CHAPTER 11

Concurrency

THREADS allow multiple activities to proceed concurrently. Concurrent programming is harder than single-threaded programming, because more things can go wrong, and failures can be hard to reproduce. You can't avoid concurrency. It is inherent in the platform and a requirement if you are to obtain good performance from multicore processors, which are now ubiquitous. This chapter contains advice to help you write clear, correct, well-documented concurrent programs.

Item 78: Synchronize access to shared mutable data

The `synchronized` keyword ensures that only a single thread can execute a method or block at one time. Many programmers think of synchronization solely as a means of *mutual exclusion*, to prevent an object from being seen in an inconsistent state by one thread while it's being modified by another. In this view, an object is created in a consistent state (Item 17) and locked by the methods that access it. These methods observe the state and optionally cause a *state transition*, transforming the object from one consistent state to another. Proper use of synchronization guarantees that no method will ever observe the object in an inconsistent state.

This view is correct, but it's only half the story. Without synchronization, one thread's changes might not be visible to other threads. Not only does synchronization prevent threads from observing an object in an inconsistent state, but it ensures that each thread entering a synchronized method or block sees the effects of all previous modifications that were guarded by the same lock.

The language specification guarantees that reading or writing a variable is *atomic* unless the variable is of type `long` or `double` [JLS, 17.4, 17.7]. In other words, reading a variable other than a `long` or `double` is guaranteed to return a value that was stored into that variable by some thread, even if multiple threads modify the variable concurrently and without synchronization.

You may hear it said that to improve performance, you should dispense with synchronization when reading or writing atomic data. This advice is dangerously wrong. While the language specification guarantees that a thread will not see an arbitrary value when reading a field, it does not guarantee that a value written by one thread will be visible to another. **Synchronization is required for reliable communication between threads as well as for mutual exclusion.** This is due to a part of the language specification known as the *memory model*, which specifies when and how changes made by one thread become visible to others [JLS, 17.4; Goetz06, 16].

The consequences of failing to synchronize access to shared mutable data can be dire even if the data is atomically readable and writable. Consider the task of stopping one thread from another. The libraries provide the `Thread.stop` method, but this method was deprecated long ago because it is inherently *unsafe*—its use can result in data corruption. **Do not use `Thread.stop`.** A recommended way to stop one thread from another is to have the first thread poll a `boolean` field that is initially `false` but can be set to `true` by the second thread to indicate that the first thread is to stop itself. Because reading and writing a `boolean` field is atomic, some programmers dispense with synchronization when accessing the field:

```
// Broken! - How long would you expect this program to run?
public class StopThread {
    private static boolean stopRequested;

    public static void main(String[] args)
            throws InterruptedException {
        Thread backgroundThread = new Thread(() -> {
            int i = 0;
            while (!stopRequested)
                i++;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
    }
}
```

You might expect this program to run for about a second, after which the main thread sets `stopRequested` to `true`, causing the background thread's loop to terminate. On my machine, however, the program *never* terminates: the background thread loops forever!

The problem is that in the absence of synchronization, there is no guarantee as to when, if ever, the background thread will see the change in the value of `stopRequested` made by the main thread. In the absence of synchronization, it's quite acceptable for the virtual machine to transform this code:

```
while (!stopRequested)
    i++;
```

into this code:

```
if (!stopRequested)
    while (true)
        i++;
```

This optimization is known as *hoisting*, and it is precisely what the OpenJDK Server VM does. The result is a *liveness failure*: the program fails to make progress. One way to fix the problem is to synchronize access to the `stopRequested` field. This program terminates in about one second, as expected:

```
// Properly synchronized cooperative thread termination
public class StopThread {
    private static boolean stopRequested;

    private static synchronized void requestStop() {
        stopRequested = true;
    }

    private static synchronized boolean stopRequested() {
        return stopRequested;
    }

    public static void main(String[] args)
            throws InterruptedException {
        Thread backgroundThread = new Thread(() -> {
            int i = 0;
            while (!stopRequested())
                i++;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(1);
        requestStop();
    }
}
```

Note that both the write method (`requestStop`) and the read method (`stopRequested`) are synchronized. It is *not* sufficient to synchronize only the write method! **Synchronization is not guaranteed to work unless both read and write operations are synchronized.** Occasionally a program that synchronizes only writes (or reads) may *appear* to work on some machines, but in this case, appearances are deceiving.

The actions of the synchronized methods in `StopThread` would be atomic even without synchronization. In other words, the synchronization on these methods is used *solely* for its communication effects, not for mutual exclusion. While the cost of synchronizing on each iteration of the loop is small, there is a correct alternative that is less verbose and whose performance is likely to be better. The locking in the second version of `StopThread` can be omitted if `stopRequested` is declared volatile. While the `volatile` modifier performs no mutual exclusion, it guarantees that any thread that reads the field will see the most recently written value:

```
// Cooperative thread termination with a volatile field
public class StopThread {
    private static volatile boolean stopRequested;

    public static void main(String[] args)
            throws InterruptedException {
        Thread backgroundThread = new Thread(() -> {
            int i = 0;
            while (!stopRequested)
                i++;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
    }
}
```

You do have to be careful when using `volatile`. Consider the following method, which is supposed to generate serial numbers:

```
// Broken - requires synchronization!
private static volatile int nextSerialNumber = 0;

public static int generateSerialNumber() {
    return nextSerialNumber++;
}
```

The intent of the method is to guarantee that every invocation returns a unique value (so long as there are no more than 2^{32} invocations). The method's state consists of a single atomically accessible field, `nextSerialNumber`, and all possible values of this field are legal. Therefore, no synchronization is necessary to protect its invariants. Still, the method won't work properly without synchronization.

The problem is that the increment operator (++) is not atomic. It performs *two* operations on the `nextSerialNumber` field: first it reads the value, and then it writes back a new value, equal to the old value plus one. If a second thread reads the field between the time a thread reads the old value and writes back a new one, the second thread will see the same value as the first and return the same serial number. This is a *safety failure*: the program computes the wrong results.

