SAMPLE CHAPTER

Vital techniques of Java 7 and polyglot programming

The Well-Grounded Java Developer

Benjamin J. Evans
Martijn Verburg

FOREWORD BY Dr. Heinz Kabutz

MANNING

The Well-Grounded Java Developer

by Benjamin J. Evans
Martijn Verburg

Chapter 4

Copyright 2013 Manning Publications

brief contents

PART 1 DEVELOPING WITH JAVA 7 1

1 ■ Introducing Java 7 3
2 ■ New I/O 20

PART 2 VITAL TECHNIQUES 51

3 ■ Dependency Injection 53
4 ■ Modern concurrency 76
5 ■ Class files and bytecode 119
6 ■ Understanding performance tuning 150

PART 3 POLYGLOT PROGRAMMING ON THE JVM.....................191

7 ■ Alternative JVM languages 193
8 ■ Groovy: Java's dynamic friend 213
9 ■ Scala: powerful and concise 241
10 ■ Clojure: safer programming 279

PART 4 CRAFTING THE POLYGLOT PROJECT 311

11 ■ Test-driven development 313
12 ■ Build and continuous integration 342
13 ■ Rapid web development 380
14 ■ Staying well-grounded 410

Modern concurrency

This chapter covers

- Concurrency theory
- Block-structured concurrency
- The java.util.concurrent libraries
- Lightweight concurrency with the fork/join framework
- The Java Memory Model (JMM)

In this chapter, we'll begin with basic concepts and a whistle-stop tour of block-structured concurrency. This was the only game in town prior to Java 5, and it's still worth understanding. Next, we'll cover what every working developer should know about `java.util.concurrent` and how to use the basic concurrency building blocks it provides.

We'll conclude with a look at the new fork/join framework, so that by the end of the chapter, you'll be ready to start applying these new concurrency techniques in your own code. You'll also have enough theory to fully grasp the different views of concurrency that we'll discuss in later parts of the book, when we meet non-Java languages.

This chapter isn't intended to be a complete statement of everything you'll ever need to know about concurrency—it's enough to get you started and give you an

But I already know about Thread!

It's one of the most common (and potentially deadly) mistakes a developer can make—to assume that an acquaintance with `Thread`, `Runnable`, and the language-level basic primitives of Java's concurrency mechanism is enough to be a competent developer of concurrent code. In fact, the subject of concurrency is a very large one, and good multithreaded development is difficult and continues to cause problems for even the best developers with years of experience under their belts.

One other point you should be aware of is that the area of concurrency is undergoing a massive amount of active research at present—research that will certainly have an impact on Java and the other languages you'll use over the course of your career. If we were to pick one fundamental area of computing that's likely to change radically in terms of industry practice over the next five years, it would be concurrency.

appreciation of what you'll need to learn more about, and to stop you being dangerous when writing concurrent code. But you'll need to know more than we can cover here if you're going to be a truly first-rate developer of multithreaded code. There are a number of excellent books about nothing but Java concurrency—two of the best are *Concurrent Programming in Java*, second edition, by Doug Lea (Prentice Hall, 1999), and *Java Concurrency in Practice* by Brian Goetz and others (Addison-Wesley Professional, 2006).

The aim of this chapter is to make you aware of the underlying platform mechanisms that explain why Java's concurrency works the way it does. We'll also cover enough general concurrency theory to give you the vocabulary to understand the issues involved, and to teach you about both the necessity and the difficulty involved in getting concurrency right. In fact, that's where we're going to start.

4.1 Concurrency theory—a primer

To make sense of Java's approach to concurrent programming, we're going to start off by talking about theory. First, we'll discuss the fundamentals of the Java threading model.

After that, we'll discuss the impact that "design forces" have in the design and implementation of systems. We'll talk about the two most important of these forces, *safety* and *liveness*, and mention some of the others. After that we'll turn to why the forces are often in conflict, and look at some reasons for overhead in concurrent systems.

We'll conclude this section by looking at an example of a multithreaded system, and illustrate how `java.util.concurrent` is a very natural way to write code.

4.1.1 Explaining Java's threading model

Java's threading model is based on two fundamental concepts:

- Shared, visible-by-default mutable state
- Preemptive thread scheduling

Let's consider some of the most important aspects of these ideas:

- Objects can be easily shared between all threads within a process.
- Objects can be changed ("mutated") by any threads that have a reference to them.
- The thread scheduler can swap threads on and off cores at any time, more or less.
- Methods must be able to be swapped out while they're running (otherwise a method with an infinite loop would steal the CPU forever).

  This, however, runs the risk of an unpredictable thread swap leaving a method "half-done" and an object in an inconsistent state. There is also the risk of changes made in one thread not being visible in other threads when they need to be. To mitigate these risks, we come to the last point.
- Objects can be *locked* to protect vulnerable data.

Java's thread- and lock-based concurrency is very low-level, and often hard to work with. To cope with this, a set of concurrency libraries, known as `java.util.concurrent`, was introduced in Java 5. This provided a set of tools for writing concurrent code that many programmers find easier to use than the classic block-structured concurrency primitives.

Lessons learned

Java was the first mainstream programming language to have built-in support for multithreaded programming. This represented a huge step forward at the time, but now, 15 years later, we've learned a lot more about how to write concurrent code.

It turns out that some of Java's initial design decisions are quite difficult for most programmers to work with. This is unfortunate, because the increasing trend in hardware is toward processors with many cores, and the only good way to take advantage of those cores is with concurrent code. We'll discuss some of the difficulties of concurrent code in this chapter. The subject of modern processors naturally requiring concurrent programming is covered in some detail in chapter 6 where we discuss performance.

As developers become more experienced with writing concurrent code, they find themselves running up against recurring concerns that are important to their systems. We call these concerns "design forces." They're high-level forces that exist (and often conflict) in the design of practical concurrent OO systems.

We're going to spend a little bit of time looking at some of the most important of these forces in the next couple of sections.

4.1.2 Design concepts

The most important design forces were catalogued by Doug Lea as he was doing his landmark work producing `java.util.concurrent`:

- Safety (also known as *concurrent type safety*)
- Liveness
- Performance
- Reusability

Let's look at each of these forces now.

Safety and concurrent type safety

Safety is about ensuring that object instances remain self-consistent regardless of any other operations that may be happening at the same time. If a system of objects has this property, it's said to be *concurrently type-safe*.

As you might guess from the name, one way to think about concurrency is in terms of an extension to the regular concepts of object modeling and type safety. In nonconcurrent code, you want to ensure that regardless of what public methods you call on an object, it's in a well-defined and consistent state at the end of the method. The usual way to do this is to keep all of an object's state private and expose a public API of methods that only alter the object's state in a self-consistent way.

Concurrent type safety is the same basic concept as type safety for an object, but applied to the much more complex world in which other threads are potentially operating on the same objects on different CPU cores at the same time.

> **Staying safe**
>
> One strategy for safety is to never return from a non-private method in an inconsistent state, and to never call any non-private method (and certainly not a method on any other object) while in an inconsistent state. If this is combined with a way of protecting the object (such as a synchronization lock or critical section) while it's inconsistent, the system can be guaranteed to be safe.

Liveness

A live system is one in which every attempted activity eventually either progresses or fails.

The key word in the definition is *eventually*—there is a distinction between a transient failure to progress (which isn't that bad in isolation, even if it's not ideal) and a permanent failure. Transient failures could be caused by a number of underlying problems, such as:

- Locking or waiting to acquire a lock
- Waiting for input (such as network I/O)
- Temporary failure of a resource
- Not enough CPU time available to run the thread

Permanent failures could be due to a number of causes. These are some of the most common:

- Deadlock
- Unrecoverable resource problem (such as if the NFS goes away)
- Missed signal

We'll discuss locking and several of these other problems later in the chapter, although you may already be familiar with some or all of them.

PERFORMANCE

The performance of a system can be quantified in a number of different ways. In chapter 6, we'll talk about performance analysis and techniques for tuning, and we'll introduce a number of other metrics you should know about. For now, think of performance as being a measure of how much work a system can do with a given amount of resources.

REUSABILITY

Reusability forms a fourth design force, because it isn't really covered by any of the other considerations. A concurrent system that has been designed for easy reuse is sometimes very desirable, although this isn't always easy to implement. One approach is to use a reusable toolbox (like `java.util.concurrent`) and build non-reusable application code on top of it.

4.1.3 How and why do the forces conflict?

The design forces are often in opposition to each other, and this tension can be viewed as a central reason why designing good concurrent systems is difficult.

- Safety stands in opposition to liveness—safety is about ensuring that bad things don't happen, whereas liveness requires progress to be made.
- Reusable systems tend to expose their internals, which can cause problems with safety.
- A naïvely written safe system will typically not be very performant, as it usually resorts to the heavy use of locking to provide safety guarantees.

The balance that you should ultimately try to achieve is for the code to be flexible enough to be useful for a wide range of problems, closed enough to be safe, and still reasonably live and performant. This is quite a tall order, but, fortunately, there are some practical techniques to help with this. Here are some of the most common in rough order of usefulness:

- Restrict the external communication of each subsystem as much as possible. Data hiding is a powerful tool for aiding with safety.
- Make the internal structure of each subsystem as deterministic as possible. For example, design in static knowledge of the threads and objects in each subsystem, even if the subsystems will interact in a concurrent, nondeterministic way.

- Apply policy approaches that client apps must adhere to. This technique is powerful, but relies on user apps cooperating, and it can be hard to debug if a badly behaved app disobeys the rules.
- Document the required behavior. This is the weakest of the alternatives, but it's sometimes necessary if the code is to be deployed in a very general context.

The developer should be aware of each of these possible safety mechanisms and should use the strongest possible technique, while being aware that there are circumstances in which only the weaker mechanisms are possible.

4.1.4 Sources of overhead

There are many aspects of a concurrent system that can contribute to the inherent overhead:

- Locks and monitors
- Number of context switches
- Number of threads
- Scheduling
- Locality of memory
- Algorithm design

This should form the basis of a checklist in your mind. When developing concurrent code, you should ensure that you have thought about everything on this list, before considering the code "done."

> **Algorithm design**
> This is an area in which developers can really distinguish themselves—learning about algorithm design will make you a better programmer in any language. Two of the best books are *Introduction to Algorithms* by Thomas H. Corman et al. (MIT, 2009)—don't be deceived by the title, this is a serious work—and *The Algorithm Design Manual*, second edition, by Steven Skiena (Springer-Verlag, 2008). For both single-threaded and concurrent algorithms, these are excellent choices for further reading.

We'll mention many of these sources of overhead in this chapter (and in chapter 6, about performance).

4.1.5 A transaction processor example

To round off this rather theoretical section, let's apply some of this theory to the design of an example concurrent application. We'll see how we might approach it using a high-level view of the classes from `java.util.concurrent`.

Consider a basic transaction processing system. A simple and standard way to construct such an application is to have different phases of the application correspond to different parts of the business process. Each phase is then represented by a thread

pool that takes in work items one by one, does an amount of processing on each item, and hands off the item to the next thread pool. In general, it's good design to have each thread pool concentrate on processing that is pertinent to one specific functional area. You can see an example application in figure 4.1.

If you design applications like this, you can improve throughput because you can have several work items in flight at once. One work item can be in processing in the Credit Check phase at the same time as another is in Stock Check. Depending on the details of the application, there can even be multiple different orders in Stock Check at the same time.

Designs of this type are very well-suited to being implemented using the classes found in `java.util.concurrent`. The package contains thread pools for execution (and a nice set of factory methods in the `Executors` class to create them) and queues for handing work off between pools. There are also concurrent data structures (for building shared caches and other use cases) and many other useful low-level tools.

