Chapter 8

Contax T

Train Queue

Queues

Learning Objectives

- Describe the operation of a queue
- Describe the characteristics of First In/First Out (FIFO) processing
- List at least four examples of applications that utilize queues
- State the type of data structure used to implement the Queue and Queue<T> classes
- Describe what it means to enqueue an item into a queue
- Describe what it means to dequeue an item into a queue
- List and describe the members of the Queue and Queue<T> classes
- Use the non-generic Queue class in a program
- Use the generic Queue<T> class in a program
- Describe the functionality provided by each interface implemented by the Queue class
- Describe the functionality provided by each interface implemented by the Queue<T> class

Introduction

Queues manifest themselves in many areas of our lives and are workhorse data structures in the areas of computers and computer science. Anytime you've waited in line for something, you've participated in queue operations. The first person to arrive in line at the bank is the first person to receive service from the next available teller. Likewise, most modern operating systems process events that have been waiting in some type of queue. (*See C# For Artists: The Art, Philosophy, and Science of Object-Oriented Programming, Chapter 12, for a discussion of the Microsoft Windows Message Queue*) In multitasking operating systems where executing threads are swapped in and out of the processor, waiting threads kick their heels in a queue until given another crack at the processor.

In this chapter I'll introduce you to the queue data structure. I'll show you how queues work and explain their characteristic operations: *enqueue* and *dequeue*. I'll then show you how a custom queue data structure might be implemented with the help of a *circular array*. Next, I'll demonstrate the use of the System.Collections.Queue and the System.Collections.Generic.Queue<T> classes.

When you finish reading this chapter you'll have a solid understanding of how queues work and understand when they're the right data structure to use in your programs.

How Queues Work

A queue is a list-based data structure whose elements are inserted at one end and removed from the other. This dual-ended operation gives the queue a First-In/First-Out (FIFO) characteristic.

Characteristic Queue Operations

If you've ever waited in line at the drive-thru you've participated in queuing operations: You arrive at one end of the line, wait your turn for service, and eventually emerge from the other end of the line when your turn at service arrives. This is how a queue operates.

Queue data structures support two primary operations: *enqueue* and *dequeue*.

Enqueue

Items are added to a queue with a call to the enque operation. Each call to enque increments the number of items in the queue by one.

Dequeue

Items are removed from a queue with a call to the dequeue operation. Each call to dequeue decrements the number of items in the queue by one.

An Illustration Will Help

Figure 8-1 shows a series of enqueue and deque operations being performed on a queue. Referring to figure 8-1 — the queue initially starts empty. Four elements are added to the queue with a series of four calls to the enqueue operation. The end of the queue increments to the next open position with each successive call to enqueue. The first call to deque removes the first item from the queue. The head of the queue increments by one with each successive call to deque. The queue is empty after the fourth call to dequeue.

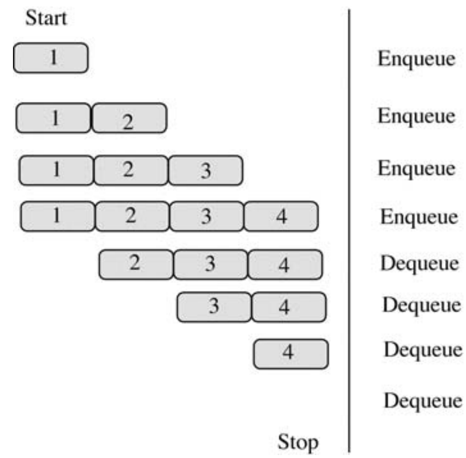

Figure 8-1: Enqueue and Dequeue Operations

Quick Review

Queues are work horse data structures in the real world as well as in the world of computers and computer science. Queues exhibit a First-In/First-Out (FIFO) characteristic; The first item to be inserted into a queue is the first item to be removed.

Queues support two primary operations: enqueue and dequeue. Items are added to a queue with a call to enqueue, and items are removed from a queue with a call to dequeue.

A Home Grown Queue Based on a Circular Array

In this section I want to show you how you might go about implementing a queue with the help of a *ring buffer* (a.k.a. circular buffer or circular array).

You can visualize a ring buffer as a circular list of elements in the shape of a ring as figure 8-2 illustrates.

