1.0. Introduction

For several generations, the most widely sold toys have been those that allow you to take small simple objects and combine them to create a replica of some real-world object. Initially starting out as a large pile of plastic blocks with varying shapes and colors, they can transform into a multistage ignition rocket ship or an elegant mansion without a roof.

Toys aren’t the only place we see this sort of design. Almost everywhere you look, this piecewise refinement is evident in many facets of everyday life. Cars aren’t built using a myriad of non-connecting objects but rather are assembled from smaller pieces to form the engine, for instance, which is then used as a component in the final automobile. Using another example, a company contains employees who form a team. These teams combine to form a section or business unit, and these units group together to form the corporation. In fact, this type of organization doesn’t just make sense in the physical realm but is prevalent in many natural things as well.

Based on this concept, it comes as no surprise that software is logically divided into smaller pieces that contribute to a solution when assembled. At the lower levels of the programming language construct hierarchy are the operator, the expression, and the control structure. An expression is any group of operators and operands that are combined to perform some type of computation, such as setting a variable, calling a function, or performing a system-related task.

In this chapter, we look at all the various C# language elements that are available to construct an application. From the basic layout of a small console application to using overloaded operators to change the semantic meaning of objects, this chapter points out the various options you have at your disposal to efficiently design your next Visual C# .NET application.
1.1. Understanding Visual C# .NET Program Layout

You want to create a simple Visual C# .NET console application and inspect the generated code.

Technique

In the Visual Studio .NET IDE, click on File, New, Project (Ctrl+Shift+N) and select Visual C# Projects from the list of project types. In the list of templates, select the Console Application template and type in a name for the project.

Comments

For most of this chapter and in several projects throughout this book, you’ll find yourself working with console-based applications. Their minimalist nature allows you to concentrate on the topic being discussed without having to traverse through extraneous, potentially distracting code.

Even though it is just a simple console-based application that currently provides no functionality, it does serve as a good illustration for some of the various pieces that make a C# application. Once you create your project, the Visual Studio .NET IDE opens the application’s main program file, which should look similar to Listing 1.1.

Listing 1.1  A Simple Visual C# .NET Console Application

```csharp
using System;
namespace _1_ProgramLayout
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class DateTimePrinter
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            Console.WriteLine( "Today is {0}", DateTime.Now );
        }
    }
}
```
The first couple of lines relate to an organizational construct known as the namespace. Namespaces convey a sense of relationships between like objects. For instance, if you have some objects with such names as Frame, Engine, Wheel, Light, and Seat, you immediately see the relationship these objects possess in that they all belong on an automobile. In this example, the namespace could be Automobile.

On the first line, the program is letting the compiler know that it wants to use some of the objects within the System namespace. Therefore, anytime you want to use one of these objects, you do not need to preface the object name with the namespace name. For instance, the WriteLine statement prints out a small message in the console. The Console object is a class within the System namespace. Without the using directive, you have to qualify each type you want to use with the namespace it is declared in. In this instance, the line in Listing 1.1 would be

```csharp
System.Console.WriteLine( "Today is {0}", DateTime.Now );
```

In the listing, you can see a namespace declaration, which is simply the name you gave the project when you initially created it. Although the wizard placed an initial type in a namespace for you to use, the namespace declaration itself is purely optional. However, we recommend that you do get into the habit of using namespaces to enhance code organization and readability.

Within the automatically generated namespace is a class called Class1, which was renamed to DateTimePrinter to more accurately describe the type’s functionality (albeit somewhat limited). A class is best described as a data type that can contain various associated methods to interact with it. Whenever an instance of that class is created or instantiated, it becomes a usable object to which you can get or set various properties, receive notification of certain events, and tell it to perform some action using one of its member functions. Chapter 2, “Objects and Components,” delves deeper into the many ways classes are used within C# and the .NET Framework.

The last important component of our simple console application is the application’s entry point, denoted by the static member function Main. Each application must contain an entry point so that the Common Language Runtime (CLR) can effectively start the application by handing control over to you. Once the Main function exits, the application ends. Just before the Main method declaration is the STAThread attribute. This attribute denotes the threading model your application uses. However, it is only applicable if you plan to use COM Interop, discussed in Chapter 24, “COM Interop,” and is ignored if you do not.
1.2. Parsing Command-Line Arguments

You want your application to support command-line argument parsing.

**Technique**

Use a string array as a parameter in your application’s entry point, the static `Main` function.

```csharp
using System;

class CommandLineArgs
{
    static void Main(string[] args)
    {
        foreach (string arg in args)
            Console.WriteLine(arg + "\n");
    }
}
```

**Comments**

The `Main` function in an application has the option of accepting a string array as a parameter. This array contains the command-line arguments passed to it by the system where each element in the array corresponds to a string on the command line delineated by quotation marks or whitespace.

