

JavaOne™

Sun's 2004 Worldwide Java Developer Conference™

Concurrency Utilities in JDK 1.5 (Tiger)

Multithreading made simple(r)

David Holmes

Director, DLTeCH Pty Ltd

Brian Goetz

Principal Consultant, Quiotix Corp

java.sun.com/javaone/sf



What This Talk Is About

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

Speakers

- **David Holmes** is a regular speaker on concurrent programming in Java, co-author of *The Java Programming Language 3rd Ed.* and a member of the JSR 166 Expert Group
 - **Brian Goetz** is author of over 50 articles on Java development and a member of the JSR 166 Expert Group
-

Agenda

Rationale and goals for JSR 166

Executors – thread pools and scheduling

Futures

Concurrent Collections

Locks, conditions and synchronizers

Atomic variables

System enhancements

Rationale for JSR 166

Developing concurrent classes is 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 JSR 166

Simplify development of concurrent applications

- Do for concurrency what the Collections framework did for data structures!
- Provide a set of basic concurrency building blocks that can be widely reused
- Enhance scalability, performance, readability, maintainability, and thread-safety of concurrent Java applications

Goals for JSR 166

- The concurrency improvements in Tiger should:
 - Make some problems trivial to solve by everyone
 - Make some problems easier to solve by concurrent programmers
 - Make some problems possible to solve by concurrency experts

Background for JSR 166

- Based mainly on package `EDU.oswego.cs.dl.util.concurrent` from <http://gee.cs.oswego.edu> by Doug Lea
 - APIs refactored based on 4+ years of usage experience
 - APIs enhanced to use generics, enumerations
 - Implementations improved to take advantage of
 - Additional native JVM constructs
 - New Java Memory Model guarantees (JSR 133)

What's new in Tiger for Concurrency?

New classes and enhancements

- Executors, Thread Pools, and Futures
- Concurrent collections: Queues, Blocking Queues, ConcurrentHashMap
- Locks and Conditions
- Synchronizers: Semaphores, Barriers, etc.
- Atomic Variables
 - Low-level *compare-and-set* operation
- Other enhancements
 - Nanosecond-granularity timing

Executors

Framework for asynchronous execution

- Standardize asynchronous invocation
- Separate submission from execution policy
 - use `anExecutor.execute(aRunnable)`
 - not `new Thread(aRunnable).start()`
- Two styles supported:
 - Actions: `Runnable`s
 - Functions (indirectly): `Callable`s
 - Also cancellation and shutdown support
- Usually created via `Executors` factory class
 - Configures flexible `ThreadPoolExecutor`
 - Customize shutdown methods, before/after hooks, saturation policies, queuing

Executor and ExecutorService

ExecutorService adds lifecycle management

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

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

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

Creating Executors

Executor factory methods

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

    static ExecutorService
        newFixedThreadPool(int n);

    static ExecutorService
        newCachedThreadPool(int n);

    static ScheduledExecutorService
        newScheduledThreadPool(int n);

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

Executors example

Web Server – poor resource management

```
class WebServer {  
  
    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 WebServer {
    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);
        }
    }
}
```

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 to a **Callable**

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

Futures example

Implementing a cache with Future

```
public class Cache<K, V> {
    Map<K, Future<V>> map = new ConcurrentHashMap();
    Executor executor = Executors.newFixedThreadPool(8);

    public V get(final K key) {
        Future<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);
            Future old = map.putIfAbsent(key, f);
            if (old == null)
                executor.execute(f);
            else
                f = old;
        }
        return f.get();
    }
}
```

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 **Timer**, but supports pooling

Concurrent Collections

Concurrent vs Synchronized

- Pre-1.5 Java class libraries included many *thread-safe*, but few truly *concurrent*, classes
- Synchronized collections:
 - Eg. `Hashtable`, `Vector`, and `Collections.synchronizedMap`
 - Often require locking during iteration
 - Monitor is a source of contention under concurrent access
- Concurrent collections:
 - Allow multiple operations to overlap each other
 - At the cost of some slight differences in semantics
 - Might not support atomic operations

Queues

- **Queue** 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**

Blocking Queues

- Extends `Queue` to provide blocking operations
 - Retrieval: wait for queue to become nonempty
 - Insertion: wait for capacity to be available
- Common in producer-consumer designs
- Can support multiple producers and consumers
- Can be bounded or unbounded
- Implementations provided:
 - `LinkedBlockingQueue` (FIFO, may be bounded)
 - `PriorityBlockingQueue` (priority, unbounded)
 - `ArrayBlockingQueue` (FIFO, bounded)
 - `SynchronousQueue` (rendezvous channel)

Blocking Queue Example

```
class LogWriter {  
    private 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

Blocking
Queue

New Concurrent Collections

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

Locks and Lock Support

Broad support: ready-built or do-it-yourself

- High-level interface

