Chapter 5

Building Blocks

The last chapter explored several techniques for constructing thread-safe classes, including delegating thread safety to existing thread-safe classes. Where practical, delegation is one of the most effective strategies for creating thread-safe classes: just let existing thread-safe classes manage all the state.

The platform libraries include a rich set of concurrent building blocks, such as thread-safe collections and a variety of *synchronizers* that can coordinate the control flow of cooperating threads. This chapter covers the most useful concurrent building blocks, especially those introduced in Java 5.0 and Java 6, and some patterns for using them to structure concurrent applications.

5.1 Synchronized collections

The *synchronized collection classes* include `Vector` and `Hashtable`, part of the original JDK, as well as their cousins added in JDK 1.2, the synchronized wrapper classes created by the `Collections.synchronizedXxx` factory methods. These classes achieve thread safety by encapsulating their state and synchronizing every public method so that only one thread at a time can access the collection state.

5.1.1 Problems with synchronized collections

The synchronized collections are thread-safe, but you may sometimes need to use additional client-side locking to guard compound actions. Common compound actions on collections include iteration (repeatedly fetch elements until the collection is exhausted), navigation (find the next element after this one according to some order), and conditional operations such as put-if-absent (check if a `Map` has a mapping for key K, and if not, add the mapping (K, V)). With a synchronized collection, these compound actions are still technically thread-safe even without client-side locking, but they may not behave as you might expect when other threads can concurrently modify the collection.

Listing 5.1 shows two methods that operate on a `Vector`, `getLast` and `deleteLast`, both of which are check-then-act sequences. Each calls `size` to determine

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.

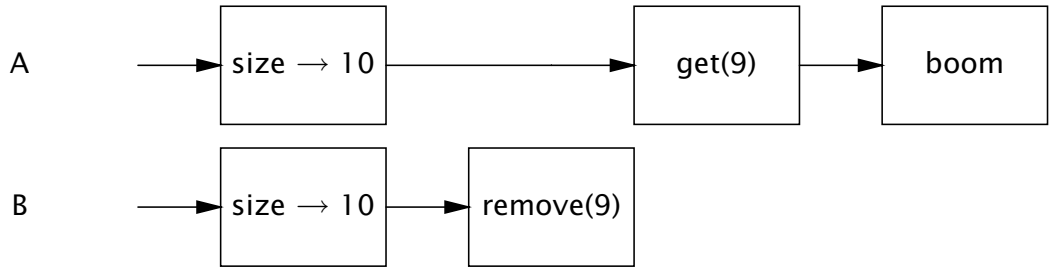

FIGURE 5.1. Interleaving of `getLast` and `deleteLast` that throws `ArrayIndexOutOfBoundsException`.

```
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.

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

LISTING 5.4. Iteration with client-side locking.

5.1.2 Iterators and ConcurrentModificationException

We use Vector for the sake of clarity in many of our examples, even though it is considered a "legacy" collection class. But the more "modern" collection classes do not eliminate the problem of compound actions. The standard way to iterate a Collection is with an Iterator, either explicitly or through the for-each loop syntax introduced in Java 5.0, but using iterators does not obviate the need to lock the collection during iteration if other threads can concurrently modify it. The iterators returned by the synchronized collections are not designed to deal with concurrent modification, and they are *fail-fast*—meaning that if they detect that the collection has changed since iteration began, they throw the unchecked ConcurrentModificationException.

These fail-fast iterators are not designed to be foolproof—they are designed to catch concurrency errors on a "good-faith-effort" basis and thus act only as early-warning indicators for concurrency problems. They are implemented by associating a modification count with the collection: if the modification count changes during iteration, hasNext or next throws ConcurrentModificationException. However, this check is done without synchronization, so there is a risk of seeing a stale value of the modification count and therefore that the iterator does not realize a modification has been made. This was a deliberate design tradeoff to reduce the performance impact of the concurrent modification detection code.[2]

Listing 5.5 illustrates iterating a collection with the for-each loop syntax. Internally, javac generates code that uses an Iterator, repeatedly calling hasNext and next to iterate the List. Just as with iterating the Vector, the way to prevent ConcurrentModificationException is to hold the collection lock for the duration of the iteration.

```
List<Widget> widgetList
    = Collections.synchronizedList(new ArrayList<Widget>());
...
// May throw ConcurrentModificationException
for (Widget w : widgetList)
    doSomething(w);
```

LISTING 5.5. Iterating a List with an Iterator.

2. ConcurrentModificationException can arise in single-threaded code as well; this happens when objects are removed from the collection directly rather than through Iterator.remove.

There are several reasons, however, why locking a collection during iteration may be undesirable. Other threads that need to access the collection will block until the iteration is complete; if the collection is large or the task performed for each element is lengthy, they could wait a long time. Also, if the collection is locked as in Listing 5.4, `doSomething` is being called with a lock held, which is a risk factor for deadlock (see Chapter 10). Even in the absence of starvation or deadlock risk, locking collections for significant periods of time hurts application scalability. The longer a lock is held, the more likely it is to be contended, and if many threads are blocked waiting for a lock throughput and CPU utilization can suffer (see Chapter 11).

An alternative to locking the collection during iteration is to clone the collection and iterate the copy instead. Since the clone is thread-confined, no other thread can modify it during iteration, eliminating the possibility of `ConcurrentModificationException`. (The collection still must be locked during the clone operation itself.) Cloning the collection has an obvious performance cost; whether this is a favorable tradeoff depends on many factors including the size of the collection, how much work is done for each element, the relative frequency of iteration compared to other collection operations, and responsiveness and throughput requirements.

5.1.3 Hidden iterators

While locking can prevent iterators from throwing `ConcurrentModificationException`, you have to remember to use locking everywhere a shared collection might be iterated. This is trickier than it sounds, as iterators are sometimes hidden, as in `HiddenIterator` in Listing 5.6. There is no explicit iteration in `HiddenIterator`, but the code in bold entails iteration just the same. The string concatenation gets turned by the compiler into a call to `StringBuilder.append(Object)`, which in turn invokes the collection's `toString` method—and the implementation of `toString` in the standard collections iterates the collection and calls `toString` on each element to produce a nicely formatted representation of the collection's contents.

The `addTenThings` method could throw `ConcurrentModificationException`, because the collection is being iterated by `toString` in the process of preparing the debugging message. Of course, the real problem is that `HiddenIterator` is not thread-safe; the `HiddenIterator` lock should be acquired before using `set` in the `println` call, but debugging and logging code commonly neglect to do this.

The real lesson here is that the greater the distance between the state and the synchronization that guards it, the more likely that someone will forget to use proper synchronization when accessing that state. If `HiddenIterator` wrapped the `HashSet` with a `synchronizedSet`, encapsulating the synchronization, this sort of error would not occur.

> Just as encapsulating an object's state makes it easier to preserve its invariants, encapsulating its synchronization makes it easier to enforce its synchronization policy.

