

# CHAPTER 7



Contax T

Acrobat Man

## Stacks

### LEARNING OBJECTIVES

- Describe the operation of a stack
- Describe the characteristics of Last In/First Out (LIFO) processing
- List at least four examples of applications that require stacks
- State the type of data structure used to implement the `Stack` and `Stack<T>` classes
- Describe what it means to push items onto a stack
- Describe what it means to pop items off of a stack
- Describe the behavior of a pop operation
- List and describe the members of the `Stack` and `Stack<T>` classes
- Use the non-generic `Stack` class in a program
- Use the generic `Stack<T>` class in a program
- Describe the functionality provided by each interface implemented by the `Stack` class
- Describe the functionality provided by each interface implemented by the `Stack<T>` class

---

## INTRODUCTION

---

Stacks play a critical role in the world of computers. Microprocessors and virtual machines utilize stacks (stack frames) to implement procedure call chaining; compilers utilize stacks to parse programming languages; application software might use a stack to implement an operation undo capability.

In this chapter I will explain how stacks work and show you an example of a custom coded stack so you can see how they work internally. I'll then discuss the `Stack` and `Stack<T>` classes in detail and present a comprehensive, non-trivial example showing each of these classes in action.

When you finish this chapter you'll have a solid understanding of how stacks work and why they are an important data structure.

---

## STACK OPERATIONS

---

A stack is a special kind of list whose elements or items are stored and accessed in last-in/first-out (LIFO) sequence. All insertions and deletions to a stack occur at only one end of the list. The business end of a stack goes by a special name: "Top".

### CHARACTERISTIC STACK OPERATIONS

A stack data structure supports two primary operations: *push* and *pop*. A third operation called *peek* also comes in handy. These operations are discussed in detail below.

#### **Push**

The push operation adds an item to the top of a stack. Subsequent calls to push add newer items to the top of the stack. The number of items contained in the stack increments by one with each push.

#### **Pop**

The pop operation removes the top element from the stack. The last item pushed onto the stack will be the first item popped off of the stack. (LIFO). The number of items contained in the stack is reduced by one with each pop.

#### **Peek**

The peek operation is used to examine the item at the top of the stack in place without removing the item.

### AN ILLUSTRATION WILL HELP

Figure 7-1 shows a representation of a stack and the effects of several push and pop operations.

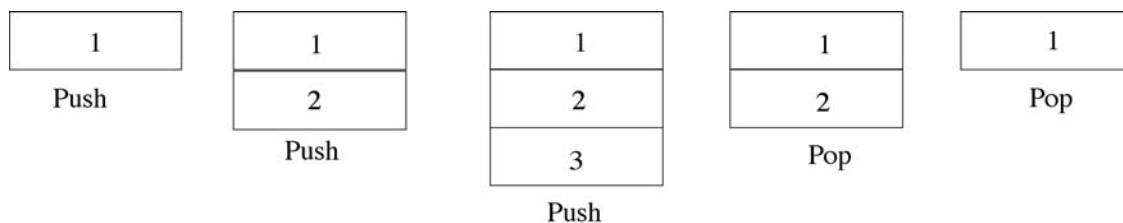


Figure 7-1: Stack Showing Effects of Push and Pop Operations

Referring to figure 7-1 — in this picture I have represented the stack as growing downward with each successive push operation. That is, the *top* of the stack grows downward with each successive push operation. After the first push operation item 1 sits on the top of the stack. After the second push operation item 2 sits on the top of the stack, and finally, after the third push operation, item 3 sits on top of the stack. Item 3 is the first element removed as a result of the first pop operation. Item 2 is removed as a result of the second call to pop, which leaves item 1 at the top of the stack.

## WHAT'S ACTUALLY BEING PUSHED AND POPPED?

That's a good question, and since this is a book about C# and the .NET Framework, the answer is one of two things: 1) you're either going to push a value type object, or 2) a reference type object. It's important to know the difference between the two.

### ***PUSHING A VALUE TYPE OBJECT ONTO A STACK***

A value type object implements value type copy semantics. By this I mean that when one value type is assigned to another, the value of one is copied to the other. In the .NET Framework, value type objects are structures and are defined using the `struct` keyword. If the structure is complex and contains many fields, each field's value will be copied from one instance of the value type object to another. This value type copy behavior is implemented automatically by the .NET runtime environment.

Now, when you use the non-generic version of `Stack`, found in the `System.Collections` namespace, you will encounter a performance penalty when pushing value types onto the stack. This performance penalty occurs because value types must be boxed into objects before being pushed onto the non-generic stack. The end result is that a reference to the boxed value type is actually pushed onto the stack and the boxed value type object is created on the heap.

If, on the other hand, you use the generic `Stack<T>` class and specify a value type for 'T', the resulting data structure is optimized for that value type and no boxing or unboxing occurs. However, the performance penalty you incur with pushing and popping will be commensurate with the complexity of the value type in question.

### ***PUSHING A REFERENCE TYPE OBJECT ONTO A STACK***

The result of pushing a reference type onto a stack is that the stack contains only references to objects in the heap. What you must be aware of in this situation is the number of active references that point to the same object. For example, suppose you have a reference R to object O. If you push R onto the stack, the top of the stack now points to O as well. Two references to O are now active. The danger of having too many active references to one object is that as long as there is one active reference to an object the .NET runtime garbage collector cannot free up and reclaim the memory for future use. This advice applies not only to the use of stacks, but to .NET programming in general.