One way to fix `generateSerialNumber` is to add the `synchronized` modifier to its declaration. This ensures that multiple invocations won't be interleaved and that each invocation of the method will see the effects of all previous invocations. Once you've done that, you can and should remove the `volatile` modifier from `nextSerialNumber`. To bulletproof the method, use `long` instead of `int`, or throw an exception if `nextSerialNumber` is about to wrap.

Better still, follow the advice in Item 59 and use the class `AtomicLong`, which is part of `java.util.concurrent.atomic`. This package provides primitives for lock-free, thread-safe programming on single variables. While volatile provides only the communication effects of synchronization, this package also provides atomicity. This is exactly what we want for `generateSerialNumber`, and it is likely to outperform the synchronized version:

```
// Lock-free synchronization with java.util.concurrent.atomic
private static final AtomicLong nextSerialNum = new AtomicLong();

public static long generateSerialNumber() {
    return nextSerialNum.getAndIncrement();
}
```

The best way to avoid the problems discussed in this item is not to share mutable data. Either share immutable data (Item 17) or don't share at all. In other words, **confine mutable data to a single thread.** If you adopt this policy, it is important to document it so that the policy is maintained as your program evolves. It is also important to have a deep understanding of the frameworks and libraries you're using because they may introduce threads that you are unaware of.

It is acceptable for one thread to modify a data object for a while and then to share it with other threads, synchronizing only the act of sharing the object reference. Other threads can then read the object without further synchronization,

so long as it isn't modified again. Such objects are said to be *effectively immutable* [Goetz06, 3.5.4]. Transferring such an object reference from one thread to others is called *safe publication* [Goetz06, 3.5.3]. There are many ways to safely publish an object reference: you can store it in a static field as part of class initialization; you can store it in a volatile field, a final field, or a field that is accessed with normal locking; or you can put it into a concurrent collection (Item 81).

In summary, **when multiple threads share mutable data, each thread that reads or writes the data must perform synchronization.** In the absence of synchronization, there is no guarantee that one thread's changes will be visible to another thread. The penalties for failing to synchronize shared mutable data are liveness and safety failures. These failures are among the most difficult to debug. They can be intermittent and timing-dependent, and program behavior can vary radically from one VM to another. If you need only inter-thread communication, and not mutual exclusion, the `volatile` modifier is an acceptable form of synchronization, but it can be tricky to use correctly.

Item 79: Avoid excessive synchronization

Item 78 warns of the dangers of insufficient synchronization. This item concerns the opposite problem. Depending on the situation, excessive synchronization can cause reduced performance, deadlock, or even nondeterministic behavior.

To avoid liveness and safety failures, never cede control to the client within a synchronized method or block. In other words, inside a synchronized region, do not invoke a method that is designed to be overridden, or one provided by a client in the form of a function object (Item 24). From the perspective of the class with the synchronized region, such methods are *alien*. The class has no knowledge of what the method does and has no control over it. Depending on what an alien method does, calling it from a synchronized region can cause exceptions, deadlocks, or data corruption.

To make this concrete, consider the following class, which implements an *observable* set wrapper. It allows clients to subscribe to notifications when elements are added to the set. This is the *Observer* pattern [Gamma95]. For brevity's sake, the class does not provide notifications when elements are removed from the set, but it would be a simple matter to provide them. This class is implemented atop the reusable `ForwardingSet` from Item 18 (page 90):

```
// Broken - invokes alien method from synchronized block!
public class ObservableSet<E> extends ForwardingSet<E> {
    public ObservableSet(Set<E> set) { super(set); }

    private final List<SetObserver<E>> observers
            = new ArrayList<>();

    public void addObserver(SetObserver<E> observer) {
        synchronized(observers) {
            observers.add(observer);
        }
    }

    public boolean removeObserver(SetObserver<E> observer) {
        synchronized(observers) {
            return observers.remove(observer);
        }
    }

    private void notifyElementAdded(E element) {
        synchronized(observers) {
            for (SetObserver<E> observer : observers)
                observer.added(this, element);
        }
    }
```

```
    @Override public boolean add(E element) {
        boolean added = super.add(element);
        if (added)
            notifyElementAdded(element);
        return added;
    }

    @Override public boolean addAll(Collection<? extends E> c) {
        boolean result = false;
        for (E element : c)
            result |= add(element);  // Calls notifyElementAdded
        return result;
    }
}
```

Observers subscribe to notifications by invoking the `addObserver` method and unsubscribe by invoking the `removeObserver` method. In both cases, an instance of this *callback* interface is passed to the method.

```
@FunctionalInterface public interface SetObserver<E> {
    // Invoked when an element is added to the observable set
    void added(ObservableSet<E> set, E element);
}
```

This interface is structurally identical to `BiConsumer<ObservableSet<E>,E>`. We chose to define a custom functional interface because the interface and method names make the code more readable and because the interface could evolve to incorporate multiple callbacks. That said, a reasonable argument could also be made for using `BiConsumer` (Item 44).

On cursory inspection, `ObservableSet` appears to work fine. For example, the following program prints the numbers from `0` through `99`:

```
public static void main(String[] args) {
    ObservableSet<Integer> set =
            new ObservableSet<>(new HashSet<>());

    set.addObserver((s, e) -> System.out.println(e));

    for (int i = 0; i < 100; i++)
        set.add(i);
}
```

Now let's try something a bit fancier. Suppose we replace the `addObserver` call with one that passes an observer that prints the `Integer` value that was added to the set and removes itself if the value is `23`:

```
set.addObserver(new SetObserver<>() {
    public void added(ObservableSet<Integer> s, Integer e) {
        System.out.println(e);
        if (e == 23)
            s.removeObserver(this);
    }
});
```

Note that this call uses an anonymous class instance in place of the lambda used in the previous call. That is because the function object needs to pass itself to `s.removeObserver`, and lambdas cannot access themselves (Item 42).

