Lecture 10
Sharing Objects
ThreadLocal

- Another mechanism for localizing objects in threads so that thread-safety is guaranteed
  - A ThreadLocal object can be seen as a container for other objects, e.g.
    - ThreadLocal<List<Long>> idList;
    - idList is a ThreadLocal object containing several List<Long> objects
  - Each thread accessing a ThreadLocal object is given its own copy of a contained object
    E.g. any thread accessing idList is given its own local List<Long> object by idList
- Since objects in a ThreadLocal container are local to individual threads, no need for synchronization to ensure thread-safety
- ThreadLocal objects often used when single-threaded applications with global variables are made multi-threaded
  - The global variable is made into a ThreadLocal variable
  - Each thread then has its own copy of the formerly global variable
ThreadLocal API

• Key methods for ThreadLocal<T>
  – public T get ()
    Get instance of T associated with thread executing get ()
  – public void set (T e)
    Change instance of T associated with thread executing set () to e
  – protected T initialValue ()
    Define how to compute initial value associated with a thread (called when get() invoked first time, provided set () not called previously)
  – public void remove ()
    Remove object associated with thread

• How to define initialValue? Usually via anonymous inner classes
Anonymous Inner Classes

• Used to create subclasses of a given class without using subclass declarations
• Example

```java
ThreadLocal<
  ArrayList<Long>
  > threadIds =
new ThreadLocal<
  ArrayList<Long>
  > () {
  protected ArrayList<Long> initialValue () {
    return new ArrayList<Long> ();
  }
}
```

– The `{ protected ... }` is an anonymous inner class
– It creates a subclass of `ThreadLocal<
  ArrayList<Long>
  >` in which the default `initialValue()` method is overridden
Example: ManagerThread

```java
public class ManagerThread extends Thread {

    private static ThreadLocal<ArrayList<Long>> threadIds
        = new ThreadLocal<ArrayList<Long>>() {
            protected ArrayList<Long> initialValue () { return new ArrayList<Long> (); } 
        }
    private int numWorkers;

    ManagerThread (String name, int n) { this.setName (name); numWorkers = n; }

    private void startWorker () {
        WorkerThread t = new WorkerThread ();
        ArrayList<Long> workerIds = threadIds.get();
        workerIds.add(t.getId());
        t.start ();
    }

    public void run () {
        for (int i = 0; i < numWorkers; i++) startWorker ();
        String output = getName() + " worker ids:  ";
        for (Long id : threadIds.get()) output += ( Long.toString(id) + " ");
        System.out.println (output);
    }
}
```
Immutability

• Synchronization incurs overhead
  – Locking reduces performance
  – Ensuring thread-safety makes code more complex

• How to reduce overhead?
  – Don’t share objects among threads if you don’t have to
  – Use immutable objects whenever you can!
Immutable Objects

• Why do we need synchronization? To cope with changes to object state
  – If fields in a method are modified while a method executes, the invariants in the class spec might be temporarily invalidated
  – Without synchronization these invalid values are visible to threads with access to the object
• If object’s don’t change, then there is no need to synchronize!
  – If invariant holds when object is created, then they are guaranteed to remain true
  – *Immutable objects* have this property: once they are created, their state never changes
Immutable Objects

• Typically created with fields declared as final
  – e.g.
    ```java
    final int a = 7;
    ```
  – Final fields can never have their values changed outside a constructor, and can only be assigned once inside a constructor

• True immutability requires more, though
  – Final fields may store a reference to a mutable object, e.g.
    ```java
    final MutablePoint p = new ...;
    ```
  – Even though this reference cannot change, the methods of `p` can still be used to change the state of `p`

• So an object is immutable if
  – All its fields are `final`
  – Its state can never change (i.e. no mutable subobjects are published)

• Is this sufficient?
Mutability and Visibility

- Final fields change values once!
  - When a constructor is first called, fields are allocated and given default values
  - As the constructor executes, new values are computed and assigned to fields
- If a constructor publishes `this`, then another thread might see the value of a `final` field before it has been assigned to. (In other words: data race)
Immutability and Publishing this