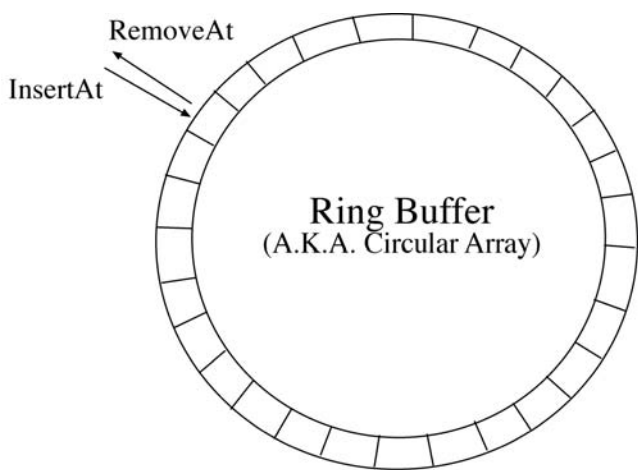

Figure 8-2: Empty Ring Buffer

Referring to figure 8-2 — initially, the ring buffer is empty and the InsertAt and RemoveAt indexes point to the same location. As elements are added to the ring buffer, the InsertAt index increments by one while the RemoveAt index remains unchanged as is shown in figure 8-3.

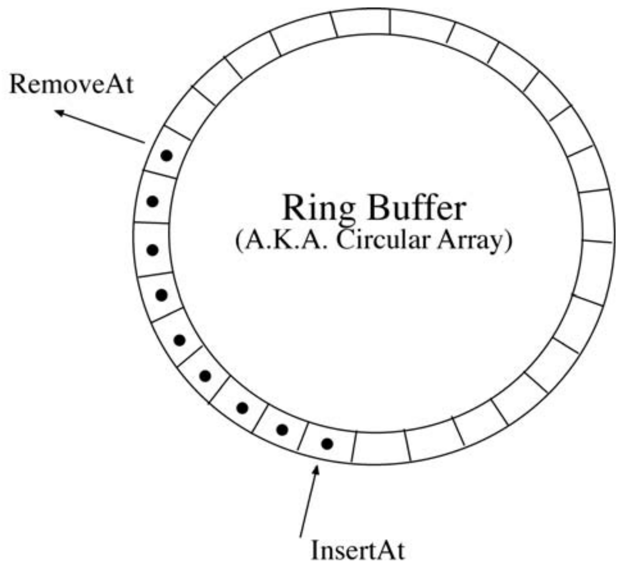

Figure 8-3: Ring Buffer After Items Have Been Inserted

In reality, memory is not circular and so a ring buffer must be implemented in terms of an ordinary array as figure 8-4 illustrates.

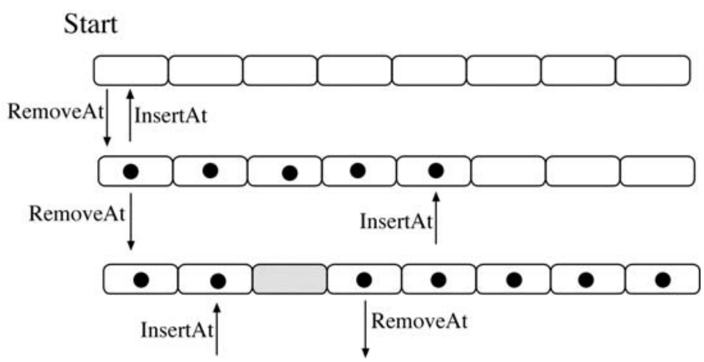

Figure 8-4: Ring Buffer Implemented with an Ordinary Array

Referring to figure 8-4 — an ordinary array has a limit to the number of elements it can hold. Initially, the array is empty and the InsertAt and RemoveAt indexes point to the same location. The insertion of elements into the array increments the InsertAt index while the RemoveAt index refers to the first item inserted into the array. When items are removed from the array, the RemoveAt index increments. If items have been removed from the array before the InsertAt index reaches the end of the array, it can be reset to point to the first element. This maximizes the use of space within the array. However, if no elements have been removed from the array by the time the InsertAt index reaches the end of the array, one of two things must happen: 1) either the insert operation must throw some type of exception indicating the array is full, or the array must be resized to accommodate additional elements. This is the approach taken with the CircularArray class given in example 8.1.

8.1 CircularArray.cs

```
using System;

public class CircularArray {
  private object[] _array = null;
  private int _insertAt = 0;
  private int _removeAt = 0;
  private int _count = 0;
  private const int INITIAL_SIZE = 25;
  private bool _debug = true;