1.3. Creating Multiple Application Entry Points

You want your application to contain several entry points.

**Technique**

For each entry point you want to add, create a separate class containing a static `Main` member function. To change which entry point is run by the CLR, go to the project properties (Alt+P,P) and select Common Properties, General. Select the function you want to use as the application entry point in the Startup Object field. You must then recompile your project any time you change the application entry point because you can only change the entry point at compile time, not at runtime.
Comments

There are several reasons why creating several application entry points is advantageous. One key advantage is that it allows you to control the flow of your application to test different avenues of functionality. For instance, you can create an entry point that serves to display a tremendous amount of debugging information at runtime. When you are confident that the unit tests you created are working correctly, you can create a final application entry point that removes most of the debugging information. However, in the past, you might have had to comment out or remove the critical debugging information. By creating a separate entry point, you gain the advantage of preserving your unit tests that you can run simply by changing a property within your project. With a little clever command-line compiling, you can even create an automated test harness that compiles your application, specifying one of your unit-test application entry points; verify that the results are satisfied; and repeat that process for each entry point.

To compile a C# program from the command line and specify which startup object to use as the application entry point, use the /main command-line argument for the compiler. Listing 1.2 contains a project source file that contains three different application entry points. To compile this file on the command line and specify an application entry point, you invoke the C# compiler using the following:

csc.exe Class1.cs /main:_3_MultEntryPoints.EntryPointTest.Main3Entry

Listing 1.2 A C# Application Containing Multiple Application Entry Points

```csharp
using System;

namespace _3_MultEntryPoints
{
    class EntryPointTest
    {
        class Main1Entry
        {
            [STAThread]
            static void Main(string[] args)
            {
                Console.WriteLine( "Running in Main" );
            }
        }
        class Main2Entry
        {
            [STAThread]
            static void Main(string[] args)
            {
                Console.WriteLine( "Running in Main2" );
            }
        }
    }
}
```
1.4. Referencing Assemblies and Namespaces

You want to use a class, but the compiler says that the class cannot be found.

**Technique**
Add a reference to the assembly that defines the class by clicking on the Project, Add Reference from the main menu. Select the .NET tab, locate the assembly that contains the type you want to use, and double-click it.

**Comments**
The .NET Framework uses information in an assembly to resolve type information when you want to use a class. An assembly is simply the file that is created when you compile a project such as an .exe or dynamic link library (.dll) file. When you perform the steps of referencing an assembly, the compiler knows to look in each referenced assembly during compilation to resolve any types that are encountered within the source file.

1.5. Creating Valid Identifier Names

You want to ensure that the identifier follows the rules of valid identifier naming.

**Technique**
Your identifier cannot be the same as that of a C# keyword. You are free to use any combination of letters, numbers, and underscore characters as long as the identifier does not start with a number. Furthermore, you can use the @ symbol as the first character in your identifier. Listing 1.3 shows several valid and invalid identifier names.
Listing 1.3  **Demonstrating Valid and Invalid Identifier Names**

```csharp
namespace _5_Identifiers
{
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            int i;          // valid
            int @int;       // valid but confusing
            int INT;        // valid since C# is case sensitive
            int 4_Score;    // Not valid. Cannot begin with a digit
            int in@t;       // not valid. @ symbol not at beginning
            int int;        // not valid. Cannot use keywords
        }
    }
}
```

**Comments**

Identifiers name certain programming elements within a program. They include such items as namespace names, classes, and variables. It is good programming practice to create identifiers that convey important information about the element it is defined as. In the days before .NET, Hungarian notation was used as the de facto standard for identifier naming, at least for those using the C++ language within a Windows program. Although some people argue for or against its suitability as a standard, the fact remains that it was a standard which could alleviate some of the issues that go along with reading someone else's source code. With the release of .NET, Microsoft dropped the Hungarian-notation guideline due to the cross-language support within the .NET Framework. Some of these identifier guidelines include using a combination of Pascal and Camel case for character capitalization within an identifier as well as using semantics in the identifier name. If you want to learn more about the guidelines Microsoft uses for the .NET Framework, search MSDN using the phrase “naming guidelines.”

