Chapter 9

Contax T

Concrete Stairs

HashTables and Dictionaries

Learning Objectives

- *Describe the purpose and use of a hashtable*
- *Describe the purpose and use of a dictionary*
- *Describe the purpose and use of key/value pairs*
- *Describe the purpose of a key*
- *List and describe the interfaces a class must implement if it's to be used as a key*
- *Describe the functionality provided by the IDictionary interface*
- *Use the Hashtable class in a program*
- *Use the Dictionary<TKey, TValue> class in a program*
- *Extract the key collection from a Hashtable or Dictionary*
- *Extract the value collection from a Hashtable or Dictionary*

Introduction

Many times in your programming career you'll want to store an object in a collection and access that object via a unique *key*. This is the purpose of hashtables and dictionaries. Both hashtables and dictionaries perform the same type of service storing key/value pairs, but function differently on the insides. The Hashtable is the non-generic class and Dictionary<TKey, TValue> is its generic replacement.

In this chapter, I'm going to explain how hashtables work. Along the way I'll introduce you to concepts like *buckets*, *hash functions*, *keys*, *values*, *collisions*, and *growth factors*. I'll start the discussion with an overview of how hashtables work followed by a detailed demonstration of the operation of a chained hashtable with the help of a comprehensive coding example. In the end, you'll be able to create your own hashtable class, but that's the beauty of the .NET collections framework. You won't have to!

A few topics I'd like to cover in this chapter I'm going to put off until Chapter 10 — Coding For Collections. These include how to create immutable objects, how to write a custom comparer, and how to write a case insensitive hashcode provider, among others. You don't need to know these specialized topics just yet to fully use the Hashtable or Dictionary<TKey, TValue> classes. You will, however, need to read chapter 10 before you can use your own custom developed classes as hashtable or dictionary keys.

So, in the absence of violent objection, let's get started.

How a Hashtable Works

A hashtable is an array that employs a special operation called a *hash function* to calculate an object's location within the array. The object being inserted into the table is referred to as the *value*. A hash function is applied to the value's associated *key* which results in an integer value that lies somewhere between the first array element (0) and the last array element (n-1). Figure 9-1 offers a simple illustration of a hashtable and a hash function being applied to a key.

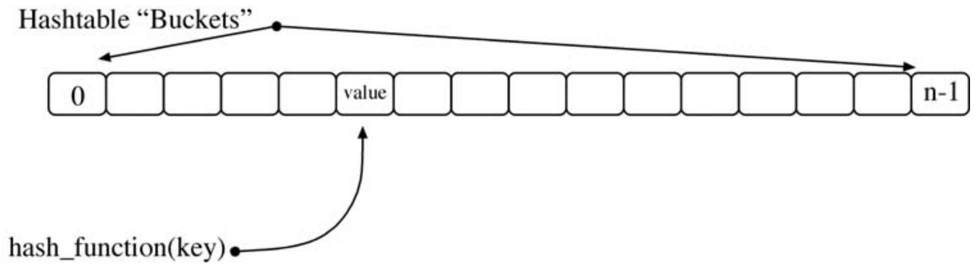


Figure 9-1: Typical Hashtable Showing Hash Function Being Applied to a Key

Referring to figure 9-1 — the hashtable elements are referred to as *buckets*. The hash function transforms the key into an integer value that falls within the bounds of the array. Into this location the key's corresponding object is placed. All subsequent hashtable accesses using that particular key will "hash" to the same location.

Hashtable Collisions

A potential problem with hashtables arises when the hash function calculates the same hash value for two different keys. When this happens a *collision* is said to have occurred. There are several ways to resolve hashtable collisions and their complete treatment here is beyond the scope of this book, but if you're interested, I recommend you refer to Donald Knuth's excellent treatment of the subject. (Donald Knuth, The Art of Computer Programming, Vol. 3, Sorting and Searching, Second Edition)

I will, however, discuss and demonstrate one collision resolution strategy referred to as *chaining*. Chaining can be used to resolve hashtable collisions by storing values whose keys have hashed to the same bucket in a chain of elements. See figure 9-2.

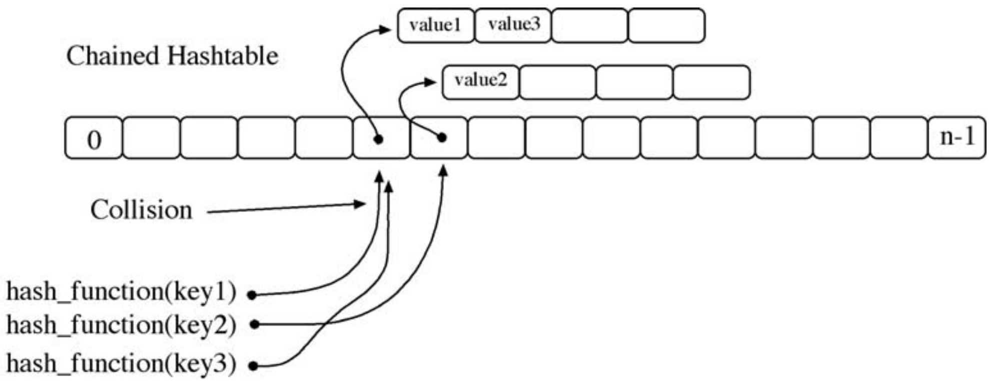


Figure 9-2: Collision Resolution within a Chained Hashtable

Referring to figure 9-2 — collisions occur when two different keys hash to the same bucket location. The corresponding values are stored in a linked list or similar structure.

