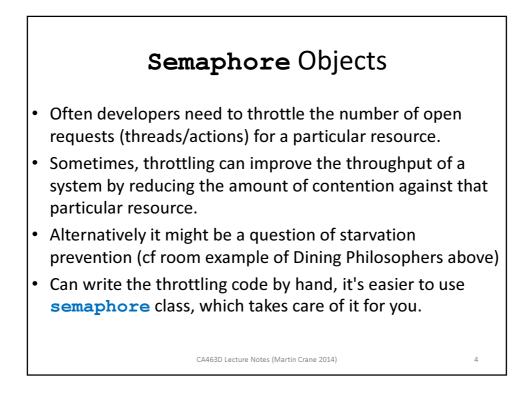
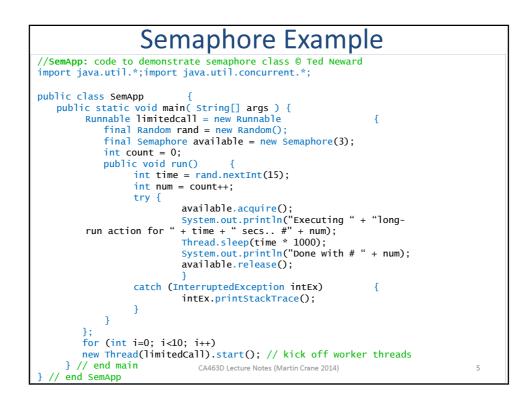


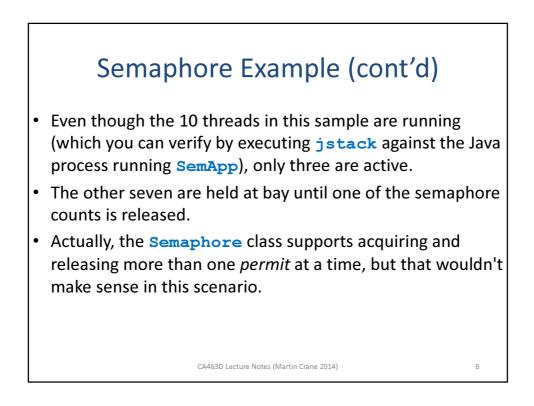
Features in Brief

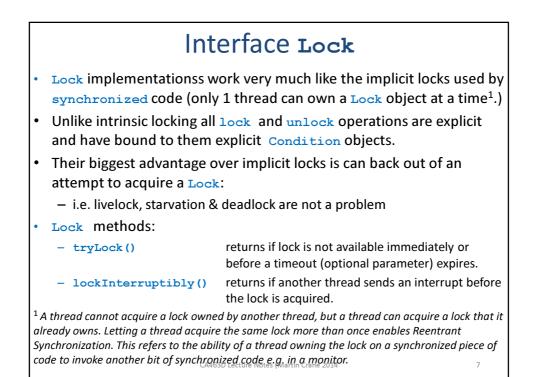
- Semaphore objects are similar to what we have come up against already; acquire () & release () take the place of P, V (resp)
- Lock objects support locking idioms that simplify many concurrent applications (don't confuse with their *implicit* cousins seen above!)
- **Executors** define a high-level API for launching, managing threads.
- **Executor** implementations provide thread pool management suitable for large-scale applications.
- Concurrent Collections support concurrent management of large collections of data in HashTables, different kinds of Queues etc.
- **Future** objects are enhanced to have their status queried and return values when used in connection with asynchronous threads.
- Atomic variables (eg AtomicInteger) support atomic operations on single variables have features that minimize synchronization and help avoid memory consistency errors.

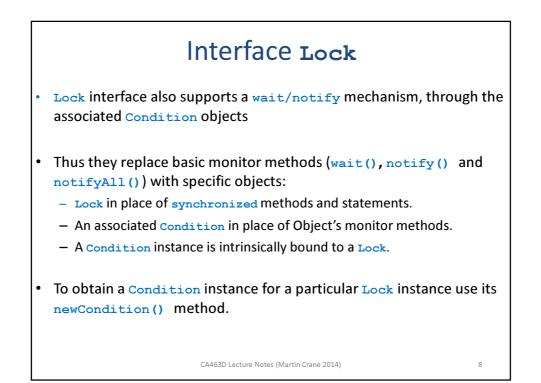
CA463D Lecture Notes (Martin Crane 2014)



