JSR-166: Concurrency Utilities

- The java.util.concurrent package aims to do for concurrency what java.util.Collections did for data structures. It makes
  - Some problems go away
  - Some problems trivial to solve by everyone
  - Some problems easier to solve by concurrent programmers
  - Some problems possible to solve by experts

Whenever you are about to use...
Object.wait, notify, notifyAll,
synchronized,
new Thread();
Check first if there is a class that ...
  - automates what you are trying to do, or
  - would be a simpler starting point for your own solution

JSR-166 Components

- Executors, Thread pools, and Futures
- Queues and blocking queues
- Timing
- Locks and Conditions
- Synchronizers: Semaphores, Barriers, etc
- Atomic variables
- Miscellany

Present and Future

- JSR-166 is based on over 3 years experience with
  EDU.oswego.cs.dl.util.concurrent
- Many refactorings and functionality improvements
- Additional native/JVM support
  - Timing, atomics, built-in monitor extensions
- A preliminary release of JSR-166 APIs, implementations, and JVM enhancements will be available soon

Main Interfaces

<table>
<thead>
<tr>
<th>Lock</th>
<th>Condition</th>
<th>Collection</th>
</tr>
</thead>
<tbody>
<tr>
<td>void lock()</td>
<td>void await()</td>
<td>boolean add(Object x)</td>
</tr>
</tbody>
</table>
| void unlock() | void signal()      | Object poll()...
| boolean trylock() | newCondition()    | |
| ReentrantLock | LinkedQ            | BlockingQueue   |
|               | Executor           | ArrayBQ         |
|               | Executor           | LinkedBQ        |
|               | ThreadExecutor     |                 |
|               | execute(Runnable r)|                 |
Executors

- Standardize asynchronous invocation
  - use `anExecutor.execute(aRunnable)`
  - not `new Thread(aRunnable).start()`
- Two styles supported:
  - Actions: Runnables
  - Functions (indirectly): Callables
  - Also cancellation and shutdown support
- Most access via `Executors` utility class
  - Configures very flexible `ThreadExecutor`
  - Also `ScheduledExecutor` for time-delayed tasks

Thread Pools in Service Designs

Thread Pools

- `ThreadExecutors` can vary in:
  - The kind of task queue
  - Maximum and minimum number of threads
  - Shutdown policy
    - Immediate, wait for current tasks
    - Keep-alive interval until idle threads die
    - To be later replaced by new ones if necessary
    - Before/after methods around tasks
  - Factory methods package some common settings
    - `newSingleThreadExecutor()`
    - `newFixedThreadPool(int nthreads)`
    - `newCachedThreadPool()`
    - Reuses threads when available, else constructs

Thread Pool Example

```java
class WebService {
    public static void main(String[] args) {
        Executor pool = Executors.newFixedThreadPool(7);
        ServerSocket socket = new ServerSocket(999);
        for (;;) {
            final Socket connection = socket.accept();
            pool.execute(new Runnable() {
                public void run() {
                    new Handler().process(connection);
                }
            });
        }
    }
}
class Handler { void process(Socket s); }
```
Futures and Callables

- **Callable** is functional analog of **Runnable**
  ```
  interface Callable<V> {
    V call() throws Exception;
  }
  ```
  - Normally implement with an inner class that supplies arguments to the function
- **Future** holds result of asynchronous call, normally to a **Callable**
  ```
  interface Future<V> {
    V get() throws InterruptedException, ExecutionException;
    // plus timeout versions and misc
  }
  ```

Futures Example

```java
class ImageRenderer { Image render(byte[] raw); }

class App { // ...
    Executor executor = ...; // any executor
    ImageRenderer renderer = new ImageRenderer();

    public void display(final byte[] rawimage) {
        try {
            Future<Image> image = Executors.invoke(executor, new Callable<Image>() {
                public Image call() {
                    return renderer.render(rawImage);
                }
            });
            draw Borders(); // do other things ...
            drawCaption(); // ... while executing
            drawImage(image.get()); // use future
        } catch (Exception ex) {
            cleanup();
            return;
        }
    }
}
```

Queues

- **Queue** interface added to **java.util**
  ```
  interface Queue<E> extends Collection<E> {
    boolean add(E x);
    E poll();
    E remove() throws NoSuchElementException;
    E peek();
    E element() throws NoSuchElementException;
  }
  ```
  - Retrofit (non-thread-safe) **java.util.LinkedList** to implement
  - Add (non-thread-safe) **java.util.PriorityQueue**
  - Fast thread-safe non-blocking **java.util.concurrent.LinkedBlockingQueue**