### ***VALUE TYPE BOXING IN ACTION***

Example 7.1 gives the code for a short program that pushes 25 million integers onto two different types of stacks: the non-generic `System.Collections.Stack`, and the generic `System.Collections.Generic.Stack<T>`,

*7.1 PushValueTypeDemo.cs*

```

1      using System;
2      using System.Collections;
3      using System.Collections.Generic;
4
5      public class PushValueTypeDemo {
6          public static void Main(){
7              Stack stack1 = new Stack();
8              Stack<int> stack2 = new Stack<int>();
9              const int COUNT = 25000000;
10
11             DateTime start = DateTime.Now;
12             for(int i = 0; i < COUNT; i++){
13                 stack1.Push(i);
14             }
15             TimeSpan elapsed_time = (DateTime.Now - start);
16             Console.WriteLine("Time to push {0:N} integers to non-generic stack: {1}", COUNT, elapsed_time);
17
18             start = DateTime.Now; // reset start time

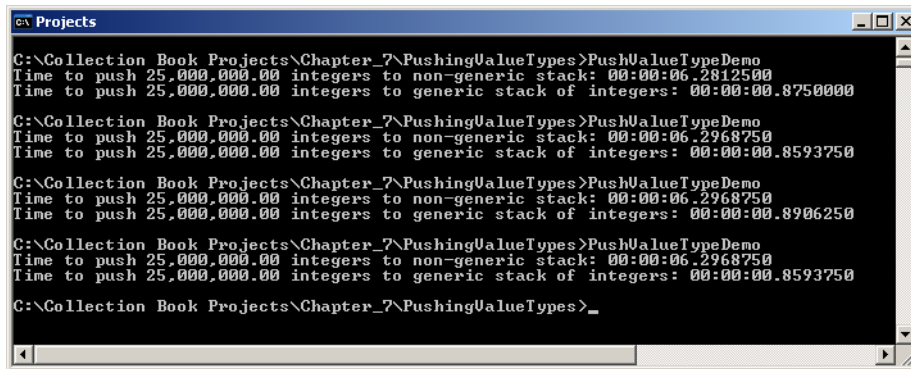
```

```

19     for(int i = 0; i < COUNT; i++){
20         stack2.Push(i);
21     }
22     elapsed_time = (DateTime.Now - start);
23     Console.WriteLine("Time to push {0:N} integers to generic stack of integers: {1}",
24         COUNT, elapsed_time);
25 }
26

```

Referring to example 7.1 — on lines 7 and 8 an instance each of `Stack` and `Stack<int>` is created followed by the definition of a constant named `COUNT` which is initialized to 25,000,000. On line 11, the `DateTime.Now` property is used to initialize the variable named `start`. In the first `for` loop which begins on the next line, 25 million integers are pushed onto the non-generic stack. When the `for` loop exits, the elapsed time is calculated and the results printed to the console. The same process is then repeated with the generic `Stack<int>`. Figure 7-2 shows the results of running this program.



```

C:\Collection Book Projects\Chapter_7\PushingValueTypes>PushValueTypeDemo
Time to push 25,000,000.00 integers to non-generic stack: 00:00:06.2812500
Time to push 25,000,000.00 integers to generic stack of integers: 00:00:00.8750000

C:\Collection Book Projects\Chapter_7\PushingValueTypes>PushValueTypeDemo
Time to push 25,000,000.00 integers to non-generic stack: 00:00:06.2968750
Time to push 25,000,000.00 integers to generic stack of integers: 00:00:00.8593750

C:\Collection Book Projects\Chapter_7\PushingValueTypes>PushValueTypeDemo
Time to push 25,000,000.00 integers to non-generic stack: 00:00:06.2968750
Time to push 25,000,000.00 integers to generic stack of integers: 00:00:00.8906250

C:\Collection Book Projects\Chapter_7\PushingValueTypes>PushValueTypeDemo
Time to push 25,000,000.00 integers to non-generic stack: 00:00:06.2968750
Time to push 25,000,000.00 integers to generic stack of integers: 00:00:00.8593750

C:\Collection Book Projects\Chapter_7\PushingValueTypes>_

```

Figure 7-2: Results of Running Example 7.1

Referring to figure 7-2 — I executed the program four times. In each run the boxing of integer value types as they are pushed onto the non-generic stack extracted a performance penalty.

## Disassembling Example 7.1

If you use the MSIL disassembler to disassemble the executable file created by compiling example 7.1 you'll get an output that looks similar to example 7.2. (Note: This is the disassembled `Main()` method.)

*7.2 Disassembled `Main()` Method from Example 7.1*

```

1     .method public hidebysig static void  Main() cil managed
2     {
3         .entrypoint
4         // Code size          177 (0xb1)
5         .maxstack 3
6         .locals init (class [mscorlib]System.Collections.Stack V_0,
7             class [System]System.Collections.Generic.Stack`1<int32> V_1,
8             valuetype [mscorlib]System.DateTime V_2,
9             int32 V_3,
10            valuetype [mscorlib]System.TimeSpan V_4,
11            bool V_5)
12            IL_0000: nop
13            IL_0001: newobj          instance void [mscorlib]System.Collections.Stack::.ctor()
14            IL_0006: stloc.0
15            IL_0007: newobj          instance void class [System]System.Collections.Generic.Stack`1<int32>::.ctor()
16            IL_000c: stloc.1
17            IL_000d: call          valuetype [mscorlib]System.DateTime [mscorlib]System.DateTime::get_Now()
18            IL_0012: stloc.2
19            IL_0013: ldc.i4.0
20            IL_0014: stloc.3
21            IL_0015: br.s          IL_002a
22            IL_0017: nop
23            IL_0018: ldloc.0
24            IL_0019: ldloc.3
25            IL_001a: box          [mscorlib]System.Int32
26            IL_001f: callvirt     instance void [mscorlib]System.Collections.Stack::Push(object)
27            IL_0024: nop
28            IL_0025: nop
29            IL_0026: ldloc.3

