SAFE OBJECT SHARING UNDER THE JVM

Topics

- Visibility
- Publication & Escape
- Thread Confinement
- Immutability (revisited) – Design Options
- Safe Publication / Sharing Objects Safely

Visibility of Reads & Writes

- No guarantee readers will see effects of writes from different threads.
- To ensure write visibility, *must* use synchronization.

```
public class OOPS {
    private static bool go = false ;
    private static int hiker = 24 ;

    private static class RT extends Thread {
        public void run( ) {
            while ( ! go )
                 Thread.yield( ) ; // means give up CPU to waiting threads
             }
             System.out.println( hiker ) ;
        }
    }

    public static void main( String[] args) {
        new RT( ).start( )  ;
        hiker = 42 ;
        go = true ;
    }
}
```

How many threads?

What gets printed?

May print 42 and exit (yay!)

May print 24 and exit (hmm)

Nothing & never exits (ouch!)

How long would you expect this program to run?

```
public class StopThread{
    private static boolean= stopRequested;

    private static class RT extends Thread {
        public void run( ) {
            int i = 0;
            while ( ! stopRequested )     // conventional way to kill a thread
                 i++;                     // don't use Thread.stop()
        }
    }
    public static void main( String[] args) {
        new RT( ).start( )  ;
        Thread.SECONDS.sleep(1);  // Thread.sleep using SECONDS units.
        stopRequested = true ;
    }
}
```

In the absence of synchronization, there is no guarantee as to when, if ever, RT will see the value of stopRequested that was made by the main thread.

Compiler “optimization”

```
public class StopThread{
    private static boolean = stopRequested;

    private static class RT extends Thread {
        public void run( ) {
           i = 0;
           if ( ! stopRequested )    // only need to read stopRequested
               while (true)          // once, since it is not being altered
                  i++;               // in this method!
        }
    }
    public static void main( String[] args) {
        new RT( ).start( )  ;
        Thread.SECONDS.sleep(1);  // Thread.sleep using SECONDS units.
        stopRequested = true ;
    }
}
```

Visibility: Stale Data

In the absence of synchronization:

- Compilers can rearrange computations as long as this is invisible to the thread executing the code.
- JIT optimizer can rearrange the emitted host processor instructions.
- Multiple processors are free to cache anything.

MORAL

Reasoning about the order in which memory operations will happen w/o proper synchronization is nearly always incorrect.

Declaring a variable volatile

```
public class StopThread{           //This works as expected!
    private static volatile boolean = stopRequested;

    private static class RT extends Thread {
        public void run( ) {
            int i = 0;
            while ( ! stopRequested )
                 i++;
        }
    }
    public static void main( String[] args) {
        new RT( ).start( )  ;
        Thread.SECONDS.sleep(1);  // Thread.sleep using SECONDS units.
        stopRequested = true ;
    }
}
```

Volatile tells the compiler/VM to disable optimizations and always read the variable from main memory.

Volatility and Locking

- Volatility only guarantees atomicity on per-variable access.
- Locking (**`synchronized`**) guarantees atomicity of a sequence of changes.
- Only use **volatile** on a variable *A* when
  - Writes to *A* do not depend on current value or
    Can guarantee only one writing thread for *A*.
  - *A* is not part of state invariant involving other variables.
  - Locking not required for any other reason when *A* is accessed.

Publication & Escape

- An object is ***published*** when made available to code outside current class's scope.
  - Putting it in a public instance or static variable.
  - Returning it from a (non-private) method.
  - Passing it as an argument to a method in another class.
  - **Caveat**: Passing object of an inner class to a method publishes the *parent* object to the method as well.
- Publishing one object may indirectly publish others.
- Publishing an object that should not have been means the object has ***escaped***.
  - From sequential systems, we know this
    - Will break encapsulation.
    - May lead to invariant violations (e.g., class's internal rules).
  - Publishing an object before fully constructed can compromise safety (adherence to its contract).

Publication: Effects of Object Escape

```
public class UnsafeStates{
   private String[] states = new String[] { “AK”, “AL”, ….};

   public String[] getStates() {
         return states:
   }
}
```

- What was supposed to be private has escaped and effectively made public.
- In a threaded application this is much more difficult to detect.

MORAL

If encapsulation is *valuable* in sequential systems, it is *essential* under concurrency.

Publication: Practice Safe Construction

DO NOT ALLOW **this** TO ESCAPE DURING CONSTRUCTION!

- Objects are in predictable state *only after constructor returns*.
- If **this** escapes during construction, threads may see inconsistent state.
- Do not pass **this** to methods in other objects in constructor.
- Do not start threads in constructor (creating them is OK).
- Do not set GUI listeners in constructor.
- Use factories

Publication: Factories Can Prevent **this** Escaping

```
public class DemoT {
  private final Thread dt ;

  private DemoT() {
    dt = new Thread() ;
  }

  public static DemoT newDemo() {
    DemoT demo = new DemoT() ;
    demo.dt.start() ;
    return demo;
  }
}

        . . .

DemoT demo_t = DemoT.newDemo() ;
```

```
public class DemoL{
  private final EvListener evl;

  private DemoL() {
    evl = new EvListener() ;
  }

  public static DemoL newDemo(EvSource es) {
    DemoL demo = new DemoL() ;
    es.setListener( demo.evl ) ;
    return demo ;
  }
}

        . . .

DemoL demo_l = DemoL.newDemo(evSource) ;
```

Thread Confinement

Data that aren't shared need not be synchronized.

