CHAPTER 5:
Queues

Java Software Structures:
Designing and Using Data Structures

Third Edition
John Lewis & Joseph Chase

© 2010 Pearson Addison-Wesley. All rights reserved.
Chapter Objectives

- Examine queue processing
- Define a queue abstract data type
- Demonstrate how a queue can be used to solve problems
- Examine various queue implementations
- Compare queue implementations
Queues

- A queue is a collection whose elements are added on one end and removed from the other.
- Therefore a queue is processed in a FIFO fashion: first in, first out.
- Elements are removed in the same order they arrive.
- Any waiting line is a queue:
  - the check out line at a grocery store
  - the cars at a stop light
  - an assembly line
Queues

- A queue is usually depicted horizontally
- One end of the queue is the *rear* (or *tail*), where elements are added
- The other end is the *front* (or *head*), from which elements are removed
- Unlike a stack, which operates on one end of the collection, a queue operates on both ends
A conceptual view of a queue
Queue Operations

• The term *enqueue* is used to refer to the process of adding an element to a queue
• Likewise, *dequeue* is the process of removing an element
• Like a stack, a pure queue does not allow the user to access the elements in the middle of the queue
• We include a `toString` method for convenience
The operations on a queue

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enqueue</td>
<td>Adds an element to the rear of the queue.</td>
</tr>
<tr>
<td>dequeue</td>
<td>Removes an element from the front of the queue.</td>
</tr>
<tr>
<td>first</td>
<td>Examines the element at the front of the queue.</td>
</tr>
<tr>
<td>isEmpty</td>
<td>Determines if the queue is empty.</td>
</tr>
<tr>
<td>size</td>
<td>Determines the number of elements on the queue.</td>
</tr>
<tr>
<td>toString</td>
<td>Returns a string representation of the queue.</td>
</tr>
</tbody>
</table>

**Figure 5.2** The operations on a queue
The QueueADT interface in UML

FIGURE 5.3 The QueueADT interface in UML
The QueueADT interface

```java
/**
 * QueueADT defines the interface to a queue collection.
 * 
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 8/12/08
 */

package jss2;

public interface QueueADT<T> {
    /**
     * Adds one element to the rear of this queue.
     * 
     * @param element the element to be added to the rear of this queue
     */
    public void enqueue (T element);
```
The QueueADT interface (continued)

```java
/**
 * Removes and returns the element at the front of this queue.
 * @return  the element at the front of this queue
 */
public T dequeue();

/**
 * Returns without removing the element at the front of this queue.
 * @return  the first element in this queue
 */
public T first();

/**
 * Returns true if this queue contains no elements.
 * @return  true if this queue is empty
 */
public boolean isEmpty();
```
The QueueADT interface (continued)

/**
 * Returns the number of elements in this queue.
 * @return the integer representation of the size of this queue
 */
public int size();

/**
 * Returns a string representation of this queue.
 * @return the string representation of this queue
 */
public String toString();
}
Coded Messages

• Let's use a queue to help us encode and decode messages
• A *Ceasar cipher* encodes a message by shifting each letter in a message by a constant amount $k$
• If $k$ is 5, A becomes F, B becomes G, etc.
• However, this is fairly easy to break
• An improvement can be made by changing how much a letter is shifted depending on where the letter is in the message
A repeating key is a series of integers that determine how much each character is shifted. For example, consider the repeating key 3 1 7 4 2 5. The first character in the message is shifted 3, the next 1, the next 7, and so on. When the key is exhausted, we just start over at the beginning of the key.
An encoded message using a repeating key

<table>
<thead>
<tr>
<th>Encoded Message:</th>
<th>n o v a n j g h l m u u r x l v</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key:</td>
<td>3 1 7 4 2 5 3 1 7 4 2 5 3 1 7 4</td>
</tr>
<tr>
<td>Decoded Message:</td>
<td>k n o w l e d g e i s p o w e r</td>
</tr>
</tbody>
</table>

