High Level Concurrency Objects

So far, this lesson has focused on the low-level APIs that have been part of the Java platform from the very beginning. These APIs are adequate for very basic tasks, but higher-level building blocks are needed for more advanced tasks. This is especially true for massively concurrent applications that fully exploit today's multiprocessor and multi-core systems.

In this section we'll look at some of the high-level concurrency features introduced with version 5.0 of the Java platform. Most of these features are implemented in the new **java.util.concurrent packages**. There are also new concurrent data structures in the Java Collections Framework.

- **Lock objects** support locking idioms that simplify many concurrent applications.
- **Executors** define a high-level API for launching and managing threads. Executor implementations provided by java.util.concurrent provide thread pool management suitable for large-scale applications.
- **Concurrent collections** make it easier to manage large collections of data, and can greatly reduce the need for synchronization.
- **Atomic variables** have features that minimize synchronization and help avoid memory consistency errors.
- **ThreadLocalRandom** (in JDK 7) provides efficient generation of pseudorandom numbers from multiple threads.

Lock Objects
Synchronized code relies on a simple kind of reentrant lock. This kind of lock is easy to use, but has many limitations. More sophisticated locking idioms are supported by the java.util.concurrent.locks package. We won't examine this package in detail, but instead will focus on its most basic interface, Lock.

Lock objects work very much like the implicit locks used by synchronized code. As with implicit locks, only one thread can own a Lock object at a time. Lock objects also support a wait/notify mechanism, through their associated Condition objects.

The biggest advantage of Lock objects over implicit locks is their ability to back out of an attempt to acquire a lock. The tryLock method backs out if the lock is not available immediately or before a timeout expires (if specified). The lockInterruptibly method backs out if another thread sends an interrupt before the lock is acquired.

Let's use Lock objects to solve the deadlock problem we saw in Liveness. Alphonse and Gaston have trained themselves to notice when a friend is about to bow. We model this improvement by requiring that our Friend objects must acquire locks for both participants before proceeding with the bow. Here is the source code for the improved model, Safelock. To demonstrate the versatility of this idiom, we assume that Alphonse and Gaston are so infatuated with their newfound ability to bow safely that they can't stop bowing to each other:

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.Random;
```

```
public class Safelock {
    static class Friend {
        private final String name;
        private final Lock lock = new ReentrantLock();

        public Friend(String name) {
            this.name = name;
        }

        public String getName() {
            return this.name;
        }

        public boolean impendingBow(Friend bower) {
            Boolean myLock = false;
            Boolean yourLock = false;
            try {
                myLock = lock.tryLock();
                yourLock = bower.lock.tryLock();
            } finally {
                if (! (myLock && yourLock)) {
                    if (myLock) {
                        lock.unlock();
                    }
                    if (yourLock) {
                        bower.lock.unlock();
                    }
                }
            }
            return myLock && yourLock;
        }

        public void bow(Friend bower) {
            if (impendingBow(bower)) {
                try {
                    System.out.format("%s: %s has"
                        + " bowed to me!%n",
                        this.name, bower.getName());
                    bower.bowBack(this);
                } finally {
                    lock.unlock();
                    bower.lock.unlock();
                }
            } else {
                System.out.format("%s: %s started"
                    + " to bow to me, but saw that"
                    + " I was already bowing to"
                    + " him.%n",
                    this.name, bower.getName());
```

```
        }
    }

    public void bowBack(Friend bower) {
        System.out.format("%s: %s has" +
            " bowed back to me!%n",
            this.name, bower.getName());
    }
}

static class BowLoop implements Runnable {
    private Friend bower;
    private Friend bowee;

    public BowLoop(Friend bower, Friend bowee) {
        this.bower = bower;
        this.bowee = bowee;
    }

    public void run() {
        Random random = new Random();
        for (;;) {
            try {
                Thread.sleep(random.nextInt(10));
            } catch (InterruptedException e) {}
            bowee.bow(bower);
        }
    }
}


