Java and Android Concurrency

Sharing Objects

fausto.spoto@univr.it

git@bitbucket.org:spoto/java-and-android-concurrency.git

git@bitbucket.org:spoto/java-and-android-concurrency-examples.git

Visibility

Synchronization has two goals:

- ensure mutual exclusion (everybody knows this)
- ensure visibility (nobody knows this)

What does this print?

```
public class NoVisibility {
    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(String[] args) {
        new ReaderThread().start();
        number = 42;
        ready = true;
    }
}
```

The Heisenberg Principle of Concurrent Programming

In the absence of synchronization, the compiler, processor, and runtime can do some downright weird things to the order in which operations appear to execute. Attempts to reason about the order in which memory actions "must" happen in insufficiently synchronized multithreaded programs will almost certainly be incorrect

A field might contain any of the values ever written into the field, but not necessarily the last one (out-of-thin-air safety). For `long` and `double` fields, even a value never written into the field might be seen!

Synchronization Guarantees Visibility

```
@NotThreadSafe
public class MutableInteger {
    private int value;

    public int get() {
        return value;
    }

    public void set(int value) {
        this.value = value;
    }
}
```

Synchronization Guarantees Visibility

```
@NotThreadSafe
public class MutableInteger {
    private int value;

    public int get() {
        return value;
    }

    public synchronized void set(int value) {
        this.value = value;
    }
}
```

Synchronization Guarantees Visibility

```
@ThreadSafe
public class SynchronizedInteger {
    private @GuardedBy("this") int value;

    public synchronized int get() {
        return value;
    }

    public synchronized void set(int value) {
        this.value = value;
    }
}
```

Locking, Visibility and `volatile` Fields

Locking guarantees visibility

When thread *A* executes a `synchronized` block, and subsequently thread *B* enters a `synchronized` block guarded by the same lock, the values of variables that were visible to *A* prior to realeasing the lock are guaranteed to be visible to *B* upon acquiring the lock

`volatile` guarantees visibility

When thread *A* writes to a volatile field and subsequently thread *B* reads the same field, the values of all variables that were visible to *A* prior to writing become visible to *B* after reading

```
volatile boolean asleep;

void tryToSleep() {
  while (!asleep)
    countSomeSheep();
}
```

Use `volatile` Variables Sparingly

- they have higher access cost
- compound operations are not atomic: `count++`

Locking can guarantee both visibility and atomicity; `volatile` variables can only guarantee visibility

Use `volatile` variables only if

1. writes do not depend on previous value or only one thread performs writes
2. the variable does not participatre in invariants with other variables
3. locking is not needed for any other reason

Publication and Escape

Publication

Publishing an object means making it available to code outside of its current scope:

- publishing internal state can compromise encapsulation and thread-safety
- publishing not fully constructed objects can compromise thread-safety

Escape

An object that is published when it should not have been is said to have escaped its intended scope

Examples of Escape

Escape through public fields

```
public static Set<Secret> knownSecrets;

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

Escape through return value

```
private String[] states = new String[]{ "AK", "AL" ... };

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

Examples of Escape

Escape through overriddable methods

```
private String[] states = new String[]{ "AK", "AL" ... };

...doSomething(states);

protected void doSomething(String[] ss) { ... }
```

Escape through non-static inner classes

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

Escape of Raw Objects

The examples with overriddable methods and non-static inner classes are particularly bad since they can allow a partially initialized object (raw) to escape its constructor

Partially initialized objects might not be usable from other threads, even if the class seems thread-safe

Do not allow the `this` reference to escape during construction

Thread Confinement

If an object is confined to a thread, that is, it can only be accessed by that single thread, then there is no need to make it thread-safe, since all its uses are automatically thread-safe

Ad-hoc thread confinement

- Java's Swing graphical library
- Android graphical library
- database connections from a connection pool

Stack confinement

Local variables are intrinsically confined to the executing thread

- always for primitive values
- if not published, for reference values

Thread Confinement through ThreadLocal

ThreadLocals are a sort of per-thread static fields. Calls to get yield the value associated to the currently executing thread, either initialized through initialValue or subsequently modified through set

```
private ThreadLocal<Connection> connectionHolder
  = new ThreadLocal<>() {
    public Connection initialValue() {
      return DriverManager.getConnection(DB_URL);
    }
  };

public Connection getConnection() {
  return connectionHolder.get();
}
```

Immutability

Immutable objects are good

Immutable objects are thread-safe and can be shared and published without synchronization

An object is immutable if

1. its state cannot be modified after construction
2. all its fields are `final`
3. it is properly constructed (`this` does not escape during construction)

An Immutable Object Can Well Use Modifiable Objects

```
@Immutable
public final class ThreeStooges {
  private final Set<String> stooges = new HashSet<>();

  public ThreeStooges() {
    stooges.add("Moe");
    stooges.add("Larry");
    stooges.add("Curly");
  }

  public boolean isStooge(String name) {
    return stooges.contains(name);
  }

  public String getStoogeNames() {
    return stooges.toString();
  }
}
```

Immutable Objects and Collections in State

Frequently, one is tempted to return a modifiable collection from the state of an object: this breaks encapsulation, makes the state escape and makes the object mutable:

```
@Mutable
public class C {
  private final Set<Element> set = new HashSet<>();
  ...
  public Set<Element> getElements() {
    return set;
  }
}
```

Immutable Objects and Collections in State

Bad solution: return a copy of the set

- the user might modify the copy and think this actually does something

```
@Immutable
public class C {
  private final Set<Element> set = new HashSet<>();
  ...
  public Set<Element> getElements() {
    return new HashSet<>(set);
  }
}
```

