IT UNIVERSITY OF COPENHAGEN

SUBMISSION OF WRITTEN WORK

Class code: PRCPP

Name of course: Practical Concurrent and Parallel Programming

Course manager: Peter Sestoft

Course e-portfolio:

Thesis or project title:

Supervisor:

Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1. Jacob Benjamin Cholewa	29/03-1992	jbec@itu.dk
2.		@itu.dk
3.		@itu.dk
4.		@itu.dk
5.		@itu.dk
6.		@itu.dk
7.		@itu.dk

Practical Concurrent and Parallel Programming

Exam Assignment January 2016

Jacob B. Cholewa
jbec@itu.dk

January 12, 2016

I hereby declare that I have answered these exam questions myself without any outside help.

Jacob Cholewa

Signature

12/01/16

Date

Question 1

Part 1.1

The output for `TestLocking0.java` clearly indicates that it is not thread-safe.

```
mac610262:src jbec$ java TestLocking0
Sum is 1505632.000000 and should be 2000000.000000
mac610262:src jbec$ java TestLocking0
Sum is 1490208.000000 and should be 2000000.000000
mac610262:src jbec$ java TestLocking0
Sum is 1497894.000000 and should be 2000000.000000
mac610262:src jbec$ java TestLocking0
Sum is 1505498.000000 and should be 2000000.000000
```

The actual sum deviates from the expected sum leading me to believe that a race condition occurs in the code.

Part 1.2

The problem is that while the `addInstance` method locks the object instance, `addStatic` locks the class. Hence the field `sum` is guarded by multiple locks. This allows for multiple threads to simultaneously access the `sum` variable, therefore not upholding mutual exclusion, potentially causing the race condition making it not threadsafe.

Part 1.3

A simple solution is to change `addInstance` so that it is guarded by the class lock used by the static synchronized methods

```java
public void addInstance(double x) {
    synchronized(Mystery.class){
        sum += x;
    }
}
```

This ensures mutual exclusion as `sum` is now guarded by the same lock. Rerunning the code shows that the expected result and the actual result are now the same.

```
mac610262:src jbec$ java TestLocking0
Sum is 2000000.000000 and should be 2000000.000000
mac610262:src jbec$ java TestLocking0
Sum is 2000000.000000 and should be 2000000.000000
mac610262:src jbec$ java TestLocking0
Sum is 2000000.000000 and should be 2000000.000000
mac610262:src jbec$ java TestLocking0
Sum is 2000000.000000 and should be 2000000.000000
```

Question 2

Part 2.1

The simplest way would be to make a synchronized version that guards `items` and `size` with an instance lock. This would ensure safe concurrent access to the arraylist. This is implemented by adding the synchronized keyword to all methods in the class.

Part 2.2

While the naïve approach described above makes the arraylist threadsafe, it does not allow parallel access and thus doesn't scale. Actually I expect the synchronized version to perform significantly worse when used by many threads compared to only a single thread.

Part 2.3

A simple answer to why the purposed pattern is not threadsafe is found in the sample given in the assignment. The example clearly shows that both `add` and `set` accesses `items` and `size`. As the methods uses different locks, concurrent access to `items` and `size` can occur making it not threadsafe.

> "When thread *A* executes a synchronized block, and subsequently thread *B* enters a synchronized block guarded by the same lock, the values of variables that were visible to *A* prior to releasing the lock are guaranteed to be visible to *B* upon acquiring the lock."
>
> — Goetz and Peierls [1, p. 37]

Because the methods uses different locks, visibility is also not guaranteed between threads.

Part 2.4

While it is possible that a version might exist that makes this threadsafe, it will still require mutual exclusion when accessing `items` and `size`. I don't see many other (simple) ways than to fully lock both `items` and `size` when accessed. Thus it won't make much sense to have a lock for the methods if we either way have to lock the only two shared fields. Then we could as well just only lock those instead.

Question 3

Part 3.1

The `totalSize` field can be made threadsafe by either using an `AtomicInteger` or by having `totalSize` guarded by a static lock object. The following snippet shows how it would be implemented in the code using an `AtomicInteger`

Listing 3.1: Making totalSize threadsafe using an `AtomicInteger`

```
private static AtomicInteger totalSize = new AtomicInteger();

public boolean add(double x) {
    if (size == items.length) {
        ...
    }
    items[size] = x;
    size++;
    totalSize.incrementAndGet();
    return true;
}

public static int totalSize() {
    return totalSize.get();
}
```

Part 3.2

The `allLists` field can be make threadsafe by guarding it with a static lock object. It can be implemented in code in following way.

Listing 3.2: Making `allLists` threadsafe by guarding it with a static lock

```
private static HashSet<DoubleArrayList> allLists = new HashSet<>();
private static final Object ListsLock = new Object();

public DoubleArrayList() {
    synchronized(ListsLock){
        allLists.add(this);
    }
}

public static HashSet<DoubleArrayList> allLists() {
    synchronized(ListsLock){
        return allLists;
    }
}
```

Question 4

Part 4.1

The following code shows the implementation of the Sorting state implemented as described in the assignment text.

Listing 4.1: Implemented code for the Sorting stage

```
static class SortingStage implements Runnable {
    private final BlockingDoubleQueue in;
    private final BlockingDoubleQueue out;
    private final double[] heap;
    private int itemCount;
    private int heapSize = 0;

    public SortingStage(BlockingDoubleQueue in, BlockingDoubleQueue out,
        int capacity, int itemCount){
        this.in = in;
        this.out = out;
        this.itemCount = itemCount;
        heap = new double[capacity];
    }

    public void run() {
        while(itemCount > 0){
            double x = in.take();
            if(heapSize < heap.length){ //heap not full, put x into it
                heap[heapSize++] = x;
                DoubleArray.minheapSiftup(heap, heapSize-1, heapSize-1);
            } else if (x <= heap[0]){ //x is small, forward
                out.put(x);
                itemCount--;
            } else { //forward least, replace with x
                double least = heap[0];
                heap[0] = x;
                DoubleArray.minheapSiftdown(heap,0,heapSize-1);
                out.put(least);
                itemCount--;
            }
        }
    }
}
```

See the fully implemented `SortingPipeline.java` in appendix A.1

Part 4.2

The following code shows how the pipeline is initiated and started.

Listing 4.2: Implemented code for setting up and starting stages

```java
private static void sortPipeline(double[] arr, int P, BlockingDoubleQueue[]
     queues) {
    int n = arr.length / P;

    //Initializing the sorting stages
    Thread[] threads = new Thread[P+2];
    for(int i = 1; i <= P; i++){
        threads[i-1] = new Thread(new SortingStage(queues[i-1], queues[i],
            n, arr.length+(P-i)*n));
    }

    //Initializing the drain
    threads[P] = new Thread(new SortedChecker(arr.length, queues[P]));

    //Initializing the source. The source is purposefully last in the array
         so that it will be started lastly.
    threads[P+1] = new Thread(new DoubleGenerator(arr, arr.length, queues
        [0]));

    //Starting the stages
    for(int i = 0; i < threads.length; i++){
        threads[i].start();
    }

    //Joining the stages
    for(int i = 0; i < threads.length; i++){
        try{ threads[i].join(); }
        catch(InterruptedException e){ throw new RuntimeException(e); }
    }
}
```

Question 5

Part 5.1

The following code wraps `ArrayBlockingQueue` so that it fits our `BlockingDoubleQueue` interface.

