Introduction to Computer Science

In this course we'll work on the following skills:

Specification. Practice writing clear and precise mathematical specifications (see CSC165H for more on this).

Design. Know the properties of good program design, and practice it. Learn some standard metaphors that have proven useful in the design of software, and practice using them (eg. stacks, queues, graphs and so on). These are known as abstract data types (ADTs).

Analysis. Reason about an algorithm's efficiency and correctness, perhaps before any code is written.

Implementation. Learn programming constructs that allow the clean implementation of ADTs. Learn about recursion. Practice producing clean, modular, well designed, easy to understand code.

Verification. Design systematic and thorough test suites. Practice unit testing and writing clear documentation of tests.

Let's get going with the Queue and Stack ADTs, along with tools in Java which allow us to implement them effectively. In the process we will illustrate each of the above skills.

Abstract Data Type: Queue

Properties. A queue is a sequence of objects. The objects are removed in the order they are inserted. This is referred to as the first in, first out rule (**FIFO**). The objects at the front and back of the queue are called the head and tail.

Primitive Operations:

enqueue(o)	Put object o on the end of the queue.
dequeue()	Remove and return the object at the front of the queue.
head()	Reveal the front object in the queue, but do not remove it.
size()	Return the number of objects currently in the queue.
capacity()	Return the maximum number of objects the queue can hold.

Other operations can be defined in terms of these primitive ones. For example:

- **isEmpty()** can be defined by testing whether size() is 0.
- **isFull()** can be defined by testing whether size() equals the capacity().

These last two could also be considered primitive operations. There is a balance between minimizing the set of primitive operations, versus providing many convenient operations for users.

Abstract Data Type: Stack

Properties. A stack is a sequence of objects. The objects are removed in the *reverse order* they were inserted. This is referred to as the last in, first out rule (**LIFO**). The last element to arrive is said to be at the top of the stack.

Primitive Operations:

push(o)	Put object o on the top of the stack.
pop()	Remove and return top object in the stack.
size()	Return the number of objects currently in the stack.
capacity()	Return the maximum number of objects the stack can hold.

Note we might use top() to refer to an operation which returns the top element of the stack, but not actually remove it;

- **top()** can be defined using pop() and then push(o), where o is the object just popped.

This is analogous to the operation head() for a Queue.

Is there an easy way to define head() in terms of the other primitive operations in Queue?

Implementing an ADT in Java

To implement a queue or stack we need,

- Variables to represent the data.
- Methods which can perform the operations.

The implemented methods should follow the same pattern as the primitive operations described above.

Java **interfaces** allow us to express the primitive operations any queue implementation must have. Such a specification has the advantage that:

- It is then trivial to switch between different implementations of the same queue interface (i.e., the same primitive operations).
- Some members of a programming team can be assigned the task of implementing the interface, while others can work on the code which uses that interface. This is the **divide and conquer** approach to solving complex problems.

A Java Interface

An interface in Java is like a class, but without any instance variables or method bodies.

```
public interface Queue {
    /** Append the element o to this queue.
     *  @param  o  the element to be added. */
    void enqueue(Object o);

    /** Return the first object in the queue, but
     *  do not remove it.  The queue must not be
     *  empty.
     *  @return  the first object in the queue. */
    Object head();

    /** Return and remove the first object in the
     *  queue.  The queue must not be empty.
     *  @return  the first object in the queue. */
    Object dequeue();

    /** The number of elements in this queue.
     *  @return  the current number of elements. */
    int size();

    /** The maximum number of elements this queue
     *  can hold.
     *  @return  maximum capacity */
    int capacity();

    /** Is this queue full?
     *  @return  true if no more elements can
     *           be added.  */
    boolean isFull();

}
```

All interface members are automatically public (by convention this keyword is omitted).

Using Queue

We cannot create instances of Queue (there are no method bodies).

```
public class Broken {
  public static void main(String[] args) {
    Queue a = new Queue(15);
    a.enqueue(new Integer(66));
  }
}
```

However if ArrayQueue is a class which implements Queue, then we can, of course, create an instance of it using its constructor and the `new` operator. Moreover, we can **cast** a reference of type ArrayQueue to the interface type, Queue. (See the first line of the main method below.)

```
public class CastAbout {
  public static void main(String[] args) {
    Queue a = new ArrayQueue(15);       // Cast
    a.enqueue(new Integer(66));         // Cast
    Integer b = (Integer) a.dequeue();  // Cast
    int k = b.intValue();
  }
}
```

Here the variable `a` has interface type Queue, but it refers to the original object of class ArrayQueue created by the `new` operator.

To be able to do this cast, the ArrayQueue class must be defined in a special way, making it clear that it “implements Queue”. We will show how to do this further below.

