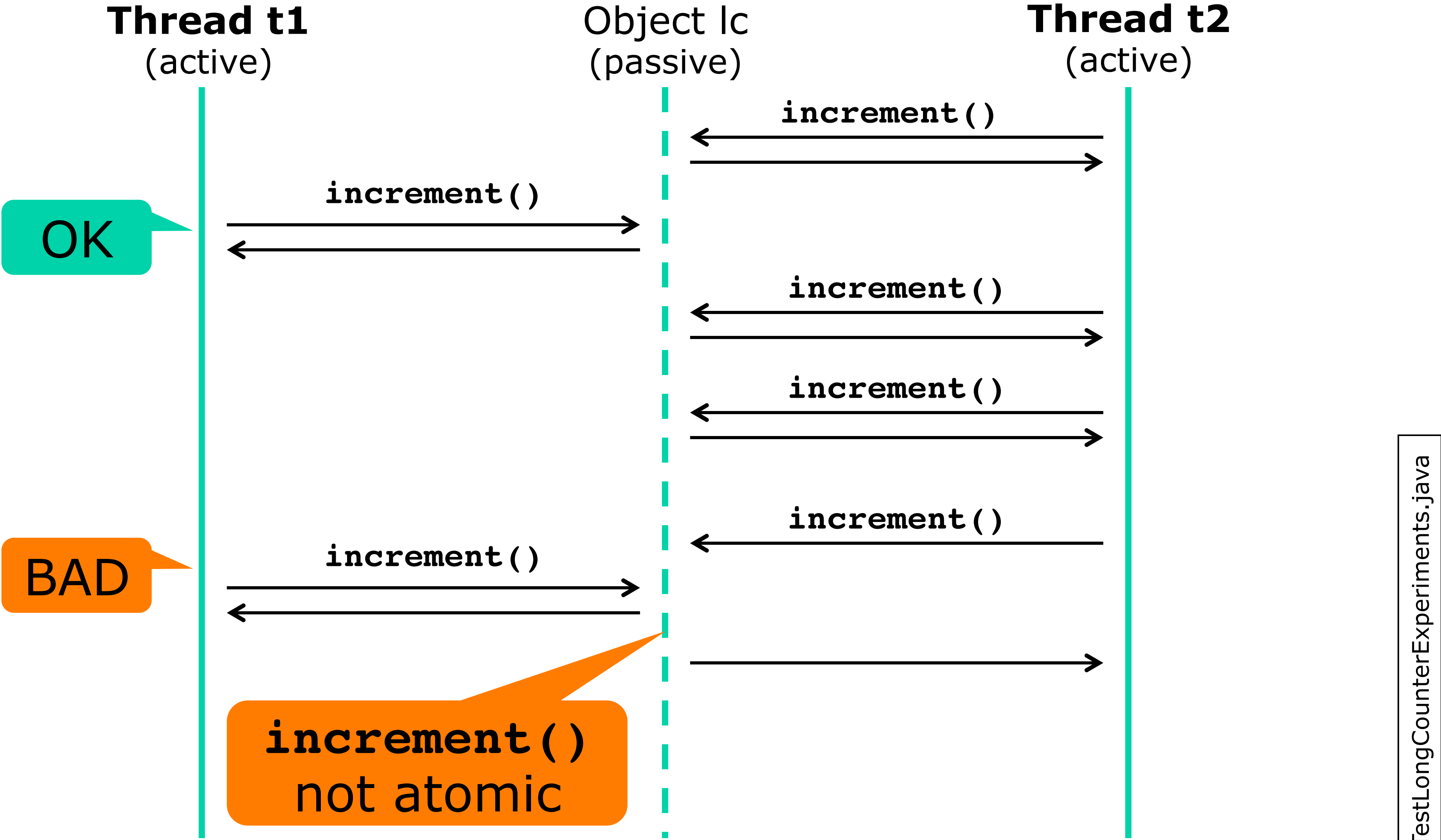
- We want to *atomically* update **`lastNumber`** and **`lastFactors`**

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 = 0;
  private long[] lastFactors = new long[0];
  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

- Correct but subtle

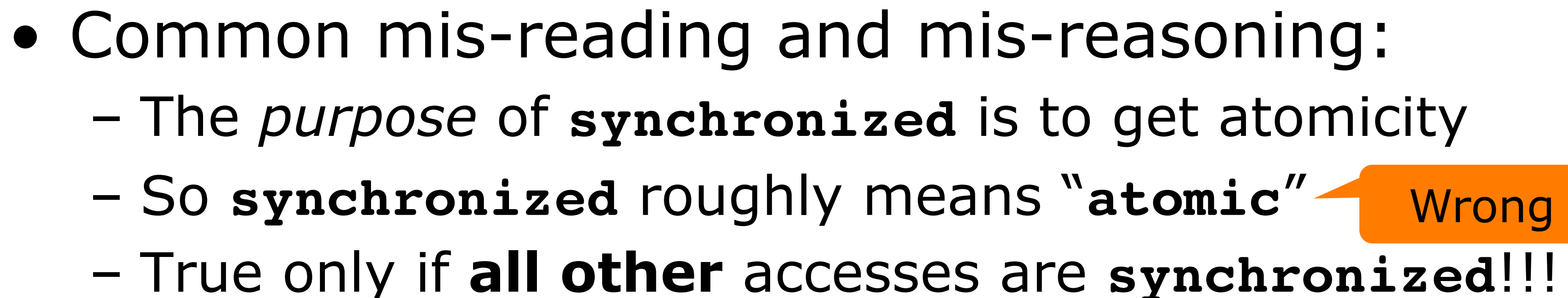
Using locks for atomicity

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`!!!



Wrapping 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();
  }
}
```

NB!

The fields cannot change between test and return

Q: Why?

Like Goetz p. 49

- Immutable, so automatically thread-safe

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

NB!

