VISUAL BASIC 2005 EXPRESS: NOW PLAYING

by Wallace Wang

NO STARCH
PRESS

San Francisco

24

ADVANCED DATA STRUCTURES: QUEUES, STACKS, AND HASH TABLES

To provide greater flexibility in storing information, Visual Basic includes three data structures called *queues*, *stacks*, and *hash tables*, which, as with arrays and collections, do nothing more than store data in a list. However, they offer different ways to search, store, and remove data, which can make them more useful than arrays or collections, depending on what you want your program to do.

Using a Queue

Understanding Queues

A *queue* always adds data to the end of a list but removes data from the front of the list. A queue takes its name from the way it stores and removes data, which mimics the way people wait in line: the first person in line is the first one to leave. Because of the way a queue adds and removes data, queues are sometimes called a *First In, First Out (FIFO)* data structure.

A queue can be useful for creating an inventory program in which you want to remove the oldest stored item first. If you were to use a different data structure, such as a collection, to remove an item, old items could be left sitting idly by or rotting away in your inventory, which would waste your company's resources.

Figure 24-1 shows a queue in three different stages. The top stage shows the number 61 being added to an existing queue already full of data. The middle stage shows how the existing queue has grown by one additional item, in this case the number 61. The bottom stage shows that the only way to remove an item from a queue is from the front of the queue, which is the opposite end from where new data is added.

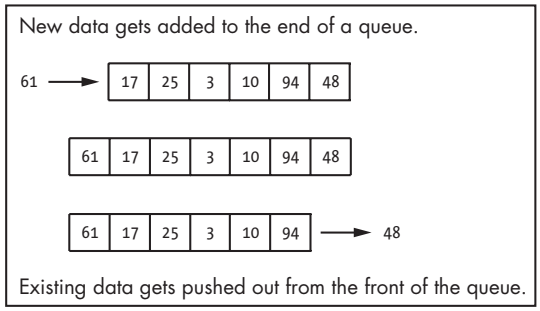

Figure 24-1: A queue stores new data at one end of the list and removes old data from the other end.

To create a queue, you need to declare a variable as a queue, like so:

```
Dim VariableName As New Queue
```

When you create a queue, it contains no data, so the first step to using a queue is to stuff it with data. Like a collection, a queue can store different types of data, such as string, integers, and boolean values.

Adding Data to a Queue with the Enqueue Method

When you add data to a queue, the data is stored at the end of the queue. Each time you add data to a queue, the queue gets longer and longer. To add data to a queue, use the `Enqueue` method, which looks like this:

```
QueueName.Enqueue (Data)
```

where *QueueName* is the name of the queue and *Data* is the data you want to store in a queue.

The following code creates a queue and stores three items in it: a number, a string, and a boolean True value, as shown in Figure 24-2:

```
Dim MyQueue As New Queue
MyQueue.Enqueue (29)
MyQueue.Enqueue ("Cat")
MyQueue.Enqueue (True)
```

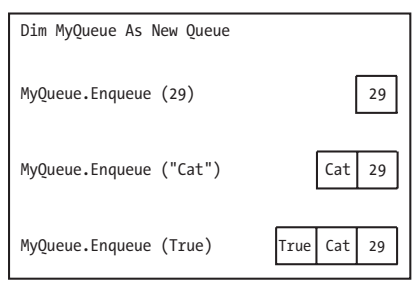

Figure 24-2: The Enqueue method stores data in a queue, putting the first stored item at the front of the list and adding new items to the back of the list.

Checking the Contents of a Queue

As you fill a queue with data, you may lose track of what data is actually stored in it. To search the queue for specific data, use the `Contains` method, which returns a True or False value, like so:

```
QueueName.Contains (Data)
```

where *QueueName* is the name of the queue and *Data* is the data you want to find in the queue.

The following code stores three items in a queue and then searches to determine whether the queue contains the string "Cat". Because the queue contains "Cat", the MsgBox command displays *True* in a message box.

```
Dim MyQueue As New Queue
MyQueue.Enqueue (29)
MyQueue.Enqueue ("Cat")
MyQueue.Enqueue (True)
MsgBox(MyQueue.Contains("Cat"))
```

When you search for data in a queue, you must search for the exact data. For example, if a queue contains the string "Cat" (uppercase C) but you search for the string cat (lowercase c), the Contains method will return False because "Cat" is not exactly identical to the string "cat".