- Objects accessible from only one thread are thread confined.
  - Thus they are thread safe even if they are not in and of themselves.
  - Example: Swing components - only accessed by the event thread.
  - Example: JDBC Connections.
- Thread confinement approaches:
  - **Ad hoc** - Confinement is responsibility of implementation.
  - **Stack Confinement**– Object references only available via local variables
    - **What do we have to be careful about when using this approach?**
  - **ThreadLocal** (library support)
    - Java class that maintains a table associating object references with Thread instances – eliminates sharing
    - **What code smell could thread-local variables potentially introduce?**

ThreadLocal Confinement

- ThreadLocal is for global state that is on a per-thread basis.
- Example: Singletons in sequential system duplicated on per-thread basis.
- Our example: Per thread logging to Vector of Strings.

Classic Singleton Logger

```
import java.util.Vector ;

public class Logger {
    private Vector<String> log =
        new Vector<String>() ;

    private Logger(){}

    public void logit(String message) {
        log.add(message) ;
    }

    public void dump(String prefix) {
        for ( String s : log ) {
            System.out.println(prefix +
                ": " + s) ;
        }
    }
    private static Logger theLog = null ;

    public static Logger theLog() {
        if ( theLog == null ) {
            theLog = new Logger() ;
        }
        return theLog ;
    }
}
```

ThreadLocal Confinement

- Change the Singleton to a ThreadLocal.
- Interface to the class is unchanged - just the internal details of the factory are altered

ThreadLocal - per thread Singleton logger

```
import java.util.Vector ;

public class LoggerT {
    private Vector<String> log =
        new Vector<String>() ;

    private LoggerT(){}

    public void logit(String message) {
        log.add(message) ;
    }

    public void dump(String prefix) {
        for ( String s : log ) {
            System.out.println(prefix +
                ": " + s) ;
        }
    }
    private static ThreadLocal<LoggerT> tl_log =
        new ThreadLocal<LoggerT>() ;

    public static LoggerT theLog() {
        if ( tl_log.get() == null ) {
            tl_log.set( new LoggerT() ) ;
        }
        return tl_log.get() ;
    }
}
```

Immutability

- An object is immutable (in Java) iff
  - Its state cannot be modified after construction.
  - All its fields are **final**; **AND**
  - It is *properly constructed* (**this** does not escape).

- An object whose fields are all final may still be mutable.

How is this Possible?

- Declaring fields final documents to future maintainers which fields are not expected to change

Make all fields final unless they need to be mutable.

Safe Publication

- Published objects must be published safely.
- Chief violation of safety is publishing partially constructed objects.
- A consistent view of object state requires synchronization.

```
public class Bad {
  public Holder h = null ;

  public void init() {
    h = new Holder( 42 )
  }
}
```

```
public class Holder {
  private int n ;

  public Holder(int n) {
    this.n = n ;
  }

  public int getN() {
    return n ;
  }

  public void assertSane() {
    if ( n != n ) {
      throw AssertionError("OOPS") ;
    }
  }
}
```

Is this safe?
Why or Why Not?

Safe Publication: Mutable Objects

- Published objects must be published safely.
- The chief violation of safety is publishing partially constructed objects.

```
public class Bad {
  public Holder h = null ;

  public void init() {
    h = new Holder( 42 )
  }
}
```

Need to synchronize here

```
public class Holder {
  private int n ;

  public Holder(int n) {
    this.n = n ;
  }

  public int getN() {
    return n ;
  }

  public void assertSane() {
    if ( n != n ) {
      throw AssertionError("OOPS") ;
    }
  }
}
```

Safe Publication: Immutable Objects

- Immutable objects can be used even if safe reference publication is *not* synchronized.

```
public class Bad {
  public Holder h = null ;

  public void init() {
    h = new Holder( 42 )
  }
}
```

```
public class Holder {
  private final int n ;

  public Holder(int n) {
    this.n = n ;
  }

  public int getN() {
    return n ;
  }

  public void assertSane() {
    if ( n != n ) {
      throw AssertionError("OOPS") ;
    }
  }
}
```

Safe Sharing Heuristics

- **Thread confined**
  - Shared only within the thread.
  - No synch. needed.
- **Shared read-only**
  - Objects that are not mutated can be shared w/o synch.
  - Includes immutable & effectively immutable objects.
- **Shared thread-safe objects**
  - Have necessary synchronization "built-in"
  - Can access from multiple threads w/o special synch.
- **Guarded**
  - Not inherently thread-safe.
  - Only access when specific lock is held.
  - Threads must agree on *which* lock is required!