```



```
    public bool IsEmpty {
      get { return (_count == 0)?true:false; }
    }

    public int Count {
      get { return _count; }
    }

    public CircularArray(int initial_size, bool debug) {
      _array = new object[initial_size];
      _debug = debug;
    }

    public CircularArray():this(INITIAL_SIZE, true){ }


    public void Insert(object item){
      if(item == null){
        throw new ArgumentException("Cannot insert null items!");
      }

      if((_insertAt >= _array.Length) && (_removeAt == 0)) { // we’ve inserted elements and removed none
        this.GrowArray();
      } else if((_insertAt >= _array.Length) && (_removeAt > 0)){ // There’s room at the beginning
         _insertAt = 0; // reset
         _array[_insertAt++] = item;
         _count++;
         return;
      } else if((_insertAt > 0) && (_insertAt == _removeAt)){ // Queue is full - grow and reorgansize
         this.GrowAndReorganizeArray();
      }
       _array[_insertAt++] = item;
       _count++;
    } // end Insert() method


    public object Remove(){
      if(_count == 0){
        throw new InvalidOperationException("Array is empty!");
      }
      if(_removeAt >= _array.Length){
        _removeAt = 0;
      }
      object return_object = _array[_removeAt];
      _array[_removeAt++] = null;

      _count--;
      if((_count == 0) && (_removeAt == _insertAt)){ // reset insertion and removal points
        _removeAt = 0;
        _insertAt = 0;
      }
      return return_object;
    } // end Remove() method


    public object Peek(){
      if(_count == 0){
        throw new InvalidOperationException("Array is empty!");
      } else {
        return _array[_removeAt];
      }
    } // end Peek() method


    private void GrowArray(){
      if(_debug){
        Console.WriteLine("-----Entering GrowArray Method------");
      }
      object[] temp_array = new object[_array.Length];
      for(int i = 0; i < _array.Length; i++){
        temp_array[i] = _array[i];
      }

      _array = new object[_array.Length * 2]; // double the size of the array

      for(int i = 0; i < temp_array.Length; i++){
        _array[i] = temp_array[i];
      }

      if(_debug){
        Console.WriteLine("-----Leaving GrowArray Method------");
```



```
92          }
93        } // end GrowArray() method
94
95        private void GrowAndReorganizeArray(){
96          if(_debug){
97            Console.WriteLine("-----Entering GrowAndReorganizeArray Method------");
98          }
99
100         object[] temp_array = new object[_array.Length];
101         for(int i = 0; i < _array.Length; i++){
102           temp_array[i] = _array[i];
103         }
104
105         int old_length = _array.Length;
106
107         _array = new object[old_length * 2]; // double the size of the array
108
109         int j = 0;
110         for(int i = _removeAt; i < old_length; i++){
111           _array[j++] = temp_array[i];
112         }
113
114         for(int i = 0; i < _insertAt; i++){
115           _array[j++] = temp_array[i];
116         }
117
118         _removeAt = 0;
119         _insertAt = _count;
120
121         if(_debug){
122           Console.WriteLine("-----Leaving GrowAndReorganizeArray Method------");
123         }
124       }
125
126
127     } // end CircularArray class definition
```

Referring to example 8.1 — an array of objects named _array serves as the basis for the circular array. The fields _insertAt, _removeAt, and _count are used to manage the internal state of the circular array. When a CircularArray object is created, its initial size can be specified via the constructor, or, if the default constructor is called, its initial size will be set to 25 elements.

The Insert() method starts on line 27 and checks first to ensure that inserted elements are not null. Next, the `if` statement on line 32 checks to see if any elements have been removed from the array. If not, the array is full and it must be expanded to hold more elements. This is accomplished with a call to the GrowArray() method.

If there's room at the beginning of the array because the _removeAt index has been incremented, elements are inserted there. If, however, the _insertAt and _removeAt indexes are equal, then the array is full and must be expanded as well as reorganized before new elements can be added. This is accomplished with a call to the GrowAndReorganizeArray() method.

The Remove() method starts on line 47. If the array is empty, the method throws an InvalidOperationException, otherwise, the next object in the array is returned and the _removeAt index is incremented by one. When the value of _removeAt approaches the value of the length of the array, it's reset to zero. If, after the removal of an element, the number of elements in the array equals zero, both the _insertAt and _removeAt indexes are reset to zero.

In this fashion, the CircularArray class expands its array to hold additional elements. When necessary, it can both expand and reorganize its array.

Example 8.2 gives the code for the HomeGrownQueue class whose functionality is based on the CircularArray class.

