University of Toronto
CSC148—Intro. to Computer Science **St. George Campus, Fall 2007**

Assignment 1: Contracts, Interfaces, and Exceptions

Due: 10:10am, Mon. Oct. 15 (in class).

Purpose. This assignment gives you practice writing ADTs, meeting method contracts, and testing code using JUnit.

Marking. This assignment is worth 8% of your final mark. We will not mark all parts of this assignment. But you don't know which parts will be marked, so do them all.

Take Note of any Vital Rules. Throughout this course a "vital rule" means something that you must follow if you are to get any marks at all for the question. If you break a vital rule, you will automatically get a mark of zero on the corresponding part of the assignment. This might seem overly strict, but we want you to practice doing things a certain way. We use these vital rules to make sure you don't work around the main point of the question. We have listed the vital rules for this assignment further below.

Preparation. Make an A1 directory on your hard drive. Copy A1handout.zip from the course homepage to this directory. In a command prompt window, cd to this directory, and type:

```
unzip A1handout.zip
```

This will create several "*.java" files which contain the handout code. Finally, move A1handout.zip out of this directory, so you don't overwrite your work by mistake.

Browse the handout code in your A1 directory. The file XQueue.java provides an interface which extends the Queue interface. That is, the XQueue interface includes all the methods listed in Queue.java together with the new ones listed in XQueue.java.

In the method comments in the handout code when we refer to the index of an object in a queue we always mean the **logical index.** The logical index is given in terms of the FIFO ordering of elements in the queue ADT. That is, the head element has logical index 0, the next has (logical) index 1, and so on until the last element, which has index size-1. Note the logical index is distinct from the array index within the contents[] array where, in CircXQueue for example, the head element could be at array index 115. The users of our ADT methods typically don't want to know about the array index. So whenever we refer to index in a method comment, we mean logical index.

Notice the method iterator() in XQueue.java returns (a reference to) an object of some class which implements the interface Iterator. Read the description of the java.util.Iterator interface on the Java 2 (version 1.4.2) API. This API is at http://java.sun.com/j2se/1.4.2/docs/api/ . The description of the same interface in Java 1.6 includes parameterized types, which we will cover later in the course. You should ignore Java 1.6's API for this assignment. (However, you can still run your assignment using the either the Java 1.4 or Java 1.6 compiler and run-time environment.)

Also read the page for java.util.ConcurrentModificationException provided in the Java 2 API (see the above URL), taking particular note of the description of fail-fast iterators. A further example of the behaviour of a fail-fast iterator is provided below.

Your Job.

1. CirXQueue.java provides a partial implementation of an XQueue that is based on the CircularQueue discussed in the lectures. In particular, the methods dequeue, enqueue, head, size, and capacity have already been completed. In the handout code other methods and constructors simply throw

an UnsupportedOperationException. Your job is to complete all the methods and constructors in CirXQueue which throw this exception in the handout code. The completed methods and constructors should satisfy the contracts provided by the existing method headers.

2. The file CirXQueueTester.java contains a JUnit class which runs some simple tests on the methods that have already been implemented in the handout code. For each new XQueue method you implement in CirXQueue, test it with at least one simple JUnit test added to the CirXQueueTester class. These tests should be of typical situations in which the methods may be used. *We do not expect thorough testing here.*

3. Write a class definition for FleXQueue which extends CirXQueue. The class FleXQueue should provide an XQueue which automatically adjusts the capacity of the queue according to the size of the queue. That is, when an enqueue method is called with the size of the queue equaling its capacity, the enqueue method should automatically increase the capacity and then append the new item. Similarly, the XQueue's dequeue() and remove(int) methods, along with the iterator's remove() method, should also cause the queue's capacity to shrink if the queue size becomes substantially smaller than the capacity. These methods must still meet the terms of their contracts, as specified by the method headers, and not change the logical order of the items in the queue in any other way than specified there.

   You can decide on what values to use for the new capacities when your queue grows and shrinks. Make sure that your queue shrinks when it is less than about half full. But do not bother shrinking a queue whose capacity is already small (e.g. a capacity 16 or less).

   In addition, write two constructors in FleXQueue.java corresponding to the two constructors in CirXQueue, namely FleXQueue(int), and FleXQueue(XQueue), along with a third constructor FleXQueue() which builds a queue with some default capacity.

4. Write a JUnit test class called FleXQueueTester which provides at least one simple test for each method implemented in FleXQueue.java. (A great way to start is simply by copying CirXQueueTester.java to FleXQueueTester.java, and replacing all strings CirX with FleX.)