```
interface Lock {
    void lock();
    void lockInterruptibly() throws IE;
    boolean tryLock();
    boolean tryLock(long time,
                    TimeUnit unit) throws IE;
    void unlock();
    Condition newCondition() throws
        UnsupportedOperationException;
}
```

- Sophisticated base class for customized locks
 - `AbstractQueuedSynchronizer`
- Low-level facilities to build specialised locks
 - `LockSupport`: `park()`, `unpark(Thread t)`

Reentrant Lock

Flexible, high-performance lock implementation

- The `ReentrantLock` class implements a reentrant mutual exclusion lock with the same semantics as built-in monitor locks (`synchronized`), but with extra features
 - Can interrupt a thread waiting to acquire a lock
 - Can specify a timeout while waiting for a lock
 - Can poll for lock availability
 - Supports multiple wait-sets per lock via the `Condition` interface
- Outperforms built-in monitor locks in most cases, but slightly less convenient to use (requires `finally` block to release lock)

Locks Example

- Used extensively throughout `java.util.concurrent`
- Must use `finally` block to release lock

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

Read/write Locks

Allow concurrent readers or an exclusive writer

- **ReadWriteLock** defines a pair of locks

```
interface ReadWriteLock {  
    Lock readLock();  
    Lock writeLock();  
}
```

- Various implementation policies are possible
- The **ReentrantReadWriteLock** class:
 - Provides reentrant read and write locks
 - Allows writer to acquire read lock
 - Allows writer to downgrade to read lock
 - Supports “fair” and “writer preference” acquisition

Read/write Lock Example

```
class RWDictionary {
    private final Map<String, Data> m =
        new TreeMap<String, Data>();
    private final ReentrantReadWriteLock rwl =
        new ReentrantReadWriteLock();
    private final Lock r = rwl.readLock();
    private final Lock w = rwl.writeLock();

    public Data get(String key) {
        r.lock(); try { return m.get(key); }
        finally { r.unlock(); }
    }
    public Data put(String key, Data value) {
        w.lock(); try { return m.put(key, value); }
        finally { w.unlock(); }
    }
    public void clear() {
        w.lock(); try { m.clear(); }
        finally { w.unlock(); }
    }
}
```

Conditions

Monitor-like operations for working with Locks

- **Condition** lets you wait for a condition to hold

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

- Improvements over `wait()`/`notify()`
 - Multiple conditions per lock
 - Absolute and relative time-outs
 - Timed waits tell you why you returned
 - Convenient uninterruptible wait

Condition Example

```
class BoundedBuffer {
    Lock lock = new ReentrantLock();
    Condition notFull = lock.newCondition();
    Condition notEmpty = lock.newCondition();
    Object[] items = new Object[100];
    int putptr, takeptr, count;
    public void put(Object x) throws IE {
        lock.lock(); try {
            while (count == items.length) notFull.await();
            items[putptr] = x;
            if (++putptr == items.length) putptr = 0;
            ++count;
            notEmpty.signal();
        } finally { lock.unlock(); }
    }
    public Object take() throws IE {
        lock.lock(); try {
            while (count == 0) notEmpty.await();
            Object x = items[takeptr];
            if (++takeptr == items.length) takeptr = 0;
            --count;
            notFull.signal();
            return x;
        } finally { lock.unlock(); }
    }
}
```

Synchronizers

Utilities 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

Cyclic Barrier Example

```
class Solver { // Code sketch
    void solve(final Problem p, int nThreads) {
        final CyclicBarrier barrier =
            new CyclicBarrier(nThreads,
                new Runnable() {
                    public void run() { p.checkConvergence(); }}
            );
        for (int i = 0; i < nThreads; ++i) {
            final int id = i;
            Runnable worker = new Runnable() {
                final Segment segment = p.createSegment(id);
                public void run() {
                    try {
                        while (!p.converged()) {
                            segment.update();
                            barrier.await();
                        }
                    }
                    catch (Exception e) { return; }
                }
            };
            new Thread(worker).start();
        }
    }
}
```

Atomic Variables

Holder classes for scalars, references and fields

- Support atomic operations
 - Compare-and-set (CAS)
 - Get and set and arithmetic (where applicable)
- Ten main classes:
 - { *int, long, reference* } X { *value, field, array* }
 - E.g. **AtomicInteger** useful for counters, sequence numbers, statistics gathering
- Essential for writing efficient code on MPs
 - Nonblocking data structures & optimistic algorithms
 - Reduce overhead/contention updating “hot” fields
- JVM uses best construct available on platform
 - CAS, load-linked/store-conditional, locks

System Enhancements

- Nanosecond-granularity timer support via `System.nanoTime()`, but can only be used for measuring *relative* time
 - Nanosecond accuracy is not guaranteed – a JVM should provide the most accurate measurement available

Don't reinvent the wheel!

- Whenever you are about to use...
`Object.wait, notify, notifyAll,`
`synchronized,`
`new Thread(aRunnable).start();`
- Check first if there is a class in `java.util.concurrent` that...
 - Does it already, or
 - Would be a simpler starting point for your own solution

For More Information

- JavaDoc for `java.util.concurrent` – in Tiger 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”
 - Addison Wesley, 1999 ISBN 0-201-31009-0
- TS 2136 – Concurrency Utilities in Practice

Q&A

Concurrency Utilities in Tiger



JavaOne™

Sun's 2004 Worldwide Java Developer Conference™

Concurrency Utilities in JDK 1.5 (Tiger)

Multithreading made simple(r)

David Holmes

Director, DLTeCH Pty Ltd

Brian Goetz

Principal Consultant, Quiotix Corp

java.sun.com/javaone/sf