Collisions can be kept to a minimum if the hash function is especially good, meaning that for a large number of different keys the resulting hash values are uniformly distributed over the range of available bucket values. Also, the size of the hashtable can be increased as the number of keys grows to decrease the likelihood of a collision. These concepts are demonstrated in the following section.

HomeGrownHashTable

The HomeGrownHashtable class presented in this section implements a complete example of a chained hashtable. The complete example consists of the KeyValuePair structure, presented in example 9.1, and the HomeGrownHashtable class, which is presented in example 9.2.

9.1 KeyValuePair.cs

```
using System;

public struct KeyValuePair : IComparable<KeyValuePair>, IComparable<string> {

  private string _key;
  private string _value;

  public KeyValuePair(string key, string value){
    _key = key;
    _value = value;
  }

  public string Key {
    get { return _key; }
    set { _key = value; }
  }

  public string Value {
    get { return _value; }
    set { _value = value; }
  }

  public int CompareTo(KeyValuePair other){
    return this._key.CompareTo(other.Key);
  }

  public int CompareTo(string other){
    return this._key.CompareTo(other);
  }
} // End KeyValuePair class definition
```

Referring to example 9.1 — the KeyValuePair structure represents the key/value pair that will be inserted into the hashtable. It defines two private fields named _key and _value and their corresponding public properties. The KeyValuePair structure also implements two interfaces: IComparable<KeyValuePair> and IComparable<string>. It implements these interfaces by defining two methods named CompareTo(KeyValuePair other) and CompareTo(string other). The implementation of the IComparable interfaces is necessary if KeyValuePair objects are to be used in sorting and searching operations. For a more detailed discussion on how to prepare your custom structures and classes for use in collections see Chapter 10 — Coding for Collections.

Example 9.2 lists the code for the HomeGrownHashtable class. It uses the KeyValuePair structure as part of its implementation.

9.2 HomeGrownHashtable.cs

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Text;

public class HomeGrownHashtable {

  private float _loadFactor = 0;
  private List<string> _keys = null;
  private List<KeyValuePair>[] _table = null;
  private int _tableSize = 0;
  private int _loadLimit = 0;
  private int _count = 0;

  private const float MIN_LOAD_FACTOR = .65F;
  private const float MAX_LOAD_FACTOR = 1.0F;


  // Constructor methods
  public HomeGrownHashtable(float loadFactor, int initialSize){
    if((loadFactor < MIN_LOAD_FACTOR) || (loadFactor > MAX_LOAD_FACTOR)){
      Console.WriteLine("Load factor must be between {0} and {1}." +
                        "Load factor adjusted to {1}", MIN_LOAD_FACTOR, MAX_LOAD_FACTOR);
      _loadFactor = MAX_LOAD_FACTOR;
    } else {
      _loadFactor = loadFactor;
      }

    _keys = new List<string>();
    _tableSize = this.DoublePrime(initialSize/2);
    _table = new List<KeyValuePair>[_tableSize];
    _loadLimit = (int)Math.Ceiling(_tableSize * _loadFactor);

    for(int i = 0; i<_table.Length; i++){
      _table[i] = new List<KeyValuePair>();
    }
  } // end constructor


  public HomeGrownHashtable():this(MAX_LOAD_FACTOR, 6){  }


  public string[] Keys {
    get { return _keys.ToArray(); }
  } // end Keys property


  public void Add(string key, string value){
    if((key == null) || (value == null)){
      throw new ArgumentException("Key and Value cannot be null.");
    }

    string upperCaseKey = key.ToUpper();

    if(_keys.Contains(upperCaseKey)){
      throw new ArgumentException("No duplicate keys allowed in HomeGrownHashtable!");
    }

    _keys.Add(upperCaseKey);
    _keys.Sort();
    int hashValue = this.GetHashValue(upperCaseKey);
    KeyValuePair item = new KeyValuePair(upperCaseKey, value);
    _table[hashValue].Add(item);
    _table[hashValue].Sort();

    if((++_count) >= _loadLimit){
      this.GrowTable();
    }
  }// end Add() method


  public string Remove(string key){
    if(key == null){
      throw new ArgumentException("Key string cannot be null!");
    }

    string upperCaseKey = key.ToUpper();

    if(_keys.Contains(upperCaseKey)){
      _keys.Remove(upperCaseKey);
    } else {
      throw new ArgumentException("Key does not exist in table!");
    }