```
public class HiddenIterator {
    @GuardedBy("this")
    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 addTenThings() {
        Random r = new Random();
        for (int i = 0; i < 10; i++)
            add(r.nextInt());
        System.out.println("DEBUG: added ten elements to " + set);
    }
}
```

LISTING 5.6. Iteration hidden within string concatenation. *Don't do this.*

Iteration is also indirectly invoked by the collection's `hashCode` and `equals` methods, which may be called if the collection is used as an element or key of another collection. Similarly, the `containsAll`, `removeAll`, and `retainAll` methods, as well as the constructors that take collections as arguments, also iterate the collection. All of these indirect uses of iteration can cause `ConcurrentModificationException`.

5.2 Concurrent collections

Java 5.0 improves on the synchronized collections by providing several *concurrent* collection classes. Synchronized collections achieve their thread safety by serializing all access to the collection's state. The cost of this approach is poor concurrency; when multiple threads contend for the collection-wide lock, throughput suffers.

The concurrent collections, on the other hand, are designed for concurrent access from multiple threads. Java 5.0 adds `ConcurrentHashMap`, a replacement for synchronized hash-based `Map` implementations, and `CopyOnWriteArrayList`, a replacement for synchronized `List` implementations for cases where traversal is the dominant operation. The new `ConcurrentMap` interface adds support for common compound actions such as put-if-absent, replace, and conditional remove.

Replacing synchronized collections with concurrent collections can offer dramatic scalability improvements with little risk.

Java 5.0 also adds two new collection types, `Queue` and `BlockingQueue`. A `Queue` is intended to hold a set of elements temporarily while they await processing. Several implementations are provided, including `ConcurrentLinkedQueue`, a

traditional FIFO queue, and `PriorityQueue`, a (non concurrent) priority ordered queue. `Queue` operations do not block; if the queue is empty, the retrieval operation returns `null`. While you can simulate the behavior of a `Queue` with a `List`—in fact, `LinkedList` also implements `Queue`—the `Queue` classes were added because eliminating the random-access requirements of `List` admits more efficient concurrent implementations.

`BlockingQueue` extends `Queue` to add blocking insertion and retrieval operations. If the queue is empty, a retrieval blocks until an element is available, and if the queue is full (for bounded queues) an insertion blocks until there is space available. Blocking queues are extremely useful in producer-consumer designs, and are covered in greater detail in Section 5.3.

Just as `ConcurrentHashMap` is a concurrent replacement for a synchronized hash-based `Map`, Java 6 adds `ConcurrentSkipListMap` and `ConcurrentSkipListSet`, which are concurrent replacements for a synchronized `SortedMap` or `SortedSet` (such as `TreeMap` or `TreeSet` wrapped with `synchronizedMap`).

5.2.1 `ConcurrentHashMap`

The synchronized collections classes hold a lock for the duration of each operation. Some operations, such as `HashMap.get` or `List.contains`, may involve more work than is initially obvious: traversing a hash bucket or list to find a specific object entails calling `equals` (which itself may involve a fair amount of computation) on a number of candidate objects. In a hash-based collection, if `hashCode` does not spread out hash values well, elements may be unevenly distributed among buckets; in the degenerate case, a poor hash function will turn a hash table into a linked list. Traversing a long list and calling `equals` on some or all of the elements can take a long time, and during that time no other thread can access the collection.

`ConcurrentHashMap` is a hash-based `Map` like `HashMap`, but it uses an entirely different locking strategy that offers better concurrency and scalability. Instead of synchronizing every method on a common lock, restricting access to a single thread at a time, it uses a finer-grained locking mechanism called *lock striping* (see Section 11.4.3) to allow a greater degree of shared access. Arbitrarily many reading threads can access the map concurrently, readers can access the map concurrently with writers, and a limited number of writers can modify the map concurrently. The result is far higher throughput under concurrent access, with little performance penalty for single-threaded access.

`ConcurrentHashMap`, along with the other concurrent collections, further improve on the synchronized collection classes by providing iterators that do not throw `ConcurrentModificationException`, thus eliminating the need to lock the collection during iteration. The iterators returned by `ConcurrentHashMap` are *weakly consistent* instead of fail-fast. A weakly consistent iterator can tolerate concurrent modification, traverses elements as they existed when the iterator was constructed, and may (but is not guaranteed to) reflect modifications to the collection after the construction of the iterator.

As with all improvements, there are still a few tradeoffs. The semantics of methods that operate on the entire `Map`, such as `size` and `isEmpty`, have been

slightly weakened to reflect the concurrent nature of the collection. Since the result of `size` could be out of date by the time it is computed, it is really only an estimate, so `size` is allowed to return an approximation instead of an exact count. While at first this may seem disturbing, in reality methods like `size` and `isEmpty` are far less useful in concurrent environments because these quantities are moving targets. So the requirements for these operations were weakened to enable performance optimizations for the most important operations, primarily `get`, `put`, `containsKey`, and `remove`.

The one feature offered by the synchronized `Map` implementations but not by `ConcurrentHashMap` is the ability to lock the map for exclusive access. With `Hashtable` and `synchronizedMap`, acquiring the `Map` lock prevents any other thread from accessing it. This might be necessary in unusual cases such as adding several mappings atomically, or iterating the `Map` several times and needing to see the same elements in the same order. On the whole, though, this is a reasonable tradeoff: concurrent collections should be expected to change their contents continuously.

Because it has so many advantages and so few disadvantages compared to `Hashtable` or `synchronizedMap`, replacing synchronized `Map` implementations with `ConcurrentHashMap` in most cases results only in better scalability. Only if your application needs to lock the map for exclusive access[3] is `ConcurrentHashMap` not an appropriate drop-in replacement.

5.2.2 Additional atomic `Map` operations

Since a `ConcurrentHashMap` cannot be locked for exclusive access, we cannot use client-side locking to create new atomic operations such as put-if-absent, as we did for `Vector` in Section 4.4.1. Instead, a number of common compound operations such as put-if-absent, remove-if-equal, and replace-if-equal are implemented as atomic operations and specified by the `ConcurrentMap` interface, shown in Listing 5.7. If you find yourself adding such functionality to an existing synchronized `Map` implementation, it is probably a sign that you should consider using a `ConcurrentMap` instead.

5.2.3 `CopyOnWriteArrayList`

`CopyOnWriteArrayList` is a concurrent replacement for a synchronized `List` that offers better concurrency in some common situations and eliminates the need to lock or copy the collection during iteration. (Similarly, `CopyOnWriteArraySet` is a concurrent replacement for a synchronized `Set`.)

The copy-on-write collections derive their thread safety from the fact that as long as an effectively immutable object is properly published, no further synchronization is required when accessing it. They implement mutability by creating and republishing a new copy of the collection every time it is modified. Iterators for the copy-on-write collections retain a reference to the backing array that was current at the start of iteration, and since this will never change, they need to

3. Or if you are relying on the synchronization side effects of the synchronized `Map` implementations.