If you want to know only the data currently stored at the front of the queue, use the Peek method, like so:

```
Dim MyQueue As New Queue
MyQueue.Enqueue (29)
MyQueue.Enqueue ("Cat")
MyQueue.Enqueue (True)
MsgBox(MyQueue.Peek)
```

In this example, the number 29 is stored at the front of the queue because it was the first item placed in the queue. Therefore, the Peek method returns the number *29* and displays that in a message box.

NOTE *The Peek method doesn't remove any data from the queue. It simply returns the value of the next chunk of data that you can remove from the queue.*

Counting the Contents of a Queue

The number of items that a queue can hold may vary as your program stores and removes data from the queue. To determine the current number of items stored in a queue, use the Count method, like so:

```
QueueName.Count
```

where *QueueName* is the name of the queue.

The following BASIC code adds three items to a queue, runs the Count method, and displays the number *3* in a message box after counting three items in the queue:

```
Dim MyQueue As New Queue
MyQueue.Enqueue (29)
MyQueue.Enqueue ("Cat")
MyQueue.Enqueue (True)
MsgBox(MyQueue.Count)
```

Removing Data from a Queue

You can remove data only from the front of the queue, which means that the oldest item you stored in the queue will also be the next item you can remove. To remove an item from a queue, use the Dequeue method, like so:

```
QueueName.Dequeue
```

where *QueueName* is the name of the queue.

When you use the Dequeue method, you physically remove the first item from the queue, as shown in Figure 24-3.

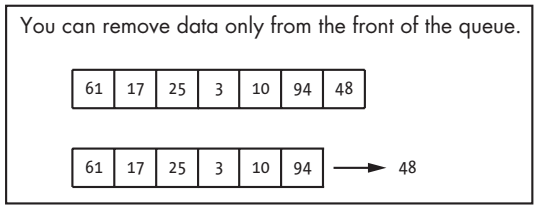

Figure 24-3: The Dequeue method removes the first item from the front of the queue, shortening the length of the queue by one item.

The following BASIC code adds three items to a queue, then uses Dequeue to remove the first item. In this case, the first item is the number *29*, so the MsgBox command displays *29* in a message box:

```
Dim MyQueue As New Queue
Dim MyData As Object
MyQueue.Enqueue (29)
MyQueue.Enqueue ("Cat")
MyQueue.Enqueue (True)
MyData = MyQueue.Dequeue
MsgBox(MyData)
```

Because a queue stores data as object data types, when you want to retrieve data from a queue, you must always store it in a variable that represents an object data type. In the preceding code, the MyData variable represents an object data type.

As an alternative to removing data from a queue one item at a time, you can empty a queue completely by using the Clear method, like so:

```
QueueName.Clear()
```

where *QueueName* is the name of the queue.

The following code adds three items to a queue, displays the total number of items (*3*), empties the queue, and then runs the Count method to count the total number of items stored in the queue, which is zero:

```
Dim MyQueue As New Queue
MyQueue.Enqueue (29)
MyQueue.Enqueue ("Cat")
MyQueue.Enqueue (True)
MsgBox(MyQueue.Count)
MyQueue.Clear()
MsgBox(MyQueue.Count)
```

In this example, the first MsgBox command displays a *3* in a message box. Next, the Clear method runs and empties the queue before the second MsgBox command displays a *0* in a message box to let you know the queue is really empty.

When you remove data from a queue using the Dequeue method, you can store the removed data in a variable, but when you empty a queue using the Clear method, you lose all the emptied data. To save the entire contents of a queue before emptying it, copy the data to an array first.

Copying Data from a Queue to an Array

As an alternative to removing data from a queue one item at a time using the `Dequeue` method, you can copy all the data from a queue and place it in an array. When you copy data from an array, the contents of your queue is unchanged.

