Concurrency Extensions in Java 1.5

`java.util.concurrent`

- Adoptation of Dough Lea's utility classes [Lea 00]
- Synchronization: *semaphores*, *barriers*, *latches*, *buffers*
- Execution control: *thread pools*, *futures*

`java.util.concurrent.atomic`

- Simple atomic operations on scalar values
- *Read-compare-update* operations for non-blocking synchr.
- May be implemented by monitors, OS, or hardware

`java.util.concurrent.locks`

- Basic locking mechanisms for critical regions (as in C#)
- Provides *condition queues* (as in Pthreads)
- Uses a primitive event-mechanism for implementation

Locks

- `Lock`: Common interface for mutex-like synchronization primitives

Operations

- For a `Lock` object `l` protecting a critical region:
- `l.lock()` Enter critical region
  `l.unlock()` Leave critical region
  `l.tryLock()` Try to enter region — abandon if occupied.
  `l.newCondition()` Get a condition queue attached to the region.
- Also timed and interruptible versions of the `lock` operation.
- Class `ReentrantLock` is an implementation of `Lock`

Reader/Writer Locks

- A `ReadWriteLock` is a reading lock coupled with a writing lock.

Condition

- Interface for condition queues associated with locks.

Operations

- For a `Condition` object `c` associated with region of lock `l`.
- `c.await()` Atomically leave region and enter queue `c`.
  `c.signal()` Wake a thread in `c` (if any) and continue.
  `c.signalAll()` Wake all threads in `c` and continue.
- Also timed and interruptible versions of `await`
- No means of inspecting queue (size etc.)
- Allows for *spurious wakeups* (idiosyncrasy of Pthreads)

Full Java Monitors

```
class BoundedBuffer {
  final Lock mutex = new ReentrantLock();
  final Condition notFull  = mutex.newCondition();
  final Condition notEmpty = mutex.newCondition();

  final Object[] items = new Object[100];
  int putptr, takeptr, count;

  public void put(Object x) throws InterruptedException {
    mutex.lock();
    try {
      while (count == items.length)
        notFull.await();
      items[putptr] = x;
      if (++putptr == items.length) putptr = 0;
      ++count;
      notEmpty.signal();
    } finally {
      mutex.unlock();
    }
  }

  ⋮
```

Lock Support

- Private (per thread) binary semaphores for implementing locks.
- Used as private *waiting places*

Operations

- The Java `LockSupport` class associates with each thread t:
  - A *permit flag* s_t $(0 \leq s \leq 1)$
- Operations

  `park()` $\langle\, s_{cur} = 1 \rightarrow s_{cur} := 0 \,\rangle$

  `unpark(`t`)` $\langle\, s_t := 1 \,\rangle$
- Also `park` with timeout
- Persistence of permit helps to avoid race conditions

Use of Lock Support

```
class FIFOMutex {
  private int count = 0;
  private Queue<Thread> waiters = new ConcurrentLinkedQueue<Thread>();

  public void lock() {
    Thread current = Thread.currentThread();
    if (current != waiters.peek()) {
       waiters.add(current);

       while (waiters.peek() != current) {
         LockSupport.park();
    }
    count++;
  }

  public void unlock() {
    if (Thread.currentThread() == waiters.peek()) {
      if (--count == 0) {
        waiters.remove();
        LockSupport.unpark(waiters.peek());
      }
    }
    else ...
  }
}
```

Java Semaphores

- Generalized Dijkstra semaphores
- Lots of *peek-and-poke* operations

Operations

- A Java `Semaphore` consists of:
  - A *permission count* s $(s \geq 0)$
  - A queue of waiting threads
  - Queue may be *(strongly) fair* (FIFO)
- Operations