```
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 assignment

Like Goetz p. 50

- Only one mutable field, atomic assignment
- Easy to implement, easy to see it is correct
- Drawback: cost of creating cache objects
  - Not a problem with modern garbage collectors

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
  - Better make fields **`final`** and remove the setters

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-

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-

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

Josh Bloch designed the Java collection classes

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

Bloch p. 73

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 OneValueCache 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 volatile or locking
  - Seems a little dangerous?

Avoiding shared mutable state

- Avoiding sharing between threads:
  - Stack confinement: Local variables are never shared between threads
  - Thread confinement via ThreadLocal objects
  - Ad hoc thread confinement: Swing GUI components are accessed only by the GUI thread
- Avoiding mutable state:
  - Make fields **`final`** as far as possible
  - Replace multiple mutable fields by a single mutable reference to an immutable object

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 some other part of the program
- This is a problem in sequential code too, but much worse in concurrent code
  - Minimize Mutability!

Java monitor pattern

An object following the *Java monitor pattern* encapsulates all its mutable state (in **`private`** fields) and guards it with the object's own intrinsic lock (**`synchronized`**).

Goetz p. 60

- Monitors invented 1974 by Hansen and Hoare
  - A way to encapsulate mutable state in concurrency
- Java monitor pattern implements monitors
  - If you use care and discipline!
  - Per Brinch Hansen critical of Java, 1999 paper
- Modern (Java) data structures are subtler ...
  - Illustrated by Goetz VehicleTracker example

A class of mutable points

- MutablePoint, like java.awt.Point

Design mistake

```
class MutablePoint {
  public int x, y;
  public MutablePoint() {
    x = 0; y = 0;
  }
  public MutablePoint(MutablePoint p) {
    this.x = p.x; this.y = p.y;
  }
}
```

Not thread-safe

Copy constructor

TestVehicleTracker.java

Goetz p. 64

- Q: Why not thread-safe?