Listing 5.1: Implemented code for wrapping `ArrayBlockingQueue`

```
class WrappedArrayDoubleQueue implements BlockingDoubleQueue{
    private final ArrayBlockingQueue<Double> queue;

    public WrappedArrayDoubleQueue(){
        this.queue = new ArrayBlockingQueue<Double>(50);
    }

    public WrappedArrayDoubleQueue(int capacity){
        this.queue = new ArrayBlockingQueue<Double>(capacity);
    }

    public void put(double item){
        try{ queue.put(item); }
        catch(InterruptedException e){ throw new RuntimeException(e); }
    }

    public double take(){
        try{ return queue.take(); }
        catch(InterruptedException e){ throw new RuntimeException(e); }
    }
}
```

Part 5.2

The result of running the code results in a sorted list of elements as expected. The program terminated by itself indicating that all the stages ended as desired.

```
mac610262:src jbec$ java SortingPipeline
0.1 1.1 2.1 3.1 4.1 5.1 6.1 7.1 8.1 9.1 10.1 11.1 12.1 13.1 14.1 15.1 16.1
    17.1 18.1 19.1 20.1 21.1 22.1 23.1 24.1 25.1 26.1 27.1 28.1 29.1 30.1
    31.1 32.1 33.1 34.1 35.1 36.1 37.1 38.1 39.1
```

Part 5.3

The results of benchmarking the current pipeline with $count = 100.000$ and $P = 4$. It is difficult to comment on the performance yet without having it compared to other implementations, but considering the implementation and the number of elements to sort, the results seams fair.

```
# OS:   Mac OS X; 10.11; x86_64
# JVM:  Oracle Corporation; 1.8.0_60
# CPU:  null; 8 "cores"
# Date: 2016-01-11T14:45:02+0100
Sorting pipe    125.9 ms    1.71    4
```


Jacob B. Cholewa jbec@itu.dk

Question 6

Part 6.1

The following queue implementation is inspired by the `OneItemQueue` presented in the slides from lecture 5. It uses a cyclic array. The cyclic array is implemented using a normal array and a `head` and `tail` pointer. The pointers will loop back to zero if they get out of bound providing the cyclic behavior. The blocking is implemented using `wait` and `notify`.

Listing 6.1: Implementation of the `BlockingNDoubleQueue` blocking fixed size queue

```java
class BlockingNDoubleQueue implements BlockingDoubleQueue{
    private final double[] arr = new double[50];
    private int head = 0, tail = 0, count = 0;

    public synchronized void put(double item){
        while(count == arr.length){
            try{ this.wait(); }
            catch(InterruptedException exn) { }
        }

        arr[tail] = item;
        tail = ++tail == arr.length ? 0 : tail;
        count++;
        this.notify();
    }

    public synchronized double take(){
        while(count == 0){
            try{ this.wait(); }
            catch(InterruptedException exn) { }
        }

    double item = arr[head];
        head = ++head == arr.length ? 0 : head;
        count --;
        this.notify();
        return item;
    }
    }
```

Part 6.2

The queue presented above is threadsafe as the only two methods, `put` and `take` is synchronized. Thus the array, the two pointers, and the counter is guarded by the instance object meaning that only one thread can access them at a time. A thread can be blocked - forced to wait - if the queue is either empty or full. The thread will be forced release the lock and wait until the blocking condition is no longer met. Then the thread will again try to acquire the lock before continuing. This thereby follow the monitor pattern making it threadsafe.

Part 6.3

Running the pipeline with the `BlockingNDoubleQueue` yields following results.

```
# OS:   Mac OS X; 10.11; x86_64
# JVM:  Oracle Corporation; 1.8.0_60
```


Jacob B. Cholewa jbec@itu.dk

```
# CPU:  null; 8 "cores"
# Date: 2016-01-11T16:52:52+0100
Sorting pipe    470.4 ms    36.56   2
```

These results are quite interesting as it becomes clear that the new queue is a performance bottleneck compared to the results from 5.3 as these results are almost 4 times slower. This might be due to that fact that this queue allows for no parallelism plausibly causing threads to be often blocked by each other.

Question 7

Part 7.1

The following code presents an unbounded queue. The queue is implemented using a linkedlist to make it unbounded. This means, in contrast to the previous queue, that it can hold as unlimited number of items (while available memory still, of course, is a limitation). This queue is also blocking forcing threads to wait if no elements is available in the queue.

Listing 7.1: Implementation of the `UnboundedDoubleQueue` blocking queue

```java
class UnboundedDoubleQueue implements BlockingDoubleQueue{

    public Node head;
    public Node tail;

    public UnboundedDoubleQueue(){
        Node n = new Node(0,null);
        tail = head = n;
    }

    public synchronized void put(double item){
        tail.next = new Node(item,null); //Setting next
        tail = tail.next; //Moving tail

        this.notify(); //Notifying a thread waiting for elements
    }

    public synchronized double take(){
        while(head.next == null){
            try{ this.wait(); }
            catch(InterruptedException exn) { }
        }

        Node first = head;
        head = first.next;
        return head.value;
    }

    class Node{
        public Node next;
        public final double value;

        public Node(double value, Node next){
            this.next = next;
            this.value = value;
        }
    }
}
```

Part 7.2

This implementation also follows the monitor pattern and is therefore threadsafe. In this queue, producers are never forced to wait as the queue is unbounded. Only consumers wait if there is no more elements to consume.

Part 7.3

```
# OS:   Mac OS X; 10.11; x86_64
# JVM:  Oracle Corporation; 1.8.0_60
# CPU:  null; 8 "cores"
# Date: 2016-01-11T18:01:39+0100
Sorting pipe    262.1 ms    14.29   2
```

This queue performs significantly better than the previous. In the previous queue the `DoubleGenerator`, which will produce faster then the sort stages can consume, was forced to stop and wait for the queue to become non-full. In this queue the `DoubleGenerator` is never forced to wait which I suspect to be the reason for the vast speedup.

Question 8

Part 8.1

Listing 8.1: Implementation of the `UnboundedDoubleQueue` blocking queue

```
class NoLockNDoubleQueue implements BlockingDoubleQueue{

    private final double[] arr = new double[50];
    private volatile int head = 0, tail = 0;

    public void put(double item){
        while(tail - head == arr.length){} //Spin
        arr[tail % arr.length] = item;
        tail++;
    }

    public double take() {
        while(tail - head == 0){} //Spin
        double item = arr[head % arr.length];
        head++;
        return item;
    }
}
```

Part 8.2

I use volatile and final in two lines. Firstly the double array used for storing elements in the queue is final. This is due to the fact that 1) it will never change, and 2) the final keyword ensures visibility. The volatile keyword is used when declaring `head` and `tail`. This is to ensure visibility.

