Simpler, Faster, Better: Concurrency Utilities in JDK™ Software Version 5.0

Brian Goetz

Principal Consultant, Quiotix Corp

David Holmes

Staff Engineer, Sun Microsystems Inc.

TS-4915

Goal

Learn how to use the new concurrency utilities (the `java.util.concurrent` package) to replace error-prone or inefficient code and to better structure applications

Agenda

Overview of `java.util.concurrent`

Concurrent Collections

Threads Pools and Task Scheduling

Locks, Conditions and Synchronizers

Atomic Variables

Rationale for `java.util.concurrent`

Developing concurrent classes was just too hard

- The built-in concurrency primitives – **`wait()`**, **`notify()`**, and **`synchronized`** – are, well, primitive
  - Hard to use correctly
  - Easy to use incorrectly
  - Specified at too low a level for most applications
  - Can lead to poor performance if used incorrectly
- Too much wheel-reinventing!

Goals for `java.util.concurrent`

Simplify development of concurrent applications

- Provide a set of basic concurrency building blocks
- Something for everyone:
  - Make some problems trivial to solve by everyone
    - Develop thread-safe classes, such as servlets, built on concurrent building blocks like **`ConcurrentHashMap`**
  - Make some problems easier to solve by concurrent programmers
    - Develop concurrent applications using thread pools, barriers, latches, and blocking queues
  - Make some problems possible to solve by concurrency experts
    - Develop custom locking classes, lock-free algorithms

Agenda

Overview of `java.util.concurrent`

Concurrent Collections

Threads Pools and Task Scheduling

Locks, Conditions and Synchronizers

Atomic Variables

Concurrent Collections

Concurrent vs. Synchronized

- Pre Java 5: *thread-safe* but not *concurrent* classes
- Thread-safe synchronized collections:
  - **`Hashtable`**, **`Vector`**, **`Collections.synchronizedMap`**
  - Monitor is source of contention under concurrent access
  - Often require locking during iteration
- Concurrent collections:
  - Allow multiple operations to overlap each other
    - Big performance advantage
    - At the cost of some slight differences in semantics
  - Might not support atomic operations