Because a queue can vary in size, you first need to create a queue and a dynamic array that stores object data types. Then, after filling the queue with data, you can use the `ToArray` method to assign the contents of the queue to that array, as follows:

```
Dim QueueName As New Queue
Dim ArrayName() As Object
QueueName.Enqueue(Data)
ArrayName = QueueName.ToArray()
```

where *QueueName* is the name of your queue, *ArrayName* is the name of the dynamic array that holds object data types, and *Data* is the data, such as a number or string, that you want to store in your queue.

When you copy data from a queue to an array, the first item in the queue is stored at index 0 of the array, the second item is stored in index 1 of the array, and so on.

The following code stores four items in a queue, copies them into an array, and then uses a For-Next loop to display each item in the array. Because arrays begin counting at index position 0, the For-Next loop has to start counting at 0 and stop counting at the total number of items in the queue minus one, which is 3.

```
Dim cat As New Queue
Dim MyArray() As Object
Dim J As Integer
cat.Enqueue("Bo")
cat.Enqueue(False)
cat.Enqueue(75.4)
cat.Enqueue("Nuit")
MyArray = cat.ToArray()
For J = 0 To (cat.Count- 1)
    MsgBox(MyArray(J))
Next
```

An alternative to copying data from a queue to an array is to use the `CopyTo` method. Unlike the `ToArray` method, which always stores the first item of the queue in the first position (index 0) of an array, the `CopyTo` method lets you store data from the queue starting in any index position of the array, like so:

```
QueueName.CopyTo(ArrayName, Index)
```

where *QueueName* is the name of your queue, *ArrayName* is the name of the dynamic array that holds object data types, and *Index* is the index position where you want to start storing data in the array. If the value of *Index* is `0`, the first item in the queue is stored at index position 0 of the array, the second item is stored in index position 1 of the array, and so on.

Before you can copy data from a queue to an array, you must resize your array with the `ReDim` keyword, as shown here:

```
Dim cat As New Queue
Dim MyArray() As Object
Dim I As Integer
```

```
cat.Enqueue("Bo")
cat.Enqueue(False)
cat.Enqueue(75.4)
cat.Enqueue("Nuit")
ReDim MyArray(6)
cat.CopyTo(MyArray, 2)
For I = 0 To 6
    MsgBox(MyArray(I))
Next
```

This example resizes the dynamic array (`MyArray`) to hold seven elements (0 through 6) and then uses the `CopyTo` method to copy the data from the queue into the array, starting at index position 2, which is the third position in the array. When the For-Next loop runs, it first displays two blank message boxes (because the initial two elements of `MyArray` are empty) and then displays the data copied from the queue.

Using a Stack

Understanding Stacks

A *stack* is a data structure that mimics a stack of plates stored in a cafeteria. The last item stored in the stack is also the first item removed from the stack, often known as a *Last In, First Out (LIFO)* data structure, as shown in Figure 24-4.

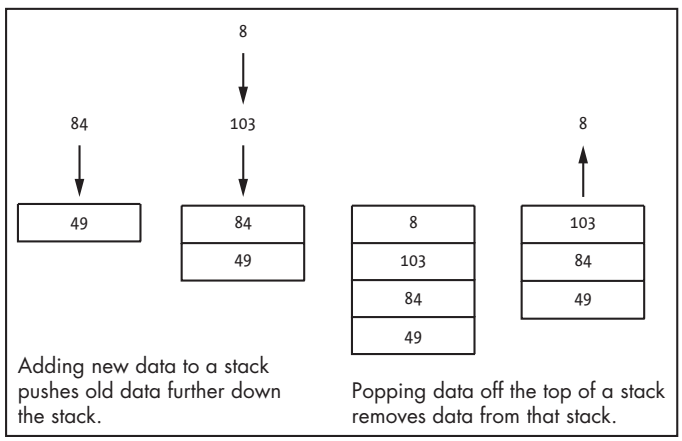

Figure 24-4: A stack pushes new data on top of old data and pops off data to remove it from the top of the stack.

To create a stack, you need to declare a variable as a stack, like so:

```
Dim VariableName As New Stack
```

When you create a stack, it contains no data, so the first step is to *push* data on to the top of the stack. To remove data from a stack, you *pop* it off the top.

