

Compositive structures (part II)

Subjects:

- Arrays
- Iteration constructs
- Decision constructs and the relational/equality operators

Arrays

The built-in class `Array` is a structure that can contain a series of elements of the same type (all `int` values and all `double` values, for example).

Consider the problem of averaging a set of six floating-point numbers. Each of the six numbers requires its own `double` storage:

```
double d0 = 5;  
double d1 = 2;  
double d2 = 7;  
double d3 = 3.5;  
double d4 = 6.5;  
double d5 = 8;
```

Computing the average of those variables might look like this:

```
double sum = d0 + d1 + d2 + d3 + d4 + d5;  
double average = sum / 6;
```

Listing each element by name is tedious. Maybe it's not so tedious when you have only 6 numbers to average, but imagine averaging 600 (or even 6 million) floating-point values.

The fixed-value array

Fortunately, you don't need to name each element separately. **C# provides the array structure that can store a sequence of values.** Using an array, you can put all your `doubles` into one variable, like this:

```
double[] doublesArray = {5, 2, 7, 3.5, 6.5, 8, 1, 9, 1, 3};
```

You can also declare an **empty array** without initializing it:

```
double[] doublesArray = new double[6];
```

This line allocates space for six `doubles` but doesn't initialize them. The `Array` class, on which all C# arrays are based, provides a special syntax that makes it more convenient to use. **The paired brackets `[]` refer to the way you access individual elements in the array:**

```
doublesArray[0] //Corresponds to d0 (that is, 5)
```

```
doublesArray[1] //Corresponds to d1 (that is, 2)
```

```
...
```

The 0th element of the array corresponds to `d0`, the 1th element to `d1`, the 2th element to `d2`, and so on. Programmers commonly refer to the 0th element as “`doublesArray` sub-0,” to the first element as “`doublesArray` sub-1,” and so on.

The array's element numbers — 0, 1, 2, ... — are known as the **index**.

The variable-length array

In C#, the array index starts at 0 and not at 1. *The first element is the zeroth element.* If you insist on using normal speech, just be aware that the first element is always at index 0 and the second element is at index 1.

The format for declaring a **variable-size array** differs slightly from that of a fixed-size, fixed-value array:

```
double[] doublesArrayVariable = new double[N]; //Variable, versus  
double[] doublesArrayFixed = new double[10]; // Fixed
```

Here, N represents the number of elements to allocate.

An **array** is a set of data items, accessed using a numerical index. More specifically, an array is a set of contiguous data points of the same type (an array of `ints`, an array of `strings`, an array of `SportsCars`, and so on).

Arrays

```
class Program
{

    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with Arrays *****");
        SimpleArrays();
    }

    static void SimpleArrays()
    {
        Console.WriteLine("=> Simple Array Creation.");
        // Assign an array ints containing 3 elements {0 - 2}
        int[] myInts = new int[3];
        // Initialize a 100 item string array, numbered {0 - 99}
        string[] booksOnDotNet = new string[100];
        Console.WriteLine();
    }
}
```

When declaring a C# array using this syntax, **the number used in the array declaration represents the total number of items, not the upper bound**. Also note that the **lower bound of an array always begins at 0**. Thus, when you write `int [] myInts[3]`, you end up with a array holding three elements (`{0, 1, 2}`).

C# Array Initialization Syntax

In addition to filling an array element by element, you are also able to fill the items of an array using C# array initialization syntax. To do so, **specify each array item within the scope of curly brackets ({}).** This syntax can be helpful when you are creating an array of a known size and wish to quickly specify the initial values.

