In this chapter, you will learn about operators, control flow statements, and the C# preprocessor. **Operators** provide syntax for performing different calculations or actions appropriate for the operands within the calculation. **Control flow statements** provide the means for conditional logic within a program or looping over a section of code multiple times. After introducing the `if` control flow statement, the chapter looks at the concept of Boolean expressions, which are embedded within many control flow statements. Included is mention of how integers will not cast (even explicitly) to `bool` and the advantages of this restriction. The chapter ends with a discussion of the C# preprocessor directives.

**Operators**

Now that you have been introduced to the predefined data types (refer to Chapter 2), you can begin to learn more about how to use these data types in combination with operators in order to perform calculations. For example, you can make calculations on variables that you have declared.

**BEGINNER TOPIC**

**Operators**

Operators are used to perform mathematical or logical operations on values (or variables) called **operands** to produce a new value, called the **result**. For example, in Listing 3.1 the subtraction operator, `-`, is used to subtract two operands, the numbers 4 and 2. The result of the subtraction is stored in the variable `difference`. 

```csharp
int a = 4;
int b = 2;
difference = a - b;
```
Operators are generally broken down into three categories: unary, binary, and ternary, corresponding to the number of operands 1, 2, and 3, respectively. This section covers some of the most basic unary and binary operators. Introduction to the ternary operator appears later in the chapter.

**Plus and Minus Unary Operators (+, -)**

Sometimes you may want to change the sign of a numerical value. In these cases, the unary minus operator (-) comes in handy. For example, Listing 3.2 changes the total current U.S. debt to a negative value to indicate that it is an amount owed.

Listing 3.2: Specifying Negative Values

```csharp
// National Debt to the Penny
decimal debt = -15236332233848.35M;
```

Using the minus operator *is equivalent to subtracting the operand from zero.*

The unary plus operator (+) rarely\(^2\) has any effect on a value. It is a superfluous addition to the C# language and was included for the sake of symmetry.

**Arithmetic Binary Operators (+, -, *, /, %)**

Binary operators require two operands. C# uses infix notation for binary operators: The operator appears between the left and right operands. The result of every binary operator other than assignment must be used somehow: for example, by using it as an operand in another expression such as an assignment.

---

\(^1\) As of January 12, 2012, according to www.treasurydirect.gov.

\(^2\) The unary + operator is defined to take operands of type `int`, `uint`, `long`, `ulong`, `float`, `double`, and `decimal` (and nullable versions of those types). Using it on other numeric types such as `short` will convert its operand to one of these types as appropriate.
Language Contrast: C++—Operator-Only Statements

In contrast to the rule mentioned above, C++ will allow a single binary expression to form the entirety of a statement, such as $4+5$, to compile. In C#, only call, increment, decrement, and object creation expressions are allowed to be the entirety of a statement.

The subtraction example in Listing 3.3 is an example of a binary operator—more specifically, an arithmetic binary operator. The operands appear on each side of the arithmetic operator and then the calculated value is assigned. The other arithmetic binary operators are addition (+), division (/), multiplication (*), and remainder (%)—sometimes called the mod operator.

Listing 3.3: Using Binary Operators

```csharp
class Division
{
    static void Main()
    {
        int numerator;
        int denominator;
        int quotient;
        int remainder;

        System.Console.Write("Enter the numerator: ");
        numerator = int.Parse(System.Console.ReadLine());

        System.Console.Write("Enter the denominator: ");
        denominator = int.Parse(System.Console.ReadLine());

        quotient = numerator / denominator;
        remainder = numerator % denominator;

        System.Console.WriteLine(
            "\{0\} / \{1\} = \{2\} with remainder \{3\}",
            numerator, denominator, quotient, remainder);
    }
}
```

Output 3.1 shows the results of Listing 3.3.
In the highlighted assignment statements above, the division and remainder operations are executed before the assignments. The order in which operators are executed is determined by their precedence and associativity. The precedence for the operators used so far is as follows.

***Production, please ensure that the items in the following numbered list are numbered sequentially, starting with 1.***

1. *, /, and % have highest precedence.
2. + and - have lower precedence.
3. = has the lowest precedence of these six operators.

Therefore, you can assume that the statement behaves as expected, with the division and remainder operators executing before the assignment.