Like a queue, a stack can store different types of data, such as string, integers, and boolean values. Stacks are often used to store data temporarily. For example, calculator programs use stacks to store different numbers and mathematical operators (plus sign, division sign, and so on) that the user chooses. Each time the user types a number, the program stores that number on the stack. When the user types a mathematical operator such as the plus sign (+), that is also stored on the stack. As soon as the user types another number, the program pops the mathematical operator off the stack along with the previously stored number on the stack and calculates a final result.

Pushing Data

To store data on a stack, use the `Push` method, like so:

```
StackName.Push(Data)
```

where *StackName* is the name of the stack where you want to add new data and *Data* is the data you want to push on top of the stack.

The following code shows how to push three items on to a stack, as shown in Figure 24-5:

```
Dim cat As New Stack
cat.Push("Bo")
cat.Push(False)
cat.Push(304)
```

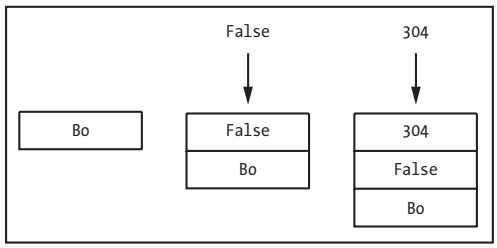

Figure 24-5: The string Bo *is the first item pushed on a stack and will be the last item that can be removed from the stack.*

Checking the Contents of a Stack

Once you've stored data on a stack, you can use the `Peek` method to see the last data stored on top of the stack. The `Peek` method doesn't remove the data from the stack; it just returns the value of that data.

To use the `Peek` method, use the following syntax:

```
StackName.Peek
```

The following BASIC code pushes two items onto a stack and then runs the `MsgBox` command to display data in a message box:

```
Dim idiots As New Stack
idiots.Push("Pauly")
idiots.Push("Bobby")
MsgBox(idiots.Peek)
```

This example pushes the string `"Pauly"` onto the stack and then shoves another string, `"Bobby"`, right on top of it. When the `Peek` method runs, it returns the value of the last item stored on the stack, which is the string `"Bobby"`, then the `MsgBox` command displays *Bobby* in a message box.

The `Peek` method is handy for letting you check on the value of the last item stored on top of a stack, but if you want to search for data that may be stored somewhere else in a stack, you'll have to use the `Contains` method, which uses this syntax:

```
StackName.Contains(Data)
```

where *StackName* is the name of the stack that you want to search and *Data* is the data you want to find inside the stack.

The Contains method returns a True or False value. If you search and find data exactly matching any data stored in a stack, the Contains method returns a True value; otherwise, it returns a False value.

The following code pushes three items onto a stack and then searches for an item:

```
Dim wmd As New Stack
wmd.Push("Smallpox")
wmd.Push("Uranium")
wmd.Push("Nerve gas")
MsgBox(wmd.Contains("Uranium"))
```

In this example, the Contains method searches for the string "Uranium" in the stack named wmd. Because the string "Uranium" is already stored in the stack, the Contains method returns a True value and the MsgBox command displays *True* in a message box.

If you search for "uranium" (lowercase u) in a stack but the stack contains only the string "Uranium", (uppercase U), the Contains method will return a False value because the strings are not exactly identical.

NOTE *The Contains method tells you only whether certain data is stored in a stack; it won't tell you where in the stack that data may be stored.*

Counting the Contents of a Stack

A stack can grow and shrink as you push data in and pop data off it. To count the number of items stored in a stack, use the Count method, like so:

```
StackName.Count
```

The following code pushes three items onto a stack and then uses the Count method to count the total number of items in the stack (3).

```
Dim wmd As New Stack
wmd.Push("Smallpox")
wmd.Push("Uranium")
wmd.Push("Nerve gas")
MsgBox(wmd.Count)
```

In this example, the Count method returns a value of 3, so the MsgBox command displays *3* in a message box.

Popping Data from a Stack

To remove data from a stack, use the Pop method, which removes the last item stored on the stack. If you want to save the contents of the popped data, you need to store the popped data into a variable declared to hold object data types, such as the following:

```
Dim VariableName As Object
VariableName = StackName.Pop
```

The following code pushes three items onto a stack and then uses the Pop method to remove the last item pushed onto the stack.

```
Dim axis As New Stack
axis.Push("North Korea")
axis.Push("Iraq")
axis.Push("IRS")
MsgBox(axis.Pop)
```

In this example, the `Pop` method pops off the last item stored, which is `IRS`, so the `MsgBox` command displays *IRS* in a message box.