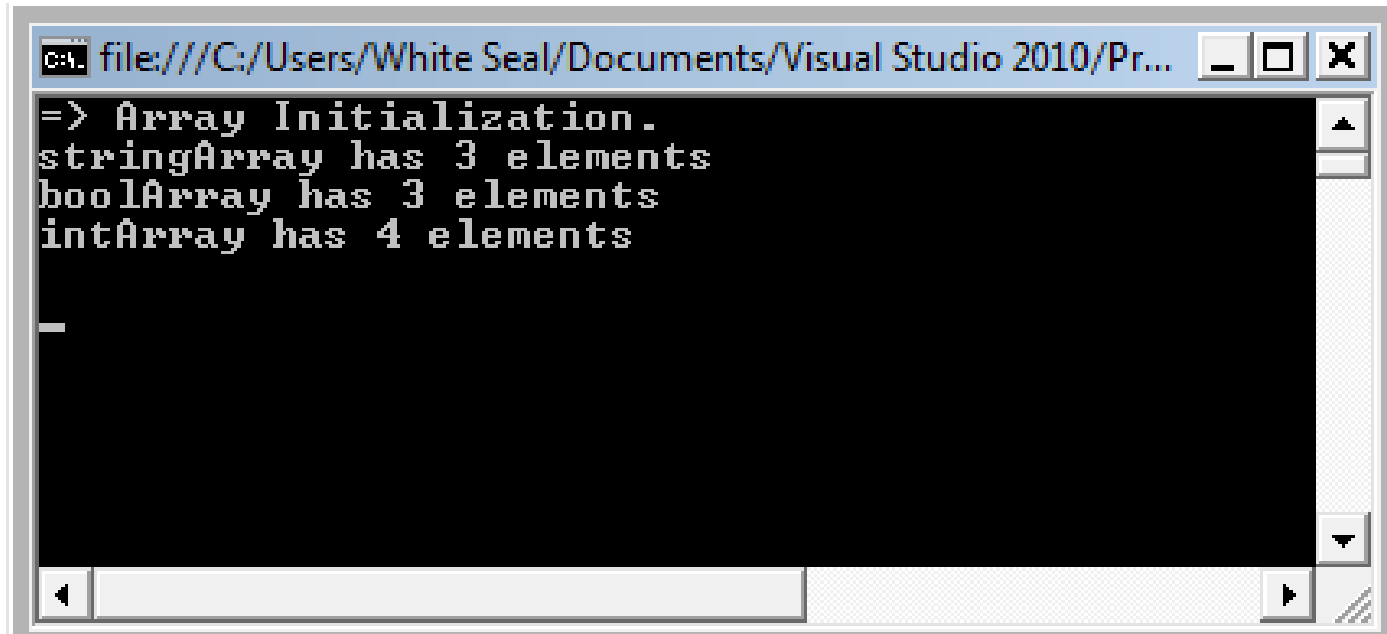
```
static void ArrayInitialization()
{
    Console.WriteLine("=> Array Initialization.");

    // Array initialization syntax using the new keyword.
    string[] stringArray = new string[] { "one", "two", "three" };
    Console.WriteLine("stringArray has {0} elements", stringArray.Length);

    // Array initialization syntax without using the new keyword.
    bool[] boolArray = { false, false, true };
    Console.WriteLine("boolArray has {0} elements", boolArray.Length);

    // Array initialization with new keyword and size.
    int[] intArray = new int[4] { 20, 22, 23, 0 };
    Console.WriteLine("intArray has {0} elements", intArray.Length);
    Console.WriteLine();
}
```

C# Array Initialization Syntax



```
file:///C:/Users/White Seal/Documents/Visual Studio 2010/Pr...  
=> Array Initialization.  
stringArray has 3 elements  
boolArray has 3 elements  
intArray has 4 elements
```

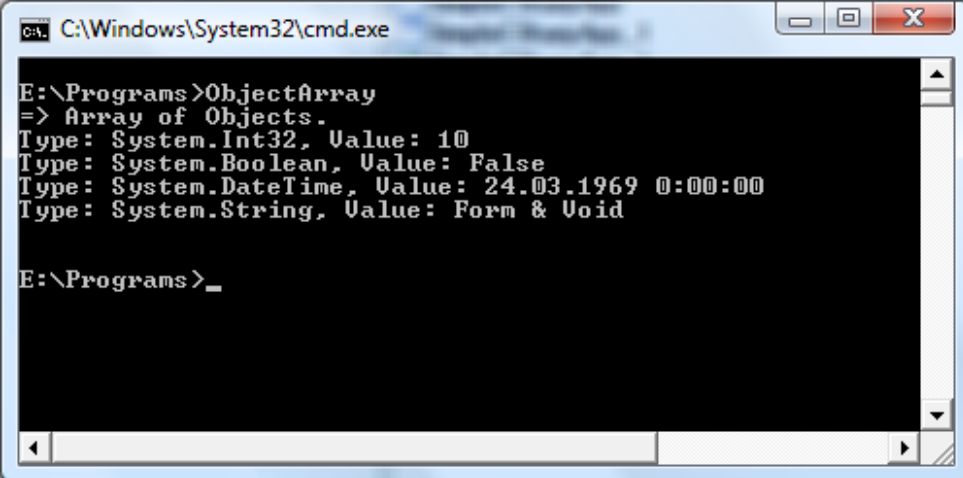
Notice that **when you make use of this “curly bracket” syntax, you do not need to specify the size of the array** (seen when constructing the `stringArray` type), given that this will be inferred by the number of items within the scope of the curly brackets. Also notice that use of the `new` keyword is optional.

