Lecture 9
Composing Objects
“Composing Objects?”

• So far, focus has been in implementing single classes / objects
  – Thread safety
  – Publication
  – Etc.

• Systems consist of many classes and objects working in conjunction with one another

• *Object composition*: use of objects inside other objects
  – Commonplace!
  – There are issues in the context of multi-threading
Designing a Thread-Safe Class

• Recall: a correct class is thread-safe if its use by multiple threads does not invalidate the class specification

• Class specification:
  – Invariants for fields
  – Pre / post / exception conditions for methods

• To design a thread-safe class
  – Identify the state variables (aka fields) for the objects in the class
  – Determine the specification (invariants, etc.)
  – Determine synchronization policy for ensuring correctness in the presence of threads
The Java Monitor Pattern

• One commonly used synchronization policy
  – Make all fields private
  – Make all methods synchronized
• If the class is correct, then the Java Monitor Pattern ensures thread-safety
  – Used frequently in library classes, e.g. Vector
• Disadvantage: performance
  – Each method locks whole object for the duration of its execution
  – Finer-grained locking (using e.g. locks based on invariants) can lead to better performance
Instance Confinement

- If you are implementing a class from scratch you can design a synchronization policy to ensure thread-safety.
- What if you want to re-use an existing non-thread-safe class in a multi-threaded setting?
  - Specification of class might not be preserved!
  - Race conditions could result.
- Instance confinement is one solution!
  - Embedded unsafe object inside thread-safe object.
  - Thread-safe object handles all access to unsafe object.
Example: Thread-Safe Set

```java
public class SyncIntegerHashSet {

    // Invariant: an integer is in the set if and only if it was
    // added to the set.

    private final Set<Integer> mySet = new HashSet<Integer>();

    public synchronized void addInteger(Integer i) { mySet.add(i); }
    public synchronized boolean containsInteger(Integer i) {
        return mySet.contains(i);
    }
}
```

SyncIntegerHashSet is thread-safe, even though HashSet is not!

- SyncIntegerHashSet uses Java monitor pattern to ensure only one thread at a time is accessing its objects
- So only thread at a time can be accessing the HashSet
- Since mySet is private, it is confined to the SyncIntegerHashSet object that owns it
Instance Confinement Observations

• Confined object should not be published or allowed to escape its confining instance
• Non-thread-safe library classes contain wrapper factory methods to provide thread-safe implementations (see p. 60 in book)
Delegating Thread Safety

• Is the following class thread-safe?

    public class ZeroCounter {

        // Invariant: count records the number of 0s processed since the most recent
        // reset, or since the object was created, provided count never exceeds MAX_VALUE

        final private BoundedCounter count = new BoundedCounter(Integer.MAX_VALUE);

        public void processInt(int i) {
            if (i == 0) count.inc();
        }

        public int getCount() { return count.current(); }

        public void reset() { count.reset(); }
    }

• Yes!
  — Operations are not synchronized
  — However, they only involve operations on local variables and a thread-safe BoundedCounter object

• This is an example of delegation of thread-safety: ZeroCounter delegates its safety obligations to BoundedCounter, which is thread-safe.
When Does Delegation Work?

• Delegation ensures thread-safety if
  – Instance fields are thread-safe
  – No invariant constrains multiple instance fields
  – No method makes more than one call to any method in an object referenced by an instance field
  – Instance fields are not published

• Otherwise, other synchronization mechanisms must be used