To clear out all data stored in a stack (without saving any of it), you can use the `Clear` method, like so:

```
StackName.Clear
```

The following code pushes three items onto a stack, uses the `Count` method to display *3* in a message box, uses the `Clear` method to remove all data from the stack, and then uses the `Count` method again to display *0* in a message box, showing you that the stack is now empty:

```
Dim axis As New Stack
axis.Push("North Korea")
axis.Push("Iraq")
axis.Push("IRS")
MsgBox(axis.Count)
axis.Clear
MsgBox(axis.Count)
```

Copying Data from a Queue to an Array

Popping data off a stack one item at a time can be tedious, so you can also copy the entire contents of a stack to an array using either the `ToArray` or `CopyTo` methods.

To use the `ToArray` method, you first create a dynamic array that stores object data types, then you use the `ToArray` method to assign the contents of the stack to that array, as follows:

```
Dim StackName As New Stack
Dim ArrayName() As Object
StackName.Push(Data)
ArrayName = StackName.ToArray()
```

where *StackName* is the name of your stack; *ArrayName* is the name of your dynamic array that holds object data types; and *Data* is the data, such as a number or string, that you want to store in your stack.

When you copy data from a stack to an array, Visual Basic pops the top item off the stack (the last item stored on the stack) and stores it in the first position of the array (index position 0). It then pops the next item off the stack and stores it in index position 1 of the array, and so on.

The following code stores three items on a stack, copies them into an array, and then uses a For-Next loop to display each item in the array. Because arrays begin counting at index position 0, the For-Next loop has to start counting at 0 and stop counting at the total number of items in the stack minus one, which is 2.

```
Dim axis As New Stack
Dim MyArray() As Object
```

```
Dim I As Integer
axis.Push("North Korea")
axis.Push("Iraq")
axis.Push("IRS")
MyArray = axis.ToArray()
For I = 0 To axis.Count - 1
    MsgBox(MyArray(I))
Next
```

This example runs the `MsgBox` command three times. The first time the `MsgBox` command displays *IRS*, the second time it displays *Iraq*, and the third and last time it displays *North Korea* in a message box.

Another way to copy data from a stack to an array is to use the `CopyTo` method. Unlike the `ToArray` method, which always stores the first item of the queue in the first position (index 0) of an array, the `CopyTo` method lets you store data from the stack starting with any index position of the array:

```
StackName.CopyTo(ArrayName, Index)
```

where *`StackName`* is the name of your stack, *`ArrayName`* is the name of your dynamic array that holds object data types, and *`Index`* is the index position where you want to start storing data in the array. If the *`Index`* value is `0`, the first item popped off the stack is stored at index position 0 of the array, the second item is stored in index position 1, and so on.

Before you can copy data from a stack to an array, you must resize your array using the `ReDim` keyword, as the following code demonstrates:

```
Dim axis As New Stack
Dim MyArray() As Object
Dim I As Integer
axis.Push("North Korea")
axis.Push("Iraq")
axis.Push("IRS")
ReDim MyArray(5)
axis.CopyTo(MyArray, 1)
For I = 0 To 5
    MsgBox(MyArray(I))
Next
```

This example resizes the dynamic array (`MyArray`) to hold six elements (0 through 5) and then uses the `CopyTo` method to pop data, one at a time, from the stack into the array, starting at index position 1, which is the second position in the array. When the For-Next loop runs, it first displays one blank message box (because the initial element of the `MyArray` is empty at index position 0), and then displays the data copied from the stack.

Using a Hash Table

Understanding Hash Tables

A *hash table* acts like a list that stores data along with a unique *key* that can identify that data, as shown in Figure 24-6. The data associated with a key can be as simple as a single number or string or as complicated as a stack, a structure, or a two-dimensional array. These keys will help you to identify and find that data later on, no matter where it is in the hash table.