Array of Objects

System.Object is the ultimate base class to each and every type (including fundamental data types) in the .NET type system. Given this fact, if you were to define an array of objects, the subitems could be anything at all.

```
static void ArrayOfObjects()
{
    Console.WriteLine("=> Array of Objects.");
    // An array of objects can be anything at all.
    object[] myObjects = new object[4];
    myObjects[0] = 10;
    myObjects[1] = false;
    myObjects[2] = new DateTime(1969, 3, 24);
    myObjects[3] = "Form & Void";
    foreach (object obj in myObjects)
    {
        // Print the type and value for each item in array.
        Console.WriteLine("Type: {0}, Value: {1}",
                           obj.GetType(), obj);
    }
    Console.WriteLine();
}
```

Here, as we are iterating over the contents of `myObjects`, we print out the underlying type of each item using the `GetType()` method of `System.Object` as well as the value of the current item.



```
C:\Windows\System32\cmd.exe

E:\Programs>ObjectArray
=> Array of Objects.
Type: System.Int32, Value: 10
Type: System.Boolean, Value: False
Type: System.DateTime, Value: 24.03.1969 0:00:00
Type: System.String, Value: Form & Void

E:\Programs>_
```

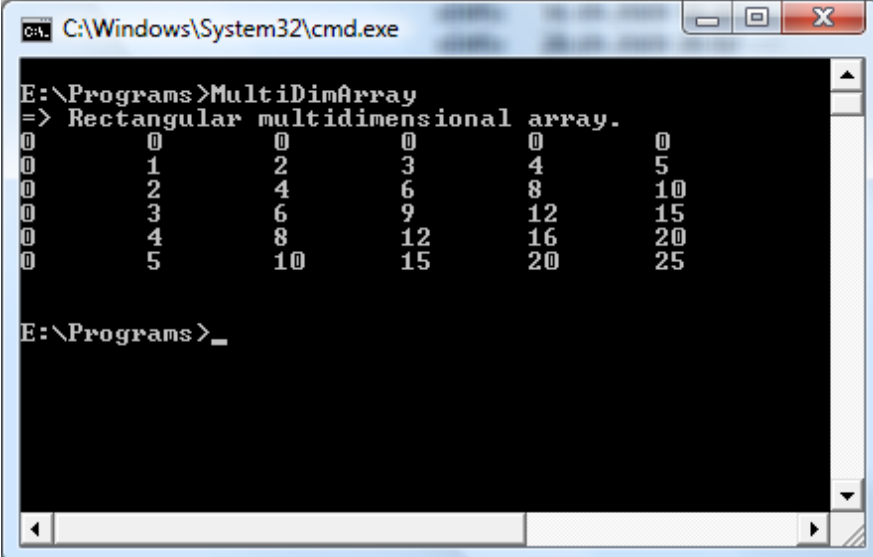

Multidimensional Arrays

In addition to the single-dimension arrays you have seen thus far, **C# also supports two varieties of multidimensional arrays**. The first of these is termed a **rectangular array**, which is simply an array of multiple dimensions, where each row is of the same length.

```
static void RectMultidimensionalArray()
{
    Console.WriteLine("=> Rectangular multidimensional array.");

    // A rectangular MD array.
    int[,] myMatrix;
    myMatrix = new int[6,6];

    // Populate (6 * 6) array.
    for(int i = 0; i < 6; i++)
        for(int j = 0; j < 6; j++)
            myMatrix[i, j] = i * j;
    // Print (6 * 6) array.
    for(int i = 0; i < 6; i++)
    {
        for(int j = 0; j < 6; j++)
            Console.Write(myMatrix[i, j] + "\t");
        Console.WriteLine();
    }
    Console.WriteLine();
}
```



```
C:\Windows\System32\cmd.exe
E:\Programs>MultiDimArray
=> Rectangular multidimensional array.
0 0 0 0 0 0
0 1 2 3 4 5
0 2 4 6 8 10
0 3 6 9 12 15
0 4 8 12 16 20
0 5 10 15 20 25

E:\Programs>
```

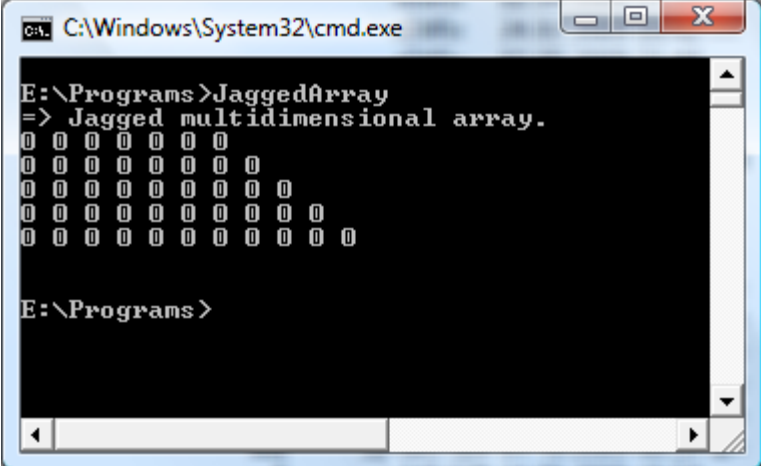
Jagged arrays

Jagged arrays contain some number of inner arrays, each of which may have a unique upper limit

```
static void JaggedMultidimensionalArray()
{
    Console.WriteLine("=> Jagged multidimensional array.");

    // A jagged MD array (i.e., an array of arrays).
    // Here we have an array of 5 different arrays.
    int[][] myJagArray = new int[5][];

    // Create the jagged array.
    for (int i = 0; i < myJagArray.Length; i++)
        myJagArray[i] = new int[i + 7];
    // Print each row (remember, each element is defaulted to
    zero!)
    for(int i = 0; i < 5; i++)
    {
        for(int j = 0; j < myJagArray[i].Length; j++)
            Console.Write(myJagArray[i][j] + " ");
        Console.WriteLine();
    }
    Console.WriteLine();
}
```