    _keys.Sort();
    int hashValue = this.GetHashValue(upperCaseKey);
    string return_value = string.Empty;
    for(int i = 0; i<_table[hashValue].Count; i++){
      if(_table[hashValue][i].Key == upperCaseKey){
        return_value = _table[hashValue][i].Value;
        _table[hashValue].RemoveAt(i);
        _table[hashValue].Sort();
        break;
      }
    }

    return return_value;
  } // end Remove() method


  private void GrowTable(){
    List<KeyValuePair>[] temp = new List<KeyValuePair>[_table.Length];
    for(int i=0; i<_table.Length; i++){
      temp[i] = _table[i];
    }

    _table = new List<KeyValuePair>[this.DoublePrime(_tableSize)];

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

    for(int i=temp.Length; i<_table.Length; i++){
      _table[i] = new List<KeyValuePair>();
    }
  } // end GrowTable() method


  public string this[string key]{
    get {
          if((key == null) || (key == string.Empty)){
            throw new ArgumentException("Index key value cannot be null or empty!");
          }
          string return_value = string.Empty;
          string upperCaseKey = key.ToUpper();
          if(_keys.Contains(upperCaseKey)){
            int hashValue = this.GetHashValue(upperCaseKey);
            for(int i = 0; i<_table[hashValue].Count; i++){
              if(_table[hashValue][i].Key == upperCaseKey){
                return_value = _table[hashValue][i].Value;
                break;
              }
            }
          }
          return return_value;
        }

    set {
         if((key == null) || (key == string.Empty)){
            throw new ArgumentException("Index key value cannot be null or empty!");
          }

          if((value == null) || (value == string.Empty)){
            throw new ArgumentException("String value cannot be null or empty!");
          }
         string upperCaseKey = key.ToUpper();
          if(_keys.Contains(upperCaseKey)){
            int hashValue = this.GetHashValue(upperCaseKey);
            for(int i = 0; i<_table[hashValue].Count; i++){
              if(_table[hashValue][i].Key == upperCaseKey){
               KeyValuePair kvp = new KeyValuePair(upperCaseKey, value);
               _table[hashValue].RemoveAt(i);
               _table[hashValue].Add(kvp);
               _table[hashValue].Sort();
                break;
              }
            }
          }
    }
  } // end indexer


  private int DoublePrime(int currentPrime){
    currentPrime *= 2;
    int limit = 0;
    bool prime = false;
    while(!prime){
      currentPrime++;
      prime = true;
      limit = (int)Math.Sqrt(currentPrime);
      for(int i = 2; i<=limit; i++){
        if((currentPrime % i) == 0){
        prime = false;
        break;
        }
      }
    }
    return currentPrime;
  } // end DoublePrime() method


  private int GetHashValue(string key){
    int hashValue = ( Math.Abs(key.GetHashCode()) % _tableSize);
    return hashValue;
  } // end GetHashValue() method