But, you might ask, what would we have done before the advent of Java 5, when we didn't have these classes available? In many cases, application groups would come up with their own concurrent programming libraries—they'd end up building components similar in aims to the ones found in `java.util.concurrent`. But many of these bespoke components would have design problems, and subtle (or not-so-subtle) concurrency bugs. If `java.util.concurrent` didn't exist, application developers would end up reinventing much of it for themselves (probably in a buggy, badly tested form).

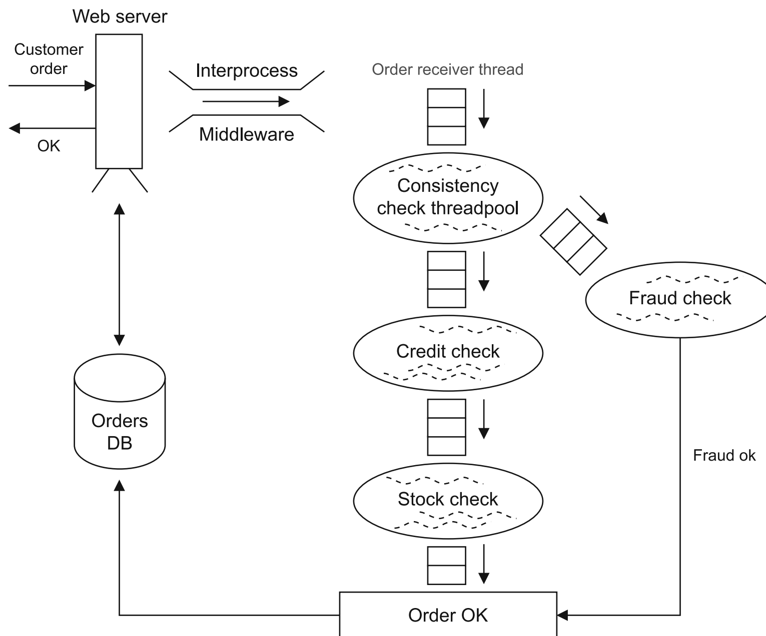


Figure 4.1 An example multithreaded application

With this example in mind, let's turn to our next subject—a review of Java's "classic" concurrency and a close look at why programming with it can be difficult.

4.2 Block-structured concurrency (pre-Java 5)

Much of this chapter is taken up with discussing alternatives to the block-synchronization-based approach to concurrency. But to get the most out of the discussion of the alternatives, it's important to have a firm grasp of what's good and bad about the classic view of concurrency.

To that end, we'll discuss the original, quite low-level way of tackling multithreaded programming using Java's concurrency keywords—`synchronized`, `volatile`, and so on. This discussion will take place in the context of the design forces and with an eye to what will come in the next sections.

Following on from that, we'll briefly consider the lifecycle of a thread, and then discuss common techniques (and pitfalls) of concurrent code, such as fully synchronized objects, deadlocks, the `volatile` keyword, and immutability.

Let's get started with a review of synchronization.

4.2.1 Synchronization and locks

As you already know, the `synchronized` keyword can be applied either to a block or to a method. It indicates that before entering the block or method, a thread must acquire the appropriate lock. For a method, that means acquiring the lock belonging to the object instance (or the lock belonging to the `Class` object for `static synchronized` methods). For a block, the programmer should indicate which object's lock is to be acquired.

Only one thread can be progressing through any of an object's synchronized blocks or methods at once; if other threads try to enter, they're suspended by the JVM. This is true regardless of whether the other thread is trying to enter either the same or a different synchronized block on the same object. In concurrency theory, this type of construct is referred to as a *critical section*.

> **NOTE** Have you ever wondered why the Java keyword used for a critical section is `synchronized`? Why not "critical" or "locked"? What is it that's being *synchronized*? We'll return to this in section 4.2.5, but if you don't know or have never thought about it, you may want to take a couple of minutes to ponder it before continuing.

We're really focusing on some of the newer concurrency techniques in this chapter. But as we're talking about synchronization, let's look at some basic facts about synchronization and locks in Java. Hopefully you already have most (or all) of these at your fingertips:

- Only objects—not primitives—can be locked.
- Locking an array of objects doesn't lock the individual objects.
- A synchronized method can be thought of as equivalent to a `synchronized (this) { ... }` block that covers the entire method (but note that they're represented differently in bytecode).

- A `static synchronized` method locks the `Class` object, because there's no instance object to lock.
- If you need to lock a class object, consider carefully whether you need to do so explicitly, or by using `getClass()`, because the behavior of the two approaches will be different in a subclass.
- Synchronization in an inner class is independent of the outer class (to see why this is so, remember how inner classes are implemented).
- `synchronized` doesn't form part of the method signature, so it can't appear on a method declaration in an interface.
- Unsynchronized methods don't look at or care about the state of any locks, and they can progress while synchronized methods are running.
- Java's locks are reentrant. That means a thread holding a lock that encounters a synchronization point for the same lock (such as a synchronized method calling another synchronized method in the same class) will be allowed to continue.

WARNING Non-reentrant locking schemes exist in other languages (and can be synthesized in Java—see the Javadoc for `ReentrantLock` in `java.util.concurrent.locks` if you want the gory details) but they're generally painful to deal with, and they're best avoided unless you really know what you're doing.

That's enough review of Java's synchronization. Now let's move on to discuss the states that a thread moves through during its lifecycle.

4.2.2 The state model for a thread

In figure 4.2, you can see how a thread lifecycle progresses—from creation to running, to possibly being suspended, before running again (or blocking on a resource), and eventually completing.

A thread is initially created in the Ready state. The scheduler will then find a core for it to run upon, and some small amount of waiting time may be involved if the machine is heavily loaded. From there, the thread will usually consume its time allocation and be placed back into the Ready state to await further processor time slices. This is the action of the forcible thread scheduling that we mentioned in section 4.1.1.

As well as the standard action by the scheduler, the thread itself can indicate that it isn't able to make use of the core at this time. This can be because the program code indicates that the thread should pause before continuing (via `Thread.sleep()`) or because the thread must wait until notified (usually that some external condition has been met). Under these circumstances, the thread is removed from the core and releases all its locks. It can only run again by being woken up (after sleeping for the right length of time, or because it has received the appropriate signal) and placed back in the Ready state.

The thread can be blocked because it's waiting on I/O or to acquire a lock held by another thread. In this case, the thread isn't swapped off the core but is kept busy, waiting for the lock or data to become available. If this happens, the thread will continue to execute until the end of its timeslice.

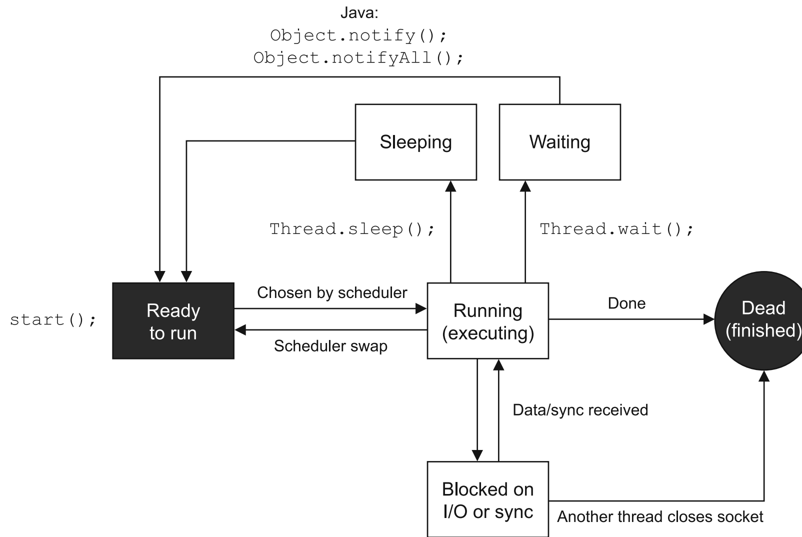


Figure 4.2 The state model of a Java thread

Let's move on to talk about one well-known way to solve the synchronization problem. This is the idea of fully synchronized objects.

4.2.3 Fully synchronized objects

Earlier in this chapter, we introduced the concept of concurrent type safety and mentioned one strategy for achieving this (in the "Staying Safe" sidebar). Let's look at a more complete description of this strategy, which is usually called *fully synchronized objects*. If all of the following rules are obeyed, the class is known to be thread-safe and will also be live.

A fully synchronized class is a class that meets all of these conditions:

- All fields are always initialized to a consistent state in every constructor.
- There are no public fields.
- Object instances are guaranteed to be consistent after returning from any non-private method (assuming the state was consistent when the method was called).
- All methods provably terminate in bounded time.
- All methods are synchronized.
- There is no calling of another instance's methods while in an inconsistent state.
- There is no calling of any non-private method while in an inconsistent state.

Listing 4.1 shows an example of such a class from the backend of an imaginary distributed microblogging tool. The `ExampleTimingNode` class will receive updates by having its `propagateUpdate()` method called and can also be queried to see if it has received a specific update. This situation provides a classic conflict between a read and a write operation, so synchronization is used to prevent inconsistency.

Listing 4.1 A fully synchronized class

```
public class ExampleTimingNode implements SimpleMicroBlogNode {

  private final String identifier;                          // No public fields

  private final Map<Update, Long> arrivalTime
➥ = new HashMap<>();

  public ExampleTimingNode(String identifier_) {            // All fields initialized in constructor
    identifier = identifier_;
  }

  public synchronized String getIdentifier() {              // All methods are synchronized
    return identifier;
  }

  public synchronized void propagateUpdate(
➥ Update update_) {
    long currentTime = System.currentTimeMillis();
    arrivalTime.put(update_, currentTime);
  }

  public synchronized boolean confirmUpdateReceived(
➥ Update update_) {
    Long timeRecvd = arrivalTime.get(update_);
    return timeRecvd != null;
  }
}
```

This seems fantastic at first glance—the class is both safe and live. The problem comes with performance—just because something is safe and live doesn't mean it's necessarily going to be very quick. You have to use `synchronized` to coordinate all the accesses (both `get` and `put`) to the `arrivalTime` map, and that locking is ultimately going to slow you down. This is a central problem of this way of handling concurrency.

> **Code fragility**
>
> In addition to the performance problems, the code in listing 4.1 is quite fragile. You can see that you never touch `arrivalTime` outside of a `synchronized` method (and in fact there's only `get` and `put` access), but this is only possible because of the small amount of code in play. In real, larger systems, this would not be possible due to the amount of code. It's very easy for bugs to creep into larger codebases that use this approach, which is another reason that the Java community began to look for more robust approaches.

4.2.4 Deadlocks

Another classic problem of concurrency (and not just Java's take on it) is the *deadlock*. Consider listing 4.2, which is a slightly extended form of the last example. In this version, as well as recording the time of the last update, each node that receives an update informs another node of that receipt.

This is a naïve attempt to build a multithreaded update handling system. It's designed to demonstrate deadlocking—you shouldn't use this as the basis for real code.

Listing 4.2 A deadlocking example

```
public class MicroBlogNode implements SimpleMicroBlogNode {
  private final String ident;

  public MicroBlogNode(String ident_) {
    ident = ident_;
  }

  public String getIdent() {
    return ident;
  }

  public synchronized void propagateUpdate(Update upd_, MicroBlogNode
     backup_) {
    System.out.println(ident +": recvd: "+ upd_.getUpdateText()
    ➥ +" ; backup: "+backup_.getIdent());
    backup_.confirmUpdate(this, upd_);
  }

  public synchronized void confirmUpdate(MicroBlogNode other_, Update
     update_) {
    System.out.println(ident +": recvd confirm: "+
    ➥ update_.getUpdateText() +" from "+other_.getIdent()k);
  }
}

final MicroBlogNode local =                                  ◁ Keyword final
➥ new MicroBlogNode("localhost:8888");                         is required
final MicroBlogNode other = new MicroBlogNode("localhost:8988");
final Update first = getUpdate("1");
final Update second = getUpdate("2");

new Thread(new Runnable() {
  public void run() {
    local.propagateUpdate(first, other);                ◁ First update sent to first thread
  }
}).start();

new Thread(new Runnable() {
  public void run() {
    other.propagateUpdate(second, local);               ◁ Second update sent to other thread
  }
}).start();
```

At first glance, this code looks sensible. You have two updates being sent to separate threads, each of which has to be confirmed on backup threads. This doesn't seem too outlandish a design—if one thread has a failure, there is another thread that can potentially carry on.