### 1.6. Working with Numeric Types

**You need to store data in a variable as an integral type.**

**Technique**

Choosing the correct data type can almost be considered an art. Keep in mind the range of possible values for the data you need to store in memory and whether you need a signed or unsigned data type.
Comments

There are eight types designed to work with numerical data. Each of the types is designated based on its size and whether it is signed. The smallest integral types available to C# are the 8-bit `byte` with a range of 0 to 255 and the 8-bit `sbyte`, a signed data type with a range of –128 to 127. At the far end of the spectrum are the 64-bit `ulong` and the signed version, the `long`. Table 1.1 shows the possible numerical data types as well as their sizes in bits and range.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Size (Bits)</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>sbyte</td>
<td>8</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>byte</td>
<td>8</td>
<td>0 to 255</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>-32,768 to 32,767</td>
</tr>
<tr>
<td>ushort</td>
<td>16</td>
<td>0 to 65,535</td>
</tr>
<tr>
<td>char</td>
<td>16</td>
<td>0 to 65,535</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>-2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>uint</td>
<td>32</td>
<td>0 to 4,294,967,295</td>
</tr>
<tr>
<td>long</td>
<td>64</td>
<td>-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807</td>
</tr>
<tr>
<td>ulong</td>
<td>64</td>
<td>0 to 18,446,744,073,709,551,615</td>
</tr>
</tbody>
</table>

When working with integral data types, you have to weigh two options. First of all, even though we live in an age where memory is in abundance, you should still attempt to keep memory usage to a minimum. Sometimes, your application might need to run in a low-memory environment, and by choosing a design that optimizes memory usage, you can benefit from this strategy. Secondly, you must ensure that the data type you choose is large enough to hold any value you assign to it. If you inadvertently assign a number outside of a data type’s range, your application will encounter a situation that it is not prepared to deal with. Based on your project properties and whether you are using exception handling, which is explained later in this book, the CLR will do one of two things. First, your application will throw an `OverflowException`. It is then your responsibility to handle this exception or face the consequences of an application crash.

By default, your project is created with overflow checking turned off. This default means that the CLR does not check whether a value being assigned to a data type causes an overflow. Rather, it takes a safer approach, which is either to flip the bits of the data type, thereby causing it to change sign, or to simply roll the value back to 0 in the case of unsigned data. As you might guess, this process can have undesirable side effects in your application and can lead to subtle defects that might be hard to find.

To turn on overflow checking in your project, open the property pages of your project by choosing Project, Properties (Alt+P). Select Configuration Properties, Build. Set the property labeled Check for Arithmetic Overflow to `True` so that data types are...
checked for overflow, as shown in Figure 1.1. If you believe that once you thoroughly test your code with all possible inputs your data types will not overflow, you can consider setting this property to False. The safest and most advantageous route is to enable this property when you build in debug mode and turn it off, as an optimization technique, during release builds.

![Image of project properties for checking arithmetic overflows.]

**Figure 1.1** Changing project properties to check for arithmetic overflows.

### 1.7. Working with Floating-Point Numbers

Your application needs to work with floating-point values and you need to determine the correct type to use.

**Technique**

If memory is your primary concern and the range of possible values isn’t too large, use the `float` data type. If you need very large values knowing that you need to sacrifice precision, use the `double` data type. If you need accurate and precise data, use the `decimal` data type.

**Comments**

Just as with integral types, choosing the proper floating-point number is essential to creating a robust application. There are two main points to consider when deciding which floating-point data type to choose: size and precision.
The .NET floating-point data types allow you to use exponential notation. For instance, the float data type has a range of \( \pm 1.5 \times 10^{-45} \) to \( \pm 3.4 \times 10^{38} \). Although this range is certainly large, the precision of the float only allows for a maximum of seven digits. Any digits after the seventh digit are simply changed to a 0. For financial applications, this change could result in the improper calculation of money, which could have harsh repercussions from the clients who use your software. If, however, you are programming a video game, you might not need precision greater than seven digits. You might have small side effects but nothing like the side effects a loss of precision on a financial calculation would yield.

The double data type has the largest range of all the floating types. It can contain values in the range of \( 5.0 \times 10^{-324} \) to \( 1.7 \times 10^{308} \). Its precision is limited to just 15 to 16 digits. Although the precision is larger than that of a float, it still might not be suited for financial calculations.

The decimal data type is the only floating type not based on an IEEE specification; it is an exclusive .NET Framework type. This 128-bit number contains a precision of 28 to -29 digits, which makes it the most precise floating-point type. However, its range isn’t as widespread as the double data type. It can contain values in the range from \( 1.0 \times 10^{-28} \) to \( 7.9 \times 10^{28} \).

To resolve ambiguities when working with floating-point numbers, C# contains a suffix for each type. For instance, if you want to assign a number to a double type, use the suffix d as in 3.14d. To designate a float data type, supercede the number with a f. Finally, to use a decimal number, use the m suffix.

### 1.8. Creating Value Types with struct

You want to create a new data type that behaves more like a value than a class.