8.2 HomeGrownQueue.cs

```
using System;

public class HomeGrownQueue {
  private CircularArray _ca = null;
  private const int INITIAL_SIZE = 25;

  public HomeGrownQueue(int initial_size, bool debug){
    _ca = new CircularArray(initial_size, debug);
  }

  public HomeGrownQueue():this(INITIAL_SIZE, true){ }

  public bool IsEmpty {
    get { return _ca.IsEmpty; }
  }
```



```
  public int Count {
    get { return _ca.Count; }
  }

  public void Enqueue(object item){
    try{
      _ca.Insert(item);
    }catch(Exception){
      Console.WriteLine("Cannot enqueue null item!");
    }
  }

  public object Dequeue(){
    object return_object = null;
    try{
     return_object = _ca.Remove();
    }catch(Exception){
      throw new InvalidOperationException("Queue is empty!");
    }
    return return_object;

  }

  public object Peek(){
    object return_object;
    try{
     return_object = _ca.Peek();
    }catch(Exception){
      throw new InvalidOperationException("Queue is empty!");
    }
    return return_object;
  }

}
```

Referring to example 8.2 — the implementation of HomeGrownQueue is easy since the heavy lifting is done by the CircularArray class. The HomeGrownQueue implements a façade software design pattern and simply acts as a wrapper for the CircularArray class providing the Enqueue() and Dequeue() methods you expect from a queue. It also provides an IsEmpty property and a Peek() method, which in turn calls the CircularArray.Peek() method.

Example 8.3 gives the code for a MainApp class that puts the HomeGrownQueue through its paces.

8.3 MainApp.cs (Demonstrating HomeGrownQueue)

```
using System;

public class MainApp {
  public static void Main(){
   HomeGrownQueue queue = new HomeGrownQueue(); // default size of 25 elements

    for(int i = 0; i < 40; i++){ // test Growth Capability
      queue.Enqueue(i);
    }

    Console.WriteLine("Count = {0}", queue.Count);

    int itemsInQueue = queue.Count;

    Console.WriteLine("Next item to be removed from queue is: {0}", queue.Peek());

    for(int i = 0; i < itemsInQueue ; i++) {
      Console.Write(queue.Dequeue() + " ");
    }

    Console.WriteLine();

    try{
     queue.Dequeue(); // try to remove one more element

    }catch(Exception e){
      Console.WriteLine(e);
    }

   queue = new HomeGrownQueue(); // start again with 25 elements

   for(int i = 0; i < 23; i++){
     queue.Enqueue(i);
   }

   for(int i = 0; i < 10; i++){
     Console.Write(queue.Dequeue() + " ");
```

```
38         }
39         Console.WriteLine();
40
41         queue.Enqueue(23);
42         queue.Enqueue(24);
43         queue.Enqueue(25);
44         queue.Enqueue(26);
45         queue.Enqueue(27);
46         queue.Enqueue(28);
47         queue.Enqueue(29);
48         queue.Enqueue(30);
49         queue.Enqueue(31);
50         queue.Enqueue(32);
51         queue.Enqueue(33);
52         queue.Enqueue(34);
53         queue.Enqueue(35);
54         queue.Enqueue(36);
55         queue.Enqueue(37);
56         queue.Enqueue(38);
57         queue.Enqueue(39);
58         queue.Enqueue(40);
59         queue.Enqueue(41);
60         for(int i = 42; i < 134; i++){
61           queue.Enqueue(i);
62         }
63
64         for(int i = 0; i< 83; i++){
65           Console.Write(queue.Dequeue() + " ");
66         }
67
68         for(int i = 134; i < 289; i++){
69           queue.Enqueue(i);
70         }
71
72         Console.WriteLine("Count = " + queue.Count);
73
74         while(queue.Count > 0){
75           Console.Write(queue.Dequeue() + " ");
76         }
77
78         Console.WriteLine("Count = " + queue.Count);
79       }
80     }
```

Referring to example 8.3 — on line 5, an instance of HomeGrownQueue is created with the default constructor which creates an internal array in the CircularArray class with 25 elements. The for statement on line 7 tests the growth capability by creating and inserting 40 integers into the queue. The rest of the program enqueues and dequeues various numbers of integers to trigger both the growth and grow and reorganize capability. Figure 8-5 shows the results of running this program.