Vehicle tracker as a monitor class

V1

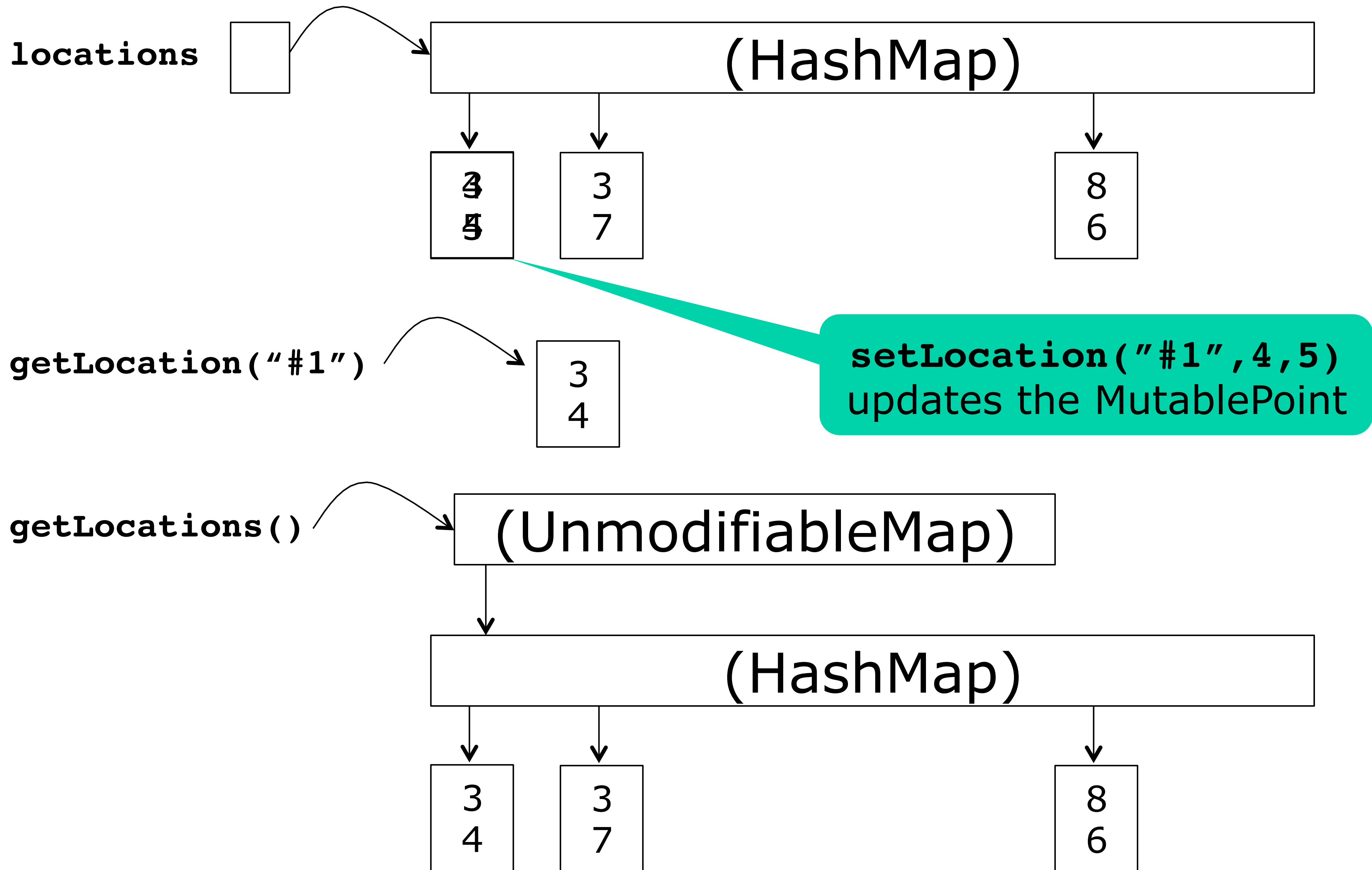
Goetz p. 63

TestVehicleTracker.java

```
class MonitorVehicleTracker {
  private final Map<String, MutablePoint> locations;
  public MonitorVehicleTracker(Map<String, MutablePoint> locations) {
    this.locations = deepCopy(locations);
  }
  public synchronized Map<String, MutablePoint> getLocations() {
    return deepCopy(locations);
  }
  public synchronized MutablePoint getLocation(String id) {
    MutablePoint loc = locations.get(id);
    return loc == null ? null : new MutablePoint(loc);
  }
  public synchronized void setLocation(String id, int x, int y) {
    MutablePoint loc = locations.get(id);
    loc.x = x;
    loc.y = y;
  }
  private static Map<String, MutablePoint> deepCopy(Map<String, MutablePoint> m) {
    Map<String, MutablePoint> result = new HashMap<String, MutablePoint>();
    for (String id : m.keySet())
      result.put(id, new MutablePoint(m.get(id)));
    return Collections.unmodifiableMap(result);
  }
}
```

- Protects its state in field locations
- But why all that copying?

MonitorVehiclerTracker memory



A class of immutable points

- Immutable Point class:

```
class Point {
  public final int x, y;
  public Point(int x, int y) {
    this.x = x; this.y = y;
  }
}
```

TestVehicleTracker.java

Goetz p. 64

- Automatically thread-safe

Thread safety by delegation and immutable points

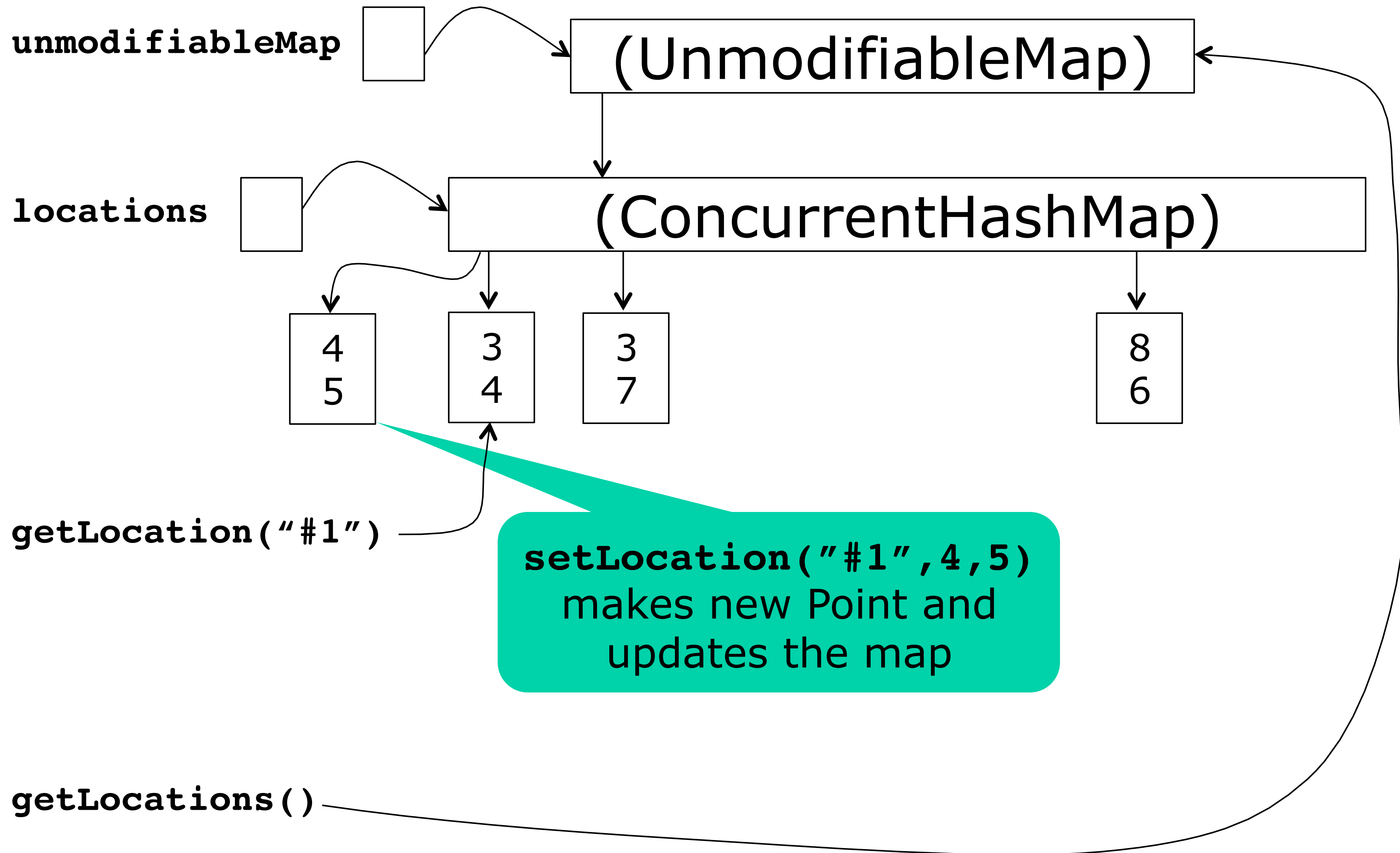
V2

Goetz p. 65

```
class DelegatingVehicleTracker {
  private final ConcurrentMap<String, Point> locations;
  private final Map<String, Point> unmodifiableMap;
  public DelegatingVehicleTracker(Map<String, Point> points) {
    locations = new ConcurrentHashMap<String, Point>(points);
    unmodifiableMap = Collections.unmodifiableMap(locations);
  }
  public Map<String, Point> getLocations() {
    return unmodifiableMap;
  }
  public Point getLocation(String id) {
    return locations.get(id);
  }
  public void setLocation(String id, int x, int y) {
    locations.replace(id, new Point(x, y));
  }
}
```

- No defensive copying any longer
  - Less mutability can give *better* performance!
- Q: Why not just cast **`locations`** to an interface without setters?