```
public interface ConcurrentMap<K,V> extends Map<K,V> {
    // Insert into map only if no value is mapped from K
    V putIfAbsent(K key, V value);

    // Remove only if K is mapped to V
    boolean remove(K key, V value);

    // Replace value only if K is mapped to oldValue
    boolean replace(K key, V oldValue, V newValue);

    // Replace value only if K is mapped to some value
    V replace(K key, V newValue);
}
```

LISTING 5.7. `ConcurrentMap` interface.

synchronize only briefly to ensure visibility of the array contents. As a result, multiple threads can iterate the collection without interference from one another or from threads wanting to modify the collection. The iterators returned by the copy-on-write collections do not throw `ConcurrentModificationException` and return the elements exactly as they were at the time the iterator was created, regardless of subsequent modifications.

Obviously, there is some cost to copying the backing array every time the collection is modified, especially if the collection is large; the copy-on-write collections are reasonable to use only when iteration is far more common than modification. This criterion exactly describes many event-notification systems: delivering a notification requires iterating the list of registered listeners and calling each one of them, and in most cases registering or unregistering an event listener is far less common than receiving an event notification. (See [CPJ 2.4.4] for more information on copy-on-write.)

5.3 Blocking queues and the producer-consumer pattern

Blocking queues provide blocking `put` and `take` methods as well as the timed equivalents `offer` and `poll`. If the queue is full, `put` blocks until space becomes available; if the queue is empty, `take` blocks until an element is available. Queues can be bounded or unbounded; unbounded queues are never full, so a `put` on an unbounded queue never blocks.

Blocking queues support the *producer-consumer* design pattern. A producer-consumer design separates the identification of work to be done from the execution of that work by placing work items on a "to do" list for later processing, rather than processing them immediately as they are identified. The producer-consumer pattern simplifies development because it removes code dependencies between producer and consumer classes, and simplifies workload management

by decoupling activities that may produce or consume data at different or variable rates.

In a producer-consumer design built around a blocking queue, producers place data onto the queue as it becomes available, and consumers retrieve data from the queue when they are ready to take the appropriate action. Producers don't need to know anything about the identity or number of consumers, or even whether they are the only producer—all they have to do is place data items on the queue. Similarly, consumers need not know who the producers are or where the work came from. `BlockingQueue` simplifies the implementation of producer-consumer designs with any number of producers and consumers. One of the most common producer-consumer designs is a thread pool coupled with a work queue; this pattern is embodied in the `Executor` task execution framework that is the subject of Chapters 6 and 8.

The familiar division of labor for two people washing the dishes is an example of a producer-consumer design: one person washes the dishes and places them in the dish rack, and the other person retrieves the dishes from the rack and dries them. In this scenario, the dish rack acts as a blocking queue; if there are no dishes in the rack, the consumer waits until there are dishes to dry, and if the rack fills up, the producer has to stop washing until there is more space. This analogy extends to multiple producers (though there may be contention for the sink) and multiple consumers; each worker interacts only with the dish rack. No one needs to know how many producers or consumers there are, or who produced a given item of work.

The labels "producer" and "consumer" are relative; an activity that acts as a consumer in one context may act as a producer in another. Drying the dishes "consumes" clean wet dishes and "produces" clean dry dishes. A third person wanting to help might put away the dry dishes, in which case the drier is both a consumer and a producer, and there are now two shared work queues (each of which may block the drier from proceeding.)

Blocking queues simplify the coding of consumers, since `take` blocks until data is available. If the producers don't generate work fast enough to keep the consumers busy, the consumers just wait until more work is available. Sometimes this is perfectly acceptable (as in a server application when no client is requesting service), and sometimes it indicates that the ratio of producer threads to consumer threads should be adjusted to achieve better utilization (as in a web crawler or other application in which there is effectively infinite work to do).

If the producers consistently generate work faster than the consumers can process it, eventually the application will run out of memory because work items will queue up without bound. Again, the blocking nature of `put` greatly simplifies coding of producers; if we use a *bounded queue*, then when the queue fills up the producers block, giving the consumers time to catch up because a blocked producer cannot generate more work.

Blocking queues also provide an `offer` method, which returns a failure status if the item cannot be enqueued. This enables you to create more flexible policies for dealing with overload, such as shedding load, serializing excess work items and writing them to disk, reducing the number of producer threads, or throttling producers in some other manner.

Bounded queues are a powerful resource management tool for building reliable applications: they make your program more robust to overload by throttling activities that threaten to produce more work than can be handled.

While the producer-consumer pattern enables producer and consumer *code* to be decoupled from each other, their *behavior* is still coupled indirectly through the shared work queue. It is tempting to assume that the consumers will always keep up, so that you need not place any bounds on the size of work queues, but this is a prescription for rearchitecting your system later. *Build resource management into your design early using blocking queues—it is a lot easier to do this up front than to retrofit it later.* Blocking queues make this easy for a number of situations, but if blocking queues don't fit easily into your design, you can create other blocking data structures using `Semaphore` (see Section 5.5.3).

The class library contains several implementations of `BlockingQueue`. `LinkedBlockingQueue` and `ArrayBlockingQueue` are FIFO queues, analogous to `LinkedList` and `ArrayList` but with better concurrent performance than a synchronized `List`. `PriorityBlockingQueue` is a priority-ordered queue, which is useful when you want to process elements in an order other than FIFO. Just like other sorted collections, `PriorityBlockingQueue` can compare elements according to their natural order (if they implement `Comparable`) or using a `Comparator`.

The last `BlockingQueue` implementation, `SynchronousQueue`, is not really a queue at all, in that it maintains no storage space for queued elements. Instead, it maintains a list of queued *threads* waiting to enqueue or dequeue an element. In the dish-washing analogy, this would be like having no dish rack, but instead handing the washed dishes directly to the next available dryer. While this may seem a strange way to implement a queue, it reduces the latency associated with moving data from producer to consumer because the work can be handed off directly. (In a traditional queue, the enqueue and dequeue operations must complete sequentially before a unit of work can be handed off.) The direct handoff also feeds back more information about the state of the task to the producer; when the handoff is accepted, it knows a consumer has taken responsibility for it, rather than simply letting it sit on a queue somewhere—much like the difference between handing a document to a colleague and merely putting it in her mailbox and hoping she gets it soon. Since a `SynchronousQueue` has no storage capacity, `put` and `take` will block unless another thread is already waiting to participate in the handoff. Synchronous queues are generally suitable only when there are enough consumers that there nearly always will be one ready to take the handoff.

5.3.1 Example: desktop search

One type of program that is amenable to decomposition into producers and consumers is an agent that scans local drives for documents and indexes them for later searching, similar to Google Desktop or the Windows Indexing service. `DiskCrawler` in Listing 5.8 shows a producer task that searches a file hierarchy

for files meeting an indexing criterion and puts their names on the work queue; `Indexer` in Listing 5.8 shows the consumer task that takes file names from the queue and indexes them.

The producer-consumer pattern offers a thread-friendly means of decomposing the desktop search problem into simpler components. Factoring file-crawling and indexing into separate activities results in code that is more readable and reusable than with a monolithic activity that does both; each of the activities has only a single task to do, and the blocking queue handles all the flow control, so the code for each is simpler and clearer.

The producer-consumer pattern also enables several performance benefits. Producers and consumers can execute concurrently; if one is I/O-bound and the other is CPU-bound, executing them concurrently yields better overall throughput than executing them sequentially. If the producer and consumer activities are parallelizable to different degrees, tightly coupling them reduces parallelizability to that of the less parallelizable activity.

Listing 5.9 starts several crawlers and indexers, each in their own thread. As written, the consumer threads never exit, which prevents the program from terminating; we examine several techniques for addressing this problem in Chapter 7. While this example uses explicitly managed threads, many producer-consumer designs can be expressed using the `Executor` task execution framework, which itself uses the producer-consumer pattern.

5.3.2 Serial thread confinement

The blocking queue implementations in `java.util.concurrent` all contain sufficient internal synchronization to safely publish objects from a producer thread to the consumer thread.

For mutable objects, producer-consumer designs and blocking queues facilitate *serial thread confinement* for handing off ownership of objects from producers to consumers. A thread-confined object is owned exclusively by a single thread, but that ownership can be "transferred" by publishing it safely where only one other thread will gain access to it and ensuring that the publishing thread does not access it after the handoff. The safe publication ensures that the object's state is visible to the new owner, and since the original owner will not touch it again, it is now confined to the new thread. The new owner may modify it freely since it has exclusive access.

Object pools exploit serial thread confinement, "lending" an object to a requesting thread. As long as the pool contains sufficient internal synchronization to publish the pooled object safely, and as long as the clients do not themselves publish the pooled object or use it after returning it to the pool, ownership can be transferred safely from thread to thread.

One could also use other publication mechanisms for transferring ownership of a mutable object, but it is necessary to ensure that only one thread receives the object being handed off. Blocking queues make this easy; with a little more work, it could also done with the atomic `remove` method of `ConcurrentMap` or the `compareAndSet` method of `AtomicReference`.

