Lecture 7: Enhanced Concurrency in Java

- **`java.util.concurrent:`**
  - **`Semaphore class`**
  - **`Interface Lock/ Class Condition`**
  - **`Bounded Buffers`** Implementation
  - **`Bank Account`** Implementation
  - **`Interface Executor`**
  - **`Futures/Callables`**

Recent Developments in `java.util.concurrent`

- Up to now, have focused on the low-level APIs that have been part of the Java platform from the very beginning.
- These APIs are adequate for basic tasks, but need higher-level constructs for more advanced tasks (esp for massively parallel applications exploiting multi-core systems).
- In this lecture we'll examine some high-level concurrency features introduced in more recent Java releases.
- Most of these features are implemented in the new `java.util.concurrent` packages.
- There are also new concurrent data structures in the Java `Collections` Framework.

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!)
- `Executor`s define a high-level API for launching, managing threads.
- `Executor` implementations provide thread pool management suitable for large-scale applications.
- Concurrent `Collection`s 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.

`Semaphore` Objects

- Often developers need to throttle the number of open requests (threads/actions) for a particular resource.
- Sometimes, throttling can improve the throughput of a system by reducing the amount of contention against that particular resource.
- Alternatively it might be a question of starvation prevention (cf room example of Dining Philosophers above)
- Can write the throttling code by hand, it's easier to use **`semaphore`** class, which takes care of it for you.

Semaphore Example

```
//SemApp: code to demonstrate semaphore class © Ted Neward
import java.util.*;import java.util.concurrent.*;

public class SemApp      {
    public static void main( String[] args ) {
          Runnable limitedcall = new Runnable                  {
              final Random rand = new Random();
              final Semaphore available = new Semaphore(3);
              int count = 0;
              public void run()      {
                    int time = rand.nextInt(15);
                    int num = count++;
                    try {
                              available.acquire();
                              System.out.println("Executing “ + “long-
          run action for “ + time + “ secs.. #" + num);
                              Thread.sleep(time * 1000);
                              System.out.println("Done with # “ + num);
                              available.release();
                              }
                    catch (InterruptedException intEx)              {
                              intEx.printStackTrace();
                    }
              }
          };
          for (int i=0; i<10; i++)
          new Thread(limitedCall).start(); // kick off worker threads
      } // end main
} // end SemApp
```

Semaphore Example (cont'd)

- Even though the 10 threads in this sample are running (which you can verify by executing `jstack` against the Java process running `SemApp`), only three are active.
- The other seven are held at bay until one of the semaphore counts is released.
- Actually, the `Semaphore` class supports acquiring and releasing more than one *permit* at a time, but that wouldn't make sense in this scenario.

Interface `Lock`

- `Lock` implementationss work very much like the implicit locks used by `synchronized` code (only 1 thread can own a `Lock` object at a time[1].)
- Unlike intrinsic locking all `lock` and `unlock` operations are explicit and have bound to them explicit `Condition` objects.
- Their biggest advantage over implicit locks is can back out of an attempt to acquire a `Lock`:
  - i.e. livelock, starvation & deadlock are not a problem
- `Lock` methods:
  - `tryLock()` returns if lock is not available immediately or before a timeout (optional parameter) expires.
  - `lockInterruptibly()` returns if another thread sends an interrupt before the lock is acquired.

[1] *A thread cannot acquire a lock owned by another thread, but a thread can acquire a lock that it already owns. Letting a thread acquire the same lock more than once enables Reentrant Synchronization. This refers to the ability of a thread owning the lock on a synchronized piece of code to invoke another bit of synchronized code e.g. in a monitor.*

Interface `Lock`

- `Lock` interface also supports a `wait/notify` mechanism, through the associated `Condition` objects
- Thus they replace basic monitor methods (`wait()`, `notify()` and `notifyAll()`) with specific objects:
  - `Lock` in place of `synchronized` methods and statements.
  - An associated `Condition` in place of Object's monitor methods.
  - A `Condition` instance is intrinsically bound to a `Lock`.
- To obtain a `Condition` instance for a particular `Lock` instance use its `newCondition()` method.