DelegatingVehicleTracker memory



V2A

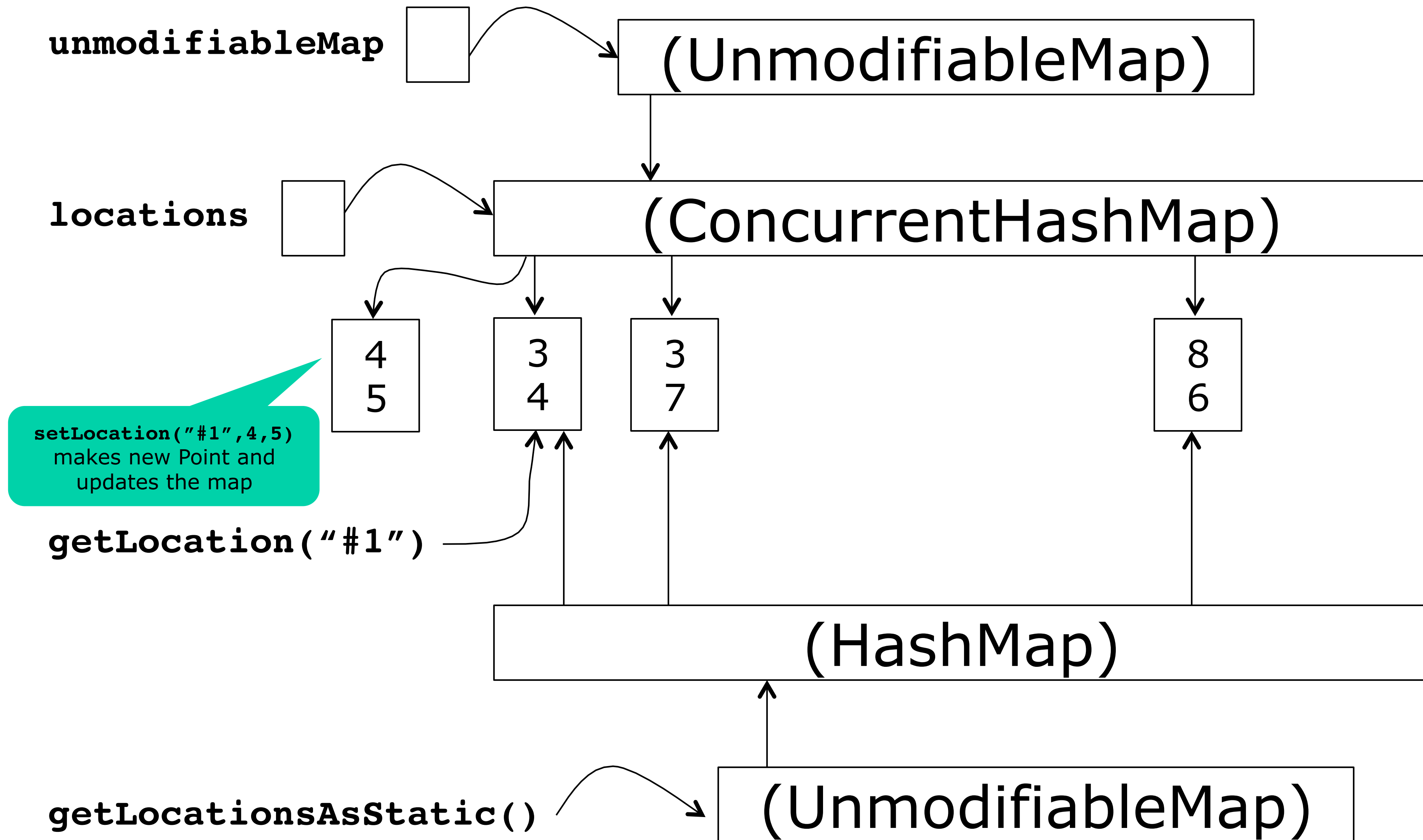
Alternative getLocations()

- Returns unmodifiable view
  - of snapshot copy of hashmap,
  - referring to the existing immutable points

Goetz p. 66

```
public Map<String, Point> getLocationsAsSnapshot() {
  return Collections.unmodifiableMap(new HashMap<String, Point>(locations));
}
```

DelegatingVehicleTracker memory with static getLocations result



Immutability is good

- Can simplify thread-safety
- Can speed up some operations
- Microsoft .NET has new immutable collections
  - http://msdn.microsoft.com/en-us/library/dn385366%28v=vs.110%29.aspx
  - http://blogs.msdn.com/b/bclteam/archive/2012/12/18/preview-of-immutable-collections-released-on-nuget.aspx
- Different from unmodifiable collections
  - No underlying modifiable collection
  - Enumeration is safe, including thread-safe
- Java 8 does not have immutable collections

Safe mutable point class

- Mutable point as monitor

```
public class SafePoint {
  private int x, y;
  private SafePoint(int[] a) { this(a[0], a[1]); }
  public SafePoint(SafePoint p) { this(p.get()); }
  public SafePoint(int x, int y) { this.set(x, y); }
  public synchronized int[] get() {
    return new int[]{x, y};
  }
  public synchronized void set(int x, int y) {
    this.x = x; this.y = y;
  }
}
```

