Retrospective on making a thread-safe class better!
You may extend, but not always
  Depends, it does, on the code maze
Is the fear of making things worse
  Making you scamper from that source?
Composition is the wind in your sails
  Use it, when all else fails

Shrideep Pallickara
Computer Science
Colorado State University

Frequently asked questions from the previous class survey

- Why can’t a shared data structure be used by synchronized methods of separate classes?
- Are all standard data structures typically synchronized?
- When is it preferrable to make object references public?
- If you use explicit locks, is it possible to acquire the object lock?
Topics covered in this lecture

- State ownership
- Guarding state with private locks
- Adding functionality to thread safe classes
- Synchronized collections

This place is always such a mess
Sometimes I think I'd like to watch it burn
I'm so alone
Feel just like somebody else
Man, I ain't changed, but I know I ain't the same
One Headlight, Jakob Dylan, The Wallflowers

STATE OWNERSHIP
State ownership

- Defining which variables form an object's state
  - We wish to consider only that which the object owns

- Ownership
  - Not explicitly specified in the language
  - Element of program design

State ownership: Encapsulation and ownership go together

- Object encapsulates the state it owns
  - Owns the state it encapsulates

- Owner gets to decide on the locking protocol

- If you publish a reference to a mutable object?
  - You no longer have exclusive control
Instance confinement

- Object may not be thread-safe
  - But we could still use it in a thread-safe fashion

- Ensure that:
  - It is accessed by only one thread
  - All accesses guarded by a lock

Confinement and locking working together

```java
public class PersonSet {
    private final Set<Person> mySet = new HashSet<Person>();

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

    public synchronized boolean containsPerson(Person p) {
        return mySet.contains(p);
    }
}
```
Looking at our previous example

- State of PersonSet managed by HashSet, which is not thread-safe
- But mySet is
  - Private
  - Not allowed to escape
  - Confined to PersonSet

But we have made no assumptions about Person

- If it is mutable, additional synchronization is needed
  - When accessing Person from PersonSet
- Reliable way to achieve this?
  - Make Person thread-safe
- Less-reliable way?
  - Guard Person objects with a lock
  - Ensure that clients follow protocol of acquiring appropriate lock, before accessing Person
Instance confinement is the **easiest way** to build thread-safe classes

- Class that confines its state can be analyzed for thread-safety
  - Without having to examine the whole program

**Guarding state with private locks**
Guarding state with a private lock

```java
public class PrivateLock {
    private final Object myLock = new Object();

    private Widget widget; //guarded by myLock

    public void someMethod() {
        synchronized(myLock) {
            //Access and modify the state of the widget
        }
    }
}
```

Why guard state with a private lock?

- Doing so encapsulates the lock
  - Client code cannot acquire it!
- Publicly accessible lock allows client code to participate in its synchronization policy
  - Correctly or incorrectly
- Clients that improperly acquire an object’s lock cause liveness issues
- Verifying correctness with public locks requires examining the entire program not just a class
A Vehicle Tracker application

- Each vehicle
  - Identified by a String
  - Location represented by (x, y) coordinates

- VehicleTracker class
  - Tracks the identity and location of all known vehicles
Viewer thread and Updater Thread

**Viewer**

Map<String, Point> locations = vehicles.getLocations();
for (String key: locations.keySet())
    renderVehicle(key, locations.get(key));

**Updater**

public void vehicleMoved(VehicleMovedEvent evt) {
    Point loc = evt.getNewLocation();
    vehicles.setLocation(evt.getVehicleId(), loc.x, loc.y);
}

The MonitorVehicleTracker

public class MonitorVehicleTracker {
    private final Map<String, MutablePoint> 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 IllegalArgumentException(...)}
        loc.x = x;
        loc.y = y;
    }
    private deepCopy() { ... }
}
The tracker class is thread-safe, even though 
**MutablePoint** may not be

```java
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;
    }
}
```

What the **deepCopy()** looks like

```java
public class MonitorVehicleTracker {
    ...
    private Map<String, MutablePoint> 
    deepCopy(Map<String, MutablePoint> m) {
        Map<String, MutablePoint> result = 
        new HashMap<String, MutablePoint>();
        for (String id: m.keySet())
            result.put(id, new MutablePoint(m.get(id)));
        return Collections.unmodifiableMap(result);
    }
}
```
**The Collections utility class**

- List<String> readOnlyList = Collections.unmodifiableList(myList);

- **Note:**
  - Nothing to *differentiate* this as a read-only list
  - You have access to the mutator methods
    - But calling them results in an UnsupportedException

---

**Delegating thread-safety**

```java
public class DelegatingVehicleTracker {
    private final ConcurrentHashMap<String, Point> locations;
    private final Map<String, Point> unmodifiableMap;

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

    public Map<String, Point> getLocations() { return unmodifiableMap; }

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

    public void setLocation(String id, int x, int y) {
        if (locations.replace(id, new ImmutablePoint(x, y)) == null)
            throw new IllegalArgumentException("Invalid Vehicle ID");
    }
}
```
**Immutable Point**

```java
public class ImmutablePoint {
    public final int x, y;

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

**When delegation fails**