`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 block-structured 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`

Bounded Buffer using **Lock** & **Condition**

```
class BoundedBuffer {
   final Lock lock = new ReentrantLock();
   final Condition notFull = lock.newCondition();
   final Condition notEmpty= lock.newCondition();
   final Object[] items = new Object[100];
   int putptr, takeptr, count;

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

   public Object take() throws
                   InterruptedException {
       lock.lock();// Acquire lock on object
       try {
         while (count == 0)
           notEmpty.await();
         Object x = items[takeptr];
         if (++takeptr == items.length)
                   takeptr = 0;
         --count;
         notFull.signal();
         return x;
        }
        finally {
        lock.unlock(); // release the lock
       }
   }
}
```

Bank Account Example using Lock & Condition Objects

```
package net.jcip.examples;
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import static java.util.concurrent.TimeUnit.NANOSECONDS;
/**
 * DeadlockAvoidance: * Avoiding lock-ordering deadlock using tryLock *
 * @author Brian Goetz and Tim Peierls
 */
public class DeadlockAvoidance {
    private static Random rnd = new Random();
    public boolean transferMoney(Account fromAcct,
                                 Account toAcct,
                                 DollarAmount amount,
                                 long timeout,
                                 TimeUnit unit)
            throws InsufficientFundsException, InterruptedException {
        long fixedDelay = getFixedDelayComponentNanos(timeout, unit);
        long randMod = getRandomDelayModulusNanos(timeout, unit);
        long stopTime = System.nanoTime() + unit.toNanos(timeout);
        while (true) {
            if (fromAcct.lock.tryLock()) {
                try {
                    if (toAcct.lock.tryLock()) {
                        try {
                            if (fromAcct.getBalance().compareTo(amount) < 0)
                                throw new InsufficientFundsException();
                            else {
                                fromAcct.debit(amount);
                                toAcct.credit(amount);
                                return true;
                            }
                        } finally {
                            toAcct.lock.unlock();
                        }
                    }
```

Bank Account Example using Lock & Condition Objects (cont'd)

```
            } finally {
                fromAcct.lock.unlock();
            }
        }
        if (System.nanoTime() < stopTime)
            return false;
        NANOSECONDS.sleep(fixedDelay + rnd.nextLong() % randMod);
    }
}


private static final int DELAY_FIXED = 1;
private static final int DELAY_RANDOM = 2;

static long getFixedDelayComponentNanos(long timeout, TimeUnit unit]
    return DELAY_FIXED;
}

static long getRandomDelayModulusNanos(long timeout, TimeUnit unit)
    return DELAY_RANDOM;
}

static class DollarAmount implements Comparable<DollarAmount> {
    public int compareTo(DollarAmount other) {
        return 0;
    }

    DollarAmount(int dollars) {
    }
}
```

```
class Account {
    public Lock lock;

    void debit(DollarAmount d) {
    }

    void credit(DollarAmount d) {
    }

    DollarAmount getBalance() {
        return null;
    }
}

class InsufficientFundsException extends ]
}
```

```
}
```

Bank Account using Lock & Condition Objects (cont'd)

- With intrinsic locks deadlock can be serious, so `tryLock()` is used to allow control to be regained if all the locks cannot be acquired.
- `tryLock()` returns if lock is unavailable immediately or before a timeout expires (parameters specified).
- At `fromAcct.lock.tryLock` code trys to acquire `lock` on `fromAcct`:
  - If successful, it moves to try and acquire that the lock on `toAcct`.
  - If former is successful but the latter is unsuccessful, one can back off, release the one acquired and retry at a later time.
  - On acquiring both locks & if sufficient money in the `fromAcct`, `debit()` on this object is called for the sum amount & `credit()` on `toAcct` is called with the same quantity & true is returned as value of boolean `TransferMoney()`.
  - If there are insufficient funds, an exception to that effect is returned.

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 **`Runnable`**s.
- As also seen, the **`Runnable`** interface declares a single method named **`run`**.
- **`Runnable`**s 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*.

Executors (cont'd)

- Threads in a thread pool execute the **`Runnable`** objects passed to the **`execute`** method.
- The **`Executor`** assigns each **`Runnable`** to one of the available threads in the thread pool.
- If no threads are available, the **`Executor`** creates a new thread or waits for a thread to become available and assigns that thread the **`Runnable`** that was passed to method **`execute`**.
- Depending on the **`Executor`** type, there may be a limit to the number of threads that can be created.
- A subinterface of Executor (Interface **`ExecutorService`**) declares other methods to manage both **`Executor`** and task /thread life cycle
- An object implementing the **`ExecutorService`** sub-interface can be created using static methods declared in class **`Executors`**.