public static void main(String[] args) {
    final Friend alphonse =
        new Friend("Alphonse");
    final Friend gaston =
        new Friend("Gaston");
    new Thread(new BowLoop(alphonse, gaston)).start();
    new Thread(new BowLoop(gaston, alphonse)).start();
}
}
```

Executors

In all of the previous examples, there's a close connection between the task being done by a new thread, as defined by its Runnable object, and the thread itself, as defined by a Thread object. This works well for small applications, but in large-scale applications, it makes sense to separate thread management and creation from the rest of the application. Objects that encapsulate these functions are known as executors. The following subsections describe executors in detail.

- **Executor Interfaces** define the three executor object types.

- **Thread Pools** are the most common kind of executor implementation.
- **Fork/Join** is a framework (new in JDK 7) for taking advantage of multiple processors.

Executor Interfaces

The **java.util.concurrent package** defines three executor interfaces:

- **Executor**, a simple interface that supports launching new tasks.
- **ExecutorService**, a subinterface of Executor, which adds features that help manage the life-cycle, both of the individual tasks and of the executor itself.
- **ScheduledExecutorService**, a subinterface of ExecutorService, supports future and/or periodic execution of tasks.

Typically, variables that refer to executor objects are declared as one of these three interface types, not with an executor class type.

The Executor Interface

The Executor interface provides a single method, **execute**, designed to be a drop-in replacement for a common thread-creation idiom. If r is a Runnable object, and e is an Executor object you can replace

(new Thread(r)).start();
with
e.execute(r);

However, the definition of execute is less specific. The low-level idiom creates a new thread and launches it immediately. Depending on the Executor implementation, execute may do the same thing, but is more likely to use an existing worker thread to run r, or to place r in a queue to wait for a worker thread to become available. (We'll describe worker threads in the section on Thread Pools.)

The executor implementations in java.util.concurrent are designed to make full use of the more advanced ExecutorService and ScheduledExecutorService interfaces, although they also work with the base Executor interface.

The ExecutorService Interface

The ExecutorService interface supplements execute with a similar, but more versatile submit method. Like execute, **submit accepts Runnable objects, but also accepts Callable objects,** which allow the task to return a value. The submit method returns a Future object, which is used to retrieve the Callable return value and to manage the status of both Callable and Runnable tasks.

ExecutorService also provides methods for submitting large collections of Callable objects. Finally, ExecutorService provides a number of methods for managing the shutdown of the executor. To support immediate shutdown, tasks should handle interrupts correctly.

The ScheduledExecutorService Interface

The ScheduledExecutorService interface supplements the methods of its parent ExecutorService with schedule, which executes a Runnable or Callable task after a specified delay. In addition, the interface defines scheduleAtFixedRate and scheduleWithFixedDelay, which executes specified tasks repeatedly, at defined intervals.

Thread Pools
Most of the executor implementations in java.util.concurrent use thread pools, which consist of worker threads. This kind of thread exists separately from the Runnable and Callable tasks it executes and is often used to execute multiple tasks.

Using worker threads minimizes the overhead due to thread creation. Thread objects use a significant amount of memory, and in a large-scale application, allocating and deallocating many thread objects creates a significant memory management overhead.

One common type of thread pool is the fixed thread pool. This type of pool always has a specified number of threads running; if a thread is somehow terminated while it is still in use, it is automatically replaced with a new thread. Tasks are submitted to the pool via an internal queue, which holds extra tasks whenever there are more active tasks than threads.

An important advantage of the fixed thread pool is that applications using it degrade gracefully. To understand this, consider a web server application where each HTTP request is handled by a separate thread. If the application simply creates a new thread for every new HTTP request, and the system receives more requests than it can handle immediately, the application will suddenly stop responding to all requests when the overhead of all those threads exceed the capacity of the system. With a limit on the number of the threads that can be created, the application will not be servicing HTTP requests as quickly as they come in, but it will be servicing them as quickly as the system can sustain.

A simple way to create an executor that uses a fixed thread pool is to invoke the newFixedThreadPool factory method in java.util.concurrent.Executors This class also provides the following factory methods:

- **The newCachedThreadPool** method creates an executor with an expandable thread pool. This executor is suitable for applications that launch many short-lived tasks.
- **The newSingleThreadExecutor** method creates an executor that executes a single task at a time.
- Several factory methods are **ScheduledExecutorService** versions of the above executors.

If none of the executors provided by the above factory methods meet your needs, constructing instances of java.util.concurrent.ThreadPoolExecutor or java.util.concurrent.ScheduledThreadPoolExecutor will give you additional options.

Fork/Join
The fork/join framework is an implementation of the ExecutorService interface that helps you take advantage of multiple processors. It is designed for work that can be broken into smaller pieces recursively. The goal is to use all the available processing power to enhance the performance of your application.

As with any ExecutorService implementation, the fork/join framework distributes tasks to worker threads in a thread pool. The fork/join framework is distinct because it uses a work-stealing algorithm. Worker threads that run out of things to do can steal tasks from other threads that are still busy.

The center of the fork/join framework is the ForkJoinPool class, an extension of the AbstractExecutorService class. ForkJoinPool implements the core work-stealing algorithm and can execute ForkJoinTask processes.

Basic Use

The first step for using the fork/join framework is to write code that performs a segment of the work. Your code should look similar to the following pseudocode:

