```
@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;
    }
}
```

LISTING 4.1. Simple thread-safe counter using the Java monitor pattern.

immutability, thread confinement, and locking is used to maintain thread safety, and which variables are guarded by which locks. To ensure that the class can be analyzed and maintained, document the synchronization policy.

4.1.1 Gathering synchronization requirements

Making a class thread-safe means ensuring that its invariants hold under concurrent access; this requires reasoning about its state. Objects and variables have a *state space*: the range of possible states they can take on. The smaller this state space, the easier it is to reason about. By using final fields wherever practical, you make it simpler to analyze the possible states an object can be in. (In the extreme case, immutable objects can only be in a single state.)

Many classes have invariants that identify certain states as *valid* or *invalid*. The `value` field in `Counter` is a `long`. The state space of a `long` ranges from `Long.MIN_VALUE` to `Long.MAX_VALUE`, but `Counter` places constraints on `value`; negative values are not allowed.

Similarly, operations may have postconditions that identify certain *state transitions* as invalid. If the current state of a `Counter` is 17, the *only* valid next state is 18. When the next state is derived from the current state, the operation is necessarily a compound action. Not all operations impose state transition constraints; when updating a variable that holds the current temperature, its previous state does not affect the computation.

Constraints placed on states or state transitions by invariants and postconditions create additional synchronization or encapsulation requirements. If certain states are invalid, then the underlying state variables must be encapsulated, otherwise client code could put the object into an invalid state. If an operation has invalid state transitions, it must be made atomic. On the other hand, if the class does not impose any such constraints, we may be able to relax encapsulation or serialization requirements to obtain greater flexibility or better performance.

Encapsulation simplifies making classes thread-safe by promoting *instance confinement*, often just called *confinement* [CPJ 2.3.3]. When an object is encapsulated within another object, all code paths that have access to the encapsulated object are known and can be therefore be analyzed more easily than if that object were accessible to the entire program. Combining confinement with an appropriate locking discipline can ensure that otherwise non-thread-safe objects are used in a thread-safe manner.

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

Confined objects must not escape their intended scope. An object may be confined to a class instance (such as a private class member), a lexical scope (such as a local variable), or a thread (such as an object that is passed from method to method within a thread, but not supposed to be shared across threads). Objects don't escape on their own, of course—they need help from the developer, who assists by publishing the object beyond its intended scope.

`PersonSet` in Listing 4.2 illustrates how confinement and locking can work together to make a class thread-safe even when its component state variables are not. The state of `PersonSet` is managed by a `HashSet`, which is not thread-safe. But because `mySet` is private and not allowed to escape, the `HashSet` is confined to the `PersonSet`. The only code paths that can access `mySet` are `addPerson` and `containsPerson`, and each of these acquires the lock on the `PersonSet`. All its state is guarded by its intrinsic lock, making `PersonSet` thread-safe.

```
@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);
    }
}
```

LISTING 4.2. Using confinement to ensure thread safety.

This example makes no assumptions about the thread-safety of `Person`, but if

```
@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;
    }
}
```

LISTING 4.5. Mutable point class similar to `java.awt.Point`.

counter, it is easy to see that `CountingFactorizer` is thread-safe. We could say that `CountingFactorizer` *delegates* its thread safety responsibilities to the `AtomicLong`: `CountingFactorizer` is thread-safe because `AtomicLong` is.[3]

4.3.1 Example: vehicle tracker using delegation

As a more substantial example of delegation, let's construct a version of the vehicle tracker that delegates to a thread-safe class. We store the locations in a `Map`, so we start with a thread-safe `Map` implementation, `ConcurrentHashMap`. We also store the location using an immutable `Point` class instead of `MutablePoint`, shown in Listing 4.6.

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

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

LISTING 4.6. Immutable `Point` class used by `DelegatingVehicleTracker`.

`Point` is thread-safe because it is immutable. Immutable values can be freely shared and published, so we no longer need to copy the locations when returning them.

3. If `count` were not `final`, the thread safety analysis of `CountingFactorizer` would be more complicated. If `CountingFactorizer` could modify `count` to reference a different `AtomicLong`, we would then have to ensure that this update was visible to all threads that might access the count, and that

DelegatingVehicleTracker in Listing 4.7 does not use any explicit synchronization; all access to state is managed by ConcurrentHashMap, and all the keys and values of the Map are immutable.

```
@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;
    }

    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);
    }
}
```

LISTING 4.7. Delegating thread safety to a ConcurrentHashMap.