```
public class FileCrawler implements Runnable {
    private final BlockingQueue<File> fileQueue;
    private final FileFilter fileFilter;
    private final File root;
    ...
    public void run() {
        try {
            crawl(root);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    private void crawl(File root) throws InterruptedException {
        File[] entries = root.listFiles(fileFilter);
        if (entries != null) {
            for (File entry : entries)
                if (entry.isDirectory())
                    crawl(entry);
                else if (!alreadyIndexed(entry))
                    fileQueue.put(entry);
        }
    }
}

public class Indexer implements Runnable {
    private final BlockingQueue<File> queue;

    public Indexer(BlockingQueue<File> queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            while (true)
                indexFile(queue.take());
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

Listing 5.8. Producer and consumer tasks in a desktop search application.

```
public static void startIndexing(File[] roots) {
    BlockingQueue<File> queue = new LinkedBlockingQueue<File>(BOUND);
    FileFilter filter = new FileFilter() {
        public boolean accept(File file) { return true; }
    };

    for (File root : roots)
        new Thread(new FileCrawler(queue, filter, root)).start();

    for (int i = 0; i < N_CONSUMERS; i++)
        new Thread(new Indexer(queue)).start();
}
```

Listing 5.9. Starting the desktop search.

5.3.3 Deques and work stealing

Java 6 also adds another two collection types, `Deque` (pronounced "deck") and `BlockingDeque`, that extend `Queue` and `BlockingQueue`. A `Deque` is a double-ended queue that allows efficient insertion and removal from both the head and the tail. Implementations include `ArrayDeque` and `LinkedBlockingDeque`.

Just as blocking queues lend themselves to the producer-consumer pattern, deques lend themselves to a related pattern called *work stealing*. A producer-consumer design has one shared work queue for all consumers; in a work stealing design, every consumer has its own deque. If a consumer exhausts the work in its own deque, it can steal work from the *tail* of someone else's deque. Work stealing can be more scalable than a traditional producer-consumer design because workers don't contend for a shared work queue; most of the time they access only their own deque, reducing contention. When a worker has to access another's queue, it does so from the tail rather than the head, further reducing contention.

Work stealing is well suited to problems in which consumers are also producers—when performing a unit of work is likely to result in the identification of more work. For example, processing a page in a web crawler usually results in the identification of new pages to be crawled. Similarly, many graph-exploring algorithms, such as marking the heap during garbage collection, can be efficiently parallelized using work stealing. When a worker identifies a new unit of work, it places it at the end of its own deque (or alternatively, in a *work sharing* design, on that of another worker); when its deque is empty, it looks for work at the end of someone else's deque, ensuring that each worker stays busy.

5.4 Blocking and interruptible methods

Threads may *block*, or pause, for several reasons: waiting for I/O completion, waiting to acquire a lock, waiting to wake up from `Thread.sleep`, or waiting for the result of a computation in another thread. When a thread blocks, it is usually suspended and placed in one of the blocked thread states (`BLOCKED`, `WAITING`, or

TIMED_WAITING). The distinction between a blocking operation and an ordinary operation that merely takes a long time to finish is that a blocked thread must wait for an event that is beyond its control before it can proceed—the I/O completes, the lock becomes available, or the external computation finishes. When that external event occurs, the thread is placed back in the RUNNABLE state and becomes eligible again for scheduling.

The put and take methods of BlockingQueue throw the checked InterruptedException, as do a number of other library methods such as Thread.sleep. When a method can throw InterruptedException, it is telling you that it is a blocking method, and further that if it is *interrupted*, it will make an effort to stop blocking early.

Thread provides the interrupt method for interrupting a thread and for querying whether a thread has been interrupted. Each thread has a boolean property that represents its interrupted status; interrupting a thread sets this status.

Interruption is a *cooperative* mechanism. One thread cannot force another to stop what it is doing and do something else; when thread *A* interrupts thread *B*, *A* is merely requesting that *B* stop what it is doing when it gets to a convenient stopping point—if it feels like it. While there is nothing in the API or language specification that demands any specific application-level semantics for interruption, the most sensible use for interruption is to cancel an activity. Blocking methods that are responsive to interruption make it easier to cancel long-running activities on a timely basis.

When your code calls a method that throws InterruptedException, then your method is a blocking method too, and must have a plan for responding to interruption. For library code, there are basically two choices:

Propagate the InterruptedException. This is often the most sensible policy if you can get away with it—just propagate the InterruptedException to your caller. This could involve not catching InterruptedException, or catching it and throwing it again after performing some brief activity-specific cleanup.

Restore the interrupt. Sometimes you cannot throw InterruptedException, for instance when your code is part of a Runnable. In these situations, you must catch InterruptedException and restore the interrupted status by calling interrupt on the current thread, so that code higher up the call stack can see that an interrupt was issued, as demonstrated in Listing 5.10.

You can get much more sophisticated with interruption, but these two approaches should work in the vast majority of situations. But there is one thing you should *not* do with InterruptedException—catch it and do nothing in response. This deprives code higher up on the call stack of the opportunity to act on the interruption, because the evidence that the thread was interrupted is lost. *The only situation in which it is acceptable to swallow an interrupt is when you are extending* Thread *and therefore control all the code higher up on the call stack.* Cancellation and interruption are covered in greater detail in Chapter 7.

```
public class TaskRunnable implements Runnable {
    BlockingQueue<Task> queue;
    ...
    public void run() {
        try {
            processTask(queue.take());
        } catch (InterruptedException e) {
            // restore interrupted status
            Thread.currentThread().interrupt();
        }
    }
}
```

LISTING 5.10. Restoring the interrupted status so as not to swallow the interrupt.

5.5 Synchronizers

Blocking queues are unique among the collections classes: not only do they act as containers for objects, but they can also coordinate the control flow of producer and consumer threads because `take` and `put` block until the queue enters the desired state (not empty or not full).

