CMSC 433 – Programming Language Technologies and Paradigms

Sharing Objects

Administrivia

- Are you getting your money's worth?
  - Are you reviewing the slides?
  - Are you experimenting with concurrency?
    - Have you downloaded the source code examples from class repo and from JCIP?
    - Have you run some concurrent programs?
    - Have you written some concurrent programs?

Volatile Fields

- Use volatile variables only when:
  - Writes don't depend on current value, or you can ensure that only a single thread ever updates the value
  - The variable does not participate in invariants with other state variables
  - Locking is not needed while the variable is being accessed

Publication & Escape

- Publishing an object means making it available outside its current context
  - e.g., storing it externally, returning, passing it as a parameter
- If you publish an object when you shouldn't that the object has *escaped*
- If Object *obj* escapes, everything reachable from *obj* has escaped too

Example

```
class Secrets {
    public static Set<Secret> knownSecrets;

public void initialize() {
        knownSecrets = new HashSet<Secret>();
    }
}
```

- *knownSecrets* is public, so all entries have escaped

ListenerClass

```
public class ListenerClass {
    public ListenerClass(EventSource source) {
        source.registerListener(new EventListener() {
            public void onEvent(Event e) {
                doSomething(e);
            }
        });
    }
…
    interface EventListener {
        void onEvent(Event e);
    }
}
```

- *this* reference escapes with inner class *EventListener*

Safe Construction Practices

- Don’t let the **this** reference escape during object construction
- Avoid creating and then starting a thread in a constructor

this Reference Escapes

```
public class RunnableExample {
   class RunnableThread implements Runnable {
      Thread runner;
       public RunnableThread() {
          runner = new Thread(this);
          runner.start();
       }
    }

    public static void main(String[] args) {
      RunnableThread thread3 = new RunnableThread();
    }
}
```

Better Listener

```
public class SafeListener {
    private final EventListener listener;

    private SafeListener() {
        listener = new EventListener() {
            public void onEvent(Event e) {
                doSomething(e);
            }
        };
    }

    public static SafeListener newInstance(EventSource source) {
        SafeListener safe = new SafeListener();
        source.registerListener(safe.listener);
        return safe;
    }
```

Thread Confinement

- If an object is only accessed in a single thread, you don't have to synchronize access to it
- Some designs partition activities or tasks into individual threads
  - Ad-hoc Confinement
  - Stack Confinement
  - ThreadLocal

Ad-hoc Thread Confinement

```
public class ThreadPerConnectionServer
                                extends LoggingServerCore {
…
  public void process(DataRecord record) {
    (new Thread(new StdOutMsgHandler(record))).start();
  }
…
}
```

Ad-hoc Thread Confinement

```
public class StdOutMsgHandler extends MsgHandler {

  public StdOutMsgHandler(DataRecord record) {
    super(record);
  }

  public void run() {
    System.out.println("Server Side: writing record:" + record);
  }
}
```

Stack Confinement

- Ad-hoc confinement can't be enforced
- Stack confinement
  - Make objects accessible only through local variables

Stack Confinement

```
public class Matches {
  // assumes entries is not modified during operation
  public List<Integer> matches (List<Integer> entries, Integer val) {
      ArrayList<Integer> matchList = new ArrayList<Integer>();
      for (Integer entry : entries) {
        if (entry == val) {
          matchList.add(entry);
        }
      }
   return matchList;
  }
}
```

ThreadLocal

- Suppose you have multiple web server objects, where each runs in its own thread, and each can serve pages from different document directories
  - Could define a docRoot field in WebServer class
- Or, define the docRoot as a variable tied to the Thread
  - Easiest way to do this is to use java.lang.ThreadLocal
    - Equivalent to adding instance variables to all Thread objects
- All methods running in the thread access docRoot
- No interference because ALL accesses to a given docRoot are within same thread

ThreadLocal

```
public class WebServerThread {
      static final ThreadLocal docRoot = new ThreadLocal();
 ...
      public WebServerThread(int port, File root) {
...
        docRoot.set(root);
      }
}
…

processRequestWithWebServer(webServer,request)
…
public class WebServer {
      public void proces(request)
       File root = (File) WebServerThread.docRoot.get();
        // service request
      }
…
```

ThreadLocal

- Use for variables that apply per-activity, not per-object
  - e.g., Timeout values, transaction IDs, current directories, default parameters
- Use as replacements for static variables
  - When different threads should use different values

Immutability

- Immutable objects are always thread safe
- An object is immutable if:
  - Its state can't be modified after construction
  - Its fields are all final
  - It is properly constructed

Final Fields

- Are initialized once, but never changed
- Allow programmers to implement thread-safe immutable objects without synchronization
- An object is considered to be completely initialized when its constructor finishes
  - A thread that can only see a reference to an object after that object has been completely initialized is guaranteed to see correctly initialized values for that object's final fields

Unsafe Publication

```
public class Holder {
    private int n;
    public Holder(int n) {
      this.n = n;
    }
    public void assertSanity()
    {
       assert (n == n);
    }
}
```

```
public class User{
  public Holder holder;

  public void initialize() {
    holder = new Holder(42);
  }
}
```

Unsafe Publication

- To another thread, the holder object could appear to be incompletely constructed
  - The reference could be stale (null)
  - The fields in the Holder object could be stale

Safe Publication Idioms

- Mutable objects must be safely published
- Requires synchronization by both publishing and using threads
- To publish safely both an object's reference and its state must be made visible to other threads at the same time

Safe Publication Idioms

- A properly constructed object can be safely published by:
  - Initializing an object reference from a static initializer
  - Storing a reference to it into a volatile field or AtomicReference
  - Storing a reference to it into a final field of a properly constructed object
  - Storing a reference to it into a field that is properly guarded by a lock

Effectively Immutable Objects

- Objects that are not immutable, but are not modified after publication can be thought of as *effectively immutable*
- Safely published effectively immutable objects can be used safely by any thread without additional synchronization

Mutable Objects

- Safe publication ensures visibility of an object at the time it's published
- All subsequent accesses must be synchronized

Safe Publishing

- Publication requirements depend on mutability status
  - Immutable – any mechanism
  - Effectively Immutable – must be safely published
  - Mutable – safely published and thread safe or all accesses guarded by a lock

Sharing Objects Safely

- Some Policies for using and sharing objects in a concurrent program
- Thread-confined
  - Object owned and used exclusively in 1 thread
- Shared read-only
  - Object can be freely shared, but can't be modified
- Shared thread safe
  - Object performs its own synchronization, so can be freely shared
- Guarded
  - Object can accessed only if a specific lock is held