**Technique**

Define a structure using the `struct` keyword. A struct behaves similar to a class in that it can contain any number of member variables, methods, properties, and indexers, but memory for it is allocated on the stack rather than the heap, which also implies that it is passed by value rather than by reference. The following code demonstrates a temperature value type that performs a conversion if the Fahrenheit property is accessed:

```csharp
struct TemperatureStruct
{
    public double Celsius;

    public double Fahrenheit
    {
        get
        {
```
Chapter 1  Operators, Expressions, and Control Structures

```
return ((9d/5d)*Celsius)+32;
```

```
set
{
    Celsius = (5d/9d)*(value-32);
}
```

Comments

Sometimes, a language’s built-in data types just aren’t sufficient to use as a data-storing mechanism. Although using them would most certainly work, you generally need several related variables to describe all the pieces of pertinent information about a single object or abstract value type. Structures, or structs, were designed to organize and group built-in data types to create user-defined types that could emulate values.

One of the most prominent distinguishing differences between structs and classes is that you do not have to instantiate a struct using the `new` operator. You don’t have to declare a default constructor because the compiler will ensure that all members are initialized by using a process known as static flow analysis. Sometimes, however, the analysis is unable to determine whether a member can be initialized. You can make the determination by declaring a private member variable within your struct. In this case, you have to create a custom parameterized constructor, and clients using your struct must create it using the `new` operator. Based on this information, you might make the assumption that using `new` on the struct would then allocate it on the heap. We can in fact verify that it does not by looking at the intermediate language (IL) code generated by the compiler using a tool, ILDasm, which is perfect when you want to investigate a certain behavior.

To run the ILDasm tool, you need to navigate to the `bin` directory of the .NET Framework SDK. You can double-click on the file to run it, but there are some advanced options available if you run the executable and specify the command-line argument `/ADV`. For this exercise, you can simply run it without the advanced options. Listing 1.4 shows an application that contains a class called `TemperatureClass` and a struct named `TemperatureStruct`. When you build this application, open the assembly in ILDasm. Expand the tree item of the namespace that contains the class and struct definition, which in this case is `_8_ValueTypes`. You should see three data types defined in the assembly: the main entry point, the temperature class, and the temperature struct. Expand the tree item that denotes the class where the entry point is located. Finally, double-click on the `Main` function to display the disassembled IL code shown in Figure 1.2.
Listing 1.4  Comparing Differences Between a Class and a Struct

```csharp
using System;

namespace _8_ValueTypes
{
    class EntryPoint
    {
        [STAThread]
        static void Main(string[] args)
        {
            TemperatureStruct ts = new TemperatureStruct();
            TemperatureClass tc = new TemperatureClass();

            Console.Write( "Enter degrees in Celsius: " );
            string celsius = Console.ReadLine();
            ts.Celsius = Convert.ToDouble(celsius);

            Console.WriteLine( "Temperature in Fahrenheit = {0}", ts.Fahrenheit );
        }
    }

    class TemperatureClass
    {
        public double Fahrenheit { get; set; }
        public double Celsius { get; set; }
    }

    class TemperatureStruct
    {
        public double Celsius { get; set; }
        public double Fahrenheit { get; set; }
    }
}
```

Figure 1.2  ILDasm is a good tool to understand the inner workings of your application and how it interacts with the CLR.
Listing 1.4  Continued
{
    private double degreesCelsius;
    public double Fahrenheit {
        get {
            return ((9d/5d)*degreesCelsius)+32;
        }
        set {
            degreesCelsius = (5d/9d)*(value-32);
        }
    }
    public double Celsius {
        get {
            return degreesCelsius;
        }
        set {
            degreesCelsius = value;
        }
    }
}

struct TemperatureStruct {
    public double Celsius;
    public double Fahrenheit {
        get {
            return ((9d/5d)*Celsius)+32;
        }
        set {
            Celsius = (5d/9d)*(value-32);
        }
    }
}
Chapter 1  Operators, Expressions, and Control Structures

One particular portion of the IL for the Main function in Listing 1.4 is the four lines following the declaration of the locals that are used within the function:

```
IL_0000:  ldloca.s   ts
IL_0002:  initobj    _8_ValueTypes.TemperatureStruct
IL_0008:  newobj     instance void_8_ValueTypes.TemperatureClass::.ctor()
IL_000d:  stloc.1
```

The first two lines initialize the struct. The important piece of information here is the instructions `ldloca` and `initobj`. Compare these two instructions with the two that are used for initialization of the temperature class, `newobj` and `stloc`. When you use the `new` operator on the class, it generates the `newobj` instruction using the class constructor as the operand. However, when you use `new` on the struct, it performs a `ldloca.s`, which is an instruction used to fetch an object from the stack. Based on this information, even though you use the `new` keyword on a struct, it still follows the rules in that it is a stack-allocated rather than a heap-allocated object.

**Note**
Most of the built-in C# data types are structs themselves. For instance, the `int` is simply an alias for the `System.Int32` structure.

## 1.9. Converting Data Types

You need to convert a value of one type to an equivalent value of another type.

