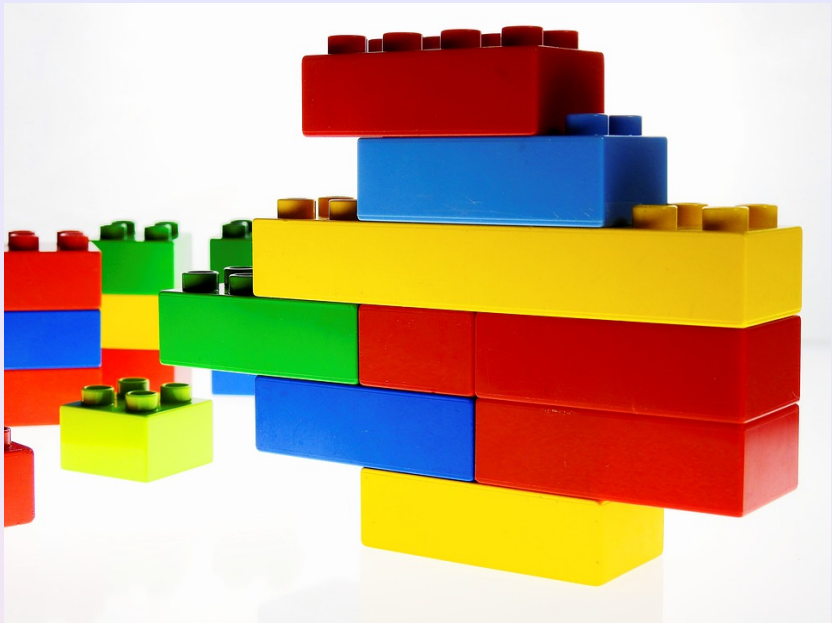

Java and Android Concurrency

Composing Objects

fausto.spoto@univr.it

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

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

Three Questions

This section will answer three questions:

1. is a thread-safe class necessarily built from thread-safe components?
2. if a class is composed from thread-safe components, is it thread-safe as well?
3. if a class is thread-safe, is a subclass still thread-safe?

The answer will constantly be "not always, it depends"...

Designing a Thread-Safe Class

- identify the variables that form the object's state
  - a subset of everything reachable from its fields (that is, the objects it owns, think about collection classes)
- identify the invariants that constrain the state variables
  - not all values might be valid
  - there might be constraints between state variables
  - operations might have preconditions
- establish a policy for managing concurrent access to the object's state

Instance Confinement

One can use a non-thread-safe instance field in a thread-safe class, as long as it is properly encapsulated and accessed through a suitable synchronization policy. The following example uses the monitor pattern:

```
@ThreadSafe
public class PersonSet {
  @GuardedBy("this") private final Set<Person> mySet = new HashSet<>();

  public synchronized void addPerson(Person p) {
    mySet.add(p);
  }

  public synchronized boolean containsPerson(Person p) {
    return mySet.contains(p);
  }

  interface Person {}
}
```

Lock Encapsulation

An alternative implementation might use a private lock, to avoid interference from outside the class:

```
@ThreadSafe
public class PersonSet {
  @GuardedBy("itself") private final Set<Person> mySet = new HashSet<>();

  public void addPerson(Person p) {
    synchronized (mySet) {
      mySet.add(p);
    }
  }

  public boolean containsPerson(Person p) {
    synchronized (mySet) {
      return mySet.contains(p);
    }
  }

  interface Person {}
}
```

Instance Confinement through Defensive Copy

We will use this non-thread-safe class and still get a thread-safe class:

```
@NotThreadSafe
public class MutablePoint {
  public int x, y;

  public MutablePoint() {
    x = 0;
    y = 0;
  }

  public MutablePoint(MutablePoint p) {
    this.x = p.x;
    this.y = p.y;
  }
}
```

