- A queue is consistent with the general concept of
  - a waiting line to buy movie tickets
  - a request to print a document
  - crawling the web to retrieve documents

- A queue is a linear collection whose elements are added on one end and removed from the other

- Queue elements are processed in a first in, first out (FIFO) manner

- Elements are removed from the queue in the same order in which they are placed on the queue
package javafoundations;

public interface Queue<T> {
    // Adds element to rear of the queue
    public void enqueue(T element);
    // Removes and returns element at front of queue
    public T dequeue();
    // Returns reference to first element without removing
    public T first();
    // Returns true if queue contains no elements
    public boolean isEmpty();
    // Returns number of elements
    public int size();
    // Returns string representation
    public String toString();
}

A radix sort is unusual because it does not involve comparisons between elements!

The technique used in the radix sort is based on the structure of the sort key

Separate queues are created for each possible value of each digit or character of the sort key

Radix = The number of queues, or the number of possible values

if we were sorting strings made up of lowercase alphabetic characters, the radix would be 26, one for each possible character

if we were sorting decimal numbers, then the radix would be 10, one for each digit 0 to 9

The radix sort makes a pass through the values for each position in the sort key

<table>
<thead>
<tr>
<th>Digit</th>
<th>1s position</th>
<th>10s position</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Original list:

442  503  312  145  250  341  325  102  420  143

<table>
<thead>
<tr>
<th>Digit</th>
<th>1s position</th>
<th>10s position</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>503  102</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>312</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>325  420</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>145  143  442 341</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>250</td>
</tr>
</tbody>
</table>
Radix Sort

<table>
<thead>
<tr>
<th>Digit</th>
<th>100s position</th>
<th>front of queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>145 143 102</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>250</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>341 325 312</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>442 420</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>503</td>
<td></td>
</tr>
</tbody>
</table>

* Adds a new element to the rear of the queue, which is stored at the high end of the array

```java
package javafoundations;
import javafoundations.exceptions.*;
public class ArrayQueue<T> implements Queue<T> {
    private final int DEFAULT_CAPACITY = 10;
    private int count;
    private T[] queue;

    //Creates empty queue
    public ArrayQueue(){
        count = 0;
        queue = (T[]) (new Object[DEFAULT_CAPACITY]);
    }

    // (more...)
```
public T dequeue() throws EmptyCollectionException {
    if (count == 0)
        throw new EmptyCollectionException("
Dequeue operation failed. The queue is empty");

    T result = queue[0];
    count--;
    //left shift the elements to keep the front at element 0
    for(int index = 0; index < count; index++)
        queue[index] = queue[index+1];

    queue[count] = null;
    return result;
}
//Left as programming projects:
//public T first() throws EmptyCollectionException{}
//public void enqueue (T element){}
//public boolean isEmpty(){
//public int size() {}
//public String toString() {}
package javafoundations;
import javafoundations.exceptions.*;

public class CircularArrayQueue<T> implements Queue<T> {
    private final int DEFAULT_CAPACITY = 10;
    private int front, rear, count;
    private T[] queue;

    public CircularArrayQueue() {
        front = rear = count = 0;
        queue = (T[]) (new Object[DEFAULT_CAPACITY]);
    }

    public void enqueue(T element) {
        if (count == queue.length)
            expandCapacity();
        queue[rear] = element;
        rear = (rear+1) % queue.length;
        count++;
    }

    public void expandCapacity() {
        T[] larger = (T[]) (new Object[queue.length*2]);
        for (int index=0; index < count; index++)
            larger[index] = queue[(front+index) % queue.length];
        front = 0;
        rear = count;
        queue = larger;
    }

    // The following methods are left as Programming Projects.
    public T dequeue() throws EmptyCollectionException { }
    public T first() throws EmptyCollectionException { }
    public int size() { }
    public boolean isEmpty() { }
    public String toString() { }
}
Both stacks and queues can be implemented very efficiently.

In almost all cases, the operations are not affected by the number of elements in the collection.

All operations for a stack (push, pop, peek, etc.) are O(1).

Almost all operations for a queue are O(1).

The only exception is the dequeue operation for the ArrayQueue implementation – the shifting of elements makes it O(n).

The dequeue operation for the CircularArrayQueue is O(1) because of the ability to eliminate the shifting of elements.