**Technique**

```csharp
long l = 0;
int i = 0;
l = i;
```

If however you need to convert a data type from a larger to a smaller size, you need to explicitly convert the value using the `System.Convert` class or through a technique known as *casting*:

```csharp
long l = 0;
int i = 0;
i = Convert.ToInt32(l);
i = (int)l;
```

For any objects you want to convert to a string, simply use the `ToString` method, which is defined for all .NET value types:

```csharp
int i = 0;
string s = i.ToString();
```
Every value type defined in the .NET Framework contains a static method named `Parse`. It allows you to convert a string value to that value type. The string itself must be formatted correctly or a `FormatException` will occur.

```csharp
string sNum = "3.1415936535";
double dNum = Double.Parse(sNum);
```

**Comments**

Data conversion is a process that tends to occur frequently. The most frequent case is when you need to call a function whose parameters are different types than what you have been using in your application. Regardless of how different those two types are, an implicit or an explicit conversion will take place.

*Implicit* conversion occurs when the data type you want to convert to is a larger size than the data type you are converting from. For example, if a method needs a `long` data type (64 bits in C#) and your variable is an `int`, then the compiler will perform the necessary implicit conversion for you. In other words, you can simply pass the `int` variable into the method that is expecting a `long` parameter, and it will work without any conversion necessary on your part. The high 32 bits of the `long` will simply be all zeros.

If you need to convert a larger value to a smaller value, you have to perform an *explicit* conversion. The easiest way to determine whether you need to do so, other than memorizing all the conversion rules, is receiving an error from the compiler indicating that it could not perform an implicit conversion. To explicitly convert one data type to another, you can use the `System.Convert` class. This class contains several methods and several overloads of those methods to effectively perform explicit conversions. For example, to convert a `long` variable to an `int`, use the `System.Convert.ToInt32` method, passing the `long` variable as the parameter. Looking at this class, you'll see most of the data conversions you need.

A shorthand way of performing an explicit conversion is through a technique known as *casting*. You perform casting by prefacing the variable you want to convert with the data type to convert to within parenthesis. To cast a `long` to an `int`, you use the following:

```csharp
long l;
int i = (int) l;
```

### 1.10. Performing Relational Operations

You need to perform a relational comparison on two values or objects.

**Technique**

C# relational operators compare the actual values of value types but compare references to objects rather than what those objects contain. The format of a relational operation in C# is `expr1 operator expr2` as in `x == y`. Listing 1.5 demonstrates different relational operations by comparing both value types and objects.
using System;

namespace _10_Relations
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class ComparingRelations
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        static void Main(string[] args)
        {
            // compare 2 values for equality
            int a = 12;
            int b = 12;
            Console.WriteLine( a == b );
            Console.WriteLine( (object)a == (object)b );

            // compare 2 objects which contains overloaded == operator
            string c = "hello";
            string d = "hello";
            Console.WriteLine( (object) c==(object) d );

            // compare 2 objects for equality
            ClassCompare x = new ClassCompare();
            ClassCompare y;
            x.val = 1;
            y = x;
            Console.WriteLine( x == y );

            // changing 1 object also changes the other
            x.val = 2;
            Console.WriteLine( y.val.ToString() );
        }
    }

    class ClassCompare
    {
        public int val = 0;
    }
}
Comments

Testing relationships between values or objects is an important concept in programming. It provides the foundation for program flow and is used extensively in human/computer interaction. You use relational operations for determining how two values or objects relate to one another to decide what the next step in your program flow will be. Some of these include testing to see whether two values are equal, testing whether one value is greater than another, or testing whether one object is the same as another object, a process known as type-testing.

One key thing you need to keep in mind when creating relational operations is the difference between testing two value types and testing two objects. When you test value types, such as two `int` values, you are testing the value that those two types contain. Compare it to testing two different objects, shown later in the code, in which testing them means you are testing to see whether they are the same objects. In other words, when you are comparing two objects, you are comparing their locations in memory. If they both point to the same location, then they are considered equal. To test this equality, the last test in the code changes the value in one object and the next line prints a member variable in the second object. When you run this code, you’ll notice that even though the variable was changed in one object, the other changed as well because you modified the same memory location that the second one pointed to. Additionally, when creating classes, you might want to overload the `Equals` method so that any comparisons done on your object will compare its internal values rather than the object references.

One particular thing to watch for is a reference type overloading a relational operator. Overloading an operator, discussed later in this chapter, is a technique to change the behavior of a built-in operator so that it makes better sense given the context of the objects you are working with. As an example, code in Listing 1.5 evaluates to `true` even though the two objects being compared are different. The `string` class within the .NET Framework has overloaded the relational operators so that the comparison happens on the internal string rather than the objects themselves. Comparing the strings of two `string` objects makes more sense than comparing two actual `string` objects themselves.