**FIGURE 5.4** An encoded message using a repeating key
Coded Messages

- We'll use a queue to store the values of the key
- We'll dequeue a value when needed
- After using a key value, we then enqueue it back onto the end of the queue
- That way the queue represents the constantly cycling values in the key
The Codes class

```java
/**
 * Codes demonstrates the use of queues to encrypt and decrypt messages.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 08/12/08
 */

import jss2CircularArrayQueue;

public class Codes
{
    /**
     * Encode and decode a message using a key of values stored in
     * a queue.
     */
    public static void main ( String[] args)
    {
        int[] key = {5, 12, -3, 8, -9, 4, 10};
        Integer keyValue;
        String encoded = "", decoded = "";
        String message = "All programmers are playwrights and all " +
            "computers are lousy actors.";
```
The Codes class (continued)

```java
CircularArrayQueue<Integer> keyQueue1 = new CircularArrayQueue<Integer>();
CircularArrayQueue<Integer> keyQueue2 = new CircularArrayQueue<Integer>();

/** load key queue */
for (int scan=0; scan < key.length; scan++)
{
    keyQueue1.enqueue (new Integer(key[scan]));
    keyQueue2.enqueue (new Integer(key[scan]));
}

/** encode message */
for (int scan=0; scan < message.length(); scan++)
{
    keyValue = keyQueue1.dequeue();
    encoded += (char) ((int)message.charAt(scan) + keyValue.intValue());
    keyQueue1.enqueue (keyValue);
}
```
The Codes class (continued)

System.out.println("Encoded Message:
" + encoded + 
"");

/** decode message */
for (int scan=0; scan < encoded.length(); scan++)
{
    keyValue = keyQueue2.dequeue();
    decoded += (char) ((int)encoded.charAt(scan) - keyValue.intValue());
    keyQueue2.enqueue (keyValue);
}

System.out.println("Decoded Message:
" + decoded);
}
UML description of the Codes program

**FIGURE 5.5** UML description of the Codes program
Ticket Counter Simulation

• Now let's use a queue to simulate the waiting line at a movie theatre
• The goal is to determine how many cashiers are needed to keep the wait time below 7 minutes
• We'll assume:
  – customers arrive on average every 15 seconds
  – processing a request takes two minutes once a customer reaches a cashier
The Customer class

```java
/**
 * Customer represents a waiting customer.
 * 
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 08/12/08
 */

public class Customer {
    private int arrivalTime, departureTime;

    /**
     * Creates a new customer with the specified arrival time.
     * 
     * @param arrives  the integer representation of the arrival time
     * @param arrivalTime = arrives;
     * @param departureTime = 0;
     */
```
The Customer class (continued)

```java
/**
 * Returns the arrival time of this customer.
 * *
 * @return  the integer representation of the arrival time
 */
public int getArrivalTime()
{
    return arrivalTime;
}

/**
 * Sets the departure time for this customer.
 * *
 * @param departs  the integer representation of the departure time
 **/
public void setDepartureTime (int departs)
{
    departureTime = departs;
}
```
The Customer class (continued)

```java
/**
 * Returns the departure time of this customer.
 * @return the integer representation of the departure time
 */
public int getDepartureTime()
{
    return departureTime;
}

/**
 * Computes and returns the total time spent by this customer.
 * @return the integer representation of the total customer time
 */
public int totalTime()
{
    return departureTime – arrivalTime;
}
```
The TicketCounter class