NOTE *Every key stored in a hash table must be unique; you cannot use the same key twice, or your program won't work.*

Keys act like shortcuts to help you find data. For example, you could have a key named CEO and the data associated with it could be a structure that stores the name, address, and phone number of the company's current CEO. To find the CEO's phone number, you could look up the key CEO and that person's name would pop up. Without keys, you would have to search for the specific data you want to find. If you can search for the correct key, you'll always be able to find the CEO's private information.

Keys	Data
"VB"	"Visual Basic"
666	"Politicians"
"BAK"	"Backup"
"RB"	80394
"DLL"	"Dynamic Link Library"
2005	"RealBasic"
95	"Buggy OS"

Figure 24-6: A hash table stores data in a list where each chunk of data is associated with a unique key.

To create a hash table, declare a variable `Hashtable`, like so:

```
Dim VariableName As New Hashtable
```

Hash tables are initially empty, so you need to store data in them. This data can be any data type, such as a mix of integer and single data types.

Adding Data to a Hash Table

When you add data to a hash table, you need to add the actual data plus the unique key that you want to associate with that data. Both the key and the data you want to add to the hash table can be any data type, such as a number or a string.

To add data to a hash table, use the `Add` method, like so:

```
HashtableName.Add(Key, Data)
```

where *`HashtableName`* is the name of the hash table where you want to add data, *`Key`* is the unique value used to identify the data you want to add, and *`Value`* is the actual data to add to the hash table.

The following code adds two items to a hash table:

```
Dim ColdHash as New Hashtable
ColdHash.Add(1, "Potatoes")
ColdHash.Add("y", "Meat")
```

While this example adds two items to a hash table, nothing will appear to happen because you can't actually see anything being added to the hash table.

Retrieving Data from a Hash Table

Once you've stored keys and data in a hash table, you can use the key to retrieve the data later. When you retrieve data using a key, you simply copy the data from the hash table into a variable; you don't physically remove the data from the hash table itself.

To retrieve data from a hash table, use the `Item` method, like so:

```
HashtableName.Item(Key)
```

where *HashtableName* is the name of the hash table where you want to retrieve data and *Key* is the unique item associated with the data stored in the hash table.

For example, the following code adds two items to a hash table, searches for the data associated with the y key, and displays that data in a message box:

```
Dim ColdHash as New Hashtable
ColdHash.Add(1, "Potatoes")
ColdHash.Add("y", "Meat")
MsgBox(ColdHash.Item("y"))
```

In this example, the hash table contains two items (Meat and Potatoes) and the Item method retrieves the data associated with the y key, which happens to be the Meat string. As a result, the MsgBox command displays the item associated with the y key, *Meat*, in a message box.

Copying Values from a Hash Table to an Array

The Item method can retrieve data from a hash table one item at a time. If you want to retrieve all the data stored in a hash table, you can use the CopyTo method to copy everything out of a hash table and into a one-dimensional array.

To use the CopyTo method, you need to use the index position where you want to start copying data from the hash table into the array, like so:

```
HashtableName.CopyTo(ArrayName, Index)
```

where *HashtableName* is the name of your hash table, *ArrayName* is the name of your dynamic array that holds DictionaryEntry data types, and *Index* is the index position where you want to start storing data in the array. (If this value is 0, the first item from the hash table is stored at index position 0 of the array, the second item is stored in index position 1 of the array, and so on.)

A DictionaryEntry data type stores both a key and its associated value. While a data type can usually hold only a single item, a DictionaryEntry data type is specially designed to work with hash tables to hold two items: a key (either a string or a number) and any associated data (a string, a number, a collection, an array, a queue, and so on).

Before you can copy data from a hash to an array, you must resize your array using the ReDim keyword, as the following code demonstrates:

```
Dim ColdHash As New Hashtable
Dim BreakfastArray() As DictionaryEntry
Dim stuff As DictionaryEntry
ColdHash.Add(1, "Potatoes")
ColdHash.Add("y", "Meat")
ReDim BreakfastArray(ColdHash.Count - 1)
ColdHash.CopyTo(BreakfastArray, 0)
For Each stuff In BreakfastArray
MsgBox(stuff.Key & "   " & stuff.Value)
Next
```

This example creates a dynamic array that can hold DictionaryEntry data types and a variable that can also hold DictionaryEntry data types. After adding the string Potatoes to the hash table and the string Meat after that, the program next resizes the BreakfastArray variable to hold the number of items in the hash table minus one.