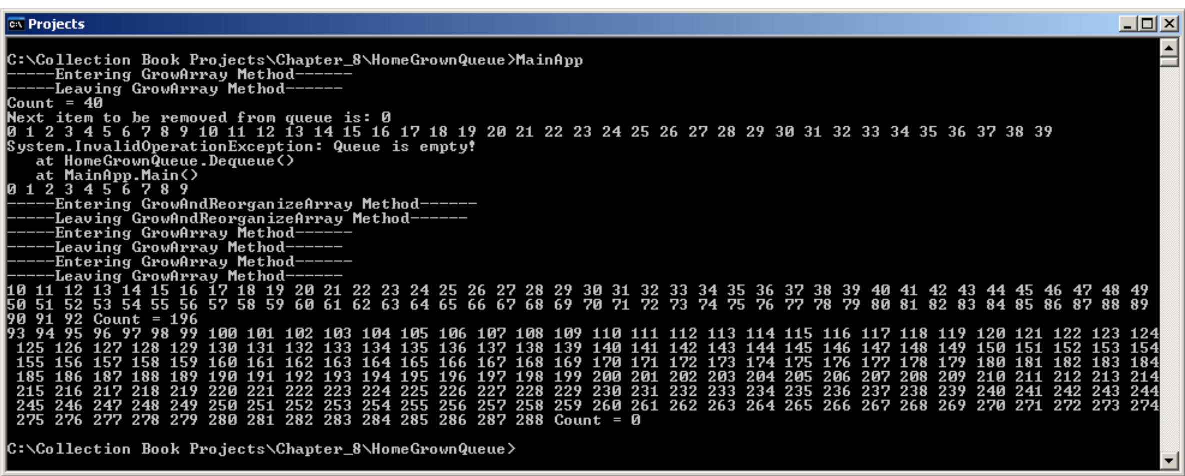

Figure 8-5: Results of Running Example 8.3

Quick Review

A ring buffer (a.k.a. circular array) can serve as the foundational data structure for a queue class. When implementing a circular array, you must carefully manage the insert and remove indexes to gain maximum space efficiency. The CircluarArray class implements an internal array growth capability as well as a grow and reorganize capability. The HomeGrownQueue class serves as a wrapper class for the CircularArray class.

The Queue Class

The System.Collections.Queue class is a non-generic collection class. It uses a circular array to implement queue functionality. And since it's a collection class, it has a lot more functionality that the HomeGrownQueue presented in the previous section.

Queue Class Inheritance Hierarchy

Figure 8-6 gives a UML class diagram of the Queue class.

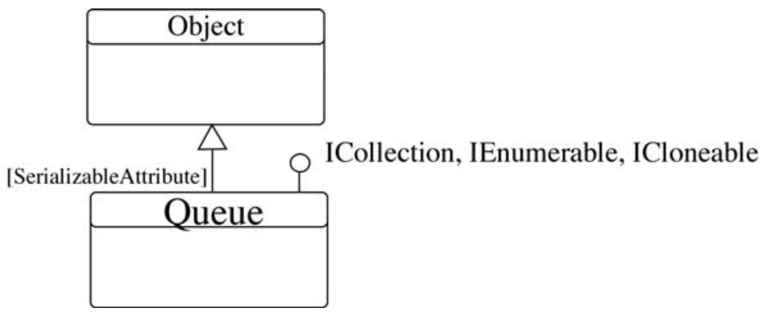

Figure 8-6: Queue Class Inheritance Hierarchy

Referring to figure 8-6 — the Queue class extends Object and implements ICollection, IEnumerable, and ICloneable. The following sections describe in more detail the purpose of each of these interfaces. It is also serializable.

Functionality Provided by the IEnumerable Interface

The IEnumerable interface, along with the supporting IEnumerator interface, enables you to iterate over the elements in the queue using the `foreach` statement. The direction of iteration begins with the queue's first, or oldest, element and ends with the last element inserted, or youngest, element in the queue.

Functionality Provided by the ICollection Interface

The ICollection interface inherits from IEnumerable and provides a CopyTo() method that can be used to copy the elements contained in the queue to an array. The ICollection interface also provides the Count, IsSynchronized, and SyncRoot properties. The Count property returns the number of elements contained in the collection. The IsSynchronized and SyncRoot properties are used in conjunction with multithreading programming techniques which is discussed in detail in Chapter 13 — Thread Programming.

Functionality Provided by the ICloneable Interface

The ICloneable interface exposes the Clone() method which is used to make a shallow copy of the queue.

Palindrome Checker

A palindrome is a sequence of characters of numbers that can be read the same way in both directions. For example, the word “Eve” is a palindrome as is the phrase, “Madam, I’m Adam!” Example 8.4 gives the code for a program that uses both a queue and a stack to read a sequence of characters and determine if they form a palindrome.