The screenshot shows a Windows command prompt window titled "C:\Windows\System32\cmd.exe". The prompt is at "E:\Programs>". The user has entered "JaggedArray", and the output is displayed on the screen. The output matches the code's execution: a message followed by a jagged array of zeros.

```
C:\Windows\System32\cmd.exe
E:\Programs>JaggedArray
=> Jagged multidimensional array.
0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
E:\Programs>
```

Arrays as parameters

Once you have created an array, you are free to pass it as a parameter and receive it as a member return value.

```
static void PrintArray(int[] myInts)
{
    for(int i = 0; i < myInts.Length; i++)
        Console.WriteLine("Item {0} is {1}", i, myInts[i]);
}

static string[] GetStringArray()
{
    string[] theStrings = { "Hello", "from", "GetStringArray" };
    return theStrings;
}
```

These methods may be invoked as you would expect:

```
static void PassAndReceiveArrays()
{
    Console.WriteLine("=>Arrays as params and return values.");
    // Pass array as parameter.
    int[] ages = {20, 22, 23, 0} ;
    PrintArray(ages);
    // Get array as return value.
    string[] strs = GetStringArray();
    foreach(string s in strs)
        Console.WriteLine(s);
    Console.WriteLine();
}
```

The System.Array Base Class

Every array you create gathers much of its functionality from the `System.Array` class. Using these common members, we are able to operate on an array using a consistent object model.

Member of Array class	Meaning in life
<code>Clear()</code>	This static method sets a range of elements in the array to empty values (0 for value items, static for object references).
<code>CopyTo()</code>	This method is used to copy elements from the source array into the destination array.
<code>Length</code>	This property returns the number of items within the array.
<code>Rank</code>	This property returns the number of dimensions of the current array.
<code>Reverse()</code>	This static method reverses the contents of a one-dimensional array.
<code>Sort()</code>	This static method sorts a one-dimensional array of intrinsic types. If the elements in the array implement the <code>IComparer</code> interface, you can also sort your custom types.

The System.Array Base Class

Let's see some of these members in action.

```
static void SystemArrayFunctionality()
{
    Console.WriteLine("=> Working with System.Array.");

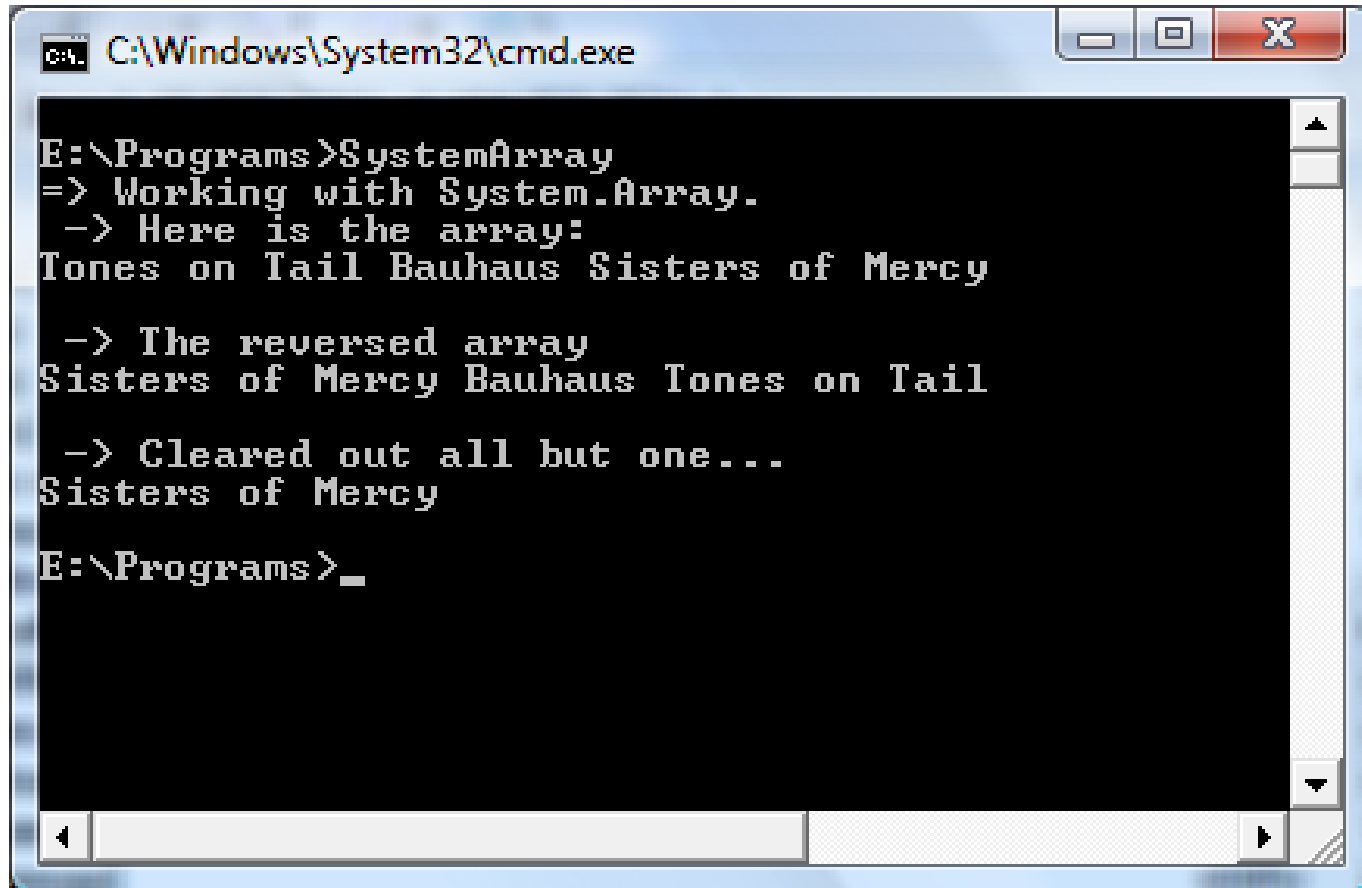
    // Initialize items at startup.
    string[] gothicBands = {"Tones on Tail", "Bauhaus", "Sisters of Mercy"};

    // Print out names in declared order.
    Console.WriteLine(" -> Here is the array:");
    for (int i=0; i<=gothicBands.GetUpperBound(0); i++)
        Console.Write(gothicBands[i] + " ");
    Console.WriteLine("\n");

    // Reverse them...
    Array.Reverse(gothicBands);
    Console.WriteLine(" -> The reversed array");
    // ... and print them.
    for (int i=0; i<=gothicBands.GetUpperBound(0); i++)
        Console.Write(gothicBands[i] + " ");
    Console.WriteLine("\n");

    // Clear out all but the final member.
    Console.WriteLine(" -> Cleared out all but one...");
    Array.Clear(gothicBands, 1, 2);
    for (int i=0; i<=gothicBands.GetUpperBound(0); i++)
        Console.Write(gothicBands[i] + " ");
    Console.WriteLine();
}
```