NOTE *Remember that the first array element is at index 0. Because the total number of items stored in the hash table is 2, the program resizes the array that consists of two elements, one at index position 0 and a second at index position 1, which is the value of 2, the total number of items in the hash table minus 1.*

The `CopyTo` method copies all the data out of the hash table and stores it in the `BreakfastArray`, starting at index position 0, the first position of the array. Next, the For Each loop starts at the beginning of the hash table and displays its contents, which includes both the key and the data.

The For Each loop starts at the beginning of the array and displays each key/data pair in a message box. The first message box displays *y Meat* and the second message box displays *1 Potatoes*.

Counting Data in a Hash Table

Because a hash table can grow and shrink depending on the amount of data stored in it at any given time, use the `Count` method to count the number of items currently stored in a hash table, with this syntax:

```
HashtableName.Count
```

where *HashtableName* is the name of the hash table that contains the data you want to count.

The following code adds two items to a hash table and then uses the `Count` method to count the number of items stored in the hash table:

```
Dim ColdHash as New Hashtable
ColdHash.Add(1, "Potatoes")
ColdHash.Add("y", "Meat")
MsgBox(ColdHash.Count)
```

After adding two items to the hash table, the `Count` method returns a value of `2`, which the `MsgBox` command displays in a message box.

Checking Whether Keys or Data Are Stored in a Hash Table

Once you've added data to a hash table, you can check to see whether the hash table contains a certain key or chunk of data, using the `ContainsKey` or `ContainsValue` methods. Both methods return a True or False value.

The `ContainsKey` method works like this:

```
HashtableName.ContainsKey(Key)
```

where *HashtableName* is the name of the hash table that contains the data you want to count and *Key* is the key you want to look for in the hash table.

The `ContainsValue` method works like this:

```
HashtableName.ContainsValue(Value)
```

where *HashtableName* is the name of the hash table that contains the data you want to count and *Value* is the data you want to look for in the hash table.

NOTE *Neither the `ContainsKey` nor `ContainsValue` method alters or removes the data from the hash table.*

The following code adds two items to a hash table and then uses the ContainsKey and ContainsValue methods to determine whether the hash table contains a specific key or value:

```
Dim ColdHash as New Hashtable
ColdHash.Add(1, "Potatoes")
ColdHash.Add("y", "Meat")
MsgBox(ColdHash.ContainsKey(1))
MsgBox(ColdHash.ContainsValue("MSG"))
```

In this example, the first MsgBox command displays a True value because the ContainsKey method finds the key 1 in the hash table. However, the second MsgBox command displays a False value because the ContainsValue method searches for MSG stored in the hash table. Because MSG isn't stored in the hash table, the ContainsValue method returns a False value.

Removing Data from a Hash Table

Just as you can add data to a hash table, you can also remove data, either by removing items one at a time or by clearing out the entire hash table all at once. To wipe out all the data in a hash table, use the Clear method:

```
HashtableName.Clear()
```

where *HashtableName* is the name of the hash table that contains the data you want to count.

The following code adds two items to a hash table and then uses the Count method to count all items in the hash table and display the total in a message box using the MsgBox command. The first MsgBox command displays *2*. Then the Clear method runs and empties the hash table before the second MsgBox command runs and displays *0*.

```
Dim ColdHash as New Hashtable
ColdHash.Add(1, "Potatoes")
ColdHash.Add("y", "Meat")
MsgBox(ColdHash.Count)
ColdHash.Clear()
MsgBox(ColdHash.Count)
```

Rather than clear out an entire hash table, you can use the Remove method to selectively remove individual items from a hash table by deleting the data associated with a specific key, like so:

```
HashtableName.Remove(Key)
```

where *HashtableName* is the name of the hash table that contains the data you want to count and *Key* is the key associated with the data you want to remove.