Blocking Queues

```java
interface BlockingQueue<E> extends Queue<E> {
    void put(E x) throws InterruptedException;
    boolean offer(E x, long timeout, TimeUnit unit)
    throws InterruptedException;

    E take() throws InterruptedException;
    E poll(long timeout, TimeUnit unit)
    throws InterruptedException;
}
```
**Blocking Queue Example**

```java
class LoggedService { // ...
    final BlockingQueue<String> msgQ =
        new LinkedBlockingQueue<String>();
    public void serve() throws InterruptedException {
        // ... perform service ...
        String status = ... ;
        msgQ.put(status);
    }
    public LoggedService() { // start background thread
        Runnable logr = new Runnable() {
            public void run() {
                try {
                    for(;;)
                        System.out.println(msgQ.take());
                } catch(InterruptedException ie) {}}
        }
        Executors.newSingleThreadExecutor().execute(logr);
    }
}
```

**TimeUnits**

- Standardize time usage across APIs, without forcing use of inappropriate units
  - SECONDS, MILLISECONDS, MICROSECONDS, NANOSECONDS
  - `x = queue.poll(3, TimeUnit.SECONDS)`
- TimeUnit class also supplies conversions and other time-based utilities
- Provides high resolution timing support
  - static long nanoTime()
  - Value is unrelated to `java.util.Date`, `System.currentTimeMillis` etc

**Locks**

- Flexibility at expense of verbosity
  - `lock.lock();
   try {
      action();
   } finally {
      lock.unlock();
   }`
- Overcomes limitations of `synchronized`
  - Doesn’t force block structured locking/unlocking
  - Allow interruptible lock acquisition and “try lock”
  - Can define customized implementations

**Lock API**

```java
interface Lock {
    void lock();
    void lockInterruptibly() throws IE;
    boolean trylock();
    boolean trylock(long timeout,
                    TimeUnit unit) throws IE;
    void unlock();
    Condition newCondition();
}
```

- Concrete `ReentrantLock` implementation
  - Fast, scalable with synchronized block semantics, and additional query methods
  - Also `FairReentrantLock` subclass with slower but more predictable first-in-first-out arbitration
class ParticleUsingLock {
    private int x, y;
    private final Random rng = new Random();
    private final Lock lock = new ReentrantLock();

    public void move() throws InterruptedException {
        lock.lockInterruptibly(); // allow cancellation
        try {
            x += rng.nextInt(3) - 1;
            y += rng.nextInt(3) - 1;
        } finally {
            lock.unlock();
        }
    }

    public void draw(Graphics g) {
        lock.lock(); // no interrupts – AWT Event Thread
        try {
            int lx, ly;
            lx = x; ly = y;
        } finally {
            lock.unlock();
        }
        g.drawRect(lx, ly, 10, 10);
    }
}

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

A pair of locks for enforcing multiple-reader, single-writer access
- Each used in the same way as ordinary locks
- Concrete ReentrantReadWriteLock
- Almost always the best choice for apps
- Each lock acts like a reentrant lock
- Write lock can “downgrade” to read lock (not vice-versa)

interface Condition {
    void await() throws IE;
    void awaitUninterruptibly();
    long awaitNanos(long nanos) throws IE;
    boolean awaitUntil(Date deadline) throws IE;
    void signal();
    void signalAll();
}

Allows more than one wait condition per object
- Even for built-in locks, via Locks utility class
- Allows much simpler implementation of some classic concurrent designs

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;
            notFull.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;
            notEmpty.signal();
            return x;
        } finally {
            lock.unlock();
        }
    }
}
Synchronizers

A small collection of small classes that:
- Provide good solutions to common special-purpose synchronization problems
- Provide better ways of thinking about designs
  - But worse ways when they don't naturally apply!
- Can be tricky or tedious to write yourself

Semaphore, FairSemaphore
CountDownLatch
CyclicBarrier
Exchanger

Semaphores

- Semaphores can be seen as permit holders
  - Create with initial number of permits
  - acquire takes a permit, waiting if necessary
  - release adds a permit
  - But no actual permits change hands.
    - Semaphore just maintains the current count.
- Can use for both “locking” and “synchronizing”
  - With initial permits=1, can serve as a lock
  - Useful in buffers, resource controllers
  - Use in designs prone to missed signals
    - Semaphores “remember” past signals

Semaphores in Resource Pools

class ResourcePool {
    FairSemaphore available =
        new FairSemaphore(N);
    Object[] items = ... ;

    public Object getItem() throws IE {
        available.acquire();
        return nextAvailable();
    }

    public void returnItem(Object x) {
        if (unmark(x))
            available.release();
        synchronized Object nextAvailable();
        synchronized boolean unmark(Object x);
    }
}

