**class java.lang.Object**

* boolean equals(Object other)
* String toString()
* int hashCode()

**interface java.lang.Comparable**

* int compareTo(Object other) // return value < 0 if this is less than other  
   // return value = 0 if this is equal to other  
   // return value > 0 if this is greater than other

**class java.lang.Integer implements java.lang.Comparable**

* Integer(int value) // constructor
* int intValue()
* boolean equals(Object other)
* String toString()
* int compareTo(Object other) // specified by java.lang.Comparable

**class java.lang.Double implements java.lang.Comparable**

* Double(double value) // constructor
* double doubleValue()
* boolean equals(Object other)
* String toString()
* int compareTo(Object other) // specified by java.lang.Comparable

**class java.lang.String implements java.lang.Comparable**

* int compareTo(Object other) // specified by java.lang.Comparable
* boolean equals(Object other)
* int length()
* String substring(int from, int to) // returns the substring beginning at from  
   // and ending at to-1
* String substring(int from) // returns substring(from, length())
* int indexOf(String s) // returns the index of the first occurrence of s;  
   // returns -1 if not found

**class java.lang.Math**

* static int abs(int x)
* static double abs(double x)
* static double pow(double base, double exponent)
* static double sqrt(double x)

**class java.util.Random**

* int nextInt(int n) // returns an integer in the range from 0 to n-1
* double nextDouble()

**interface java.util.List**

* boolean add(Object x)
* int size()
* Iterator iterator()
* ListIterator listIterator()

**class java.util.ArrayList implements java.util.List**

* Methods in addition to the List methods:
* Object get(int index)
* Object set(int index, Object x) // replaces the element at index with x
* void add(int index, Object x) // inserts x at position index, sliding elements   
   // at position index and higher to the right   
   // (adds 1 to their indices) and adjusts size
* Object remove(int index) // removes element from position index, sliding elements   
   // at position index + 1 and higher to the left   
   // (subtracts 1 from their indices) and adjusts size

**class java.util.LinkedList implements java.util.List**

* Methods in addition to the List methods:
* void addFirst(Object x)
* void addLast(Object x)
* Object getFirst()
* Object getLast()
* Object removeFirst()
* Object removeLast()

**interface java.util.Set**

* boolean add(Object x)
* boolean contains(Object x)
* boolean remove(Object x)
* int size()
* Iterator iterator()

**class java.util.HashSet implements java.util.Set**

**class java.util.TreeSet implements java.util.Set**

**interface java.util.Map**

* Object put(Object key, Object value)
* Object get(Object key)
* boolean containsKey(Object key)
* int size()
* Set keySet()

**class java.util.HashMap implements java.util.Map**

**class java.util.TreeMap implements java.util.Map**

**interface java.util.Iterator**

* boolean hasNext()
* Object next()
* void remove()

**interface java.util.ListIterator extends java.util.Iterator**

* Methods in addition to the Iterator methods
* void add(Object x)
* void set(Object x)

**Implementation classes for linked list and tree nodes**

public class ListNode

{

public ListNode(Object initValue, ListNode initNext)

{ value = initValue; next = initNext; }

public Object getValue() { return value; }

public ListNode getNext() { return next; }

public void setValue(Object theNewValue) { value = theNewValue; }

public void setNext(ListNode theNewNext) { next = theNewNext; }

private Object value;

private ListNode next;

}

public class TreeNode

{

public TreeNode(Object initValue)

{ value = initValue; left = null; right = null; }

public TreeNode(Object initValue, TreeNode initLeft, TreeNode initRight)

{ value = initValue; left = initLeft; right = initRight; }

public Object getValue() { return value; }

public TreeNode getLeft() { return left; }

public TreeNode getRight() { return right; }

public void setValue(Object theNewValue) { value = theNewValue; }

public void setLeft(TreeNode theNewLeft) { left = theNewLeft; }

public void setRight(TreeNode theNewRight) { right = theNewRight; }

private Object value;

private TreeNode left;

private TreeNode right;

}

**Interface for stacks** (\* See note at end of reference)

public interface Stack

{

// postcondition: returns true if stack is empty;

// otherwise, returns false

boolean isEmpty();

// precondition: stack is [e1, e2, ..., en] with n >= 0

// postcondition: stack is [e1, e2, ..., en, x]

void push(Object x);

// precondition: stack is [e1, e2, ..., en] with n >= 1

// postcondition: stack is [e1, e2, ..., e(n-1)]; returns en

// throws an unchecked exception if the stack is empty

Object pop();

// precondition: stack is [e1, e2, ..., en] with n >= 1

// postcondition: returns en

// throws an unchecked exception if the stack is empty

Object peekTop();

}

**Interface for queues** (\* See note at end of reference)

public interface Queue

{

// postcondition: returns true if queue is empty;

// otherwise, returns false

boolean isEmpty();

// precondition: queue is [e1, e2, ..., en] with n >= 0

// postcondition: queue is [e1, e2, ..., en, x]

void enqueue(Object x);

// precondition: queue is [e1, e2, ..., en] with n >= 1

// postcondition: queue is [e2, ..., en]; returns e1

// throws an unchecked exception if the queue is empty

Object dequeue();

// precondition: queue is [e1, e2, ..., en] with n >= 1

// postcondition: returns e1

// throws an unchecked exception if the queue is empty

Object peekFront();

}

**Interface for priority queues** (\* See note at end of reference)

public interface PriorityQueue

{

// postcondition: returns true if the number of elements in

// the priority queue is 0;

// otherwise, returns false

boolean isEmpty();

// postcondition: x has been added to the priority queue; the number

// of elements in the priority queue is increased by 1.

void add(Object x);

// postcondition: The smallest item in the priority queue is removed

// and returned; the number of elements in the priority

// queue is decreased by 1.

// throws unchecked exception if priority queue is empty

Object removeMin();

// postcondition: The smallest item in the priority queue is returned; the

// priority queue is unchanged

// throws unchecked exception if priority queue is empty

Object peekMin();

}

**\* Note regarding use of stacks, queues, and priority queues**

When a stack, queue, or priority queue object needs to be instantiated, code such as the following is used:

Queue q = new ListQueue(); // ListQueue implements Queue