```
if (my portion of the work is small enough)
  do the work directly
else
  split my work into two pieces
  invoke the two pieces and wait for the results
```

Wrap this code in a ForkJoinTask subclass, typically using one of its more specialized types, either RecursiveTask (which can return a result) or RecursiveAction.

After your ForkJoinTask subclass is ready, create the object that represents all the work to be done and pass it to the invoke() method of a ForkJoinPool instance.

Blurring for Clarity

To help you understand how the fork/join framework works, consider the following example. Suppose that you want to blur an image. The original source image is represented by an array of integers, where each integer contains the color values for a single pixel. The blurred destination image is also represented by an integer array with the same size as the source.

Performing the blur is accomplished by working through the source array one pixel at a time. Each pixel is averaged with its surrounding pixels (the red, green, and blue components are averaged), and the result is placed in the destination array. Since an image is a large array, this process can take a long time. You can take advantage of concurrent processing on multiprocessor systems by implementing the algorithm using the fork/join framework. Here is one possible implementation:

```
public class ForkBlur extends RecursiveAction {
    private int[] mSource;
    private int mStart;
    private int mLength;
    private int[] mDestination;

    // Processing window size; should be odd.
    private int mBlurWidth = 15;

    public ForkBlur(int[] src, int start, int length, int[] dst) {
        mSource = src;
        mStart = start;
        mLength = length;
        mDestination = dst;
    }

    protected void computeDirectly() {
        int sidePixels = (mBlurWidth - 1) / 2;
        for (int index = mStart; index < mStart + mLength; index++) {
            // Calculate average.
```

```
            float rt = 0, gt = 0, bt = 0;
            for (int mi = -sidePixels; mi <= sidePixels; mi++) {
               int mindex = Math.min(Math.max(mi + index, 0),
                                     mSource.length - 1);
               int pixel = mSource[mindex];
               rt += (float)((pixel & 0x00ff0000) >> 16)
                     / mBlurWidth;
               gt += (float)((pixel & 0x0000ff00) >>  8)
                     / mBlurWidth;
               bt += (float)((pixel & 0x000000ff) >>  0)
                     / mBlurWidth;
            }

            // Reassemble destination pixel.
            int dpixel = (0xff000000     ) |
                 (((int)rt) << 16) |
                 (((int)gt) <<  8) |
                 (((int)bt) <<  0);
            mDestination[index] = dpixel;
         }
      }

  ...
```

Now you implement the abstract compute() method, which either performs the blur directly or splits it into two smaller tasks. A simple array length threshold helps determine whether the work is performed or split.

```
protected static int sThreshold = 100000;

protected void compute() {
   if (mLength < sThreshold) {
      computeDirectly();
      return;
   }

   int split = mLength / 2;

   invokeAll(new ForkBlur(mSource, mStart, split, mDestination),
             new ForkBlur(mSource, mStart + split, mLength - split,
                          mDestination));
}
```

If the previous methods are in a subclass of the RecursiveAction class, then setting up the task to run in a ForkJoinPool is straightforward, and involves the following steps:

Create a task that represents all of the work to be done.

```
   // source image pixels are in src
```

```
// destination image pixels are in dst
ForkBlur fb = new ForkBlur(src, 0, src.length, dst);

Create the ForkJoinPool that will run the task.

ForkJoinPool pool = new ForkJoinPool();

Run the task.

pool.invoke(fb);
```

For the full source code, including some extra code that creates the destination image file, see the ForkBlur example.

Standard Implementations

Besides using the fork/join framework to implement custom algorithms for tasks to be performed concurrently on a multiprocessor system (such as the ForkBlur.java example in the previous section), there are some generally useful features in Java SE which are already implemented using the fork/join framework. One such implementation, introduced in Java SE 8, is used by the java.util.Arrays class for its parallelSort() methods. These methods are similar to sort(), but leverage concurrency via the fork/join framework. Parallel sorting of large arrays is faster than sequential sorting when run on multiprocessor systems. However, how exactly the fork/join framework is leveraged by these methods is outside the scope of the Java Tutorials. For this information, see the Java API documentation.

Another implementation of the fork/join framework is used by methods in the java.util.streams package, which is part of Project Lambda scheduled for the Java SE 8 release. For more information, see the Lambda Expressions section.

Concurrent Collections

The java.util.concurrent package includes a number of additions to the Java Collections Framework. These are most easily categorized by the collection interfaces provided:

- **BlockingQueue** defines a first-in-first-out data structure that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue.
- **ConcurrentMap** is a subinterface of java.util.Map that defines useful atomic operations. These operations remove or replace a key-value pair only if the key is present, or add a key-value pair only if the key is absent. Making these operations atomic helps avoid synchronization. The standard general-purpose implementation of ConcurrentMap is ConcurrentHashMap, which is a concurrent analog of HashMap.
- **ConcurrentNavigableMap** is a subinterface of ConcurrentMap that supports approximate matches. The standard general-purpose implementation of ConcurrentNavigableMap is ConcurrentSkipListMap, which is a concurrent analog of TreeMap.

All of these collections help avoid Memory Consistency Errors by defining a happens-before relationship between an operation that adds an object to the collection with subsequent operations that access or remove that object.

Atomic Variables

The java.util.concurrent.atomic package defines classes that support atomic operations on single variables. All classes have get and set methods that work like reads and writes on volatile variables. That is, a set has a happens-before relationship with any subsequent get on the same variable. The atomic compareAndSet method also has these memory consistency features, as do the simple atomic arithmetic methods that apply to integer atomic variables.

To see how this package might be used, let's return to the Counter class we originally used to demonstrate thread interference:

