Parallel Programming Practice

Sharing Objects

Susanne Cech Previtali
Thomas Gross

Last update: 2009-10-22, 13:02
Publication

An object is *published* when

- It has been made available outside of its current scope

How?

- Store a reference where other code can access it
- Return a reference from a non-private method
- Pass a reference to a method in another class
Properties

‣ Follow chain of references

‣ “Alien” method calls of a class C with object as argument

‣ Methods in other classes

‣ Overridable methods of C

Any method which is not private, static, or final can be overridden.

```java
@ThreadSafe
public class CachedFactorizer implements Servlet {
    @GuardedBy("this") private BigInteger lastNumber;
    @GuardedBy("this") private BigInteger[] lastFactors;

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = null;

        synchronized (this) {
            if (i.equals(lastNumber)) factors = lastFactors.clone();
        }

        if (factors == null) {
            factors = factor(i);
            synchronized (this) {
                lastNumber = i;
                lastFactors = factors.clone();
            }
        }

        encodeIntoResponse(resp, factors);
    }
}
```

Object graph of `CachedFactorizer`

- `this`
- `CachedFactorizer`
  - `lastNumber`
  - `lastFactors`
- `BigInteger`
- `BigInteger`
- `BigInteger`
- `BigInteger`
- `BigInteger`
- `BigInteger`

`BigInteger` is immutable

returned reference
Problems with escaped objects

An object is *escaped* when

- It is published and should not have been published

Consequences

- Any caller can modify object
Proper construction

Object is *not* properly constructed if *this* escapes during construction

- Consistent state only after constructor returns

Do not

- Start a thread in the constructor
- Call a overridable method in the constructor
How to prevent escape

Thread confinement
Immutability
Safe publication
Thread confinement
Thread confinement

Avoid escaping of objects by *not* sharing

Thread confinement
- A single thread accesses data $\Rightarrow$ thread safe

Kinds
- Ad-hoc thread confinement
- Stack confinement
- `ThreadLocal`
1 Ad-hoc thread confinement

Implementation is responsible

- Fragile

Special case: volatile variables

- Ensure that only one thread writes the volatile variable
- Remember visibility guarantees of volatile writes
2 Stack confinement

Object is reachable only through local variables

- Local variables exist only on stack
- Stack accessible only to current thread

Enforcement

- Obvious for primitive types (no reference)
- References: Programmer must take care and not publish reference
public int loadTheArk(Collection<Animal> candidates) {
    SortedSet<Animal> animals =
        new TreeSet<Animal>(new SpeciesGenderComparator);
    animals.addAll(candidates);
    int numPairs = 0;
    Animal candidate = null;
    for (Animal a : animals) {
        if (candidate == null || !candidate.isPotentialMate(a))
            candidate = a;
        else {
            ark.load(new AnimalPair(candidate, a));
            numPairs++;
            candidate = null;
        }
    }
    return numPairs;
}
Animals.loadTheArk(Collection<Animal> candidates)

final not final

Animals
- ark
- species
- gender

Ark
- loadedAnimals

HashSet

AnimalPair
- one
- two

HashSet

AnimalPair
- one
- two

Animals
- candidates
- animals
- numPairs
- candidate
- a

Animal
- sp: FROG
- g: MALE

Animal
- sp: ELEPHANT
- g: MALE

Animal
- sp: FROG
- g: MALE
3 ThreadLocal

Associate a per-thread value with an object

- Separate copy of a value for each thread
- Conceptual: `Map<Thread, T>`

Examples

- Mutable singletons, global variables
## ThreadLocal API

```java
java.lang.ThreadLocal<T>
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T get()</td>
<td>Value of the current thread’s copy. If <code>value == null</code>: return <code>initialValue()</code></td>
</tr>
<tr>
<td>T initialValue()</td>
<td>Typically overridden (default: <code>return null;</code>)</td>
</tr>
<tr>
<td>void remove()</td>
<td>Remove value of copy of current thread.</td>
</tr>
<tr>
<td>void set(T value)</td>
<td>Set copy of current thread to <code>value</code>.</td>
</tr>
</tbody>
</table>
Corrected ThreadLocal example

```java
public class UniqueThreadIdGenerator {
    private static final AtomicInteger uniqueId =
        new AtomicInteger(0);

    private static final ThreadLocal<Integer> uniqueNum =
        new ThreadLocal<Integer>() {
            protected Integer initialValue() {
                return uniqueId.getAndIncrement();
            }
        };

    public static int getCurrentThreadId() {
        return uniqueNum.get();
    }
}

See also: http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6475885
```
Immutability
Immutability

An object is immutable if

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

Immutable objects are *always* thread-safe

- No synchronization needed
Attention 1

Immutability ≠ declare all fields final

- Final fields can hold references to mutable objects
- An object with final fields can still be mutable
Attention 2

Reference is immutable ≠ object is immutable

```java
class B {
    C c;
}

class C {
    final int x;
}
```

- `c` is not final
- `x` is final

```
c x: 2
```

```
x: 4
```

---

2102: Parallel Programming Practice, HS 2009  20
Immutable example

```java
@Immutable
public final class ThreeFriends {
    private final Set<String> friends = new HashSet<String>();

    public ThreeFriends() {
        friends.add("Moe");
        friends.add("Larry");
        friends.add("Curly");
    }

    public boolean isFriend(String name) {
        return friends.contains(name);
    }
}
```

*Set is mutable but ThreeFriends is designed not to be mutable*

*Update state with replacing old object with a new one*
Definition of immutability revisited

An object is immutable if

- all public fields are final,
- all public final reference fields refer to other immutable objects, and
- constructors and methods do not publish references to any internal state which is potentially mutable by the implementation.
Weak atomicity for immutable objects

