Implementing Logical Expressions

Logical Operators

Logical *expressions* are composed of *operators* and *operands*. You can define your own operators in C#, but that's a more advanced topic. C# provides many built-in operators that you can use to create logical expressions, which are useful when implementing conditional statements, as you saw in the previous lesson. Logical expressions evaluate to `true` or `false`, and as such they are often called `boolean` expressions, since they are evaluated as `bool` types (and can be assigned to boolean variables).

Comparison Operators

You've already seen several comparison operators. Below are many of the built-in operators:

```
==  // equal
!=  // not equal
>   // greater than
<   // less than
>=  // greater than or equal
<=  // less than or equal
```

Tip {.tip .javascript}
Unlike in JavaScript, C# uses strict comparison rules, so its equality comparisons will work like `===` and `!==` by default.

Each of the comparison operators requires two operands, one on each side of the expression. For example:

```
x < 10
y >= 0
```

```
using System;

namespace ConsoleApplication
{
    public class Program
    {
        public static void Main()
        {
            int x = 15;
            int y = 10;
            Console.WriteLine(x < 10);
            Console.WriteLine(y >= 0);
```

```
        }
    }
}
```

To test if a value is between two numbers, you need to check the two conditions separately. The following is not a legal C# expression:

```
1 <= x <= 10 // x between 1 and 10 inclusive - DOES NOT COMPILE
```

To combine multiple expression, you use logical operators.

Conditional Logical Operators

Conditional logical operators are used to combine multiple logical expressions. The most common logical operators are *and*, *or*, and *not*, which are represented as follows:

```
&&  // logical AND
||  // logical OR
!   // logical NOT (often read as 'bang')
^   // logical XOR (exclusive OR)
```

The `&&`, `||`, and `^` operators require two operands; the `!` operator takes only one, and is applied as a prefix. For example:

```
true && true    // true
true && false   // false
false && false  // false

true || true    // true
true || false   // true
false || false  // false

true ^ true     // false
true ^ false    // true
false ^ false   // false

!true           // false
!false          // true
```

```
using System;

namespace ConsoleApplication
{
    public class Program
    {
        public static void Main()
        {
            Console.WriteLine(true && true);    // true
```

```
            Console.WriteLine(true && false);   // false
            Console.WriteLine(false && false);  // false

            Console.WriteLine(true || true);    // true
            Console.WriteLine(true || false);   // true
            Console.WriteLine(false || false);  // false

            Console.WriteLine(true ^ true);     // false
            Console.WriteLine(true ^ false);    // true
            Console.WriteLine(false ^ false);   // false

            Console.WriteLine(!true);           // false
            Console.WriteLine(!false);          // true
        }
    }
}
```

Note {.note}
C# also includes *bitwise* logical operators, `&` (AND), `|` (OR). These are used to perform binary comparisons of numeric values, and generally aren't used directly for conditional expressions. The `^` (XOR) operator can be used with both boolean and integral operands.

Logical operations are applied left to right, and will *short-circuit*. That is, if the left operand of an `&&` operator is false, the right operand will not be evaluated. This is often important, since the operands themselves may be method calls, not variables. You'll learn more about methods in lesson 12.

Logical expressions are grouped using parentheses, which modify their order of operations just as in algebra. For example:

```
int a = 5;
int b = 10;
if ((a < b) && (b < 20))
{
    // do something
}
```

```
using System;

namespace ConsoleApplication
{
    public class Program
    {
        public static void Main()
        {
            int a = 5;
            int b = 10;
            if ((a < b) && (b < 20))
            {
                Console.WriteLine("I'm in here!");
            }
```

```
        }
    }
}
```

Tip {.tip .newLanguage}
Be careful with parentheses in logical expressions - it's easy to forget to close one. It can be a good practice, especially when you're getting started, to type all of the pairs of parentheses first, and then fill in the values and expressions.

Flags

A boolean (`bool`) variable is often referred to as a *flag*. Flags can be useful as a means giving a name to a particular condition. For example:

```
int x = 10;
bool isPositive = x > 0;
if (isPositive)
{
    // do something
}
```

```
using System;

namespace ConsoleApplication
{
    public class Program
    {
        public static void Main()
        {
            int x = 10;
            bool isPositive = x > 0;
            if (isPositive)
            {
                Console.WriteLine("I'm in here!");
            }
        }
    }
}
```

Often, programs will include complex conditional logic, and it can be helpful to simplify some or all of this complexity into named variables. Flags can be attached to objects as *properties*, such that you can test for their value as part of the object itself (example: `if (x.IsPositive)` . However, when writing *object-oriented* programs, it's often better to avoid using flags, since they can lead to program designs that are more procedural and don't encapsulate behavior effectively within objects. You'll learn more about object design in the Encapsulation and Object-Oriented Design lesson.

Tip {.tip .newLanguage}
It's a typical convention to name boolean variables with an "Is" or "is" prefix, since this makes it clear the variable is a boolean and also makes reading conditional statements more clear as well. Avoid naming flags negatively (example: "IsNotPositive"), since this can become confusing, especially when the flag is negated with the ！ operator.