```
class Counter {
   private int c = 0;

   public void increment() {
      c++;
   }

   public void decrement() {
      c--;
   }

   public int value() {
      return c;
   }

}
```

One way to make Counter safe from thread interference is to make its methods synchronized, as in SynchronizedCounter:

```
class SynchronizedCounter {
   private int c = 0;

   public synchronized void increment() {
      c++;
   }

   public synchronized void decrement() {
      c--;
   }

   public synchronized int value() {
      return c;
   }

}
```

For this simple class, synchronization is an acceptable solution. But for a more complicated class, we might want to avoid the liveness impact of unnecessary synchronization. Replacing the int field with an

AtomicInteger allows us to prevent thread interference without resorting to synchronization, as in AtomicCounter:

```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
   private AtomicInteger c = new AtomicInteger(0);

   public void increment() {
      c.incrementAndGet();
   }

   public void decrement() {
      c.decrementAndGet();
   }

   public int value() {
      return c.get();
   }
}
```

Concurrent Random Numbers

In JDK 7, java.util.concurrent includes a convenience class, ThreadLocalRandom, for applications that expect to use random numbers from multiple threads or ForkJoinTasks.

For concurrent access, using ThreadLocalRandom instead of Math.random() results in less contention and, ultimately, better performance.

All you need to do is call ThreadLocalRandom.current(), then call one of its methods to retrieve a random number. Here is one example:

```
int r = ThreadLocalRandom.current() .nextInt(4, 77);
```

Class ConcurrentHashMap<K,V>

java.lang.Object
  java.util.AbstractMap<K,V>
    java.util.concurrent.ConcurrentHashMap<K,V>

Type Parameters:
  K - the type of keys maintained by this map
  V - the type of mapped values

All Implemented Interfaces:
  Serializable, ConcurrentMap<K,V>, Map<K,V>

public class ConcurrentHashMap<K,V>
extends AbstractMap<K,V>
implements ConcurrentMap<K,V>, Serializable

A hash table supporting full concurrency of retrievals and high expected concurrency for updates. This class obeys the same functional specification as Hashtable, and includes versions of methods corresponding to each method of Hashtable. However, even though all operations are thread-safe, retrieval operations do not entail locking, and there is not any support for locking the entire table in a way that prevents all access. This class is fully interoperable with Hashtable in programs that rely on its thread safety but not on its synchronization details.

Retrieval operations (including get) generally do not block, so may overlap with update operations (including put and remove). Retrievals reflect the results of the most recently completed update operations holding upon their onset. (More formally, an update operation for a given key bears a happens-before relation with any (non-null) retrieval for that key reporting the updated value.) For aggregate operations such as putAll and clear, concurrent retrievals may reflect insertion or removal of only some entries. Similarly, Iterators, Spliterators and Enumerations return elements reflecting the state of the hash table at some point at or since the creation of the iterator/enumeration. They do not throw ConcurrentModificationException. However, iterators are designed to be used by only one thread at a time. Bear in mind that the results of aggregate status methods including size, isEmpty, and containsValue are typically useful only when a map is not undergoing concurrent updates in other threads. Otherwise the results of these methods reflect transient states that may be adequate for monitoring or estimation purposes, but not for program control.

The table is dynamically expanded when there are too many collisions (i.e., keys that have distinct hash codes but fall into the same slot modulo the table size), with the expected average effect of maintaining roughly two bins per mapping (corresponding to a 0.75 load factor threshold for resizing). There may be much variance around this average as mappings are added and removed, but overall, this maintains a commonly accepted time/space tradeoff for hash tables. However, resizing this or any other kind of hash table may be a relatively slow operation. When possible, it is a good idea to provide a size estimate as an optional initialCapacity constructor argument. An additional optional loadFactor constructor argument provides a further means of customizing initial table capacity by specifying the table density to be used in calculating the amount of space to allocate for the given number of elements. Also, for compatibility with previous versions of this class, constructors may optionally specify an expected concurrencyLevel as an additional hint for internal sizing. Note that using many keys with ex-

actly the same hashCode() is a sure way to slow down performance of any hash table. To ameliorate impact, when keys are Comparable, this class may use comparison order among keys to help break ties.

A Set projection of a ConcurrentHashMap may be created (using newKeySet() or newKeySet(int)), or viewed (using keySet(Object) when only keys are of interest, and the mapped values are (perhaps transiently) not used or all take the same mapping value.

A ConcurrentHashMap can be used as scalable frequency map (a form of histogram or multiset) by using LongAdder values and initializing via computeIfAbsent. For example, to add a count to a ConcurrentHashMap<String,LongAdder> freqs, you can use freqs.computeIfAbsent(k -> new LongAdder()).increment();

This class and its views and iterators implement all of the optional methods of the Map and Iterator interfaces.

Like Hashtable but unlike HashMap, this class does not allow null to be used as a key or value.

ConcurrentHashMaps support a set of sequential and parallel bulk operations that, unlike most Stream methods, are designed to be safely, and often sensibly, applied even with maps that are being concurrently updated by other threads; for example, when computing a snapshot summary of the values in a shared registry. There are three kinds of operation, each with four forms, accepting functions with Keys, Values, Entries, and (Key, Value) arguments and/or return values. Because the elements of a ConcurrentHashMap are not ordered in any particular way, and may be processed in different orders in different parallel executions, the correctness of supplied functions should not depend on any ordering, or on any other objects or values that may transiently change while computation is in progress; and except for forEach actions, should ideally be side-effect-free. Bulk operations on Map.Entry objects do not support method setValue.

forEach: Perform a given action on each element. A variant form applies a given transformation on each element before performing the action.

search: Return the first available non-null result of applying a given function on each element; skipping further search when a result is found.

reduce: Accumulate each element. The supplied reduction function cannot rely on ordering (more formally, it should be both associative and commutative). There are five variants:

Plain reductions. (There is not a form of this method for (key, value) function arguments since there is no corresponding return type.)

Mapped reductions that accumulate the results of a given function applied to each element.