Casting and an object's Class

It is worth taking a second look at the `CastAbout` code:

```
public class CastAbout {
  public static void main(String[] args) {
    Queue a = new ArrayQueue(15);       // Cast
    a.enqueue(new Integer(66));         // Cast
    Integer b = (Integer) a.dequeue();  // Cast
    int k = b.intValue();
  }
}
```

- Casting never changes the memory storage of an object, it **only** changes how that object is referenced. (This is not true for casts of primitive data types, such as `float` and `int`.)
- All the compiler knowns about the object referred to by `a`, above, is that it is an instance of a class that implements the Queue interface. Therefore it is known to have all the methods specified in Queue, along with all the methods inherited from Object.
- Another example is the cast `(Object) new Integer(66)`, which is implicit in the `enqueue` method call. This produces a reference to an instance of the Integer class having contents 66, but the type of this reference is Object. (Why is this useful?)
- Thirdly, `Integer b = (Integer) q.dequeue()` casts the reference of type Object returned by `dequeue()` to an Integer type. This is ok for the above program since we know the object being returned is of the Integer class.

The Plan for ArrayQueue

Let us implement the methods in Queue by creating some class, say TryQueue. One possible approach using an array is as follows:

Operation	contents[*]					size
`q=new TryQueue(5)`	null	null	null	null	null	0
`q.enqueue(''C'')`	C	null	null	null	null	1
`q.enqueue(''R'')`	C	R	null	null	null	2
`q.dequeue()`	R	x	null	null	null	1
⋮			⋮			⋮
`q.enqueue(''H'')`	R	A	S	H	null	4
`q.dequeue()`	A	S	H	x	null	3
`q.dequeue()`	S	H	x	x	null	2

Enqueue is an easy operation (try writing the enqueue method!). But the dequeue operation requires that all remaining queue elements be shifted one the left (try writing this method too!). (The x's above do not belong to the stored queue, they are used only to make the end of the queue clear. For the following implementation, what values do they correspond to?).

This plan leads to the following `ArrayQueue` class definition. Note we **must** implement each of the methods in the Queue interface.

Implementing Queue

```
/**
 * A Queue with fixed capacity.
 * Operations are constant-time except for
 * construction which is O(capacity) and
 * dequeue which is O(size()).
 */
public class ArrayQueue implements Queue {

    /** The number of elements in me. */
    private int size;

    /** contents[0 .. size-1] contains my elements. */
    private Object[] contents;

    // Representation invariant:
    //   Let capacity denote contents.length.
    //   0 <= size <= capacity.
    //   If size is 0, I am empty.
    //   If size > 0:
    //     contents[0] is the head
    //     contents[size-1] is the tail
    //     contents[0 .. size-1] contains the
    //       Objects in the order they were inserted.

    /** An ArrayQueue with capacity for n elements.
     *  Requires n>0.  */
    public ArrayQueue(int n) {
        contents = new Object[n];
    }

    /** Append o to me.
      * Requires: size() < capacity */
    public void enqueue(Object o) {
        contents[size++] = o;
    }
```

Continued on next slide...

So far this looks just like a normal class definition, only the phrase

"`implements Queue`" is different.

Implementing Queue (Cont.)

```
    /** Remove and return my front element.
      * This is O(size()).
      * Requires: size() != 0. */
    public Object dequeue() {
        Object head = contents[0];

        // Move all the other elements up one spot.
        for (int i = 0; i < size - 1; ++i) {
            contents[i] = contents[i+1];
        }

        if (size > 0)
          --size;
        return head;
    }

    /** Return my front element.
      * Requires: size() != 0. */
    public Object head() {
        return contents[0];
    }

    /** Return the number of elements in me. */
    public int size() {
        return size;
    }

    /** Return the maximum number I can hold. */
    public int capacity() {
        return contents.length;
    }

    /** Am I full? */
    public boolean isFull() {
        return size == capacity();
    }

} // End of ArrayQueue
```

Two Views of an Interface

In summary, from the point of view of the team which is implementing an interface, the interface represents an **obligation**:

- Methods with the same signature and functionality of those specified by the interface must be provided by the class being written.
- These methods must be public.
- Other public methods, besides those in the Object class, should not be included[1], since they won't be visible after casting to the interface (i.e., the typical use of the class being defined).

Whether or not `ArrayQueue` meets this obligation still needs to be tested (see below).

On the other hand, for the teams which are going to use the defined class, the interface represents a **guarantee**:

- All the primitive methods described in the interface, with the same signatures, will be provided.
- These methods will implement the primitive operations specified in the comments of the interface.

For example, we used this guarantee in writing the `CastAbout` code.