Immutable Objects and Collections in State

Bad solution: return a `Collections.unmodifiableSet`

- the user might modify the copy, because he thinks this actually does something, and get an exception, but only at runtime

```
@Immutable
public class C {
  private final Set<Element> set = new HashSet<>();
  ...
  public Set<Element> getElements() {
    return Collections.unmodifiableSet(set);
  }
}
```

Immutable Objects and Collections in State

Good solution: make the class iterable or return a new `Iterable` instead of a `Set`

- in 99% of the cases, iterations is all the user wants to do

```
@Immutable
public class C implements Iterable<Element> {
  private final Set<Element> set = new HashSet<>();
  ...
  public Iterator<Element> iterator() {
    return set.iterator();
  }
}
```

Immutable Objects and Collections in State

Good solution: use internal iteration, passing a task as an interface or lambda-expression

```
@Immutable
public class C {
  private final Set<Element> set = new HashSet<>();

  public interface Task {
    void process(Element e);
  }

  ...
  public void forEach(Task task) {
    for (Element e: set)
      task.process(e);
  }
}
```

Immutable Objects and Collections in State

Good solution: define and return a new `ImmutableSet` copy

- for the 1% of the cases, when the user wants to do more than iterate

```
public interface ImmutableSet<E> extends Iterable<E> {
  int size();
}

@Immutable
public class C {
  private final Set<Element> set = new HashSet<>();
  ...
  public ImmutableSet<Element> getElements() {
    return new ImmutableSet<Element>() {
      public int size() { return set.size(); }
      public Iterator<Element> iterator() { return set.iterator();
    };
  }
}
```

Don’t Be Afraid of Immutability

An immutable object has only a single state, but references to an immutable object can be updated!

Moreover:

- no locking cost
- no defensive copies
- instances can be shared instead of duplicated
- reduced generational garbage-collection

An Immutable Object for the Factorization Cache

```
@Immutable
public class OneValueCache {
  private final BigInteger lastNumber;
  private final BigInteger[] lastFactors;

  public OneValueCache(BigInteger i, BigInteger[] factors) {
    lastNumber = i;
    lastFactors = Arrays.copyOf(factors, factors.length);
  }

  public BigInteger[] getFactors(BigInteger i) {
    if (lastNumber == null || !lastNumber.equals(i))
      return null;
    else
      return Arrays.copyOf(lastFactors, lastFactors.length);
  }
}
```

Cached Factorizer without Synchronization

```
@ThreadSafe
public class VolatileCachedFactorizer extends StatelessFactorizer {
  private volatile OneValueCache cache
      = new OneValueCache(null, null);

  @Override
  protected void doPost(HttpServletRequest request,
                        HttpServletResponse response) {

    BigInteger i = extractFromRequest(request);
    BigInteger[] factors = cache.getFactors(i);
    if (factors == null) {
      factors = factor(i);
      cache = new OneValueCache(i, factors);
    }
    encodeIntoResponse(response, factors);
  }
}
```

Unsafe Publication

Very often, we do want to share objects. But publication must be done in a safe way, or otherwise very weird things might happen

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

public class Holder {
  private int n;
  public Holder(int n) { this.n = n; }
  public void assertSanity() {
    if (n != n) throw new AssertionError("This is false");
  }
}
```

1. `holder` might be seen to contain `null` or a stale, old `Holder`
2. `n`'s value might suddenly change, making the assertion fail

Safe Publication

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

public class Holder {
  private final int n;
  public Holder(int n) { this.n = n; }
  public void assertSanity() {
    if (n != n) throw new AssertionError("This is false");
  }
}
```

Making just `n` `final` would only satisfy the assertion: the object is published, but the reference to it might be stale

Making `holder` `volatile` solves all problems

Safe Publication Idioms

Immutable objects can be published in any way. Mutable objects must be safely published, so that their internal state is guaranteed to be visible. This can happen in many ways:

- by storing the object into a `volatile` field
- by storing the object into a `final` field of a properly constructed object
- by initializing the object inside a class static initializer
- by storing the object into a properly guarded field
- by passing the object to a synchronized collection class from the standard Java library

Exercise 1: Parallel Matrix Multiplication

Make the matrix multiplication constructor in the following code parallel, through the use of multithreading

- how much faster do you think it will run then on a *n*-core machine?
- once the multiplication is divided across different threads, why is the resulting matrix safely published to the calling thread?

Exercise 1: Sequential Matrix Multiplication

```
public class Matrix {
  private final double[][] elements;
  private final static Random random = new Random();

  public Matrix(int m, int n) {
    this.elements = new double[m][n];
    for (int x = 0; x < n; x++)
      for (int y = 0; y < m; y++)
        elements[y][x] = random.nextDouble() * 100.0 - 50.0;
  }

  public int getM() {
    return elements.length;
  }

  public int getN() {
    return elements[0].length;
  }
```

Exercise 1: Sequential Matrix Multiplication

```
  // modify this constructor, make it run in parallel!
  private Matrix(Matrix left, Matrix right) {
    int m = left.getM();
    int p = left.getN();
    int n = right.getN();

    this.elements = new double[m][n];
    for (int x = 0; x < n; x++)
      for (int y = 0; y < m; y++) {
        double sum = 0.0;
        for (int k = 0; k < p; k++)
          sum += left.elements[y][k] * right.elements[k][x];
        this.elements[y][x] = sum;
      }
  }

  ...
}
```

Esercise 2: Implement a Swing GUI for the Chat Servlets

Build a Swing GUI for the chat servlets

- it should include a button that allows one to send a new chat message (author and text)
- and a text area that can be refreshed to show the last 20 messages from the chat