Reductions to scalar doubles, longs, and ints, using a given basis value.

These bulk operations accept a parallelismThreshold argument. Methods proceed sequentially if the current map size is estimated to be less than the given threshold. Using a value of Long.MAX_VALUE suppresses all parallelism. Using a value of 1 results in maximal parallelism by partitioning into enough subtasks to fully utilize the ForkJoinPool.commonPool() that is used for all parallel computations. Normally, you would initially choose one of these extreme values, and then measure performance of using in-between values that trade off overhead versus throughput.

The concurrency properties of bulk operations follow from those of ConcurrentHashMap: Any non-null result returned from get(key) and related access methods bears a happens-before relation with the associated insertion or update. The result of any bulk operation reflects the composition of these per-element relations (but is not necessarily atomic with respect to the map as a whole unless it is somehow

known to be quiescent). Conversely, because keys and values in the map are never null, null serves as a reliable atomic indicator of the current lack of any result. To maintain this property, null serves as an implicit basis for all non-scalar reduction operations. For the double, long, and int versions, the basis should be one that, when combined with any other value, returns that other value (more formally, it should be the identity element for the reduction). Most common reductions have these properties; for example, computing a sum with basis 0 or a minimum with basis MAX_VALUE.

Search and transformation functions provided as arguments should similarly return null to indicate the lack of any result (in which case it is not used). In the case of mapped reductions, this also enables transformations to serve as filters, returning null (or, in the case of primitive specializations, the identity basis) if the element should not be combined. You can create compound transformations and filterings by composing them yourself under this "null means there is nothing there now" rule before using them in search or reduce operations.

Methods accepting and/or returning Entry arguments maintain key-value associations. They may be useful for example when finding the key for the greatest value. Note that "plain" Entry arguments can be supplied using new AbstractMap.SimpleEntry(k,v).

Bulk operations may complete abruptly, throwing an exception encountered in the application of a supplied function. Bear in mind when handling such exceptions that other concurrently executing functions could also have thrown exceptions, or would have done so if the first exception had not occurred.

Speedups for parallel compared to sequential forms are common but not guaranteed. Parallel operations involving brief functions on small maps may execute more slowly than sequential forms if the underlying work to parallelize the computation is more expensive than the computation itself. Similarly, parallelization may not lead to much actual parallelism if all processors are busy performing unrelated tasks.

All arguments to all task methods must be non-null.

This class is a member of the Java Collections Framework.

public interface BlockingQueue<E>
extends Queue<E>

A Queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.

BlockingQueue methods come in four forms, with different ways of handling operations that cannot be satisfied immediately, but may be satisfied at some point in the future: one throws an exception, the second returns a special value (either null or false, depending on the operation), the third blocks the current thread indefinitely until the operation can succeed, and the fourth blocks for only a given maximum time limit before giving up. These methods are summarized in the following table:

Summary of BlockingQueue methods	Throws exception	Special value	Blocks	Times out
Insert	add(e)	offer(e)	put(e)	offer(e, time, unit)
Remove	remove()	poll()	take()	poll(time, unit)
Examine	element()	peek()	not applicable	not applicable

A BlockingQueue does not accept null elements. Implementations throw NullPointerException on attempts to add, put or offer a null. A null is used as a sentinel value to indicate failure of poll operations.

A BlockingQueue may be capacity bounded. At any given time it may have a remainingCapacity beyond which no additional elements can be put without blocking. A BlockingQueue without any intrinsic capacity constraints always reports a remaining capacity of Integer.MAX_VALUE.

BlockingQueue implementations are designed to be used primarily for producer-consumer queues, but additionally support the Collection interface. So, for example, it is possible to remove an arbitrary element from a queue using remove(x). However, such operations are in general not performed very efficiently, and are intended for only occasional use, such as when a queued message is cancelled.

BlockingQueue implementations are thread-safe. All queuing methods achieve their effects atomically using internal locks or other forms of concurrency control. However, the bulk Collection operations addAll, containsAll, retainAll and removeAll are not necessarily performed atomically unless specified otherwise in an implementation. So it is possible, for example, for addAll(c) to fail (throwing an exception) after adding only some of the elements in c.

A BlockingQueue does not intrinsically support any kind of "close" or "shutdown" operation to indicate that no more items will be added. The needs and usage of such features tend to be implementation-dependent. For example, a common tactic is for producers to insert special end-of-stream or poison objects, that are interpreted accordingly when taken by consumers.

Usage example, based on a typical producer-consumer scenario. Note that a BlockingQueue can safely be used with multiple producers and multiple consumers.