[1] Unless, as always, you have a very good reason to do otherwise.

Alternative Implementations

Many alternative implementations are possible. For example, we might consider the use of an ArrayList instead of an array for `contents`. This would allow the capacity to be as large as necessary, until the computer's memory ran out.

For teaching purposes we choose to stick with simple arrays here, since then no important implementation details are hidden from the student. (Eg. ArrayList and Vector are implemented in Java using arrays.)

Another problem with ArrayQueue, besides the maximum length constraint, is that the `dequeue` method must shift all the elements over by one. This will get slower and slower as the queue gets longer. The time taken is roughly proportional to the queue length, as the length gets large. This is what is meant by the comment, "This is `O(size)`", in the method dequeue() above. We will study more about this "big-Oh" notation later in this course.

Using Wrap-Around

We can avoid the need to shift every object over by allowing the `head` element to be anywhere in the `contents` array, rather than just at index 0.

Operation	contents[*]					head	tail
`q=new TryQueue(5)`	null	null	null	null	null	0	4
`q.enqueue(''C'')`	C	null	null	null	null	0	0
`q.enqueue(''R'')`	C	R	null	null	null	0	1
⋮			⋮			⋮	⋮
`q.dequeue()`	x	R	A	S	null	1	3
`q.enqueue(''H'')`	x	R	A	S	H	1	4
`q.dequeue()`	x	x	A	S	H	2	4
`q.enqueue(''D'')`	D	x	A	S	H	2	0

Also, to avoid having the queue contents crash into the upper limit of the `contents` array, we can index into the array using modular arithmetic. That is, to get the next array element after the `k`-th one, we use `contents[j]` for `j=(k+1) % contents.length`. (The x's above are used to make the location of the queue clear. For the following implementation, what values do they correspond to?).

This leads to the `CircularQueue` . . .

Refresher for mod

The mod operation, $x \mod d$ gives the remainder when integer x is divided by integer d.

In Java this is written as $x \,\%\, d$. The resulting value is defined to be the remainder r, where

$$r = x - n * d, \quad \text{for } n = \text{chop}(x/d),$$

and chop(z) is just z **rounded towards zero**.

Examples

x	y	$x \,\%\, d$	
2	5	2	Remainder after dividing 2 by 5.
29	5	4	Remainder after dividing 9 by 5.
100	5	0	Remainder after dividing 100 by 5.
-1	5	-1	Note $x\%d \leq 0$ when $x < 0$.
-18	5	-3	Note chop(-18/5) = chop(-3.6) = -3.
-10	5	0	Remainder after dividing -10 by 5.

CircularQueue

```
/** A Queue with fixed capacity and  constant-time
  * operations (except for creation). */
public class CircularQueue implements Queue {

    /** The number of elements in me. */
    private int size;

    /** The index of the head of the queue. */
    private int head;

    /** The items in me,
        stored in contents[head .. tail],
        with wraparound. */
    private Object[] contents;

    // Representation invariant:
    //   Let "capacity" be contents.length.  Then
    //     0 <= head < capacity,
    //     0 <= size <= capacity, and
    //     tail = (head+size-1+capacity) % capacity.
    //   If size is 0, I am empty.
    //   If size > 0:
    //     contents[head] is the head
    //     contents[tail] is the tail
    //     if head <= tail,
    //       contents[head .. tail] contains
    //       the Objects in the order they were
    //       inserted.
    //     if head > tail,
    //       contents[head .. capacity-1, 0 .. tail]
    //       contains the Objects in the order they
    //       were inserted.

    /** A CircularQueue with capacity for n elements.
     *  @param n the capacity, requires n>0. */
    public CircularQueue(int n) {
        contents = new Object[n];
    }
```

Continued on next slide...

CircularQueue (Cont.)

```
    /** Append o to me.
        Requires: size() < capacity */
    public void enqueue(Object o) {
        if (size < contents.length) {
          int newTail = (head+size) % contents.length;
          contents[newTail] = o;
          ++size;
        }
    }
    /** Remove and return my front element.
        Requires: size() != 0.
        @return the front element */
    public Object dequeue() {
        if (size>0) {
          Object result = contents[head];
          head = (head+1) % contents.length;
          --size;
          return result;
        }
        return null;
    }

    /** Return the number of elements in me. */
    public int size() {
      return size;
    }
    /** Return the first object, don't remove it.
     *  I better not be empty. */
    public Object head() {
        return contents[head];
    }
    /** Return the maximum number I can hold. */
    public int capacity() {
        return contents.length;
    }
    /** Am I full? */
    public boolean isFull() {
        return size == capacity();
    }
} // End of CircularQueue
```