8.4 PalindromeTester.cs

```
using System;
using System.Collections;

public class PalindromeTester {

  private int _letterCount;
  private int _checkedCharacters;
  private char _fromStack;
  private char _fromQueue;
  private bool _stillPalindrome;
  private Stack _stack;
  private Queue _queue;

  // Default constructor - Nothing to do, really
  public PalindromeTester(){ }

  public bool Test(String inputString){

    _stack = new Stack();
    _queue = new Queue();
    _letterCount = 0;

    foreach(char c in inputString){
      if(Char.IsLetter(c)){
        _letterCount++;
        char _c = Char.ToLower(c);
        _stack.Push(_c);
        _queue.Enqueue(_c);
      }
    }

    _stillPalindrome = true;
    _checkedCharacters = 0;
    while(_stillPalindrome && (_checkedCharacters < _letterCount)){
      _fromStack = (char)_stack.Pop();
      _fromQueue = (char)_queue.Dequeue();
      if(_fromStack != _fromQueue){
        _stillPalindrome = false;
      }
      _checkedCharacters++;
    }
    return _stillPalindrome;
  }

  public static void Main(){

     PalindromeTester pt = new PalindromeTester();
     string input_string = String.Empty;
     while(true){
       Console.Write("Please enter a possible palindrome for testing or \"Quit\" to exit: ");
       input_string = Console.ReadLine();

       if(input_string == "Quit") {
         return;
       }

       if(pt.Test(input_string)){
         Console.BackgroundColor = ConsoleColor.DarkBlue;
         Console.Beep(400, 600);
         Console.WriteLine("YES!!! \"{0}\" is a palindrome!", input_string);
         Console.BackgroundColor = ConsoleColor.Black;
         Console.ResetColor();
       }else{
         Console.BackgroundColor = ConsoleColor.Red;
         Console.Beep(78, 3000);
         Console.WriteLine("Sorry, \"{0}\" is not a palindrome...", input_string);
         Console.ResetColor();
       }
     }
  }
}
```


Referring to example 8.4 — the PalindromeTester class uses a queue and a stack, both from the System.Collections namespace, to test character sequences. The real work is performed by the Test() method, which begins on line 17. The method initializes the stack and the queue and sets the _letterCount field to 0. The `foreach` statement on line 23 iterates over the input string. If the character is a letter, it increments _letterCount by 1, converts it to lower case, and pushes it on the stack. Characters that aren't letters are simply ignored. Remember, pushing the letters onto the stack has the effect of reversing the string.

The palindrome testing begins on line 32. The `while` statement on line 34 pops a character off the stack and dequeues a character from the queue and compares the two. If they match, it continues checking. If a match fails, then the sequence was not a palindrome. Note that when characters are popped off the stack and dequeued from the queue, they must be cast to their proper type before the comparison can be made.

The Main() method begins on line 45. It creates and instance of PalindromeTester and then executes the while loop on line 49 which repeatedly asks for input from the users until they enter the string "Quit".

Figure 8-7 shows the results of running this program.

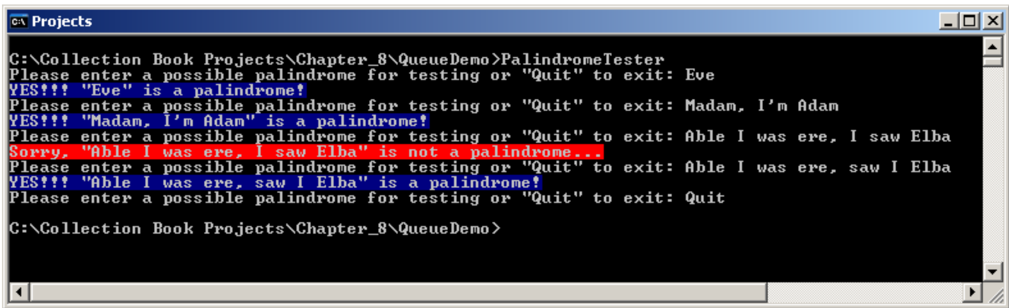

Figure 8-7: Results of Running Example 8.4

Referring to figure 8-7 — when the user enters a palindrome, the console background color is set to blue and a congratulatory message is written to the console. If the string is not a palindrome, the background color is set to red and the user receives the "Sorry..." message. How many palindromes can you think of?

Quick Review

A circular array is used to implement the System.Collections.Queue class. The Queue class extends System.Object and implements the IEnumerable, ICollection, and ICloneable interfaces. It is also serializable. Because it is a non-generic class, objects inserted into the queue must be cast to their proper type when they are dequeued.

The Queue<T> Class

The System.Collections.Generic.Queue<T> class is the generic version of the Queue class. The benefit to using the Queue<T> class is, among other things, not having to cast objects when they are dequeued.

Queue<T> Class Inheritance Hierarchy

Figure 8-8 gives the UML class diagram of the Queue<T> class inheritance hierarchy. Referring to figure 8-8 — the Queue<T> class extends System.Object and implements the ICollection, IEnumerable<T>, and IEnumerable interfaces. It is also serializable. It directly implements the ICollection<T> interface, just like the Stack<T> class does. The following sections describe the functionality provided by each of these interfaces.