Part 8.3

`wait` and `notify` can only be used within synchronized blocks guarded by the same lock; and for good reason. Consider this simple example [1].

```
public void produce(double item){
    queue.put(item);
    notify();
}

public double consume(){
    while(queue.isEmpty())
        wait();
    return queue.take();
}
```

For this example it might be the case that:

1. Thread A calls consume. The consumer goes into the while loop because the buffer is empty.
2. Before Thread A calls `wait`, Thread B executes produce and calls `notify`.

[1] Inspired by `http://stackoverflow.com/questions/2779484/why-must-wait-always-be-in-synchronized-block`

3. Now Thread A calls `wait`, however, it might happen that `notify` is never called again because
   (a) B finished, and A therefore stays asleep, even though the queue is no longer empty.
   (b) B is waiting for A thus causing a deadlock.

Part 8.4

The construct of this queue is such that only a single thread will call `put`, and another will call `take`. These are the only two threads calling the methods. Because both methods depend on knowing both `head` and `tail` to calculate if the queue is full or empty, visibility is required. The two pointers is only written from one thread each; Namely, `head` is written to by the thread calling `take`, while tail is written to by the thread calling `put`. Therefore visibility is strong enough to ensure threadsafe access to the two pointers.

Concurrent access to the same index of the double array can never happen. This is due to the fact that if $tail - head == 0$, then they will point to the same index, but only `put` can access the index due to the spin loop in `take`. In the same manor, if $tail - head == arr.length$, then they will point to the same index, but only `take` can access the index due to the spin loop in `put`.

Visibility of the elements put into and removed from the double array is guaranteed because read and writing to volatile fields have the same guarantee as locking and unlocking, namely that everything A did in or prior to a read/write of a volatile field is visible to B when performing a read/write to the same field.

Therefore the queue is threadsafe, but of course only under the constraint that only one thread calls `take` and another only calls `put`. If more than these two thread call the queue, then it is not threadsafe.

Part 8.5

I tried three different scenarios

1. Removed volatile from `head`
2. Removed volatile from `tail`
3. Removed volatile from both

The first one was to remove volatile from `head`. This resulted in following output:

```
mac610262:src jbec$ java SortingPipeline
...
Elements out of order: 2062.10 before 2062.10
Elements out of order: 3461.10 before 3461.10
Elements out of order: 10689.1 before 10689.1
Elements out of order: 16950.1 before 16950.1
Elements out of order: 19756.1 before 19756.1
Elements out of order: 20723.1 before 20723.1
Elements out of order: 22962.1 before 22962.1
Elements out of order: 33233.1 before 33233.1
Elements out of order: 38267.1 before 38267.1
Elements out of order: 38380.1 before 38380.1
Elements out of order: 55126.1 before 55126.1
Elements out of order: 79719.1 before 79719.1
...
```

It is clear to see that some elements was emitted more than once as the input array contains no duplicates. What likely happens is that a `head` increment is not visible to the other thread. Thus the `put` method is supposed to spin when the array is full, it might just happen that it override another element with an element already added once before.

The exact same behavior is observed when the `tail` is not volatile. This might might be due to the fact that if the thread calling `take` does not see the `tail` increment, then it might very well be that the method returns an item even though the queue is empty, which would be an item it had already returned, although it was in fact suppose to spin.

Lastly, removing volatile from both fields make the code go into an instant deadlock probably because both threads is spinning waiting for each other.

Part 8.6

This is the results of running the current progress on the pipeline with $P = 4$ (after restoring the volatile fields).

```
# OS:   Mac OS X; 10.11; x86_64
# JVM:  Oracle Corporation; 1.8.0_60
# CPU:  null; 8 "cores"
# Date: 2016-01-11T18:56:06+0100
Sorting pipe    43.9 ms     1.52    8
```

Part 8.7

The results with $P = 2$.

```
OS:   Mac OS X; 10.11; x86_64
JVM:  Oracle Corporation; 1.8.0_60
CPU:  null; 8 "cores"
Date: 2016-01-11T20:49:19+0100
rting pipe    34.1 ms     0.48    8
```

The results with $P = 8$

```
# OS:   Mac OS X; 10.11; x86_64
# JVM:  Oracle Corporation; 1.8.0_60
# CPU:  null; 8 "cores"
# Date: 2016-01-11T20:56:08+0100
Sorting pipe    5855.0 ms   151.62  2
```

While the execution with $P = 2$ performed better than $P = 4$, the execution with $P = 8$ performed very poorly.

First of all, the bad performance of $P = 8$ is due to the fact that we now have more stages than cores in my machine. Because every stage is not running in parallel, and the stages are being scheduled in and out, the stages end up spend most of their time at halt because they quickly consumed what was left in the queue, or fill up the outgoing queue, and then spend the remaining time waiting before getting descheduled. This is of course very inefficient.

The fact that $P = 2$ performs better than $P = 4$ I think is due to the fact that there is less overhead to the computations. Passing the numbers from queue to queue and all the time reorganizing the local heap is a costly affair. Running with $P = 1$ was sightly slower than $P = 2$ with $39ms$.

To confirm the hypothesis about $P = 8$ I executed the code to get the results for $P = 6$. I'm not expecting any drastic performance decrease because $6 + 2$ is the number of cores in my computer.

```
# OS:   Mac OS X; 10.11; x86_64
# JVM:  Oracle Corporation; 1.8.0_60
# CPU:  null; 8 "cores"
```

```
# Date: 2016-01-11T20:54:58+0100
Sorting pipe    47.2 ms     1.54    8
```

The results shows that $P = 6$ is almost as fast as $P = 4$.

Question 9

Part 9.1

This is the exact version as given in exercise 12 except for the one `return null` subtituted with a `continue` in the take method.

Listing 9.1: Implementation of the `MSUnboundedDoubleQueue`

```java
class MSUnboundedDoubleQueue implements BlockingDoubleQueue{
    private final AtomicReference<Node> head, tail;
    public MSUnboundedDoubleQueue(){
        Node sentinal = new Node(0,null);
        head = new AtomicReference<Node>(sentinal);
        tail = new AtomicReference<Node>(sentinal);
    }
    public void put(double item){
        Node node = new Node(item,null);
        while(true){
            Node last = tail.get(),
                 next = last.next.get();
            if(last == tail.get()){
                if(next == null){
                    if(last.next.compareAndSet(next,node)){
                        tail.compareAndSet(last,node);
                        return;
                    }
                } else
                    tail.compareAndSet(last,next);
            }
        }
    }
    public double take(){
        while(true){
            Node first = head.get(),
                 last = tail.get(),
                 next = first.next.get();
            if(first == head.get()){
                if(first == last){
                    if(next == null)
                        continue;
                     else
                        tail.compareAndSet(last,next);
                } else {
                    double result = next.value;
                    if(head.compareAndSet(first,next))
                        return result;
                }
            }
        }
    }
    class Node{
        public final AtomicReference<Node> next;
        public final double value;

        public Node(double value, Node next){
            this.next = new AtomicReference<>(next);
            this.value = value;
        }
    }
}
```


Jacob B. Cholewa jbec@itu.dk

Part 9.2

This code is identical to the original `MSQueue` except for a `return null` substituted for a `continue` in `take`. I argue that this does not break the correctness of the algorithm (as proved in [2]) as it is the same as making two consecutive `take` calls.