A *synchronizer* is any object that coordinates the control flow of threads based on its state. Blocking queues can act as synchronizers; other types of synchronizers include semaphores, barriers, and latches. There are a number of synchronizer classes in the platform library; if these do not meet your needs, you can also create your own using the mechanisms described in Chapter 14.

All synchronizers share certain structural properties: they encapsulate state that determines whether threads arriving at the synchronizer should be allowed to pass or forced to wait, provide methods to manipulate that state, and provide methods to wait efficiently for the synchronizer to enter the desired state.

5.5.1 Latches

A *latch* is a synchronizer that can delay the progress of threads until it reaches its *terminal* state [CPJ 3.4.2]. A latch acts as a gate: until the latch reaches the terminal state the gate is closed and no thread can pass, and in the terminal state the gate opens, allowing all threads to pass. Once the latch reaches the terminal state, it cannot change state again, so it remains open forever. Latches can be used to ensure that certain activities do not proceed until other one-time activities complete, such as:

- Ensuring that a computation does not proceed until resources it needs have been initialized. A simple binary (two-state) latch could be used to indicate "Resource *R* has been initialized", and any activity that requires *R* would wait first on this latch.

- Ensuring that a service does not start until other services on which it depends have started. Each service would have an associated binary latch; starting service *S* would involve first waiting on the latches for other services on which *S* depends, and then releasing the *S* latch after startup completes so any services that depend on *S* can then proceed.
- Waiting until all the parties involved in an activity, for instance the players in a multi-player game, are ready to proceed. In this case, the latch reaches the terminal state after all the players are ready.

`CountDownLatch` is a flexible latch implementation that can be used in any of these situations; it allows one or more threads to wait for a set of events to occur. The latch state consists of a counter initialized to a positive number, representing the number of events to wait for. The `countDown` method decrements the counter, indicating that an event has occurred, and the `await` methods wait for the counter to reach zero, which happens when all the events have occurred. If the counter is nonzero on entry, `await` blocks until the counter reaches zero, the waiting thread is interrupted, or the wait times out.

`TestHarness` in Listing 5.11 illustrates two common uses for latches. `TestHarness` creates a number of threads that run a given task concurrently. It uses two latches, a "starting gate" and an "ending gate". The starting gate is initialized with a count of one; the ending gate is initialized with a count equal to the number of worker threads. The first thing each worker thread does is wait on the starting gate; this ensures that none of them starts working until they all are ready to start. The last thing each does is count down on the ending gate; this allows the master thread to wait efficiently until the last of the worker threads has finished, so it can calculate the elapsed time.

Why did we bother with the latches in `TestHarness` instead of just starting the threads immediately after they are created? Presumably, we wanted to measure how long it takes to run a task *n* times *concurrently*. If we simply created and started the threads, the threads started earlier would have a "head start" on the later threads, and the degree of contention would vary over time as the number of active threads increased or decreased. Using a starting gate allows the master thread to release all the worker threads at once, and the ending gate allows the master thread to wait for the *last* thread to finish rather than waiting sequentially for each thread to finish.

5.5.2 `FutureTask`

`FutureTask` also acts like a latch. (`FutureTask` implements `Future`, which describes an abstract result-bearing computation [CPJ 4.3.3].) A computation represented by a `FutureTask` is implemented with a `Callable`, the result-bearing equivalent of `Runnable`, and can be in one of three states: waiting to run, running, or completed. Completion subsumes all the ways a computation can complete, including normal completion, cancellation, and exception. Once a `FutureTask` enters the completed state, it stays in that state forever.

The behavior of `Future.get` depends on the state of the task. If it is completed, `get` returns the result immediately, and otherwise blocks until the task transitions

```
public class TestHarness {
    public long timeTasks(int nThreads, final Runnable task)
            throws InterruptedException {
        final CountDownLatch startGate = new CountDownLatch(1);
        final CountDownLatch endGate = new CountDownLatch(nThreads);

        for (int i = 0; i < nThreads; i++) {
            Thread t = new Thread() {
                public void run() {
                    try {
                        startGate.await();
                        try {
                            task.run();
                        } finally {
                            endGate.countDown();
                        }
                    } catch (InterruptedException ignored) { }
                }
            };
            t.start();
        }

        long start = System.nanoTime();
        startGate.countDown();
        endGate.await();
        long end = System.nanoTime();
        return end-start;
    }
}
```

LISTING 5.11. Using `CountDownLatch` for starting and stopping threads in timing tests.

to the completed state and then returns the result or throws an exception. FutureTask conveys the result from the thread executing the computation to the thread(s) retrieving the result; the specification of FutureTask guarantees that this transfer constitutes a safe publication of the result.

FutureTask is used by the Executor framework to represent asynchronous tasks, and can also be used to represent any potentially lengthy computation that can be started before the results are needed. Preloader in Listing 5.12 uses FutureTask to perform an expensive computation whose results are needed later; by starting the computation early, you reduce the time you would have to wait later when you actually need the results.

```
public class Preloader {
    private final FutureTask<ProductInfo> future =
        new FutureTask<ProductInfo>(new Callable<ProductInfo>() {
            public ProductInfo call() throws DataLoadException {
                return loadProductInfo();
            }
        });
    private final Thread thread = new Thread(future);

    public void start() { thread.start(); }

    public ProductInfo get()
            throws DataLoadException, InterruptedException {
        try {
            return future.get();
        } catch (ExecutionException e) {
            Throwable cause = e.getCause();
            if (cause instanceof DataLoadException)
                throw (DataLoadException) cause;
            else
                throw launderThrowable(cause);
        }
    }
}
```

LISTING 5.12. Using FutureTask to preload data that is needed later.

Preloader creates a FutureTask that describes the task of loading product information from a database and a thread in which the computation will be performed. It provides a start method to start the thread, since it is inadvisable to start a thread from a constructor or static initializer. When the program later needs the ProductInfo, it can call get, which returns the loaded data if it is ready, or waits for the load to complete if not.

Tasks described by Callable can throw checked and unchecked exceptions, and any code can throw an Error. Whatever the task code may throw, it is

wrapped in an ExecutionException and rethrown from Future.get. This complicates code that calls get, not only because it must deal with the possibility of ExecutionException (and the unchecked CancellationException), but also because the cause of the ExecutionException is returned as a Throwable, which is inconvenient to deal with.

When get throws an ExecutionException in Preloader, the cause will fall into one of three categories: a checked exception thrown by the Callable, a RuntimeException, or an Error. We must handle each of these cases separately, but we will use the launderThrowable utility method in Listing 5.13 to encapsulate some of the messier exception-handling logic. Before calling launderThrowable, Preloader tests for the known checked exceptions and rethrows them. That leaves only unchecked exceptions, which Preloader handles by calling launderThrowable and throwing the result. If the Throwable passed to launderThrowable is an Error, launderThrowable rethrows it directly; if it is not a RuntimeException, it throws an IllegalStateException to indicate a logic error. That leaves only RuntimeException, which launderThrowable returns to its caller, and which the caller generally rethrows.

```
/** If the Throwable is an Error, throw it; if it is a
 *  RuntimeException return it, otherwise throw IllegalStateException
 */
public static RuntimeException launderThrowable(Throwable t) {
    if (t instanceof RuntimeException)
        return (RuntimeException) t;
    else if (t instanceof Error)
        throw (Error) t;
    else
        throw new IllegalStateException("Not unchecked", t);
}
```

LISTING 5.13. Coercing an unchecked Throwable to a RuntimeException.