```

```

30     IL_0027: ldc.i4.1
31     IL_0028: add
32     IL_0029: stloc.3
33     IL_002a: ldloc.3
34     IL_002b: ldc.i4      0x17d7840
35     IL_0030: clt
36     IL_0032: stloc.s     V_5
37     IL_0034: ldloc.s     V_5
38     IL_0036: brtrue.s    IL_0017
39     IL_0038: call        valuetype [mscorlib]System.DateTime [mscorlib]System.DateTime::get_Now()
40     IL_003d: ldloc.2
41     IL_003e: call        valuetype [mscorlib]System.TimeSpan
[mscorlib]System.DateTime::op_Subtraction(valuetype [mscorlib]System.DateTime, valuetype
[mscorlib]System.DateTime)
42     IL_0043: stloc.s     V_4
43     IL_0045: ldstr      "Time to push {0:N} integers to non-generic stack: "
44     + "{1}"
45     IL_004a: ldc.i4      0x17d7840
46     IL_004f: box         [mscorlib]System.Int32
47     IL_0054: ldloc.s     V_4
48     IL_0056: box         [mscorlib]System.TimeSpan
49     IL_005b: call        void [mscorlib]System.Console::WriteLine(string,
50                                           object,
51                                           object)
52     IL_0060: nop
53     IL_0061: call        valuetype [mscorlib]System.DateTime [mscorlib]System.DateTime::get_Now()
54     IL_0066: stloc.2
55     IL_0067: ldc.i4.0
56     IL_0068: stloc.3
57     IL_0069: br.s       IL_0079
58     IL_006b: nop
59     IL_006c: ldloc.1
60     IL_006d: ldloc.3
61     IL_006e: callvirt    instance void class [System]System.Collections.Generic.Stack`1<int32>::Push(!0)
62     IL_0073: nop
63     IL_0074: nop
64     IL_0075: ldloc.3
65     IL_0076: ldc.i4.1
66     IL_0077: add
67     IL_0078: stloc.3
68     IL_0079: ldloc.3
69     IL_007a: ldc.i4      0x17d7840
70     IL_007f: clt
71     IL_0081: stloc.s     V_5
72     IL_0083: ldloc.s     V_5
73     IL_0085: brtrue.s    IL_006b
74     IL_0087: call        valuetype [mscorlib]System.DateTime [mscorlib]System.DateTime::get_Now()
75     IL_008c: ldloc.2
76     IL_008d: call        valuetype [mscorlib]System.TimeSpan
[mscorlib]System.DateTime::op_Subtraction(valuetype [mscorlib]System.DateTime, valuetype
[mscorlib]System.DateTime)
77     IL_0092: stloc.s     V_4
78     IL_0094: ldstr      "Time to push {0:N} integers to generic stack of in"
79     + "tegers: {1}"
80     IL_0099: ldc.i4      0x17d7840
81     IL_009e: box         [mscorlib]System.Int32
82     IL_00a3: ldloc.s     V_4
83     IL_00a5: box         [mscorlib]System.TimeSpan
84     IL_00aa: call        void [mscorlib]System.Console::WriteLine(string,
85                                           object,
86                                           object)
87     IL_00af: nop
88     IL_00b0: ret
89     } // end of method PushValueTypeDemo::Main

```

Referring to example 7.2 — OK, before your eyes roll up into your skull take a deep breath. This will be easier to understand than you first think. It can be intimidating to decipher MSIL instructions your first time through. In this example, however, you'll only need to understand a handful of instructions. So here goes.

First, a word about the layout of the file. There are three columns. The leftmost column contains the IL address and other directives. The second column contains symbolic instructions, and the third column, if it contains anything, will have variable names, constant values, method names, object names, etc. These will be easy to figure out as you begin to get familiar with the code. I'm just going to discuss the first half of the code, the part that contains the first `for` loop of example 7.1.

Starting at the top of the listing on line 1, the text there signifies that this is a `Main` method. Line 2 contains an opening brace, and line 3 contains a directive that says this is the entry point. (`.entrypoint`). Line 4 contains a comment indicating the size of the code. The `.maxstack` directive on line 5 indicates the maximum amount of evaluation

stack space the program will utilize. You'll see the evaluation stack in action shortly. Lines 6 through 11 contain local variable declarations named `V_0` through `V_5`. `V_0` is the reference to the non-generic Stack object. `V_1` is the reference to the generic `Stack<int>` object. (Here denoted as `Stack<int32>`.) `V_2` is a `DateTime` variable and corresponds to the `start` variable declared in example 7.1. `V_3` corresponds to the counting variable `i` declared in each of the `for` loops. `V_4` corresponds to the `TimeSpan` variable `elapsed_time`. Finally, `V_5` is the boolean value that is required to evaluate each of the `for` loops.

