Threadsafe design
- Confinement (revisited)
- Java monitor pattern
- Delegating thread safety

Threadsafe design

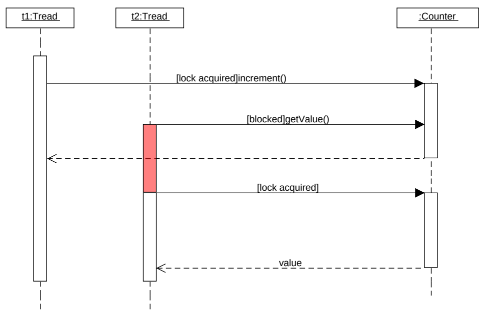
The design process for a threadsafe class should include three basic elements:

- Identify the variables (members) that form the object's state;
- Identify the invariants that constrain the state variables;
- Establish a policy for managing concurrent access to the objects state.

A threadsafe class


```
@ThreadSafe
public final class Counter {
    @GuardedBy("this") private long value = 0;

    public synchronized long getValue() {
        return value;
    }

    public synchronized long increment() {
        if (value == Long.MAX_VALUE)
            throw new IllegalStateException("counter overflow");
        return ++value;
    }
}
```


Counter object used as monitor



Synchronisation requirements

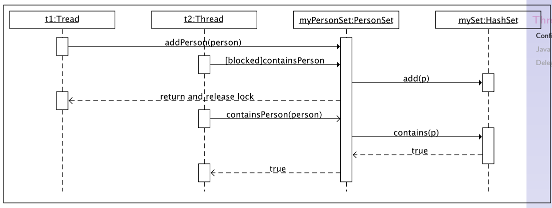
- You cannot ensure threadsafety without understanding an object's invariants and postconditions. Constraints on the valid values or state transtions for state variables can create atomicity and encapsulation requirements.
- E.g. the Counter class:
  - the state is determined by the value of `value`.
  - Counter has a constraint: `value` is not negative.
  - `increment()` has a postcondition: the only valid state is that `value` is indeed incremented by 1.

Confinement revisited

- Encapsulating data within an object confines access to the data to the object's methods, making it easier to ensure the data is always accessed with the appropriate lock held.

```
@ThreadSafe
public class PersonSet {
    @GuardedBy("this")
    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);
    }
}

interface Person {
}
```


PersonSet object used as monitor



Confinement makes threadsafeness easier

- The state of class `PersonSet` is managed by a HashSet which is not threadsafe, but mySet is private and not allowed to escape: *confinement*.
- Access methods `addPerson` and `containsPerson` acquire a lock on the `PersonSet`. Therefore `PersonSet` is *threadsafe*.
- Confinement makes it easier to build thread-safe classes because a class that confines its state can be analyzed for thread safety without having to examine the whole program.

Monitor programming model

- The monitor as defined by Hoare is similar but not the same as the Java model.
  - Hoare's monitor makes, in Java terms, all methods in a class `synchronized`. In the Java variant, locking is reentrant (by the same Thread).
- In the Java monitor pattern, all the object's state is encapsulated by the object and guarded by the object's own intrinsic lock.
- This is the lock you use when you make a method `synchronized`. As an example: see the `Counter` class.

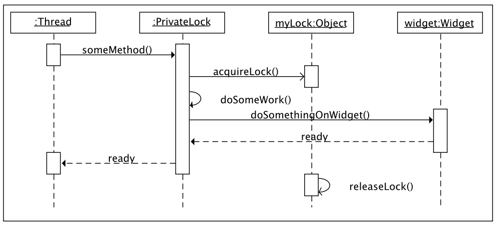
Using a auxiliary object as lock

The instance myLock of type Object is used as a lock. This is a typical Java idiom or 'Pattern'.

```
public class PrivateLock {
    private final Object myLock = new Object();
    @GuardedBy("myLock") Widget widget;

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

There is a pattern here! ***The java Monitor-pattern.***

PrivateLock

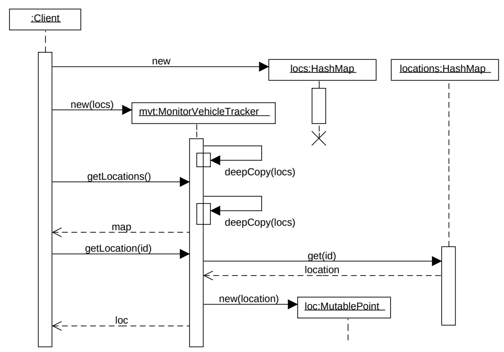


Delegating thread safety

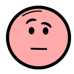
Monitor-based vehicle tracker implementation: thread safety although MutablePoint is not thread-safe.

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

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

    public synchronized Map<String, MutablePoint> getLocatio
        return deepCopy(locations);
    }
    public synchronized MutablePoint getLocation(String id)
        MutablePoint loc = locations.get(id);
        return loc == null ? null : new MutablePoint(loc);
    }
```