This small change makes the queue blocking for consumers if there is no more elements to consume.

Part 9.3

```
# OS:   Mac OS X; 10.11; x86_64
# JVM:  Oracle Corporation; 1.8.0_60
# CPU:  null; 8 "cores"
# Date: 2016-01-11T22:41:40+0100
Sorting pipe    71.8 ms     2.49    4
```

This queue version performs worse than the `NoLockNDoubleQueue` but better than the so far presented queues. I had expected this queue to be faster than the previous as this is unbounded. This was however not the case. The simplicity of `NoLockNDoubleQueue` compared to the `MSUnboundedDoubleQueue` might explain why it performs better.

Question 10

Part 10.1

The code below shows an implementation of a Bounded blocking queue using transactional memory.

Listing 10.1: Implementation of the `StmBlockNDoubleQueue`

```
class StmBlockingNDoubleQueue implements BlockingDoubleQueue{
    private final TxnDouble[] arr;
    private final TxnInteger head, tail;

    public StmBlockingNDoubleQueue(){
        arr = new TxnDouble[40];
        for(int i = 0; i < arr.length; i++){
            arr[i] = newTxnDouble(0);
        }
        head = newTxnInteger(0);
        tail = newTxnInteger(0);
    }

    public void put(double item){
        atomic(() -> {
            if(tail.get() - head.get() == arr.length){
                retry();
            } else {
                arr[tail.get() % arr.length].set(item);
                tail.increment();
            }
        });
    }
    public double take(){
        return atomic(() -> {
            if(tail.get() - head.get() == 0) {
                retry();
            } else {
                double item = arr[head.get() % arr.length].get();
                head.increment();
                return item;
            }
            //Needed to compile. Will never be called
            throw new RuntimeException();
        });
    }
}
```

Part 10.2

This queue uses transactional memory instead of locking. It is an optimistic approach to concurrency meaning that the system will try to make the desired atomic operation. If it was successful the changes is committed, otherwise the changes are abandoned. It works by recording the state of the 'universe'. It then performs the desired operation on the recorded state. If none of the variables was access during the operation then the changes are committed, otherwise they are abandoned and the operation is retried. This makes it especially important that the operation has no side effect, like printing to StdOut, as the operation might run multiple times and therefore write to the output multiple times.

I will argue for the correctness of the code above because all operations is done within atomic transactional blocks, and because this is the only way of altering internal state. Either they succeed because no concurrent write happened in the meantime, or they are retried until successful.

Part 10.3

```
# OS:   Mac OS X; 10.11; x86_64
# JVM:  Oracle Corporation; 1.8.0_60
# CPU:  null; 8 "cores"
# Date: 2016-01-11T23:18:31+0100
Sorting pipe                          387.5 ms      32.12          2
```

This queue implementation is the slowest so far except for the fully synchronized version (`BlockingNDoubleQueue`). While transactional memory is threadsafe, it is not fast in all cases. In cases with very heavy access to few variables, which is the case here where all methods uses `tail` and `head`, a lot of retries might occur because one thread continues rapid commits continuously invalidates another threads tries on slower operations.

Jacob B. Cholewa jbec@itu.dk

Question 11

Part 11.1

The `AkkaSortingPipeline` is composed of four classes, the main class `AkkaSortingPipeline`, the two actors `SortingActor` and `EchoActor` along with two message type `InitMessage` and `DoubleMessage`.

The main logic lies with the `SortingActor` shown in listing 11.1. This actor can receive two kinds of messages; An `InitMessage` telling the actor to initialize a heap with a given capacity along with the reference to the actor that it should forward too. The `DoubleMessage` is used to pass doubles between actors. When a sorter receives a `DoubleMessage` it either 1) puts it into the heap if the heap is not full, 2) forwards the double if the value is less than the lowest value in its heap, 3) substitutes the lowest value in the heap with the just received value. It forwards the lowest value and then reorganizes it's heap. This is the exact same logic as the `SortingPipleline` from previous questions.

The second actor is the `EchoActor` is the drain shown in listing 11.2. When it receives a `DoubleMessage` it simply writes the value to the standard output.

The last important class is the start class shown in listing 11.3. It first sets up a pipeline between itself, the sorters and the drain before starting to emit the double values. When all double values is sent it flushes the pipeline so that echo prints the sorted list.

Listing 11.1: Sorting actor of `AkkeSortingPipeline`

```java
class SorterActor extends UntypedActor{
    private double[] heap;
    private int heapSize = 0;
    private ActorRef out;
    public void onReceive(Object o) throws Exception{
        if(o instanceof InitMessage){
            InitMessage msg = (InitMessage) o;
            heap = new double[msg.capacity];
            heapSize = 0;
            out = msg.to;
        }else if(o instanceof DoubleMessage){
            if(heap == null) return;
            DoubleMessage msg = (DoubleMessage) o;

            if(heapSize < heap.length){
                heap[heapSize++] = msg.value;
                DoubleArray.minheapSiftup(heap, heapSize-1, heapSize-1);
            } else if (msg.value <= heap[0]){
                out.tell(msg, ActorRef.noSender());
            } else {
                double least = heap[0];
                heap[0] = msg.value;
                DoubleArray.minheapSiftdown(heap,0,heapSize-1);
                out.tell(new DoubleMessage(least), ActorRef.noSender());
            }
        }
    }
}
```

Listing 11.2: Messages and Echo actor of `AkkeSortingPipeline`

```java
class EchoActor extends UntypedActor{
```


Jacob B. Cholewa jbec@itu.dk

```java
    public void onReceive(Object o) throws Exception{
        if(o instanceof DoubleMessage){
            DoubleMessage msg = (DoubleMessage) o;
            System.out.print(msg.value + ", ");
        }
    }
}

@SuppressWarnings("serial")
class DoubleMessage implements Serializable{
    public final double value;
    public DoubleMessage(double value){
        this.value = value;
    }
}

@SuppressWarnings("serial")
class InitMessage implements Serializable{
    public final ActorRef to;
    public final int capacity;
    public InitMessage(ActorRef to, int capacity){
        this.to = to;
        this.capacity = capacity;
    }
}
```

Listing 11.3: Starting method of `AkkeSortingPipeline`

```java
public class AkkaSortingPipeline{
    public static void main(String[] args){
        final ActorSystem system = ActorSystem.create("AkkaSortingPipeline"
            );

        int P = 4; //Number of sorters
        int N = 100; //Number of elements to sort
        final double[] arr = DoubleArray.randomPermutation(N);

        //Initializing drain
        final ActorRef drain = system.actorOf(Props.create(EchoActor.class)
            , "Ekko");

        //Initializing sorters
        ActorRef[] sorters = new ActorRef[P];
        for(int i = 0; i < P; i++){
            sorters[i] = system.actorOf(Props.create(SorterActor.class),"
                Sorter"+(i+1));
        }

        //Setting up chain
        ActorRef prev = drain;
        for(int i = 0; i < P; i++){
            sorters[i].tell(new InitMessage(prev, N/P), ActorRef.noSender()
                );
            prev = sorters[i];
        }

        //Sending all elements in the double array
        for(int i = 0; i < arr.length; i++){
            prev.tell(new DoubleMessage(arr[i]),ActorRef.noSender());
        }

        //Flushing the pipe
        for(int i = 0; i < arr.length; i++){
            prev.tell(new DoubleMessage(Double.POSITIVE_INFINITY),ActorRef.
                noSender());
        }
```

```
            //Wait for enter before terminating
            try {
                System.out.println("Press return to terminate...");
                System.in.read();
            } catch(IOException e) {
                e.printStackTrace();
            } finally {
                system.shutdown();
            }
        }
    }
```