5.5.3 Semaphores

Counting semaphores are used to control the number of activities that can access a certain resource or perform a given action at the same time [CPJ 3.4.1]. Counting semaphores can be used to implement resource pools or to impose a bound on a collection.

A Semaphore manages a set of virtual *permits*; the initial number of permits is passed to the Semaphore constructor. Activities can acquire permits (as long as some remain) and release permits when they are done with them. If no permit is available, acquire blocks until one is (or until interrupted or the operation times out). The release method returns a permit to the semaphore.[4] A degenerate case

4. The implementation has no actual permit objects, and Semaphore does not associate dispensed permits with threads, so a permit acquired in one thread can be released from another thread. You

of a counting semaphore is a binary semaphore, a `Semaphore` with an initial count of one. A binary semaphore can be used as a *mutex* with nonreentrant locking semantics; whoever holds the sole permit holds the mutex.

Semaphores are useful for implementing resource pools such as database connection pools. While it is easy to construct a fixed-sized pool that fails if you request a resource from an empty pool, what you really want is to *block* if the pool is empty and unblock when it becomes nonempty again. If you initialize a `Semaphore` to the pool size, `acquire` a permit before trying to fetch a resource from the pool, and `release` the permit after putting a resource back in the pool, `acquire` blocks until the pool becomes nonempty. This technique is used in the bounded buffer class in Chapter 12. (An easier way to construct a blocking object pool would be to use a `BlockingQueue` to hold the pooled resources.)

Similarly, you can use a `Semaphore` to turn any collection into a blocking bounded collection, as illustrated by `BoundedHashSet` in Listing 5.14. The semaphore is initialized to the desired maximum size of the collection. The `add` operation acquires a permit before adding the item into the underlying collection. If the underlying `add` operation does not actually add anything, it releases the permit immediately. Similarly, a successful `remove` operation releases a permit, enabling more elements to be added. The underlying `Set` implementation knows nothing about the bound; this is handled by `BoundedHashSet`.

5.5.4 Barriers

We have seen how latches can facilitate starting a group of related activities or waiting for a group of related activities to complete. Latches are single-use objects; once a latch enters the terminal state, it cannot be reset.

Barriers are similar to latches in that they block a group of threads until some event has occurred [CPJ 4.4.3]. The key difference is that with a barrier, all the threads must come together at a barrier point *at the same time* in order to proceed. Latches are for waiting for *events*; barriers are for waiting for *other threads*. A barrier implements the protocol some families use to rendezvous during a day at the mall: "Everyone meet at McDonald's at 6:00; once you get there, stay there until everyone shows up, and then we'll figure out what we're doing next."

`CyclicBarrier` allows a fixed number of parties to rendezvous repeatedly at a *barrier point* and is useful in parallel iterative algorithms that break down a problem into a fixed number of independent subproblems. Threads call `await` when they reach the barrier point, and `await` blocks until *all* the threads have reached the barrier point. If all threads meet at the barrier point, the barrier has been successfully passed, in which case all threads are released and the barrier is reset so it can be used again. If a call to `await` times out or a thread blocked in `await` is interrupted, then the barrier is considered *broken* and all outstanding calls to `await` terminate with `BrokenBarrierException`. If the barrier is successfully passed, `await` returns a unique arrival index for each thread, which can be used to "elect" a leader that takes some special action in the next iteration. `CyclicBar-`

can think of `acquire` as consuming a permit and `release` as creating one; a `Semaphore` is not limited to the number of permits it was created with.

```
public class BoundedHashSet<T> {
    private final Set<T> set;
    private final Semaphore sem;

    public BoundedHashSet(int bound) {
        this.set = Collections.synchronizedSet(new HashSet<T>());
        sem = new Semaphore(bound);
    }

    public boolean add(T o) throws InterruptedException {
        sem.acquire();
        boolean wasAdded = false;
        try {
            wasAdded = set.add(o);
            return wasAdded;
        }
        finally {
            if (!wasAdded)
                sem.release();
        }
    }

    public boolean remove(Object o) {
        boolean wasRemoved = set.remove(o);
        if (wasRemoved)
            sem.release();
        return wasRemoved;
    }
}
```

Listing 5.14. Using Semaphore to bound a collection.

rier also lets you pass a *barrier action* to the constructor; this is a Runnable that is executed (in one of the subtask threads) when the barrier is successfully passed but before the blocked threads are released.

Barriers are often used in simulations, where the work to calculate one step can be done in parallel but all the work associated with a given step must complete before advancing to the next step. For example, in n-body particle simulations, each step calculates an update to the position of each particle based on the locations and other attributes of the other particles. Waiting on a barrier between each update ensures that all updates for step k have completed before moving on to step $k + 1$.

CellularAutomata in Listing 5.15 demonstrates using a barrier to compute a cellular automata simulation, such as Conway's Life game (Gardner, 1970). When parallelizing a simulation, it is generally impractical to assign a separate thread to each element (in the case of Life, a cell); this would require too many threads, and the overhead of coordinating them would dwarf the computation. Instead, it makes sense to *partition* the problem into a number of subparts, let each thread solve a subpart, and then merge the results. CellularAutomata partitions the board into N_{cpu} parts, where N_{cpu} is the number of CPUs available, and assigns each part to a thread.[5] At each step, the worker threads calculate new values for all the cells in their part of the board. When all worker threads have reached the barrier, the barrier action commits the new values to the data model. After the barrier action runs, the worker threads are released to compute the next step of the calculation, which includes consulting an isDone method to determine whether further iterations are required.

Another form of barrier is Exchanger, a two-party barrier in which the parties exchange data at the barrier point [CPJ 3.4.3]. Exchangers are useful when the parties perform asymmetric activities, for example when one thread fills a buffer with data and the other thread consumes the data from the buffer; these threads could use an Exchanger to meet and exchange a full buffer for an empty one. When two threads exchange objects via an Exchanger, the exchange constitutes a safe publication of both objects to the other party.

The timing of the exchange depends on the responsiveness requirements of the application. The simplest approach is that the filling task exchanges when the buffer is full, and the emptying task exchanges when the buffer is empty; this minimizes the number of exchanges but can delay processing of some data if the arrival rate of new data is unpredictable. Another approach would be that the filler exchanges when the buffer is full, but also when the buffer is partially filled and a certain amount of time has elapsed.