Goetz p. 69

V3

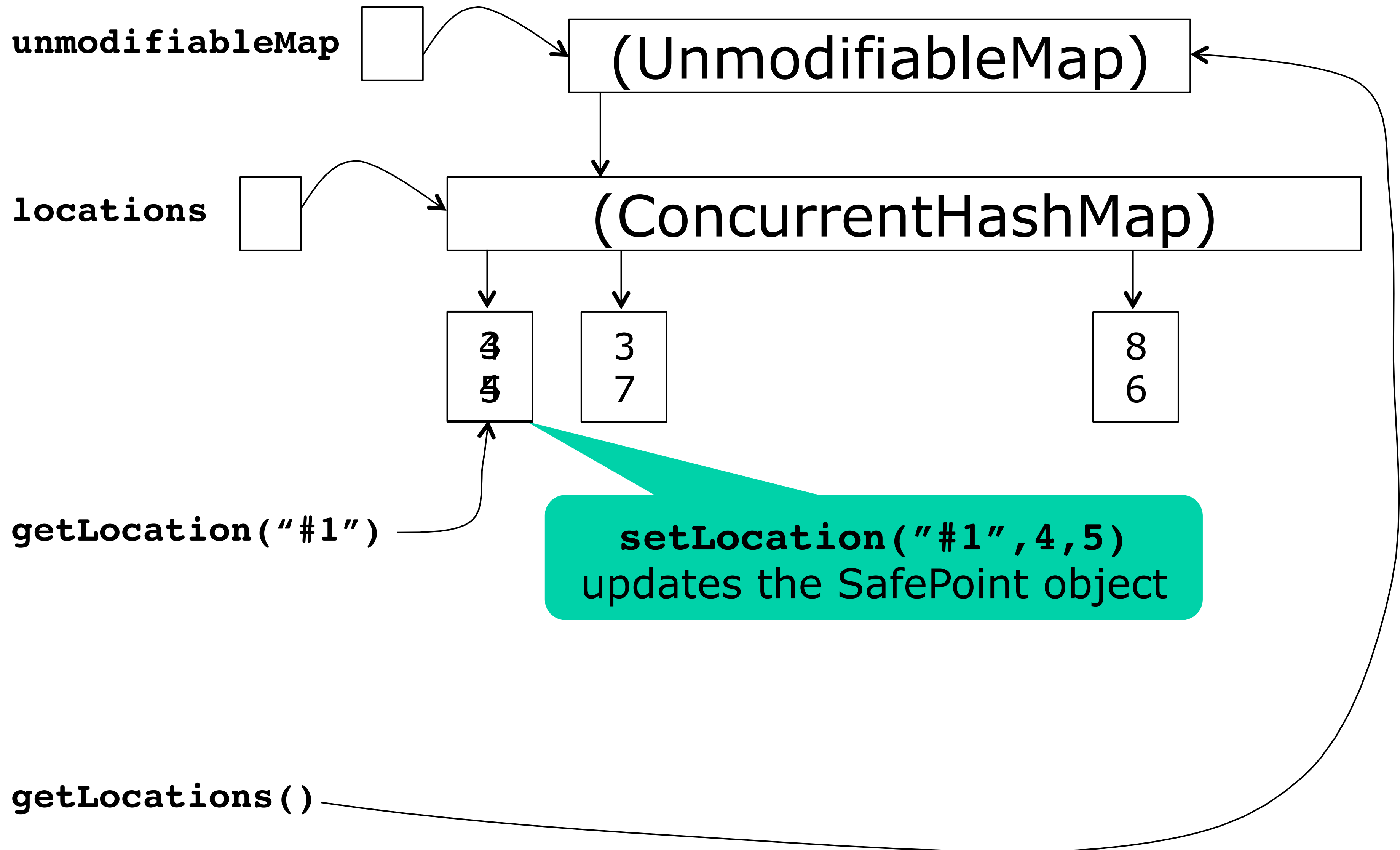
Safe publishing vehicle tracker

Goetz p. 70

```
public class PublishingVehicleTracker {
  private final Map<String, SafePoint> locations;
  private final Map<String, SafePoint> unmodifiableMap;

  public PublishingVehicleTracker(Map<String, SafePoint> locations) {
    this.locations
      = new ConcurrentHashMap<String, SafePoint>(locations);
    this.unmodifiableMap = Collections.unmodifiableMap(this.locations);
  }
  public Map<String, SafePoint> getLocations() {
    return unmodifiableMap;
  }
  public SafePoint getLocation(String id) {
    return locations.get(id);
  }
  public void setLocation(String id, int x, int y) {
    locations.get(id).set(x, y);
  }
}
```

SafePublishingVehicleTracker memory



Which VehicleTracker is best?

- All are thread-safe
  - Some due to defensive copying
  - Some due to immutability and unmodifiability
- Different meanings of setLocation:
  - setLocation **does not** affect prior getLocation/s:
    - MonitorVehicleTracker (V1)
    - DelegatingVehicleTracker with getLocationsStatic (V2A)
  - setLocation **does** affect prior getLocation/s:
    - DelegatingVehicleTracker (V2)
    - SafePublishingVehicleTracker (V3)
- Performance depends on the usage
  - Eg. more setLocation calls than getLocations calls
  - Number of results returned by getLocations

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>());
```

Adding putIfAbsent to ArrayList<T>

Not thread-safe

TestListHelper.java

```
class FirstBadListHelper<E> {
  private final List<E> list
    = Collections.synchronizedList(new ArrayList<E>());
  public boolean putIfAbsent(E x) {
    boolean absent = !list.contains(x);
    if (absent)
      list.add(x);
    return absent;
  }
}
```

test, then ...

... act

- Non-atomic test-then-act is not thread-safe
- But this is not thread-safe either. Q: Why?

Not thread-safe

Like Goetz p. 72

```
class SecondBadListHelper<E> {
  public final List<E> list = Collections.synchronizedList(new Array...);
  public synchronized boolean putIfAbsent(E x) {
    boolean absent = !list.contains(x);
    if (absent)
      list.add(x);
    return absent;
  }
}
```

Client side locking for putIfAbsent

```
class GoodListHelper<E> {
  public List<E> list = Collections.synchronizedList(new ArrayList<E>());

  public boolean putIfAbsent(E x) {
    synchronized (list) {
      boolean absent = !list.contains(x);
      if (absent)
        list.add(x);
      return absent;
    }
  }
}
```

Lock on **list** not **this**

Intention: Atomic test-then-act

Goetz p. 72

- Discuss:
  - Is the test-then-act guaranteed atomic?
  - What could undermine the atomicity?

Using composition is safer – and more work