If you run the code, you'll normally see an example of a deadlock—both threads will report receiving the update, but neither will confirm receiving the update for which they're the backup thread. The reason for this is that each thread requires the

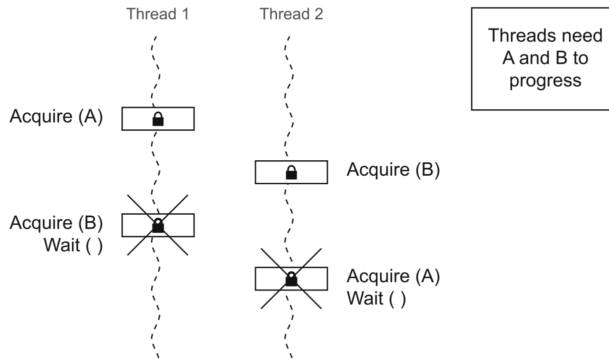


Figure 4.3 Deadlocked threads

other to release the lock it holds before the confirmation method can progress. This is illustrated in figure 4.3.

To deal with deadlocks, one technique is to always acquire locks in the same order in every thread. In the preceding example, the first thread to start acquires them in the order A, B, whereas the second thread acquires them in the order B, A. If both threads had insisted on acquiring in order A, B, the deadlock would have been avoided, because the second thread would have been blocked from running at all until the first had completed and released its locks.

In terms of the fully synchronized object approach, this deadlock is prevented because the code violates the consistent state rule. When a message arrives, the receiving node calls another object while the message is still being processed—the state isn't consistent when it makes this call.

Next, we'll return to a puzzle we posed earlier: why the Java keyword for a critical section is `synchronized`. This will lead us into a discussion of immutability and then the `volatile` keyword.

4.2.5 Why synchronized?

One of the biggest changes in concurrent programming in recent years has been in the realm of hardware. It wasn't that many years ago that a working programmer could go for years on end without encountering a system that had more than one or at most two processing cores. It was thus possible to think of concurrent programming as being about the timesharing of the CPU—threads swapping on and off a single core.

Today, anything larger than a mobile phone has multiple cores, so the mental model should be different too, encompassing multiple threads all running on different cores at the same physical moment (and potentially operating on shared data). You can see this in figure 4.4. For efficiency, each thread that is running simultaneously may have its own cached copy of data being operated on. With this picture in mind, let's turn to the question of the choice of keyword used to denote a locked section or method.

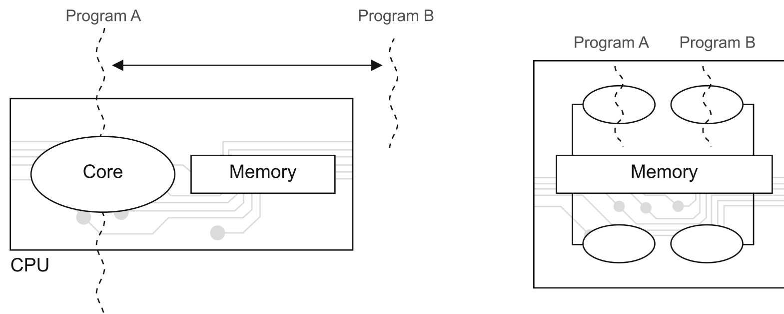


Figure 4.4 Old and new ways of thinking about concurrency and threads

We asked earlier, what is it that's being synchronized in the code in listing 4.1? The answer is: *The memory representation in different threads of the object being locked is what is being synchronized.* That is, after the `synchronized` block (or method) has completed, any and all changes that were made to the object being locked are flushed back to main memory before the lock is released, as illustrated in figure 4.5.

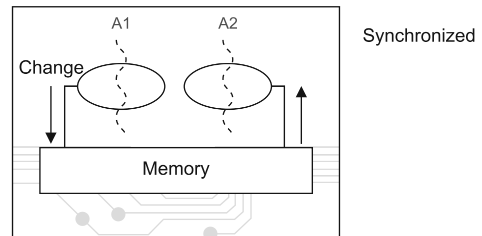


Figure 4.5 A change to an object propagates between threads via main memory

In addition, when a `synchronized` block is entered, then after the lock has been acquired, any changes to the locked object are read in *from* main memory, so the thread with the lock is synchronized to main memory's view of the object before the code in the locked section begins to execute.

4.2.6 The volatile keyword

Java has had the `volatile` keyword since the dawn of time (Java 1.0), and it's used as a simple way to deal with the synchronization of object fields, including primitives. The following rules govern a volatile field:

- The value seen by a thread is always reread from main memory before use.
- Any value written by a thread is always flushed through to main memory before the instruction completes.

This can be thought of as acting like a tiny little `synchronized` block around the operation. It allows the programmer to write simplified code, but at the cost of the extra flushes on every access. Notice also that the `volatile` variable doesn't introduce any locks, so you can't deadlock by using `volatile` variables.

One slightly more subtle consequence of `volatile` variables is that for true thread-safety, a `volatile` variable should only be used to model a variable where writes to the variable don't depend on the current state (the read state) of the variable. For cases where the current state matters, you must always introduce a lock to be completely safe.

4.2.7 Immutability

One technique that can be of great value is the use of immutable objects. These are objects that either have no state, or that have only `final` fields (which must therefore be populated in the constructors of the objects). These are always safe and live, because their state can't be mutated, so they can never be in an inconsistent state.

One problem is that any values that are required to initialize a particular object must be passed into the constructor. This can lead to unwieldy constructor calls, with many parameters. Alternatively, many coders use a `FactoryMethod` instead. This can be as simple as using a `static` method on the class, instead of a constructor, to produce new objects. The constructors are usually made `protected` or `private`, so that the static `FactoryMethods` are the only way of instantiating.

This still has the problem of potentially needing many parameters to be passed in to the `FactoryMethod`. This isn't always very convenient, especially when you may need to accumulate state from several sources before creating a new immutable object.

To solve this, you can use the Builder pattern. This is a combination of two constructs: a `static` inner class that implements a generic builder interface, and a `private` constructor for the immutable class itself.

The `static` inner class is the builder for the immutable class, and it provides the only way that a developer can get hold of new instances of the immutable type. One very common implementation is for the `Builder` class to have exactly the same fields as the immutable class, but to allow mutation of the fields.

This listing shows how you might use this to model a microblogging update (again, building on the earlier listings in this chapter).

Listing 4.3 Immutable objects and builders

```
public interface ObjBuilder<T> {
  T build();                                    // Builder interface
}

public class Update {
  private final Author author;                  // Final fields must be
  private final String updateText;              // initialized in constructor

  private Update(Builder b_) {
    author = b_.author;
    updateText = b_.updateText;
  }

  public static class Builder
      implements ObjBuilder<Update> {           // Builder class must be static inner
    private Author author;
    private String updateText;

    public Builder author(Author author_) {     // Methods on Builder return Builder for chain calls
      author = author_;
      return this;
    }

    public Builder updateText(String updateText_) {
```

```
      updateText = updateText_;
      return this;
    }
    public Update build() {
      return new Update(this);
    }
  }

}
```

hashCode() and equals() methods omitted

With this code, you could then create a new `Update` object like this:

```
Update.Builder ub = new Update.Builder();
Update u = ub.author(myAuthor).updateText("Hello").build();
```

This is a very common pattern and one that has wide applicability. In fact, we've already made use of the properties of immutable objects in listings 4.1 and 4.2.

One last point about immutable objects—the `final` keyword only applies to the object directly pointed to. As you can see in figure 4.6, the reference to the main object can't be assigned to point at object 3, but within the object, the reference to 1 can be swung to point at object 2. Another way of saying this is that a `final` reference can point at an object that has nonfinal fields.

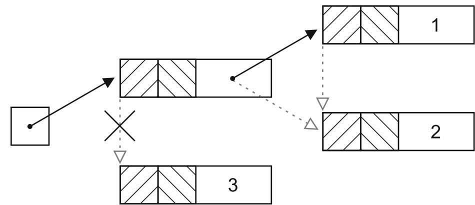


Figure 4.6 Immutability of value versus reference

Immutability is a very powerful technique, and you should use it whenever feasible. Sometimes it's just not possible to develop efficiently with only immutable objects, because every change to an object's state requires a new object to be spun up. So we're left with the necessity of dealing with mutable objects.

We'll now turn to one of the biggest topics in this chapter—a tour of the more modern and conceptually simple concurrency APIs presented in `java.util.concurrent`. We'll look at how you can start to use them in your own code.

4.3 Building blocks for modern concurrent applications

With the advent of Java 5, a new way of thinking about concurrency in Java emerged. This was spearheaded by the package `java.util.concurrent`, which contained a rich new toolbox for working with multithreaded code. This toolbox has been enhanced with subsequent versions of Java, but the classes and packages that were introduced with Java 5 still work the same way and they're still very valuable to the working developer.

We're going to take a whirlwind tour through some of the headline classes in `java.util.concurrent` and related packages, such as the atomic and locks packages. We'll get you started using the classes and look at examples of use cases for them. You should also read the Javadoc for them and try to build up your familiarity with the packages as a whole—they make programming concurrent classes much easier.

> **Migrating code**
>
> If you have existing multithreaded code that is still based on the older (pre-Java 5) approaches, you should refactor it to use `java.util.concurrent`. In our experience, your code will be improved if you make a conscious effort to port it to the newer APIs—the greater clarity and reliability will be well worth the effort expended to migrate in almost all cases.

Consider this discussion a starter toolkit for concurrent code, not a full workshop. To get the most out of `java.util.concurrent`, you'll need to read more than we can present here.

4.3.1 Atomic classes—java.util.concurrent.atomic

The package `java.util.concurrent.atomic` contains several classes that have names starting with `Atomic`. They're essentially providing the same semantics as a `volatile`, but wrapped in a class API that includes atomic (meaning all-or-nothing) methods for suitable operations. This can be a very simple way for a developer to avoid race conditions on shared data.

The implementations are written to take advantage of modern processor features, so they can be nonblocking (lock-free) if suitable support is available from the hardware and OS, which it should be for most modern systems. A common use is to implement sequence numbers, using the atomic `getAndIncrement()` method on the `AtomicInteger` or `AtomicLong`.

To be a sequence number, the class should have a `nextId()` method that will return a number guaranteed to be unique (and strictly increasing) each time it's called. This is very similar to the database concept of a sequence number (hence the name of the variable).

Let's look at a bit of code that replicates sequence numbers:

```
private final AtomicLong sequenceNumber = new AtomicLong(0);

public long nextId() {
  return sequenceNumber.getAndIncrement();
}
```

CAUTION Atomic classes don't inherit from the similarly named classes, so `AtomicBoolean` can't be used in place of a `Boolean`, and `AtomicInteger` isn't an `Integer` (but it does extend `Number`).

Next, we'll examine how `java.util.concurrent` models the core of the synchronization model—the `Lock` interface.

