WASHINGTON STATE UNIVERSITY

World Class. Face to Face.

Lecture 6

Safely Publishing Objects

Two key points from last time

- Proper Construction – this reference does not escape (to another thread) during construction
  - Remember – an object is not fully constructed until its constructor *returns*
- Thread-confined objects are safe
  - References are only on stack or in ThreadLocal objects
  - Ad-hoc thread-confinement – only one thread accesses an object by agreement (but no language enforcement)

Immutable objects are thread-safe

- Object is immutable if
  - It's state cannot be modified after construction
    - "Cannot be modified" vs "is not modified"
  - Its data fields are declared final: cannot be changed after construction (must be *declared* final and not just treated as final)
  - It's *properly constructed*
- Volatile references + immutable objects => simple thread safety w/o locks
  - Beware inadvertant publishing of private values as in Fig. 3.6

Example: atomic update using immutable object

```
Class OVCache {
  private final BigInteger last;
  private final BigInteger[] lastFactors;
  public OVCache( BigInteger i, BigInteger[] factors) {
        last = i;
        lastFactors = Arrays.copyOf(factors…);
  }
  in using code
  private volatile OVCache cache = new OVCache(null,null);
  …
  cache = new OVCache(i, factors);
```

Listings 3.12 and 3.13

Unsafe Publication

```
public Holder holder;
public void init() { holder = new Holder(42); }
public class Holder {
   private int n;
   public Holder(int n) { this.n = n; }
   public void sanityCheck() { if (n != n) … }
}
```

Holder is properly constructed but not properly published.
sanityCheck() can see two different values for holder.n!

Safe Publication

- **Recall: publication is the action that makes an object visible outside the current scope – typically an assignment of the reference**
- **Safe publication is all about making sure that the reference and the properly initialized fields in the object that it refers to become visible at the same time**

Safe publication idioms

- Initializing an object reference in a static initializer
  - i.e., static class-level data fields
- Assigning to a volatile field or AtomicReference object
- Assigning to a final field of a properly constructed object
- Assigning to a field that is properly guarded by a lock

Safe Publication Summary

- Immutable objects can be published by any mechanism
- Effectively immutable objects (ones that in fact are not changed after construction though they do not meet the strict test) must be safely published – but then can be accessed without further synchronization
- Mutable thread-safe objects must be safely published (necessary synchronization is invoked internally by the object)
- Mutable thread-unsafe objects must be safely published then accessed using proper synchronization

Chapter 3 Summary

- Visibility of changes is the main issue
- Correct synchronization, proper construction and safe publication are three requirements for programs to have well-defined behavior
- Rules are simpler for immutable objects
- There are no concurrency concerns for thread-confined objects, mutable or immutable

Chapter 4

- 50+ years of CS and SE devoted to issues of composability: how to create solutions to big problems by combining (composing) solutions to smaller problems
  - Mechanisms – e.g. intrinsic locks; GC
  - Techniques – Invariants, pre- and post-conditions; confinement; delegation; etc. (Ch4)
  - Libraries – working code embodying the techniques for specific domains (Ch 5)
- This chapter mainly about techniques

Invariants – the fundamental technique

- What property is always true of an object when it is in a correct state?
- For sequential programming, the class invariant gives you critical information about what each public method needs to achieve
- For concurrent programming, the class invariant tells you which fields are related and therefore have to be protected by a single lock

Post-conditions and Pre-conditions

- The post-condition of a method tells you what that method is supposed to accomplish
  - (A no-op will preserve the invariant but that's not very interesting or useful!)
- The pre-condition tells you what (beyond the invariant) is supposed to be true for a method to successfully reach the post-condition
  - In sequential programming calling a method when its precondition is false is an error
  - In concurrent programming we can wait for the precondition to become true