Concurrent Collections

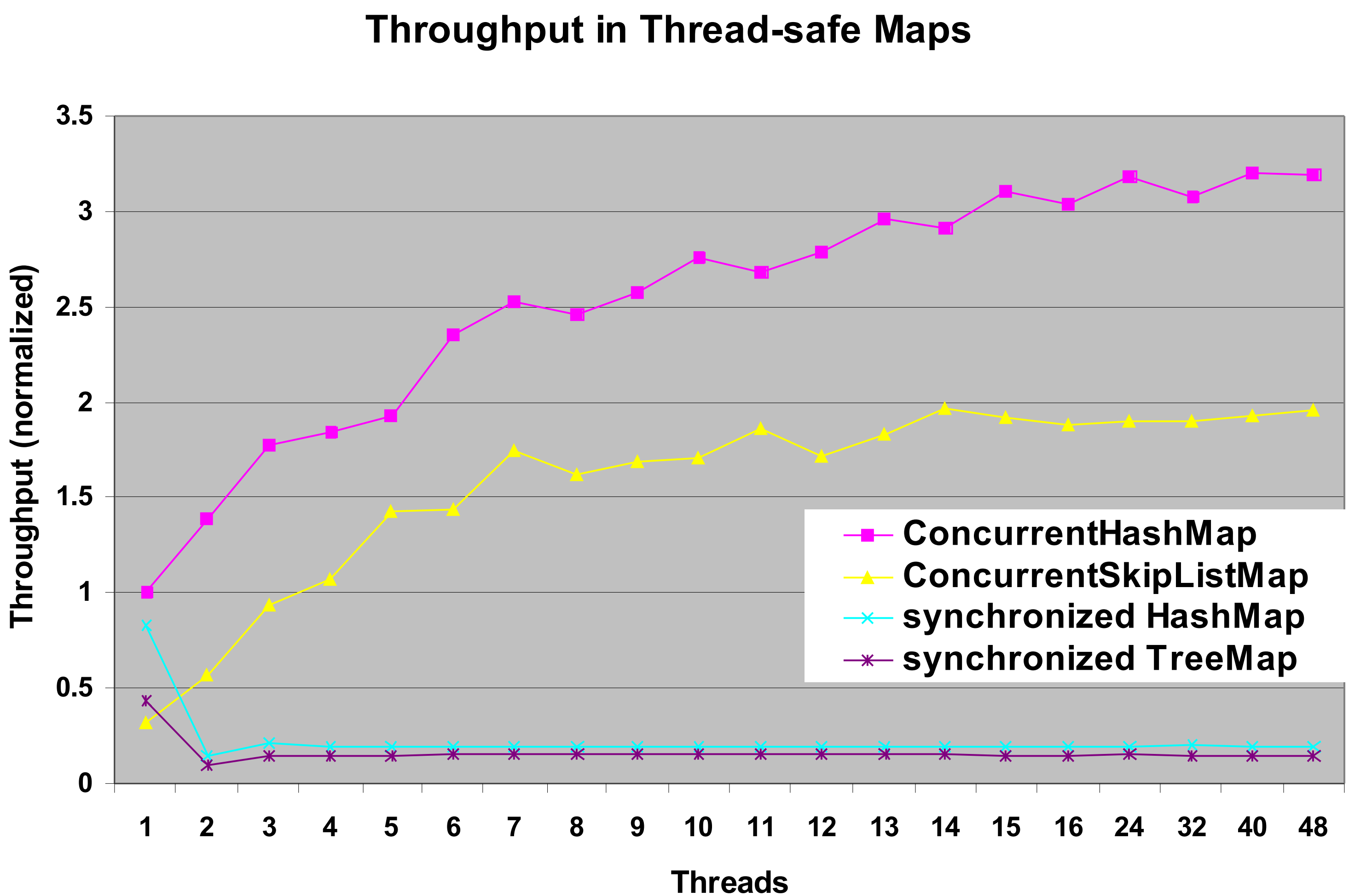
- **`ConcurrentHashMap`**
  - Concurrent (scalable) replacement for **`Hashtable`** or **`Collections.synchronizedMap`**
  - Allows reads to overlap each other
  - Allows reads to overlap writes
  - Allows up to 16 writes to overlap
  - Iterators don't throw **`ConcurrentModificationException`**
- **`CopyOnWriteArrayList`**
  - Optimized for case where iteration is much more frequent than insertion or removal
  - Ideal for event listeners

Concurrent Collections

Iteration Semantics

- Synchronized collection iteration broken by concurrent changes in another thread
  - Throws **`ConcurrentModificationException`**
  - Locking a collection during iteration hurts scalability
- Concurrent collections can be modified concurrently during iteration
  - Without locking the whole collection
  - Without **`ConcurrentModificationException`**
  - But changes may not be seen

Concurrent Collection Performance



Java 6 B77

8-way system

40% read only

60% insert

2% removals

Queues

New Interface added to `java.util`

```
interface Queue<E> extends Collection<E>   {
    boolean offer(E x);
    E poll();
    E remove() throws NoSuchElementException;
    E peek();
    E element() throws NoSuchElementException;
}
```

- Retrofit (non-thread-safe)–implemented by `LinkedList`
- Add (non-thread-safe) `PriorityQueue`
- Fast thread-safe non-blocking ConcurrentLinkedQueue
- Better performance than `LinkedList` is possible as random-access requirement has been removed