Executors Example

```
//From Deitel & Deitel PrintTask class sleeps a random time 0 - 5 seconds
import java.util.Random;

class PrintTask implements Runnable {
        private int sleepTime; // random sleep time for thread
        private String threadName; // name of thread
        private static Random generator = new Random();
        // assign name to thread
        public PrintTask(String name)
             threadName = name; // set name of thread
             sleepTime = generator.nextInt(5000); // random sleep 0-5 secs
        } // end PrintTask constructor

        // method run is the code to be executed by new thread
        public void run()
             try // put thread to sleep for sleepTime {
                 System.out.printf("%s sleeps for %d ms.\n",threadName,sleepTime );
                 Thread.sleep( sleepTime ); // put thread to sleep
             } // end try
                 // if thread interrupted while sleeping, print stack trace
             catch ( InterruptedException exception )        {
                 exception.printStackTrace();
             } // end catch
                 // print thread name
             System.out.printf( "%s done sleeping\n", threadName );
        } // end method run
} // end class PrintTask
```

Executors Example (cont'd)

- When a **PrintTask** is assigned to a processor for the first time, its **run** method begins execution.
- The static method **sleep** of class **Thread** is invoked to place the thread into the timed waiting state.
- At this point, the thread loses the processor, and the system allows another thread to execute.
- When the thread awakens, it reenters the runnable state.
- When the **PrintTask** is assigned to a processor again, the thread's name is output saying the thread is done sleeping and method **run** terminates.

Executors Example Main Code

```
//RunnableTester: Multiple threads printing at different intervals
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

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" );

                System.out.println( "Starting threads" );

                // create ExecutorService to manage threads
                ExecutorService threadExecutor
                        = Executors.newFixedThreadPool( 3 );
                // start threads and place in runnable state
                threadExecutor.execute( task1 ); // start task1
                threadExecutor.execute( task2 ); // start task2
                threadExecutor.execute( task3 ); // start task3

                threadExecutor.shutdown(); // shutdown worker threads

                System.out.println( "Threads started, main ends\n" );
    } // end main
} // end RunnableTester
```

Executors Example Main Code (cont'd)

- The code above creates three threads of execution using the `PrintTask` class.
- `main`
  - creates and names three `PrintTask` objects.
  - creates a new `ExecutorService` using method `newFixedThreadPool()` of class `Executors`, which creates a pool consisting of a fixed number (3) of threads.
  - These threads are used by `threadExecutor` to execute the `Runnable`s.
  - If `execute()` is called and all threads in `ExecutorService` are in use, the `Runnable` will be placed in a queue and assigned to the first thread that completes its previous task.

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
```

Futures/Callables

- Pre-Java 8 version of **Futures** was quite weak, only supporting waiting for future to complete.
- Also **executor** framework above works with **Runnable**s

& **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**.

Futures/Callables[1] (cont'd)

```java
package de.vogella.concurrency.callables;
import java.util.concurrent.Callable;
public class MyCallable implements Callable<Long> {
  @Override
  public Long call() throws Exception {
    long sum = 0;
    for (long i = 0; i <= 100; i++) {
      sum += i;
    }
    return sum;
  }
}
```

[1]This code and associated piece on the next page were written and are Copyright © Lars Vogel. Source Code can be found at *de.vogella.concurrency.callables*.

Futures/ Callables[1] (cont'd)

```java
package de.vogella.concurrency.callables;
import java.util.ArrayList;
import java.util.List;import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;import java.util.concurrent.Future;
public class CallableFutures {
  private static final int NTHREDS = 10;
  public static void main(String[] args) {
    ExecutorService executor = Executors.newFixedThreadPool(NTHREDS);
    List<Future<Long>> list = new ArrayList<Future<Long>>();
    for (int i = 0; i < 20000; i++) {
      Callable<Long> worker = new MyCallable();
      Future<Long> submit = executor.submit(worker);
      list.add(submit);
    }
    long sum = 0;
    System.out.println(list.size());
    // now retrieve the result
    for (Future<Long> future : list) {
      try {
        sum += future.get();
      } catch (InterruptedException e) {
        e.printStackTrace();
      } catch (ExecutionException e) {
        e.printStackTrace();
      }
    }
    System.out.println(sum);executor.shutdown();
  }
}
```


[1] Copyright © Lars Vogel, 2013