1.11. Using Logical Expressions

You want to evaluate an expression to see whether it is true.

Technique

Use a combination of logical operators on each relational expression you want to test. Logical operations consist of the logical and operator (`&&`), the logical or operator (`||`), and the logical exclusive-or operator (`^`). The following example demonstrates the logical operators as applied to Boolean values:
Chapter 1 Operators, Expressions, and Control Structures

```csharp
static void Main(string[] args)
{
    Console.WriteLine("true==true: {0}", (true==true).ToString() );
    Console.WriteLine("true==false: {0}", (true==false).ToString() );
    Console.WriteLine("false==false: {0}", (false==false).ToString() );
    Console.WriteLine("true||true: {0}", (true||true).ToString() );
    Console.WriteLine("true||false: {0}", (true||false).ToString() );
    Console.WriteLine("false||false: {0}", (false||false).ToString() );
    Console.WriteLine("true^true: {0}", (true^true).ToString() );
    Console.WriteLine("true^false: {0}", (true^false).ToString() );
    Console.WriteLine("false^false: {0}", (false^false).ToString() );
}
```

**Comments**

Logical operations test the results of two or more relational expressions. In the last recipe, you used relational operators to test the relationship of two values or objects with each other. This evaluation results in a Boolean value, `true` or `false`, which you can then act upon in some manner. One action is to compare the results from two or more of these relational expressions by using logical operations.

Logical operations belong to a larger family of operators known as bitwise operators. The term *bitwise* applies because these operators use the actual bits of a value to determine the result. For instance, a Boolean value of `true` is a single bit (at least in theory), which is “turned on” or equals 1. The Boolean value `false` is “turned off” or set to 0. By using this information, you can generate a table to see the result based on the logical or bitwise operator used on these two values, as shown in Table 1.2.

<table>
<thead>
<tr>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operator</th>
<th>Result</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>&amp;&amp;</td>
<td>True</td>
<td>Both operands must be true.</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>&amp;&amp;</td>
<td>False</td>
<td></td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>&amp;&amp;</td>
<td>False</td>
<td></td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>&amp;&amp;</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>^</td>
<td>False</td>
<td>At least one operand is true, but not both.</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>^</td>
<td>True</td>
<td></td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>^</td>
<td>True</td>
<td></td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>^</td>
<td>False</td>
<td></td>
</tr>
</tbody>
</table>
The logical and, and the logical or operators are also known as short-circuit bitwise operators. If you are using a \texttt{&&} operator, it makes sense that if the first operand is false, there is no need to test the second operand because you know that the logical statement will still evaluate to false. Furthermore, when using the \texttt{||} operator, it makes no sense to test the second operand if the first is already true because that would automatically make the statement true. The exclusive-or operator cannot use any type of short-circuit evaluation because you must test both operands regardless of the value of the first. This difference is why you see two characters for \texttt{&&} and \texttt{||} but only one character for \texttt{^}.

Furthermore, if you want to evaluate both operands and not short-circuit, each operator has a single bitwise operator implementation. However, you generally use these operators for bitwise manipulations of data.

### 1.12. Determining Operator Precedence and Associativity

You want to change the order in which your expression is evaluated by modifying operator precedence.

**Technique**

Our rule of thumb is to parenthesize the expression being evaluated to prevent any possible operator-precedence defects. Determining whether one operator has precedence over another requires understanding all precedence rules for the language you are using.

**Comments**

Operator-precedence rules are one of those things you just have to keep in the back of your head no matter how boring it is. As we said earlier, the rule of thumb is to just use parentheses around certain parts of the expression so that we know the order of evaluation. Parentheses are at the top of the food chain, so to speak, for operator precedence.

Operator precedence isn’t entirely a programming concept. Back in grade-school arithmetic, you learned that multiplication and division is performed from left to right through the expression, followed by addition and subtraction using the result of the multiplication and division expressions as new operands. For instance, the expression $2 + 4 \times 6$ equals 26 because you perform the multiplication first, with a result of 24, followed by the addition of 2. If you want to change the order of this evaluation, use parentheses around the parts of the expression you want to perform first. So if we want to perform the addition followed by the multiplication, we use the expression $(2 + 4) \times 6$ to get a result of 36. Table 1.3 shows the available C# operators and the corresponding precedence and associativity. An operator has precedence over another if it appears higher in the table (closer to the top) than the other. For instance, the $\texttt{==}$ operator has a higher precedence than the $\&$ operator. If both operators appear on the same level, then you must refer to the associativity of those operators. If the associativity is left, then the expression with the operator that appears leftmost in the table is evaluated first.
Table 1.3  C# Operator Precedence and Associativity