The System.Array Base Class



```
C:\Windows\System32\cmd.exe

E:\Programs>SystemArray
=> Working with System.Array.
-> Here is the array:
Tones on Tail Bauhaus Sisters of Mercy

-> The reversed array
Sisters of Mercy Bauhaus Tones on Tail

-> Cleared out all but one...
Sisters of Mercy

E:\Programs>_
```

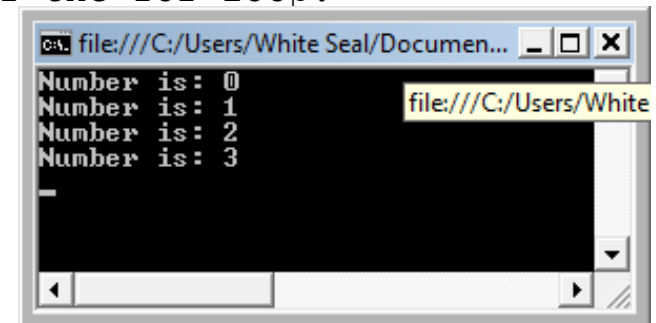
Iteration constructs

All programming languages provide ways to repeat blocks of code until a terminating condition has been met. Regardless of which language you have used in the past, the C# iteration statements should not raise too many eyebrows and should require little explanation. **C# provides the following four iteration constructs:**

- **for loop**
- **foreach/in loop**
- **while loop**
- **do/while loop**

The `for` Loop. When you need to iterate over a block of code a fixed number of times, the `for` statement provides a good deal of flexibility. In essence, you are able to specify how many times a block of code repeats itself, as well as the terminating condition.

```
// A basic for loop.
static void ForAndForEachLoop()
{
    // Note! "i" is only visible within the scope of the for loop.
    for(int i = 0; i < 4; i++)
    {
        Console.WriteLine("Number is: {0} ", i);
    }
    // "i" is not visible here.
}
```



Consult the .NET Framework SDK documentation if you require details on the C# `for` keyword.