CountDownLatch Example

class Driver { // ...
    void main(int N) throws InterruptedException {
        final CountDownLatch startSignal = new CountDownLatch(1);
        final CountDownLatch doneSignal = new CountDownLatch(N);
        for (int i = 0; i < N; ++i) // Make threads
            new Thread() {
                public void run() {
                    try {
                        startSignal.wait();
                        doWork();
                        doneSignal.countDown();
                    } catch (InterruptedException ie) {}}}.start();
        initialize();
        startSignal.countDown(); // Let all threads proceed
        doSomethingElse(); // Wait for all to complete
        cleanup();
    }
}
CyclicBarrier 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();
        }
    }
}

Exchanger Example

class FillAndEmpty {
    Exchanger ex = new Exchanger();
    Buffer initialEmptyBuffer = ... // a made-up type
    Buffer initialFullBuffer = ...;
    class FillingLoop implements Runnable {
        public void run() {
            Buffer b = initialEmptyBuffer;
            try {
                while (b != null) {
                    addToBuffer(b);
                    if (b.full()) b = (Buffer)(ex.exchange(b));
                } catch(...) ... }
    }
    class EmptyingLoop implements Runnable {
        public void run() {
            Buffer b = initialFullBuffer;
            try {
                while (b != null) {
                    takeFromBuffer(b);
                    if (b.empty()) b = (Buffer)(ex.exchange(b));
                } catch(...) ... }
    }
}

Atomics

- j.u.c.atomic contains classes representing scalars supporting "CAS"
- boolean compareAndSet(expectedV, newV)
  - Atomically set to newV if holding expectedV
  - Always used in a loop
- Essential for writing efficient code on MPs
  - Nonblocking data structures, optimistic algorithms, reducing overhead and contention when updates center on a single field
- JVMs use best construct available on platform
  - Compare-and-swap, Load-linked/Store-conditional, Locks
- j.u.c.a also supplies reflection-based classes that allow CAS on given volatile fields of other classes

Atomic Variable Example

class Random { // snippets
    private AtomicLong seed;
    Random(long s) {
        seed = new AtomicLong(s);
    }
    long next() {
        for (;;) {
            long s = seed.get();
            long nexts = s * ... + ...;
            if (seed.compareAndSet(s, nexts))
                return s;
        }
    }
}
- Faster and less contention in programs with a single Random accessed by many threads
Optimistic Linked Lists

```java
class OptimisticLinkedList {   // incomplete
    static class Node {
        volatile Object item;
        final AtomicReference<Node> next;
        Node(Object x, Node n) {
            item = x; next = new AtomicReference(n);
        }
    }
    final AtomicReference head = new AtomicReference(null);
    public void prepend(Object x) {
        if (x == null) throw new IllegalArgumentException();
        for(;;) {
            Node h = head.get();
            if (head.compareAndSet(h, new Node(x, h)) ) return;
        }
    }
    public boolean search(Object x) {
        Node p = head.get();
        while (p != null && x != null && !p.item.equals(x))
            p = p.next.get();
        return p != null && x != null;
    }
    // remove(x) is much harder!
}
```

Other JSR-166 Features

- Customizable per-Thread UncaughtExceptionHandler
- Concurrent Collection implementations
  - ConcurrentHashMap, CopyOnWriteArrayList
- Improvements to existing thread-safe collections in part based on JSR-133 Memory Model rules
- ThreadLocal.remove
  - Helps avoid resource exhaustion

JSR-133: Fixing the Memory Model

- A memory model specifies how threads and objects interact
  - Atomicity
    - Locking to obtain mutual exclusion for field updates
  - Visibility
    - Ensuring that changes made in one thread are seen in other threads
  - Ordering
    - Ensuring that you aren’t surprised by the order in which statements are executed
- Original JLS spec was broken and impossible to understand
- Included unwanted constraints on compilers and JVMs, omissions, inconsistencies
- JSR-133 still officially “in progress” but Sun JDKs conform to main rules as of 1.4.0
- The basic rules are easy. The more formal spec is not.
Additional JSR-133 Rules

- Variants of lock rule apply to volatile fields and thread control
  - Writing a volatile has same basic memory effects as unlock
  - Reading a volatile has same basic memory effects as lock
  - Similarly for thread start and termination
  - Details differ from locks in minor ways

- Final fields
  - All threads will read the final value so long as it is guaranteed to be assigned before the object could be made visible to other threads. So DON'T write:
    ```java
class Stupid implements Runnable {
    final int id;
    Stupid(int i) { new Thread(this).start(); id = i; }
    public void run() { System.out.println(id); }
}
```
- Extremely weak rules for unsynchronized, non-volatile, non-final reads and writes
  - type-safe, not-out-of-thin-air, but can be reordered, invisible