```java
@ThreadSafe
public class VolatileCachedFactorizer implements Servlet {
    private volatile OneValueCache cache =
        new OneValueCache(null, null);
    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = cache.getFactors(i);
        if (factors == null) {
            factors = factor(i);
            cache = new OneValueCache(i, factors);
        }
        encodeIntoResponse(resp, factors);
    }
}
```
Immutable holder class for atomic data

```java
@Immutable
public class OneValueCache {
    private final BigInteger lastNumber;
    private final BigInteger[] lastFactors;
    public OneValueCache(BigInteger i, BigInteger[] factors) {
        lastNumber = i;
        lastFactors = Arrays.copyOf(factors, factors.length);
    }
    public BigInteger[] getFactors(BigInteger i) {
        if (lastNumber == null || !lastNumber.equals(i))
            return null;
        else
            return Arrays.copyOf(lastFactors, lastFactors.length);
    }
}
```
Publishing immutable objects

Immutable objects can be used without synchronization

But

- When final fields refer to mutable objects, synchronization must be used to access those objects
JMM: Initialization safety

Properly constructed *immutable* objects can be shared across threads without synchronization.

All threads will see correct values set in the constructor of:

- Final fields and any variables reachable through a final field
- If the object was properly constructed object

For objects with final fields, no reordering of:

- Writes in the constructor to final fields
- Writes to variables reachable through these final fields
- With initial load of a reference of a reference to that object

⇒ Values become “frozen” when constructor completes
Initialization safety for immutable objects

@ThreadSafe
public class SafeStates {
    private final Map<String, String> states;

    public SafeStates() {
        states = new HashMap<String, String>();
        states.put("alaska", "AK");
        states.put("alabama", "AL");
        /*...
        states.put("wyoming", "WY");
    }

    public String getAbbreviation(String s) {
        return states.get(s);
    }
}

String is immutable
values that are reachable through final fields at the time the constructor finishes
Safe publication
Unsafe publication

```java
@NotThreadSafe
public class UnsafeLazyInitialization {
    private static Resource resource;

    public static Resource getInstance() {
        if (resource == null)
            resource = new Resource(); // unsafe publication
        return resource;
    }
}
```

Other threads might see

- Stale value for holder (null or older value)
- Up-to-date value for holder, but stale values for the state of holder
Safe publication

Objects that are not immutable must be safely published

- Synchronization of both the publishing and consuming thread

Establish a happens-before ordering between publishing and consuming thread

- To ensure visibility

Synchronization is required if the object can be modified after publication
Safe publication patterns

Reference and state of the object must be made visible at the same time

Consider a properly constructed object

- Initialize the reference with a static initializer
- Store the reference into a volatile field or AtomicReference
- Store the reference into a final field of a properly constructed object
- Store the reference into a field that is properly guarded by a lock
Eager safe initialization

```java
@ThreadSafe
public class SafeEagerInitialization {
    private static Resource resource = new Resource();

    public static Resource getInstance() {
        return resource;
    }
}
```

Static initializers

- Run after class loading but before class is used by any threads
- Writes are visible to all threads automatically

Consider also factory implementation
Safe lazy initialization

```java
@ThreadSafe
public class SafeLazyInitialization {
    private static Resource resource;

    public synchronized static Resource getInstance() {
        if (resource == null)
            resource = new Resource();
        return resource;
    }
}
```
Double-checked locking

```java
@NotThreadSafe
public class DoubleCheckedLocking {
    private static Resource resource;
    
    public static Resource getInstance() {
        if (resource == null) {
            synchronized (DoubleCheckedLocking.class) {
                if (resource == null) {
                    resource = new Resource();
                }
            }
        }
        return resource;
    }
}
```
Corrected double-checked locking

```java
@ThreadSafe
class DoubleCheckedLocking {
    private volatile static Resource resource;

    public static Resource getInstance() {
        if (resource == null) {
            synchronized (DoubleCheckedLocking.class) {
                if (resource == null)
                    resource = new Resource();
            }
        }
        return resource;
    }
}
```

Better: SafeLazyInitialization with factory *(see Slide 33)*
Publishing and sharing

Immutable objects

- Can be published through any mechanism
- Shared without synchronization

Effectively immutable objects

- == mutable objects that are not modified (e.g. Date)
- Must be safely published
- Shared without synchronization

Mutable objects

- Must be safely published and
- Must be either thread-safe or guarded by a lock
Document accessibility of objects

Thread-confined

- Owed by and confined to thread
- Can be modified only by owning thread

Shared read-only

- Cannot be modified
- Access without synchronization

Shared thread-safe

- Internal synchronization
- Threads can use without synchronization using public interface

Guarded

- Accessed only with specific lock
## Package net.jcip.annotations

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Target</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@ThreadSafe</td>
<td>Class</td>
<td>No synchronization needed by client. No interleaving of accesses puts object in invalid state.</td>
</tr>
<tr>
<td>@NotThreadSafe</td>
<td>Class</td>
<td></td>
</tr>
<tr>
<td>@Immutable</td>
<td>Class</td>
<td>State cannot be seen to change by callers. Implies @ThreadSafe.</td>
</tr>
<tr>
<td>@GuardedBy(&quot;lock&quot;)</td>
<td>Field, method</td>
<td>lock must be used to access field/method.</td>
</tr>
</tbody>
</table>

Study Goals