```
class Producer implements Runnable {
  private final BlockingQueue queue;
  Producer(BlockingQueue q) { queue = q; }
  public void run() {
    try {
      while (true) { queue.put(produce()); }
    } catch (InterruptedException ex) { ... handle ...}
  }
  Object produce() { ... }
}

class Consumer implements Runnable {
  private final BlockingQueue queue;
  Consumer(BlockingQueue q) { queue = q; }
  public void run() {
    try {
      while (true) { consume(queue.take()); }
    } catch (InterruptedException ex) { ... handle ...}
  }
  void consume(Object x) { ... }
}

class Setup {
  void main() {
```

```
    BlockingQueue q = new SomeQueueImplementation();
    Producer p = new Producer(q);
    Consumer c1 = new Consumer(q);
    Consumer c2 = new Consumer(q);
    new Thread(p).start();
    new Thread(c1).start();
    new Thread(c2).start();
  }
}
```

Memory consistency effects: As with other concurrent collections, actions in a thread prior to placing an object into a BlockingQueue happen-before actions subsequent to the access or removal of that element from the BlockingQueue in another thread.

This interface is a member of the Java Collections Framework.

Since:
1.5

Method Summary
All MethodsInstance MethodsAbstract Methods Modifier and Type Method and Description
boolean add(E e)
Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an IllegalStateException if no space is currently available.
boolean contains(Object o)
Returns true if this queue contains the specified element.
int drainTo(Collection<? super E> c)
Removes all available elements from this queue and adds them to the given collection.
int drainTo(Collection<? super E> c, int maxElements)
Removes at most the given number of available elements from this queue and adds them to the given collection.
boolean offer(E e)
Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and false if no space is currently available.
boolean offer(E e, long timeout, TimeUnit unit)
Inserts the specified element into this queue, waiting up to the specified wait time if necessary for space to become available.
E poll(long timeout, TimeUnit unit)
Retrieves and removes the head of this queue, waiting up to the specified wait time if necessary for an element to become available.
void put(E e)
Inserts the specified element into this queue, waiting if necessary for space to become available.
int remainingCapacity()
Returns the number of additional elements that this queue can ideally (in the absence of memory or resource constraints) accept without blocking, or Integer.MAX_VALUE if there is no intrinsic limit.
boolean remove(Object o)
Removes a single instance of the specified element from this queue, if it is present.
E take()

Retrieves and removes the head of this queue, waiting if necessary until an element becomes available.

Methods inherited from interface java.util.Queue
element, peek, poll, remove
Methods inherited from interface java.util.Collection
addAll, clear, containsAll, equals, hashCode, isEmpty, iterator, parallelStream, removeAll, removeIf, retainAll, size, spliterator, stream, toArray, toArray
Methods inherited from interface java.lang.Iterable
forEach

Method Detail
add

boolean add(E e)

Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an IllegalStateException if no space is currently available. When using a capacity-restricted queue, it is generally preferable to use offer.

Specified by:
add in interface Collection<E>
Specified by:
add in interface Queue<E>
Parameters:
e - the element to add
Returns:
true (as specified by Collection.add(E))
Throws:
IllegalStateException - if the element cannot be added at this time due to capacity restrictions
ClassCastException - if the class of the specified element prevents it from being added to this queue
NullPointerException - if the specified element is null
IllegalArgumentException - if some property of the specified element prevents it from being added to this queue

offer

boolean offer(E e)

Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and false if no space is currently available. When using a capacity-restricted queue, this method is generally preferable to add(E), which can fail to insert an element only by throwing an exception.

Specified by:
offer in interface Queue<E>
Parameters:
e - the element to add
Returns:

true if the element was added to this queue, else false
Throws:
ClassCastException - if the class of the specified element prevents it from being added to this queue
NullPointerException - if the specified element is null
IllegalArgumentException - if some property of the specified element prevents it from being added to this queue

put

void put(E e)
throws InterruptedException

Inserts the specified element into this queue, waiting if necessary for space to become available.

Parameters:
e - the element to add
Throws:
InterruptedException - if interrupted while waiting
ClassCastException - if the class of the specified element prevents it from being added to this queue
NullPointerException - if the specified element is null
IllegalArgumentException - if some property of the specified element prevents it from being added to this queue

offer

boolean offer(E e,
long timeout,
TimeUnit unit)
throws InterruptedException

Inserts the specified element into this queue, waiting up to the specified wait time if necessary for space to become available.

Parameters:
e - the element to add
timeout - how long to wait before giving up, in units of unit
unit - a TimeUnit determining how to interpret the timeout parameter
Returns:
true if successful, or false if the specified waiting time elapses before space is available
Throws:
InterruptedException - if interrupted while waiting
ClassCastException - if the class of the specified element prevents it from being added to this queue
NullPointerException - if the specified element is null
IllegalArgumentException - if some property of the specified element prevents it from being added to this queue

take

E take()
throws InterruptedException

Retrieves and removes the head of this queue, waiting if necessary until an element becomes available.

Returns:
the head of this queue
Throws:
InterruptedException - if interrupted while waiting

poll

E poll(long timeout,
TimeUnit unit)
throws InterruptedException

Retrieves and removes the head of this queue, waiting up to the specified wait time if necessary for an element to become available.

Parameters:
timeout - how long to wait before giving up, in units of unit
unit - a TimeUnit determining how to interpret the timeout parameter
Returns:
the head of this queue, or null if the specified waiting time elapses before an element is available
Throws:
InterruptedException - if interrupted while waiting

remainingCapacity

int remainingCapacity()

Returns the number of additional elements that this queue can ideally (in the absence of memory or resource constraints) accept without blocking, or Integer.MAX_VALUE if there is no intrinsic limit.

Note that you cannot always tell if an attempt to insert an element will succeed by inspecting remainingCapacity because it may be the case that another thread is about to insert or remove an element.

Returns:
the remaining capacity

remove

boolean remove(Object o)

Removes a single instance of the specified element from this queue, if it is present. More formally, removes an element e such that o.equals(e), if this queue contains one or more such elements. Returns true if this queue contained the specified element (or equivalently, if this queue changed as a result of the call).

Specified by:
remove in interface Collection<E>
Parameters:
o - element to be removed from this queue, if present