You might expect the program to print the numbers `0` through `23`, after which the observer would unsubscribe and the program would terminate silently. In fact, it prints these numbers and then throws a `ConcurrentModificationException`. The problem is that `notifyElementAdded` is in the process of iterating over the `observers` list when it invokes the observer's `added` method. The `added` method calls the observable set's `removeObserver` method, which in turn calls the method `observers.remove`. Now we're in trouble. We are trying to remove an element from a list in the midst of iterating over it, which is illegal. The iteration in the `notifyElementAdded` method is in a synchronized block to prevent concurrent modification, but it doesn't prevent the iterating thread itself from calling back into the observable set and modifying its `observers` list.

Now let's try something odd: let's write an observer that tries to unsubscribe, but instead of calling `removeObserver` directly, it engages the services of another thread to do the deed. This observer uses an *executor service* (Item 80):

```
// Observer that uses a background thread needlessly
set.addObserver(new SetObserver<>() {
   public void added(ObservableSet<Integer> s, Integer e) {
      System.out.println(e);
      if (e == 23) {
         ExecutorService exec =
               Executors.newSingleThreadExecutor();
         try {
            exec.submit(() -> s.removeObserver(this)).get();
         } catch (ExecutionException | InterruptedException ex) {
            throw new AssertionError(ex);
         } finally {
            exec.shutdown();
         }
      }
   }
});
```

Incidentally, note that this program catches two different exception types in one catch clause. This facility, informally known as *multi-catch*, was added in Java 7. It can greatly increase the clarity and reduce the size of programs that behave the same way in response to multiple exception types.

When we run this program, we don't get an exception; we get a deadlock. The background thread calls `s.removeObserver`, which attempts to lock `observers`, but it can't acquire the lock, because the main thread already has the lock. All the while, the main thread is waiting for the background thread to finish removing the observer, which explains the deadlock.

This example is contrived because there is no reason for the observer to use a background thread to unsubscribe itself, but the problem is real. Invoking alien methods from within synchronized regions has caused many deadlocks in real systems, such as GUI toolkits.

In both of the previous examples (the exception and the deadlock) we were lucky. The resource that was guarded by the synchronized region (`observers`) was in a consistent state when the alien method (`added`) was invoked. Suppose you were to invoke an alien method from a synchronized region while the invariant protected by the synchronized region was temporarily invalid. Because locks in the Java programming language are *reentrant*, such calls won't deadlock. As in the first example, which resulted in an exception, the calling thread already holds the lock, so the thread will succeed when it tries to reacquire the lock, even though another conceptually unrelated operation is in progress on the data guarded by the lock. The consequences of such a failure can be catastrophic. In essence, the lock has failed to do its job. Reentrant locks simplify the construction of multithreaded object-oriented programs, but they can turn liveness failures into safety failures.

Luckily, it is usually not too hard to fix this sort of problem by moving alien method invocations out of synchronized blocks. For the `notifyElementAdded` method, this involves taking a "snapshot" of the `observers` list that can then be safely traversed without a lock. With this change, both of the previous examples run without exception or deadlock:

```
// Alien method moved outside of synchronized block - open calls
private void notifyElementAdded(E element) {
    List<SetObserver<E>> snapshot = null;
    synchronized(observers) {
        snapshot = new ArrayList<>(observers);
    }
    for (SetObserver<E> observer : snapshot)
        observer.added(this, element);
}
```

In fact, there's a better way to move the alien method invocations out of the synchronized block. The libraries provide a *concurrent collection* (Item 81) known as `CopyOnWriteArrayList` that is tailor-made for this purpose. This `List` implementation is a variant of `ArrayList` in which all modification operations are implemented by making a fresh copy of the entire underlying array. Because the internal array is never modified, iteration requires no locking and is very fast. For most uses, the performance of `CopyOnWriteArrayList` would be atrocious, but it's perfect for observer lists, which are rarely modified and often traversed.

The `add` and `addAll` methods of `ObservableSet` need not be changed if the list is modified to use `CopyOnWriteArrayList`. Here is how the remainder of the class looks. Notice that there is no explicit synchronization whatsoever:

```
// Thread-safe observable set with CopyOnWriteArrayList
private final List<SetObserver<E>> observers =
        new CopyOnWriteArrayList<>();

public void addObserver(SetObserver<E> observer) {
    observers.add(observer);
}

public boolean removeObserver(SetObserver<E> observer) {
    return observers.remove(observer);
}

private void notifyElementAdded(E element) {
    for (SetObserver<E> observer : observers)
        observer.added(this, element);
}
```

An alien method invoked outside of a synchronized region is known as an *open call* [Goetz06, 10.1.4]. Besides preventing failures, open calls can greatly increase concurrency. An alien method might run for an arbitrarily long period. If the alien method were invoked from a synchronized region, other threads would be denied access to the protected resource unnecessarily.

As a rule, you should do as little work as possible inside synchronized regions. Obtain the lock, examine the shared data, transform it as necessary, and drop the lock. If you must perform some time-consuming activity, find a way to move it out of the synchronized region without violating the guidelines in Item 78.

The first part of this item was about correctness. Now let's take a brief look at performance. While the cost of synchronization has plummeted since the early days of Java, it is more important than ever not to oversynchronize. In a multicore world, the real cost of excessive synchronization is not the CPU time spent getting locks; it is *contention*: the lost opportunities for parallelism and the delays

imposed by the need to ensure that every core has a consistent view of memory. Another hidden cost of oversynchronization is that it can limit the VM's ability to optimize code execution.

If you are writing a mutable class, you have two options: you can omit all synchronization and allow the client to synchronize externally if concurrent use is desired, or you can synchronize internally, making the class *thread-safe* (Item 82). You should choose the latter option only if you can achieve significantly higher concurrency with internal synchronization than you could by having the client lock the entire object externally. The collections in `java.util` (with the exception of the obsolete `Vector` and `Hashtable`) take the former approach, while those in `java.util.concurrent` take the latter (Item 81).

In the early days of Java, many classes violated these guidelines. For example, `StringBuffer` instances are almost always used by a single thread, yet they perform internal synchronization. It is for this reason that `StringBuffer` was supplanted by `StringBuilder`, which is just an unsynchronized `StringBuffer`. Similarly, it's a large part of the reason that the thread-safe pseudorandom number generator in `java.util.Random` was supplanted by the unsynchronized implementation in `java.util.concurrent.ThreadLocalRandom`. When in doubt, do *not* synchronize your class, but document that it is not thread-safe.