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x), x, y, f(x), a[x], x++, x--, new, typeof, sizeof, checked, unchecked</td>
<td>Left</td>
</tr>
<tr>
<td>unary +, unary --, ++x, --x, (type)x</td>
<td>Left</td>
</tr>
<tr>
<td>*, /, %</td>
<td>Left</td>
</tr>
<tr>
<td>+, -</td>
<td>Left</td>
</tr>
<tr>
<td>&lt;&lt;, &gt;&gt;</td>
<td>Left</td>
</tr>
<tr>
<td>&lt;, &lt;=, &gt;, &gt;=, is, as</td>
<td>Left</td>
</tr>
<tr>
<td>==, !=</td>
<td>Left</td>
</tr>
<tr>
<td>&amp;</td>
<td>Left</td>
</tr>
<tr>
<td>^</td>
<td>Left</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Left</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>?: (ternary)</td>
<td>Right</td>
</tr>
<tr>
<td>=, *=, /=, %=, +=, -=, &lt;&lt;=, &gt;&gt;=, &amp;=,</td>
<td>=, ^=</td>
</tr>
</tbody>
</table>

1.13. Using if Statements

You want your application to perform a certain action based on the result of evaluating an expression.

Technique

Using a combination of relational operations on variables, you can execute a body of code if the expression evaluates to true or evaluate a different body of code if not. Use the if/else statement to control program flow.

Comments

One of the most widely used conditional statements is the if/else statement. The format of an if/else statement follows:

```csharp
if( expr )
{
}
else
{
}
```
You read this code as “If expr is true, then perform these actions; else, perform these actions.” In the previous recipe, you used relational operators to compare values and objects with one another. You can then combine the result of these relational expressions using logical expressions. You use the results of these comparisons within the if statement, which is known collectively as a conditional statement.

A shorthand way of creating an if statement is to use a ternary operator. The format of a ternary operator is

```
(conditional)? true_expression : false_expression
```

As an example, if you want to test whether a variable is equal to a certain number, you write the following:

```
Console.Write(( i==42 )? "Meaning of life" : "I don't know");
```

A common technique is to create multiple if statements to further refine the conditional comparisons of variables. For example, the following is perfectly legal and you'll notice that it can perform one of several different actions based on the conditions that are present. Listing 1.6 demonstrates how to create multiple if statements.

Listing 1.6 Creating Multiple if Statements to Control Several Actions

```csharp
namespace _13_IfStatement
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        /// <STAThread>
        static void Main(string[] args)
        {
            string input;
            Console.Write( "Enter your name: " );
            input = Console.ReadLine();
            if( input.ToLower() == "jordan" )
            {
                Console.WriteLine( "{0}, have you cleaned " + "your room yet?", input );
            }
            else if( (input.ToLower() == "jake") ||
                     (input.ToLower() == "jonah") )
            {
            }
```
Chapter 1  Operators, Expressions, and Control Structures

Listing 1.6  Continued

    Console.WriteLine( "{0}, clean your room!", input );
   }
else if( input.ToLower() == "mallorie" )
    {
        Console.WriteLine( "{0}, you may do anything you " +
                           "want.", input );
    }
else
    {
        Console.WriteLine( "Sorry {0}, I don't know you",
                           input );
    }
}

1.14. Using Looping Control Structures

You need to continuously perform a set of actions a certain number of times or
until a certain condition becomes false.

Technique

C# contains several different looping control structures to handle different cases. You use
a for statement if you need to run a loop a set number of times. If you need to execute
a block of statements while an expression remains true, use a while loop. Choose a
do/while statement if you need to execute a block of statements at least once regardless
of any pre-existing conditions and then continuously while an expression remains true.

Comments

In the last recipe, you saw how you can use if statements to execute blocks of code
based on certain conditions. However, you could only execute that block of code one
time. Looping control structures allow you to execute blocks of code several times and
stop either when an expression finally evaluates to false or when you explicitly break out
of the loop somewhere within the body of the loop using the C# keyword break.
Using loops requires care because the conditional statement to test whether the loop
should continue might never become false, which means the program gets stuck in an
endless loop. All three of the looping statements covered here have the possibility of con-
tinuing forever.
You generally use the `for` loop when you want to execute a block of code a certain amount of times. An index variable associated with the loop keeps count of the iterations that have occurred and if you tested it to see whether the loop should finish executing. The `for` statement contains three sections as shown in Listing 1.7. The first part is for initialization. In most cases, you initialize the variable used to control the `for` loop. However, you are free to initialize other variables or even call methods as long as they are separated by commas. The second component of the `for` statement is the conditional expression you want to evaluate. If the expression remains true, the `for` loop body executes. Finally, the last field of the `for` loop is for changing the variables associated with the conditional statement by either incrementing or decrementing the loop counter, for example.