Blocking Queues

`BlockingQueue` Interface

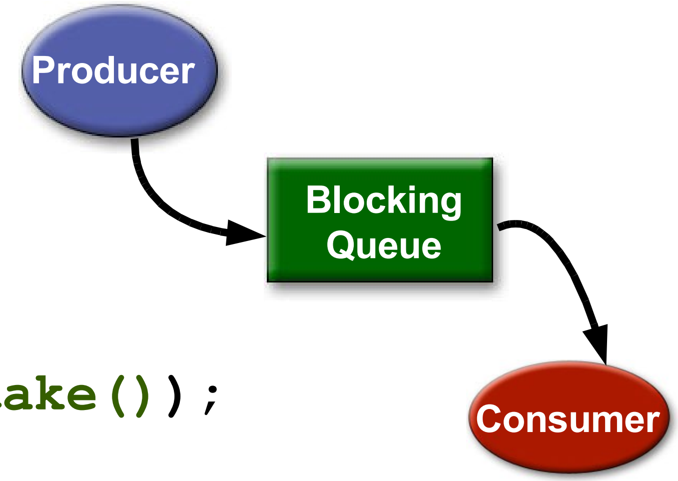
- Extends **`Queue`** to provides blocking operations
  - Retrieval: `take` – wait for queue to become nonempty
  - Insertion: `put` – wait for capacity to become available
- Several implementations:
  - `LinkedBlockingQueue`
    - Ordered FIFO, may be bounded, two-lock algorithm
  - `PriorityBlockingQueue`
    - Unordered but retrieves least element, unbounded, lock-based
  - `ArrayBlockingQueue`
    - Ordered FIFO, bounded, lock-based
  - `SynchronousQueue`
    - Rendezvous channel, lock-based in Java 5, lock-free in Java 6

BlockingQueue Example

```
class LogWriter {
   final BlockingQueue msgQ =
                      new LinkedBlockingQueue();

   public void writeMessage(String msg) throws IE {
      msgQ.put(msg);
   }

   // run in background thread
   public void logServer() {
      try {
         while (true) {
            System.out.println(msgQ.take());
         }
      }
      catch(InterruptedException ie) { ... }
   }
}
```



Producer-Consumer Pattern

- `LogWriter` example illustrates the *producer-consumer* pattern
  - Ubiquitous concurrency pattern, nearly always relies on some form of blocking queue
  - Decouples identification of work from doing the work
    - Simpler and more flexible
- `LogWriter` had many producers, one consumer
  - Thread pool has many producers, many consumers
- `LogWriter` moves IO from caller to log thread
  - Shorter code paths, fewer context switches, no contention for IO locks -> more efficient

Agenda

Overview of `java.util.concurrent`

Concurrent Collections

Threads Pools and Task Scheduling

Locks, Conditions and Synchronizers

Atomic Variables

Executors

Framework for asynchronous execution

- Standardize asynchronous invocation
  - Framework to execute `Runnable` and `Callable` tasks
- Separate submission from execution policy
  - Use `anExecutor.execute(aRunnable)`
  - Not `new Thread(aRunnable).start()`
- Cancellation and shutdown support
- Usually created via `Executors` factory class
  - Configures flexible `ThreadPoolExecutor`
  - Customize shutdown methods, before/after hooks, saturation policies, queuing

Executors

Decouple submission from execution policy

```
public interface Executor {
  void execute(Runnable command);
}
```

- Code which submits a task doesn't have to know in what thread the task will run
  - Could run in the calling thread, in a thread pool, in a single background thread (or even in another JVM!)
  - Executor implementation determines *execution policy*
    - Execution policy controls resource utilization, saturation policy, thread usage, logging, security, etc
    - Calling code need not know the execution policy

Executor and ExecutorService

ExecutorService adds lifecycle management

- ExecutorService supports both graceful and immediate shutdown

```
public interface ExecutorService extends Executor {
    void shutdown();
    List<Runnable> shutdownNow();
    boolean isShutdown();
    boolean isTerminated();
    boolean awaitTermination(long time,TimeUnit unit)
                    throws InterruptedException


    // other convenience methods for submitting tasks
}
```