If you do synchronize your class internally, you can use various techniques to achieve high concurrency, such as lock splitting, lock striping, and nonblocking concurrency control. These techniques are beyond the scope of this book, but they are discussed elsewhere [Goetz06, Herlihy08].

If a method modifies a static field and there is any possibility that the method will be called from multiple threads, you *must* synchronize access to the field internally (unless the class can tolerate nondeterministic behavior). It is not possible for a multithreaded client to perform external synchronization on such a method, because unrelated clients can invoke the method without synchronization. The field is essentially a global variable even if it is private because it can be read and modified by unrelated clients. The `nextSerialNumber` field used by the method `generateSerialNumber` in Item 78 exemplifies this situation.

In summary, to avoid deadlock and data corruption, never call an alien method from within a synchronized region. More generally, keep the amount of work that you do from within synchronized regions to a minimum. When you are designing a mutable class, think about whether it should do its own synchronization. In the multicore era, it is more important than ever not to oversynchronize. Synchronize your class internally only if there is a good reason to do so, and document your decision clearly (Item 82).

Item 80: Prefer executors, tasks, and streams to threads

The first edition of this book contained code for a simple *work queue* [Bloch01, Item 49]. This class allowed clients to enqueue work for asynchronous processing by a background thread. When the work queue was no longer needed, the client could invoke a method to ask the background thread to terminate itself gracefully after completing any work that was already on the queue. The implementation was little more than a toy, but even so, it required a full page of subtle, delicate code, of the sort that is prone to safety and liveness failures if you don't get it just right. Luckily, there is no reason to write this sort of code anymore.

By the time the second edition of this book came out, `java.util.concurrent` had been added to Java. This package contains an *Executor Framework*, which is a flexible interface-based task execution facility. Creating a work queue that is better in every way than the one in the first edition of this book requires but a single line of code:

```
ExecutorService exec = Executors.newSingleThreadExecutor();
```

Here is how to submit a runnable for execution:

```
exec.execute(runnable);
```

And here is how to tell the executor to terminate gracefully (if you fail to do this, it is likely that your VM will not exit):

```
exec.shutdown();
```

You can do *many* more things with an executor service. For example, you can wait for a particular task to complete (with the `get` method, as shown in Item 79, page 319), you can wait for any or all of a collection of tasks to complete (using the `invokeAny` or `invokeAll` methods), you can wait for the executor service to terminate (using the `awaitTermination` method), you can retrieve the results of tasks one by one as they complete (using an `ExecutorCompletionService`), you can schedule tasks to run at a particular time or to run periodically (using a `ScheduledThreadPoolExecutor`), and so on.

If you want more than one thread to process requests from the queue, simply call a different static factory that creates a different kind of executor service called a *thread pool*. You can create a thread pool with a fixed or variable number of threads. The `java.util.concurrent.Executors` class contains static factories that provide most of the executors you'll ever need. If, however, you want some-

thing out of the ordinary, you can use the `ThreadPoolExecutor` class directly. This class lets you configure nearly every aspect of a thread pool's operation.

Choosing the executor service for a particular application can be tricky. For a small program, or a lightly loaded server, `Executors.newCachedThreadPool` is generally a good choice because it demands no configuration and generally "does the right thing." But a cached thread pool is not a good choice for a heavily loaded production server! In a cached thread pool, submitted tasks are not queued but immediately handed off to a thread for execution. If no threads are available, a new one is created. If a server is so heavily loaded that all of its CPUs are fully utilized and more tasks arrive, more threads will be created, which will only make matters worse. Therefore, in a heavily loaded production server, you are much better off using `Executors.newFixedThreadPool`, which gives you a pool with a fixed number of threads, or using the `ThreadPoolExecutor` class directly, for maximum control.

Not only should you refrain from writing your own work queues, but you should generally refrain from working directly with threads. When you work directly with threads, a `Thread` serves as both a unit of work and the mechanism for executing it. In the executor framework, the unit of work and the execution mechanism are separate. The key abstraction is the unit of work, which is the *task*. There are two kinds of tasks: `Runnable` and its close cousin, `Callable` (which is like `Runnable`, except that it returns a value and can throw arbitrary exceptions). The general mechanism for executing tasks is the *executor service*. If you think in terms of tasks and let an executor service execute them for you, you gain the flexibility to select an appropriate execution policy to meet your needs and to change the policy if your needs change. In essence, the Executor Framework does for execution what the Collections Framework did for aggregation.

In Java 7, the Executor Framework was extended to support fork-join tasks, which are run by a special kind of executor service known as a fork-join pool. A fork-join task, represented by a `ForkJoinTask` instance, may be split up into smaller subtasks, and the threads comprising a `ForkJoinPool` not only process these tasks but "steal" tasks from one another to ensure that all threads remain busy, resulting in higher CPU utilization, higher throughput, and lower latency. Writing and tuning fork-join tasks is tricky. Parallel streams (Item 48) are written atop fork join pools and allow you to take advantage of their performance benefits with little effort, assuming they are appropriate for the task at hand.

A complete treatment of the Executor Framework is beyond the scope of this book, but the interested reader is directed to *Java Concurrency in Practice* [Goetz06].

Item 81: Prefer concurrency utilities to `wait` and `notify`

The first edition of this book devoted an item to the correct use of `wait` and `notify` [Bloch01, Item 50]. Its advice is still valid and is summarized at end of this item, but this advice is far less important than it once was. This is because there is far less reason to use `wait` and `notify`. Since Java 5, the platform has provided higher-level concurrency utilities that do the sorts of things you formerly had to hand-code atop `wait` and `notify`. **Given the difficulty of using `wait` and `notify` correctly, you should use the higher-level concurrency utilities instead.**

The higher-level utilities in `java.util.concurrent` fall into three categories: the Executor Framework, which was covered briefly in Item 80; concurrent collections; and synchronizers. Concurrent collections and synchronizers are covered briefly in this item.