Instance Confinement through Defensive Copy

A class tracking the current positions of a set of vehicles:

```
@ThreadSafe
public class MonitorVehicleTracker {
  @GuardedBy("this") private final
    Map<String, MutablePoint> locations;

  private static Map<String, MutablePoint> deepCopy
      (Map<String, MutablePoint> m) {

    Map<String, MutablePoint> result = new HashMap<>();
    for (String id : m.keySet())
      result.put(id, new MutablePoint(m.get(id)));

    return Collections.unmodifiableMap(result);
  }
```

Instance Confinement through Defensive Copy

```
  public MonitorVehicleTracker(Map<String, MutablePoint> locations) {
    this.locations = deepCopy(locations);
  }

  public synchronized Map<String, MutablePoint> getLocations() {
    return deepCopy(locations);
  }

  public synchronized MutablePoint getLocation(String id) {
    MutablePoint loc = locations.get(id);
    return loc == null ? null : new MutablePoint(loc);
  }

  public synchronized void setLocation(String id, int x, int y) {
    MutablePoint loc = locations.get(id);
    if (loc == null)
      throw new IllegalArgumentException("No such ID: " + id);
    loc.x = x;
    loc.y = y;
  }
}
```

Delegating Thread-Safety

If a class has only thread-safe instance fields, is it thread-safe as well?

it depends. . .

`CountingFactorizer` was thread-safe because its state coincides with that of its `AtomicLong` thread-safe only instance field and since it does not impose any extra constraint on the states of that class

An Example of Delegation of Thread-Safety

Let us delegate the thread-safety of our vehicle tracker to the thread-safety of the library class `ConcurrentHashMap`. To further simplify the work, let us use an immutable point class:

```
@Immutable
public class Point {
  public final int x, y;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

An Example of Delegation of Thread-Safety

No explicit synchronization!

```
@ThreadSafe
public class DelegatingVehicleTracker {
  private final ConcurrentMap<String, Point> locations;

  public DelegatingVehicleTracker(Map<String, Point> points) {
    locations = new ConcurrentHashMap<>(points);
  }

  public Map<String, Point> getLocations() {
    return Collections.unmodifiableMap(new HashMap<>(locations));
  }

  public Point getLocation(String id) {
    return locations.get(id);
  }

  public void setLocation(String id, int x, int y) {
    if (locations.replace(id, new Point(x, y)) == null)
      throw new IllegalArgumentException("invalid vehicle name: " + id);
  }
}
```

Delegating to More, Independent State Variables

The resulting class is thread-safe if the state variables are thread-safe and independent:

```
@ThreadSafe
public class VisualComponent {
  private final List<KeyListener> keyListeners = new CopyOnWriteArrayList<>();
  private final List<MouseListener> mouseListeners = new CopyOnWriteArrayList<>();

  public void addKeyListener(KeyListener listener) {
    keyListeners.add(listener);
  }

  public void addMouseListener(MouseListener listener) {
    mouseListeners.add(listener);
  }

  public void removeKeyListener(KeyListener listener) {
    keyListeners.remove(listener);
  }

  public void removeMouseListener(MouseListener listener) {
    mouseListeners.remove(listener);
  }
```

Delegating to More, Independent State Variables

```
  public void fireKeyListeners(KeyEvent event) {
    for (KeyListener listener: keyListeners)
      listener.keyPressed(event);
  }

  public void fireMouseListeners(MouseEvent event) {
    for (MouseListener listener: mouseListeners)
      listener.mouseClicked(event);
  }
}
```

CopyOnWriteArrayList

This thread-safe library class allows iterating on the list and its concurrent update, without throwing a `ConcurrentModificationException`. This is payed with an increased cost for list update

Invariants between State Variables Make Delegation Fail