5. The iterator method for both CirXQueue and FleXQueue should return a fail-fast iterator with the hasNext(), next(), and remove() methods all implemented according to their descriptions in the java.util.Iterator API. Specifically, the iterator's remove() method should be implemented (the API indicates it is optional, but it is required in this assignment). See below for an example of the behaviour expected from these methods.

6. In CirXQueueTester.java include at least one simple test for each iterator method (i.e., hasNext(), next(), and remove()).

Coding Style. We expect you to maintain the style of the handout code and the code written in the lecture notes. We use the industry standard style described in http://java.sun.com/docs/codeconv/. See the Java links on the course homepage for information on style checking software that you can use to get feedback on where your own code deviates from these conventions.

In marking this assignment we will deduct marks for code that deviates significantly from these style conventions. We expect the code to be appropriately commented, including the standard Javadoc comments.

Vital Rules. Do not break these three rules:

1. You cannot change or move any of the Java handout code except: a) CirXQueueTester.java (which you are free to change in any way you like), and b) by completing the missing method bodies in CirXQueue.java. Specifically, the only methods in CirXQueue.java you can change are all of those which currently throw an UnsupportedOperationException. You can delete these throws clauses and add code to provide the method implementations. You cannot add any class or instance variables to CirXQueue, nor change the protection of any existing ones from being *private*. You may add *private*

helper methods to CirXQueue.java, if you wish.

2. You cannot use any *non-private* methods, *non-private* class variables, or *non-private* instance variables in the FleXQueue class other than those dictated by the XQueue interface and the three constuctors noted above. You can use public constants if you wish. FleXQueue must extend CirXQueue (with no intervening subclasses).

3. You will need to add class(es) defining the iterator(s). If these are written as separate public classes in their own "*.java" files, then name them CirXQueueIterator.java and FleXQueueIterator.java. The only *non-private* members of these iterator classes must be the constructor(s) and the three methods hasNext(), next(), and remove().

Background on Fail-Fast Iterators. The key property of a fail-fast iterator is that it throws a ConcurrentModificationException whenever the underlying data structure has been modified in some substantial way by some method other than its own remove() method. We give a concrete example of this below:

```
// An example of a fail-fast iterator.
// Suppose q refers an XQueue which is currently of size 8

Iterator it = q.iterator();  // provides a new iterator for q

it.next();    // Returns the head element of the XQueue q.
it.hasNext(); // Returns true
it.next();    // Returns the second element of the XQueue q.
q.size();     // Returns 8 ... the size of the original q.
q.head();     // Returns the head element of the Xqueue q, as above.
it.remove();  // Removes the second element of the XQueue q.
q.resetCapacity(10); // Resets q's capacity to be 10.
it.next();    // Returns the third element of the XQueue q.

q.enqueue("hello");  // Whoops, we've just added something
                     // to the queue we are iterating over.

it.hasNext();  // throws a ConcurrentModificationException;
```

In general, once we have created an iterator for q, we can interleave calls to any of q's public methods, **except for the three methods q.dequeue(), q.enqueue(), and q.remove()**, with calls to the iterator. These three queue methods q.dequeue(), q.enqueue(), and q.remove() are the only queue methods which change the structural properties of the queue. That is, the logical order of the items and/or the number of items in the queue are changed by these three methods and the iteration may no longer be consistent with these changes. Notice q.resetCapacity() does not change the number or order of elements in the queue, and therefore calling this method between calls to the iterator's methods is allowed.

A fail-fast iterator detects any change to the structural properties of the queue that are not a consequence of calls to the iterator's own remove() method. Any of the three methods hasNext(), next(), or remove() should detect such structural changes. So, for example, in the code fragment above we could change q.enqueue("hello") to q.dequeue() or q.remove(2) and the same exception would be raised on the last line. Similarly, changing the last line above to it.next() or it.remove() would still cause the exception to be thrown.

Submitting This Assignment. Instructions for how to electronically submit your java files will be posted on the course web page. You will need to submit CirXQueue.java, CirXQueueTester.java, FleXQueue.java, FleXQueueTester.java and possibly the optional files CirXQueueIterator.java and FleXQueueIterator.java. The precise names of these files matter for the electronic submission, including capitalization.

In addition, at the beginning of class on the due date, you need to hand in printed copies of all the "*.java" files that you submit electronically. The pages corresponding to each Java file should be stapled together, separately from the pages for other files. Write your name, student number, and the name of the class definition (eg. CirXQueue) on each stapled set. Put all of these in a large manilla envelope with "CSC 148 Assignment 1", your name, and your student number written on the outside.