Functionality Provided by the ICollection Interface

The ICollection interface inherits from IEnumerable and provides a CopyTo() method that can be used to copy the elements contained in the queue to an array. The ICollection interface also provides the Count, IsSynchronized,

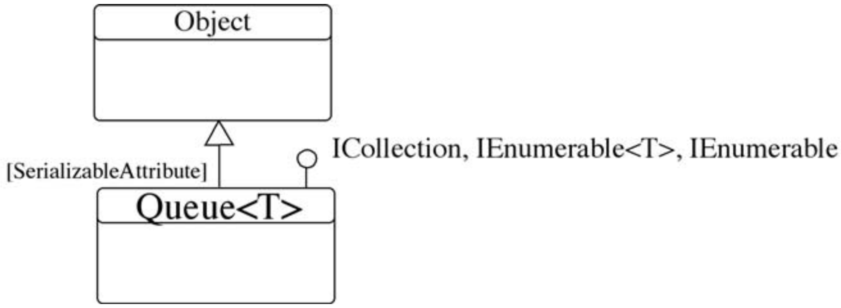

Figure 8-8: Queue<T> Class Inheritance Hierarchy

and SyncRoot properties. The Count property returns the number of elements contained in the collection. The IsSynchronized and SyncRoot properties are used in conjunction with multithreading programming techniques which is discussed in detail in Chapter 13 — Thread Programming.

Functionality Provided by the IEnumerable Interface

The IEnumerable interface, along with the supporting IEnumerator interface, enables you to iterate over the elements in the queue using the `foreach` statement. The direction of iteration begins with the queue's last inserted, or oldest, element and ends with the most recently inserted, or youngest, element in the queue.

Functionality Provided by the IEnumerable<T> Interface

The IEnumerable<T> interface extends IEnumerable and allows the elements of the generic Queue<T> class to be enumerated by the `foreach` statement.

What Happened to ICollection<T>?

The Queue<T> class is one of two generic collection classes that do not explicitly implement the ICollection<T> interface, rather, it directly implements a few of its methods in the interest of providing specialized control over access to collection elements. For example, you can only add elements to a Queue<T> class via its Enqueue() method and only remove elements via the Dequeue() and Clear() methods.

Store Simulation

Queues come in hand when programming simulations of service scenarios. The code in example 8.5 uses a Queue<T> object to store DateTime objects in the simulation of a store checkout line with one checker. As with most simulations of this type, the primary concern purpose is to calculate the average and maximum waiting times.

8.5 StoreSimulation.cs

```
using System;
using System.Collections.Generic;

public class StoreSimulation {

  private Queue<DateTime> _queue;
  private TimeSpan _simulationRunTime;
  private int _totalServed;
  private TimeSpan _totalWaitingTime;
  private TimeSpan _maximumWaitingTime;
  private Random _rand;

  public StoreSimulation(int runtimeMinutes){
    _queue = new Queue<DateTime>();
    _simulationRunTime = new TimeSpan(0, runtimeMinutes, 0);
    _rand = new Random();
  }

  public StoreSimulation():this(10){}
```



```
20
21        public void Go(){
22          DateTime startTime = DateTime.Now;
23          Console.WriteLine("Simulation started at: {0}", startTime);
24          while((DateTime.Now - startTime) < _simulationRunTime){
25
26            if((_queue.Count > 0) && ((_rand.Next() % 6) == 3)){
27              DateTime t = _queue.Dequeue();
28              TimeSpan ts = DateTime.Now - t;
29              _totalServed++;
30              _totalWaitingTime += ts;
31              Console.Write("P");
32              if(_maximumWaitingTime < ts){
33                _maximumWaitingTime = ts;
34              }
35            }
36
37            switch(_rand.Next() % 4){
38              case 1 : _queue.Enqueue(DateTime.Now);
39                      Console.Beep();
40                      Console.ForegroundColor = ConsoleColor.Yellow;
41                      Console.Write(".");
42                      Console.ResetColor();
43                       break;
44              case 2 : _queue.Enqueue(DateTime.Now);
45                       _queue.Enqueue(DateTime.Now);
46                       Console.Beep(88, 200);
47                       Console.ForegroundColor = ConsoleColor.Blue;
48                       Console.Write("..");
49                       Console.ResetColor();
50                       break;
51              default: break;
52            }
53
54          }
55
56          // Print Statistics
57
58          Console.WriteLine();
59          Console.WriteLine("--------Simulation Complete -----------");
60          Console.WriteLine("Simulation ended at: {0}", DateTime.Now);
61          Console.WriteLine("Customers served: {0}", _totalServed);
62          Console.WriteLine("Average wait time: {0}", (double)_totalWaitingTime.Minutes/_totalServed);
63          Console.WriteLine("Longest wait time: {0}", _maximumWaitingTime);
64          Console.WriteLine("Customers still in line: {0}", _queue.Count);
65        }
66
67        public static void Main(){
68          StoreSimulation ss = new StoreSimulation(5);
69          ss.Go();
70        } // end Main method
71      } // End StoreSimulation Class Definition
```

Referring to example 8.5 — the bulk of the processing takes place inside the Go() method, which begins on line 21. The simulation will run for the amount of time specified in minutes via the constructor. The Go() method writes the simulation start time to the console. The while statement beginning on line 24 processes the simulation for the duration of the runtime. If the queue contains waiting objects, and the checker is free (determined with the calculation (_rand.Next() % 6) == 3)), a DateTime object is dequeued and processed.

