



Effective Concurrency for the Java Platform

Simon Ritter

Technology Evangelist

Sun Microsystems Inc.



The Big Picture

Writing correct concurrent code is difficult, but not impossible.

Using good object-oriented design techniques can make it easier.

Motivation for Concurrency Utilities

- Developing concurrent classes was too hard
- Java has concurrency primitives:
 - > `wait()`, `notify()`, `sleep()`, `interrupt()`, `synchronized`
 - > These are too primitive
 - > Hard to use correctly
 - > Easy to use incorrectly
 - > Too low level for most applications
 - > Incorrect use can produce poor performance
 - > Developers keep re-inventing the wheel

Concurrency Utilities Goals

- Provide a good set of concurrency building blocks
- Do for concurrency what Collections did for data structures
- Beat C performance in high-end server applications
- Enhance scalability, performance, readability and thread safety of Java applications

Concurrency Utilities: JSR-166

- **Task Scheduling Framework: Executor** interface replaces direct use of **Thread**
- **Callable** and **Future**
- **Synchronisers**
 - > Semaphore, CyclicBarrier, CountdownLatch
- **Concurrent collections: BlockingQueue**
- **Lock**
- **Atomic**

Executor

Framework for asynchronous execution

- Standard asynchronous invocation
 - > **Runnable/Callable** tasks
- Separate task submission from execution policy
- No more direct **Thread** invocation
 - > Use **myExecutor.execute(aRunnable);**
 - > Not **new Thread(aRunnable).start();**

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

Thread Pool Example

```
class WebService {
    public static void main(String[] args) {
        Executor pool = Executors.newFixedThreadPool(5);
        ServerSocket socket = new ServerSocket(999);

        for (;;) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() {
                    new Handler().process(connection);
                }
            };
            pool.execute(task);
        }
    }
}

class Handler { void process(Socket s); }
```

ExecutorService for Lifecycle Support

- **ExecutorService** supports graceful and immediate shutdown

```
public interface ExecutorService extends
Executor {
    void shutdown();
    List<Runnable> shutdownNow();
    boolean isShutdown();
    boolean isTerminated();
    boolean awaitTermination(long timeout,
                               TimeUnit unit);
    // additional methods not listed
}
```

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() ;  
    // Other methods not listed  
}
```

ScheduledExecutorService

- Deferred and recurring tasks
 - > Schedule execution of **Callable** or **Runnable** to run once after a fixed delay
 - > **schedule ()**
 - > Schedule a **Runnable** to run periodically at a fixed rate
 - > **scheduleAtFixedRate ()**
 - > Schedule a **Runnable** to run periodically with a fixed delay between executions
 - > **scheduleWithFixedDelay ()**
- Submission returns a **ScheduledFuture**
 - > Can be used to cancel task

ScheduledExecutorService Example

```
ScheduledExecutorService sched =  
    Executors.newSingleThreadScheduledExecutor();  
  
public void runTwiceAnHour(long howLong) {  
    final Runnable rTask = new Runnable() {  
        public void run() { /* Work to do */ } };  
  
    final ScheduledFuture<?> rTaskFuture =  
        sched.scheduleAtFixedRate(rTask, 0, 1800, SECONDS);  
  
    sched.schedule(new Runnable {  
        public void run { rTaskFuture.cancel(true); } },  
        howLong, SECONDS);  
}
```

Callables and Futures

- **Callable** interface provides way to get a result or **Exception** from a separate thread
 - > Implement **call ()** method rather than **run ()**
- **Callable** is submitted to **ExecutorService**
 - > Call **submit ()** not **execute ()**
 - > **submit ()** returns a **Future** object
- When result is required retrieve using **get ()** method of **Future** object
 - > If result is ready it is returned
 - > If result is not ready calling thread will block

Callable Example

```
class CallableExample implements
    Callable<String> {

    public String call() {
        String result = null;

        /* Do some work and create a result */

        return result;
    }
}
```

Future Example

```
ExecutorService es =  
    Executors.newSingleThreadedExecutor();  
Future<String> f =  
    es.submit (new CallableExample());  
  
/* Do some work in parallel */  
  
try {  
    String callableResult = f.get();  
} catch (InterruptedException ie) {  
    /* Handle */  
} catch (ExecutionException ee) {  
    /* Handle */  
}
```

Locks

- Java provides basic locking via **synchronized**
- Good for many situations, but some issues
 - > Single monitor per object
 - > Not possible to interrupt thread waiting for lock
 - > Not possible to time-out when waiting for a lock
 - > Block structured approach
 - > Acquiring multiple locks is complex
 - > Advanced techniques like hand-over-hand locking are not possible
- New **Lock** interface addresses these issues

Lock Interface

- No automatic unlocking