  public void DumpContentsToScreen(){
    foreach(List<KeyValuePair> element in _table){
      foreach(KeyValuePair kvp in element){
        Console.Write(kvp.Value + " ");
      }
      Console.WriteLine();
    }
  } // end DumpContentsToScreen() method
} // end class definition
```

Referring to example 9.2 — the HomeGrownHashtable contains a List<string> object named _keys into which incoming key values are insert for future reference, and for the main table, a List<KeyValuePair> array named _table. The other fields include _tableSize, _loadFactor, _loadLimit, and _count. The *load factor* is used to calculate the *load limit*. In HomeGrownHashtable, the load factor is allowed to range between .65 and 1. When items are inserted into the hashtable, the calculated load limit is compared to the item count and if necessary, the main table is expanded to hold more elements.

The HomeGrownHashtable class defines the following methods: two constructors — one that does all the heavy lifting and a default constructor; Add(), Remove(), GrowTable(), GetHashValue(), DoublePrime(), and DumpContentsToScreen(). It also defines a Keys property and an indexer which allows values to be retrieved via their keys using familiar array notation. (i.e. Hashtable_Reference[“key”])

Let’s step through the operation of the Add() method. The Add() method takes two arguments: a key string and a value string. The incoming arguments are checked for null values and if either are null the method throws an ArgumentException. The incoming key is converted to upper case with the String.ToUpper() method. The method then searches the _keys list to see if the key has already been inserted into the hashtable. If so, no duplicate keys are allowed and the method throws an ArgumentException. If the key is not in the _keys list, it’s added to the list and the _keys list is then sorted. The key is then used to generate a hash value with the help of the GetHashValue() method. A new KeyValuePair object is created and added to the list at the _table[hashValue] location. That list is then sorted. The _count field is incremented and if necessary, the _table is expanded to hold additional elements by a call to the GrowTable() method.

Let’s now examine the GrowTable() method. As its name implies, the purpose of the GrowTable() method is to grow the main hashtable (_table) to accommodate additional elements. The table growth mechanism is triggered in the Add() method when the element count (_count) approaches the hashtable’s calculated load limit. The load limit (_loadLimit) is calculated in the body of the constructor: _loadLimit = (int)Math.Ceiling(_tableSize * _loadFactor); The first order of business in the GrowTable() method is to create a temporary List<KeyValuePair> array named temp and copy all the existing elements from _table to temp. A new array of List<KeyValuePair> elements is created double the size of the existing table rounded up to the nearest prime number. This is done with the help of the DoubleP-rime() method. The reason I did this was because in my initial version of this example I used a custom hash function which relied on the generation of prime numbers to calculate the hash value of the key. I left the DoublePrime() method in the code so you can experiment with different hash generation techniques, most of which rely on prime numbers. (Note: The DoublePrime() method replaces the usual approach of maintaining an array of precalculated prime numbers.)

The GetHashValue() method calculates a hash value based on the key. Since I'm using strings as keys, I decided to rely on the GetHashCode() method defined by the String class. This value is then modded (%) with _tableSize to yield a value between 0 and _tableSize - 1. You can experiment with different hash generation formulas by replacing key.GetHashCode() with a custom hash generation function.

The indexer, which starts on line 119, allows values stored within HomeGrownHashtable to be accessed and set with familiar array notation using the key. It consists of two parts: the get and set sections. The get section checks the key to ensure its not null or the empty string. The key is converted to upper case and its existence is checked in the _keys list. If it's in the list, a hash value is generated and used to find the value's location within the _table. The Key-ValuePair list located at that location must then be searched to find the key. When the key is found, the corresponding value is used to set return_value.

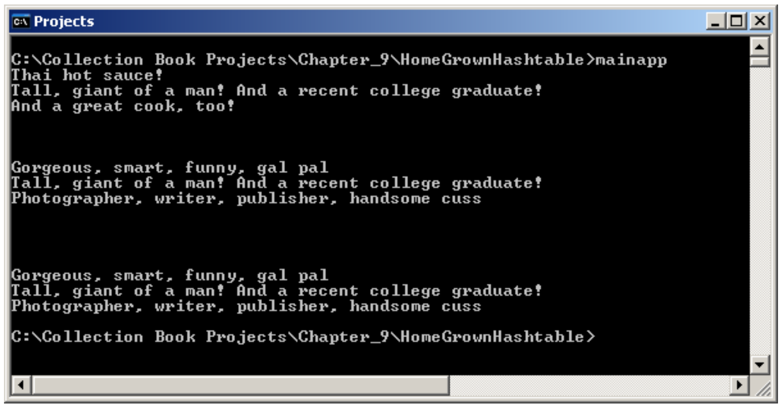
The set section works similar to the get section except that when the key is located, that KeyValuePair is removed and a new one created and added to the KeyValuePair list at that table location. The list is then sorted.

Example 9.3 offers a MainApp class that demonstrates the use of HomeGrownHashtable.

9.3 MainApp.cs (Demonstrating HomeGrownHashtable)

```
1      using System;
2
3      public class MainApp {
4        public static void Main(string[] args){
5          HomeGrownHashtable ht = new HomeGrownHashtable();
6          ht.Add("Rick", "Photographer, writer, publisher, handsome cuss");
7          ht.Add("Coralie", "Gorgeous, smart, funny, gal pal");
8          ht.Add("Kyle", "Tall, giant of a man! And a recent college graduate!");
9          ht.Add("Tati", "Thai hot sauce!");
10         Console.WriteLine(ht["Tati"]);
11         Console.WriteLine(ht["Kyle"]);
12         ht["Tati"] = "And a great cook, too!";
13         ht.DumpContentsToScreen();
14         ht.Remove("Tati");
15         ht.DumpContentsToScreen();
16       } // end Main() method
17     }
```

Referring to example 9.3 — an instance of HomeGrownHashtable is created on line 5. Lines 6 through 9 add several key/value pairs to hashtable. Lines 10 and 11 demonstrate the use of the indexer to access the values associ-ated with the keys "Tati" and "Kyle". On line 12, the indexer is used to replace the value associated with the key "Tati" with a new value. On line 13 the DumpContentsToScreen() method is called followed by the removal of the item referred to by the key "Tati" from the hashtable. The DumpContentsToScreen() method is then called one last time. Figure 9-3 shows the results of running this program.



```
C:\Collection Book Projects\Chapter_9\HomeGrownHashtable>mainapp
Thai hot sauce!
Tall, giant of a man! And a recent college graduate!
And a great cook, too!


Gorgeous, smart, funny, gal pal
Tall, giant of a man! And a recent college graduate!
Photographer, writer, publisher, handsome cuss



Gorgeous, smart, funny, gal pal
Tall, giant of a man! And a recent college graduate!
Photographer, writer, publisher, handsome cuss