MonitorVehicleTracker sequence diagram



Immutable Point class: Mwha!



```
@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;
    }
}
```


Immutable Point

If the components of our class are already thread safe, do we need an addiditional layer of thread safety?

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

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

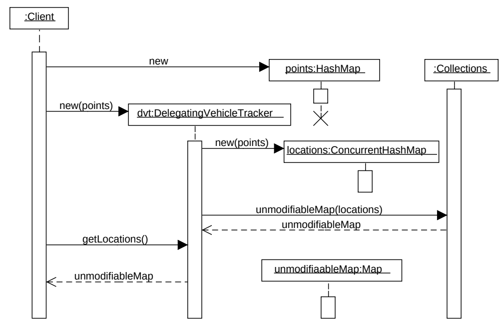
See next slide for the `DelegatingVehicleTracker` which uses a thread-safe (immutable) `Point` class and a thread-safe Map implementation to store the locations using immutable points.

Delegating thread safety

Method `getLocations()` returns a 'live' view of the vehicle locations.
If thread A calls `getLocations()` and thread B later modifies the location of some of the points, those changes are reflected in the Map returned earlier to thread A.

```
@ThreadSafe
public class DelegatingVehicleTracker {
    private final ConcurrentMap<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;
    }
```

DelegatingVehicleTracker sequence diagram



Independent state variables

Untitled Sheets
HOM
Threadsafe design
Confinement (revisited)
Java monitor pattern
Delegating thread safety

Delegation to more state variables is possible if these are independent. See the `VisualComponent` class with two List's: one for KeyListeners and one for MouseListeners.

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

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

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

Placing an object in a `CopyOnWriteArrayList`, safely publishes it to any thread that receives it from the collection.

When delegation fails

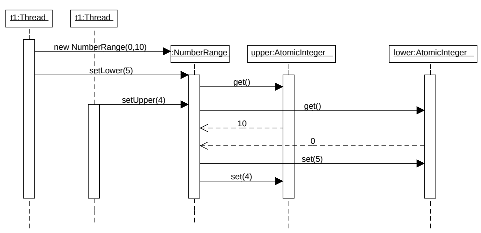
Untitled Sheets
HOM
Threadsafe design
Confinement (revisited)
Java monitor pattern
Delegating thread safety

Two atomic integers with an additional constraint:

```
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) {
        // Warning -- unsafe check-then-act
        if (i > upper.get())
            throw new IllegalArgumentException(
                  "can't set lower to " + i + " > upper");
        lower.set(i);
    }

    public void setUpper(int i) {
        // Warning -- unsafe check-then-act
```


NumberRange class does not sufficiently protect its invariants

Untitled Sheets
HOM
Threadsafe design
Confinement (revisited)
Java monitor pattern
Delegating thread safety



When delegation fails

Delegation threadsafety to two thread-safe variables: invariant is not preserved!
How could we make `NumberRange` thread-safe?

- Use locking to maintain the invariants by guarding upper and lower by a single lock

When delegation fails

Delegation threadsafety to two thread-safe variables: invariant is not preserved!
How could we make `NumberRange` thread-safe?

- Use locking to maintain the invariants by guarding upper and lower by a single lock
- avoid publishing lower and upper to avoid ruining it's invariants.

When delegation fails

Delegation threadsafety to two thread-safe variables: invariant is not preserved!
How could we make `NumberRange` thread-safe?

- Use locking to maintain the invariants by guarding upper and lower by a single lock
- avoid publishing lower and upper to avoid ruining it's invariants.

Definition

If a class is composed of multiple *independent* thread-safe state variables and has no operations that have any invalid state transitions, then it can delegate thread safety to the underlying state variables.

Publishing underlying state variables

- Under what conditions can you publish those state variables?