Iteration constructs

The `foreach` Loop. The C# `foreach` keyword allows you to iterate over all items within an array, without the need to test for the array's upper limit. Here are two examples using `foreach`, one to traverse an array of strings and the other to traverse an array of integers:

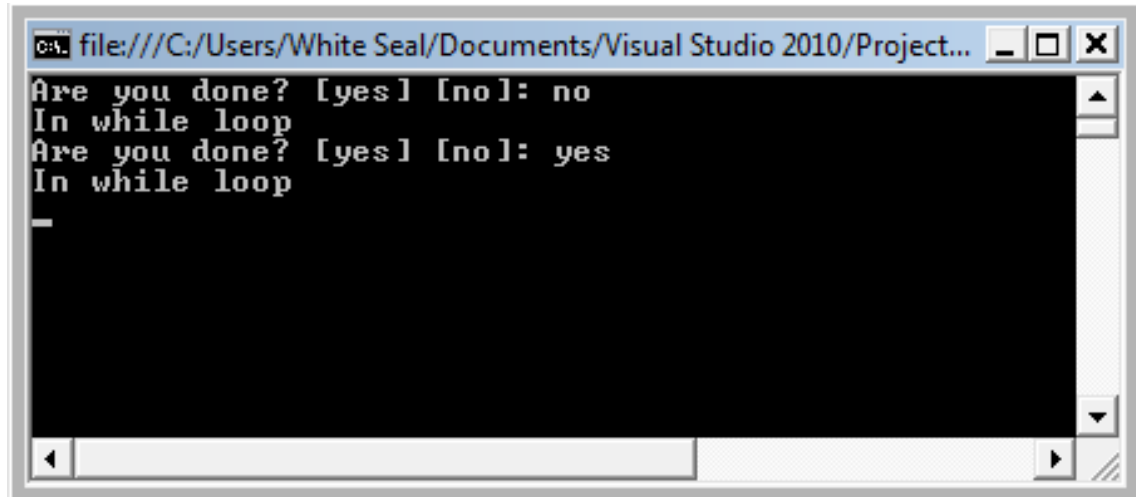
```
// Iterate array items using foreach.
static void ForAndForEachLoop()
{
    ...
    string[] carTypes = {"Ford", "BMW", "Toyota", "Honda"};
    foreach (string c in carTypes)
        Console.WriteLine(c);
    int[] myInts = {10, 20, 30, 40};
    foreach (int i in myInts)
        Console.WriteLine(i);
}
```

```
file:///C:/Users/White Seal/Documents/Visual Studio 201...
Ford
BMW
Toyota
Honda
10
20
30
40
-
```


Iteration constructs

The **while** and **do/while** Looping Constructs. The **while** looping construct is useful should you wish to execute a block of statements until some terminating condition has been reached. Within the scope of a `while` loop, **you will**, of course, **need to ensure this terminating event is indeed established**; otherwise, you will be stuck in an endless loop.

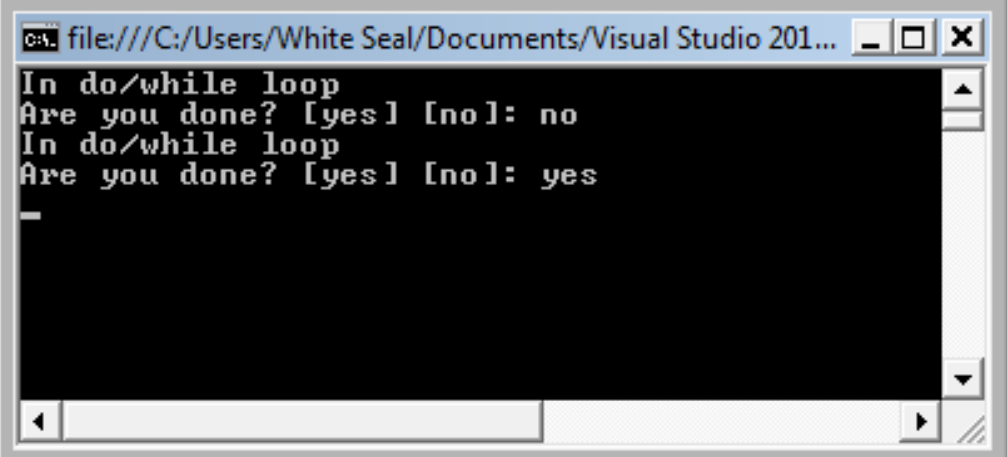
```
static void ExecuteWhileLoop()
{
    string userIsDone = "";
    // Test on a lower-class copy of the string.
    while (userIsDone.ToLower() != "yes")
    {
        Console.Write("Are you done? [yes] [no]: ");
        userIsDone = Console.ReadLine();
        Console.WriteLine("In while loop");
    }
}
```

A screenshot of a console window titled "file:///C:/Users/White Seal/Documents/Visual Studio 2010/Project...". The window shows the output of the C# program. It displays two iterations of the while loop. In the first iteration, the prompt "Are you done? [yes] [no]: " is followed by the user input "no", and then "In while loop" is printed. In the second iteration, the prompt is followed by the user input "yes", and then "In while loop" is printed. The console window has a black background with white text and standard Windows window controls at the top and bottom.