The concurrent collections are high-performance concurrent implementations of standard collection interfaces such as `List`, `Queue`, and `Map`. To provide high concurrency, these implementations manage their own synchronization internally (Item 79). Therefore, **it is impossible to exclude concurrent activity from a concurrent collection; locking it will only slow the program.**

Because you can't exclude concurrent activity on concurrent collections, you can't atomically compose method invocations on them either. Therefore, concurrent collection interfaces were outfitted with *state-dependent modify operations*, which combine several primitives into a single atomic operation. These operations proved sufficiently useful on concurrent collections that they were added to the corresponding collection interfaces in Java 8, using default methods (Item 21).

For example, `Map`'s `putIfAbsent(key, value)` method inserts a mapping for a key if none was present and returns the previous value associated with the key, or `null` if there was none. This makes it easy to implement thread-safe canonicalizing maps. This method simulates the behavior of `String.intern`:

```
// Concurrent canonicalizing map atop ConcurrentMap - not optimal
private static final ConcurrentMap<String, String> map =
        new ConcurrentHashMap<>();

public static String intern(String s) {
    String previousValue = map.putIfAbsent(s, s);
    return previousValue == null ? s : previousValue;
}
```

In fact, you can do even better. `ConcurrentHashMap` is optimized for retrieval operations, such as `get`. Therefore, it is worth invoking `get` initially and calling `putIfAbsent` only if `get` indicates that it is necessary:

```
// Concurrent canonicalizing map atop ConcurrentMap - faster!
public static String intern(String s) {
    String result = map.get(s);
    if (result == null) {
        result = map.putIfAbsent(s, s);
        if (result == null)
            result = s;
    }
    return result;
}
```

Besides offering excellent concurrency, `ConcurrentHashMap` is very fast. On my machine, the `intern` method above is over six times faster than `String.intern` (but keep in mind that `String.intern` must employ some strategy to keep from leaking memory in a long-lived application). Concurrent collections make synchronized collections largely obsolete. For example, **use `ConcurrentHashMap` in preference to `Collections.synchronizedMap`.** Simply replacing synchronized maps with concurrent maps can dramatically increase the performance of concurrent applications.

Some of the collection interfaces were extended with *blocking operations,* which wait (or *block*) until they can be successfully performed. For example, `BlockingQueue` extends `Queue` and adds several methods, including `take`, which removes and returns the head element from the queue, waiting if the queue is empty. This allows blocking queues to be used for *work queues* (also known as *producer-consumer queues*), to which one or more *producer threads* enqueue work items and from which one or more *consumer threads* dequeue and process items as they become available. As you'd expect, most `ExecutorService` implementations, including `ThreadPoolExecutor`, use a `BlockingQueue` (Item 80).

Synchronizers are objects that enable threads to wait for one another, allowing them to coordinate their activities. The most commonly used synchronizers are `CountDownLatch` and `Semaphore`. Less commonly used are `CyclicBarrier` and `Exchanger`. The most powerful synchronizer is `Phaser`.

Countdown latches are single-use barriers that allow one or more threads to wait for one or more other threads to do something. The sole constructor for `CountDownLatch` takes an `int` that is the number of times the `countDown` method must be invoked on the latch before all waiting threads are allowed to proceed.

It is surprisingly easy to build useful things atop this simple primitive. For example, suppose you want to build a simple framework for timing the concurrent execution of an action. This framework consists of a single method that takes an executor to execute the action, a concurrency level representing the number of actions to be executed concurrently, and a runnable representing the action. All of

the worker threads ready themselves to run the action before the timer thread starts the clock. When the last worker thread is ready to run the action, the timer thread "fires the starting gun," allowing the worker threads to perform the action. As soon as the last worker thread finishes performing the action, the timer thread stops the clock. Implementing this logic directly on top of `wait` and `notify` would be messy to say the least, but it is surprisingly straightforward on top of `CountDownLatch`:

```
// Simple framework for timing concurrent execution
public static long time(Executor executor, int concurrency,
            Runnable action) throws InterruptedException {
    CountDownLatch ready = new CountDownLatch(concurrency);
    CountDownLatch start = new CountDownLatch(1);
    CountDownLatch done  = new CountDownLatch(concurrency);

    for (int i = 0; i < concurrency; i++) {
        executor.execute(() -> {
            ready.countDown(); // Tell timer we're ready
            try {
                start.await(); // Wait till peers are ready
                action.run();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            } finally {
                done.countDown();  // Tell timer we're done
            }
        });
    }

    ready.await();     // Wait for all workers to be ready
    long startNanos = System.nanoTime();
    start.countDown(); // And they're off!
    done.await();      // Wait for all workers to finish
    return System.nanoTime() - startNanos;
}
```

Note that the method uses three countdown latches. The first, `ready`, is used by worker threads to tell the timer thread when they're ready. The worker threads then wait on the second latch, which is `start`. When the last worker thread invokes `ready.countDown`, the timer thread records the start time and invokes `start.countDown`, allowing all of the worker threads to proceed. Then the timer thread waits on the third latch, `done`, until the last of the worker threads finishes running the action and calls `done.countDown`. As soon as this happens, the timer thread awakens and records the end time.

A few more details bear noting. The executor passed to the `time` method must allow for the creation of at least as many threads as the given concurrency level, or the test will never complete. This is known as a *thread starvation deadlock* [Goetz06, 8.1.1]. If a worker thread catches an `InterruptedException`, it reasserts the interrupt using the idiom `Thread.currentThread().interrupt()` and returns from its `run` method. This allows the executor to deal with the interrupt as it sees fit. Note that `System.nanoTime` is used to time the activity. **For interval timing, always use `System.nanoTime` rather than `System.currentTimeMillis`.** `System.nanoTime` is both more accurate and more precise and is unaffected by adjustments to the system's real-time clock. Finally, note that the code in this example won't yield accurate timings unless `action` does a fair amount of work, say a second or more. Accurate microbenchmarking is notoriously hard and is best done with the aid of a specialized framework such as jmh [JMH].