```
final class BetterArrayList<E> implements List<E> {
  private List<E> list = new ArrayList<E>();

  public synchronized boolean putIfAbsent(E x) {
    boolean absent = !list.contains(x);
    if (absent)
      list.add(x);
    return absent;
  }

  public synchronized boolean add(E item) {
    return list.add(item);
  }

  ... approx. 30 other ArrayList<E> methods with synchronized added ...
}
```

TestListHelper.java

- Q: Are operations now guaranteed atomic?
- Better use java.util.concurrent.* collections
  - If you need to make updates concurrently

ConcurrentModificationException

```
ArrayList<String> universities = new ArrayList<String>();
universities.add("Copenhagen University");
universities.add("KVL");
universities.add("Aarhus University");
universities.add("IT University");
for (String name : universities) {
  System.out.println(name);
  if (name.equals("KVL"))
    universities.remove(name);
}
```

TestConcurrentmodification.java

Should not change the collection while iterating

Even when no thread concurrency

```
Copenhagen University
KVL
Exception ... java.util.ConcurrentModificationException
```

- The “fail-early” mechanism is not thread-safe!
- Do not rely on it in a concurrent context
  - ... instead ...

Java 8 documentation on iteration

- Collections.synchronizedList() says:

It is imperative that the user manually synchronize on the returned collection when traversing it via `Iterator`, `Spliterator` or `Stream`:

```
Collection c = Collections.synchronizedCollection(myCollection);
   ...
synchronized (c) {
    Iterator i = c.iterator(); // Must be in the synchronized block
    while (i.hasNext())
       foo(i.next());
}
```

Failure to follow this advice may result in non-deterministic behavior.

```
Collection c = Collections.synchronizedCollection(myCollection);
synchronized (c) {
  for (T item : c)
    foo(item);
}
```

Same as above code: **for** creates an Iterator

- All access to myCollection must be through c

Collections in a concurrent context

- Preferably use a modern concurrent collection class from java.util.concurrent.*
  - Iterators and **`for`** are *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

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>

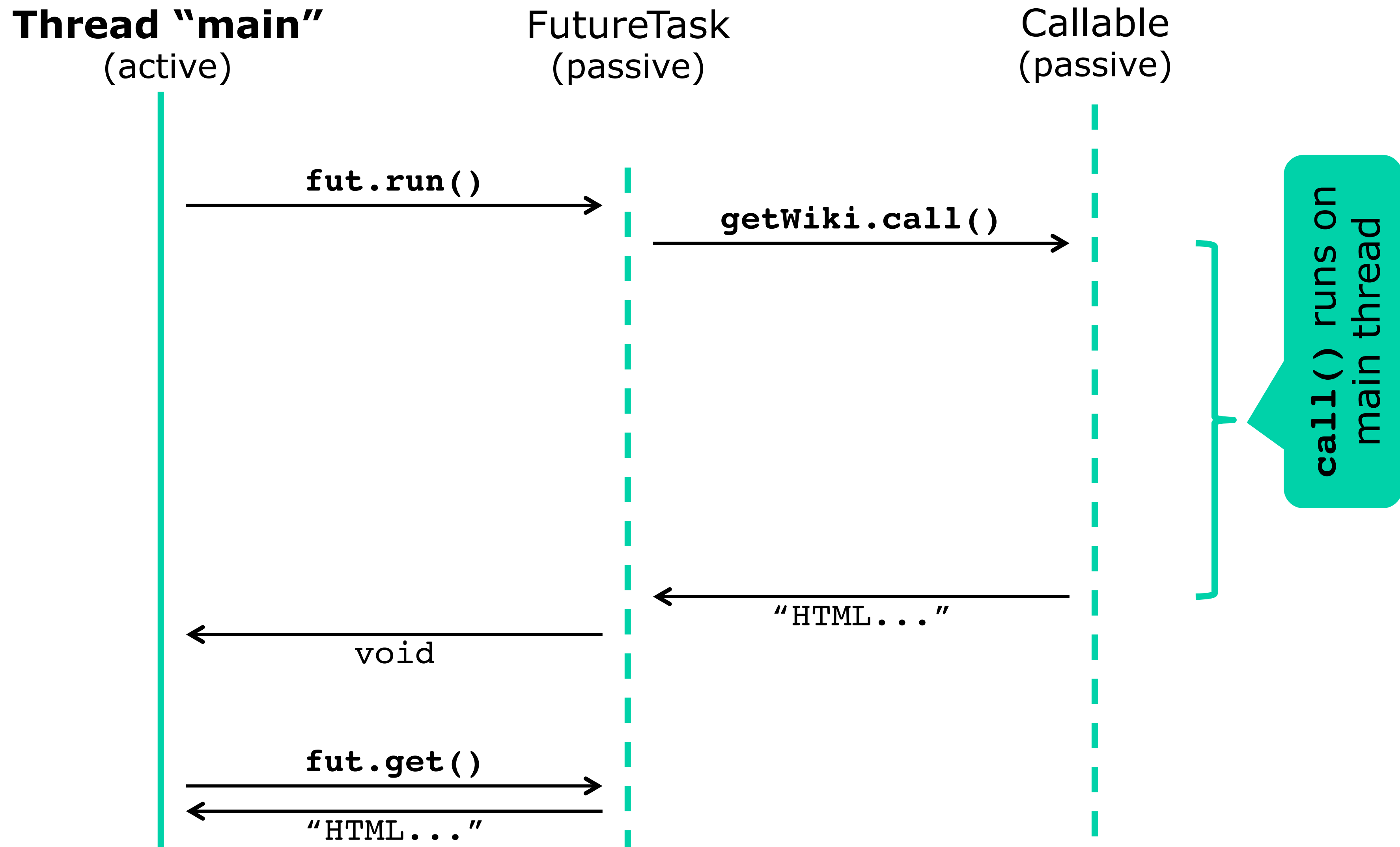
```
Callable<String> getWiki = new Callable<String>() {
  public String call() throws Exception {
    return getContents("http://www.wikipedia.org/", 10);
}};
FutureTask<String> fut = new FutureTask<String>(getWiki);
Thread t = new Thread(fut);
t.start();
try {
  String homepage = fut.get();
  System.out.println(homepage);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