Iteration constructs

Closely related to the `while` loop is the **do/while** statement. Like a simple `while` loop, **do/while is used when you need to perform some action an undetermined number of times**. The difference is that **do/while loops are guaranteed to execute the corresponding block of code at least once** (in contrast, it is possible that a simple `while` loop may never execute if the terminating condition is false from the onset).

```
static void ExecuteDoWhileLoop()
{
    string userIsDone = "";
    do
    {
        Console.WriteLine("In do/while loop");
        Console.Write("Are you done? [yes] [no]: ");
        userIsDone = Console.ReadLine();
    } while(userIsDone.ToLower() != "yes"); // Note the semicolon!
}
```

A screenshot of a console window titled "file:///C:/Users/White Seal/Documents/Visual Studio 201...". The window shows the output of the C# code. It displays two iterations of the loop. In the first iteration, the prompt "Are you done? [yes] [no]: " is followed by the input "no". In the second iteration, the same prompt is followed by the input "yes". The console text is as follows:
In do/while loop
Are you done? [yes] [no]: no
In do/while loop
Are you done? [yes] [no]: yes
The window has a standard Windows title bar and a scrollbar on the right side.

Decision Constructs and the Relational/Equality Operators

C# defines two simple constructs to alter the flow of your program, based on various contingencies:

- the **if/else** statement;
- the **switch** statement.

The `if/else` Statement. First is the `if/else` statement. These statements typically involve the use of the C# operators shown in table in order **to obtain a literal Boolean value**.

Equality / Relational operator	Example of usage	Meaning in life
<code>==</code>	<code>if (age == 30)</code>	Returns <code>true</code> only if each expression is the same
<code>!=</code>	<code>if ("Yes" != myStr)</code>	Returns <code>true</code> only if each expression is different
<code><</code>	<code>if (bonus < 2000)</code>	Returns <code>true</code> if expression A is less than, greater than, less than or equal to, or greater than or equal to expression B
<code>></code>	<code>if (bonus > 2000)</code>	
<code><=</code>	<code>if (bonus <= 2000)</code>	
<code>>=</code>	<code>if (bonus >= 2000)</code>	

Decision Constructs and the Relational/Equality Operators

Let's say you want to see whether the `string` you are working with is longer than zero characters. You may be tempted to write

```
static void ExecuteIfElse()  
{  
    // This is illegal, given that Length returns an int, not a bool.  
    string stringData = "My textual data";  
    if(stringData.Length)  
    { Console.WriteLine("string is greater than 0 characters"); }  
}
```

If you wish to make use of the `String.Length` property to determine truth or falsity, you need to modify your conditional expression to resolve to a `Boolean`.

```
// Legal, as this resolves to either true or false.  
if(stringData.Length > 0)  
{ Console.WriteLine("string is greater than 0 characters"); }
```

Decision Constructs and the Relational/Equality Operators

An **if statement** may be composed of **complex expressions** as well and can contain **else statements to perform more complex testing**. To build complex expressions, C# offers an expected set of conditional operators.

Operator	Example	Meaning in life
&&	if((age == 30) && (name == "Fred"))	Conditional AND operator
	if((age == 30) (name == "Fred"))	Conditional OR operator
!	if(!myBool)	Conditional NOT operator

Decision Constructs and the Relational/Equality Operators

The **switch** Statement. The switch statement allows you to handle program flow based on a predefined set of choices. For example, the following `Main()` logic prints a specific string message based on one of two possible selections (the default case handles an invalid selection):

```
// Switch on a numerical value.
```

```
static void ExecuteSwitch()
```

```
{
```

```
    Console.WriteLine("1 [C#], 2 [VB]");
```

```
    Console.Write("Please pick your language preference: ");
```

```
    string langChoice = Console.ReadLine();
```

```
    int n = int.Parse(langChoice);
```

```
    switch (n)
```

```
    {
```

```
        case 1:
```

```
            Console.WriteLine("Good choice, C# is a fine language.");
```

```
            break;
```

```
        case 2:
```

```
            Console.WriteLine("VB .NET: OOP, multithreading, and more!");
```

```
            break;
```

```
        default:
```

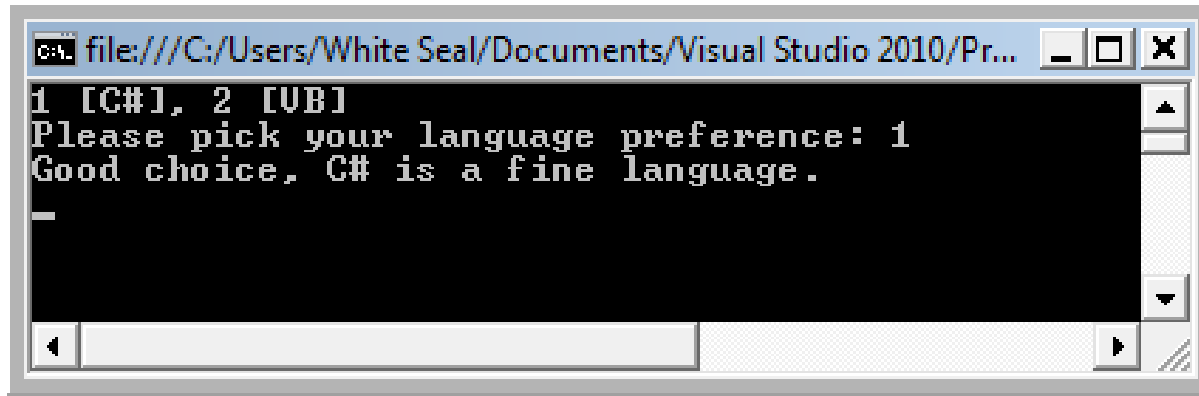
```
            Console.WriteLine("Well...good luck with that!");
```

```
            break;
```

