Chapter 4

- 50+ years of CS and SE devoted to issues of composability: how to create solutions to big problems by combining (composing) solutions to smaller problems
  - Mechanisms – e.g. intrinsic locks; GC
  - Techniques – Invariants, pre- and post-conditions; confinement; delegation; etc. (Ch4)
  - Libraries – working code embodying the techniques for specific domains (Ch 5)
- This chapter mainly about techniques

Invariants – <u>the</u> fundamental technique

- **What property is always true of an object when it is in a correct state?**
- **For sequential programming, the class invariant gives you critical information about what each public method needs to achieve**
- **For concurrent programming, the class invariant tells you which fields are related and therefore have to be protected by a single lock**

Post-conditions and Pre-conditions

- The post-condition of a method tells you what that method is supposed to accomplish
  - (A no-op will preserve the invariant but that's not very interesting or useful!)
- The pre-condition tells you what (beyond the invariant) is supposed to be true for a method to successfully reach the post-condition
  - In sequential programming calling a method when its precondition is false is an error
  - In concurrent programming we can wait for the precondition to become true

Two common problems

- Composing a thread-safe class from unsafe building blocks
- Composing a thread-safe class when the building blocks are already thread-safe

Confinement Technique (4.2) – thread-safe object built from unsafe objects

- Allow access to a thread-unsafe object only through another object that is thread-safe

```
public class PersonSet {
   private final Set<Person> mySet = new
 HashSet<Person>();
   public synchronized void add(Person p) {
      mySet.add(p); }
   public synchronized boolean contains(Person p) {
      return myset.contains(p); }
}
```

- hashSet is not ThreadSafe, PersonSet is
- Idea of ownership: PersonSet owns mySet but probably not the Persons contained in it

Danger in Confinement Technique

- Inadvertant publication of what is supposed to be private (confined) mutable state

```
public synchronized MutablePoint getLocation(String id) {
   MutablePoint loc = locations.get(id);
   return loc == null ? Null : new MutablePoint(loc);
}
Public synchronized setLocation(String id, int x, int y) {
   MutablePoint loc = locations.get(id);
   if (loc == null) { … exception …}
   loc.x = x; loc.y = y
}
```

- My preference would be to express this interface using ImmutablePoints.

Thread-safe objects built from thread-safe components – Delegating safety (4.3)

- Delegation: giving responsibility for thread safety to the object(s) containing this object's state
  - ConcurrentMap (TS) instead of Map (not TS)
  - Atomic<foo>
- If this object's state involves multiple other objects delegation may or may not work
  - If the sub-objects are independent, ok
  - If the sub-objects are related, this object must provide its own synchronization – even if all the sub-objects are themselves thread-safe

Example

```
class PongPaddle {
   private final AtomicInteger left = new AtomicInteger(0);
   private final AtomicInteger right = new AtomicInteger(1);
   public void move(int dx) { left.getAndAdd(dx);
  right.getAndAdd(dx); }
   public void changeWidth(int dw) { right.getAndAdd(dw); }
   public boolean hit(int pos) {
      return left.get()<=pos && pos <= right.get();
   }
}
```

- **No visibility concerns**
- **What is the invariant that relates left and right?**
- **What should we do to fix it?**

Reduce, Reuse, Recycle

- **Don't Repeat Yourself (DRY)**
- **I am very *anti* cut-and-paste coding**
  - **Hard on the reader**
  - **Hard on the maintainer**
  - **Instead of 1 change, n changes**
  - **Instead of 1 bug, n bugs**
- **Design code so there only needs to be one copy (use parameterization, polymorphic parameterization)**
- **Even better, reuse existing code that does almost the right thing**
- **How does this interact with synchronization?**

Adding functionality (4.4)

- Example: add putIfAbsent to a collection that already supports atomic contains() and add() methods
- Four approaches
  - Modify existing class
  - Extend existing class or
  - Wrap existing class – "client-side" locking
  - Composition -

1. Modify existing class

- Assuming the existing class is already thread-safe:
  - Introduce a new method that uses the same synchronization technique already in use
  - Best way but
  - Assumes you have control over the existing class
    - Would not be the case for library classes

2. Extend the existing class

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

- Vector is a thread-safe library class
- Vector provides enough primitive building blocks to allow construction of putIfAbsent
- Fig. 4.13 Note the benefit of re-entrant locks!
- Note the implicit assumption that we understand the way that Vector does synchronization – using intrinsic locks, in this case
- Note that we don't have any dependency on Vector's implementation

3. Client-side locking

- Assume v is a thread-safe list obtained from
  `v = Collections.synchronizedList(new ArrayList<E>());`
- Type of this object is List<E> -- not extendable
- Any code that wants to do putIfAbsent item x to such a list, v, can write

```
synchronized (v) {
   if (!v.contains(x)) v.add(x);
}
```

- Could be placed in a helper class – beware you have to synchronize on the *list* and not on the helper object – see Figs 4.14 and 4.15
- Still depending on knowing the synch policy for the wrapped object
- … and spreading the knowledge about the synchronization policy far and wide

Composition

- Mimic the idea of Collections.synchronizedList
  - Provide all the synchronization in a new object that extends the functionality of an existing object instance (not class)
  - Delegates most operations to the existing object

```
Public class ImprovedList<T> implements List<T> {
   private final List<T> list;
   public ImprovedList(List<T> list) {
      this.list = list; }
   public synchronized boolean putIfAbsent(T x) …
   public synchronized boolean contains(T x) {
      return list.contains(x); }
   …
Note this is similar to how we handled an non-thread-safe
  object
```

Intro to Chapter 5 – Building Blocks

- Chapter 4 was about low-level techniques
- This chapter is about libraries – embodiments of the techniques
- Section 5.1 Synchronized collections – read to see why you want to use Concurrent collections instead
  - The idioms described are even more unsafe than asserted in the book because of visibility problems

Chapter 5 topics

- Concurrent collections (5.2)
- The ubiquitous producer-consumer pattern (5.3)
- Interruptable methods (5.4)
- Primitive synchronizers (5.5)