- Many useful utility methods too

Creating Executors

Factory methods in the Executors class

```
public class Executors {
    static ExecutorService
        newSingleThreadedExecutor();

    static ExecutorService
        newFixedThreadPool(int poolSize);

    static ExecutorService
        newCachedThreadPool();

    static ScheduledExecutorService
        newScheduledThreadPool(int corePoolSize);

    // additional versions specifying ThreadFactory
    // additional utility methods
}
```

Executors Example

Web Server – poor resource management

```
class UnstableWebServer {

  public static void main(String[] args) {
    ServerSocket socket = new ServerSocket(80);
    while (true) {
      final Socket connection = socket.accept();
      Runnable r = new Runnable() {
        public void run() {
          handleRequest(connection);
        }
      };
      // Don't do this!
      new Thread(r).start();
    }
  }
}
```

Executors Example

Web Server – better resource management

```
class BetterWebServer {
  Executor pool = Executors.newFixedThreadPool(7);

  public static void main(String[] args) {
    ServerSocket socket = new ServerSocket(80);
    while (true) {
      final Socket connection = socket.accept();
      Runnable r = new Runnable() {
        public void run() {
          handleRequest(connection);
        }
      };
      pool.execute(r);
    }
  }
}
```

Saturation Policies

- An Executor which execute tasks in a thread pool
  - Can guarantee you will not run out of threads
  - Can manage thread competition for CPU resources
- There is still a risk of running out of memory
  - Tasks could queue up without bound
- Solution: use a *bounded task queue*
  - Just so happens that JUC provides several of these...
- If queue fills up, the *saturation policy* is applied
  - Policies available: throw, discard oldest, discard newest, or run-in-calling-thread
    - The last has the benefit of throttling the load

Futures and Callables

Representing asynchronous tasks

- `Callable` is functional analog of `Runnable`

```
interface Callable<V> {
    V call() throws Exception;
}
```

- `Future` holds result of asynchronous call, normally a `Callable`

```
interface Future<V> {
    V get() throws InterruptedException,
                   ExecutionException;
    V get(long timeout, TimeUnit unit) throws ...;
    boolean cancel(boolean mayInterrupt);
    boolean isCancelled();
    boolean isDone();
}
```

Futures Example

Implementing a concurrent cache

```
public class Cache<K, V> {
  final ConcurrentMap<K, FutureTask<V>> map =
                          new ConcurrentHashMap<K, FutureTask<V>>();

  public V get(final K key) throws InterruptedException {
    FutureTask<V> f = map.get(key);
    if (f == null) {
      Callable<V> c = new Callable<V>() {
        public V call() {
          // return value associated with key
        }
      };
      f = new FutureTask<V>(c);
      FutureTask<V> old = map.putIfAbsent(key, f);
      if (old == null)
        f.run();
      else
        f = old;
    }
    try { return f.get(); }
    catch(ExecutionException ex) { // rethrow ex.getCause() }
  }
}
```

`ScheduledExecutorService`

Deferred and recurring tasks

- `ScheduledExecutorService` can be used to:
  - Schedule a **`Callable`** or **`Runnable`** to run once with a fixed delay after submission
  - Schedule a **`Runnable`** to run periodically at a fixed rate
  - Schedule a **`Runnable`** to run periodically with a fixed delay between executions
- Submission returns a `ScheduledFutureTask` handle which can be used to cancel the task
- Like `java.util.Timer`, but supports pooling

Agenda

Overview of `java.util.concurrent`

Concurrent Collections

Threads Pools and Task Scheduling

Locks, Conditions and Synchronizers

Atomic Variables

Locks

- Use of monitor synchronization is just fine for most applications, but it has some shortcomings
  - Single wait-set per lock
  - No way to interrupt or time-out when waiting for a lock
  - Locking must be block-structured
    - Inconvenient to acquire a variable number of locks at once
    - Advanced techniques, such as hand-over-hand locking, are not possible
- Lock objects address these limitations
  - But harder to use: need `finally` block to ensure release
  - So if you don't need them, stick with `synchronized`