```
Interface Lock {  
    void lock();  
    void lockInterruptibly() throws IE;  
  
    boolean tryLock();  
    boolean tryLock(long t, TimeUnit u) throws IE;  
    //returns true if lock is aquired  
  
    void unlock();  
    Condition newCondition() throws  
        UnsupportedOperationException;  
}
```

IE = InterruptedException

RentrantLock

- Simplest concrete implementation of Lock
- Same semantics as synchronized, but with more features
- Generally better performance under contention than synchronized
- Remember Lock is not automatically released
 - > Must use a finally block to release
- Multiple wait-sets supported
 - > Using Condition interface

Lock Example

```
Lock lock = new ReentrantLock();

public void accessProtectedResource()
    throws IllegalStateException {
    lock.lock();

    try {
        // Access lock protected resource
    } finally {
        // Ensure lock is always released
        lock.unlock();
    }
}
```

ReadWriteLock Interface

- Has two locks controlling read and write access
 - > Multiple threads can acquire the read lock if no threads have a write lock
 - > Only one thread can acquire the write lock
 - > Implemented as two inner classes
 - > Methods to access locks

```
rwl.readLock().lock();  
rwl.writeLock().lock();
```
 - > **ReentrantReadWriteLock** is simplest concrete implementation
 - > Better performance for read-mostly data access

ReadWriteLock Example

```
ReentrantReadWriteLock rwl = new ReentrantReadWriteLock ();  
Lock rLock = rwl.readLock ();  
Lock wLock = rwl.writeLock ();  
ArrayList<String> data = new ArrayList<String> ();  
  
public String getData(int pos) {  
    r.lock ();  
    try { return data.get(pos); }  
    finally { r.unlock (); }  
}  
  
public void addData(int pos, String value) {  
    w.lock ();  
    try { data.add(pos, value); }  
    finally { w.unlock (); }  
}
```

Synchronizers

- Co-ordinate access and control
- **Semaphore**
 - > Manages a fixed sized pool of resources
- **CountDownLatch**
 - > One or more threads wait for a set of threads to complete an action
- **CyclicBarrier**
 - > Set of threads wait until they all reach a specified point
- **Exchanger**
 - > Two threads reach a fixed point and exchange data

Semaphore Example

```
private Semaphore available;  
private Resource[] resources;  
private boolean[] used;  
public Resource(int poolSize) {  
    available = new Semaphore(poolSize);  
    /* Initialise resource pool */  
}  
public Resource getResource() {  
    try { available.acquire() } catch (IE) {}  
    /* Return resource */  
}  
public void returnResource(Resource r) {  
    /* Return resource to pool */  
    available.release();  
}
```

BlockingQueue Interface

- Provides thread safe way for multiple threads to manipulate collection

```
Interface BlockingQueue<E> {  
    void put(E o) throws IE;  
    boolean offer(E o) throws IE;  
    boolean offer(E o, long t, TimeUnit u) throws IE;  
    E take() throws IE;  
    E poll() throws IE;  
    E poll(long t, TimeUnit u) throws IE;  
    int drainTo(Collection<? super E> c);  
    int drainTo(Collection<? super E> c, int max);  
    // Other methods not listed  
}
```

BlockingQueue Implementations

- **ArrayBlockingQueue**
 - > Bounded queue, backed by an array, FIFO
- **LinkedBlockingQueue**
 - > Optionally bounded queue, backed by linked nodes, FIFO
- **PriorityBlockingQueue**
 - > Unbounded queue
 - > Uses comparator or natural ordering to determine the order of the queue

Blocking Queue Example: 1

```
private BlockingQueue<String> msgQueue;  
  
public Logger(BlockingQueue<String> mq) {  
    msgQueue = mq;  
}  
  
public void run() {  
    try {  
        while (true) {  
            String message = msgQueue.take();  
            /* Log message */  
        }  
    } catch (InterruptedException ie) { }  
}
```

Blocking Queue Example: 2

