/** A class that implements the ADT maxheap by using an array. */ public final class MaxHeap<T extends Comparable<? super T» implements MaxHeap-Interface{

```
private T[] heap; // Array of heap entries; ignore heap[0]
private int lastIndex; // Index of last entry and number of entries
private boolean integrityOK = false;
private static final int DEFAULT_CAPACITY = 25;
private static final int MAX_CAPACITY = 10000;

public MaxHeap(T[] entries){
    this(entries.length); // Call other constructor
    lastIndex = entries.length;
    // Assertion: integrityOK = true
    // Copy given array to data field
    for (int index = 0; index < entries.length; index++)
    heap[index + 1] = entries[index];
    // Create heap
    for (int rootIndex = lastIndex / 2; rootIndex > 0; rootIndex--)
    reheap(rootIndex);
} // end constructor


// Creates an empty heap whose initial capacity is 25.
public MaxHeap(int initialCapacity)    {
    // Is initialCapacity too small?
    if (initialCapacity < DEFAULT_CAPACITY)
    initialCapacity = DEFAULT_CAPACITY;
    else // Is initialCapacity too big?
    checkCapacity(initialCapacity);
    // The cast is safe because the new array contains null entries
    @SuppressWarnings("unchecked")
    T[] tempHeap = (T[])new Comparable[initialCapacity + 1];
    heap = tempHeap;
    lastIndex = 0;
    integrityOK = true;
} // end constructor

public MaxHeap(){
    this(DEFAULT_CAPACITY); // Call next constructor
} // end default constructor

public T getMax(){
    checkIntegrity();
    T root = null;
    if (!isEmpty())
```

```
        root = heap[1];
    return root;
} // end getMax

public boolean checkCapacity(int capacity){
    return capacity <= MAX_CAPACITY;
} // end checkCapacity

public void checkIntegrity(){
    if (!integrityOK)
        throw new SecurityException("Array object is corrupt.");
} // end checkIntegrity

public boolean isEmpty(){
    return lastIndex < 1;
} // end isEmpty

public int getSize(){
    return lastIndex;
} // end getSize

public void add(T newEntry){
    checkIntegrity(); // Ensure initialization of data fields
    int newIndex = lastIndex + 1;
    int parentIndex = newIndex / 2;
    while ( (parentIndex > 0) && newEntry.compareTo(heap[parentIndex]) > 0)
    {
    heap[newIndex] = heap[parentIndex];
    newIndex = parentIndex;
    parentIndex = newIndex / 2;
    } // end while
    heap[newIndex] = newEntry;
    lastIndex++;
    ensureCapacity();
} // end add

public T removeMax(){
    checkIntegrity(); // Ensure initialization of data fields
    T root = null;
    if (!isEmpty()){
        root = heap[1]; // Return value
        heap[1] = heap[lastIndex]; // Form a semiheap
        lastIndex--; // Decrease size
        reheap(1); // Transform to a heap
    } // end if
    return root;
```

```
} // end removeMax

private void reheap(int rootIndex){
    boolean done = false;
    T orphan = heap[rootIndex];
    int leftChildIndex = 2 * rootIndex;
    while (!done && (leftChildIndex <= lastIndex) ){
        int largerChildIndex = leftChildIndex; // Assume larger
        int rightChildIndex = leftChildIndex + 1;
        if ( (rightChildIndex <= lastIndex) &&
            heap[rightChildIndex].compareTo(heap[largerChildIndex]) > 0){
                largerChildIndex = rightChildIndex;
        } // end if
        if (orphan.compareTo(heap[largerChildIndex]) < 0){
            heap[rootIndex] = heap[largerChildIndex];
            rootIndex = largerChildIndex;
            leftChildIndex = 2 * rootIndex;
        }
        else
        done = true;
    } // end while
    heap[rootIndex] = orphan;
}

public void ensureCapacity(){
    if (lastIndex >= heap.length)
        checkCapacity(2 * heap.length); // Is capacity too small?