Framework for Flexible Locking

- ```
  interface Lock {
      void       lock();
      void       lockInterruptibly() throws
                                     InterruptedException;
      boolean    tryLock();
      boolean    tryLock(long time,TimeUnit unit) throws
                                     InterruptedException;
      void       unlock();
      Condition newCondition() throws
                            UnsupportedOperationException;
  }
  ```
- High-performance implementation: `ReentrantLock`
  - Basic semantics same as use of **`synchronized`**
  - Condition object semantics like **`wait/notify`**
  ```

Simple `Lock` Example

- Used extensively within `java.util.concurrent`

```
final Lock lock = new ReentrantLock();
...
lock.lock();
try {
  // perform operations protected by lock
}
catch(Exception ex) {
  // restore invariants & rethrow
}
finally {
  lock.unlock();
}
```

- Must manually ensure lock is released

Conditions

Monitor-like operations for working with Locks

- `Condition` is an abstraction of `wait/notify`

```
interface Condition {
   void    await() throws InterruptedException;
   boolean await(long time, TimeUnit unit)
                     throws InterruptedException;
   long    awaitNanos(long nanosTimeout)
                     throws InterruptedException;
   boolean awaitUntil(Date deadline)
                     throws InterruptedException;
   void    awaitUninterruptibly();

   void    signal();
   void    signalAll();
}
```

  - Timed `await` versions report reason for return

Condition Example

```
class BoundedBuffer {
  final Lock lock           = new ReentrantLock();
  final Condition notFull   = lock.newCondition();
  final Condition notEmpty = lock.newCondition();
  ...
  void put(Object x)throws InterruptedException {
    lock.lock(); try {
      while (isFull()) notFull.await();
      doPut(x);
      notEmpty.signal();
    } finally { lock.unlock(); }
  }
  Object take() throws InterruptedException {
    lock.lock(); try {
      while (isEmpty()) notEmpty.await();
      notFull.signal();
      return doTake();
    } finally { lock.unlock(); }
  }
}
```

Synchronizers

Utility classes for coordinating access and control

- `Semaphore` – Dijkstra counting semaphore, managing a specified number of permits
- `CountDownLatch` – allows one or more threads to wait for a set of threads to complete an action
- `CyclicBarrier` – allows a set of threads to wait until they all reach a specified barrier point
- `Exchanger` – allows two threads to rendezvous and exchange data
    - Such as exchanging an empty buffer for a full one

Semaphore Example

Bound the submission of tasks to an executor

```
public class ExecutorProxy implements Executor {
  private final Semaphore tasks;
  private final Executor master;

  ExecutorProxy(Executor master, int limit) {
    this.master = master;
    tasks = new Semaphore(limit);
  }

  public void execute(Runnable r) {
    tasks.acquireUninterruptibly(); // for simplicity
    try {
      master.execute(r);
    }
    finally {
      tasks.release();
    }
  }
}
```

Agenda

Overview of `java.util.concurrent`

Concurrent Collections

Threads Pools and Task Scheduling

Locks, Conditions and Synchronizers

Atomic Variables

Atomic variables

Holder classes for scalars, references and fields

- Support atomic operations
  - Compare-and-set (CAS)
  - Get, set and arithmetic operations (where applicable)
    - Increment, decrement operations
- Abstraction of **`volatile`** variables
- Nine main classes:
  - { int, long, reference } X { value, field, array }
- E.g. **`AtomicInteger`** useful for counters, sequence numbers, statistics gathering

`AtomicInteger` Example

Construction counter for monitoring/management

- Replace this:

```
class Service {
    static int services;
    public Service() {
        synchronized(Service.class) {
            services++;
        }
    } // ...
}
```

- With this:

