Concurrency:
Multi-core Programming
& Data Processing

Lab 8

-- Queue-like constructs in Java --

ConcurrentSkipListSet class

- Safe concurrent access to a set/list
- Elements ordered in natural order (unless Comparator defined)
- Main operations:
  - boolean add(E e);
  - boolean remove(Object o);
  - boolean contains(Object o);
  - first() ; last() ; ceiling (E e) ; floor (E e)

ConcurrentLinkedQueue class

- Same as above, but with queue semantics:
  - boolean add(E e)
  - E poll()
  - E peek()
  - ...and even contains(Object o) and remove(Object o) (more list-like methods)
- No order imposed

ArrayBlockingQueue

- **Bounded** concurrent queue backed by an **array**
- Classical queue methods +
- Methods that automatically block the current thread until the operation is possible:
  - put(E e)
  - take()

LinkedBlockingQueue

- Same as above but backed by a **linked-list**
- <u>Optionally bounded</u>
- By default capacity is Integer.MAX_VALUE
    - can be set to something else

PriorityBlockingQueue

- Priority ordering of the elements inserted
  - natural order (default)
  - Comparator
- Elements **must** be comparable
- Head of the queue: "smallest" element
- <u>Unbounded</u> (default): blocks in *take()* if queue empty

DelayQueue

- Unbounded queue
- Elements **must** extend *Delayed* interface
    - time delay is associated to each element
    - removal of elements is delayed until the time expires
- Head of the queue: element with the shortest time left
- *Take()* blocks if there is no element with expired time

SynchronousQueue

- Rendezvous channel with <u>no capacity</u>
- Any *take()* call will block until an associated *put()* happens at the other end
  - ...and vice versa
- No *peak()*, no iterate, no null insertion
- For simple handoff designs

Exercise

- Implement an unbounded locked queue and unbounded lockfree queue
- Starting examples: bounded locked queue and unbounded lockfree stack in the lecture, code-examples archive on ILIAS
- Detailed instructions:
  - two "pointers", head and tail to refer to the ends of the queue in order to be able to enqueue (on the tail) and dequeue (from the head)
  - initialization: the empty queue, head and tail should refer to the same node (sentinel node – no actual value)
  - enqueue: link a node to tail.next and "move" the tail reference on the newly added next node
  - dequeue: get the value from head.next and "move" the head on the next node