Line 12 contains a `nop` (no operation) instruction. Line 13 creates an instance of the non-generic Stack and leaves its reference on the evaluation stack. The `stloc.0` instruction on line 14 pops the value from the evaluation stack and loads it into local variable 0 (`V_0`). On line 15 an instance of the generic `Stack<int>` object is created and its reference is left on the evaluation stack. The `stloc.1` instruction on the next line pops the reference off the evaluation stack and assigns it to local variable 1 (`V_1`). On line 17, the `call` instruction makes a method call to the `DateTime.get_Now()` method. (*Under the covers, properties translate into method calls.*) The resulting value obtained from that call is pushed onto the evaluation stack, and the next instruction, `stloc.2`, pops the value off the evaluation stack and assigns it to local variable 2 (`V_2`). On line 19, the `ldc.i4.0` instruction loads the value 0 onto the evaluation stack. The next instruction pops this value off the evaluation stack and assigns it to local variable 3 (`V_3`).

Now we're ready to get going on the loop. Line 21 contains a `br.s` instruction. This says to branch unconditionally to address `IL_002a`, which you'll find on line 33. The `ldloc.3` instruction pushes the value of local variable 3 onto the evaluation stack. Next, the `ldc.i4` instruction pushes the value of `0x17d7840` (hexadecimal for 25 million) onto the stack. This is followed by the `ctl` instruction (compare less than). So, the first time around 0 is less than 25 million, so the result will be true or 1 and this value is pushed onto the evaluation stack. On line 36, the `stloc.s` instruction pops this value from the stack and assigns it to local variable `V_5` (the boolean variable). This value is then pushed back onto the stack in preparation for the next instruction on line 38 which is `brtrue.s` which tells the VM to branch to address `IL_0017` if the value on the top of the evaluation stack is 1. Going to line 22 we see a `nop` instruction. On line 23 the `ldloc.o` loads the value of local variable 0 (`V_0` -- the reference to the non-generic stack) onto the evaluation stack. Next, the `ldloc.3` instruction loads the value of local variable 3 (the counting variable `i`, which is zero now.) onto the evaluation stack. On line 25, the `box` instruction boxes the value on top of the evaluation stack and then pushes it onto the stack on the next line with a `callvirt` instruction to the non-generic `Stack.Push()` method. Following two `nop` instructions the value of local variable 3 is pushed onto the evaluation stack followed by the instruction on line 30, `ldc.i4.1`, which pushes the value 1 onto the evaluation stack. These two values are added with the `add` instruction on line 31 and the result is popped from the stack and assigned to local variable 3 (`V_3`). In this way the counting variable `i` is incremented by one. Thus, the loop repeats in this fashion for 25 million iterations.

When, after 25 million iterations, the result of the comparison of the counting variable `i` and the constant `COUNT` results in false, the `brtrue.s` instruction will fail and execution will fall through to the instruction on line 39. This is where the program calculates how long it took to execute the first `for` loop. On line 39, a call to `DateTime.get_Now()` pushes the resulting value onto the evaluation stack. Next, on line 40, the value of local variable `V_2` is pushed onto the stack. (`V_2` contains the `start` value.) The instruction on line 42 performs a `TimeSpan` subtraction using the two `DateTime` values on the stack leaving the result on top of the stack. The `stloc.s` instruction on line 43 pops this value off the stack and stores it in local variable `V_4`, which corresponds to the `elapsed_time` variable in example 7.1. On line 44, the `ldstr` (load string) instruction loads a reference to the string literal indicated in quotes on the evaluation stack. This is followed by the `ldc.i4` instruction which load the value `0x17d7840` (25 million decimal) onto the evaluation stack followed by a call to the `box` instruction to box the value. Next, local variable `V_4` is loaded onto the evaluation stack, and since a `DateTime` value is a value type, it's boxed as well. The state of the stack now is a reference to a string, a reference to a boxed integer, and a reference to a boxed `DateTime` value. Finally, on line 50, a call to the `Console.WriteLine()` method prints the string and the two values to the console.

Any questions? Note that this was a great exercise because you got to see not only how C# source code is translated into MSIL instructions, but how the .NET runtime uses a stack to hold values during execution. I'll leave the tracing of the second `for` loop to you as an exercise.

## Quick Review

A stack is a specialized list whose elements are stored in last-in/first-out (LIFO) order. Stacks support two primary operations: push and pop. The push operation stores an item on top of the stack. As more items are pushed onto the stack, the older items move deeper into the stack while younger items are at the top of the stack. The most recent

item pushed onto the stack will always be at the top of the stack. The pop operation removes the most recently pushed item from the top of the stack.

---

## A HOME GROWN STACK

---

In this section I want to show you how to use an array to implement a stack. Example 7.3 gives the code for a class I call HomeGrownStack.

7.3 HomeGrownStack.cs

```

1      using System;
2
3      public class HomeGrownStack {
4
5          private object[] stack_contents;
6          private int top = -1;
7          private const int INITIAL_SIZE = 25;
8
9          public HomeGrownStack(int initial_size){
10             stack_contents = new object[initial_size];
11         }
12
13         public HomeGrownStack():this(INITIAL_SIZE){ }
14
15         public bool IsEmpty {
16             get { return (top == -1); }
17         }
18
19         public void Push(object item){
20             if(item == null){
21                 throw new ArgumentException("Cannot push null item onto stack!" );
22             }
23
24             if(++top >= stack_contents.Length){
25                 GrowStack();
26             }else{
27                 stack_contents[top] = item;
28             }
29         } // end Push method
30
31         public object Pop(){
32             if(IsEmpty){
33                 throw new InvalidOperationException("The stack is empty!");
34             }
35             object return_object = stack_contents[top];
36             stack_contents[top--] = null;
37             return return_object;
38         } // end Pop method
39
40         public object Peek(){
41             if(IsEmpty){
42                 throw new InvalidOperationException("The stack is empty!");
43             }
44             return stack_contents[top];
45         } // end Peek method
46
47         private void GrowStack(){
48             object[] temp_array = new object[stack_contents.Length];
49             for(int i = 0; i < stack_contents.Length; i++){
50                 temp_array[i] = stack_contents[i];
51             }
52
53             stack_contents = new object[stack_contents.Length * 2];
54
55             for(int i = 0; i < temp_array.Length; i++){
56                 stack_contents[i] = temp_array[i];
57             }
58         } // end GrowArray method
59
60     } // end class definition
61
62