} // end ensureCapacity

public void remove(T anEntry){
    checkIntegrity();
    int index = 1;
    while (index <= lastIndex){
        if (heap[index].equals(anEntry)){
            heap[index] = heap[lastIndex];
            lastIndex--;
            reheap(index);
        } // end if
        else
        index++;
    } // end while
} // end remove

public String toString(){
    String result = "";
    for (int index = 1; index <= lastIndex; index++)
```

```
        result += heap[index] + " ";
    return result;
} // end toString

public T[] toArray(){
    checkIntegrity();
    // The cast is safe because the new array contains null entries
    @SuppressWarnings("unchecked")
    T[] result = (T[])new Comparable[lastIndex + 1];
    for (int index = 0; index < lastIndex; index++)
        result[index] = heap[index + 1];
    return result;
} // end toArray

public boolean contains(T anEntry){
    boolean found = false;
    int index = 1;
    while (!found && (index <= lastIndex)){
        if (anEntry.equals(heap[index]))
            found = true;
        index++;
    } // end while
    return found;
} // end contains

public int getIndexOf(T anEntry){
    int where = -1;
    boolean found = false;
    int index = 1;
    while (!found && (index <= lastIndex)){
        if (anEntry.equals(heap[index])){
            found = true;
            where = index;
        } // end if
        index++;
    } // end while
    return where;
} // end getIndexOf

public void display(){
    for (int index = 1; index <= lastIndex; index++)
        System.out.print(heap[index] + " ");
    System.out.println();
} // end display

public void swap(int firstIndex, int secondIndex){
```

```
    T temp = heap[firstIndex];
    heap[firstIndex] = heap[secondIndex];
    heap[secondIndex] = temp;
} // end swap

public void heapSort(){
    checkIntegrity();
    int numberOfSwaps = 0;
    for (int index = lastIndex; index > 1; index--){
        swap(1, index);
        numberOfSwaps++;
        reheap(1);
    } // end for
    System.out.println("Number of swaps: " + numberOfSwaps);
} // end heapSort

public void changePriority(int index, T newEntry){
    checkIntegrity();
    if ( (index >= 1) && (index <= lastIndex) ){
        T oldEntry = heap[index];
        heap[index] = newEntry;
        if (oldEntry.compareTo(newEntry) < 0)
            reheap(index);
        else
            reheap2(index, lastIndex);
    } // end if
} // end changePriority

public void heapSort2(){
    checkIntegrity();
    int numberOfSwaps = 0;
    for (int index = lastIndex; index > 1; index--){
        swap(1, index);
        numberOfSwaps++;
        reheap2(1, index - 1);
    } // end for
    System.out.println("Number of swaps: " + numberOfSwaps);
} // end heapSort2


private void reheap2(int rootIndex, int lastIndex){
    boolean done = false;
    T orphan = heap[rootIndex];
    int leftChildIndex = 2 * rootIndex;
    while (!done && (leftChildIndex <= lastIndex) ){
        int largerChildIndex = leftChildIndex; // Assume larger
```

```
        int rightChildIndex = leftChildIndex + 1;
        if ( (rightChildIndex <= lastIndex) &&
            heap[rightChildIndex].compareTo(heap[largerChildIndex]) > 0){
                largerChildIndex = rightChildIndex;
        } // end if
        if (orphan.compareTo(heap[largerChildIndex]) < 0){
            heap[rootIndex] = heap[largerChildIndex];
            rootIndex = largerChildIndex;
            leftChildIndex = 2 * rootIndex;
        }
        else
        done = true;
    } // end while
    heap[rootIndex] = orphan;
}

public void heapSort3(){
    checkIntegrity();
    int numberOfSwaps = 0;
    for (int index = lastIndex / 2; index > 0; index--){
        reheap3(index, lastIndex);
    } // end for
    for (int index = lastIndex; index > 1; index--){
        swap(1, index);
        numberOfSwaps++;
        reheap3(1, index - 1);
    } // end for
    System.out.println("Number of swaps: " + numberOfSwaps);
} // end heapSort3

private void reheap3(int rootIndex, int lastIndex){
    boolean done = false;
    T orphan = heap[rootIndex];
    int leftChildIndex = 2 * rootIndex;
    while (!done && (leftChildIndex <= lastIndex) ){
        int largerChildIndex = leftChildIndex; // Assume larger
        int rightChildIndex = leftChildIndex + 1;
        if ( (rightChildIndex <= lastIndex) &&
            heap[rightChildIndex].compareTo(heap[largerChildIndex]) > 0){
                largerChildIndex = rightChildIndex;
        } // end if
        if (orphan.compareTo(heap[largerChildIndex]) < 0){
            heap[rootIndex] = heap[largerChildIndex];
            rootIndex = largerChildIndex;
            leftChildIndex = 2 * rootIndex;
        }
```

```
        else
        done = true;
    } // end while
    heap[rootIndex] = orphan;
}

public void clear(){
    checkIntegrity();
    while (lastIndex > -1){
        heap[lastIndex] = null;
        lastIndex--;
    } // end while
    lastIndex = 0;
} // end clear
// Private methods
// . . .
```

} // end MaxHeap