the size of the array and uses the resulting value to retrieve or remove the last element.

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

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

LISTING 5.1. Compound actions on a `Vector` that may produce confusing results.

These methods seem harmless, and in a sense they are—they can't corrupt the `Vector`, no matter how many threads call them simultaneously. But the caller of these methods might have a different opinion. If thread *A* calls `getLast` on a `Vector` with ten elements, thread *B* calls `deleteLast` on the same `Vector`, and the operations are interleaved as shown in Figure 5.1, `getLast` throws `ArrayIndexOutOfBoundsException`. Between the call to `size` and the subsequent call to `get` in `getLast`, the `Vector` shrank and the index computed in the first step is no longer valid. This is perfectly consistent with the specification of `Vector`—it throws an exception if asked for a nonexistent element. But this is not what a caller expects `getLast` to do, even in the face of concurrent modification, unless perhaps the `Vector` was empty to begin with.

Because the synchronized collections commit to a synchronization policy that supports client-side locking,[1] it is possible to create new operations that are atomic with respect to other collection operations as long as we know which lock to use. The synchronized collection classes guard each method with the lock on the synchronized collection object itself. By acquiring the collection lock we can make `getLast` and `deleteLast` atomic, ensuring that the size of the `Vector` does not change between calling `size` and `get`, as shown in Listing 5.2.

The risk that the size of the list might change between a call to `size` and the corresponding call to `get` is also present when we iterate through the elements of a `Vector` as shown in Listing 5.3.

This iteration idiom relies on a leap of faith that other threads will not modify the `Vector` between the calls to `size` and `get`. In a single-threaded environment, this assumption is perfectly valid, but when other threads may concurrently modify the `Vector` it can lead to trouble. Just as with `getLast`, if another thread deletes an element while you are iterating through the `Vector` and the operations are interleaved unluckily, this iteration idiom throws `ArrayIndexOutOfBoundsException`.

1. This is documented only obliquely in the Java 5.0 Javadoc, as an example of the correct iteration idiom.

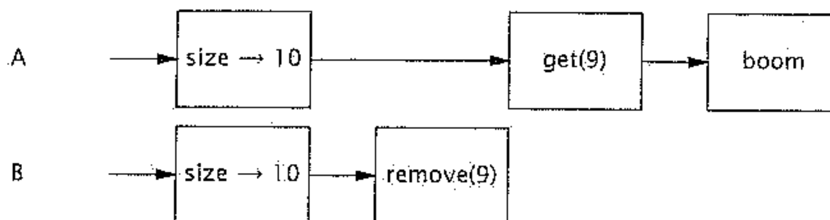

FIGURE 5.1. Interleaving of getLast and deleteLast that throws ArrayIndexOutOfBoundsException.

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

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

LISTING 5.2. Compound actions on Vector using client-side locking.

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

LISTING 5.3. Iteration that may throw ArrayIndexOutOfBoundsException.

Even though the iteration in Listing 5.3 can throw an exception, this doesn't mean Vector isn't thread-safe. The state of the Vector is still valid and the exception is in fact in conformance with its specification. However, that something as mundane as fetching the last element or iteration throw an exception is clearly undesirable.

The problem of unreliable iteration can again be addressed by client-side locking, at some additional cost to scalability. By holding the Vector lock for the duration of iteration, as shown in Listing 5.4, we prevent other threads from modifying the Vector while we are iterating it. Unfortunately, we also prevent other threads from accessing it at all during this time, impairing concurrency.

Summary of Part I

We've covered a lot of material so far! The following "concurrency cheat sheet" summarizes the main concepts and rules presented in Part I.

- *It's the mutable state, stupid.*[1]

  All concurrency issues boil down to coordinating access to mutable state. The less mutable state, the easier it is to ensure thread safety.

- *Make fields final unless they need to be mutable.*

- *Immutable objects are automatically thread-safe.*

  Immutable objects simplify concurrent programming tremendously. They are simpler and safer, and can be shared freely without locking or defensive copying.

- *Encapsulation makes it practical to manage the complexity.*

  You could write a thread-safe program with all data stored in global variables, but why would you want to? Encapsulating data within objects makes it easier to preserve their invariants; encapsulating synchronization within objects makes it easier to comply with their synchronization policy.

- *Guard each mutable variable with a lock.*

- *Guard all variables in an invariant with the same lock.*

- *Hold locks for the duration of compound actions.*

- *A program that accesses a mutable variable from multiple threads without synchronization is a broken program.*

- *Don't rely on clever reasoning about why you don't need to synchronize.*

- *Include thread safety in the design process—or explicitly document that your class is not thread-safe.*

- *Document your synchronization policy.*

1. During the 1992 U.S. presidential election, electoral strategist James Carville hung a sign in Bill Clinton's campaign headquarters reading "The economy, stupid", to keep the campaign on message.