```

Referring to example 7.3 — the HomeGrownStack class contains three fields: an array of objects named `stack_contents`, an integer variable named `top` that points to the top of the stack, and a constant named `INITIAL_SIZE` which I have initialized to 25. On line 9 the constructor method takes one integer argument that sets

the size of the stack. It uses this parameter to create the object array. The default constructor on line 13 simply calls the first constructor while supplying the INITIAL\_SIZE constant as an argument. On line 15 the IsEmpty property is defined. If the top field equals -1 it returns true, otherwise it returns false.

The Push() method definition starts on line 19. The first order of business is to ensure the incoming object reference is valid. If not, the method throws an ArgumentException. If the incoming reference is valid, the top variable is incremented and its value compared with the length of the array. If necessary, the array is dynamically grown to accommodate new items, otherwise there's enough room in the array to push the incoming reference which is assigned to the element pointed to by top.

The Pop() method begins on line 31. First, the method checks to see if the stack is empty. If so, an InvalidOperationException is thrown and the method exits. Otherwise, the element pointed to by top is returned, top is set to null and decremented by 1. The item removed from the array is returned and the method exits.

The Peek() method on line 40 throws an InvalidOperationException if the stack is empty, otherwise it returns a reference to the element on top of the stack but does not remove the element.

The GrowStack() method beginning on line 48 simply grows the array when the value of top approaches the value of the length of the array.

Example 7.4 gives a short program showing the HomeGrownStack in action. This short program reverses the order of a set of integers.

7.4 MainApp.cs (Demonstrating HomeGrownStack)

```

1      using System;
2
3      public class MainApp {
4          public static void Main(){
5              HomeGrownStack stack = new HomeGrownStack();
6              for(int i = 0; i < 37; i++){
7                  stack.Push(i);
8              }
9
10             for(int i = 0; i < 37; i++){
11                 Console.Write(stack.Pop() + " ");
12             }
13             Console.WriteLine();
14
15             // try one more Pop operation
16             try{
17                 stack.Pop();
18             }catch(Exception e){
19                 Console.WriteLine(e);
20             }
21         }
22     }
23 }

```

Referring to example 7.4 — an instance of HomeGrownStack is created on line 5. The for loop on line 6 pushes 38 integer values onto the stack. The for loop on line 10 pops each integer off the stack and writes its value to the console. This has the effect of reversing the sequence of integers generated by the for loop. Finally, one more call to the Pop() method is made inside of a try/catch block. The results of running this program appear in figure 7-3.

```

C:\Collection Book Projects\Chapter_7\HomeGrownStack>mainapp
36 35 34 33 32 31 30 29 28 27 26 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
System.InvalidOperationException: The stack is empty!
   at HomeGrownStack.Pop()
   at MainApp.Main()
C:\Collection Book Projects\Chapter_7\HomeGrownStack>

```

Figure 7-3: Results of Running Example 7.4

## Quick Review

The HomeGrownStack class demonstrates the use of an array to contain stack items. It implements the Push(), Pop(), and Peek() methods as well as the IsEmpty property. The top field is incremented each time an item is pushed onto the stack and decremented each time an item is popped off the stack.

---

## THE STACK CLASS

---

In this section I discuss the non-generic Stack class which is found in the System.Collections namespace. I'll present its inheritance hierarchy and talk about some of the operations it supports in addition to the basic stack operations push and pop.

As a non-generic collection, the Stack class pushes and pops any type of object. Value type objects, as I demonstrated earlier, are boxed before being pushed onto the stack. When you pop an object off the stack you must cast it to its proper type. If it's a value type object it will undergo an unboxing operation as well.

### STACK CLASS INHERITANCE HIERARCHY

Figure 7-4 gives the UML class diagram for the System.Collections.Stack class inheritance hierarchy.

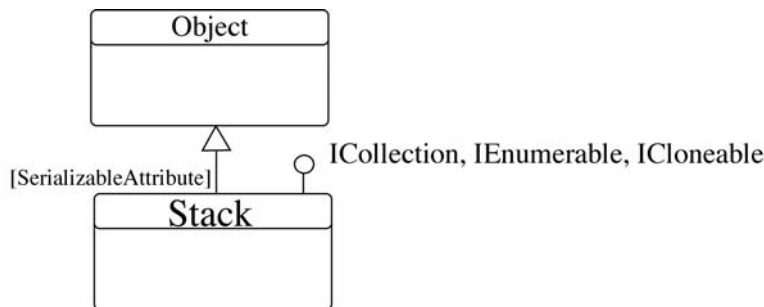


Figure 7-4: System.Collections.Stack Class Inheritance Hierarchy

Referring to figure 7-4 — the Stack class implicitly extends System.Object and implements the ICollection, IEnumerable, and ICloneable interfaces. It's 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 stack using the `foreach` statement. The direction of iteration begins with the stack's top element and ends with the oldest element on the stack.

