CS206

Queues
Queues

- Insertions and deletions are First In First Out – FIFO
  - Insert at the back
  - Delete from the front
Queues
Queueing Theory

Agner Krarup Erlang
Queue Interface

- null is returned from peek() and poll() when queue is empty
- return false from offer when cannot add to queue.

```java
public interface QueueIntf<Q> {
    boolean isEmpty();
    int size();
    boolean add(Q q)
        throws IllegalStateException;
    Q remove()
        throws NoSuchElementException;
    Q element()
        throws NoSuchElementException;
    boolean offer(Q q);
    Q poll();
    Q peek();
}
```
## Example

<table>
<thead>
<tr>
<th>Operation</th>
<th>output</th>
<th>Queue Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>offer(5)</td>
<td>TRUE</td>
<td>{5}</td>
</tr>
<tr>
<td>offer(3)</td>
<td>TRUE</td>
<td>{5, 3}</td>
</tr>
<tr>
<td>poll()</td>
<td>5</td>
<td>{3}</td>
</tr>
<tr>
<td>offer(7)</td>
<td>TRUE</td>
<td>{3, 7}</td>
</tr>
<tr>
<td>poll()</td>
<td>3</td>
<td>{7}</td>
</tr>
<tr>
<td>peek()</td>
<td>7</td>
<td>{7}</td>
</tr>
<tr>
<td>poll()</td>
<td>7</td>
<td>{}</td>
</tr>
<tr>
<td>poll()</td>
<td>null</td>
<td>{}</td>
</tr>
</tbody>
</table>
Array-based Queue

- An array of size $n$ in a circular fashion
  - $frontLoc$: index of the front element
    - where objects are read
  - $count$: number of stored elements
  - $rearLoc$: index of rear element
    - where objects are added

![Diagram of normal and wrapped-around configurations of an array-based queue.](image-url)
Circular Array and Queue

• When the queue has fewer than $n$ elements, location

  • $\text{rearLoc} = (\text{frontLoc} + \text{count}) \mod n$
Start of Queue Implementation

```java
public class ArrayQueue<Q> implements QueueIntf<Q> {
    /** the default capacity for the backing array */
    private static final int CAPACITY = 40;
    /** The array in which the queue data is stored */
    private Q[] backingArray;
    /** the number of items in the queue*/
    private int count;
    private int frontLoc;  // The array location of the end of the queue */
    private int rearLoc;   // The array location of the head of the queue */
    /** Create an array backed queue with the default capacity. */
    public ArrayQueue() {
        this(CAPACITY);
    }
    /**
     * Create an array backed queue with the given capacity
     * @param qSize the capacity for the queue */
    public ArrayQueue(int qSize) {
        count = 0;
        frontLoc = 0;
        backingArray = (Q[]) new Object[qSize];
    }
}
```

Performance and Limitations for array-based Queue

• Performance
  ▪ let $n$ be the number of objects in the queue
  ▪ The space used is $O(n)$
  ▪ Each operation runs in time $O(1)$

• Limitations
  ▪ Max size is limited and cannot be changed
  ▪ Adding to a full queue returns false (offer method)
The Comparable Interface

```java
public interface Comparable<T>

    { /* deleted more */
        /**
         * Compares this object with another, and returns a numerical result based
         * on the comparison. If the result is negative, this object sorts less
         * than the other; if 0, the two are equal, and if positive, this object
         * sorts greater than the other. To translate this into boolean, simply
         * perform <code>o1.compareTo(o2) <em>&lt;op&gt;</em> 0</code>, where op
         * is one of &lt;, &lt;=, =, !>, &gt;, or &gt;=.
         */

        int compareTo(T o);
    }
```

Short story: return 0 if equal, negative if less, positive if greater
Comparable example
Integer and String

```java
public class ComparableEx {
    public static void main(String[] args) {
        Integer i5 = new Integer(50);
        Integer i3 = new Integer(30);
        Integer j5 = new Integer(50);
        System.out.println("i5:" + i5 + "  i3:" + i3 + "  j5:" + j5);
        System.out.println("i3.compareTo(i5) " + i3.compareTo(i5));
        System.out.println("i5.compareTo(i3) " + i5.compareTo(i3));
        System.out.println("i5.compareTo(j5) " + i5.compareTo(j5));
        System.out.println("i5.equals(j5) " + i5.equals(j5));
        System.out.println("(i5 == j5) " + (i5 == j5));

        String abc = "abc";
        String def = "def";
        String abc1 = new String("abc");
        System.out.println("abc:" + abc + "  def:" + def + "  abd0:" + abc1);
        System.out.println("abc.compareTo(def) " + abc.compareTo(def));
        System.out.println("def.compareTo(abc) " + def.compareTo(abc));
        System.out.println("abc.compareTo(abc0) " + abc.compareTo(abc1));
        System.out.println("abc.equals(abc0) " + abc.equals(abc1));
        System.out.println("abc == abc0 " + (abc == abc1));
    }
}
```
Comparable Rabbit

```java
public class Rabbit implements Comparable<Rabbit> {
    enum BreedEnum {
        DwarfDutch, Angora, FrenchLop
    }
    private final BreedEnum breed;
    private final int ID;
    private final String nickname;
    public Rabbit(BreedEnum breed, int id, String nn) {
        this.breed = breed;
        this.ID = id;
        this.nickname = nn == null ? makeName() : nn;
    }
    public int compareTo(Rabbit o) {
        return id - o.getId();
    }
}
```
Comparable in SortedArray

• SortedArray implicitly used comparable as String implements it.

• So, make it explicit

```java
public class SAL<E extends Comparable<E>> {
    enum Ordering { ASCENDING, DESCENDING }
    ArrayList<Comparable<E>> sortedAL;
    public void add(Comparable<E> stringToAdd) {
        int loc = findPlace(stringToAdd);
        insertAtLoc(stringToAdd, loc);
    }
    private int findPlace(Comparable<E> toAdd) {
        int place = 0;
        while (place < sortedAL.size()) {
            if (toAdd.compareTo((E)sortedAL.get(place)) < 0) {
                break;
            }
            place++;
        } return place;
    }
}
```
A little more in SAL

```java
private int findPlace(Comparable<E> toAdd) {
    int place = 0;
    while (place < sortedAL.size()) {
        switch (theOrder) {
            case ASCENDING:
                if (toAdd.compareTo(sortedAL.get(place)) < 0) {
                    break;
                }
                break;
            case DESCENDING:
                default:
                if (toAdd.compareTo(sortedAL.get(place)) > 0) {
                    break;
                }
                break;
        }
        place++;
    }
    return place;
}
```
public class CompRabbits {
    public static void main(String[] args) {
        SAL<Rabbit> rsal = new SAL<>();
        rsal.add(new Rabbit(Rabbit.BreedEnum.Angora, 45, "Flopsy"));
        rsal.add(new Rabbit(Rabbit.BreedEnum.DwarfDutch, 46, "Mopsy"));
        rsal.add(new Rabbit(Rabbit.BreedEnum.FrenchLop, 47, "Cottontail"));
        rsal.add(new Rabbit(Rabbit.BreedEnum.Angora, 44, "Peter"));
        rsal.add(new Rabbit(Rabbit.BreedEnum.DwarfDutch, 10, "Josephine"));
        rsal.add(new Rabbit(Rabbit.BreedEnum.FrenchLop, 17, "Benjamin"));
        System.out.println(rsal);
    }
}
Queue Offer Method