- **ThreadABPrinter.java**
  public class ThreadABPrinter extends Thread {
      private ImmutableAB ab;
      ThreadABPrinter (ImmutableAB ab) { ... }
      
      public void run () {
          System.out.println ("b = " + ab.getB());
          try { Thread.sleep (100); } catch ... {} 
          System.out.println ("b = " + ab.getB());
      }
  }

- **ImproperImmutableAB.java**
  public class ImproperImmutableAB implements {...{
      public final int a;
      public final int b;
      
      ImproperImmutableAB (int a, int b) {
          this.a = a;
          new ThreadABPrinter (this).start();
          try { Thread.sleep (20); } catch ... {
              this.b = b;
          }
      }
      public int getA () { return a; }
      public int getB () { return b; }
  }

- What happens if an ImproperImmutableAB is created?
  - Thread is launched in constructor
  - this is published
  - Thread sees two different values of final field b!
Immutability Redefined

• An object is *immutable* if
  – All its fields are *final*
  – Its state never changes after construction
  – It is *properly constructed*: this does not escape during construction

• If an object is immutable, then:
  – it is thread-safe
  – it may be safely accessed / published without synchronization!
Immutability and Visibility

• What guarantees visibility of assignments to final fields in immutable objects?

• Answer: the Java Memory Model
  – If an object’s fields are all final ...
  – ... then the JMM says that all writes to these fields are immediately visible, as are all memory writes that happen-before
  – This is like behavior of volatile variables!

• This property is called *initialization safety*
Safe Publication

• Thread-safe classes define objects whose methods behave correctly in the presence of threads
• What about *publication* of (thread-safe) objects?
  – During construction an object can be in an inconsistent state
  – Even if the methods behave correctly, there may still be program errors if a thread can access a partially constructed object
• Safe publication strategies are designed to ensure this cannot happen
Unsafe Publication (JCIP pp. 50 – 51)

- A simple class (thread safe!) (Holder.java)
  ```java
class Holder {
    private int n;
    public Holder (int n) { this.n = n; }

    public void assertSanity () {
      if (n != n) throw new AssertionError ("n != n !!");
    }
}
```

- What can happen in following scenario?
  - Main creates global variable h of type Holder
  - Main starts a thread t that invokes h.assertSanity()
  - Main executes h = new Holder(42);
Answer: An AssertionError Can Be Thrown!

• A (partial) event sequence highlighting the issue
  ⟨main, write, h.n, 0⟩
  ⟨t1, read, h.n. 0⟩
  ⟨main, write, h.n, 42⟩
  ⟨t1, read, h.n, 42⟩
  ASSERTION THROWN

• What is the real problem?
  – t1 can see the state of the new object for h before its construction is complete
  – There is a data race involving h.n between main and t1
Safe Publication Practices

• To avoid publication problems for mutable objects:
  – Make sure objects are properly constructed (i.e. do not let this escape during construction)
  – Make sure state is fully constructed when reference to object is published

• How can we ensure state is fully constructed? By relying on Java Memory Model!
  – Store reference in volatile variable
    This ensures that all writes pending in constructor get performed before reference is published
  – Store reference in final field of a properly constructed object
    JMM again guarantees that all writes pending in the constructor become visible when this happens
  – Store reference in a variable that is properly locked (i.e. any reads to the variable must be in a “happens-after” relationship with the write of the reference to the variable)
    This also ensures visibility of writes in constructor before object can state can be queried

• Another approach: initialize objects in a static initializer
  – Works for static objects
  – Static initializers are invoked when classes are loaded, before threads are launched, etc.
Effectively Immutable Objects

• Classes whose fields are not final, but whose state cannot change after construction
  Example: Holder.java!
• Such objects are not guaranteed safe initialization by the Java Memory Model
  To publish such objects, safe-publication practices must be followed as just described
• Once safely published, such objects are thread-safe, however