This item only scratches the surface of what you can do with the concurrency utilities. For example, the three countdown latches in the previous example could be replaced by a single `CyclicBarrier` or `Phaser` instance. The resulting code would be a bit more concise but perhaps more difficult to understand.

While you should always use the concurrency utilities in preference to `wait` and `notify`, you might have to maintain legacy code that uses `wait` and `notify`. The `wait` method is used to make a thread wait for some condition. It must be invoked inside a synchronized region that locks the object on which it is invoked. Here is the standard idiom for using the `wait` method:

```
// The standard idiom for using the wait method
synchronized (obj) {
    while (<condition does not hold>)
        obj.wait(); // (Releases lock, and reacquires on wakeup)
    ... // Perform action appropriate to condition
}
```

Always use the wait loop idiom to invoke the `wait` method; never invoke it outside of a loop. The loop serves to test the condition before and after waiting.

Testing the condition before waiting and skipping the wait if the condition already holds are necessary to ensure liveness. If the condition already holds and the `notify` (or `notifyAll`) method has already been invoked before a thread waits, there is no guarantee that the thread will *ever* wake from the wait.

Testing the condition after waiting and waiting again if the condition does not hold are necessary to ensure safety. If the thread proceeds with the action when the condition does not hold, it can destroy the invariant guarded by the lock. There are several reasons a thread might wake up when the condition does not hold:

- Another thread could have obtained the lock and changed the guarded state between the time a thread invoked `notify` and the waiting thread woke up.
- Another thread could have invoked `notify` accidentally or maliciously when the condition did not hold. Classes expose themselves to this sort of mischief by waiting on publicly accessible objects. Any `wait` in a synchronized method of a publicly accessible object is susceptible to this problem.
- The notifying thread could be overly "generous" in waking waiting threads. For example, the notifying thread might invoke `notifyAll` even if only some of the waiting threads have their condition satisfied.
- The waiting thread could (rarely) wake up in the absence of a notify. This is known as a *spurious wakeup* [POSIX, 11.4.3.6.1; Java9-api].

A related issue is whether to use `notify` or `notifyAll` to wake waiting threads. (Recall that `notify` wakes a single waiting thread, assuming such a thread exists, and `notifyAll` wakes all waiting threads.) It is sometimes said that you should *always* use `notifyAll`. This is reasonable, conservative advice. It will always yield correct results because it guarantees that you'll wake the threads that need to be awakened. You may wake some other threads, too, but this won't affect the correctness of your program. These threads will check the condition for which they're waiting and, finding it false, will continue waiting.

As an optimization, you may choose to invoke `notify` instead of `notifyAll` if all threads that could be in the wait-set are waiting for the same condition and only one thread at a time can benefit from the condition becoming true.

Even if these preconditions are satisfied, there may be cause to use `notifyAll` in place of `notify`. Just as placing the `wait` invocation in a loop protects against accidental or malicious notifications on a publicly accessible object, using `notifyAll` in place of `notify` protects against accidental or malicious waits by an unrelated thread. Such waits could otherwise "swallow" a critical notification, leaving its intended recipient waiting indefinitely.

In summary, using `wait` and `notify` directly is like programming in "concurrency assembly language," as compared to the higher-level language provided by `java.util.concurrent`. **There is seldom, if ever, a reason to use `wait` and `notify` in new code.** If you maintain code that uses `wait` and `notify`, make sure that it always invokes `wait` from within a `while` loop using the standard idiom. The `notifyAll` method should generally be used in preference to `notify`. If `notify` is used, great care must be taken to ensure liveness.

Item 82: Document thread safety

How a class behaves when its methods are used concurrently is an important part of its contract with its clients. If you fail to document this aspect of a class's behavior, its users will be forced to make assumptions. If these assumptions are wrong, the resulting program may perform insufficient synchronization (Item 78) or excessive synchronization (Item 79). In either case, serious errors may result.

You may hear it said that you can tell if a method is thread-safe by looking for the `synchronized` modifier in its documentation. This is wrong on several counts. In normal operation, Javadoc does not include the `synchronized` modifier in its output, and with good reason. **The presence of the `synchronized` modifier in a method declaration is an implementation detail, not a part of its API.** It does not reliably indicate that a method is thread-safe.

Moreover, the claim that the presence of the `synchronized` modifier is sufficient to document thread safety embodies the misconception that thread safety is an all-or-nothing property. In fact, there are several levels of thread safety. **To enable safe concurrent use, a class must clearly document what level of thread safety it supports.** The following list summarizes levels of thread safety. It is not exhaustive but covers the common cases:

- **Immutable**—Instances of this class appear constant. No external synchronization is necessary. Examples include `String`, `Long`, and `BigInteger` (Item 17).
- **Unconditionally thread-safe**—Instances of this class are mutable, but the class has sufficient internal synchronization that its instances can be used concurrently without the need for any external synchronization. Examples include `AtomicLong` and `ConcurrentHashMap`.
- **Conditionally thread-safe**—Like unconditionally thread-safe, except that some methods require external synchronization for safe concurrent use. Examples include the collections returned by the `Collections.synchronized` wrappers, whose iterators require external synchronization.
- **Not thread-safe**—Instances of this class are mutable. To use them concurrently, clients must surround each method invocation (or invocation sequence) with external synchronization of the clients' choosing. Examples include the general-purpose collection implementations, such as `ArrayList` and `HashMap`.
- **Thread-hostile**—This class is unsafe for concurrent use even if every method invocation is surrounded by external synchronization. Thread hostility usually results from modifying static data without synchronization. No one writes a

thread-hostile class on purpose; such classes typically result from the failure to consider concurrency. When a class or method is found to be thread-hostile, it is typically fixed or deprecated. The `generateSerialNumber` method in Item 78 would be thread-hostile in the absence of internal synchronization, as discussed on page 322.

These categories (apart from thread-hostile) correspond roughly to the *thread safety annotations* in *Java Concurrency in Practice*, which are `Immutable`, `ThreadSafe`, and `NotThreadSafe` [Goetz06, Appendix A]. The unconditionally and conditionally thread-safe categories in the above taxonomy are both covered under the `ThreadSafe` annotation.

