CS181, Programming Languages
Assignment 5
(due: 11-20-2000)

For both problems submit only the source files (.java) using the WWW-Turnin. Remember to include your name in the files!

Problem one (Blocking Queue)

A blocking queue implementation is supposed to block a worker trying to dequeue something from the head of the queue, whenever the queue is empty. Imagine having a queue for storing user GUI requests, like button presses or menu actions. A GUI thread usually dequeues such requests from the queue and processes them sequentially. If no such requests are present, the GUI thread would have to check the queue for new events after some predefined timeout, consuming system resources. A better approach would be to suspend the thread and notify it when a new event arrives. A blocking queue works as follows. It implements the **enqueue** method that adds a new request at the end of the queue and notifies any waiting threads using the *Object.notify()* method. It also implements the **dequeue** method, that returns the action at the head of the queue, or blocks the calling thread using the *Object.wait()* method, until a new request is enqueued and *notify* is called.

Define the **Blocking Queue** class that implements the following methods:

- **public synchronized void enqueue(Object o)**: Adds an object at the end of the queue and notifies any waiting thread that the queue is not empty anymore.
- **public synchronized Object dequeue()**: Returns the object at the head of the queue or blocks the calling thread. (Hint: special care must be taken when multiple competing threads request objects from the queue. Some thread might be notified and in the mean time some other thread might have dequeued the last entry from the queue, thus leaving the queue empty and a worker thread without real work!)
- **public synchronized void close()**: Closes the blocking queue. Further enqueue or dequeue requests throw a closed queue exception. Also notifies all waiting threads that the queue has closed.

Use an **ArrayList** to implement the queue that holds request objects.

Problem two (Thread Pool)

In some applications it is appropriate to create a new thread each time a new request arrives into the system. There might be a case though when some actions are really demanding and consume a lot of CPU time. A lot of threads will be created and stay active for a long time, thus exhausting system resources. A better approach is to restrict the number of threads active at one time and also reuse each thread in order to avoid the creation and initialization overhead that is induced.

A thread pool (also known as work queue), uses a predefined number of threads (workers) for processing requests. The pool is initialized with only one thread awaiting at the head of a *blocking queue.* When a new request arrives, if a thread is not available to process it, a new worker must be created for that purpose. If the maximum number of threads is reached, the request must remain in the queue, until some worker becomes available. When no requests are available, workers should remain blocked at the head of the queue.

Implement the **ThreadPool** class using a **BlockingQueue** for queueing request objects. Implement the following methods:

- **public void execute(Runnable action)**: Checks if the queue is empty and if not, creates a new worker thread for processing a request, if the maximum number of allowable threads is not reached. Also, places the new action at the end of the queue. Notice that in the beginning one thread will be waiting at an empty queue, so no new threads need to be created. The new action will be enqueued and the original thread will begin executing it. If a new action is enqueued before the original thread finishes processing the old action, a new thread should be created, assuming that the maximum number of allowable threads is not yet reached.

- **public synchronized void close()**: Closes the thread pool, shutting down all workers and the blocking queue.

Implement a constructor for the **ThreadPool** class that accepts two arguments. The number of initially blocked threads (default is one) and the number of maximum threads allowed.

Hint: You should define a **PooledThread** class as shown below. When *execute()* creates a new thread, it should create a thread of this kind.

```
// this is an inner class of ThreadPool.
private class PooledThread extends Thread {
```

```
  public void run() {
    try {
      while (!closed) {
        // dequeue a runnable object from the queue and begin executing it.
        // this thread will block here, if the queue is empty.
        ((Runnable) queue.dequeue()).run();
      }
    } catch (ClosedQueueException e) {
    }
  }
}
```