5.6 Building an efficient, scalable result cache

Nearly every server application uses some form of caching. Reusing the results of a previous computation can reduce latency and increase throughput, at the cost

5. For computational problems like this that do no I/O and access no shared data, N_{cpu} or $N_{cpu} + 1$ threads yield optimal throughput; more threads do not help, and may in fact degrade performance as the threads compete for CPU and memory resources.

```
public class CellularAutomata {
    private final Board mainBoard;
    private final CyclicBarrier barrier;
    private final Worker[] workers;

    public CellularAutomata(Board board) {
        this.mainBoard = board;
        int count = Runtime.getRuntime().availableProcessors();
        this.barrier = new CyclicBarrier(count,
                new Runnable() {
                    public void run() {
                        mainBoard.commitNewValues();
                    }});
        this.workers = new Worker[count];
        for (int i = 0; i < count; i++)
            workers[i] = new Worker(mainBoard.getSubBoard(count, i));
    }

    private class Worker implements Runnable {
        private final Board board;

        public Worker(Board board) { this.board = board; }
        public void run() {
            while (!board.hasConverged()) {
                for (int x = 0; x < board.getMaxX(); x++)
                    for (int y = 0; y < board.getMaxY(); y++)
                        board.setNewValue(x, y, computeValue(x, y));
                try {
                    barrier.await();
                } catch (InterruptedException ex) {
                    return;
                } catch (BrokenBarrierException ex) {
                    return;
                }
            }
        }
    }

    public void start() {
        for (int i = 0; i < workers.length; i++)
            new Thread(workers[i]).start();
        mainBoard.waitForConvergence();
    }
}
```

Listing 5.15. Coordinating computation in a cellular automaton with `CyclicBarrier`.

of some additional memory usage.

Like many other frequently reinvented wheels, caching often looks simpler than it is. A naive cache implementation is likely to turn a performance bottleneck into a scalability bottleneck, even if it does improve single-threaded performance. In this section we develop an efficient and scalable result cache for a computationally expensive function. Let's start with the obvious approach—a simple HashMap—and then look at some of its concurrency disadvantages and how to fix them.

The Computable<A,V> interface in Listing 5.16 describes a function with input of type A and result of type V. ExpensiveFunction, which implements Computable, takes a long time to compute its result; we'd like to create a Computable wrapper that remembers the results of previous computations and encapsulates the caching process. (This technique is known as *memoization*.)

```
public interface Computable<A, V> {
    V compute(A arg) throws InterruptedException;
}

public class ExpensiveFunction
        implements Computable<String, BigInteger> {
    public BigInteger compute(String arg) {
        // after deep thought...
        return new BigInteger(arg);
    }
}

public class Memoizer1<A, V> implements Computable<A, V> {
    @GuardedBy("this")
    private final Map<A, V> cache = new HashMap<A, V>();
    private final Computable<A, V> c;

    public Memoizer1(Computable<A, V> c) {
        this.c = c;
    }

    public synchronized V compute(A arg) throws InterruptedException {
        V result = cache.get(arg);
        if (result == null) {
            result = c.compute(arg);
            cache.put(arg, result);
        }
        return result;
    }
}
```

LISTING 5.16. Initial cache attempt using HashMap and synchronization.

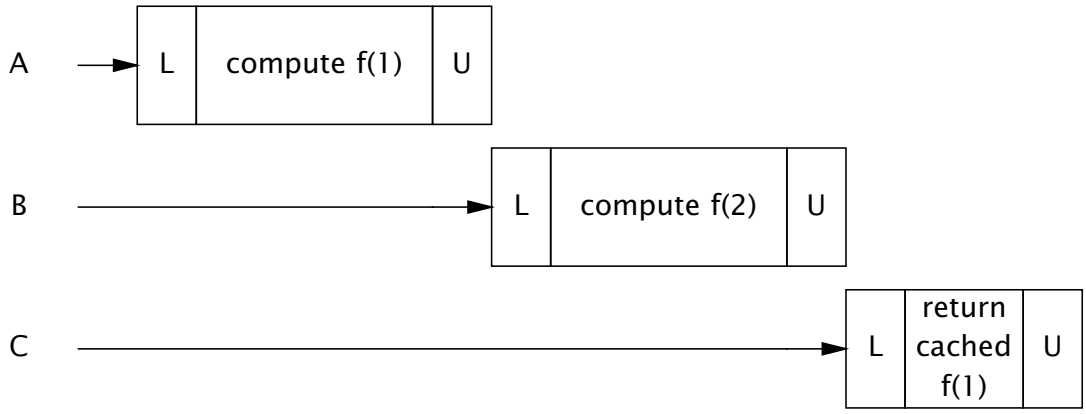

FIGURE 5.2. Poor concurrency of `Memoizer1`.

`Memoizer1` in Listing 5.16 shows a first attempt: using a `HashMap` to store the results of previous computations. The `compute` method first checks whether the desired result is already cached, and returns the precomputed value if it is. Otherwise, the result is computed and cached in the `HashMap` before returning.

`HashMap` is not thread-safe, so to ensure that two threads do not access the `HashMap` at the same time, `Memoizer1` takes the conservative approach of synchronizing the entire `compute` method. This ensures thread safety but has an obvious scalability problem: only one thread at a time can execute `compute` at all. If another thread is busy computing a result, other threads calling `compute` may be blocked for a long time. If multiple threads are queued up waiting to compute values not already computed, `compute` may actually take longer than it would have without memoization. Figure 5.2 illustrates what could happen when several threads attempt to use a function memoized with this approach. This is not the sort of performance improvement we had hoped to achieve through caching.

`Memoizer2` in Listing 5.17 improves on the awful concurrent behavior of `Memoizer1` by replacing the `HashMap` with a `ConcurrentHashMap`. Since `ConcurrentHashMap` is thread-safe, there is no need to synchronize when accessing the backing `Map`, thus eliminating the serialization induced by synchronizing `compute` in `Memoizer1`.

`Memoizer2` certainly has better concurrent behavior than `Memoizer1`: multiple threads can actually use it concurrently. But it still has some defects as a cache—there is a window of vulnerability in which two threads calling `compute` at the same time could end up computing the same value. In the case of memoization, this is merely inefficient—the purpose of a cache is to prevent the same data from being calculated multiple times. For a more general-purpose caching mechanism, it is far worse; for an object cache that is supposed to provide once-and-only-once initialization, this vulnerability would also pose a safety risk.

The problem with `Memoizer2` is that if one thread starts an expensive computation, other threads are not aware that the computation is in progress and so may start the same computation, as illustrated in Figure 5.3. We'd like to somehow represent the notion that "thread X is currently computing $f(27)$", so that if another thread arrives looking for $f(27)$, it knows that the most efficient way to find it is to head over to Thread X's house, hang out there until X is finished, and

```
public class Memoizer2<A, V> implements Computable<A, V> {
    private final Map<A, V> cache = new ConcurrentHashMap<A, V>();
    private final Computable<A, V> c;

    public Memoizer2(Computable<A, V> c) { this.c = c; }

    public V compute(A arg) throws InterruptedException {
        V result = cache.get(arg);
        if (result == null) {
            result = c.compute(arg);
            cache.put(arg, result);
        }
        return result;
    }
}
```

LISTING 5.17. Replacing HashMap with ConcurrentHashMap.