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

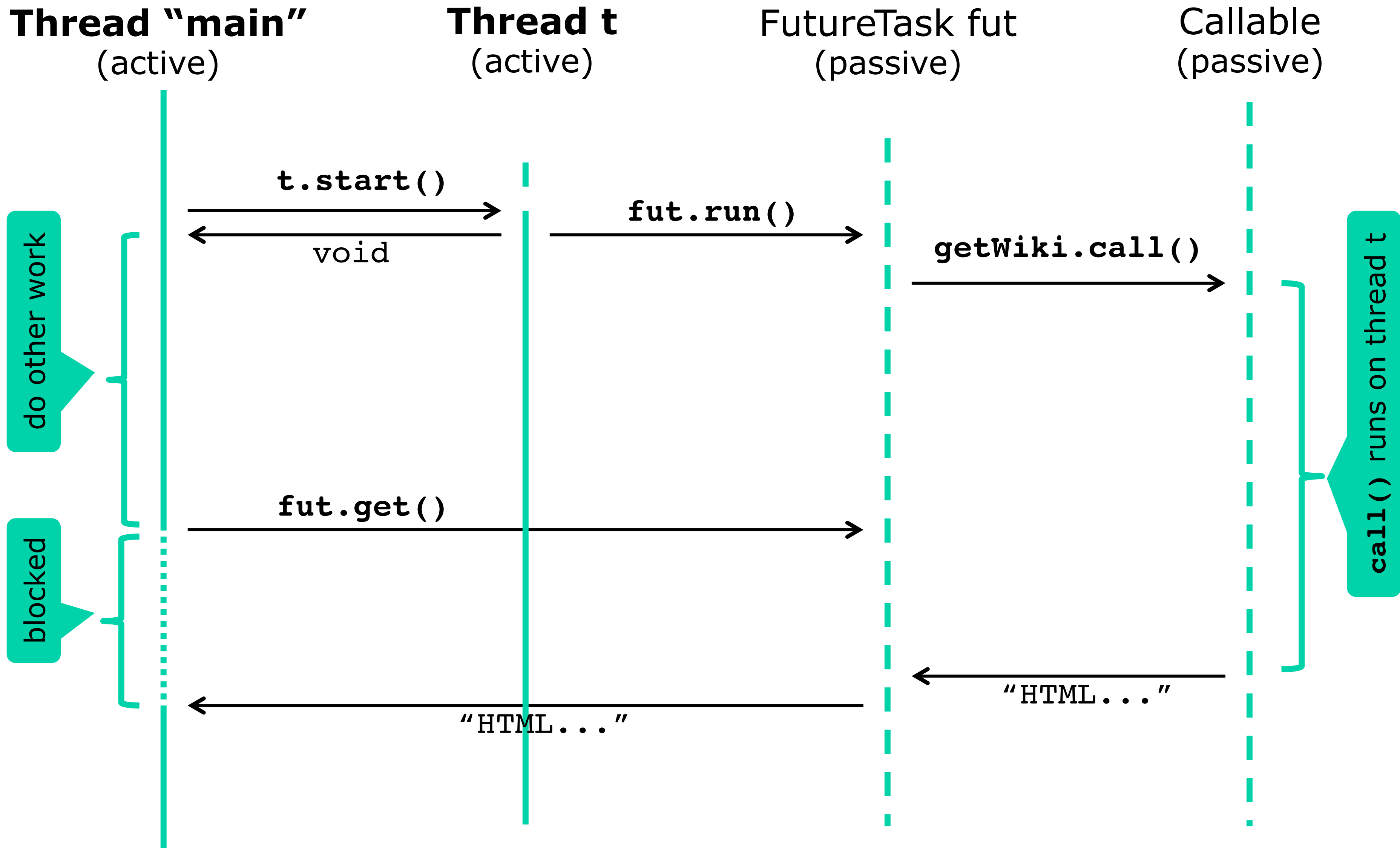
Block until **`call()`** completes

- 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 handling is simple but gross:

If **`call()`** throws **`exn`**, then **`get()`** throws ExecutionException(**`exn`**)

... and then we further wrap a RuntimeException(...) around that

```
try { String homepage = fut.get(); ... }
catch (Exception exn) { throw new RuntimeException(exn); }
```

- Goetz has a better, more complex, approach:

```
try { String homepage = fut.get(); ... }
catch (ExecutionException exn) {
  Throwable cause = exn.getCause();
  if (cause instanceof IOException)
    throw (IOException)cause;
  else
    throw launderThrowable(cause);
}
```

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

Turn others into unchecked exceptions

Like Goetz p. 97

Goetz's launderThrowable method


```
public static RuntimeException launderThrowable(Throwable t) {
  if (t instanceof RuntimeException)
    return (RuntimeException) t;
  else if (t instanceof Error)
    throw (Error) t;
  else
    throw new IllegalStateException("Not unchecked", t);
}
```

- Make a checked exception into an unchecked
  - without adding unreasonable layers of wrapping
  - cannot just **`throw cause;`** in previous slide's code
- Mostly an administrative mess
  - caused by the Java's "checked exceptions" design
  - thus not a problem in C#/.NET

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

C1

```
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);
```

- Q: Why not scalable?