  `s.acquire()` $\langle s > 0 \rightarrow s := s - 1 \rangle$

  `s.release()` $\langle s := s + 1 \rangle$

  `s.acquire(`n`)` $\langle s \geq n \rightarrow s := s - n \rangle$

  `s.release(`n`)` $\langle s := s + n \rangle$

Barrier

- `CyclicBarrier`: General resuable barrier

Workings

- A barrier is created with a fixed threshold: `new CyclicBarrier(`N`)`
- `bar.await()` Await arrival of `N` participants (threads).
  (Returns unique "arrival index")
- The `await`-operation may be timed
- Barrier becomes *broken* upon timeout, interrupt ...
- A broken barrier may be *reset* to the initial state
- An embedded *meeting action* may be provided:

```
new CyclicBarrier(N,new Runnable(){
                      public void run(){ ... }
                  })
```

Latch

- A "one-time barrier"
- Typical usage: Initialization synchronization

Workings

- A `CountDownLatch` object `latch` consists of:
  - A *count* s $(s \geq 0)$ (initialized to $N \geq 0$)
  - A queue of waiting threads
- Operations

  `latch.await()` $\langle$ **await** $s = 0$ $\rangle$

  `latch.countDown()` $\langle$ **if** $s > 0$ **then** $s := s - 1$ $\rangle$
- Cannot be reused or reset

Thread-safe Data Structures

- New `Collection` classes are **not** thread-safe
- A number of thread-safe substitutions are provided
- Efficiently implemented by *wait-free synchronization*

Classes

- Atomic operations, non-blocking:

  `ConcurrentHashMap`, `ConcurrentLinkedQueue`
- Atomic operations, potentially blocking:

  `ArrayBlockingQueue` (N-buffer)
  `LinkedBlockingQueue` (Unbounded buffer)
  `SynchronousQueue` (CSP-channel)
  `PriorityBlockingQueue`
  `DelayQueue`

Atomic Scalar Types

- Provide atomic operations on `Integer`, `Boolean`, `Long`, and `Reference` types and arrays of these
- Operations include: `set`,`get`, `add/sub` and combinations of these
- Based upon *read-check-modify* primitive

Example: AtomicInteger

- Selected operations on `AtomicInteger x` with value v

`x.addAndGet(`d`)`	$\langle\, v := v + d;$ **return** $v \,\rangle$
`x.getAndIncrement()`	$\langle$ **var** $old = v;\ v := v + 1$ **return** $old \,\rangle$
`x.compareAndSet(`old`,`new`)`	$\langle$ **if** $v = old$ **then** $v := new$; **return** $true$ **else return** $false \,\rangle$

Non-Blocking Synchronization

- Update operations avoid waiting/blocking, but may *fail*
- Requires atomic *read-check-modify* instructions

Example with Compare-and-Set

- `AtomicInteger x;`
- Atomic increment $\langle x := x + 1 \rangle$

```
repeat
   old = x.get();
until x.compareAndSet(old, old + 1);
```

- Can be extended to larger data types by updating pointers to structures.
- *Wait-free synchronization* furthermore avoids delay

Execution control

- Machinery (framework) to deal with common usages of threads

Notions

- A *task* is a piece of work represented by an `Runnable` object
- Tasks can be *submitted* to *executors*
- An *executor service* also allows for monitoring of submitted tasks
- A `Callable` object is a task that may compute a result
- A `Future` object represents the status of (cancelable) task
- A *thread pool* is an executor with:
  - A set of threads between a minimum and maximum number
  - A keep-alive time for idle threads
  - A *task queue*: None, bounded, unbounded
- If queue full, tasks may be *rejected* in different ways

Non-blocking IO

- Typical *multi-threaded server*: One thread — one connection
- Simple and efficient for small number (say < 100) of connections
- Large servers can often handle more connections than threads
- Non-blocking IO (`java.nio`): one thread — many connections

Notions

- Connections are considered a case of *channels* (also files ...)
- A set of channels may be associated with a `Selector`
- A thread may *select* among the selector channels
- The thread *awaits* arrival of data on one of the channels