CS211	Computers and Programming	
	ASSIGNMENT 6: Inheritance Revisited	
Fall 1999		**November 4, 1999**

DUE DATE: November 11.
Note that you have **one** week to complete this.

This assignment reinforces subtyping. It uses implementation and interface inheritance to implement data structures.

Problem 1: Implement a Queue

Specify an interface called IQueue which has the following methods:

- **queueSize** – returns the size of the queue
- **isQueueEmpty** – returns *true* if the queue is empty, *false* otherwise
- **enqueue** – takes an *object* and appends it to the back of the queue
- **enqueue** – takes an *object* to inserts it into the queue at the location *index*
- **dequeue** – takes an *object*, searches for it in the queue, and removes it if found. Returns *true* if the operation was successful, *false* otherwise
- **dequeue** – removes the object at the front of the queue and returns it.
- **isPresent** – takes an *object* and returns *true* if it is in the queue, *false* otherwise

Write a class called MyQueue that implements the IQueue interface by *using* the java.util.ArrayList class. API documentation for the ArrayList class can be found at: http://java.sun.com/products//jdk/1.2/docs/api/java/util/ArrayList.html. The ArrayList class provides methods that can be used to implement the methods specified in the IQueue interface.

Write a client to test the methods of MyQueue on two different object types. (Note: This is not the same as 2 instances of the same object!) Show clearly the contents of the queue before and after the method invocations. For example, the queue is initially empty, and a call to *enqueue* on *Object* ***a*** would increase the queue size as well as make ***a*** present.

Problem 2: Implement a Priority Queue

Write a class called MyPQ that extends MyQueue from Problem 1 to implement a priority queue. A priority queue is a specialized queue that determines insertion (and thereby removal) order by *priority*. It is up to you whether you want to keep the queue elements in increasing or decreasing priority.

All you have to do is override the necessary methods from Problem 1.

Write a similar client to Problem 1 (also on two object types), clearly showing the order with which the elements are now stored. (When would you use one over the other?)

Problem 3: Implement a List using Arrays

Specify an interface called IList that has the following methods:

- **size** – returns the size of the list
- **isEmpty** – returns *true* if the list is empty; *false* otherwise
- **contains** - returns *true* if the list contains the *object*; *false* otherwise
- **add** – add *object* to the front of the list
- **remove** – remove *object* from the list if found. Returns *true* if the operation was successful, *false* otherwise.
- **clear** – remove all elements in the list
- **toArray** – returns the list as an array
- **get** – returns the element at position *index* in the list
- **set** – puts *object* in the list at position *index*
- **indexOf** – returns the index in the list where the *first* occurrence of the *object* can found
- **lastIndexOf** – returns the index in the list where the *last* occurrence of the *object* can found
- **noOfOccurrences** – returns the number of occurrences of *object* in the list

Write a class called MyList that implements the IList interface using *an array of Comparables*. The list should however grow and shrink as elements are inserted and removed from the list. This is achieved by creating a new array when elements are inserted and removed from the list.

Write a client to test *each* of the methods of *MyList*. Pay emphasis to showing that the array resizes itself with the *add* and *remove* methods.

Problem 4: Using a Comparator

A class can implement the Comparable interface to provide a *natural ordering* of its objects. The class StudentRecord (introduced in the lectures notes) provides an example of this approach. What if we want to order the students *only* by their id? The method compareTo() implemented by the StudentRecord class is then not adequate – it uses various criteria to compare students. The solution is again to provide a *separate object* that knows how to compare two StudentRecord objects by their student id. Such an object is called a *comparator*, and it usually implements the following interface:

```
interface Comparator {
    int compare(Object o1, Object o2);
}
```

A negative integer, zero, or a positive integer returned by the compare() method indicates that the first argument is less than, equal to, or greater than the second.

Write a comparator (i.e. a class) for the StudentRecord class.

Modify the mergeSort() method in SortComparableArray class (provided with the lecture notes) so that the method takes a comparator as parameter to mergesort any array of Object.

Write a client to test the new mergesort() method with an array of at least 5 students.

What to hand in:

Hand in the *design documentation* (class and collaboration diagrams), the *code* (preferably documented with javadoc comments) and the *client outputs* from Problem 1 through Problem 4.