Thread-safe scalable cache, using concurrent hashmap

C2

```
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

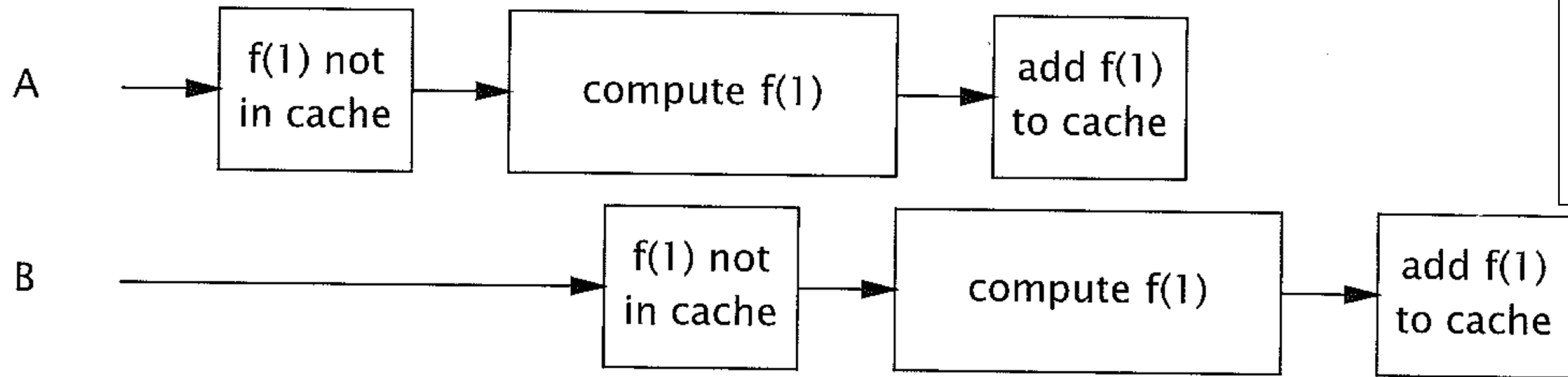


FIGURE 5.3. Two threads computing the same value when using `Memoizer2`.

- 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()`**

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

C3

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 = new Callable<V>() {
         public V call() throws InterruptedException {
           return 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) { throw launderThrowable(...); }
  }
```

If arg not in cache ...

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

... run it on calling thread

Block until completed

Memoizer3 can still duplicate work

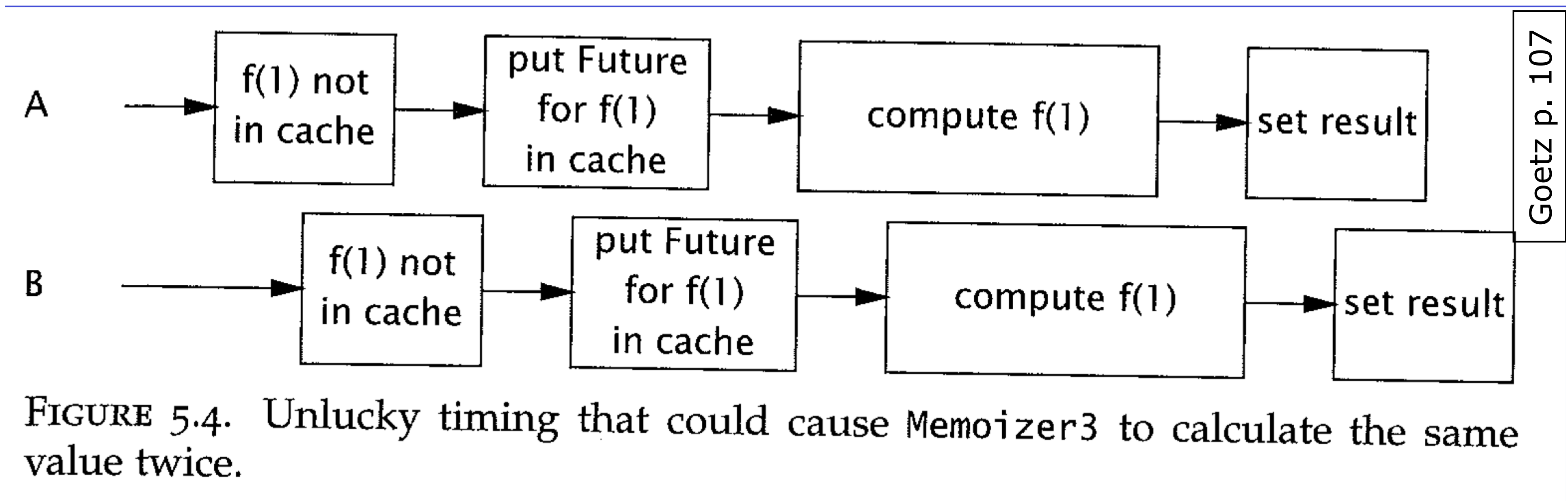


FIGURE 5.4. Unlucky timing that could cause Memoizer3 to calculate the same value twice.

- 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

C4

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 = new Callable<V>() {
         public V call() throws InterruptedException {
           return c.compute(arg);
      }};
      FutureTask<V> ft = new FutureTask<V>(eval);
      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, but looks complicated

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

C5

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, new Function<...>(){
       public Future<V> apply(final A arg) {
         Callable<V> eval = new Callable<V>() {
             public V call() throws InterruptedException {
               return c.compute(arg);
             }};
         ftr.set(new FutureTask<V>(eval));
         return ftr.get();
       }});
    if (ftr.get() != null)
      ftr.get().run();
    try { return f.get(); }
    catch (ExecutionException e) { throw launderThrowable(...); }
  }
```

make future

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

This week

- Reading
  - Goetz et al chapters 4 and 5
  - Bloch item 15
- Exercises
  - Mandatory hand-in Thursday at 23:55
  - Goals: Build a threadsafe class, use built-in collection classes, use the “future” concept
- Read before for next week’s lecture
  - *Java Precisely* 3rd ed. §11.13, 11.14, 23, 24, 25
  - Available in PDF on LearnIT