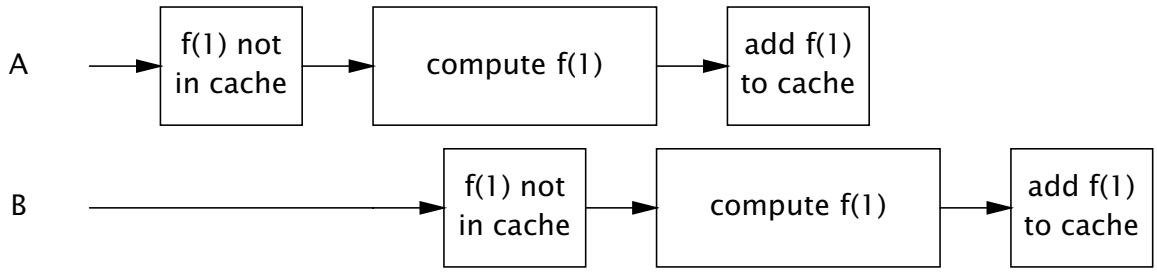

FIGURE 5.3. Two threads computing the same value when using Memoizer2.

then ask "Hey, what did you get for $f(27)$?"

We've already seen a class that does almost exactly this: FutureTask. FutureTask represents a computational process that may or may not already have completed. FutureTask.get returns the result of the computation immediately if it is available; otherwise it blocks until the result has been computed and then returns it.

Memoizer3 in Listing 5.18 redefines the backing Map for the value cache as a ConcurrentHashMap<A,Future<V>> instead of a ConcurrentHashMap<A,V>. Memoizer3 first checks to see if the appropriate calculation has been started (as opposed to finished, as in Memoizer2). If not, it creates a FutureTask, registers it in the Map, and starts the computation; otherwise it waits for the result of the existing computation. The result might be available immediately or might be in the process of being computed—but this is transparent to the caller of Future.get.

The Memoizer3 implementation is almost perfect: it exhibits very good concurrency (mostly derived from the excellent concurrency of ConcurrentHashMap), the result is returned efficiently if it is already known, and if the computation is in progress by another thread, newly arriving threads wait patiently for the result. It has only one defect—there is still a small window of vulnerability in which

```
public class Memoizer3<A, V> implements Computable<A, V> {
    private final Map<A, Future<V>> cache
            = new ConcurrentHashMap<A, Future<V>>();
    private final Computable<A, V> c;

    public Memoizer3(Computable<A, V> c) { this.c = c; }

    public V compute(final A arg) throws InterruptedException {
        Future<V> f = cache.get(arg);
        if (f == null) {
            Callable<V> eval = new Callable<V>() {
                public V call() throws InterruptedException {
                    return c.compute(arg);
                }
            };
            FutureTask<V> ft = new FutureTask<V>(eval);
            f = ft;
            cache.put(arg, ft);
            ft.run(); // call to c.compute happens here
        }
        try {
            return f.get();
        } catch (ExecutionException e) {
            throw launderThrowable(e.getCause());
        }
    }
}
```

Listing 5.18. Memoizing wrapper using `FutureTask`.

two threads might compute the same value. This window is far smaller than in `Memoizer2`, but because the `if` block in `compute` is still a nonatomic check-then-act sequence, it is possible for two threads to call `compute` with the same value at roughly the same time, both see that the cache does not contain the desired value, and both start the computation. This unlucky timing is illustrated in Figure 5.4.

`Memoizer3` is vulnerable to this problem because a compound action (put-if-absent) is performed on the backing map that cannot be made atomic using locking. `Memoizer` in Listing 5.19 takes advantage of the atomic `putIfAbsent` method of `ConcurrentMap`, closing the window of vulnerability in `Memoizer3`.

Caching a `Future` instead of a value creates the possibility of *cache pollution*: if a computation is cancelled or fails, future attempts to compute the result will also indicate cancellation or failure. To avoid this, `Memoizer` removes the `Future` from the cache if it detects that the computation was cancelled; it might also be desirable to remove the `Future` upon detecting a `RuntimeException` if the computation might succeed on a future attempt. `Memoizer` also does not address

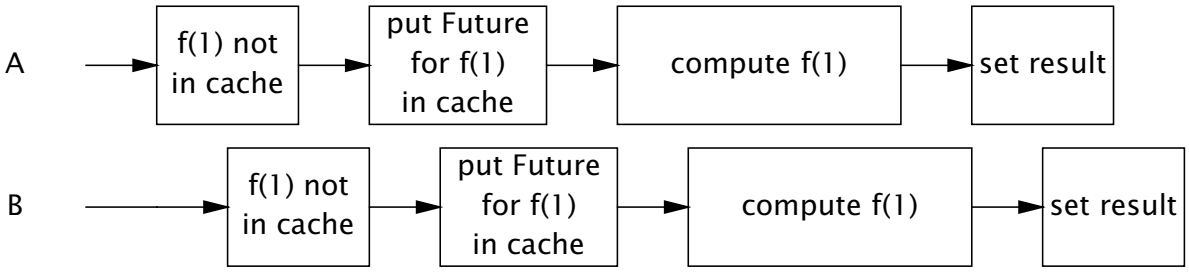

FIGURE 5.4. Unlucky timing that could cause `Memoizer3` to calculate the same value twice.

cache expiration, but this could be accomplished by using a subclass of `FutureTask` that associates an expiration time with each result and periodically scanning the cache for expired entries. (Similarly, it does not address cache eviction, where old entries are removed to make room for new ones so that the cache does not consume too much memory.)

With our concurrent cache implementation complete, we can now add real caching to the factorizing servlet from Chapter 2, as promised. `Factorizer` in Listing 5.20 uses `Memoizer` to cache previously computed values efficiently and scalably.

```
public class Memoizer<A, V> implements Computable<A, V> {
    private final ConcurrentMap<A, Future<V>> cache
        = new ConcurrentHashMap<A, Future<V>>();
    private final Computable<A, V> c;

    public Memoizer(Computable<A, V> c) { this.c = c; }

    public V compute(final A arg) throws InterruptedException {
        while (true) {
            Future<V> f = cache.get(arg);
            if (f == null) {
                Callable<V> eval = new Callable<V>() {
                    public V call() throws InterruptedException {
                        return c.compute(arg);
                    }
                };
                FutureTask<V> ft = new FutureTask<V>(eval);
                f = cache.putIfAbsent(arg, ft);
                if (f == null) { f = ft; ft.run(); }
            }
            try {
                return f.get();
            } catch (CancellationException e) {
                cache.remove(arg, f);
            } catch (ExecutionException e) {
                throw launderThrowable(e.getCause());
            }
        }
    }
}
```

Listing 5.19. Final implementation of `Memoizer`.

```
@ThreadSafe
public class Factorizer implements Servlet {
    private final Computable<BigInteger, BigInteger[]> c =
        new Computable<BigInteger, BigInteger[]>() {
            public BigInteger[] compute(BigInteger arg) {
                return factor(arg);
            }
        };
    private final Computable<BigInteger, BigInteger[]> cache
        = new Memoizer<BigInteger, BigInteger[]>(c);

    public void service(ServletRequest req,
                        ServletResponse resp) {
        try {
            BigInteger i = extractFromRequest(req);
            encodeIntoResponse(resp, cache.compute(i));
        } catch (InterruptedException e) {
            encodeError(resp, "factorization interrupted");
        }
    }
}
```

Listing 5.20. Factorizing servlet that caches results using `Memoizer`.