/**
 * TicketCounter demonstrates the use of a queue for simulating a waiting line.
 * 
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 08/12/08
 */

import jss2.*;

public class TicketCounter
{
    final static int PROCESS = 120;
    final static int MAX_CASHIERS = 10;
    final static int NUM_CUSTOMERS = 100;

    public static void main ( String[] args)
    {
        Customer customer;
        LinkedQueue<Customer> customerQueue = new LinkedQueue<Customer>();
        int[] cashierTime = new int[MAX_CASHIERS];
        int totalTime, averageTime, departs;
The TicketCounter class (continued)

```java
/** process the simulation for various number of cashiers */
for (int cashiers=0; cashiers < MAX_CASHIERS; cashiers++)
{
    /** set each cashiers time to zero initially*/
    for (int count=0; count < cashiers; count++)
        cashierTime[count] = 0;

    /** load customer queue */
    for (int count=1; count <= NUM_CUSTOMERS; count++)
        customerQueue.enqueue(new Customer(count*15));

    totalTime = 0;

    /** process all customers in the queue */
    while (!(customerQueue.isEmpty()))
    {
        for (int count=0; count <= cashiers; count++)
        {
            if (!(customerQueue.isEmpty()))
```
The TicketCounter class (continued)

```java
{ 
    customer = customerQueue.dequeue();
    if (customer.getArrivalTime() > cashierTime[count])
        departs = customer.getArrivalTime() + PROCESS;
    else
        departs = cashierTime[count] + PROCESS;
    customer.setDepartureTime (departs);
    cashierTime[count] = departs;
    totalTime += customer.totalTime();
}
}

/** output results for this simulation */
averageTime = totalTime / NUM_CUSTOMERS;
System.out.println ("Number of cashiers: " + (cashiers+1));
System.out.println ("Average time: " + averageTime + "\n");
```
UML description of the TicketCounter program

**Figure 5.6** UML description of the TicketCounter program

© 2010 Pearson Addison-Wesley. All rights reserved.
The results of the ticket counter simulation

<table>
<thead>
<tr>
<th>Number of Cashiers:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Time (sec):</td>
<td>5317</td>
<td>2325</td>
<td>1332</td>
<td>840</td>
<td>547</td>
<td>355</td>
<td>219</td>
<td>120</td>
<td>120</td>
<td>120</td>
</tr>
</tbody>
</table>

**FIGURE 5.7** The results of the ticket counter simulation
The **LinkedQueue** Class

- Like a stack, a queue can be implemented using an underlying array or a linked list.
- A linked version can use the `LinearNode` class yet again.
- In addition to keeping a reference to the beginning of the list, we will keep a second reference to the end.
- An integer `count` will keep track of the number of elements in the queue.
A linked implementation of a queue

**Figure 5.8** A linked implementation of a queue
The LinkedQueue class

```java
/**
 * LinkedQueue represents a linked implementation of a queue.
 * 
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 08/12/08
 */

package jss2;
import jss2.exceptions./*;

public class LinkedQueue<T> implements QueueADT<T>
{
    private int count;
    private LinearNode<T> front, rear;

    /**
     * Creates an empty queue.
     */
    public LinkedQueue()
    {
        count = 0;
        front = rear = null;
    }
```
**
* Adds the specified element to the rear of this queue.
*
* @param element  the element to be added to the rear of this queue
*/
public void enqueue (T element)
{
    LinearNode<T> node = new LinearNode<T>(element);

    if (isEmpty())
        front = node;
    else
        rear.setNext (node);

    rear = node;
    count++;
}
The queue after adding element E

**Figure 5.9** The queue after adding element E
/**
 * Removes the element at the front of this queue and returns a
 * reference to it. Throws an EmptyCollectionException if the
 * queue is empty.
 *
 * @return the element at the front of this queue
 * @throws EmptyCollectionException if an empty collection exception occurs
 */
public T dequeue() throws EmptyCollectionException {
    if (isEmpty())
        throw new EmptyCollectionException("queue");

    T result = front.getElement();
    front = front.getNext();
    count--;

    if (isEmpty())
        rear = null;

    return result;
}
The queue after a dequeue operation

**FIGURE 5.10** The queue after a dequeue operation
LinkedQueue – other operations

- The remaining operations in the linked queue implementation are fairly straightforward and are similar to those in the stack collection.
- The first operation is implemented by returning a reference to the element at the front of the queue.
- The isEmpty operation returns true if the count of elements is 0, and false otherwise.
- The size operation simply returns the count of elements in the queue.
- The toString operation returns a string made up of the toString results of each individual element.
Implementing Queues with arrays