C:\Collection Book Projects\Chapter_9\HomeGrownHashtable>
```

Figure 9-3: Results of Running Example 9.3

Quick Review

A hashtable is an array that employs a special operation called a *hash function* to calculate an object's location within the array. The object being inserted into the table is referred to as the *value*. A hash function is applied to the value's associated *key* which results in an integer value that lies somewhere between the first array element (0) and the last array element (n-1).

The HomeGrownHashtable class implements a chained hashtable where each hashtable *bucket* points to a List<KeyValuePair> object into which KeyValuePair objects are inserted. The hashtable's load limit determines when the hashtable should be grown to accommodate additional elements. The load limit is calculated by multiplying the table size by the load factor.

Hashtable *collisions* can occur when two different keys hash to the same hash value. The chained hash table resolves collisions by allowing collisions to occur and storing the KeyValuePair objects in a list at that location which must then be searched to find the key/value pair of interest.

Hashtable Class

The Hashtable class, located in the System.Collections namespace, stores hashtable elements as DictionaryEntry objects. A DictionaryEntry object consists of Key and Value properties and methods inherited from the System.Object class.

Unlike my HomeGrownHashtable discussed in the previous section, the Hashtable class doesn't use chaining to resolve collisions. According to Microsoft's documentation it uses a technique called *double hashing*. Double hashing works like this: If a key hashes to a bucket value already occupied by another key, the hash function is altered slightly and the key is rehashed. If that bucket location is empty the value is stored there, if it's occupied, the key must again be rehashed until an empty location is found.

Figure 9-4 shows the UML class diagram for the HashTable inheritance hierarchy.

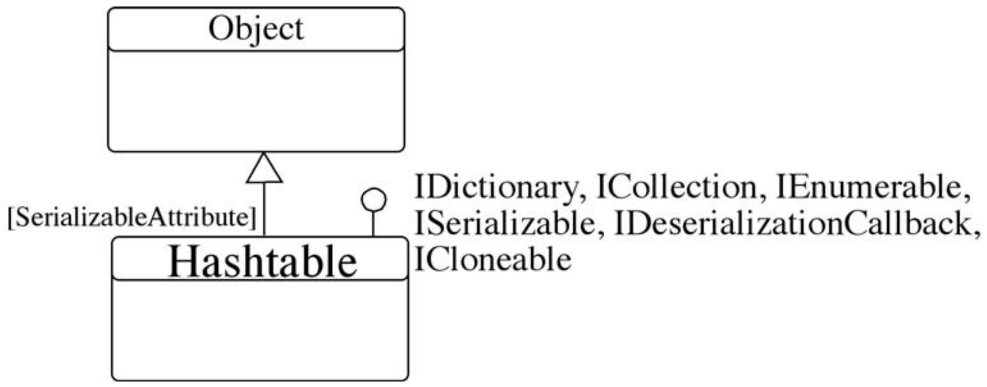

Object
[SerializableAttribute]
Hashtable
IDictionary, ICollection, IEnumerable,
ISerializable, IDeserializationCallback,
ICloneable

Figure 9-4: Hashtable Class Inheritance Hierarchy

Referring to figure 9-4 — The Hashtable class implements the IDictionary, ICollection, IEnumerable, ISerializable, IDeserializableCallback, and ICloneable interfaces. The functionality provided by each of these interfaces is discussed in more detail below.

Functionality Provided by the IEnumerable Interface

The IEnumerable interface allows the items contained within the Hashtable collection to be iterated over with the `foreach` statement. Note that each element of a Hashtable collection is a DictionaryEntry object. The code to iterate over each element of a Hashtable with a `foreach` statement would look something like the following code snippet, assuming there exists a reference to a Hashtable object named `ht`:

```
foreach(DictionaryEntry item in ht){
  Console.WriteLine(item.Key);
}
```

The code snippet above would print out the value of each key to the console.

The Hashtable class also provides several properties, two of which are Keys and Values. The Keys property returns an ICollection of the Hashtable's keys and the Values property returns an ICollection of the Hashtable's Values. You can use a `foreach` statement to step through each of these collections as the following code snippet demonstrates, assuming that keys are strings:

```
foreach(string key in ht.Keys){
  Console.WriteLine(key);
}
```

In this example, each key in the Keys collection is written to the console. Later, in the Hashtable example program, I'll show you another way to step through the items in a Hashtable using its indexer.

Remember — when stepping through a collection with the `foreach` statement, you can't modify the elements.

Functionality Provided by the ICollection Interface

The ICollection interface declares the GetEnumerator() and CopyTo() methods as well as the IsSynchronized and SyncRoot properties. The IsSynchronized and SyncRoot properties are discussed in greater detail in Chapter 14 — Collections and Threads.

Functionality Provided by the IDictionary Interface

The IDictionary interface declares the Add(), Remove(), and Contains() methods, the indexer (shown as Item in the properties list), and the Keys and Values properties.

Functionality Provided by the ISerializable and IDeserializationCallBack Interfaces

The ISerializable and IDeserializationCallback interfaces indicate that the Hashtable class requires custom serialization over and above simply tagging the class definition with the Serializable attribute. Custom serialization and deserialization is discussed in detail in Chapter 17 — Collections and I/O.

Functionality Provided by the ICloneable Interface

The ICloneable interface makes it possible to make copies of Hashtable objects.

Hashtable In Action

Example 9.4 demonstrates the use of the Hashtable collection. In this example, the program reads a text file line-by-line and stores each line in the Hashtable collection using the line number as the key. The name of the text file must be supplied on the command line when the program is executed.

9.4 HashtableDemo.cs

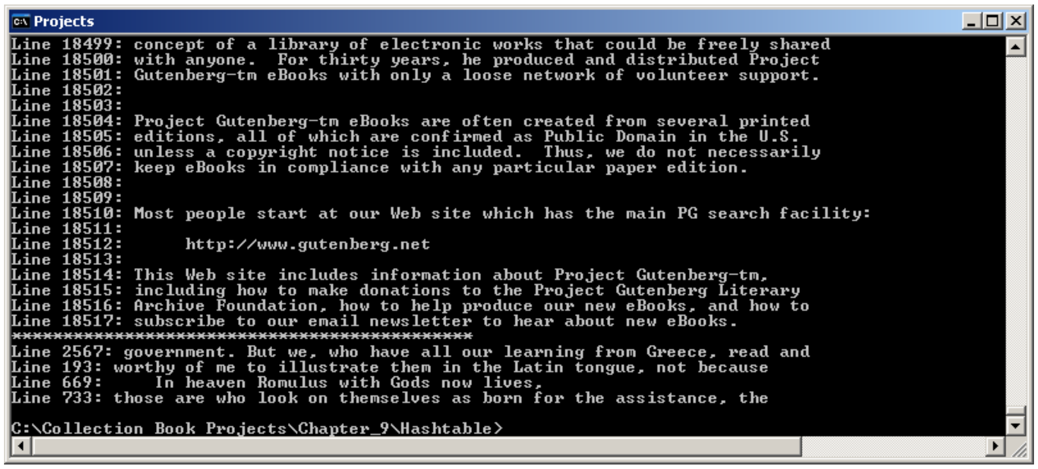
```
using System;
using System.Collections;
using System.IO;
using System.Text;