If you forget to assign the result of one of these binary operators, you will receive the compile error shown in Output 3.2.

---

**BEGINNER TOPIC**

**Parentheses, Associativity, Precedence, and Evaluation**

When an expression contains multiple operators it can be unclear what precisely the operands of each operator are. For example, in the expression `x+y*z` clearly the expression `x` is an operand of the addition and `z` is an operand of the multiplication. But is `y` an operand of the addition or the multiplication?

**Parentheses** allow you to unambiguously associate an operand with its operator. If you wish `y` to be a summand, you can write the expression as `(x+y)*z`; if you want it to be a multiplicand, you can write `x+(y*z)`.

However, C# does not require you to parenthesize every expression containing more than one operator; instead, the compiler can use associativity and precedence to figure out from the context what parentheses you have omitted. **Associativity** determines how
similar operators are parenthesized; **precedence** determines how dissimilar operators are parenthesized.

A binary operator may be “left-associative” or “right-associative,” depending on whether the expression “in the middle” belongs to the operator on the left or the right. For example, a-b-c is assumed to mean (a-b)-c, and not a-(b-c); subtraction is therefore said to be “left-associative.” Most operators in C# are left-associative; the assignment operators are right-associative.

When the operators are dissimilar, the **precedence** for those operators is used to determine which side the operand in the middle belongs to. For example, multiplication has higher precedence than addition, and therefore, the expression x+y*z is evaluated as x+(y*z) rather than (x+y)*z.

It is often still a good practice to use parentheses to make the code more readable even when use of parentheses does not change the meaning of the expression. For example, when performing a Celsius-to-Fahrenheit temperature conversion, (c*9.0/5.0)+32.0 is easier to read than c*9.0/5.0+32.0, even though the parentheses are completely unnecessary.

**Guidelines**

DO use parentheses to make code more readable, particularly if the operator precedence is not clear to the casual reader.

Clearly, operators of higher precedence must execute before adjoining operators of lower precedence: in x+y*z the multiplication must be executed before the addition because the result of the multiplication is the left-hand operand of the addition. However, it is important to realize that precedence and associativity affect only the order in which the operators themselves are executed; they do not in any way affect the order in which the operands are evaluated.

Operands are always evaluated left-to-right in C#. In an expression with three method calls such as A()+B()*C(), first A() is evaluated, then B(), then C(), then the multiplication operator determines the product, and then the addition operator determines the sum. Just because C() is involved in a multiplication and A() is involved in a lower-precedence addition does not imply that method invocation C() happens before method invocation A().
Language Contrast: C++: Evaluation Order of Operands

In contrast to the rule mentioned above, the C++ specification allows an implementation broad latitude to decide the evaluation order of operands. When given an expression such as A()+B()*C(), a C++ compiler can choose to evaluate the function calls in any order, just so long as the product is one of the summands. For example, a legal compiler could evaluate B(), then A(), then C(), then the product, and then the sum.

Using the Addition Operator with Strings
Operators can also work with non-numeric operands. For example, it is possible to use the addition operator to concatenate two or more strings, as shown in Listing 3.4.

Listing 3.4: Using Binary Operators with Non-Numeric Types

```csharp
class FortyTwo
{
    static void Main()
    {
        short windSpeed = 42;
        System.Console.WriteLine("The original Tacoma Bridge in Washington\nwas brought down by a " + windSpeed + " mile/hour wind.");
    }
}
```

Output 3.3 shows the results of Listing 3.4.

```
Output 3.3

The original Tacoma Bridge in Washington
was brought down by a 42 mile/hour wind.
```

Because sentence structure varies among languages in different cultures, developers
should be careful not to use the addition operator with strings that require localization. Composite formatting is preferred (refer to Chapter 1).

Guidelines

DO favor composite formatting over the addition operator for concatenating strings.

Using Characters in Arithmetic Operations

When introducing the char type in the preceding chapter, we mentioned that even though it stores characters and not numbers, the char type is an integral type (“integral” means it is based on an integer). It can participate in arithmetic operations with other integer types. However, interpretation of the value of the char type is not based on the character stored within it, but rather on its underlying value. The digit 3, for example, contains a Unicode value of 0x33 (hexadecimal), which in base 10 is 51. The digit 4, on the other hand, contains a Unicode value of 0x34, or 52 in base 10. Adding 3 and 4 in Listing 3.5 results in a hexadecimal value of 0x167, or 103 in base 10, which is equivalent to the letter g.