Part 11.2

The result produced is as expected as all the elements occur in ordered sequence

```
mac610262:src jbec$ java -cp scala.jar:akka-actor.jar:akka-config.jar:.
    AkkaSortingPipeline
Press return to terminate...
0.1, 1.1, 2.1, 3.1, 4.1, 5.1, 6.1, 7.1, 8.1, 9.1, 10.1, 11.1, 12.1, 13.1,
    14.1, 15.1, 16.1, 17.1, 18.1, 19.1, 20.1, 21.1, 22.1, 23.1, 24.1, 25.1,
     26.1, 27.1, 28.1, 29.1, 30.1, 31.1, 32.1, 33.1, 34.1, 35.1, 36.1,
    37.1, 38.1, 39.1, 40.1, 41.1, 42.1, 43.1, 44.1, 45.1, 46.1, 47.1, 48.1,
     49.1, 50.1, 51.1, 52.1, 53.1, 54.1, 55.1, 56.1, 57.1, 58.1, 59.1,
    60.1, 61.1, 62.1, 63.1, 64.1, 65.1, 66.1, 67.1, 68.1, 69.1, 70.1, 71.1,
     72.1, 73.1, 74.1, 75.1, 76.1, 77.1, 78.1, 79.1, 80.1, 81.1, 82.1,
    83.1, 84.1, 85.1, 86.1, 87.1, 88.1, 89.1, 90.1, 91.1, 92.1, 93.1, 94.1,
     95.1, 96.1, 97.1, 98.1, 99.1,
```

Question 12

Part 12.1

The implemented code is devided into two parts. A `StreamSorter` class shown in listing 12.1, and the initialization class shown in listing 12.2

Listing 12.1: `StreamSorter` class

```
class StreamSorter{
    private double[] heap;
    private int heapSize = 0;

    public StreamSorter(int capacity){
        heap = new double[capacity];
    }

    public DoubleStream pipe(double x){
        if(heapSize < heap.length){
            heap[heapSize++] = x;
            DoubleArray.minheapSiftup(heap, heapSize-1, heapSize-1);
            return DoubleStream.empty();
        } else if (x <= heap[0]){
            return DoubleStream.of(x);
        } else {
            double least = heap[0];
            heap[0] = x;
            DoubleArray.minheapSiftdown(heap,0,heapSize-1);
            return DoubleStream.of(least);
        }
    }
}
```

Listing 12.2: Starting method for the `StreamSorter`

```
public static void main(String[] args) {
    final int count = 60, P = 3;
    final double[] arr = DoubleArray.randomPermutation(count);

    //Combining the array stream with an infinit stream of infinity
    DoubleStream input = DoubleStream.concat(DoubleStream.of(arr),
        DoubleStream.iterate(0, x -> Double.POSITIVE_INFINITY));

    //Chaning the StreamSorters
    for(int i = 0; i < P; i++){
        input = input.flatMap(new StreamSorter(count/P)::pipe);
    }

    //Chaning the output writer
    input.limit(count).forEach(x -> System.out.print(x + ", "));
}
```

Running the code yielded the following output

```
mac610262:src jbec$ java SortingPipeline
0.0, 0.1, 1.1, 2.1, 3.1, 4.1, 5.1, 6.1, 7.1, 8.1, 9.1, 10.1, 11.1, 12.1,
    13.1, 14.1, 15.1, 16.1, 17.1, 18.1, 19.1, 20.1, 21.1, 22.1, 23.1, 24.1,
     25.1, 26.1, 27.1, 28.1, 29.1, 30.1, 31.1, 32.1, 33.1, 34.1, 35.1,
    36.1, 37.1, 38.1, 39.1, 40.1, 41.1, 42.1, 43.1, 44.1, 45.1, 46.1, 47.1,
     48.1, 49.1, 50.1, 51.1, 52.1, 53.1, 54.1, 55.1, 56.1, 57.1, 58.1,
```

Part 12.2

The output is not at all sorted, and half of the time the program crashes with an `ArrayIndexOutOfBound` exception. Functions used with parallel streams must be stateless, but because the `StreamSorter` is stateful, it is causing the program to malfunction.

Bibliography

[1] Brian Goetz and Tim Peierls. *Java concurrency in practice.* Pearson Education, 2006.

[2] Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275. ACM, 1996.

