High Level Java Concurrency

• Mutex Locks
• Executors
• Atomic variables
• Concurrent collections
• Random number generation
java.util.concurrent
Mutex Locks

- **Lock interface**
  - `lock()`: acquires a lock and sleeps if necessary
  - `tryLock(ms)`: tries to acquire a lock
    - returns true on success and false on failure
    - can specify optional ms, in which case it will timeout after that length of time
    - tryLock allows thread to back out without sleeping if lock is unavailable
  - `unlock()`: releases the lock
  - `lockInterruptibly()`: like lock but allows thread to be interrupted while waiting by throwing `InterruptedException`
Mutex Locks (cont)

• ReentrantLock
  – implementing class
  – ReentrantLock(fair=false)
    • fair = true: longest waiting thread gets lock
    • avoids starvation
Mutex Example

• Adding together 2 numbers: The add method for each Box should add the value from the parameter Box to the value in this box and print the computed sum
  – Deadlock with synchronized methods: [http://web.eecs.utk.edu/~bvz/cs365/notes/concurrency/BadLock.java](http://web.eecs.utk.edu/~bvz/cs365/notes/concurrency/BadLock.java)
  – Solving deadlock with mutex locks: [http://web.eecs.utk.edu/~bvz/cs365/notes/concurrency/GoodLock.java](http://web.eecs.utk.edu/~bvz/cs365/notes/concurrency/GoodLock.java)
Tasks and Thread Pools

• A *task* is a computation that you want repeated one or more times
  – it should be embedded in a thread
• A *thread pool* is a pool of one or more worker threads to which tasks may be assigned
• When a task is submitted to a thread pool, it is placed on a queue and ultimately executed by one of the worker threads
Executors

- Executors manage thread pools
  - *Executor*, a simple interface that supports launching new tasks.
  - *ExecutorService*, a subinterface of *Executor*, which adds features that help manage the lifecycle, both of the individual tasks and of the executor itself.
  - *ScheduledExecutorService*, a subinterface of *ExecutorService*, supports future and/or periodic execution of tasks.
Executor Class

• The Executor class provides a collection of factory methods that create thread pools which are managed using one of the three desired executor interfaces
Executor Interface

• allows you to submit Runnable tasks to a thread pool via the execute method
ExecutorService

• allows you to submit either Runnable or Callable tasks via the submit method
  – Callable tasks may return a value. This value may be retrieved using the Future object returned by the submit method.
  – The Future object represents the pending result of that task.
    • You access the result using the get() method. The thread will wait until the result is returned
    • The Future object also allows you to cancel the execution of the task
ExecutorService (cont)

• allows you to shutdown a thread pool
  – shutdown(): accepts no new tasks but finishes execution of all running and waiting tasks
  – shutdownNow()
    • accepts no new tasks
    • kills waiting tasks
    • tries to kill running tasks by calling interrupt(): up to each task as to whether or not they actually die
ExecutorService

• Examples
  – ThreadPoolTester
  – CallableTester
ExecutorService
(Fork/Join Pools)

• designed for work that can be recursively divided into smaller tasks

• pseudocode

  if (my portion of the work is small enough)
      do the work directly
  else
      split my work into two pieces
      invoke the two pieces
      wait for the results
• wrap code in a ForkJoinTask subclass, typically either
  – RecursiveTask: returns a value
  – RecursiveAction: does not return a value
• can submit a collection of recursive sub-tasks for execution using ForkJoinTask’s `invokeAll()` method
  – takes an arbitrary length, comma-separated list of ForkJoinTask objects as a parameter
  – returns when isDone() is true for each task
Fork/Join (cont)

• create a ForkJoinPool instance to initiate recursive task
  – call invoke method with ForkJoinTask object

• examples:
  – IncrementTask
  – Sort
ScheduledExecutorService

- Allows you to schedule repeating tasks
  - fixed rate: execute every \( n \) time units (useful for clocks)
  - fixed delay: execute every \( n \) time units after the termination of the current task (can cause drift in a clock)
- Can cancel a repeating task by calling cancel on its returned Future object
ScheduledExecutorService

• Also allows you to schedule a one-shot task at a future time
Example

• The following example prints “beep” every 10 seconds for an hour

http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ScheduledExecutorService.html
Concurrent Collections

• Many collection classes fail “fast” if a concurrent modification is attempted
  – “best effort” only so unreliable

• Synchronized classes: operations are atomic
  – BlockingQueue: FIFO class that blocks when empty or full
    • good for producer/consumer problems
  – ConcurrentMap: good for hash tables
  – ConcurrentNavigable Map: good for sorted maps
  – Vector
ThreadLocalRandom

• A random number generator isolated to current thread
  – internally seeded: seed is not user settable
  – avoids sharing/contention with Math.random()
  – faster than generating your own Random number objects
ThreadLocalRandom (cont)

• Usage:
  ThreadLocalRandom.current().nextX(...) where X is Int, Long, etc

• Bounded ranges also possible
  ThreadLocalRandom.current().nextInt(4, 73);