public class HashtableDemo {

  public static void Main(string[] args){
    FileStream fs = null;
    StreamReader reader = null;
    Hashtable ht = new Hashtable();
    try {
       fs = new FileStream(args[0], FileMode.Open);
       reader = new StreamReader(fs);

       int line_count = 1;
       string input_line = string.Empty;
       while((input_line = reader.ReadLine()) != null){
         string line_number_string = (line_count++).ToString();
         if(!ht.Contains(line_number_string)){
           ht.Add(line_number_string, input_line);
23            }
24          }
25
26          }catch(IndexOutOfRangeException){
27            Console.WriteLine("Please enter the name of a text file on the command line " +
28                                "when running the program!");
29          }catch(Exception e){
30             Console.WriteLine(e);
31          } finally {
32            if(fs != null){
33              fs.Close();
34            }
35            if(reader != null){
36              reader.Close();
37            }
38          }
39
40
41         for(int i = 1; i<=ht.Keys.Count; i++){
42           Console.WriteLine("Line {0}: {1}", i, ht[i.ToString()]);
43         }
44
45         Console.WriteLine("*********************************************");
46         Console.WriteLine("Line {0}: {1}", 2567, ht[2567.ToString()]);
47         Console.WriteLine("Line {0}: {1}", 193, ht[193.ToString()]);
48         Console.WriteLine("Line {0}: {1}", 669, ht[669.ToString()]);
49         Console.WriteLine("Line {0}: {1}", 733, ht[733.ToString()]);
50
51     } // end Main() method
52   } // end HashtableDemo class
```

Referring to example 9.4 — a FileStream reference named `fs` is declared on line 10 and initialized to null. On line 11 a StreamReader reference named `reader` is declared and also initialized to null. In the body of the try/catch block, which begins on line 13, the FileStream object is created using the first command line argument (args[0]). The FileStream object is then used to create the StreamReader object on the following line. On line 17, a local variable named `line_count` is declared and initialized to 1. An `input_line` variable of type string is declared on the following line and initialized to string.Empty. The `while` loop on line 19 processes each line of the text file. It first formulates a line number string (`line_number_string`) and checks its existence within the hashtable via the Contains() method. If it's not in the hashtable, the `input_line` is added using the `line_number_string` as its key.

The `for` statement on line 41 steps through each element of the hashtable using its indexer and writing the retrieved value to the console. Lines 46 through 49 access individual elements of the hashtable via the indexer.

The text file used for this example is named Book.txt. It contains the complete text of Cicero's Tusculan Disputations, by Marcus Tullius Cicero. It was downloaded from the Project Gutenberg website. (www.gutenberg.net)

Figure 9-5 shows the results of running this program. Note that figure 9-5 only shows the last few lines of the output of 18517 lines of text printed to the console.



```
Line 18499: concept of a library of electronic works that could be freely shared
Line 18500: with anyone.  For thirty years, he produced and distributed Project
Line 18501: Gutenberg-tm eBooks with only a loose network of volunteer support.
Line 18502:
Line 18503:
Line 18504: Project Gutenberg-tm eBooks are often created from several printed
Line 18505: editions, all of which are confirmed as Public Domain in the U.S.
Line 18506: unless a copyright notice is included.  Thus, we do not necessarily
Line 18507: keep eBooks in compliance with any particular paper edition.
Line 18508:
Line 18509:
Line 18510: Most people start at our Web site which has the main PG search facility:
Line 18511:
Line 18512:      http://www.gutenberg.net
Line 18513:
Line 18514: This Web site includes information about Project Gutenberg-tm,
Line 18515: including how to make donations to the Project Gutenberg Literary
Line 18516: Archive Foundation, how to help produce our new eBooks, and how to
Line 18517: subscribe to our email newsletter to hear about new eBooks.
*********************************************
Line 2567: government. But we, who have all our learning from Greece, read and
Line 193: worthy of me to illustrate them in the Latin tongue, not because
Line 669:     In heaven Romulus with Gods now lives,
Line 733: those are who look on themselves as born for the assistance, the