Appendix A SortingPipeline.java

Listing A.1: Source code for SortingPipeline.java

```java
// Pipelined sorting using P>=1 stages, each maintaining an internal
// collection of size S>=1.  Stage 1 contains the largest items, stage
// 2 the second largest, ..., stage P the smallest ones.  In each
// stage, the internal collection of items is organized as a minheap.
// When a stage receives an item x and its collection is not full, it
// inserts it in the heap.  If the collection is full and x is less
// than or equal to the collections's least item, it forwards the item
// to the next stage; otherwise forwards the collection's least item
// and inserts x into the collection instead.

// When there are itemCount items and stageCount stages, each stage
// must be able to hold at least ceil(itemCount/stageCount) items,
// which equals (itemCount-1)/stageCount+1.

// sestoft@itu.dk * 2016-01-10

import java.util.stream.DoubleStream;
import java.util.function.DoubleFunction;
import java.util.function.Function;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.function.IntToDoubleFunction;
import java.util.concurrent.atomic.AtomicReference;
import java.util.concurrent.atomic.AtomicInteger;

//Multiverse import
import org.multiverse.api.references.*;
import static org.multiverse.api.StmUtils.*;

// Multiverse locking:
import org.multiverse.api.LockMode;
import org.multiverse.api.Txn;
import org.multiverse.api.callables.TxnVoidCallable;

public class SortingPipeline {
    public static void main(String[] args) {
        final int count = 100_000, P = 4;
        final double[] arr = DoubleArray.randomPermutation(count);

        //Combining the array stream with an infinit stream of infinity
        DoubleStream input = DoubleStream.concat(DoubleStream.of(arr),
            DoubleStream.iterate(0, x -> Double.POSITIVE_INFINITY));//.
            parallel();

        //Chaning the StreamSorters
        for(int i = 0; i < P; i++){
            input = input.flatMap(new StreamSorter(count/P)::pipe);
        }

        //Chaning the output writer
        input.limit(500).forEach(x -> System.out.print(x + ", "));
    }

    /*public static void main(String[] args) {
        SystemInfo();
        Mark7("Sorting pipe", j -> {
```

```java
            final int count = 60, P = 4;
            final double[] arr = DoubleArray.randomPermutation(count);

            final BlockingDoubleQueue[] queues = new BlockingDoubleQueue[P
                +1];

            for(int i = 0; i < P+1; i++){
                //queues[i] = new BlockingNDoubleQueue();
                //queues[i] = new UnboundedDoubleQueue();
                //queues[i] = new NoLockNDoubleQueue();
                //queues[i] = new MSUnboundedDoubleQueue();
                queues[i] = new StmBlockingNDoubleQueue();
            }

            sortPipeline(arr, P, queues);
            return arr[0];
        });
    }*/

    private static void sortPipeline(double[] arr, int P,
        BlockingDoubleQueue[] queues) {
        int n = arr.length / P;

        //Initializing the sorting stages
        Thread[] threads = new Thread[P+2];
        for(int i = 1; i <= P; i++){
            threads[i-1] = new Thread(new SortingStage(queues[i-1], queues[
                i], n, arr.length+(P-i)*n));
        }

        //Initializing the drain
        threads[P] = new Thread(new SortedChecker(arr.length, queues[P]));

        //Initializing the source. The source is purposefully last in the
            array so that it will be started lastly.
        threads[P+1] = new Thread(new DoubleGenerator(arr, arr.length,
            queues[0]));

        //Starting the stages
        for(int i = 0; i < threads.length; i++){
            threads[i].start();
        }

        //Joining the stages
        for(int i = 0; i < threads.length; i++){
            try{ threads[i].join(); }
            catch(InterruptedException e){ throw new RuntimeException(e); }
        }
    }

    static class SortingStage implements Runnable {
        private final BlockingDoubleQueue in;
        private final BlockingDoubleQueue out;
        private final double[] heap;
        private int itemCount;
        private int heapSize = 0;

        public SortingStage(BlockingDoubleQueue in, BlockingDoubleQueue out
            , int capacity, int itemCount){
            this.in = in;
            this.out = out;
            this.itemCount = itemCount;
            heap = new double[capacity];
        }


        public void run() {
```

```java
            while(itemCount > 0){
                double x = in.take();
                if(heapSize < heap.length){
                    heap[heapSize++] = x;
                    DoubleArray.minheapSiftup(heap, heapSize-1, heapSize-1)
                        ;
                } else if (x <= heap[0]){
                    out.put(x);
                    itemCount--;
                } else {
                    double least = heap[0];
                    heap[0] = x;
                    DoubleArray.minheapSiftdown(heap,0,heapSize-1);
                    out.put(least);
                    itemCount--;
                }
            }
        }
    }

    static class DoubleGenerator implements Runnable {
        private final BlockingDoubleQueue output;
        private final double[] arr;  // The numbers to feed to output
        private final int infinites;

        public DoubleGenerator(double[] arr, int infinites,
            BlockingDoubleQueue output) {
            this.arr = arr;
            this.output = output;
            this.infinites = infinites;
        }

        public void run() {
            for (int i=0; i<arr.length; i++)  // The numbers to sort
                output.put(arr[i]);
            for (int i=0; i<infinites; i++)   // Infinite numbers for wash-
                out
                output.put(Double.POSITIVE_INFINITY);
        }
    }

    static class SortedChecker implements Runnable {
        // If DEBUG is true, print the first 100 numbers received
        private final static boolean DEBUG = false;
        private final BlockingDoubleQueue input;
        private final int itemCount; // the number of items to check

        public SortedChecker(int itemCount, BlockingDoubleQueue input) {
            this.itemCount = itemCount;
            this.input = input;
        }

        public void run() {
            int consumed = 0;
            double last = Double.NEGATIVE_INFINITY;
            while (consumed++ < itemCount) {
                double p = input.take();
                if (DEBUG && consumed <= 100)
                    System.out.print(p + " ");
                if (p <= last)
                    System.out.printf("Elements out of order: %g before %g%
                        n", last, p);
                last = p;
            }
            if (DEBUG)
                System.out.println();
        }
```

```java
    }

    // --- Benchmarking infrastructure ---

    // NB: Modified to show milliseconds instead of nanoseconds

    public static double Mark7(String msg, IntToDoubleFunction f) {
        int n = 10, count = 1, totalCount = 0;
        double dummy = 0.0, runningTime = 0.0, st = 0.0, sst = 0.0;
        do {
            count *= 2;
            st = sst = 0.0;
            for (int j=0; j<n; j++) {
                Timer t = new Timer();
                for (int i=0; i<count; i++)
                    dummy += f.applyAsDouble(i);
                runningTime = t.check();
                double time = runningTime * 1e3 / count;
                st += time;
                sst += time * time;
                totalCount += count;
            }
        } while (runningTime < 0.25 && count < Integer.MAX_VALUE/2);
        double mean = st/n, sdev = Math.sqrt((sst - mean*mean*n)/(n-1));
        System.out.printf("%-25s %15.1f ms %10.2f %10d%n", msg, mean, sdev,
             count);
        return dummy / totalCount;
    }

    public static void SystemInfo() {
        System.out.printf("# OS:   %s; %s; %s%n",
                System.getProperty("os.name"),
                System.getProperty("os.version"),
                System.getProperty("os.arch"));
        System.out.printf("# JVM:  %s; %s%n",
                System.getProperty("java.vendor"),
                System.getProperty("java.version"));
        // The processor identifier works only on MS Windows:
        System.out.printf("# CPU:  %s; %d \"cores\"%n",
                System.getenv("PROCESSOR_IDENTIFIER"),
                Runtime.getRuntime().availableProcessors());
        java.util.Date now = new java.util.Date();
        System.out.printf("# Date: %s%n",
                new java.text.SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ssZ").
                    format(now));
    }

    // Crude wall clock timing utility, measuring time in seconds

    static class Timer {
        private long start, spent = 0;
        public Timer() { play(); }