Publishing underlying state variables

- Under what conditions can you publish those state variables?
- It depends! on what invariants your class imposes on those variables.

Publishing underlying state variables

- Under what conditions can you publish those state variables?
- It depends! on what invariants your class imposes on those variables.
- Making the state variable `value` of the `Counter` class `public` (=publishing!) clients could change it into an invalid value.

Publishing underlying state variables

- Under what conditions can you publish those state variables?
- It depends! on what invariants your class imposes on those variables.
- Making the state variable `value` of the `Counter` class `public` (=publishing!) clients could change it into an invalid value.
- Don't publish if you don't need to!

Publishing underlying state variables

- Under what conditions can you publish those state variables?
- It depends! on what invariants your class imposes on those variables.
- Making the state variable `value` of the `Counter` class `public` (=publishing!) clients could change it into an invalid value.
- Don't publish if you don't need to!
- If there aren't any constraints, you can publish without compromising thread safety.

Publishing underlying state variables

- Under what conditions can you publish those state variables?
- It depends! on what invariants your class imposes on those variables.
- Making the state variable `value` of the `Counter` class `public` (=publishing!) clients could change it into an invalid value.
- Don't publish if you don't need to!
- If there aren't any constraints, you can publish without compromising thread safety.

Definition
If a state variable is thread-safe, does not participate in any invariants that constrain its value, and has no prohibited state transitions for any of its operations, then it can safely be published.

A SafePoint class


```
@ThreadSafe
public class SafePoint {
    @GuardedBy("this") private int x, y;

    private SafePoint(int[] a) {
        this(a[0], a[1]);
    }

    public SafePoint(SafePoint p) {
        this(p.get());
    }

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

    public synchronized int[] get() {
        return new int[]{x, y};
    }

    public synchronized void set(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```


The PublishingVehicleTracker class

- SafePoint returns both x and y values at once by returning an array without undermining thread safety.
- Note the use of a private constructor in `SafePoint`!

```
@ThreadSafe
public class PublishingVehicleTracker {
    private final Map<String, SafePoint> locations;
    private final Map<String, SafePoint> unmodifiableMap;

    public PublishingVehicleTracker(Map<String, SafePoint> locations)
        this.locations =
            new ConcurrentHashMap<String, SafePoint>(locations);
        this.unmodifiableMap =
            Collections.unmodifiableMap(this.locations);
    }

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

    public SafePoint getLocation(String id) {
        return locations.get(id);
```


The PublishingVehicleTracker class

- Delegation to the underlying `Concurrenthashmap` secures thread safety in `PublishingVehicleTracker`.
- `getLocations()` returns an unmodifiable copy of the underlying map.
- A caller of `getLocations()` could change the locations of the veicles but cannot add or remove vehicles themselves.
- If there are additional constraints on the valid values of vehicle locations `PublishingVehicleTracker` is not thread safety anymore.

Adding functionality

- We have a thread-safe class!!
- We want to add a new operation without underminig its threadsafety.
- Example: put-if-absent operation. Should be atomic.
- You do not own the Vector code, so use subclassing.

```java
@ThreadSafe
public class BetterVector <E> extends Vector<E> {
    // When extending a serializable class,
    // you should redefine serialVersionUID
    static final long serialVersionUID = -3963416950630760754L;

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


Client-side locking

- What if you don't know your collection type?
- E.g. by using the `Collections.synchronizedList` wrapper.
- Then use a "helper" class:

```java
@NotThreadSafe
class BadListHelper <E> {
    public List<E> list =
        Collections.synchronizedList(new ArrayList<E>());

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

Client-side locking - 2

- Why doesn't this work?
- putIfAbsent is synchronized!
- It uses the *wrong lock*! (*illusion of synchronization*)
- Use the same lock that the list uses ..!
- 'Check-then-act' also here:
- The documentation on Vector and on synchronized wrapper classes shows that they support client-side locking.

Client-side locking - 3

- OK now.

```java
@ThreadSafe
class GoodListHelper <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;
        }
    }
}
```

Composition

- Delegation instead of subclassing: often a better approach.
- It is assumed that the client after having constructed the `ImprovedList` will access the underlying list only through the `ImprovedList`.
- Uses the java monitor pattern.

```
@ThreadSafe
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) {
        boolean contains = list.contains(x);
        if (contains)
            list.add(x);
        return !contains;
    }
```