```
private ArrayBlockingQueue messageQueue =
    new ArrayBlockingQueue<String>(10);
Logger logger = new Logger(messageQueue);

public void run() {
    String someMessage;
    try {
        while (true) {
            /* Do some processing */
            /* Blocks if no space available */
            messageQueue.put(someMessage);
        }
    } catch (InterruptedException ie) { }
}
```

Concurrent Collections

- **ConcurrentMap** (interface)
 - > Extends Map interface with atomic operations
- **ConcurrentHashMap**
 - > Fully concurrent retrieval
 - > Tunable concurrency for updates
 - > Constructor takes number of expected concurrent threads
- **ConcurrentLinkedList**
 - > Unbounded, thread safe queue, FIFO
- **CopyOnWriteArrayList**
 - > Optimised for frequent iteration, infrequent modifications

Java SE 6:

New Concurrency Features

- **Deque** interface for double ended queues
 - > **ArrayDeque**, **LinkedBlockingDeque**, **ConcurrentLinkedDeque**
- Concurrent skiplists
 - > **ConcurrentSkipListMap**, **ConcurrentSkipListSet**
- **AbstractQueuedLongSynchronizer**
 - > Version of **AbstractQueuedSynchronizer** that uses a long for internal state information

Annotations For Concurrency

Document thread-safety

- Build on existing thread-safe classes
 - > But how do you know if a class is thread-safe?
 - > The documentation *should* say, but frequently doesn't
 - > Can be dangerous to guess
 - > Should assume not thread-safe unless otherwise specified
- Document thread-safety design intent
 - > Class annotations: **@ThreadSafe**, **@NotThreadSafe**
@ThreadSafe
`public class ConcurrentHashMap { }`
- With class-level thread-safety annotations:
 - > Clients will know whether the class is thread-safe
 - > Maintainers will know what promises must be kept
 - > Tools can help identify common mistakes

Document thread-safety

- Use `@GuardedBy` to document your locking protocols
- Annotating a field with `@GuardedBy("this")` means:
 - > Only access the field when holding the lock on “this”

`@ThreadSafe`

```
public class PositiveInteger {
    // INVARIANT: value > 0
    @GuardedBy("this") private int value = 1;

    public synchronized int getValue() { return value; }

    public void setValue(int value) {
        if (value <= 0)
            throw new IllegalArgumentException(...);
        synchronized (this) {
            this.value = value;
        }
    }
}
```

- Simplifies maintenance and avoids common mistakes
 - > Like adding a new code path and forgetting to synchronize
 - > Improper maintenance is a big source of concurrency bugs

Use Immutable Objects

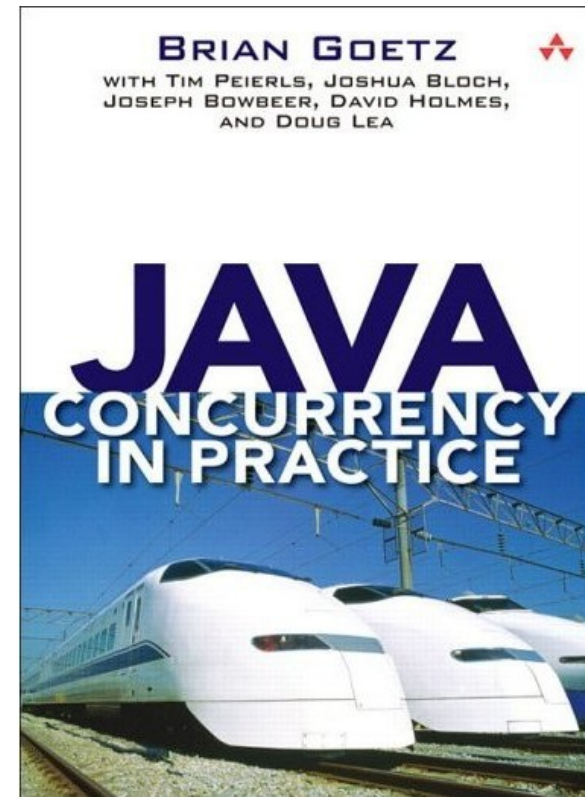
- An immutable object is one whose
 - > State cannot be changed after construction
 - > All fields are final
 - > Not optional – critical for thread-safety of immutable objects
- *Immutable objects are automatically thread-safe!*
- Simpler
 - > Can only ever be in one state, controlled by the constructor
- Safer
 - > Can be freely shared with unknown or malicious code, who cannot subvert their invariants
- More scalable
 - > No synchronization required when sharing!
- Mark with **@Immutable**

Summary

- New concurrency features are very powerful
- Lots of great features
- Take time to learn how to use them correctly
- Use them!

For More Information

- Books
 - > *Java Concurrency in Practice* (Goetz, et al)
 - > See <http://www.jcip.net>
 - > *Concurrent Programming in Java* (Lea)
 - > *Effective Java* (Bloch)



Resources

- java.sun.com/j2se
- www.jcp.org/en/jsr/detail?id=166
- www.jcp.org/en/jsr/detail?id=270 (Java SE 6)
- gee.cs.oswego.edu/dl/cpj/ (Doug Lea)
- Java SE 6.0 javadocs



Thank you!

simon.ritter@sun.com