- One strategy for implementing a queue would be to fix one end of the queue at position 0 of an array like we did with the ArrayStack.
- Queues, operate on both ends which would force the shifting of elements either on enqueue or dequeue.
- A better approach is to use a conceptual circular array and not fix either end of the queue.
The CircularArrayQueue Class

• If we do not fix one end of the queue at index 0, we will not have to shift the elements
• A circular queue is an implementation of a queue using an array that conceptually loops around on itself
• That is, the last index is thought to precede index 0
• We keep track of integers that indicate where the front and rear of the queue are at any given time
A circular array implementation of a queue

**Figure 5.12** A circular array implementation of a queue
A queue straddling the end of a circular array

Figure 5.13 A queue straddling the end of a circular array
Changes in a circular array implementation of a queue

**FIGURE 5.14** Changes in a circular array implementation of a queue
Circular Queues

• When an element is enqueued, the value of rear is incremented
• But it must take into account the need to loop back to 0:

  \[ \text{rear} = (\text{rear}+1) \% \text{queue.length}; \]

• Note that this array implementation can also reach capacity and may need enlarging
The CircularArrayQueue class

/**
 * CircularArrayQueue represents an array implementation of a queue in
 * which the indexes for the front and rear of the queue circle back to 0
 * when they reach the end of the array.
 * *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0 08/12/08
 */

package jss2;
import jss2.exceptions.*;
import java.util.Iterator;

public class CircularArrayQueue<T> implements QueueADT<T>
{
    private final int DEFAULT_CAPACITY = 100;
    private int front, rear, count;
    private T[] queue;
The CircularArrayQueue class

/**
 * Creates an empty queue using the default capacity.
 */
public CircularArrayQueue()
{
    front = rear = count = 0;
    queue = (T[]) (new Object[DEFAULT_CAPACITY]);
}

/**
 * Creates an empty queue using the specified capacity.
 *
 * @param initialCapacity  the integer representation of the initial size of the circular array queue
 */
public CircularArrayQueue (int initialCapacity)
{
    front = rear = count = 0;
    queue = ( (T[]) (new Object[initialCapacity]) );
}
/**
 * Adds the specified element to the rear of this queue, expanding
 * the capacity of the queue array if necessary.
 * 
 * @param element  the element to add to the rear of the queue
 */

public void enqueue (T element)
{
    if (size() == queue.length)
        expandCapacity();

    queue[rear] = element;
    rear = (rear+1) % queue.length;

    count++;
}
CircularArrayQueue – the dequeue operation

/**
 * Removes the element at the front of this queue and returns a
 * reference to it. Throws an EmptyCollectionException if the
 * queue is empty.
 * *
 * @return the reference to the element at the front
 * @throws EmptyCollectionException if an empty collections exception occurs
 */
public T dequeue() throws EmptyCollectionException {
    if (isEmpty())
        throw new EmptyCollectionException("queue");

    T result = queue[front];
    queue[front] = null;
    front = (front+1) % queue.length;
    count--;

    return result;
}
CircularArrayQueue – the expandCapacity operation

```java
/**
 * Creates a new array to store the contents of this queue with
 * twice the capacity of the old one.
 */
public void expandCapacity()
{
    T[] larger = (T[])(new Object[queue.length * 2]);

    for(int scan=0; scan < count; scan++)
    {
        larger[scan] = queue[front];
        front=(front+1) % queue.length;
    }

    front = 0;
    rear = count;
    queue = larger;
}
```
Queue Implementations

• The enqueue operation is $O(1)$ for both implementations
• The dequeue operation is $O(1)$ for linked and circular array implementations, but $O(n)$ for the noncircular array version due to the need to shift the elements in the queue