Listing 3.5: Using the Plus Operator with the char Data Type

```csharp
int n = '3' + '4';
char c = (char)n;
System.Console.WriteLine(c); // Writes out g.
```

Output 3.4 shows the results of Listing 3.5.

Output 3.4

g

You can use this trait of character types to determine how far two characters are from each other. For example, the letter f is three characters away from the letter c. You can determine this value by subtracting the letter c from the letter f, as Listing 3.6 demonstrates.
Listing 3.6: Determining the Character Difference between Two Characters

```csharp
int distance = 'f' - 'c';
System.Console.WriteLine(distance);
```

Output 3.5 shows the results of Listing 3.6.

**OUTPUT 3.5**

| 3 |

Special Floating-Point Characteristics

The binary floating-point types, `float` and `double`, have some special characteristics, such as the way they handle precision. This section looks at some specific examples, as well as some unique floating-point type characteristics.

A `float`, with seven decimal digits of precision, can hold the value 1,234,567 and the value 0.1234567. However, if you add these two floats together, the result will be rounded to 1234567, because the exact result requires more precision than the seven significant digits that a `float` can hold. The error introduced by rounding off to seven digits can become large compared to the value computed, especially with repeated calculations. (See also the upcoming Advanced Topic, Unexpected Inequality with Floating-Point Types.)

Note that internally the binary floating-point types actually store a binary fraction, not a decimal fraction. This means that “representation error” inaccuracies can occur with a simple assignment, such as `double number = 140.6F`. The exact value of 140.6 is the fraction $\frac{703}{5}$, but the denominator of that fraction is not a power of two, and therefore, it cannot be represented exactly by a binary floating-point number. The value actually represented is the closest fraction with a power of two in the denominator that will fit into the 16 bits of a `float`.

Since the `double` can hold a more accurate value than the `float` can store, the C# compiler will actually evaluate this expression to `double number = 140.600006103516` because 140.600006103516 is the closest binary fraction to 140.6 as a `float`. This fraction is slightly larger than 140.6 when represented as a `double`.

<table>
<thead>
<tr>
<th>Guidelines</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVOID binary floating-point types when exact decimal arithmetic is</td>
</tr>
</tbody>
</table>
required; use the decimal floating-point type instead.

ADVANCED TOPIC

Unexpected Inequality with Floating-Point Types

Because floating-point numbers can be unexpectedly rounded off to nondecimal fractions, comparing floating-point values for equality can be quite confusing. Consider Listing 3.7.

Listing 3.7: Unexpected Inequality Due to Floating-Point Inaccuracies

```csharp
decimal decimalNumber = 4.2M;
double doubleNumber1 = 0.1F * 42F;
double doubleNumber2 = 0.1D * 42D;
float floatNumber = 0.1F * 42F;

Trace.Assert(decimalNumber != (decimal)doubleNumber1);
//@ Displays: 4.2 != 4.20000006258488
System.Console.WriteLine(
    "{0} != {1}", decimalNumber, (decimal)doubleNumber1);

Trace.Assert((double)decimalNumber != doubleNumber1);
//@ Displays: 4.2 != 4.20000006258488
System.Console.WriteLine(
    "{0} != {1}", decimalNumber, doubleNumber1);

Trace.Assert((float)decimalNumber != floatNumber);
//@ Displays: (float)4.2M != 4.2F
System.Console.WriteLine(
    "(float){0}M != {1}F", 
    (float)decimalNumber, floatNumber);

Trace.Assert(doubleNumber1 != (double)floatNumber);
//@ Displays: 4.20000006258488 != 4.200000028610229
System.Console.WriteLine(
    "{0} != {1}", doubleNumber1, (double)floatNumber);

Trace.Assert(doubleNumber1 != doubleNumber2);
//@ Displays: 4.20000006258488 != 4.2
System.Console.WriteLine(
    "{0} != {1}", doubleNumber1, doubleNumber2);

Trace.Assert(floatNumber != doubleNumber2);
//@ Displays: 4.2F != 4.2D
```
System.Console.WriteLine("{0}F  !=  (1)D", floatNumber, doubleNumber2);

Trace.Assert((double)4.2F != 4.2D);
// Display: 4.19999980926514 != 4.2
System.Console.WriteLine("(0) != (1)", (double)4.2F, 4.2D);

Trace.Assert(4.2F != 4.2D);
// Display: 4.2F != 4.2D
System.Console.WriteLine("(0)F != (1)D", 4.2F, 4.2D);

Output 3.6 shows the results of Listing 3.7.