```
class Service {
  static AtomicInteger services =
      new AtomicInteger();
  public Service() {
    services.getAndIncrement();
  }
  // ...
}
```

Atomic Compare-and-Set (CAS)

- `boolean compareAndSet(int expected, int update)`
  - Atomically sets value to `update` if currently `expected`
  - Returns true on successful update
- Direct hardware support in all modern processors
  - CAS, cmpxchg, ll/sc
- High-performance on multi-processors
  - No locks, so no lock contention and no blocking
  - But can fail
    - So algorithms must implement retry loop
- Foundation of many concurrent algorithms

Sneak Preview of Java 6

- Double-ended queues: `Deque`, `BlockingDeque`
  - Implementations: `ArrayDeque`, `LinkedBlockingDeque`, `ConcurrentLinkedDeque`
- Concurrent skiplists: `ConcurrentSkipList{Map|Set}`
- Enhanced navigation of sorted maps/sets
  - `Navigable{Map|Set}`
- Miscellaneous algorithmic enhancements
  - More use of lock-free algorithms in utilities
  - VM performance improvements for intrinsic locking
- M&M support for locks & conditions

java.util.concurrent

- Executors
  - `Executor`
  - `ExecutorService`
  - `ScheduledExecutorService`
  - `Callable`
  - `Future`
  - `ScheduledFuture`
  - `Delayed`
  - `CompletionService`
  - `ThreadPoolExecutor`
  - `ScheduledThreadPoolExecutor`
  - `AbstractExecutorService`
  - `Executors`
  - `FutureTask`
  - `ExecutorCompletionService`
- Queues
  - `BlockingQueue`
  - `ConcurrentLinkedQueue`
  - `LinkedBlockingQueue`
  - `ArrayBlockingQueue`
  - `SynchronousQueue`
  - `PriorityBlockingQueue`
  - `DelayQueue`
- Concurrent Collections
  - `ConcurrentMap`
  - `ConcurrentHashMap`
  - `CopyOnWriteArray{List,Set}`
- Synchronizers
  - `CountDownLatch`
  - `Semaphore`
  - `Exchanger`
  - `CyclicBarrier`
- Locks: `java.util.concurrent.locks`
  - `Lock`
  - `Condition`
  - `ReadWriteLock`
  - `AbstractQueuedSynchronizer`
  - `LockSupport`
  - `ReentrantLock`
  - `ReentrantReadWriteLock`
- Atomics: `java.util.concurrent.atomic`
  - `Atomic[Type]`
  - `Atomic[Type]Array`
  - `Atomic[Type]FieldUpdater`
  - `Atomic{Markable,Stampable}Reference`

Summary

- Whenever you are about to use
  - `Object.wait`, `notify`, `notifyAll`
  - `new Thread(aRunnable).start();`
  - `synchronized`
- Check first in `java.util.concurrent` if there is a class that ...
  - Does it already, or
  - Let's you do it a simpler, or better way, or
  - Provides a better starting point for your own solution
- Don't reinvent the wheel!

For More Information

- JavaDoc for **`java.util.concurrent`** – in JDK 5.0 download or on Sun website
- Doug Lea's concurrency-interest mailing list
  - http://gee.cs.oswego.edu/dl/concurrency-interest/index.html
- Concurrent Programming in Java (Lea)
  - Addison-Wesley, 1999 ISBN 0-201-31009-0
- Java Concurrency in Practice (Goetz, et al)
  - Addison-Wesley, 2006, ISBN 0-321-34960-1
- JUC Backport to JDK 1.4
  - **`http://www.mathcs.emory.edu/dcl/util/backport-util-concurrent/`**

Q&A

Brian Goetz

David Holmes

Simpler, Faster, Better: Concurrency Utilities in JDK™ Software Version 5.0

Brian Goetz

Principal Consultant, Quiotix Corp

David Holmes

Staff Engineer, Sun Microsystems Inc.

TS-4915