4.3.2 Locks—java.util.concurrent.locks

The block-structured approach to synchronization is based around a simple notion of what a lock is. This approach has a number of shortcomings:

- There is only one type of lock.
- It applies equally to all synchronized operations on the locked object.
- The lock is acquired at the start of the synchronized block or method.
- The lock is released at the end of the block or method.
- The lock is either acquired or the thread blocks—no other outcomes are possible.

If we were going to reengineer the support for locks, there are several things we could potentially change for the better:

- Add different types of locks (such as reader and writer locks).
- Not restrict locks to blocks (allow a lock in one method and unlock in another).
- If a thread cannot acquire a lock (for example, if another thread has the lock), allow the thread to back out or carry on or do something else—a `tryLock()` method.
- Allow a thread to attempt to acquire a lock and give up after a certain amount of time.

The key to realizing all of these possibilities is the `Lock` interface in `java.util.concurrent.locks`. This ships with a couple of implementations:

- `ReentrantLock`—This is essentially the equivalent of the familiar lock used in Java synchronized blocks, but it's slightly more flexible.
- `ReentrantReadWriteLock`—This can provide better performance in cases where there are many readers but few writers.

The `Lock` interface can be used to completely replicate any functionality that is offered by block-structured concurrency. Here is the deadlock example rewritten to use the `ReentrantLock`.

Listing 4.4 Rewriting deadlock example to use `ReentrantLock`

```
private final Lock lock = new ReentrantLock();

public void propagateUpdate(Update upd_, MicroBlogNode backup_) {
  lock.lock();                                        <- Each thread locks own lock first
  try {
    System.out.println(ident +": recvd: "+
    ➥ upd_.getUpdateText() +" ; backup: "+
    ➥ backup_.getIdent());
      backup_.confirmUpdate(this, upd_);              <- Calls confirmUpdate() to acknowledge in other thread
  } finally {
    lock.unlock();
  }
}
```

```
public void confirmUpdate(MicroBlogNode other_, Update upd_) {
  lock.lock();                                          // 1 Attempts to lock other thread
  try{
    System.out.println(iden +": recvd confirm: "+
    ➥ upd_.getUpdateText() +" from "+ other_.getIdentifier());
  } finally {
    lock.unlock();
  }
}
```

The attempt ❶ to lock the other thread will generally fail, because it's already locked (as per figure 4.3). That's how the deadlock arises.

Using try ... finally with lock

The pattern of `lock()` with a `try` ... `finally` block, where the lock is released is a good addition to your toolbox. It works very well if you're replicating a situation that is similar to one where you'd have used block-structured concurrency. On the other hand, if you need to pass around the `Lock` objects (such as by returning it from a method), you can't use this pattern.

Using `Lock` objects can be considerably more powerful than a block-structured approach, but it is still sometimes hard to use them to design a robust locking strategy.

There are a number of strategies for dealing with deadlocks, but there's one in particular that doesn't work that you should be aware of. Consider the version of the `propagateUpdate()` method shown in the next listing (and imagine that the same change has been made to the `confirmUpdate()` code). In this example, we've replaced the unconditional lock with `tryLock()` with a timeout. This is an attempt to remove the deadlock by giving other threads a chance to get at the lock.

Listing 4.5 A flawed attempt to fix deadlock

```
public void propagateUpdate(Update upd_, MicroBlogNode backup_) {
  boolean acquired = false;

  while (!acquired) {
    try {
      int wait = (int)(Math.random() * 10);
      acquired = lock.tryLock(wait, TimeUnit.MILLISECONDS);   // Try and lock, with random timeout
      if (acquired) {
        System.out.println(ident +": recvd: "+
        ➥ upd_.getUpdateText() +" ; backup: "+backup_.getIdent());
        backup_.confirmUpdate(this, update_);                // Confirm on other thread
      } else {
        Thread.sleep(wait);
      }
    } catch (InterruptedException e) {
    } finally {
```

```
      if (acquired) lock.unlock();          <-- Only unlock if locked
    }
  }
}
```

If you run the code in listing 4.5, you'll see that it seems to resolve the deadlock, but only sometimes. You'll see the "received confirm of update" text, but only some of the time.

In fact, the deadlock hasn't really been resolved, because if the initial lock is obtained (in `propagateUpdate()`) the thread calls `confirmUpdate()` and never releases the first lock until completion. If both threads manage to acquire their first lock before either can call `confirmUpdate()`, the threads will still be deadlocked.

The real solution is to ensure that if the attempt to get the second lock fails, the thread should release the lock it's holding and wait briefly, as shown in the next listing. This gives the other threads a chance to get a complete set of the locks needed to progress.

Listing 4.6 Fixing deadlock

```
public void propagateUpdate(Update upd_, MicroBlogNode backup_) {
  boolean acquired = false;
  boolean done = false;

  while (!done) {
    int wait = (int)(Math.random() * 10);
    try {
      acquired = lock.tryLock(wait, TimeUnit.MILLISECONDS);
      if (acquired) {
        System.out.println(ident +": recvd: "+
        ➥ upd_.getUpdateText() +" ; backup: "+backup_.getIdent());
        done = backupNode_.tryConfirmUpdate(this, update_);   <-- Examine return from tryConfirmUpdate()
      }
    } catch (InterruptedException e) {
    } finally {
      if (acquired) lock.unlock();
    }
    if (!done) try {
      Thread.sleep(wait);
    } catch (InterruptedException e) { }          <-- If not done, release lock and wait
  }
}

public boolean tryConfirmUpdate(MicroBlogNode other_, Update upd_) {
  boolean acquired = false;
  try {
    int wait = (int)(Math.random() * 10);
    acquired = lock.tryLock(wait, TimeUnit.MILLISECONDS);

    if (acquired) {
      long elapsed = System.currentTimeMillis() - startTime;
      System.out.println(ident +": recvd confirm: "+
      ➥ upd_.getUpdateText() +" from "+other_.getIdent()
      ➥ +" - took "+ elapsed +" millis");
```

```
      return true;
    }
  } catch (InterruptedException e) {
  } finally {
    if (acquired) lock.unlock();
  }

  return false;
}
```

In this version, you examine the return code of `tryConfirmUpdate()`. If it returns `false`, the original lock will be released. The thread will pause briefly, allowing the other thread to potentially acquire its lock.

Run this code a few times, and you should see that both threads are basically always able to progress—you've eliminated the deadlock. You may like to experiment with some different forms of the preceding versions of the deadlock code—the original, the flawed solution, and the corrected form. By playing with the code, you can get a better understanding of what is happening with the locks, and you can begin to build your intuition about how to avoid deadlock issues.

> **Why does the flawed attempt seem to work sometimes?**
> You've seen that the deadlock still exists, so what is it that causes the code in the flawed solution to sometimes succeed? The extra complexity in the code is the culprit. It affects the JVM's thread scheduler and makes it less easy to predict. This means that it will sometimes schedule the threads so that one of them (usually the first thread) is able to get into `confirmUpdate()` and acquire the second lock before the second thread can run. This is also possible in the original code, but much less likely.

We've only scratched the surface of the possibilities of `Lock`—there are a number of ways of producing more complex lock-like structures. One such concept, the latch, is our next topic.

4.3.3 CountDownLatch

The `CountDownLatch` is a simple synchronization pattern that allows for multiple threads to all agree on a minimum amount of preparation that must be done before any thread can pass a synchronization barrier.

This is achieved by providing an `int` value (the count) when constructing a new instance of `CountDownLatch`. After that point, two methods are used to control the latch: `countDown()` and `await()`. The former reduces the count by 1, and the latter causes the calling thread to wait until the count reaches 0 (it does nothing if the count is already 0 or less). This simple mechanism allows the minimum preparation pattern to be easily deployed.

In the following listing, a group of processing threads within a single process want to know that at least half of them have been properly initialized (assume that initialization

of a processing thread takes a certain amount of time) before the system as a whole starts sending updates to any of them.

Listing 4.7 Using latches to help with initialization

```
public static class ProcessingThread extends Thread {
  private final String ident;
  private final CountDownLatch latch;

  public ProcessingThread(String ident_, CountDownLatch cdl_) {
    ident = ident_;
    latch = cdl_;
  }
  public String getIdentifier() {
    return identifier;
  }
  public void initialize() {                   // initialize node
    latch.countDown();
  }
  public void run() {
    initialize();
  }
}

final int quorum = 1 + (int)(MAX_THREADS / 2);
final CountDownLatch cdl = new CountDownLatch(quorum);

final Set<ProcessingThread> nodes = new HashSet<>();
try {
  for (int i=0; i<MAX_THREADS; i++) {
    ProcessingThread local = new ProcessingThread("localhost:"+
      (9000 + i), cdl);
    nodes.add(local);
    local.start();
  }
  cdl.await();                                 // Begin sending—quorum reached
} catch (InterruptedException e) {
} finally {
}
```

In the code, you set up a latch with a quorum value. Once that many threads are initialized, you can start processing. Each thread will cause a `countDown()` once it has finished initialization, so the main thread need only wait until the quorum level has been reached before starting (and sending updates, although we omitted that part of the code).

The next class we'll discuss is one of the most useful classes in the multithreaded developer's toolkit: the `ConcurrentHashMap` from `java.util.concurrent`.

4.3.4 *ConcurrentHashMap*

The `ConcurrentHashMap` class provides a concurrent version of the standard `HashMap`. This is an improvement on the `synchronizedMap()` functionality provided in the `Collections` class, because those methods return collections that have more locking than is strictly necessary.

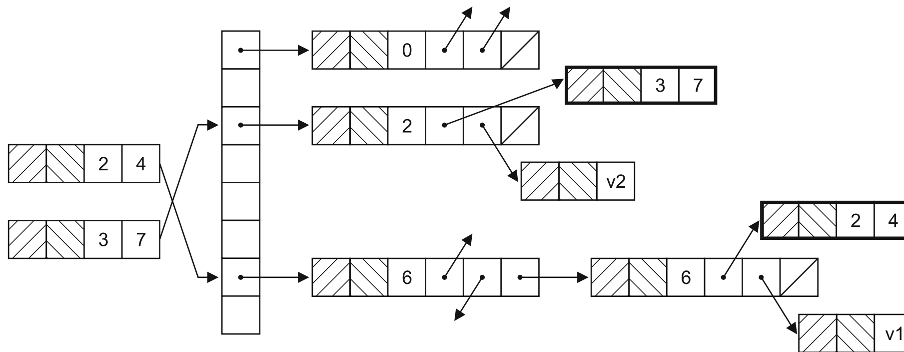


Figure 4.7 The classic view of a `HashMap`

As you can see from figure 4.7, the classic `HashMap` uses a function (the hash function) to determine which "bucket" it will store the key/value pair in. This is where the "hash" part of the class's name comes from. This suggests a rather straightforward multithreaded generalization—instead of needing to lock the whole structure when making a change, it's only necessary to lock the bucket that's being altered.

> **TIP** A well-written implementation of a concurrent `HashMap` will be essentially lock-free on reads, and for writes will only lock the bucket being modified. Java basically achieves this, but there are some additional low-level details that most developers won't need to worry about too much.

The `ConcurrentHashMap` class also implements the `ConcurrentMap` interface, which contains some new methods to provide truly atomic functionality:

- `putIfAbsent()`—Adds the key/value pair to the HashMap if the key isn't already present.
- `remove()`—Atomically removes the key/value pair only if the key is present and the value is equal to the current state.
- `replace()`—The API provides two different forms of this method for atomic replacement in the `HashMap`.

As an example, you can replace the `synchronized` methods in listing 4.1 with regular, unsynchronized access if you alter the `HashMap` called `arrivalTime` to be a `ConcurrentHashMap` as well. Notice the lack of locks in the following listing—there is no explicit synchronization at all.

