Lecture 6
Synchronization and Visibility
Atomicity

• Atomic operations are uninterruptible
  – They have either not started, or have finished: there is no “middle”
  – Procedural abstraction: permits method calls to be viewed as atomic, even though they consist of multiple operations
  – Concurrency can break procedural abstraction!

• Thread-safety: use of locking or other mechanisms to give illusion of atomicity to method calls \emph{vis à vis} a class specification
Atomicity in Java

• What is guaranteed to be atomic in Java?
  – Reads, writes of non-64-bit primitive types (ints, chars, floats, etc.)
  – Reads, writes of references (32-bit and 64-bit)

• Guarantee: if you read a non-64-bit primitive-typed variable, you will see a value that some thread actually wrote to it
  – This guarantee is sometimes called out-of-thin-air safety
64-bit Reads, Writes

• Not guaranteed to be atomic!
  
  – **E.g.** `double x = 1.0;`
  
    • `x` is a 64-bit variable
    • Java spec says a JVM can implement this as two 32-bit writes
    • If a thread reads this variable during a write operation to it, it can get 32 “stale” bits and 32 “fresh” bits (a value that no thread ever wrote)!

  – **Other data type like this:** `long`

• To safely reads or write these variables we need to use synchronization
Atomic* classes

• AtomicInteger
• AtomicBoolean
• AtomicLong
• AtomicReference<V>
• ...

• Operations get() and set(), and more
  – More later
Synchronization and Visibility

• Two aspects to an operation
  – Atomicity: does it have a “middle” that other threads can see?
  – Visibility: when is its effect perceived by other threads?

• Visibility is tricky
What Can Following Code Do?
(adapted from textbook, Listing 3.1)

public class NoVisibilityAlt {
    private static boolean ready;
    private static int number;

    private static class ReaderThread extends Thread {
        public void run () {
            while (!ready)
                Thread.yield ();
            System.out.println (number);
        }
    }

    public static void main(…) {
        new ReaderThread ().start ();
        number = 42;
        ready = true;
    }
}

• Most of the time it prints 42
• It could print 0
• It could even never terminate!
• Why?
  – Assignments to number, ready are atomic
  – However, visibility is not guaranteed
    • Java language specification lets compilers reorder statements, use caches, etc.
    • So while number = 42 is atomic, the operation’s effect may not be visible until after thread executes println!
    • In this case, previous stale value of number is what thread sees
Reordering in Java

• Java permits effects of statements to be *reordered*
  – `number = 42` could update thread-local cache
  – `ready = true` could update main memory
  – Other thread might only see main memory and not cache

• Reorderings often driven by memory hardware / firmware
  – Sequential behavior is preserved
  – Behavior of multi-threaded applications is problematic
Dealing with Visibility: volatile

- Previous example highlights visibility anomalies in Java
  - Java language spec allows (unrelated) operations to be reordered, so long as sequential consistency is preserved
    - e.g.
      new ReaderThread().start();
      ready = true;
      number = 42;
    - Assignments to number, ready can be reordered because they are unrelated
  - This can wreak havoc with threaded applications

- Some visibility problems can be fixed by declaring variables to be volatile
  - Declaring variables volatile indicates they are shared, and operations should not be reordered
    - E.g.
      private static volatile boolean ready;
      private static volatile int number;
  - Ensures that assignment to number occurs before ready is made true, and that there is no delay in thread seeing truth of ready

- Volatility does not make non-reads/writes atomic, however! It just affects visibility of atomic operations
Visibility and Locking (1/3)

- Locking also fixes visibility problems!
- Consider following fragment from synchronized BoundedCounter class:
  ```java
  public synchronized int current () {
      return value;
  }
  ...
  public synchronized void inc () {
      if (!isMaxed()) ++value;
  }
  ```

- Further suppose a class implementing threads that increment a counter:
  ```java
  public class BoundedCounterIncThread implements Runnable {
      private BoundedCounter counter;
      BoundedCounterIncThread (BoundedCounter c){ this.counter = c; }
      public void run () { counter.inc(); }
  }
  ```
Visibility and Locking (2/3)

• What is output of following?

```java
public static void main(String[] args) throws InterruptedException {
    BoundedCounter c = new BoundedCounter (2);
    Thread t1 = new Thread (new BoundedCounterIncThread (c));
    Thread t2 = new Thread (new BoundedCounterIncThread (c));
    t1.start();
    t2.start();
    t1.join();
    t2.join();
    System.out.println (c.current());
}
```
Visibility and Locking (3/3)

• Answer: 2
• Why?
  The results of the inc operations performed by one thread are visible to the other
• A general principle of Java
  – When a lock is released, operations guarded by the lock become visible to operations following the reacquisition of the same lock
  – In the previous example, the intrinsic lock of the BoundedCounter object \( c \) plays this role!
Locking and Visibility (from textbook)

Thread A

\[ y = 1 \]
\[ \text{lock M} \]
\[ x = 1 \]
\[ \text{unlock M} \]

Everything before unlock M ...

Thread B

\[ \text{lock M} \]
\[ i = x \]
\[ \text{unlock M} \]
\[ j = y \]

... is visible to everything after lock M
Another way to view volatile

This
• volatile int x;
  — x = ...; // write
  — ... = ... x ...; // read

is equivalent to this
• SyncInt x = new SyncInt ();
  — x.set(...); // write
  — ... = ... x.get() ...; // read

public class SyncInt {
    private int value; // guarded by this

    public synchronized int get() { return value; }
    public synchronized void set(int x) { value = x; }
}
Visibility in Detail

• The Java Memory Model (part of the Java Language Specification) defines precisely how visibility works
  – JCIP Chapter 16; more detail next time

• Key notions
  – Event sequences
  – “happens-before”

• Intuitively: if an event happens before another, the effect of the first event is visible to the second