Returns:
true if this queue changed as a result of the call
Throws:
ClassCastException - if the class of the specified element is incompatible with this queue (optional)
NullPointerException - if the specified element is null (optional)

contains

boolean contains(Object o)

Returns true if this queue contains the specified element. More formally, returns true if and only if this queue contains at least one element e such that o.equals(e).

Specified by:
contains in interface Collection<E>
Parameters:
o - object to be checked for containment in this queue
Returns:
true if this queue contains the specified element
Throws:
ClassCastException - if the class of the specified element is incompatible with this queue (optional)
NullPointerException - if the specified element is null (optional)

drainTo

int drainTo(Collection<? super E> c)

Removes all available elements from this queue and adds them to the given collection. This operation may be more efficient than repeatedly polling this queue. A failure encountered while attempting to add elements to collection c may result in elements being in neither, either or both collections when the associated exception is thrown. Attempts to drain a queue to itself result in IllegalArgumentException. Further, the behavior of this operation is undefined if the specified collection is modified while the operation is in progress.

Parameters:
c - the collection to transfer elements into
Returns:
the number of elements transferred

Throws:
UnsupportedOperationException - if addition of elements is not supported by the specified collection
ClassCastException - if the class of an element of this queue prevents it from being added to the specified collection
NullPointerException - if the specified collection is null
IllegalArgumentException - if the specified collection is this queue, or some property of an element of this queue prevents it from being added to the specified collection

drainTo

int drainTo(Collection<? super E> c,
int maxElements)

Removes at most the given number of available elements from this queue and adds them to the given collection. A failure encountered while attempting to add elements to collection c may result in elements being in neither, either or both collections when the associated exception is thrown. Attempts to drain a queue to itself result in IllegalArgumentException. Further, the behavior of this operation is undefined if the specified collection is modified while the operation is in progress.

Parameters:
c - the collection to transfer elements into
maxElements - the maximum number of elements to transfer
Returns:
the number of elements transferred
Throws:
UnsupportedOperationException - if addition of elements is not supported by the specified collection
ClassCastException - if the class of an element of this queue prevents it from being added to the specified collection
NullPointerException - if the specified collection is null
IllegalArgumentException - if the specified collection is this queue, or some property of an element of this queue prevents it from being added to the specified collection

Note that constructors cannot be synchronized — using the synchronized keyword with a constructor is a syntax error. Synchronizing constructors doesn't make sense, because only the thread that creates an object should have access to it while it is being constructed.

Warning: When constructing an object that will be shared between threads, be very careful that a reference to the object does not "leak" prematurely. For example, suppose you want to maintain a List called instances containing every instance of class. You might be tempted to add the following line to your constructor:

instances.add(this);

But then other threads can use instances to access the object before construction of the object is complete.

Locks In Synchronized Methods

When a thread invokes a synchronized method, it automatically acquires the **intrinsic lock** for that method's object and releases it when the method returns. The lock release occurs even if the return was caused by an uncaught exception.

You might wonder what happens when a **static synchronized method is invoked**, **since a static method is associated with a class, not an object. In this case, the thread acquires the intrinsic lock for the Class object associated with the class.** Thus access to class's static fields is controlled by a lock that's distinct from the lock for any instance of the class.

Reentrant Synchronization

Recall that a thread cannot acquire a lock owned by another thread. But a thread *can* acquire a lock that it already owns. Allowing a thread to acquire the same lock more than once enables *reentrant synchronization*. This describes a situation where synchronized code, directly or indirectly, invokes a method that also contains synchronized code, and both sets of code use the same lock. Without reentrant syn-

chronization, synchronized code would have to take many additional precautions to avoid having a thread cause itself to block.

Atomic Access

In programming, an *atomic* action is one that effectively happens all at once. An atomic action cannot stop in the middle: it either happens completely, or it doesn't happen at all. No side effects of an atomic action are visible until the action is complete.

We have already seen that an increment expression, such as c++, does not describe an atomic action. Even very simple expressions can define complex actions that can decompose into other actions. However, there are actions you can specify that are atomic:

- Reads and writes are atomic for reference variables and for most primitive variables (all types except long and double).
- Reads and writes are atomic for *all* variables declared volatile (*including* long and double variables).

Atomic actions cannot be interleaved, so they can be used without fear of thread interference. However, this does not eliminate all need to synchronize atomic actions, because memory consistency errors are still possible. Using volatile variables reduces the risk of memory consistency errors, because any write to a volatile variable establishes a happens-before relationship with subsequent reads of that same variable. This means that changes to a volatile variable are always visible to other threads. What's more, it also means that when a thread reads a volatile variable, it sees not just the latest change to the volatile, but also the side effects of the code that led up the change.

Using simple atomic variable access is more efficient than accessing these variables through synchronized code, but requires more care by the programmer to avoid memory consistency errors. Whether the extra effort is worthwhile depends on the size and complexity of the application.

Starvation and Livelock

Starvation and livelock are much less common a problem than deadlock, but are still problems that every designer of concurrent software is likely to encounter.

Starvation

Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads. For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

Livelock

A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then *livelock* may result. As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked — they are simply too busy responding to each other to resume work. This is comparable to two people attempting to pass each other in a corridor: Alphonse moves to his left to let Gaston pass, while Gaston moves to his right to let Alphonse pass. Seeing that they are still blocking each other, Alphone moves to his right, while Gaston moves to his left. They're still blocking each other, so...