Practical Concurrent and Parallel Programming 3

Peter Sestoft
IT University of Copenhagen

Friday 2014-09-12*
Plan for today

- Java Monitor pattern
- Defensive copying, VehicleTracker
- Standard collection classes not thread-safe
- Extending collection classes
- ConcurrentModificationException
- FutureTask<T>, asynchronous execution
- (Silly complications of checked exceptions)
- Building a scalable result cache
Comments on exercises

• Exercise schedule:
  – 1000-1200: 2A14
  – 1200-1400: **2A14 ← change**!

• True:
  – If program p fails when tested, then it is not thread-safe

• **False:**
  – If program p does not fail when tested, then it is thread-safe
    
    *NEVER reason like that*
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*).

- 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
LongCounter as monitor, and documenting thread-safety

- Use the @GuardedBy annotation on fields:

```java
class LongCounter {
    @GuardedBy("this")
    private long count = 0;
    public synchronized void increment() { count++; }
    public synchronized long get() { return count; }
}
```

- Compile files with

```bash
javac -cp ~/lib/jsr305-3.0.0.jar ThreadsafeLongCounter.java
```

- Annotations show the programmer’s *intent*
  - Annotations are *not* checked by the Java compiler
  - Week 6 we see a tool for checking @GuardedBy
A class of mutable points

• MutablePoint, like java.awt.Point

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

• Q: Why not thread-safe?
Vehicle tracker as a monitor class

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

```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);
    }
}
```
MonitorVehiclerTracker memory

locations

getLocation("#1")

setLocation("#1",4,5) updates the MutablePoint

getLocations()

(UnmodifiableMap)

(HashMap)

(setLocation("#1",4,5)

(locations

(HashMap)

(getLocation("#1")

3 4

3 7

8 6

3 4

3 7

8 6

(HashMap)
A class of immutable points

• Immutable Point class:

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

• Automatically thread-safe
Thread safety by delegation and immutable points

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

unmodifiableMap

(locations)

(UnmodifiableMap)

(ConcurrentHashMap)

getLocation("#1")

setLocation("#1", 4, 5)
makes new Point and updates the map

getLocations()
Alternative getLocations()

- Returns unmodifiability view
  - of static copy of hashmap,
  - referring to the existing immutable points

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

Goetz p. 66
DelegatingVehicleTracker memory with static getLocations result

getLocation("#1")

getLocationsAsStatic()

setLocation("#1", 4, 5) makes new Point and updates the map

unmodifiableMap

(locations

ConcurrentHashMap)

(HashMap)

(UnmodifiableMap)
Immutability is GOOD

• Can speed up some operations
• Can simplify thread-safety
• Microsoft .NET has new immutable collections
• 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

```java
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
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 get_location(String id) {
        return locations.get(id);
    }

    public void setLocation(String id, int x, int y) {
        locations.get(id).set(x, y);
    }
}
SafePublishingVehicleTracker memory

unmodifiableMap

(locations)

(UnmodifiableMap)

(setLocation("#1", 4, 5)
updates the SafePoint object)

getLocations()

getLocation("#1")
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 (V3)
  – setLocation does affect prior getLocation/s:
    • DelegatingVehicleTracker (V2)
    • SafePublishingVehicleTracker (V4)

• Performance depends on the usage
  – More setLocation calls than getLocations calls
  – Number of results returned by getLocations
Plan for today

• Java Monitor pattern
• Defensive copying, VehicleTracker
• **Standard collection classes not threadsafe**
• Extending collection classes
• ConcurrentModificationException
• FutureTask<T> and asynchronous execution
• (Silly complications of checked exceptions)
• Building a scalable result cache
The classic collection classes are not threadsafe

- 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

```java
final Collection<Integer> coll = new HashSet<Integer>();
f finallyInt 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);
    }
});
```
Adding putIfAbsent to ArrayList<T>

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

```java
class FirstBadListHelper<E> {
    public 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;
    }
}
```

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

Goetz p. 72
Client side locking for putIfAbsent

```java
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;
        }
    }
}
```

• Discuss:
  – Is the test-then-act guaranteed atomic?
  – What could undermine the atomicity?
Using composition is safer – and more work

```java
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 ...
}
```

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

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

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

Exception ... java.util.ConcurrentModificationException
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, Splitter or Stream:

```java
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.

```java
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 in 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
Plan for today

- Java Monitor pattern
- Defensive copying, VehicleTracker
- Standard collection classes not thread-safe
- Extending collection classes
- ConcurrentModificationException
- `FutureTask<T>, asynchronous execution`
- (Silly complications of checked exceptions)
- Building a scalable result cache
Callable<T> versus Runnable

• A Runnable contains a `void` method:

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

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

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

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

• 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
  – Possible because it implements Runnable
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); } 

• 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

Thread “main”  
(active)  

FutureTask  
(passive)  

call() runs on main thread

Callable  
(passive)  

```java
fut.run()  

getWiki.call()  

"HTML..."
```

```java
fut.get()  

"HTML..."
```

```java
void
```
Asynchronous FutureTask

Thread “main” (active)

Thread t (active)

FutureTask fut (passive)

Callable (passive)

t.start() → fut.run() → getWiki.call()

void → fut.get() → “HTML…”

“HTML…” → call() runs on thread t

do other work

blocked
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
Plan for today

• Java Monitor pattern
• Defensive copying, VehicleTracker
• Standard collection classes not threadsafe
• Extending collection classes
• ConcurrentModificationException
• FutureTask<T>, asynchronous execution
• (Silly complications of checked exceptions)
• Building a scalable result cache
Goetz’s “scalable result cache”

- Interface representing functions from A to V

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

- Example 1: Our prime factorizer

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

- Example 2: Fetching a web page

```java
class FetchWebpage implements Computable<String, String> {
    public String compute(String url) {
        ... create Http connection, fetch webpage ...
    }
}
```
Thread-safe but non-scalable cache

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

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

- Q: Why not scalable?

Goetz p. 103
Thread-safe scalable cache, using concurrent hashmap

```java
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;
    }
}
```

• 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

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

Figure 5.3. Two threads computing the same value when using Memoizer2.
Thread-safe scalable cache using `FutureTask<V>`, v. 3

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(...); } 
    }
}
Memoizer3 can still duplicate work

- 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()`

Figure 5.4. Unlucky timing that could cause Memoizer3 to calculate the same value twice.
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(...); }
    }
}
The technique used in Memoizer

• (Before Java 8) one cannot atomically test-then-create-future-and-add-it

• Hence, suggested by Bloch item 69:
  – Make a fast (non-atomic) test for arg in cache
  – If not there, create 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)

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(...); }
    }
}
Parts of Java are 20 years old, have some design mistakes

• Never use these Thread methods (API):
  – destroy()
  – countStackFrames()
  – resume()
  – stop()
  – suspend()

• Avoid thread groups (Bloch item 73)

• Prefer non-synchronized
  – StringBuilder over StringBuffer
  – ArrayList or CopyOnWriteArrayList over Vector
  – HashMap or ConcurrentHashMap over HashTable
This week

• Reading
  – Goetz et al chapters 4 and 5

• Exercises
  – Build a threadsafe class, use built-in collection classes, use the future concept

• Read before next week’s lecture
  – Sestoft: Microbenchmarks in Java and C#
  – Optional: McKenney chapter 3