The following code adds two items to a hash table and then removes the data associated with the y key:

```
Dim ColdHash as New Hashtable
ColdHash.Add(1, "Potatoes")
ColdHash.Add("y", "Meat")
MsgBox(ColdHash.Count)
```

```
ColdHash.Remove("y")
MsgBox(ColdHash.Count)
```

This example uses the `Count` method to display the number of items in the hash table (in this case, *2*), then it removes the data associated with the `y` key, which happens to be the `"Meat"` string. Next, the `Count` method counts the total number of items in the hash table and displays that in a message box, which is now *1*.

Hands-on Tutorial: Playing with Queues, Stacks, and Hash Tables

This tutorial stores identical data in a queue, stack, and hash table so you can see how different data structures store identical information.

1. Start a new project, then click OK to display a blank form.
2. Choose View ▸ Toolbox to display the Toolbox.
3. Click the Button control in the Toolbox, mouse over the form, and click the left mouse button to create a button on the form.
4. Double-click the Button1 control on the form. The Code window appears.
5. Type the following between the `Private Sub` and `End Sub` lines so the `Button1_Click` event procedure in the Code window looks like this:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)_
Handles Button1.Click
        Dim MyQueue As New Queue
        Dim MyStack As New Stack
        Dim MyHashTable As New Hashtable
        Const Data1 As Integer = 29
        Const Data2 As String = "Joe Smith"
        Const Data3 As Single = 39.05
        MyQueue.Enqueue(Data1)
        MyQueue.Enqueue(Data2)
        MyQueue.Enqueue(Data3)

        MyStack.Push(Data1)
        MyStack.Push(Data2)
        MyStack.Push(Data3)

        MyHashTable.Add(1, Data1)
        MyHashTable.Add(2, Data2)
        MyHashTable.Add(3, Data3)

        MsgBox(MyQueue.Peek)
        MsgBox(MyStack.Peek)
        MsgBox(MyHashTable.Item (2))
End Sub
```

- The first three lines (which start with the `Dim` keyword) create a queue, a stack, and a hash table, respectively. The fourth through sixth lines, which start with the `Const` keyword, which defines a variable with a fixed or constant value, create three constant values: an integer (`29`), a string, (`"Joe Smith"`), and a single-precision number (`39.05`).

- The next three lines add the constant values (`29`, `"Joe Smith"`, and `39.05`) to the queue data structure using the `Enqueue` method.
- The next three lines add the constant values (`29`, `"Joe Smith"`, and `39.05`) to the stack data structure using the `Push` method.
- The next three lines add the constant values (`29`, `"Joe Smith"`, and `39.05`) to the hash table using the `Add` method. In addition, each item is assigned a key so the first item gets a key of `1`, the second gets a key of `2`, and the third gets a key of `3`.
- The last three lines use the `MsgBox` command to show you the contents of one item in each data structure. The first `MsgBox` command uses the `Peek` method to show you the next item that you can remove from the queue, which is the number *29*, the first item stored in the queue.
- The second `MsgBox` command uses the `Peek` method to show the next item that you can pop off the stack, which is the number *39.05*, the last item added to the stack.
- The third `MsgBox` command uses the `Item` method to retrieve the item identified by the `2` key value, which happens to be the *Joe Smith* string added to the hash table.

6. Press F5. Your user interface appears.
7. Click the Button1 control. The first `MsgBox` command displays a message box with the number *29*.
8. Click OK. The second `MsgBox` command displays a message box with the number *39.05*. Notice that the last item added to a stack is the first item you can remove from a stack, which is the exact opposite of adding and removing data to a queue.
9. Click OK. The third `MsgBox` command displays a message box with the string *Joe Smith* in it. Click OK.
10. Press ALT-F4 to stop your program, then choose File ▸ Close Project. A Close Project dialog box appears. Click Discard.

KEY FEATURES TO REMEMBER

Queues, stacks, and hash tables are just fancier ways to store data than an array or a collection. All three types of data structures can store object data types, which means you can mix different types of data in these data structures.

- A queue is known as a FIFO (First In, First Out) data structure because the first item stored in the queue is also the first item retrieved from the queue.
- A stack is known as a LIFO (Last In, First Out) data structure because the first item stored on the stack is the last item retrieved.
- Adding data to a stack is known as pushing data on the stack. Removing data from a stack is known as popping data from the stack.
- A hash table stores data along with a unique key that identifies that specific chunk of data.