C:\Collection Book Projects\Chapter_9\Hashtable>
```

Figure 9-5: Results of Running Example 9.4

Quick Review

The Hashtable is a non-generic collection class that stores its item as DictionaryEntry objects. The Hashtable class resolves collisions via *double hashing*, which is a process by which a key is rehashed using a modified hashing function until the collision has been resolved by the generation of a unique bucket location.

Dictionary<TKey, TValue> Class

The Dictionary<TKey, TValue> class is the strongly-typed, generic version of the non-generic Hashtable class. The Dictionary<TKey, TValue> class also differs from the Hashtable class in the way it handles collisions. It uses chaining. Like the HomeGrownHashtable presented earlier, values whose keys hash to the same bucket are stored in a list, however, unlike the HomeGrownHashtable example, the Dictionary<TKey, TValue> class uses a different chain management algorithm which I'm positive is much more efficient than the approach I used.

Another huge difference between the Dictionary<TKey, TValue> class and the Hashtable class is the large number of extension methods provided by the System.Linq.Enumerable class that can be used on the Dictionary.

Figure 9-6 offers a UML class diagram showing the inheritance hierarchy of the Dictionary<TKey, TValue> class.

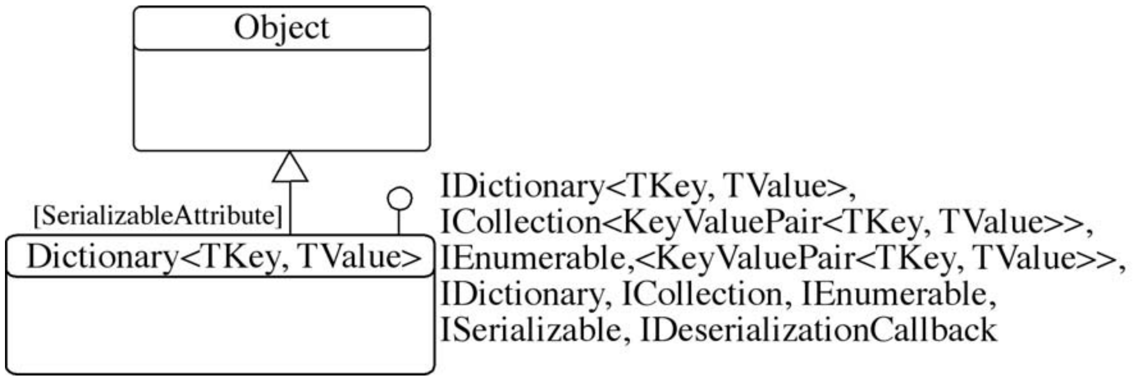

Object
[SerializableAttribute]
Dictionary<TKey, TValue>
IDictionary<TKey, TValue>,
ICollection<KeyValuePair<TKey, TValue>>,
IEnumerable,<KeyValuePair<TKey, TValue>>,
IDictionary, ICollection, IEnumerable,
ISerializable, IDeserializationCallback

Figure 9-6: Dictionary<TKey, TValue> Class Inheritance Hierarchy

Referring to figure 9-6 — the Dictionary<TKey, TValue> class implements the IDictionary<TKey, TValue>, ICollection<KeyValuePair<TKey, TValue>>, IEnumerable<KeyValuePair<TKey, TValue>>, IDictionary, ICollection, IEnumerable, ISerializable, and IDeserializationCallback interfaces. Each of these interfaces are discussed in greater detail below.

Functionality Provided by the IEnumerable and IEnumerable<KeyValuePair<TKey, TValue>> Interfaces

The IEnumerable and IEnumerable<KeyValuePair<TKey, TValue>> interfaces allow the items within a Dictionary<TKey, TValue> class to be iterated over with a `foreach` statement. Note that each element with a Dictionary<TKey, TValue> collection is a KeyValuePair<TKey, TValue> object. Assuming there was a reference to a Dictionary<string, int> object named names_and_ages, you could step through each element of the collection using a `foreach` statement similar to the following code snippet:

```
foreach(KeyValuePair<string, int> entry in names_and_ages){
  Console.WriteLine("{0} is {1} years old!", entry.Key, entry.Value);
}
```

Functionality Provided by the ICollection and ICollection<KeyValuePair<TKey, TValue>> Interfaces

The ICollection and ICollection<KeyValuePair<TKey, TValue>> interfaces tag the Dictionary<TKey, TValue> class as a collection type. The ICollection interface declares object synchronization properties IsSynchronized and SyncRoot, while the ICollection<KeyValuePair<TKey, TValue>> interface declares the Add(), Remove(), and Contains() methods and the Count property. These interfaces also declare the GetEnumerator() methods required to iterate over the collection with a `foreach` statement.

Functionality Provided by the IDictionary and IDictionary<KeyValuePair<TKey, TValue>> Interfaces

The IDictionary and IDictionary<KeyValuePair<TKey, TValue>> interfaces provide the non-generic and generic versions of Keys and Values properties, the indexer, and the ContainsKey() and the TryGetValue() methods.

Functionality Provided by the ISerializable and IDeserializationCallback Interfaces

The ISerializable and IDeserializationCallback interfaces indicate that the Dictionary<TKey, TValue> collection requires custom serialization code over and beyond what the Serializable attribute alone provides.

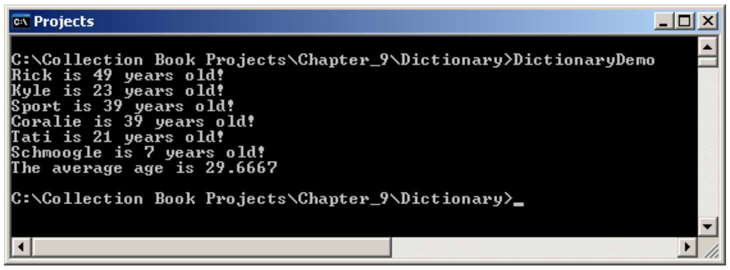
Dictionary<TKey, TValue> Example

Example 9.5 presents a short program demonstrating the use of the Dictionary<TKey, TValue> collection.

9.5 DictionaryDemo.cs

```
1       using System;
2       using System.Collections.Generic;
3       using System.Linq;
4
5       public class DictionaryDemo {
6
7          public static void Main(){
8            Dictionary<string, int> names_and_ages = new Dictionary<string, int>();
9            names_and_ages.Add("Rick", 49);
10           names_and_ages.Add("Kyle", 23);
11           names_and_ages.Add("Sport", 39);
12           names_and_ages.Add("Coralie", 39);
13           names_and_ages.Add("Tati", 21);
14           names_and_ages.Add("Schmoogle", 7);
15
16           foreach(KeyValuePair<string, int> entry in names_and_ages){
17             Console.WriteLine("{0} is {1} years old!", entry.Key, entry.Value);
18           }
19
20           Console.WriteLine("The average age is {0:F4}", names_and_ages.Values.Average());
21
22         } //  end Main() method
23      } // end DictionaryDemo class
```

Referring to example 9.5 — a Dictionary<string, int> reference named `names_and_ages` is declared and created on line 8. In this case, the keys will be strings and the values will be integers. Lines 9 through 14 add several entries into the dictionary. The `foreach` statement on line 16 steps through each KeyValuePair entry in the dictionary and prints the key and value to the console. Line 20 extracts the values from the dictionary via the Values property and calls the extension method Average() to calculate the average age. Figure 9-7 shows the results of running this program.



```
C:\Collection Book Projects\Chapter_9\Dictionary>DictionaryDemo
Rick is 49 years old!
Kyle is 23 years old!
Sport is 39 years old!
Coralie is 39 years old!
Tati is 21 years old!
Schmoogle is 7 years old!
The average age is 29.6667

C:\Collection Book Projects\Chapter_9\Dictionary>_
```

Figure 9-7: Results of Running Example 9.5

Quick Review

The Dictionary<TKey, TValue> collection is a strongly-typed version of the Hashtable class that uses chaining instead of double hashing to resolve collisions. The Dictionary<TKey, TValue> collection stores its items as KeyValuePair objects. (See System.Collections.Generic.KeyValuePair<TKey, TValue> structure.)

Summary

A hashtable is an array that employs a special operation called a *hash function* to calculate an object's location within the array. The object being inserted into the table is referred to as the *value*. A hash function is applied to the value's associated *key* which results in an integer value that lies somewhere between the first array element (0) and the last array element (n-1).

The HomeGrownHashtable class implements a chained hashtable where each hashtable *bucket* points to a List<KeyValuePair> object into which KeyValuePair objects are inserted. The hashtable's load limit determines when the hashtable should be grown to accommodate additional elements. The load limit is calculated by multiplying the table size by the load factor.

Hashtable *collisions* can occur when two different keys hash to the same hash value. The chained hash table resolves collisions by allowing collisions to occur and storing the KeyValuePair objects in a list at that location which must then be searched to find the key/value pair of interest.

The Hashtable is a non-generic collection class that stores its item as DictionaryEntry objects. The Hashtable class resolves collisions via *double hashing*, which is a process by which a key is rehashed using a modified hashing function until the collision has been resolved by the generation of a unique bucket location.

The Dictionary<TKey, TValue> collection is a strongly-typed version of the Hashtable class that uses chaining instead of double hashing to resolve collisions. The Dictionary<TKey, TValue> collection stores its items as KeyValuePair objects. (See System.Collections.Generic.KeyValuePair<TKey, TValue> structure.)

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]

Arash Partow. The General Purpose Hash Function Algorithms Library (GeneralHashFunctionLibrary.java). [http://www.partow.net/programming/hashfunctions/index.html]

Project Gutenberg [http://www.gutenberg.net]

Notes

© 2011 Rick Miller — All Rights Reserved