The switch statement on line 37 is used to generate new "customers" either one at a time or two at a time. To track the generation and processing of customers I set the console foreground color to different colors to signify different generation and processing events. This makes the simulation easier and more fun to follow as it runs.

Finally, the short Main() method on line 57 creates an instance of StoreSimulation with a simulation runtime of 5 minutes and calls the Go() method to start processing. Figure 8-9 shows the results of running this program.

Quick Review

The Queue<T> class extends System.Object and implements the IEnumerable, IEnumerable<T>, and ICollection interfaces. It implements the ICollection<T> interface directly to limit insertion of objects into the queue via the Enqueue() method and the removal of objects via the Dequeue() and Clear() methods.

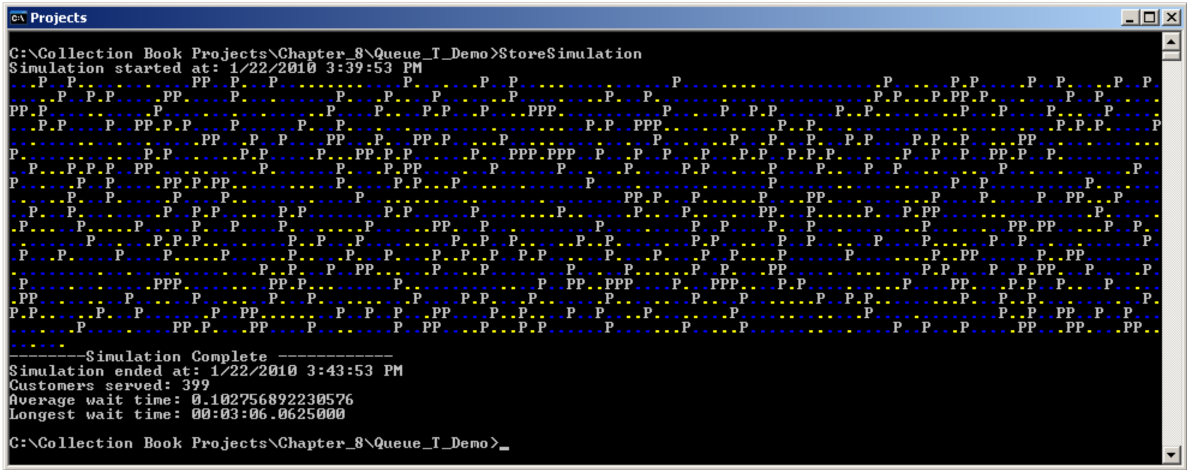

Figure 8-9: Results of Running Example 8.5

Summary

Queues are work horse data structures in the real world as well as in the world of computers and computer science. Queues exhibit a First-In/First-Out (FIFO) characteristic; The first item to be inserted into a queue is the first item to be removed.

Queues support two primary operations: enqueue and dequeue. Items are added to a queue with a call to enqueue, and items are removed from a queue with a call to dequeue.

A ring buffer (a.k.a. circular array) can serve as the foundational data structure for a queue class. When implementing a circular array, you must carefully manage the insert and remove indexes to gain maximum space efficiency. The CircluarArray class implements an internal array growth capability as well as a grow and reorganize capability. The HomeGrownQueue class serves as a wrapper class for the CircularArray class.

A circular array is used to implement the System.Collections.Queue class. The Queue class extends System.Object and implements the IEnumerable, ICollection, and ICloneable interfaces. It is also serializable. Because it is a non-generic class, objects inserted into the queue must be cast to their proper type when they are dequeued.

The Queue<T> class extends System.Object and implements the IEnumerable, IEnumerable<T>, and ICollection interfaces. It implements the ICollection<T> interface directly to limit insertion of objects into the queue via the Enqueue() method and the removal of objects via the Dequeue() and Clear() methods.

References

Donald E. Knuth. The Art of Computer Programming, Vol. 1, Fundamental Algorithms. Third Edition. Addison-Wesley, Reading, Massachusetts. 1997. ISBN: 0-201-89683-4.

Microsoft Developer Network (MSDN) [http://www.msdn.com]

Notes