Documenting a conditionally thread-safe class requires care. You must indicate which invocation sequences require external synchronization, and which lock (or in rare cases, locks) must be acquired to execute these sequences. Typically it is the lock on the instance itself, but there are exceptions. For example, the documentation for `Collections.synchronizedMap` says this:

> It is imperative that the user manually synchronize on the returned map when iterating over any of its collection views:
>
> ```
> Map<K, V> m = Collections.synchronizedMap(new HashMap<>());
> Set<K> s = m.keySet();  // Needn't be in synchronized block
>     ...
> synchronized(m) {  // Synchronizing on m, not s!
>     for (K key : s)
>         key.f();
> }
> ```
>
> Failure to follow this advice may result in non-deterministic behavior.

The description of a class's thread safety generally belongs in the class's doc comment, but methods with special thread safety properties should describe these properties in their own documentation comments. It is not necessary to document the immutability of enum types. Unless it is obvious from the return type, static factories must document the thread safety of the returned object, as demonstrated by `Collections.synchronizedMap` (above).

When a class commits to using a publicly accessible lock, it enables clients to execute a sequence of method invocations atomically, but this flexibility comes at a price. It is incompatible with high-performance internal concurrency control, of the sort used by concurrent collections such as `ConcurrentHashMap`. Also, a client can mount a denial-of-service attack by holding the publicly accessible lock for a prolonged period. This can be done accidentally or intentionally.

To prevent this denial-of-service attack, you can use a *private lock object* instead of using synchronized methods (which imply a publicly accessible lock):

```
// Private lock object idiom - thwarts denial-of-service attack
private final Object lock = new Object();

public void foo() {
    synchronized(lock) {
        ...
    }
}
```

Because the private lock object is inaccessible outside the class, it is impossible for clients to interfere with the object's synchronization. In effect, we are applying the advice of Item 15 by encapsulating the lock object in the object it synchronizes.

Note that the `lock` field is declared `final`. This prevents you from inadvertently changing its contents, which could result in catastrophic unsynchronized access (Item 78). We are applying the advice of Item 17, by minimizing the mutability of the `lock` field. **Lock fields should always be declared `final`.** This is true whether you use an ordinary monitor lock (as shown above) or a lock from the `java.util.concurrent.locks` package.

The private lock object idiom can be used only on *unconditionally* thread-safe classes. Conditionally thread-safe classes can't use this idiom because they must document which lock their clients are to acquire when performing certain method invocation sequences.

The private lock object idiom is particularly well-suited to classes designed for inheritance (Item 19). If such a class were to use its instances for locking, a subclass could easily and unintentionally interfere with the operation of the base class, or vice versa. By using the same lock for different purposes, the subclass and the base class could end up "stepping on each other's toes." This is not just a theoretical problem; it happened with the `Thread` class [Bloch05, Puzzle 77].

To summarize, every class should clearly document its thread safety properties with a carefully worded prose description or a thread safety annotation. The `synchronized` modifier plays no part in this documentation. Conditionally thread-safe classes must document which method invocation sequences require external synchronization and which lock to acquire when executing these sequences. If you write an unconditionally thread-safe class, consider using a private lock object in place of synchronized methods. This protects you against synchronization interference by clients and subclasses and gives you more flexibility to adopt a sophisticated approach to concurrency control in a later release.

Item 83: Use lazy initialization judiciously

Lazy initialization is the act of delaying the initialization of a field until its value is needed. If the value is never needed, the field is never initialized. This technique is applicable to both static and instance fields. While lazy initialization is primarily an optimization, it can also be used to break harmful circularities in class and instance initialization [Bloch05, Puzzle 51].

As is the case for most optimizations, the best advice for lazy initialization is "don't do it unless you need to" (Item 67). Lazy initialization is a double-edged sword. It decreases the cost of initializing a class or creating an instance, at the expense of increasing the cost of accessing the lazily initialized field. Depending on what fraction of these fields eventually require initialization, how expensive it is to initialize them, and how often each one is accessed once initialized, lazy initialization can (like many "optimizations") actually harm performance.

That said, lazy initialization has its uses. If a field is accessed only on a fraction of the instances of a class *and* it is costly to initialize the field, then lazy initialization may be worthwhile. The only way to know for sure is to measure the performance of the class with and without lazy initialization.

In the presence of multiple threads, lazy initialization is tricky. If two or more threads share a lazily initialized field, it is critical that some form of synchronization be employed, or severe bugs can result (Item 78). All of the initialization techniques discussed in this item are thread-safe.

Under most circumstances, normal initialization is preferable to lazy initialization. Here is a typical declaration for a normally initialized instance field. Note the use of the `final` modifier (Item 17):

```
// Normal initialization of an instance field
private final FieldType field = computeFieldValue();
```

If you use lazy initialization to break an initialization circularity, use a synchronized accessor because it is the simplest, clearest alternative:

```
// Lazy initialization of instance field - synchronized accessor
private FieldType field;

private synchronized FieldType getField() {
    if (field == null)
        field = computeFieldValue();
    return field;
}
```

Both of these idioms (*normal initialization* and *lazy initialization with a synchronized accessor*) are unchanged when applied to static fields, except that you add the `static` modifier to the field and accessor declarations.

If you need to use lazy initialization for performance on a static field, use the *lazy initialization holder class idiom*. This idiom exploits the guarantee that a class will not be initialized until it is used [JLS, 12.4.1]. Here's how it looks:

```
// Lazy initialization holder class idiom for static fields
private static class FieldHolder {
    static final FieldType field = computeFieldValue();
}

private static FieldType getField() { return FieldHolder.field; }
```

When `getField` is invoked for the first time, it reads `FieldHolder.field` for the first time, causing the initialization of the `FieldHolder` class. The beauty of this idiom is that the `getField` method is not synchronized and performs only a field access, so lazy initialization adds practically nothing to the cost of access. A typical VM will synchronize field access only to initialize the class. Once the class is initialized, the VM patches the code so that subsequent access to the field does not involve any testing or synchronization.