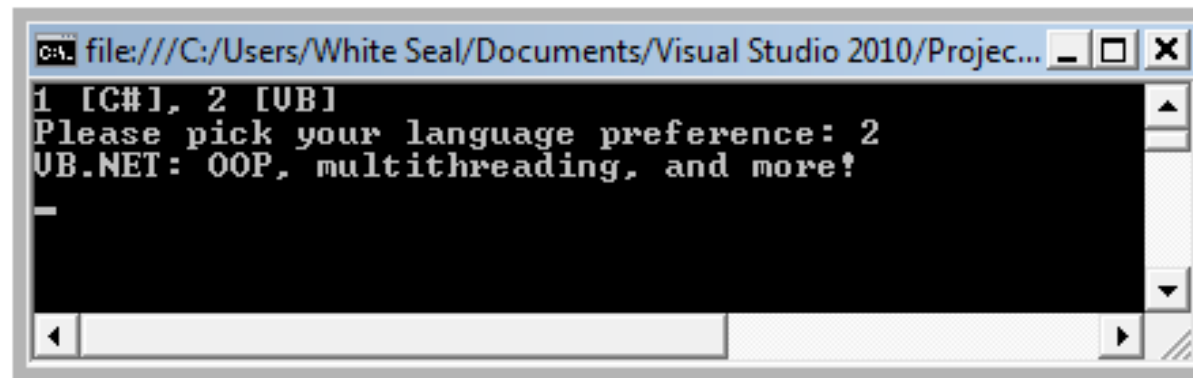
```
    }
```

```
}
```

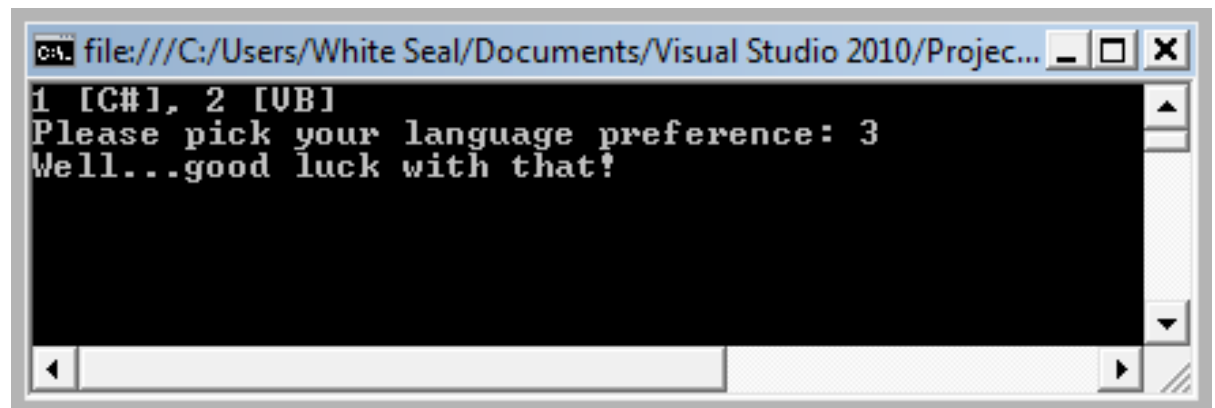
Decision Constructs and the Relational/Equality Operators



```
file:///C:/Users/White Seal/Documents/Visual Studio 2010/Pr...
1 [C#], 2 [VB]
Please pick your language preference: 1
Good choice, C# is a fine language.
```



```
file:///C:/Users/White Seal/Documents/Visual Studio 2010/Projec...
1 [C#], 2 [VB]
Please pick your language preference: 2
UB.NET: OOP, multithreading, and more!
```



```
file:///C:/Users/White Seal/Documents/Visual Studio 2010/Projec...
1 [C#], 2 [VB]
Please pick your language preference: 3
Well...good luck with that!
```

Thanks for attention

Happy
New
Year

A 3D rendering of the words "Happy New Year" in a bold, red, sans-serif font. The text is arranged in three lines: "Happy" on top, "New" in the middle, and "Year" at the bottom. The letters have a slight shadow and are reflected on a glossy white surface below them. The background is white and filled with numerous small, red, diamond-shaped confetti pieces scattered throughout.