Listing 4.8 Using `ConcurrentHashMap`

```
public class ExampleMicroBlogTimingNode implements SimpleMicroBlogNode {
  ...
private final Map<Update, Long> arrivalTime =
➥ new ConcurrentHashMap <>();
  ...
public void propagateUpdate(Update upd_) {
    arrivalTime.putIfAbsent(upd_, System.currentTimeMillis());
```

```
  }
  public boolean confirmUpdateReceived(Update upd_) {
    return arrivalTime.get(upd_) != null;
  }
}
```

The ConcurrentHashMap is one of the most useful classes in java.util.concurrent. It provides additional multithreaded safety and higher performance, and it has no serious drawbacks in normal usage. The counterpart to it for List is the CopyOnWriteArrayList, which we'll discuss next.

4.3.5 CopyOnWriteArrayList

As the name suggests, the CopyOnWriteArrayList class is a replacement for the standard ArrayList class. CopyOnWriteArrayList has been made threadsafe by the addition of copy-on-write semantics, which means that any operations that mutate the list will create a new copy of the array backing the list (as shown in figure 4.8). This also means that any iterators formed don't have to worry about any modifications that they didn't expect.

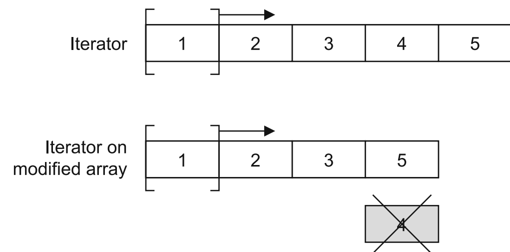


Figure 4.8 Copy-on-write array

This approach to shared data is ideal when a quick, consistent snapshot of data (which may occasionally be different between readers) is more important than perfect synchronization (and the attendant performance hit). This is often seen in non-mission-critical data.

Let's look at an example of copy-on-write in action. Consider a timeline of microblogging updates. This is a classic example of data that isn't 100 percent mission-critical and where a performant, self-consistent snapshot for each reader is preferred over total global consistency. This listing shows a holder class that represents an individual user's view of their timeline. (Then we'll use that class in listing 4.10 to show exactly how the copy-on-write behavior works.)

Listing 4.9 Copy-on-write example

```
public class MicroBlogTimeline {
  private final CopyOnWriteArrayList<Update> updates;
  private final ReentrantLock lock;
  private final String name;
  private Iterator<Update> it;

  public void addUpdate(Update update_) {
    updates.add(update_);
  }
  public void prep() {
    it = updates.iterator();
  }
```

Constructor omitted

Set up Iterator

```
  public void printTimeline() {
    lock.lock();                                    Need to
    try {                                           lock here
      if (it != null) {
        System.out.print(name+ ": ");
        while (it.hasNext()) {
          Update s = it.next();
          System.out.print(s+ ", ");
        }
        System.out.println();
      }
    } finally {
      lock.unlock();
    }
  }
}
```

This class is specifically designed to illustrate the behavior of an `Iterator` under copy-on-write semantics. You need to introduce locking in the print method to prevent the output being jumbled between the two threads, and to allow you to see the separate state of the two threads.

You can call the `MicroBlogTimeline` class from the code shown here.

Listing 4.10 Exposing copy-on-write behavior

```
final CountDownLatch firstLatch = new CountDownLatch(1);
final CountDownLatch secondLatch = new CountDownLatch(1);
final Update.Builder ub = new Update.Builder();

final List<Update> l = new CopyOnWriteArrayList<>();            ❶ Set up initial state
l.add(ub.author(new Author("Ben")).updateText("I like pie").build());
l.add(ub.author(new Author("Charles")).updateText(
➥ "I like ham on rye").build());

ReentrantLock lock = new ReentrantLock();
final MicroBlogTimeline tl1 = new MicroBlogTimeline("TL1", l, lock);
final MicroBlogTimeline tl22 = new MicroBlogTimeline("TL2", l, lock);

Thread t1 = new Thread() {
  public void run() {
    l.add(ub.author(new Author("Jeffrey")).updateText(
    ➥ "I like a lot of things").build());
    tl1.prep();
    firstLatch.countDown();
    try { secondLatch.await(); }
    ➥ catch (InterruptedException e) { }
    tl1.printTimeline();
  }
};

Thread t2 = new Thread(){                                Enforce strict
  public void run(){                                     event ordering
    try {                                                with latches
      firstLatch.await();
      l.add(ub.author(new Author("Gavin")).updateText(
      ➥ "I like otters").build());
```

```
            tl2.prep();
            secondLatch.countDown();
        } catch (InterruptedException e) { }
        tl2.printTimeline();
    }
};
t1.start();
t2.start();
```

Enforce strict event ordering with latches

There is a lot of scaffolding in the listing—unfortunately this is difficult to avoid. There are quite a few things to notice about this code:

- `CountDownLatch` is used to maintain close control over what is happening between the two threads.
- If the `CopyOnWriteArrayList` was replaced with an ordinary `List` (❶), the result would be a `ConcurrentModificationException`.
- This is also an example of a `Lock` object being shared between two threads to control access to a shared resource (in this case, `STDOUT`). This code would be much messier if expressed in the block-structured view.

The output of this code will look like this:

```
TL2: Update [author=Author [name=Ben], updateText=I like pie, createTime=0],
     Update [author=Author [name=Charles], updateText=I like ham on rye,
     createTime=0], Update [author=Author [name=Jeffrey], updateText=I like a
     lot of things, createTime=0], Update [author=Author [name=Gavin],
     updateText=I like otters, createTime=0],

TL1: Update [author=Author [name=Ben], updateText=I like pie, createTime=0],
     Update [author=Author [name=Charles], updateText=I like ham on rye,
     createTime=0], Update [author=Author [name=Jeffrey], updateText=I like a
     lot of things, createTime=0],
```

As you can see, the second output line (tagged as `TL1`) is missing the final update (the one that mentions otters), despite the fact that the latching meant that `mbex1` was accessed after the list had been modified. This demonstrates that the `Iterator` contained in `mbex1` was copied by `mbex2`, and that the addition of the final update was invisible to `mbex1`. This is the copy-on-write property that we want these objects to display.

Performance of CopyOnWriteArrayList

The use of the `CopyOnWriteArrayList` class does require a bit more thought than using `ConcurrentHashMap`, which really is a drop-in concurrent replacement for `HashMap`. This is because of performance issues—the copy-on-write property means that if the list is altered while a read or a traversal is taking place, the entire array must be copied.

This means that if changes to the list are common, compared to read accesses, this approach won't necessarily yield high performance. But as we'll say repeatedly in chapter 6, the only way to reliably get well-performing code is to test, retest, and measure the results.

The next major common building block of concurrent code in `java.util.concurrent` is the `Queue`. This is used to hand off work elements between threads, and it can be used as the basis for many flexible and reliable multithreaded designs.

4.3.6 Queues

The queue is a wonderful abstraction (and no, we're not just saying that because we live in London, the world capital of queuing). The queue provides a simple and reliable way to distribute processing resources to work units (or to assign work units to processing resources, depending on how you want to look at it).

There are a number of patterns in multithreaded Java programming that rely heavily on the thread-safe implementations of `Queue`, so it's important that you fully understand it. The basic `Queue` interface is in `java.util`, because it can be an important pattern even in single-threaded programming, but we'll focus on the multithreaded use cases and assume that you have already encountered queues in basic use cases.

One very common use case, and the one we'll focus on, is the use of a queue to transfer work units between threads. This pattern is often ideally suited for the simplest concurrent extension of `Queue`—the `BlockingQueue`.

BlockingQueues

The `BlockingQueue` is a queue that has two additional special properties:

- When trying to `put()` to the queue, it will cause the putting thread to wait for space to become available if the queue is full.
- When trying to `take()` from the queue, it will cause the taking thread to block if the queue is empty.

These two properties are very useful because if one thread (or pool of threads) is outstripping the ability of the other to keep up, the faster thread is forced to wait, thus regulating the overall system. This is illustrated in figure 4.9.

> **Two implementations of BlockingQueue**
>
> Java ships with two basic implementations of the `BlockingQueue` interface: the `LinkedBlockingQueue` and the `ArrayBlockingQueue`. They offer slightly different properties; for example, the array implementation is very efficient when an exact bound is known for the size of the queue, whereas the linked implementation may be slightly faster under some circumstances.

Using WorkUnit

The `Queue` interfaces are all generic—they're `Queue<E>`, `BlockingQueue<E>`, and so on. Although it may seem strange, it's sometimes wise to exploit this and introduce an artificial container class to wrap the items of work.

For example, if you have a class called `MyAwesomeClass` that represents the units of work that you want to process in a multithreaded way, then rather than having this,

```
BlockingQueue<MyAwesomeClass>
```

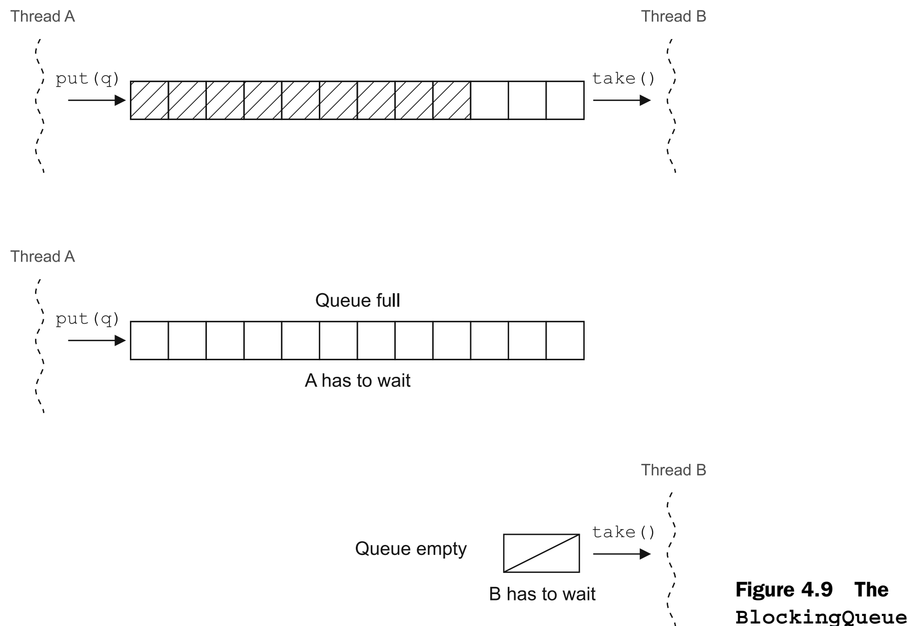


Figure 4.9 The `BlockingQueue`

it can be better to have this,

```
BlockingQueue<WorkUnit<MyAwesomeClass>>
```

where `WorkUnit` (or `QueueObject`, or whatever you want to call the container class) is a packaging interface or class that may look something like this:

```
public class WorkUnit<T> {
  private final T workUnit;

  public T getWork(){ return workUnit; }

  public WorkUnit(T workUnit_) {
    workUnit = workUnit_;
  }
}
```

The reason for doing this is that this level of indirection provides a place to add additional metadata without compromising the conceptual integrity of the contained type (`MyAwesomeClass` in this example).

This is surprisingly useful. Use cases where additional metadata is helpful are abundant. Here are a few examples:

- Testing (such as showing the change history for an object)
- Performance indicators (such as time of arrival or quality of service)
- Runtime system information (such as how this instance of `MyAwesomeClass` has been routed)

It can be much harder to add in this indirection after the fact. If you find that more metadata is required in certain circumstances, it can be a major refactoring job to add in what would have been a simple change in the `WorkUnit` class.

A BlockingQueue example

Let's see the `BlockingQueue` in action in a simple example—pets waiting to be seen by a veterinarian. This example represents a collection of pets that may be seen at a vet's surgery.