```java
public class NumberRange {
    private final AtomicInteger lower = new AtomicInteger(0);
    private final AtomicInteger upper = new AtomicInteger(0);

    public void setLower(int i) {
        if (i > upper.get()) {
            throw IllegalArgumentException("lower > upper!");
        }
    }

    public void setUpper(int i) {
        if (i < lower.get()) {
            throw IllegalArgumentException("upper < lower!");
        }
    }

    public boolean isInRange(int i) {
        return (i >= lower.get() && i <= upper.get());
    }
}
```
Adding functionality to existing thread-safe classes

- Sometimes we have a thread-safe class that supports *almost all* the operations we need.
- We should be able to add a new operation to it *without undermining* its thread safety.
Adding a put-if-absent function to a List

- The operation put-if-absent must be atomic
- If List does not have X and we add X twice?
  - It's a problem because the collection should only have one X
- But if put-if-absent is not atomic?
  - Two threads could see that X is absent and the list then has 2 copies of X

Adding additional operations

1. Safest way is to **modify** the original class
2. **Extend** the class
   - Often base classes do not expose enough of their state to allow this approach
3. Place the extension code in a “**helper class**”
4. **Composition**
Extending Vector to have a put-if-absent method

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

Client side locking

- Sometimes extending a class or adding a method is not possible
- For e.g., if ArrayList is wrapped with a Collections.SynchronizedList wrapper
  - Client code does not even know the class of the List object
- In such situations, the 3rd strategy of using a helper class comes in
Client-side locking

```java
public class ListHelper<E> {

    private List<E> list =
        Collections.synchronizedList(new ArrayList<E>());

    ...

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

Using the intrinsic lock of ListHelper to synchronize access to List

Client-side locking: Let’s try again ...

```java
public class ListHelper<E> {

    private 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;
        }
    }
}
```
Contrasting extending a class AND client-side locking

- Extending a class to add an atomic operation?
  - Distributes locking code over multiple classes in the object hierarchy

- Client-side locking is even more fragile
  - We put locking code for a Class C in classes that are completely unrelated to it

Composition: A less fragile alternative to adding an atomic operation

```java
public class ImprovedList<T> implements List<T> {
    private final List<T> list = new ArrayList<T>();
    ...
    public synchronized boolean putIfAbsent(T x) {
        boolean absent = !list.contains(x);
        if (absent) {
            list.add(x);
        }
        return absent;
    }
    public synchronized void clear() {list.clear();}
    // delegate other list methods ...
}
```
More about the ImprovedList

- No worries even if the underlying List is not thread-safe
- ImprovedList uses its intrinsic lock
- Extra layer of synchronization may add small performance penalty
  - But it is much better than attempting to mimick the locking strategy of another object

Synchronized Collections
Synchronized collections

- These include classes such as Vector and Hashtable

- There is also the **synchronized wrapper** classes
  - Created by Collections.synchronizedX factory methods
    - E.g., Collections.synchronizedList(List list), Collections.synchronizedMap(Map m), Collections.synchronizedSet(Set s)

Problems with synchronized collections

- Thread-safe but **additional client-side locking needed** to guard **compound actions**
  - Iteration
  - Navigation
    - Find the next element
  - Conditional operations
    - Put-if-absent
Compound actions producing confusing results

```java
public Object getLast(Vector list) {
    int lastIndex = list.size() - 1;
    return list.get(lastIndex);
}

public void deleteLast(Vector list) {
    int lastIndex = list.size() - 1;
    list.remove(lastIndex);
}
```

Interleaving of `getLast` and `deleteLast`

A

1. `size -> 10`
2. `remove(9)`

B

1. `size -> 10`
2. `get(9)`
3. `Uh oh!`
Are there problems with this code?

```java
for (int i=0; i < vector.size(); i++) {
    doSomething(vector.get(i));
}
```

There is a chance that other threads may modify `vector` between the calls to `size()` and `get()`.

Compound actions using client-side locking

```java
public Object getLast(Vector list) {
    synchronized(list) {
        int lastIndex = list.size() - 1;
        return list.get(lastIndex);
    }
}

public void deleteLast(Vector list) {
    synchronized(list) {
        int lastIndex = list.size() - 1;
        list.remove(lastIndex);
    }
}
```
Iterators

- The standard way to **iterate** over a Collection is with an Iterator
- Using iterators does not mean that you don’t need to lock the collection
- Iterators returned by synchronized collections are *not designed for concurrent modification*

Iterators in synchronized collections

- Iterators of synchronized collections are **fail-fast**
- If they detect that the collection has changed since iteration began?
  - Unchecked ConcurrentModificationException is thrown
Fail-fast iterators are not designed to be fool proof

- Designed to catch concurrency errors on a good-faith basis
- Associate a modification count with the collection
- If the modification count changes during iteration?
  - hasNext() or next() throws ConcurrentModificationException

Let’s look at this code snippet

```java
List<Widget> widgetList = Collections.synchronizedList(new ArrayList<Widget>());
...
for (Widget w: widgetList)
  doSomething(w);

// May throw ConcurrentModificationException
```

Internally javac generates code that uses Iterator and repeatedly calls hasNext() and next() to iterate the List
How to prevent the ConcurrentModificationException

- Hold the collection lock for the duration of the iteration
- Is this desirable?

Issues with locking a collection during iteration

- Other threads that need to access the collection will block
- If the collection is large or if the task performed on each element is lengthy?
  - The wait could be really long
Locking collection and scalability

- The longer a lock is held
  - The more likely it will be contained

- If many threads are waiting for a lock?
  - Throughput and CPU utilization plummet

- ALTERNATIVE:
  - Deep-copy the collection and iterate over the copy
  - The copy is thread-confined

Hidden Iterators

```java
public class HiddenIterator {
    private final Set<Integer> set = new HashSet<Integer>();
    public synchronized void add(Integer i) { set.add(i); }
    public synchronized void remove(Integer i) { set.remove(i); }
    public void diagnostics() {
        System.out.println("DEBUG: Elements in set: " + set);
    }
}
```

- Lock should have been acquired for the System.out
- Iterators are also invoked for hashCode and equals
The contents of this slide-set are based on the following references

- https://www.javaspecialists.eu/archive/Issue192b.html