#### **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 stack 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 stack.

### BALANCED SYMBOL CHECKER

The following example shows the Stack class in action. The `BalancedSymbolChecker` class implements logic to parse a sequence of characters and look for balanced sets of parenthesis `[] ( [] ) []`, braces `[] { [] } []`, and brackets `[] [ [] ] []`.

7.5 *BalancedSymbolChecker.cs*

```

1  using System;
2  using System.Collections;
3
4  public class BalancedSymbolChecker {
5
6      private const char OPEN_PAREN = '(';
7      private const char CLOSE_PAREN = ')';
8      private const char OPEN_BRACKET = '[';
9      private const char CLOSE_BRACKET = ']';
10     private const char OPEN_BRACE = '{';
11     private const char CLOSE_BRACE = '}';
12
13
14     public char GetNextSymbol(){
15         char c;
16         do {
17             if((c = (char)Console.Read()) == '\r'){
18                 return '\0';
19             }
20         }while( (c != OPEN_PAREN) && (c != CLOSE_PAREN) && (c != OPEN_BRACKET) && (c != CLOSE_BRACKET)
21             && (c != OPEN_BRACE) && (c != CLOSE_BRACE) );
22
23         return c;
24     } // end GetNextSymbol method
25
26     public bool CheckMatch(char openSymbol, char closeSymbol){
27         if((openSymbol == OPEN_PAREN) && (closeSymbol != CLOSE_PAREN) ||
28            (openSymbol == OPEN_BRACKET) && (closeSymbol != CLOSE_BRACKET) ||
29            (openSymbol == OPEN_BRACE) && (closeSymbol != CLOSE_BRACE) ) {
30             Console.WriteLine("Open Symbol " + openSymbol + " does not match " + closeSymbol);
31             return false;
32         }
33         return true;
34     }
35
36     public bool CheckBalance(){
37         char c, match;
38         int errors = 0;
39
40         Stack pendingTokens = new Stack();
41         while((c = GetNextSymbol()) != '\0'){
42             switch(c){
43                 case OPEN_PAREN:
44                 case OPEN_BRACKET:
45                 case OPEN_BRACE: pendingTokens.Push(c);
46                     break;
47                 case CLOSE_PAREN:
48                 case CLOSE_BRACKET:
49                 case CLOSE_BRACE: {
50                     if(pendingTokens.Count == 0){
51                         Console.WriteLine("Invalid symbol sequence: " + c);
52                         return false;
53                     }else{
54                         match = (char)pendingTokens.Pop();
55                         if(! CheckMatch(match, c)){
56                             return false;
57                         }else{
58                             Console.WriteLine("Matching symbols {0} and {1} found", match, c);
59                         }
60                     }
61                     break;
62                 }
63             }
64         }
65
66         while(pendingTokens.Count > 0){
67             match = (char) pendingTokens.Pop();
68             Console.WriteLine("Unmatched symbol: " + match);
69             errors++;
70         }
71         return (errors > 0) ? false:true;
72     }
73 } // end BalancedSymbolChecker class definition
74

```

Referring to example 7.5 — the `BalancedSymbolChecker` class defines a set of constants that represent each of the six symbols of interest. It defines three methods: `GetNextSymbol()`, `CheckMatch()`, and `CheckBalance()`. The `GetNextSymbol()` method uses the `Console.Read()` method to read a line of text from the console. Each subsequent call to the `Console.Read()` method will return the next character from the line of text until it encounters the end-of-

line sequence. In this example I have used the carriage return `\r` character to signal the end of the character sequence. The `GetNextSymbol()` method ignores all characters other than one of the six symbols. When it encounters one of the six symbols it returns that symbol, otherwise it returns `\0` to signal it has reached the end of the character sequence.

The `CheckMatch()` method compares two symbols with each other. If they match it returns true; if not it returns false.

These two methods are used in the `CheckBalance()` method which begins on line 36. On line 40, a `Stack` named `pendingTokens` is used to hold symbols for future evaluation. Most of the action happens within the body of the `while` loop starting on line 41. While there is a symbol to evaluate it is presented to the `switch` statement on line 42. If it's an opening symbol it's pushed onto the stack. If it's a closing symbol and the `pendingTokens.Count == 0` then it has encountered an invalid symbol sequence. If the `pendingTokens` stack contains symbols, the last symbol is popped off the stack and compared with the closing symbol. If they match, a message is written to the console displaying the matching symbols, if not, the method returns false.

Example 7.6 offers a short program demonstrating the use of the `BalancedSymbolChecker` class.