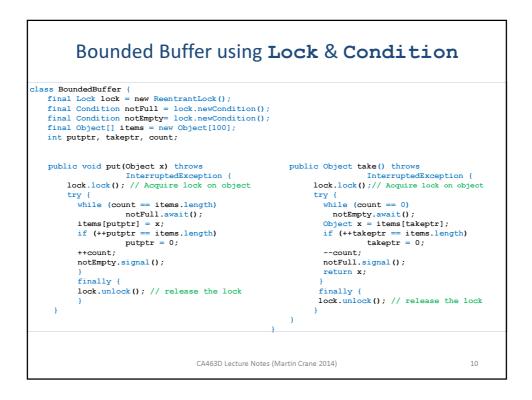


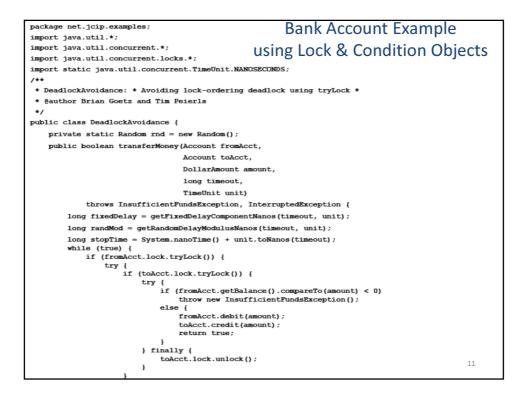




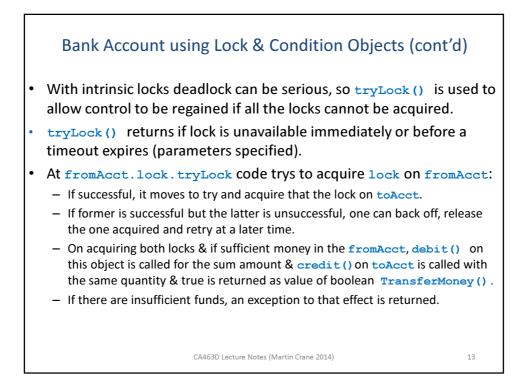


Reentrantlocks & synchronized Methods Reentrantlock implements lock interface with the same mutual exclusion guarantees as synchronized. Acquiring a **Reentrantlock** has the same memory semantics as entering a synchronized block and releasing a Reentrantlock has the same memory semantics as exiting a **synchronized** block. So why use a **Reentrantlock** in the first place? - Using synchronized provides access to the implicit lock associated with every object, but forces all lock acquisition/release to occur in a blockstructured way: if multiple locks are acquired they must be released in the opposite order. - **Reentrantlock** allows for a more flexible locking/releasing mechanism. So why not deprecate synchronized? - Firstly, a lot of legacy Java code uses it and - Secondly, there are performance implications to using **Reentrantlock** CA463D Lecture Notes (Martin Crane 2014)

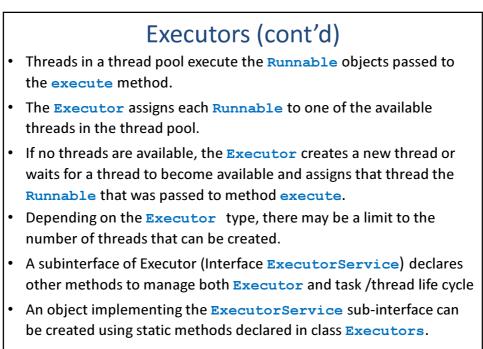




| <pre>} finally { fromAcct.lock.unlock(); } if (System.nanoTime() < stopTime) return false; NANOSECONDS.sleep(fixedDelay + rnd.nextLon } } private static final int DELAY_FIXED = 1; private static final int DELAY_RANDOM = 2; static long getFixedDelayComponentNanos(long timeou return DELAY_FIXED; } static long getRandomDelayModulusNanos(long timeou return DELAY_RANDOM; } static class DollarAmount implements Comparable<do 0;="" compareto(dollaramount="" dollaramount(int="" dollars)="" int="" other)="" pre="" public="" return="" {="" }="" }<=""></do></pre> | Lock & Con g() * randMod); ut, TimeUnit unit; t, TimeUnit unit) | <pre>count Example usin dition Objects (cor class Account { public Lock lock; void debit(DollarAmount d) { void credit(DollarAmount d) { void credit(DollarAmount d) { return null; } class InsufficientFundsException } </pre> | nt'd) |
|--|--|---|-------|
| CA463D Lectu | ure Notes (Martin Crane 2014 | 4) | 12 |