Listing 4.11 Modeling pets in Java

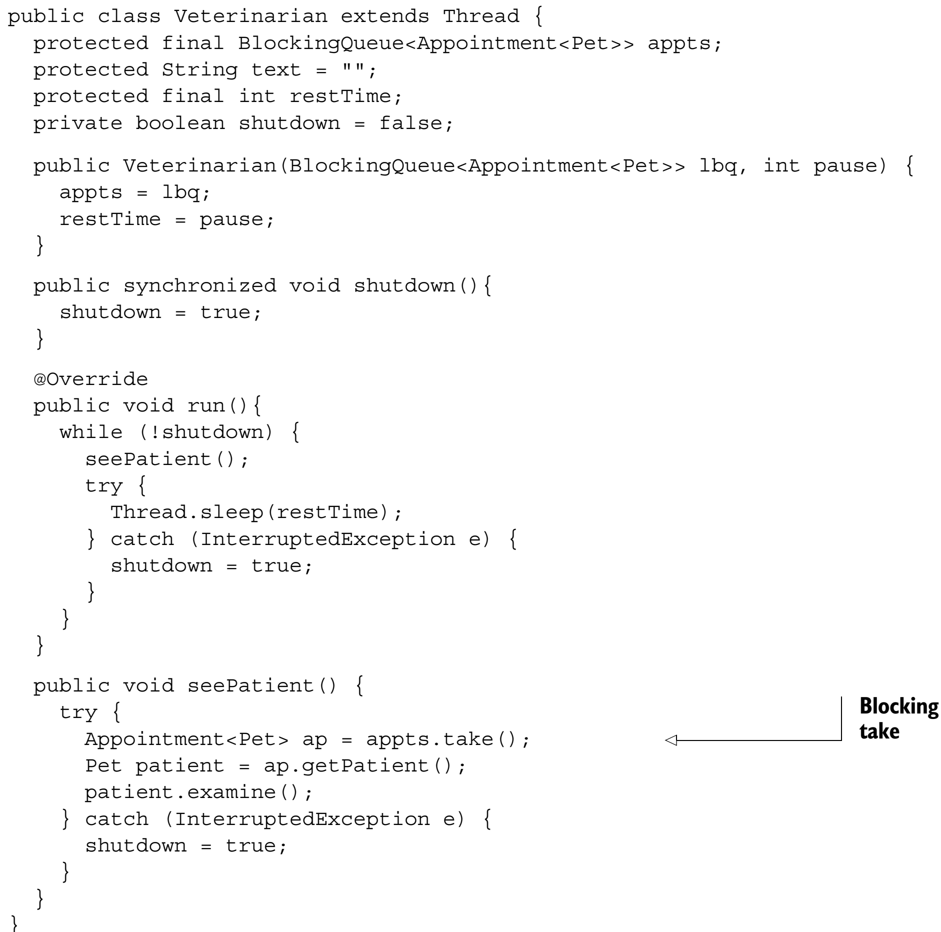
```
public abstract class Pet {
  protected final String name;

  public Pet(String name) {
    this.name = name;
  }
  public abstract void examine();
}

public class Cat extends Pet {
  public Cat(String name) {
    super(name);
  }
  public void examine(){
    System.out.println("Meow!");
  }
}

public class Dog extends Pet
  public Dog(String name) {
    super(name);
  }
  public void examine(){
    System.out.println("Woof!");
  }
}

public class Appointment<T> {
  private final T toBeSeen;

  public T getPatient(){ return toBeSeen; }

  public Appointment(T incoming) {
    toBeSeen = incoming;
  }
}
```

From this simple model, you can see that we can model the veterinarian's queue as `LinkedBlockingQueue<Appointment<Pet>>`, with the `Appointment` class taking the role of `WorkUnit`.

The veterinarian object is constructed with a queue (where appointments will be placed, by an object modeling a receptionist) and a pause time, which is the amount of downtime the veterinarian has between appointments.

We can model the veterinarian as shown in the next listing. As the thread runs, it repeatedly calls `seePatient()` in an infinite loop. Of course, in the real world, this would be unrealistic, because the veterinarian would probably want to go home for evenings and weekends, rather than hanging around the office waiting for sick animals to show up.

Listing 4.12 Modeling a veterinarian




```
public class Veterinarian extends Thread {
  protected final BlockingQueue<Appointment<Pet>> appts;
  protected String text = "";
  protected final int restTime;
  private boolean shutdown = false;

  public Veterinarian(BlockingQueue<Appointment<Pet>> lbq, int pause) {
    appts = lbq;
    restTime = pause;
  }

  public synchronized void shutdown(){
    shutdown = true;
  }

  @Override
  public void run(){
    while (!shutdown) {
      seePatient();
      try {
        Thread.sleep(restTime);
      } catch (InterruptedException e) {
        shutdown = true;
      }
    }
  }

  public void seePatient() {
    try {
      Appointment<Pet> ap = appts.take();
      Pet patient = ap.getPatient();
      patient.examine();
    } catch (InterruptedException e) {
      shutdown = true;
    }
  }
}
```

Inside the `seePatient()` method, the thread will dequeue appointments and examine the pets corresponding to each in turn, and will block if there are no appointments currently waiting on the queue.

FINE-GRAINED CONTROL OF BLOCKINGQUEUE

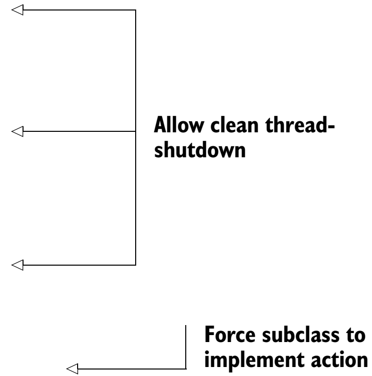
In addition to the simple `take()` and `offer()` API, `BlockingQueue` offers another way to interact with the queue that provides even more control, at the cost of a bit of extra complexity. This is the possibility of putting or taking with a timeout, to allow

the thread encountering issues to back out from its interaction with the queue and do something else instead.

In practice, this option isn't often used, but it can be a useful technique on occasion, so we'll demonstrate it for completeness. You can see it in the following example from our microblogging scenario.

Listing 4.13 `BlockingQueue` behavior example

```
public abstract class MicroBlogExampleThread extends Thread {
  protected final BlockingQueue<Update> updates;
  protected String text = "";
  protected final int pauseTime;
  private boolean shutdown = false;

  public MicroBlogExampleThread(BlockingQueue<Update> lbq_, int pause_) {
    updates = lbq_;
    pauseTime = pause_;
  }

  public synchronized void shutdown(){
    shutdown = true;
  }

  @Override
  public void run(){
    while (!shutdown) {
      doAction();
      try {
        Thread.sleep(pauseTime);
      } catch (InterruptedException e) {
        shutdown = true;
      }
    }
  }
  public abstract void doAction();
}

final Update.Builder ub = new Update.Builder();
final BlockingQueue<Update> lbq = new LinkedBlockingQueue<>(100);

MicroBlogExampleThread t1 = new MicroBlogExampleThread(lbq, 10) {
  public void doAction(){
    text = text + "X";
    Update u = ub.author(new Author("Tallulah")).updateText(text).build();
    boolean handed = false;
    try {
      handed = updates.offer(u, 100, TimeUnit.MILLISECONDS);
    } catch (InterruptedException e) {
    }
    if (!handed) System.out.println(
      "Unable to hand off Update to Queue due to timeout");
  }
};

MicroBlogExampleThread t2 = new MicroBlogExampleThread(lbq, 1000) {
  public void doAction(){
    Update u = null;
```



```
    try {
      u = updates.take();
    } catch (InterruptedException e) {
      return;
    }
  }
};
t1.start();
t2.start();
```

Running this example as is shows how the queue will quickly fill, meaning that the offering thread is outpacing the taking thread. Within a very short time, the message "`Unable to hand off Update to Queue due to timeout`" will start to appear.

This represents one extreme of the "connected thread pool" model—when the upstream thread pool is running quicker than the downstream one. This can be problematic, introducing such issues as an overflowing `LinkedBlockingQueue`. Alternatively, if there are more consumers than producers, the queue can empty. Fortunately Java 7 has a new twist on the `BlockingQueue` that can help—the `TransferQueue`.

TransferQueues—new in Java 7

Java 7 introduced the `TransferQueue`. This is essentially a `BlockingQueue` with an additional operation—`transfer()`. This operation will immediately transfer a work item to a receiver thread if one is waiting. Otherwise it will block until there is a thread available to take the item. This can be thought of as the "recorded delivery" option—the thread that was processing the item won't begin processing another item until it has handed off the current item. This allows the system to regulate the speed at which the upstream thread pool takes on new work.

It would also be possible to regulate this by using a blocking queue of bounded size, but the `TransferQueue` has a more flexible interface. In addition, your code may show a performance benefit by replacing a `BlockingQueue` with a `TransferQueue`. This is because the `TransferQueue` implementation has been written to take into account modern compiler and processor features and can operate with great efficiency. As with all discussions of performance, however, you must measure and prove benefits and not simply assume them. You should also be aware that Java 7 ships with only one implementation of `TransferQueue`—the linked version.

In the next code example, we'll look at how easy it is to drop in a `TransferQueue` as a replacement for a `BlockingQueue`. Just these simple changes to listing 4.13 will upgrade it to a `TransferQueue` implementation, as you can see here.

Listing 4.14 Replacing a `BlockingQueue` with a `TransferQueue`

```
public abstract class MicroBlogExampleThread extends Thread {
  protected final TransferQueue<Update> updates;
  ...

public MicroBlogExampleThread(TransferQueue<Update> lbq_, int pause_) {
    updates = lbq_;
    pauseTime = pause_;
  }
```

```
  ...
  }

final TransferQueue<Update> lbq = new LinkedTransferQueue<Update>(100);

MicroBlogExampleThread t1 = new MicroBlogExampleThread(lbq, 10) {
  public void doAction(){
    ...
try {
      handed = updates.tryTransfer(u, 100, TimeUnit.MILLISECONDS);
    } catch (InterruptedException e) {
    }
    ...
    }
};
```

This concludes our tour of the main building blocks that provide the raw materials for developing solid multithreaded applications. The next step is to combine them with the engines that drive concurrent code—the executor frameworks. These allow tasks to be scheduled and controlled, which lets you assemble efficient concurrent flows for handling work items and to build large multithreaded applications.

4.4 Controlling execution

We've spent some time in this chapter discussing work as abstract units. There's a subtlety to this, however. The part that we haven't mentioned so far is that these units are smaller than a `Thread`—they provide a way of running the computations contained in the work unit without having to spin up a new thread for each unit. This is often a much more efficient way of working with multithreaded code because it means that the `Thread` startup cost doesn't need to be paid for each unit. Instead, the threads that are actually executing the code are reused; after they finish processing one task, they will carry on with a new unit.

For the cost of a bit of extra complexity, you can access such abstractions as thread pools, worker and manager patterns and executors—some of the most versatile patterns in the developer's vocabulary. The classes and interfaces we'll focus on most closely are those that model tasks (`Callable`, `Future`, and `FutureTask`) and the executor classes, in particular `ScheduledThreadPoolExecutor`.

4.4.1 Modeling tasks

Our ultimate goal is to have tasks (or work units) that can be scheduled without spinning up a new thread for each one. Ultimately, this means that they have to be modeled as code that can be called (usually by an executor), rather than directly as a runnable thread.

We'll look at three different ways of modeling tasks—the `Callable` and `Future` interfaces and the `FutureTask` class.

CALLABLE INTERFACE

The `Callable` interface represents a very common abstraction. It represents a piece of code that can be called and that returns a result. Despite being a straightforward

idea, this is actually a subtle and powerful concept that can lead to some extremely useful patterns.