*7.6 MainApp.cs (Demonstrating BalancedSymbolChecker)*

```

1      using System;
2
3      public class MainApp {
4          public static void Main() {
5              BalancedSymbolChecker checker = new BalancedSymbolChecker();
6              char c = '\0';
7              while(((c = checker.GetNextSymbol()) != '\0')) {
8                  Console.Write(c + " ");
9              }
10             Console.WriteLine();
11             Console.WriteLine("-----");
12             checker.CheckBalance();
13         }
14     }

```

Referring to example 7.6 — an instance of `BalancedSymbolChecker` is created on line 5. The `while` loop uses the `GetNextSymbol()` method to read a line of characters from the console and print the extracted symbols to the console. On line 12 the `CheckBalance()` method is called, which will parse another line of characters and check the input symbols for balance. Figure 7-5 shows the results of running this program.

```

C:\Collection Book Projects\Chapter_7\BalancedSymbolChecker>mainapp
878<8783<738<<yuihju io i>>[[[]]><<<<>>>[[]]]>
-----
878<8783<738<<yuihju io i>>[[[]]>
Matching symbols < and > found
Matching symbols < and > found
Matching symbols < and > found
Matching symbols [ and ] found
Matching symbols [ and ] found
Matching symbols < and > found
C:\Collection Book Projects\Chapter_7\BalancedSymbolChecker>_

```

Figure 7-5: Results of Running Example 7.6

## Quick Review

The `System.Collections.Stack` is a non-generic collection which stores any type of object. Value type objects will undergo a boxing operation when they are pushed onto the stack. This results in their references being pushed onto the stack and the object it points to is created in the heap. When the value type object is popped off the stack, it will undergo an unboxing operation.

## THE STACK<T> CLASS

The generic Stack<T> class is the direct replacement for the non-generic Stack class. It provides a lot more functionality in the form of extension methods defined by the System.Linq.Enumerable class. It also provides optimal performance when used with value type objects as they do not require boxing/unboxing operations when being pushed on and popped off the stack.

### STACK<T> CLASS INHERITANCE HIERARCHY

Figure 7-6 gives the UML class diagram showing the inheritance hierarchy of the Stack<T> class.

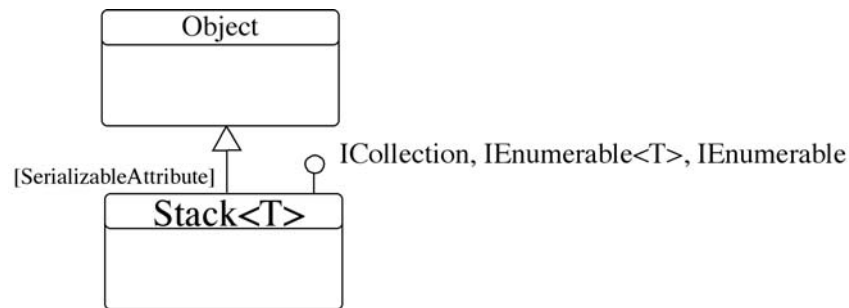


Figure 7-6: Stack<T> Class Inheritance Hierarchy

Referring to figure 7-6 — the Stack<T> class implicitly extends System.Object and implements the ICollection, IEnumerable, and IEnumerable<T> interfaces.

### FUNCTIONALITY PROVIDED BY THE IENUMERABLE INTERFACE

The IEnumerable interface, along with the supporting IEnumerator interface, enables you to iterate over the elements in the stack using the `foreach` statement. The direction of iteration begins with the stack's top element and ends with the oldest element on the stack.

### 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 stack 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 IENUMERABLE<T> INTERFACE

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

### WHAT HAPPENED TO ICOLLECTION<T>?

The Stack<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 Stack<T> class via its `Push()` method, and only remove elements via the `Pop()` and `Clear()` methods.

## COMMAND LINE POSTFIX CALCULATOR

Example 7.7 shows a generic Stack<T> class in action in a program that implements a postfix command-line calculator. The LineCalc class provides addition, subtraction, multiplication, division, and exponent operations.

*7.7 LineCalc.cs*

```

1  using System;
2  using System.Collections.Generic;
3
4  public class LineCalc {
5
6      private Stack<double> stack = new Stack<double>();
7      private const char ADD = '+';
8      private const char SUB = '-';
9      private const char MULT = '*';
10     private const char DIV = '/';
11     private const char EXP = '^';
12     private const char EQUALS = '=';
13
14
15     public void ProcessLine(string input){
16
17         try {
18             double operand = Double.Parse(input);
19             stack.Push(operand);
20
21         }catch(Exception){
22             this.ProcessOperator(input);
23         }
24     }
25
26     public void ProcessOperator(string input){
27         switch(input[0]){
28             case ADD: Add();
29                     break;
30
31             case SUB: Sub();
32                     break;
33
34             case MULT: Mult();
35                     break;
36
37             case DIV: Div();
38                     break;
39
40             case EXP: Exp();
41                     break;
42
43             case EQUALS: Equals();
44                     break;
45
46             default: Console.WriteLine("Invalid Operator!");
47                     break;
48         }
49     }
50
51     public void Add(){
52         if(stack.Count >= 2){
53             double operand_1 = stack.Pop();
54             double operand_2 = stack.Pop();
55             double result = operand_1 + operand_2;
56             stack.Push(result);
57             Console.WriteLine("Add result: {0}", result);
58         }else{
59             Console.WriteLine("Note enough operands on stack!");
60         }
61     }
62
63     public void Sub(){
64         if(stack.Count >= 2){
65             double operand_1 = stack.Pop();
66             double operand_2 = stack.Pop();
67             double result = operand_2 - operand_1;
68             stack.Push(result);
69             Console.WriteLine("Sub result: {0}", result);
70         }else{
71             Console.WriteLine("Note enough operands on stack!");
72         }
73     }
74