        public double check() { return (System.nanoTime()-start+spent)/1e9;
             }
        public void pause() { spent += System.nanoTime()-start; }
        public void play() { start = System.nanoTime(); }
    }
}

// ----------------------------------------------------------------------

// Queue interface

interface BlockingDoubleQueue {
    double take();
    void put(double item);
}
```

```java

class WrappedArrayDoubleQueue implements BlockingDoubleQueue{

    private final ArrayBlockingQueue<Double> queue;

    public WrappedArrayDoubleQueue(){
        this.queue = new ArrayBlockingQueue<Double>(50);
    }

    public WrappedArrayDoubleQueue(int capacity){
        this.queue = new ArrayBlockingQueue<Double>(capacity);
    }

    public void put(double item){
        try{ queue.put(item); }
        catch(InterruptedException e){ throw new RuntimeException(e); }
    }

    public double take(){
        try{ return queue.take(); }
        catch(InterruptedException e){ throw new RuntimeException(e); }
    }
}


class BlockingNDoubleQueue implements BlockingDoubleQueue{

    private final double[] arr = new double[50];
    private int head = 0, tail = 0, count = 0;

    public synchronized void put(double item){
        while(count == arr.length){
            try{ this.wait(); }
            catch(InterruptedException exn) { }
        }

        arr[tail] = item;
        tail = ++tail == arr.length ? 0 : tail;
        count++;
        this.notify();
    }

    public synchronized double take(){
        while(count == 0){
            try{ this.wait(); }
            catch(InterruptedException exn) { }
        }

        double item = arr[head];
        head = ++head == arr.length ? 0 : head;
        count --;
        this.notify();
        return item;
    }
}

class UnboundedDoubleQueue implements BlockingDoubleQueue{

    public Node head;
    public Node tail;

    public UnboundedDoubleQueue(){
        Node n = new Node(0,null);
        tail = head = n;
    }

    public synchronized void put(double item){
```

```java
        tail.next = new Node(item,null); //Setting next
        tail = tail.next; //Moving tail

        this.notify(); //Notifying a thread waiting for elements
    }

    public synchronized double take(){
        while(head.next == null){
            try{ this.wait(); }
            catch(InterruptedException exn) { }
        }

        Node first = head;
        head = first.next;
        return head.value;
    }

    class Node{
        public Node next;
        public final double value;

        public Node(double value, Node next){
            this.next = next;
            this.value = value;
        }
    }
}

class NoLockNDoubleQueue implements BlockingDoubleQueue{

    private final double[] arr = new double[50];
    private volatile int head = 0, tail = 0;

    public void put(double item){
        while(tail - head == arr.length){} //Spin
        arr[tail % arr.length] = item;
        tail++;
    }

    public double take() {
        while(tail - head == 0){} //Spin
        double item = arr[head % arr.length];
        head++;
        return item;
    }
}

class MSUnboundedDoubleQueue implements BlockingDoubleQueue{

    private final AtomicReference<Node> head, tail;


    public MSUnboundedDoubleQueue(){
        Node sentinal = new Node(0,null);
        head = new AtomicReference<Node>(sentinal);
        tail = new AtomicReference<Node>(sentinal);
    }

    public void put(double item){
        Node node = new Node(item,null);
        while(true){
            Node last = tail.get(),
                 next = last.next.get();
            if(last == tail.get()){
                if(next == null){
                    if(last.next.compareAndSet(next,node)){
                        tail.compareAndSet(last,node);
```

```java
                        return;
                    }
                } else {
                    tail.compareAndSet(last,next);
                }
            }
        }
    }
    public double take(){
        while(true){
            Node first = head.get(),
                 last = tail.get(),
                 next = first.next.get();
            if(first == head.get()){
                if(first == last){
                    if(next == null){
                        continue;
                    } else {
                        tail.compareAndSet(last,next);
                    }
                } else {
                    double result = next.value;
                    if(head.compareAndSet(first,next)){
                        return result;
                    }
                }
            }
        }
    }

    class Node{
        public final AtomicReference<Node> next;
        public final double value;

        public Node(double value, Node next){
            this.next = new AtomicReference<>(next);
            this.value = value;
        }
    }
}


class StmBlockingNDoubleQueue implements BlockingDoubleQueue{
    private final TxnDouble[] arr;
    private final TxnInteger head, tail;

    public StmBlockingNDoubleQueue(){
        arr = new TxnDouble[40];
        for(int i = 0; i < arr.length; i++){
            arr[i] = newTxnDouble(0);
        }
        head = newTxnInteger(0);
        tail = newTxnInteger(0);
    }

    public void put(double item){
        atomic(() -> {
            if(tail.get() - head.get() == arr.length){
                retry();
            } else {
                arr[tail.get() % arr.length].set(item);
                tail.increment();
            }
        });
    }
    public double take(){
        return atomic(() -> {
```

```java
            if(tail.get() - head.get() == 0) {
                retry();
            } else {
                double item = arr[head.get() % arr.length].get();
                head.increment();
                return item;
            }
            //Needed to compile. Will never be called
            throw new RuntimeException();
        });
    }
}

// ----------------------------------------------------------------------


class StreamSorter{

    private double[] heap;
    private int heapSize = 0;

    public StreamSorter(int capacity){
        heap = new double[capacity];
    }

    public DoubleStream pipe(double x){
        if(heapSize < heap.length){
            heap[heapSize++] = x;
            DoubleArray.minheapSiftup(heap, heapSize-1, heapSize-1);
            return DoubleStream.empty();
        } else if (x <= heap[0]){
            return DoubleStream.of(x);
        } else {
            double least = heap[0];
            heap[0] = x;
            DoubleArray.minheapSiftdown(heap,0,heapSize-1);
            return DoubleStream.of(least);
        }
    }
}

// ----------------------------------------------------------------------

class DoubleArray {
    public static double[] randomPermutation(int n) {
        double[] arr = fillDoubleArray(n);
        shuffle(arr);
        return arr;
    }

    private static double[] fillDoubleArray(int n) {
        double[] arr = new double[n];
        for (int i = 0; i < n; i++)
            arr[i] = i + 0.1;
        return arr;
    }

    private static final java.util.Random rnd = new java.util.Random();

    private static void shuffle(double[] arr) {
        for (int i = arr.length-1; i > 0; i--)
            swap(arr, i, rnd.nextInt(i+1));
    }

    // Swap arr[s] and arr[t]
    private static void swap(double[] arr, int s, int t) {
        double tmp = arr[s]; arr[s] = arr[t]; arr[t] = tmp;
```

```java
    }

    // Minheap operations for parallel sort pipelines.
    // Minheap invariant:
    // If heap[0..k-1] is a minheap, then heap[(i-1)/2] <= heap[i] for
    // all indexes i=1..k-1.  Thus heap[0] is the smallest element.

    // Although stored in an array, the heap can be considered a tree
    // where each element heap[i] is a node and heap[(i-1)/2] is its
    // parent. Then heap[0] is the tree's root and a node heap[i] has
    // children heap[2*i+1] and heap[2*i+2] if these are in the heap.

    // In heap[0..k], move node heap[i] downwards by swapping it with
    // its smallest child until the heap invariant is reestablished.

    public static void minheapSiftdown(double[] heap, int i, int k) {
        int child = 2 * i + 1;
        if (child <= k) {
            if (child+1 <= k && heap[child] > heap[child+1])
                child++;
            if (heap[i] > heap[child]) {
                swap(heap, i, child);
                minheapSiftdown(heap, child, k);
            }
        }
    }

    // In heap[0..k], move node heap[i] upwards by swapping with its
    // parent until the heap invariant is reestablished.
    public static void minheapSiftup(double[] heap, int i, int k) {
        if (0 < i) {
            int parent = (i - 1) / 2;
            if (heap[i] < heap[parent]) {
                swap(heap, i, parent);
                minheapSiftup(heap, parent, k);
            }
        }
    }
}
```


