Overview of the \texttt{util.concurrent} package

Doug Lea
State University of New York at Oswego
dl@cs.oswego.edu
http://gee.cs.oswego.edu
Outline

• Goals and structure
• Principal interfaces and implementations
  Sync: acquire/release protocols
  Channel: put/take protocols
  Executor: executing Runnable tasks
    – Each has a few other associated interfaces and support classes
• Brief mentions of other classes and features
Goals

• A few simple interfaces
  – But cover most problems for which programmers would otherwise need to write tricky code

• High-quality implementations
  – correct, conservative, efficient, portable

• Possible basis for future standardization
  – Gain experience and collect feedback
Structure

Sync
- void acquire()
- void release()
- boolean attempt(long ms)

Semaphore ... Mutex

Executor
- void execute(Runnable r)

QueuedExecutor ... PooledExecutor

Channel
- void put(Object x)
- Object take();
- void offer(long ms, Object x)
- Object poll(long ms)

BoundedBuffer ... LinkedQueue

classic interface + implementation design, with some opportunistic subclassing
Sync

- Main interface for acquire/release protocols
  - Used for custom locks, resource management, other common synchronization idioms
  - Coarse-grained interface
    - Doesn’t distinguish different lock semantics
- Implementations
  - Mutex, ReentrantLock, Latch, CountDown, Semaphore, WaiterPreferenceSemaphore, FIFO Semaphore, PrioritySemaphore
  - Also, utility implementations such as ObservableSync, LayeredSync that simplify composition and instrumentation
Exclusion Locks

```java
try {
    lock.acquire();
    try {
        action();
    } finally {
        lock.release();
    }
} catch (InterruptedException ie) { ... }
```

- Use when synchronized blocks don’t apply
  - Time-outs, back-offs
  - Assuring interruptibility
  - Hand-over-hand locking
  - Building Posix-style condvars
Exclusion Example

class ParticleUsingMutex {
    int x;  int y;
    final Random rng = new Random();
    final Mutex mutex = new Mutex();

    public void move() {
        try {
            mutex.acquire();
            try {  x += rng.nextInt(2)−1; y += rng.nextInt(2)−1; }
            finally { mutex.release(); } } 
        } catch (InterruptedException ie) { 
            Thread.currentThread().interrupt(); }
    }

    public void draw(Graphics g) {
        int lx, ly;
        try { 
            mutex.acquire();
            try { lx = x; ly = y; }
            finally { mutex.release(); } }
        } catch (InterruptedException ie) { 
            Thread.currentThread().interrupt(); return; }
        g.drawRect(lx, ly, 10, 10);
    }
}
class CellUsingBackoff {
    private long val;
    private final Mutex mutex = new Mutex();

    void swapVal(CellUsingBackoff other)
        throws InterruptedException {
        if (this == other) return; // alias check
        for (;;) {
            mutex.acquire();
            try {
                if (other.mutex.attempt(0)) {
                    try {
                        long t = val;
                        val = other.val;
                        other.val = t;
                        return;
                    } finally { other.mutex.release(); }  
                }
                finally { mutex.release(); }  
            try {
                other.mutex.acquire();
                try {
                    if (other.mutex.attempt(0)) {
                        try {
                            long t = val;
                            val = other.val;
                            other.val = t;
                            return;
                        } finally { other.mutex.release(); }  
                    }
                    finally { mutex.release(); }  
                }
            } finally { mutex.release(); }  
            Thread.sleep(100); // heuristic retry interval
        }
    }
}
interface ReadWriteLock {
    Sync readLock();
    Sync writeLock();
}

• Manage a pair of locks
  – Used via same idioms as ordinary locks

• Can be useful for Collection classes
  – Semi–automated via SyncSet, SyncMap, ...

• Implementation classes vary in lock policy
ReadWriteLock Example

- Sample wrapper class that can perform any Runnable inside a given read or write lock

class WithRWLock {
    final ReadWriteLock rw;
    public WithRWLock(ReadWriteLock l) { rw = l; }

    public void performRead(Runnable readCommand)
    throws InterruptedException {
        rw.readLock().acquire();
        try {
            readCommand.run();
        } finally {
            rw.readlock().release();
        }
    }

    public void performWrite(...) // similar
}

http://gee.cs.oswego.edu
Latch

- A Latch is a condition starting out false, but once set true, remains true forever
  - Initialization flags
  - End-of-stream conditions
  - Thread termination
  - Event occurrence indicators
- A CountDown is similar but fires after a pre-set number of releases, not just one.
- Very simple but widely used classes
  - Replace error-prone constructions
class Worker implements Runnable {
    Latch startSignal;
    Worker(Latch l) { startSignal = l; }
    public void run() {
        startSignal.acquire();
        // ... doWork();
    }
}

class Driver { // ...
    void main() {
        Latch ss = new Latch();
        for (int i = 0; i < N; ++i) // make threads
            new Thread(new Worker(ss)).start();

        doSomethingElse(); // don't let run yet

        ss.release();       // now let all threads proceed
    }
}
Semaphores

• Conceptually serve as permit holders
  – Construct with initial number of permits (usually 0)
  – acquire waits if needed for a permit, then takes one
  – release adds a permit

• But no actual permits change hands.
  – Semaphore just maintains the current count.