One typical use of a `Callable` is the anonymous implementation. The last line of this snippet sets `s` to be the value of `out.toString()`:

```
final MyObject out = getSampleObject();

Callable<String> cb = new Callable<String>() {
  public String call() throws Exception {
    return out.toString();
  }
};
String s = cb.call();
```

Think of an anonymous implementation of `Callable` as being a deferred invocation of the single abstract method, `call()`, which the implementation must provide.

Callable is an example of what is sometimes called a SAM type (short for "single abstract method")—this is the closest that Java 7 gets to having functions as first-class types. We'll talk more about the concept of functions as values or first-class types in later chapters, when we encounter them in non-Java languages.

FUTURE INTERFACE

The `Future` interface is used to represent an asynchronous task, in the sense of a future result from a task that may not have finished yet. We met these briefly in chapter 2 when we talked about NIO.2 and asynchronous I/O.

These are the primary methods on a `Future`:

- `get()`—This gets the result. If the result isn't yet available, `get()` will block until it is. There's also a version that takes a timeout, which won't block forever.
- `cancel()`—This allows the computation to be canceled before completion.
- `isDone()`—This allows the caller to determine whether the computation has finished.

The next snippet shows a sample use of a `Future` in a prime number finder:

```
Future<Long> fut = getNthPrime(1_000_000_000);

Long result = null;
while (result == null) {
  try {
    result = fut.get(60, TimeUnit.SECONDS);
  } catch (TimeoutException tox) { }
  System.out.println("Still not found the billionth prime!");
}
System.out.println("Found it: "+ result.longValue());
```

In this snippet, you should imagine that `getNthPrime()` returns a `Future` that is executing on some background thread (or even on multiple threads)—perhaps on one of the executor frameworks we'll discuss in the next subsection. Even on modern hardware, this calculation may be running for a long time—you may need the Future's `cancel()` method after all.

FutureTask class

The FutureTask class is a commonly used implementation of the Future interface, which also implements Runnable. As you'll see, this means that a FutureTask can be fed to executors, which is a crucial point. The interface is basically those of Future and Runnable combined: get(), cancel(), isDone(), isCancelled(), and run(), although this last method would be called by the executor, rather than directly.

Two convenience constructors for FutureTask are also provided: one that takes a Callable and one that takes a Runnable. The connections between these classes suggest a very flexible approach to tasks, allowing a job to be written as a Callable, then wrapped into a FutureTask that can then be scheduled (and cancelled if necessary) on an executor, due to the Runnable nature of FutureTask.

4.4.2 ScheduledThreadPoolExecutor

The ScheduledThreadPoolExecutor (STPE) is the backbone of the thread pool classes—it's versatile and as a result is quite common. The STPE takes in work in the form of tasks and schedules them on a pool of threads.

- The thread pools can be of a predefined size or adaptive.
- Tasks can be scheduled to execute periodically or just once.
- STPE extends the ThreadPoolExecutor class (which is similar, but lacks the periodic scheduling capabilities).

One of the most common patterns for medium- to large-scale multithreaded applications is of STPE thread pools of executing threads connected by the java.util.concurrent utility classes that we've already met (such as ConcurrentHashMap, CopyOnWriteArrayList, BlockingQueue).

STPE is only one of a number of related executors that can be obtained very easily by using factory methods available on the Executors class in java.util.concurrent. These factory methods are largely convenience methods; they allow the developer to access a typical configuration easily, while exposing the full available interface if required.

The next listing shows an example of periodic read. This is a common usage of newScheduledThreadPool(): the msgReader object is scheduled to poll() the queue, get the work item from the WorkUnit object on the queue, and then print it.

Listing 4.15 Periodic reads from an STPE

```
private ScheduledExecutorService stpe;
private ScheduledFuture<?> hndl;                    <-- Needed for cancellation

private BlockingQueue<WorkUnit<String>> lbq = new LinkedBlockingQueue<>();

private void run(){
  stpe = Executors.newScheduledThreadPool(2);       <-- Executors factory method

  final Runnable msgReader = new Runnable(){
    public void run(){
      String nextMsg = lbq.poll().getWork();
      if (nextMsg != null) System.out.println("Msg recvd: "+ nextMsg);
```

```
    }
  };
  hndl = stpe.scheduleAtFixedRate(msgReader, 10, 10,
  ➥ TimeUnit.MILLISECONDS);
}

public void cancel() {
  final ScheduledFuture<?> myHndl = hndl;

  stpe.schedule(new Runnable() {
    public void run() { myHndl.cancel(true); }      <-- Needed for cancellation
  }, 10, TimeUnit.MILLISECONDS);
}
```

In the example, an STPE wakes up a thread every 10 milliseconds and has it attempt to `poll()` from a queue. If the read returns `null` (because the queue is currently empty), nothing else happens and the thread goes back to sleep. If a work unit was received, the thread prints out the contents of the work unit.

Problems representing invocation with callable

There are a number of problems with the simple forms of `Callable`, `FutureTask`, and their relatives—notably that the type system gets in the way.

To see this, consider the case of trying to account for all possible signatures that an unknown method could have. `Callable` only provides a model of methods that take zero arguments. You'd need many different variations of `Callable` to account for all the possibilities.

In Java, you can work around this by being prescriptive about what method signatures exist in the systems you model. But as you'll see in part 3 of the book, dynamic languages don't share this static view of the world. This mismatch between type systems is a major theme to which we'll return. For now, just note that `Callable`, while useful, is a little too restrictive to build a general framework for modeling execution.

We'll now turn to one of the highlights of Java 7—the fork/join framework for lightweight concurrency. This new framework allows a wide range of concurrent problems to be handled even more efficiently than the executors we've seen in this section can do (which is no mean feat).

4.5 The fork/join framework

As we'll discuss in chapter 6, processor speeds (or, more properly, transistor counts on CPUs) have increased hugely in recent years. This has had the side effect that waiting for I/O is now a very common situation. This suggests that we could make better use of the processing capabilities inside our computers. The fork/join framework is an attempt to do just that—a way that also provides the biggest new additions to the concurrency arena in Java 7.

Fork/join is all about automatic scheduling of tasks on a thread pool that is invisible to the user. In order to do this, the tasks must be able to be broken up, in a way

that the user specifies. In many applications, fork/join has a notion of "small" and "large" tasks that is very natural for the framework.

Let's take a quick look at some of the headline facts and fundamentals related to fork/join.

- The fork/join framework introduces a new kind of executor service, called a `ForkJoinPool`.
- The `ForkJoinPool` service handles a unit of concurrency (the `ForkJoinTask`) that is "smaller" than a `Thread`.
- The `ForkJoinTask` is an abstraction that can be scheduled in a more lightweight manner by the `ForkJoinPool`.
- Fork/join usually makes use of two kinds of tasks (although they're both represented as instances of `ForkJoinTask`):
  - "Small" tasks are those that can be performed straightaway without consuming too much processor time.
  - "Large" tasks are those that need to be split up (possibly more than once) before they can be directly performed.
- The framework provides basic methods to support the splitting up of large tasks, and it has automatic scheduling and rescheduling.

One key feature of the framework is that it's expected that these lightweight tasks may well spawn other instances of `ForkJoinTask`, which will be scheduled by the same thread pool that executed their parent. This pattern is sometimes called *divide and conquer*.

We'll start with a simple example of using the fork/join framework, then briefly touch on the feature called "work-stealing," and finally discuss the features of problems that are well suited to parallel-processing techniques. The best way to get started with fork/join is with an example.

4.5.1 A simple fork/join example

As a simple example of what the fork/join framework can do, consider the following case: we have an array of updates to the microblogging service that may have arrived at different times, and we want to sort them by their arrival times, in order to generate timelines for the users, like the one you generated in listing 4.9.

To achieve this, we'll use a multithreaded sort, which is a variant of `MergeSort`. Listing 4.16 uses a specialized subclass of `ForkJoinTask`—the `RecursiveAction`. This is simpler than the general `ForkJoinTask` because it's explicit about not having any overall result (the updates will be reordered in place), and it emphasizes the recursive nature of the tasks.

The `MicroBlogUpdateSorter` class provides a way of ordering a list of updates using the `compareTo()` method on `Update` objects. The `compute()` method (which you have to implement because it's abstract in the `RecursiveAction` superclass) basically orders an array of microblog updates by the time of creation of an update.

Listing 4.16 Sorting with a RecursiveAction

```
public class MicroBlogUpdateSorter extends RecursiveAction {
  private static final int SMALL_ENOUGH = 32;        // 32 or fewer sorted serially
  private final Update[] updates;
  private final int start, end;
  private final Update[] result;

  public MicroBlogUpdateSorter(Update[] updates_) {
    this(updates_, 0, updates_.length);
  }

  public MicroBlogUpdateSorter(Update[] upds_,
  ➥ int startPos_, int endPos_) {
    start = startPos_;
    end = endPos_;
    updates = upds_;
    result = new Update[updates.length];
  }

  private void merge(MicroBlogUpdateSorter left_,
  ➥ MicroBlogUpdateSorter right_) {
    int i = 0;
    int lCt = 0;
    int rCt = 0;
    while (lCt < left_.size() && rCt < right_.size()) {
      result[i++] = (left_.result[lCt].compareTo(right_.result[rCt]) < 0)
        ? left_.result[lCt++]
        : right_.result[rCt++];
    }
    while (lCt < left_.size()) result[i++] = left_.result[lCt++];
    while (rCt < right_.size()) result[i++] = right_.result[rCt++];
  }

  public int size() {
    return end - start;
  }

  public Update[] getResult() {
    return result;
  }

  @Override
  protected void compute() {        // RecursiveAction method
    if (size() < SMALL_ENOUGH) {
      System.arraycopy(updates, start, result, 0, size());
      Arrays.sort(result, 0, size());
    } else {
      int mid = size() / 2;
      MicroBlogUpdateSorter left = new MicroBlogUpdateSorter(
      ➥ updates, start, start + mid);
      MicroBlogUpdateSorter right = new MicroBlogUpdateSorter(
      ➥ updates, start + mid, end);
      invokeAll(left, right);
      merge(left, right)
    }
  }
}
```

To use the sorter, you can drive it with some code like that shown next, which will generate some updates (that consist of a string of *X*s) and shuffle them, before passing them to the sorter. The output is the reordered updates.

Listing 4.17 Using the recursive sorter

```
List<Update> lu = new ArrayList<Update>();
String text = "";
final Update.Builder ub = new Update.Builder();
final Author a = new Author("Tallulah");

for (int i=0; i<256; i++) {
  text = text + "X";
  long now = System.currentTimeMillis();
  lu.add(ub.author(a).updateText(text).createTime(now).build());
  try {
    Thread.sleep(1);
  } catch (InterruptedException e) {
  }
}
Collections.shuffle(lu);
Update[] updates = lu.toArray(new Update[0]);

MicroBlogUpdateSorter sorter = new MicroBlogUpdateSorter(updates);
ForkJoinPool pool = new ForkJoinPool(4);
pool.invoke(sorter);

for (Update u: sorter.getResult()) {
  System.out.println(u);
}
```

Pass zero-sized array, save allocation

TimSort

With the arrival of Java 7, the default sort algorithm for arrays has changed. Previously it had been a form of QuickSort, but with Java 7 it has become “TimSort”—a version of MergeSort that has been hybridized with an insertion sort. TimSort was originally developed for Python by Tim Peters, and it has been the default sort in Python since version 2.3 (2002).

Want to see evidence of TimSort’s presence in Java 7? Just pass a `null` array of `Update` objects into listing 4.16. The comparisons inside the array sorting routine `Arrays.sort()` will fail with a null pointer exception, and you’ll see the TimSort classes in the stack trace.