```

```

75     public void Mult(){
76         if(stack.Count >= 2){
77             double operand_1 = stack.Pop();
78             double operand_2 = stack.Pop();
79             double result = operand_1 * operand_2;
80             stack.Push(result);
81             Console.WriteLine("Mult result: {0}", result);
82         }else{
83             Console.WriteLine("Note enough operands on stack!");
84         }
85     }
86
87     public void Div(){
88         if(stack.Count >= 2){
89             double operand_1 = stack.Pop();
90             double operand_2 = stack.Pop();
91             double result = operand_2 / operand_1;
92             stack.Push(result);
93             Console.WriteLine("Div result: {0}", result);
94         }else{
95             Console.WriteLine("Note enough operands on stack!");
96         }
97     }
98
99     public void Exp(){
100        if(stack.Count >= 2){
101            double operand_1 = stack.Pop();
102            double operand_2 = stack.Pop();
103            double result = 1;
104            for(int i = 0; i< operand_1; i++){
105                result *= operand_2;
106            }
107            stack.Push(result);
108            Console.WriteLine("Exp result: {0}", result);
109        }else{
110            Console.WriteLine("Note enough operands on stack!");
111        }
112    }
113
114    public void Equals(){
115        if(stack.Count >= 1){
116            Console.WriteLine("Total: {0}", stack.Pop());
117        }else{
118            Console.WriteLine("Stack empty!");
119        }
120    }
121
122
123    public static void Main(){
124        LineCalc lc = new LineCalc();
125        string input = String.Empty;
126        Console.Write("Enter operand, operator, or \"quit\" to exit --> ");
127        while((input = Console.ReadLine()) != "quit"){
128            if(input.Length > 0){
129                lc.ProcessLine(input);
130            }
131            Console.Write("Enter operand, operator, or \"quit\" to exit --> ");
132        }
133    }
134 } // end LineCalc class definition
135

```

Referring to example 7.7 — the `LineCalc` program uses a generic stack of doubles (`Stack<double>`) to push and pop operands and the results of operations. The `ProcessLine()` method first assumes the input string is a valid double and tries to parse it as such. If the string fails to parse as a double an exception is thrown and it tries again to parse the string as an operator by calling the `ProcessOperator()` method in the body of the catch block. The `ProcessOperator()` method presents the first character of the input string (`input[0]`) to the switch statement. If the operator is one of the valid operators, the corresponding operation is performed. If not, an invalid operator message is written to the console and the program returns to waiting for valid input.

Note how the stack is used to store incoming operands and how, after each operation, the result is pushed back onto the top of the stack.

Entering the equals operator '=' results in the value located at the top of the stack being popped from the stack and written to the console. In this way you can clear the calculator of the last result before proceeding with a new calculation.

Figure 7-7 shows the `LineCalc` program in action.

```

C:\Collection Book Projects\Chapter_7\LineCalc>linecalc
Enter operand, operator, or "quit" to exit --> 45
Enter operand, operator, or "quit" to exit --> 45
Enter operand, operator, or "quit" to exit --> 45
Enter operand, operator, or "quit" to exit --> +
Add result: 90
Enter operand, operator, or "quit" to exit --> +
Add result: 135
Enter operand, operator, or "quit" to exit --> +
Note enough operands on stack?
Enter operand, operator, or "quit" to exit --> 100
Enter operand, operator, or "quit" to exit --> 50
Enter operand, operator, or "quit" to exit --> -
Sub result: 50
Enter operand, operator, or "quit" to exit --> 50
Enter operand, operator, or "quit" to exit --> +
Add result: 100
Enter operand, operator, or "quit" to exit --> 4
Enter operand, operator, or "quit" to exit --> /
Div result: 25
Enter operand, operator, or "quit" to exit --> =
Total: 25
Enter operand, operator, or "quit" to exit --> =
Total: 135
Enter operand, operator, or "quit" to exit --> =
Stack empty?
Enter operand, operator, or "quit" to exit --> quit
C:\Collection Book Projects\Chapter_7\LineCalc>_

```

Figure 7-7: Results of Performing Several Operations with LineCalc

## Quick Review

The `System.Collections.Generic.Stack<T>` class is the direct replacement for the non-generic `Stack` class. The benefit to using the `Stack<T>` class is that you gain a wider array of operations via extension methods defined by the `System.Linq.Enumerable` class. Also, value types do not require boxing and unboxing operations when being pushed onto and popped off the stack.

---

## SUMMARY

---

A stack is a specialized list whose elements are stored in last-in/first-out (LIFO) order. Stacks support two primary operations: push and pop. The push operation stores an item on top of the stack. As more items are pushed onto the stack, the older items move deeper into the stack while younger items are at the top of the stack. The most recent item pushed onto the stack will always be at the top of the stack. The pop operation removes the most recently pushed item from the top of the stack.

The `HomeGrownStack` class demonstrates the use of an array to contain stack items. It implements the `Push()`, `Pop()`, and `Peek()` methods as well as the `IsEmpty` property. The `top` field is incremented each time an item is pushed onto the stack and decremented each time an item is popped off the stack.

The `System.Collections.Stack` is a non-generic collection which stores any type of object. Value type objects will undergo a boxing operation when they are pushed onto the stack. This results in their references being pushed onto the stack and the object it points to is created in the heap. When the value type object is popped off the stack, it will undergo an unboxing operation.

The `System.Collections.Generic.Stack<T>` class is the direct replacement for the non-generic `Stack` class. The benefit to using the `Stack<T>` class is that you gain a wider array of operations via extension methods defined by the `System.Linq.Enumerable` class. Also, value types do not require boxing and unboxing operations when being pushed onto and popped off the stack.

---

**REFERENCES**

---

Sten Henriksson. A Brief History of the Stack. [ <http://www.sigcis.org/files/A%20brief%20history.pdf> ]

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**

---