• Applications
  – Locks: A semaphore can be used as Mutex
  – Isolating wait sets in buffers, resource controllers
  – Designs prone to missed signals
    • Semaphores ‘remember’ past signals
class Pool {
    ArrayList items = new ArrayList();
    HashSet busy = new HashSet();
    final Semaphore available;

    public Pool(int n) {
        available = new Semaphore(n);
        // ... somehow initialize n items ...;
    }
    public Object getItem() throws InterruptedException {
        available.acquire();
        return doGet();
    }
    public void returnItem(Object x) {
        if (doReturn(x)) available.release();
    }

    synchronized Object doGet() {
        Object x = items.remove(items.size()-1);
        busy.add(x); // put in set to check returns
        return x;
    }
    synchronized boolean doReturn(Object x) {
        return busy.remove(x); // true if was present
    }
}
Barrier

• Interface for multiparty synchronization
  – Each party must wait for all others to hit barrier

• CyclicBarrier class
  – A resettable version of CountDown
  – Useful in iterative partitioning algorithms

• Rendezvous class
  – A barrier at which each party may exchange information with others
  – Behaves as simultaneous put and take of a synchronous channel
  – Useful in resource-exchange protocols
Channel

- Main interface for buffers, queues, etc.

  - producer
    - put, offer

  - channel

  - consumer
    - take, poll

- Implementations
  - LinkedQueue, BoundedLinkedQueue, BoundedBuffer, BoundedPriorityQueue, SynchronousChannel, Slot
Channel Properties

• Defined as subinterface of Puttable and Takable
  – Allows type enforcement of producer vs consumer roles

• Support time-out methods offer and poll
  – Pure balking when timeout is zero
  – All methods can throw InterruptedException

• No interface requires a size method
  – But some implementations define them
  – BoundedChannel has capacity method
class Service { // ...
    final Channel msgQ = new LinkedQueue();

    public void serve() throws InterruptedException {
        String status = doService();
        msgQ.put(status);
    }

    public Service() { // start background thread
        Runnable logger = new Runnable() {
            public void run() {
                try {
                    for(;;)
                        System.out.println(msgQ.take());
                }
                catch(InterruptedException ie) {} }};
        new Thread(logger).start();
    }
}
Executor

- Main interface for Thread–like classes
  - Pools
  - Lightweight execution frameworks
  - Custom scheduling
- Need only support `execute(Runnable r)`
  - Analogous to `Thread.start`
- Implementations
  - PooledExecutor, ThreadedExecutor, QueuedExecutor, FJTaskRunnerGroup
  - Related ThreadFactory class allows most Executors to use threads with custom attributes
PooledExecutor

• A tunable worker thread pool, with controls for:
  – The kind of task queue (any Channel)
  – Maximum number of threads
  – Minimum number of threads
  – "Warm" versus on-demand threads
  – Keep-alive interval until idle threads die
    • to be later replaced by new ones if necessary
  – Saturation policy
    • block, drop, producer-runs, etc
PooledExecutor Example

class WebService {
    public static void main(String[] args) {
        PooledExecutor pool =
            new PooledExecutor(new BoundedBuffer(10), 20);
        pool.createThreads(4);
        try {
            ServerSocket socket = new ServerSocket(9999);
            for (;;) {
                final Socket connection = socket.accept();
                pool.execute(new Runnable() {
                    public void run() {
                        new Handler().process(connection);
                    }
                });
            }
        } catch (Exception e) { } // die
    }
}

class Handler { void process(Socket s); }
Futures and Callables

- Callable is the argument and result carrying analog of Runnable

```java
interface Callable {
    Object call(Object arg) throws Exception;
}
```

- FutureResult manages asynchronous execution of a Callable

```java
class FutureResult {
    // ...
    // block caller until result is ready
    public Object get()
        throws InterruptedException, InvocationTargetException;

    public void set(Object result); // unblocks get

    // create Runnable that can be used with an Executor
    public Runnable setter(Callable function);
}
```
FutureResult Example

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

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

    public void display(byte[] rawimage) {
        try {
            FutureResult futureImage = new FutureResult();
            Runnable cmd = futureImage.setter(new Callable(){
                public Object call() {
                    return renderer.render(rawImage);
                }
            });
            executor.execute(cmd);

            drawBorders(); // do other things while executing
            drawCaption();

            drawImage((Image)(futureImage.get())); // use future
        }
        catch (Exception ex) {
            cleanup();
            return;
        }
    }
}
Other classes

• CopyOnWriteArrayList
  – Supports lock−free traversal at expense of copying entire collection on each modification
  – Well−suited for most multicast applications
    • Package also includes COW versions of java.beans multicast−based classes

• SynchronizedDouble, SynchronizedInt, SynchronizedRef, etc
  – Analogs of java.lang.Double, etc that define atomic versions of mutative operators
    • for example, addTo, inc,
  – Plus utilities such as swap, commit
Future Plans

- Concurrent Data Structures
  - Collections useful under heavy thread contention
- Support for IO-intensive programs
  - Event-based IO
- Niche implementations
  - For example, SingleSourceQueue
- Minor incremental improvements
  - Making Executors easier to compose
- End-of-lifetime
  - JDK1.3 java.util.Timer obsoletes ClockDaemon