If we had used the original MutablePoint class instead of Point, we would be breaking encapsulation by letting getLocations publish a reference to mutable state that is not thread-safe. Notice that we've changed the behavior of the vehicle tracker class slightly; while the monitor version returned a snapshot of the locations, the delegating version returns an unmodifiable but "live" view of the vehicle locations. This means that 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 to thread *A*. As we remarked earlier, this can be a benefit (more up-to-date data) or a liability (potentially inconsistent view of the fleet), depending on your requirements.

If an unchanging view of the fleet is required, getLocations could instead return a shallow copy of the locations map. Since the contents of the Map are immutable, only the structure of the Map, not the contents, must be copied, as

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

LISTING 4.13. Extending Vector to have a put-if-absent method.

4.4.1 Client-side locking

For an ArrayList wrapped with a Collections.synchronizedList wrapper, neither of these approaches—adding a method to the original class or extending the class—works because the client code does not even know the class of the List object returned from the synchronized wrapper factories. A third strategy is to extend the functionality of the class without extending the class itself by placing extension code in a "helper" class.

Listing 4.14 shows a failed attempt to create a helper class with an atomic put-if-absent operation for operating on a thread-safe List.

```
@NotThreadSafe
public class ListHelper<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;
    }
}
```

LISTING 4.14. Non-thread-safe attempt to implement put-if-absent. *Don't do this.*

Why wouldn't this work? After all, putIfAbsent is synchronized, right? The problem is that it synchronizes on the *wrong lock*. Whatever lock the List uses to guard its state, it sure isn't the lock on the ListHelper. ListHelper provides only the *illusion of synchronization*; the various list operations, while all synchronized, use different locks, which means that putIfAbsent is *not* atomic relative to other operations on the List. So there is no guarantee that another thread won't modify the list while putIfAbsent is executing.

To make this approach work, we have to use the *same* lock that the `List` uses by using *client-side locking* or *external locking*. Client-side locking entails guarding client code that uses some object X with the lock X uses to guard its own state. In order to use client-side locking, you must know what lock X uses.

The documentation for `Vector` and the synchronized wrapper classes states, albeit obliquely, that they support client-side locking, by using the intrinsic lock for the `Vector` or the wrapper collection (not the wrapped collection). Listing 4.15 shows a `putIfAbsent` operation on a thread-safe `List` that correctly uses client-side locking.

```
@ThreadSafe
public class ListHelper<E> {
    public 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;
        }
    }
}
```

LISTING 4.15. Implementing put-if-absent with client-side locking.

If extending a class to add another atomic operation is fragile because it distributes the locking code for a class over multiple classes in an object hierarchy, client-side locking is even more fragile because it entails putting locking code for class C into classes that are totally unrelated to C. Exercise care when using client-side locking on classes that do not commit to their locking strategy.

Client-side locking has a lot in common with class extension—they both couple the behavior of the derived class to the implementation of the base class. Just as extension violates encapsulation of implementation [EJ Item 14], client-side locking violates encapsulation of synchronization policy.

4.4.2 Composition

There is a less fragile alternative for adding an atomic operation to an existing class: *composition*. `ImprovedList` in Listing 4.16 implements the `List` operations by delegating them to an underlying `List` instance, and adds an atomic `putIfAbsent` method. (Like `Collections.synchronizedList` and other collections wrappers, `ImprovedList` assumes that once a list is passed to its constructor, the client will not use the underlying list directly again, accessing it only through the

```
@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;
    }

    public synchronized void clear() { list.clear(); }
    // ... similarly delegate other List methods
}
```

LISTING 4.16. Implementing put-if-absent using composition.

ImprovedList adds an additional level of locking using its own intrinsic lock. It does not care whether the underlying List is thread-safe, because it provides its own consistent locking that provides thread safety even if the List is not thread-safe or changes its locking implementation. While the extra layer of synchronization may add some small performance penalty,[7] the implementation in ImprovedList is less fragile than attempting to mimic the locking strategy of another object. In effect, we've used the Java monitor pattern to encapsulate an existing List, and this is guaranteed to provide thread safety so long as our class holds the only outstanding reference to the underlying List.

4.5 Documenting synchronization policies

Documentation is one of the most powerful (and, sadly, most underutilized) tools for managing thread safety. Users look to the documentation to find out if a class is thread-safe, and maintainers look to the documentation to understand the implementation strategy so they can maintain it without inadvertently compromising safety. Unfortunately, both of these constituencies usually find less information in the documentation than they'd like.

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

7. The penalty will be small because the synchronization on the underlying List is guaranteed to be uncontended and therefore fast; see Chapter 11.