Jacob B. Cholewa jbec@itu.dk

Appendix B AkkaSortingPipeline.java

Listing B.1: Source code for AkkaSortingPipeline.java

```java
import java.io.*;
import akka.actor.*;

public class AkkaSortingPipeline{
    public static void main(String[] args){
        final ActorSystem system = ActorSystem.create("AkkaSortingPipeline");

        final ActorRef drain = system.actorOf(Props.create(EchoActor.class), "Ekko");

        int P = 4;
        int N = 100;
        ActorRef[] sorters = new ActorRef[P];

        for(int i = 0; i < P; i++){
            sorters[i] = system.actorOf(Props.create(SorterActor.class),"Sorter"+(i+1));
        }

        //Setting up chain
        ActorRef prev = drain;
        for(int i = 0; i < P; i++){
            sorters[i].tell(new InitMessage(prev, N/P), ActorRef.noSender());
            prev = sorters[i];
        }

        final double[] arr = DoubleArray.randomPermutation(N);

        for(int i = 0; i < arr.length; i++){
            prev.tell(new DoubleMessage(arr[i]),ActorRef.noSender());
        }

        for(int i = 0; i < arr.length; i++){
            prev.tell(new DoubleMessage(Double.POSITIVE_INFINITY),ActorRef.noSender());
        }

        try {
            System.out.println("Press return to terminate...");
            System.in.read();
        } catch(IOException e) {
            e.printStackTrace();
        } finally {
            system.shutdown();
        }
    }
}

class SorterActor extends UntypedActor{
    private double[] heap;
    private int heapSize = 0;
    private ActorRef out;
    public void onReceive(Object o) throws Exception{
```

```java
        if(o instanceof InitMessage){
            InitMessage msg = (InitMessage) o;
            heap = new double[msg.capacity];
            heapSize = 0;
            out = msg.to;
        }else if(o instanceof DoubleMessage){
            if(heap == null) return;
            DoubleMessage msg = (DoubleMessage) o;

            if(heapSize < heap.length){
                heap[heapSize++] = msg.value;
                DoubleArray.minheapSiftup(heap, heapSize-1, heapSize-1);
            } else if (msg.value <= heap[0]){
                out.tell(msg, ActorRef.noSender());
            } else {
                double least = heap[0];
                heap[0] = msg.value;
                DoubleArray.minheapSiftdown(heap,0,heapSize-1);
                out.tell(new DoubleMessage(least), ActorRef.noSender());
            }
        }
    }

}

class EchoActor extends UntypedActor{
    public void onReceive(Object o) throws Exception{
        if(o instanceof DoubleMessage){
            DoubleMessage msg = (DoubleMessage) o;
            System.out.print(msg.value + ", ");
        }
    }
}

@SuppressWarnings("serial")
class DoubleMessage implements Serializable{
    public final double value;
    public DoubleMessage(double value){
        this.value = value;
    }
}

@SuppressWarnings("serial")
class InitMessage implements Serializable{
    public final ActorRef to;
    public final int capacity;
    public InitMessage(ActorRef to, int capacity){
        this.to = to;
        this.capacity = capacity;
    }
}


class DoubleArray {
    public static double[] randomPermutation(int n) {
        double[] arr = fillDoubleArray(n);
        shuffle(arr);
        return arr;
    }

    private static double[] fillDoubleArray(int n) {
        double[] arr = new double[n];
        for (int i = 0; i < n; i++)
            arr[i] = i + 0.1;
        return arr;
    }

```

```java
    private static final java.util.Random rnd = new java.util.Random();

    private static void shuffle(double[] arr) {
        for (int i = arr.length-1; i > 0; i--)
            swap(arr, i, rnd.nextInt(i+1));
    }

    // Swap arr[s] and arr[t]
    private static void swap(double[] arr, int s, int t) {
        double tmp = arr[s]; arr[s] = arr[t]; arr[t] = tmp;
    }

    // Minheap operations for parallel sort pipelines.
    // Minheap invariant:
    // If heap[0..k-1] is a minheap, then heap[(i-1)/2] <= heap[i] for
    // all indexes i=1..k-1.  Thus heap[0] is the smallest element.

    // Although stored in an array, the heap can be considered a tree
    // where each element heap[i] is a node and heap[(i-1)/2] is its
    // parent. Then heap[0] is the tree's root and a node heap[i] has
    // children heap[2*i+1] and heap[2*i+2] if these are in the heap.

    // In heap[0..k], move node heap[i] downwards by swapping it with
    // its smallest child until the heap invariant is reestablished.

    public static void minheapSiftdown(double[] heap, int i, int k) {
        int child = 2 * i + 1;
        if (child <= k) {
            if (child+1 <= k && heap[child] > heap[child+1])
                child++;
            if (heap[i] > heap[child]) {
                swap(heap, i, child);
                minheapSiftdown(heap, child, k);
            }
        }
    }

    // In heap[0..k], move node heap[i] upwards by swapping with its
    // parent until the heap invariant is reestablished.
    public static void minheapSiftup(double[] heap, int i, int k) {
        if (0 < i) {
            int parent = (i - 1) / 2;
            if (heap[i] < heap[parent]) {
                swap(heap, i, parent);
                minheapSiftup(heap, parent, k);
            }
        }
    }
}
```

Appendix C TestLocking2.java

Listing C.1: Source code for TestLocking2.java

```java
import java.util.concurrent.atomic.AtomicInteger;
import java.util.HashSet;

public class TestLocking2 {
    public static void main(String[] args) {
        DoubleArrayList dal1 = new DoubleArrayList();
        dal1.add(42.1); dal1.add(7.2); dal1.add(9.3); dal1.add(13.4);
        dal1.set(2, 11.3);
        for (int i=0; i<dal1.size(); i++)
            System.out.println(dal1.get(i));
        DoubleArrayList dal2 = new DoubleArrayList();
        dal2.add(90.1); dal2.add(80.2); dal2.add(70.3); dal2.add(60.4);
            dal2.add(50.5);
        DoubleArrayList dal3 = new DoubleArrayList();
        System.out.printf("Total size = %d%n", DoubleArrayList.totalSize())
            ;
        System.out.printf("All lists  = %s%n", DoubleArrayList.allLists());
    }
}

// Expandable array list of doubles, also keeping track of all such
// array lists and their total element count.

class DoubleArrayList {
    private static AtomicInteger totalSize = new AtomicInteger();
    //private static int totalSize = 0;
    private static HashSet<DoubleArrayList> allLists = new HashSet<>();
    private static Object ListsLock = new Object();

    // Invariant: 0 <= size <= items.length
    private double[] items = new double[2];
    private int size = 0;

    public DoubleArrayList() {
        synchronized(ListsLock){
            allLists.add(this);
        }
    }

    // Number of items in the double list
    public int size() {
        return size;
    }

    // Return item number i, if any
    public double get(int i) {
        if (0 <= i && i < size)
            return items[i];
        else
            throw new IndexOutOfBoundsException(String.valueOf(i));
    }

    // Add item x to end of list
    public boolean add(double x) {
        if (size == items.length) {
```

```java
            double[] newItems = new double[items.length * 2];
            for (int i=0; i<items.length; i++)
                newItems[i] = items[i];
            items = newItems;
        }
        items[size] = x;
        size++;
        totalSize.incrementAndGet();
        return true;
    }

    // Replace item number i, if any, with x
    public double set(int i, double x) {
        if (0 <= i && i < size) {
            double old = items[i];
            items[i] = x;
            return old;
        } else
            throw new IndexOutOfBoundsException(String.valueOf(i));
    }

    // The double list formatted as eg "[3.2, 4.7]"
    public String toString() {
        StringBuilder sb = new StringBuilder("[");
        for (int i=0; i<size; i++)
            sb.append(i > 0 ? ", " : "").append(items[i]);
        return sb.append("]").toString();
    }

    public static int totalSize() {
        return totalSize.get();
    }

    public static HashSet<DoubleArrayList> allLists() {
        synchronized(ListsLock){
            return allLists;
        }
    }
}
```