4.5.2 ForkJoinTask and work stealing

`ForkJoinTask` is the superclass of `RecursiveAction`. It’s a generic class in the return type of an action (so `RecursiveAction` extends `ForkJoinTask<Void>`). This makes `ForkJoinTask` very suitable for map-reduce approaches that return a result from boiling down a dataset.

`ForkJoinTask`s are scheduled on a `ForkJoinPool`, which is a new type of executor service designed specifically for these lightweight tasks. The service maintains a list of

tasks for each thread, and if one task finishes, the service can reassign tasks from a fully loaded thread to an idle one.

The reason for this "work-stealing" algorithm is that without it, there could be scheduling problems related to the two sizes of tasks. In general, the two sizes of tasks will take very different lengths of time to run. For example, one thread may have a run queue consisting only of small tasks, whereas another may have only large tasks. If the small tasks run five times faster than large tasks, the thread with only small tasks may well find itself idle before the large-task thread finishes.

Work-stealing has been implemented precisely to work around this problem and allow all the pool threads to be utilized throughout the lifecycle of the fork/join job. It's completely automatic and you don't need to do anything specific to get the benefits of work-stealing. It's another example of the runtime environment doing more to help developers manage concurrency, rather than making it a manual task.

4.5.3 Parallelizing problems

The promise of fork/join is tantalizing, but in practice, not every problem is easily reduced to as simple a form as the multithreaded `MergeSort` in section 4.5.1.

These are some examples of problems well suited to the fork/join approach:

- Simulating the motion of large numbers of simple objects (such as particle effects)
- Log file analysis
- Data operations where a quantity is calculated from inputs (such as map-reduce operations)

Another way of looking at this is to say that a good problem for fork/join is one that can be broken up as in figure 4.10.

One practical way of determining whether a problem is likely to reduce well is to apply this checklist to the problem and its subtasks:

- Can the problem's subtasks work without explicit cooperation or synchronization between the subtasks?
- Do the subtasks calculate some value from their data without altering it (are they what a functional programmer would call "pure" functions)?
- Is divide-and-conquer natural for the subtasks? Is one outcome of a subtask the creation of more subtasks (which could be finer-grained than the task that spawned them)?

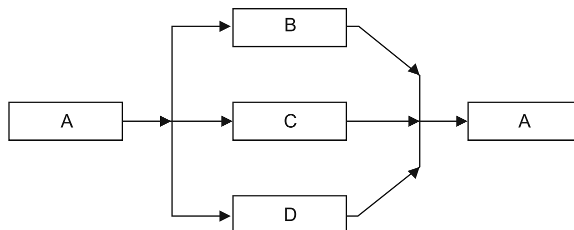


Figure 4.10 Fork and join

If the answer to the preceding questions is "Yes!" or "Mostly, but with edge cases," your problem may well be amenable to a fork/join approach. If, on the other hand, the answer to those questions is "Maybe" or "Not Really," you may well find that fork/join performs poorly, and a different synchronization approach may pay off better.

> **NOTE** The preceding checklist could be a useful way of testing to see if a problem (such as one of the kind often seen in Hadoop and NoSQL databases) could be well handled by fork/join.

Designing good multithreaded algorithms is hard, and fork/join doesn't work in every circumstance. It's very useful within its own domain of applicability, but in the end, you have to decide whether your problem fits within the framework, and if not, you must be prepared to develop your own solution, building on the superb toolbox of `java.util.concurrent`.

In the next section, we'll discuss the often-misunderstood details of the Java Memory Model (JMM). Many Java programmers are aware of the JMM and have been coding to their own understanding of it without ever being formally introduced to it. If that sounds like you, this new understanding will build upon your informal awareness and place it onto firm foundations. The JMM is quite an advanced topic, so you can skip it if you're in a hurry to get on to the next chapter.

4.6 The Java Memory Model (JMM)

The JMM is described in section 17.4 of the Java Language Specification (JLS). This is quite a formal part of the spec, and it describes the JMM in terms of synchronization actions and the mathematical construct known as a *partial order*. This is great from the point of view of language theorists and implementers of the Java spec (compiler and VM makers), but it's worse for application developers who need to understand the details of how their multithreaded code will execute.

Rather than repeat the formal details, we'll list the most important rules here in terms of a couple of basic concepts: the *Synchronizes-With* and *Happens-Before* relationships between blocks of code.

- *Happens-Before*—This relationship indicates that one block of code fully completes before the other can start.
- *Synchronizes-With*—This means that an action will synchronize its view of an object with main memory before continuing.

If you've studied formal approaches to OO programming, you may have heard the expressions *Has-A* and *Is-A* used to describe the building blocks of object orientation. Some developers find it useful to think of *Happens-Before* and *Synchronizes-With* as basic conceptual building blocks for understanding Java concurrency. This is by analogy with *Has-A* and *Is-A*, but there is no direct technical connection between the two sets of concepts.

In figure 4.11 you can see an example of a volatile write that Synchronizes-With a later read access (for the `println`).

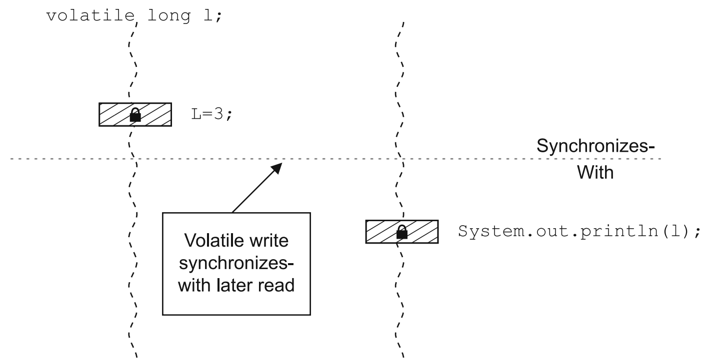


Figure 4.11 A Synchronizes-With example

The JMM has these main rules:

- An unlock operation on a monitor *Synchronizes-With* later lock operations.
- A write to a volatile variable *Synchronizes-With* later reads of the variable.
- If an action A *Synchronizes-With* action B, then A *Happens-Before* B.
- If A comes before B in program order, within a thread, then A *Happens-Before* B.

The general statement of the first two rules is that "releases happen before acquires." In other words, the locks that a thread holds when writing are released before the locks can be acquired by other operations (including reads).

There are additional rules, which are really about sensible behavior:

- The completion of a constructor *Happens-Before* the finalizer for that object starts to run (an object has to be fully constructed before it can be finalized).
- An action that starts a thread *Synchronizes-With* the first action of the new thread.
- `Thread.join()` *Synchronizes-With* the last (and all other) actions in the thread being joined.
- If X *Happens-Before* Y and Y *Happens-Before* Z then X *Happens-Before* Z (transitivity).

These simple rules define the whole of the platform's view of how memory and synchronization works. Figure 4.12 illustrates the transitivity rule.

> **NOTE** In practice, these rules are the minimum guarantees made by the JMM. Real JVMs may behave much better in practice than these guarantees suggest. This can be quite a pitfall for the developer because it's easy for the false sense of safety given by the behavior of a particular JVM to turn out to be just a quirk hiding an underlying concurrency bug.

From these minimum guarantees, it's easy to see why immutability is an important concept in concurrent Java programming. If objects can't be changed, there are no issues related to ensuring that changes are visible to all threads.

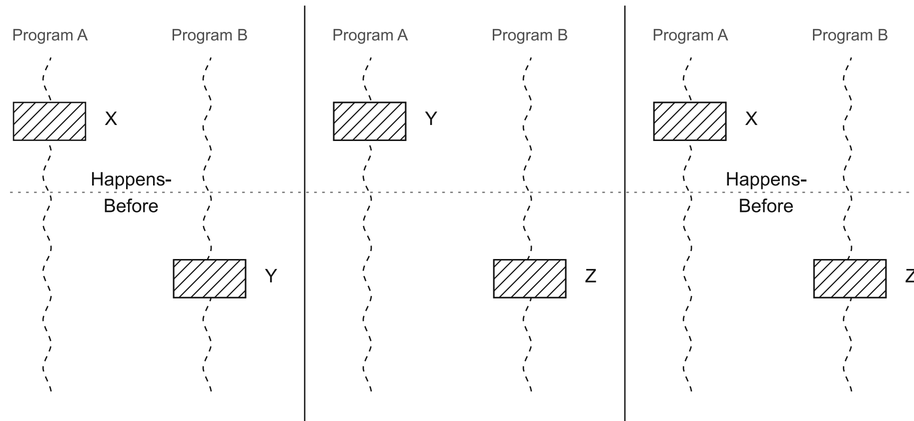


Figure 4.12 Transitivity of Happens-Before

4.7 Summary

Concurrency is one of the most important features of the Java platform, and a good developer will increasingly need a solid understanding of it. We've reviewed the underpinnings of Java's concurrency and the design forces that occur in multi-threaded systems. We've discussed the Java Memory Model and low-level details of how the platform implements concurrency.

More important to the modern Java developer, we've addressed the classes in `java.util.concurrent`, which should be your preferred toolkit for all new multi-threaded Java code. We've updated you with details of some of the brand new classes in Java 7, such as `LinkedTransferQueue` and the fork/join framework.

We hope we've prepared the ground for you to begin using the classes of `java.util.concurrent` in your own code. This is the single most important takeaway from this chapter. Although we've looked at some great theory, the most important part is the practical examples. Even if you just start with `ConcurrentHashMap` and the `Atomic` classes, you'll be using well-tested classes that can immediately provide real benefit to your code.

It's time to move on to the next big topic that will help you stand out as a Java developer. In the next chapter, you'll gain a firm grounding in another fundamental area of the platform—classloading and bytecode. This topic is at the heart of many of the platform's security and performance features, and it underpins many of the advanced techniques in the ecosystem. This makes it an ideal subject of study for the developer who wishes to gain an edge.

JAVA

The Well-Grounded **Java Developer**

B. J. Evans • M. Verburg

This book takes a fresh and practical look at new Java 7 features, new JVM languages, and the array of supporting technologies you need for the next generation of Java-based software.

You'll start with thorough coverage of Java 7 features like try-with-resources and NIO.2. You'll then explore a cross-section of emerging JVM-based languages, including Groovy, Scala, and Clojure. You will find clear examples that are practical and that help you dig into dozens of valuable development techniques showcasing modern approaches to the dev process, concurrency, performance, and much more.

What's Inside

- New Java 7 features
- Tutorials on Groovy, Scala, and Clojure
- Discovering multicore processing and concurrency
- Functional programming with new JVM languages
- Modern approaches to testing, build, and CI

Written for readers familiar with Java. No experience with Java 7 or new JVM languages required.

Ben Evans is the CEO of a Java performance firm and a member of the Java Community Process Executive Committee. **Martijn Verburg** is the CTO of a Java performance firm, coleader of the London JUG, and a popular conference speaker.

"How to become a well-grounded Java developer—and how to *stay* that way."

—From the Foreword by Dr. Heinz Kabutz
The Java Specialists' Newsletter

"At the cutting edge of Java development ... learn to speak Java 7 and next-gen languages."

—Paul Benedict
Corporate Personnel & Associates

"Buy this book for what's new in Java 7. Keep it open for lessons in expert Java."

—Stephen Harrison, PhD
FirstFuel Software

"A great collection of knowledge on the JVM platform."

—Rick Wagner, Red Hat

To download their free eBook in PDF, ePub and Kindle formats, owners of this book should visit manning.com/TheWell-GroundedJavaDeveloper

ISBN 13: 978-1-617290-06-0
ISBN 10: 1-617290-06-8

MANNING $49.99 / Can $52.99 [INCLUDING eBOOK]