Listing 1.7  Using a `for` Statement to Loop and Calculate the Factorial of a Number

```csharp
static void ComputeFactorial()
{
    ulong loopCount = 0;
    ulong factorial = 1;
    Console.Write( "Enter an integer between 1 and 50: " );
    loopCount = UInt64.Parse( Console.ReadLine() );
    for( ulong i = loopCount; i > 0; i-- )
    {
        factorial *= i;
    }
    Console.WriteLine( "{0}! = {1}", loopCount, factorial );
}
```

The `while` and `do/while` statements are quite similar in that both loop while a certain condition holds true. Once that condition evaluates to false, the loop is exited and the next statement following the loop block is executed. The major difference between the two loop structures is that the `do/while` statement's code block is guaranteed to execute at least once. The `while` statement appears at the end of the code block, as shown in Listing 1.8, instead of at the beginning as with the `while` statement.

Listing 1.8  Using the `do/while` Statement to Control Program Flow

```csharp
static void NumberGuessGame()
{
    int guess, number, guesses = 0;
    number = new Random((int)DateTime.Now.Ticks).Next( 0, 10 );
    do
    {
        ++guesses;
        Console.Write( "Enter a number between 1 and 10: " );
        guess = Int32.Parse( Console.ReadLine() );
    } while( guess != number );
    Console.WriteLine( "You took {0} tries to guess the number: {1}" , guesses, number );
}
```
Chapter 1  Operators, Expressions, and Control Structures

Listing 1.8  Continued

if( guess < number )
    Console.WriteLine( "Too low. Pick a higher number" );
else if ( guess > number )
    Console.WriteLine( "Too high. Pick a lower number" );
} while (guess != number );

Console.WriteLine( "You are correct and it only took you \{0\} guesses!", guesses );

A while statement might never run if the condition is always false during the execution of a program because the condition is evaluated before the code block is entered. If the initial condition is false, then the block is skipped and the program continues after that point. Listing 1.9 shows a while loop being used to create a countdown timer.

Listing 1.9  The while Statement

using System;

namespace _14_Looping
{
    class Game
    {
        [STAThread]
        static void Main(string[] args)
        {
            WaitForNewMinute();
            NumberGuessGame();
        }

        static void WaitForNewMinute()
        {
            int sec = -1;
            Console.Write( "The game will start in " );
            while( DateTime.Now.Second != 0 )
            {
                if( sec != DateTime.Now.Second )
                {
                    sec = DateTime.Now.Second;
                    Console.Write( "...{0}", 60-DateTime.Now.Second );
                }
            }
        }
    }
}
1.15. Breaking Out of a Loop Control Body

You need to repeatedly execute statements in a block but break out of the loop based on a certain event.

**Technique**

There are two ways to break out of a loop statement in the middle of a loop body. The first is by using the `break` keyword. Using it will break out of the loop entirely, and execution will begin at the statement following your loop block. The second way is to use the `continue` keyword, which will cause your application to skip the rest of the loop body but then reevaluate the loop conditional and possibly enter the loop again.

**Comment**

Although the `break` and `continue` keywords are rarely used, sometimes they might prove to be the only alternative. As mentioned earlier, use a `break` statement when you are within the loop body and need to exit the loop entirely. You simply exit the loop, and the program continues execution at the statement following the loop body. An example is a modification of the game in Listing 1.9 in which the loop will exit if the user enters a number that is one less than the randomly generated number.

Listing 1.10  Using the break Keyword to Make the Number-Guessing Game a Little Easier

```csharp
static void NumberGuessGame()
{
    int guess, number, guesses = 0;
    number = new Random((int)DateTime.Now.Ticks).Next(0, 10);

    do
    {
        ++guesses;

        Console.Write( "Enter a number between 1 and 10: " );
        guess = Int32.Parse( Console.ReadLine() );
    
        if (guess == number) {
            Console.WriteLine("You guessed the right number!");
            break;
        } else if (guess < number) {
            Console.WriteLine("The number is higher.");
        } else if (guess > number) {
            Console.WriteLine("The number is lower.");
        }

    } while (guesses < 10);
}
```

Listing 1.9  Continued

```csharp
    Console.WriteLine();
} } }
```
if( guess-1 == number )
    break;  // close enough

if( guess < number )
    Console.WriteLine( "Too low. Pick a higher number" );
else if ( guess > number )
    Console.WriteLine( "Too high. Pick a lower number" );

} while (guess != number );

Console.WriteLine( "You are correct and it only took you "+
"{0} tries!", guesses );

If you want to instead make the game a little harder, you could set a difficulty value and
use the continue keyword to bypass the statements that tell the user whether her guess is
too high or too low. Yes, using an if statement for these cases seems like a more logical
choice, which is a reason why we said the break and continue statements are rarely used.