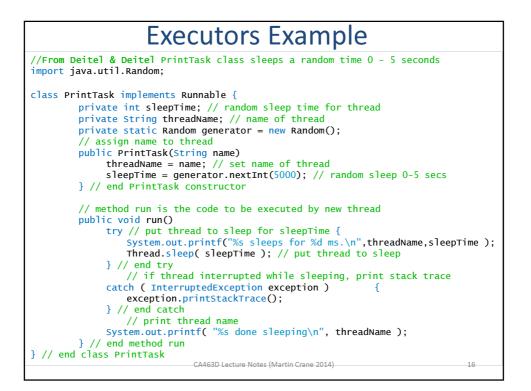


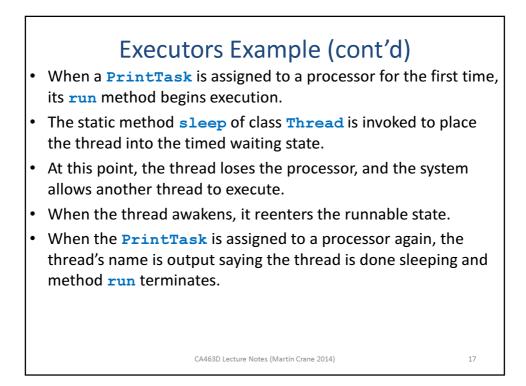
Executors • As seen above, one method of creating a multithreaded application is to implement **Runnable**. • In **J2SE** 5.0, this becomes the *preferred* means (using package java.lang) and built-in methods and classes are used to create Threads that execute the **Runnables**. As also seen, the **Runnable** interface declares a single method named **run**. • Runnables are executed by an object of a class that implements the **Executor** interface. This can be found in package java.util.concurrent. This interface declares a single method named Execute. An Executor object typically creates and manages a group of threads called a *thread pool*. CA463D Lecture Notes (Martin Crane 2014) 14



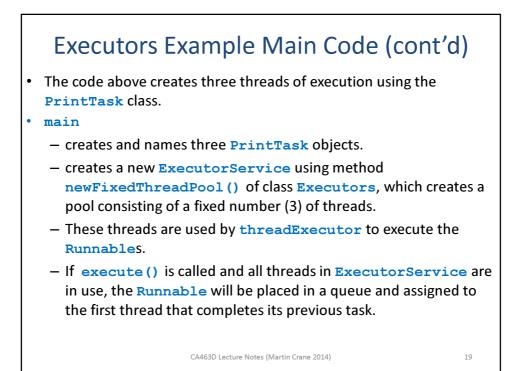
CA463D Lecture Notes (Martin Crane 2014)

15





| Executors Example Main Code | |
|--|----|
| <pre>//RunnableTester: Multiple threads printing at different intervals import java.util.concurrent.Executors; import java.util.concurrent.ExecutorService;</pre> | |
| <pre>public class RunnableTester { public static void main(String[] args) { // create and name each runnable PrintTask task1 = new PrintTask("thread1"); PrintTask task2 = new PrintTask("thread2"); PrintTask task3 = new PrintTask("thread3"); } }</pre> | |
| <pre>System.out.println("Starting threads"); // create ExecutorService to manage threads ExecutorService threadExecutor</pre> | |
| <pre>threadExecutor.shutdown(); // shutdown worker threads System.out.println("Threads started, main ends\n"); } // end main } // end RunnableTester</pre> | |
| CA463D Lecture Notes (Martin Crane 2014) | 18 |



Executors Example Main Code (cont'd) Sample Output

Starting threads Threads started, main ends thread1 sleeps for 1217 ms. thread2 sleeps for 3989 ms. thread3 sleeps for 662 ms. thread3 done sleeping thread1 done sleeping thread2 done sleeping

CA463D Lecture Notes (Martin Crane 2014)

20



- Pre-Java 8 version of **Futures** was quite weak, only supporting waiting for future to complete.
- Also executor framework above works with Runnables
 & Runnable cannot return a result.
- A Callable object allows return values after completion.
- The **Callable** object uses generics to define the type of object which is returned.
- If you submit a Callable object to an Executor, framework returns java.util.concurrent.Future.
- This **Future** object can be used to check the status of a **Callable** and to retrieve the result from the **Callable**.

CA463D Lecture Notes (Martin Crane 2014)