If you need to use lazy initialization for performance on an instance field, use the *double-check idiom*. This idiom avoids the cost of locking when accessing the field after initialization (Item 79). The idea behind the idiom is to check the value of the field twice (hence the name *double-check*): once without locking and then, if the field appears to be uninitialized, a second time with locking. Only if the second check indicates that the field is uninitialized does the call initialize the field. Because there is no locking once the field is initialized, it is *critical* that the field be declared `volatile` (Item 78). Here is the idiom:

```
// Double-check idiom for lazy initialization of instance fields
private volatile FieldType field;

private FieldType getField() {
    FieldType result = field;
    if (result == null) {  // First check (no locking)
        synchronized(this) {
            if (field == null)  // Second check (with locking)
                field = result = computeFieldValue();
        }
    }
    return result;
}
```

This code may appear a bit convoluted. In particular, the need for the local variable (`result`) may be unclear. What this variable does is to ensure that `field` is read only once in the common case where it's already initialized. While not strictly necessary, this may improve performance and is more elegant by the standards applied to low-level concurrent programming. On my machine, the method above is about 1.4 times as fast as the obvious version without a local variable.

While you can apply the double-check idiom to static fields as well, there is no reason to do so: the lazy initialization holder class idiom is a better choice.

Two variants of the double-check idiom bear noting. Occasionally, you may need to lazily initialize an instance field that can tolerate repeated initialization. If you find yourself in this situation, you can use a variant of the double-check idiom that dispenses with the second check. It is, not surprisingly, known as the *single-check idiom*. Here is how it looks. Note that `field` is still declared `volatile`:

```
// Single-check idiom - can cause repeated initialization!
private volatile FieldType field;

private FieldType getField() {
    FieldType result = field;
    if (result == null)
        field = result = computeFieldValue();
    return result;
}
```

All of the initialization techniques discussed in this item apply to primitive fields as well as object reference fields. When the double-check or single-check idiom is applied to a numerical primitive field, the field's value is checked against `0` (the default value for numerical primitive variables) rather than `null`.

If you don't care whether *every* thread recalculates the value of a field, and the type of the field is a primitive other than `long` or `double`, then you may choose to remove the `volatile` modifier from the field declaration in the single-check idiom. This variant is known as the *racy single-check idiom.* It speeds up field access on some architectures, at the expense of additional initializations (up to one per thread that accesses the field). This is definitely an exotic technique, not for everyday use.

In summary, you should initialize most fields normally, not lazily. If you must initialize a field lazily in order to achieve your performance goals or to break a harmful initialization circularity, then use the appropriate lazy initialization technique. For instance fields, it is the double-check idiom; for static fields, the lazy initialization holder class idiom. For instance fields that can tolerate repeated initialization, you may also consider the single-check idiom.

Item 84: Don't depend on the thread scheduler

When many threads are runnable, the thread scheduler determines which ones get to run and for how long. Any reasonable operating system will try to make this determination fairly, but the policy can vary. Therefore, well-written programs shouldn't depend on the details of this policy. **Any program that relies on the thread scheduler for correctness or performance is likely to be nonportable.**

The best way to write a robust, responsive, portable program is to ensure that the average number of *runnable* threads is not significantly greater than the number of processors. This leaves the thread scheduler with little choice: it simply runs the runnable threads till they're no longer runnable. The program's behavior doesn't vary too much, even under radically different thread-scheduling policies. Note that the number of runnable threads isn't the same as the total number of threads, which can be much higher. Threads that are waiting are not runnable.

The main technique for keeping the number of runnable threads low is to have each thread do some useful work, and then wait for more. **Threads should not run if they aren't doing useful work.** In terms of the Executor Framework (Item 80), this means sizing thread pools appropriately [Goetz06, 8.2] and keeping tasks short, but not *too* short, or dispatching overhead will harm performance.

Threads should not *busy-wait*, repeatedly checking a shared object waiting for its state to change. Besides making the program vulnerable to the vagaries of the thread scheduler, busy-waiting greatly increases the load on the processor, reducing the amount of useful work that others can accomplish. As an extreme example of what *not* to do, consider this perverse reimplementation of `CountDownLatch`:

```
// Awful CountDownLatch implementation - busy-waits incessantly!
public class SlowCountDownLatch {
    private int count;

    public SlowCountDownLatch(int count) {
        if (count < 0)
            throw new IllegalArgumentException(count + " < 0");
        this.count = count;
    }

    public void await() {
        while (true) {
            synchronized(this) {
                if (count == 0)
                    return;
            }
        }
    }
```

```
    public synchronized void countDown() {
        if (count != 0)
            count--;
    }
}
```

On my machine, `SlowCountDownLatch` is about ten times slower than Java's `CountDownLatch` when 1,000 threads wait on a latch. While this example may seem a bit far-fetched, it's not uncommon to see systems with one or more threads that are unnecessarily runnable. Performance and portability are likely to suffer.

When faced with a program that barely works because some threads aren't getting enough CPU time relative to others, **resist the temptation to "fix" the program by putting in calls to `Thread.yield`.** You may succeed in getting the program to work after a fashion, but it will not be portable. The same `yield` invocations that improve performance on one JVM implementation might make it worse on a second and have no effect on a third. **`Thread.yield` has no testable semantics.** A better course of action is to restructure the application to reduce the number of concurrently runnable threads.

A related technique, to which similar caveats apply, is adjusting thread priorities. **Thread priorities are among the least portable features of Java.** It is not unreasonable to tune the responsiveness of an application by tweaking a few thread priorities, but it is rarely necessary and is not portable. It is unreasonable to attempt to solve a serious liveness problem by adjusting thread priorities. The problem is likely to return until you find and fix the underlying cause.

In summary, do not depend on the thread scheduler for the correctness of your program. The resulting program will be neither robust nor portable. As a corollary, do not rely on `Thread.yield` or thread priorities. These facilities are merely hints to the scheduler. Thread priorities may be used sparingly to improve the quality of service of an already working program, but they should never be used to "fix" a program that barely works.

This page intentionally left blank