```
@NotThreadSafe
public class NumberRange {
  // INVARIANT: lower <= upper
  private final AtomicInteger lower = new AtomicInteger(0);
  private final AtomicInteger upper = new AtomicInteger(0);

  public void setLower(int i) {
    if (i > upper.get())  // check
      throw new IllegalArgumentException("can't set lower to " + i + " > upper");
    lower.set(i);         // then act
  }

  public void setUpper(int i) {
    if (i < lower.get())  // check
      throw new IllegalArgumentException("can't set upper to " + i + " < lower");
    upper.set(i);         // then act
  }

  public boolean isInRange(int i) {
    return (i >= lower.get() && i <= upper.get());
  }
}
```

Invariants between State Variables Require Synchronization

```
@ThreadSafe
public class NumberRangeSynchronized {
  // INVARIANT: lower <= upper
  private @GuardedBy("this") int lower = 0;
  private @GuardedBy("this") int upper = 0;

  public synchronized void setLower(int i) {
    if (i > upper)
      throw new IllegalArgumentException("can't set lower to " + i + " > upper");
    lower = i;
  }

  public synchronized void setUpper(int i) {
    if (i < lower)
      throw new IllegalArgumentException("can't set upper to " + i + " < lower");
    upper = i;
  }

  public synchronized boolean isInRange(int i) {
    return i >= lower && i <= upper;
  }
}
```

Extending Thread-Safe Classes

Suppose we want to add an atomic `putIfAbsent` operation to the thread-safe library class `Vector`. This is a check-then-act operation, hence synchronization is needed, despite the fact that each vector operation is already synchronized

```
@ThreadSafe
public class BetterVector<E> extends Vector<E> {
  public synchronized boolean putIfAbsent(E x) {
    boolean absent = !contains(x);
    if (absent)
      add(x);
    return absent;
  }
}
```

This works since `Vector` is explicitly designed to use its intrinsic lock for synchronization. Otherwise, changing the synchronization policy in `Vector` would break `BetterVector`

Client-Side Locking

`BetterVector` uses client-side locking, that is, locks the same lock that the base class uses for its own synchronization.
Client-side locking can also be used for composition:

```
@ThreadSafe
public class ListHelper<E> {
  public final List<E> list = Collections.synchronizedList(new ArrayList<E>());

  public boolean putIfAbsent(E x) {
    synchronized (list) {
      boolean absent = !list.contains(x);
      if (absent)
        list.add(x);
      return absent;
    }
  }
}
```

Using `synchronized (this)` instead would make the class non-thread-safe

Adding Another Layer of Synchronization

Another, less fragile, way of adding a `putIfAbsent` operation to a synchronized list:

```
@ThreadSafe
public class ImprovedList<T> implements List<T> {
  private final List<T> list;

  // PRE: list argument is thread-safe
  public ImprovedList(List<T> list) { this.list = list; }

  public synchronized boolean putIfAbsent(T x) {
    boolean absent = !list.contains(x);
    if (absent)
      list.add(x);
    return absent;
  }

  // Plain vanilla delegation for List methods
  // Mutative methods must be synchronized to ensure atomicity of putIfAbsent
  public int size() { return list.size(); }
  public synchronized boolean add(T e) { return list.add(e); }
  ...
}
```

Documenting Synchronization Policies

- Client-side locking shows that it is important to know the synchronization policy used by a class, in order to extend the class with new synchronized functionalities
- Sadly, current programming languages have no syntactical support for specifying and enforcing a synchronization policy

It remains documentation

Document a class's thread safety guarantees for its clients; document its synchronization policy for its maintainers

- use annotations `@ThreadSafe`, `@NotThreadSafe`, `@GuardedBy`, `@Immutable`
- state if client-side locking is supported, and which lock should be used

Exercize

Write a web application with two servlets:

- `NextPrime` prints the next prime number: the first invocation prints 2, the second 3, the third 5 and so on
- `NextPrimes?howmany=NNNN` prints the next consecutive `NNNN` prime numbers, such as `[53, 59, 61, 67, 71, 73, 79, 83, 89, 97]`

These two servlets must use the same next prime counter: if `NextPrime` prints 7, then `NextPrimes` will start from 11