OUTPUT 3.6

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
</tr>
<tr>
<td>4.200000006259488</td>
</tr>
<tr>
<td>(float)4.2M != 4.2F</td>
</tr>
<tr>
<td>4.200000006259488 != 4.2</td>
</tr>
<tr>
<td>4.2F != 4.2D</td>
</tr>
<tr>
<td>4.19999980926514 != 4.2</td>
</tr>
</tbody>
</table>

The Assert() methods alert the developer whenever their argument evaluates to false. However, of all the Assert() calls in this code listing, only half have arguments that evaluate to true. In spite of the apparent equality of the values in the code listing, they are in fact not equivalent due to the inaccuracies of a float.

<table>
<thead>
<tr>
<th>Guidelines</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVOID using equality conditionals with binary floating-point types. Either subtract the two values and see if their difference is less than a tolerance, or use the decimal type.</td>
</tr>
</tbody>
</table>

**Production, Advanced Topic ends here**

You should be aware of some additional unique floating-point characteristics as well. For instance, you would expect that dividing an integer by zero would result in an error, and it does with data types such as int and decimal. The float and double types instead allow for certain special values. Consider Listing 3.8, and its resultant output, Out-
Listing 3.8: Dividing a Float by Zero, Displaying NaN

```csharp
float n=0f;
// Displays: NaN
System.Console.WriteLine(n / 0);
```

OUTPUT 3.7

NaN

In mathematics, certain mathematical operations are undefined, including dividing zero by itself. In C#, the result of dividing the float zero by zero results in a special “Not a Number” value; all attempts to print the output of such a number will result in NaN. Similarly, taking the square root of a negative number with `System.Math.Sqrt(-1)` will result in NaN.

A floating-point number could overflow its bounds as well. For example, the upper bound of the `float` type is approximately $3.4 \times 10^{38}$. Should the number overflow that bound, the result would be stored as “positive infinity” and the output of printing the number would be Infinity. Similarly, the lower bound of a `float` type is $-3.4 \times 10^{38}$, and computing a value below that bound would result in “negative infinity,” which would be represented by the string `-Infinity`. Listing 3.9 produces negative and positive infinity, respectively, and Output 3.8 shows the results.

Listing 3.9: Overflowing the Bounds of a `float`

```csharp
// Displays: -Infinity
System.Console.WriteLine(-1f / 0);
// Displays: Infinity
System.Console.WriteLine(3.402823E+38f * 2f);
```

OUTPUT 3.8

-Infinity
Infinity

Further examination of the floating-point number reveals that it can contain a value very close to zero, without actually containing zero. If the value exceeds the lower threshold for the `float` or `double` type, the value of the number can be represented as “negative zero” or “positive zero,” depending on whether the number is negative or
positive, and is represented in output as -0 or 0.

**Compound Assignment Operators (+=, -=, *=, /=, %=)**

Chapter 1 discussed the simple assignment operator, which places the value of the right-hand side of the operator into the variable on the left-hand side. Compound assignment operators combine common binary operator calculations with the assignment operator. Take Listing 3.10, for example.

Listing 3.10: Common Increment Calculation

```c
int x = 123;
x = x + 2;
```

In this assignment, first you calculate the value of \(x + 2\) and then you assign the calculated value back to \(x\). Since this type of operation is relatively frequent, an assignment operator exists to handle both the calculation and the assignment with one operator. The \(+=\) operator increments the variable on the left-hand side of the operator with the value on the right-hand side of the operator, as shown in Listing 3.11.

Listing 3.11: Using the \(+=\) Operator

```c
int x = 123;
x += 2;
```

This code, therefore, is equivalent to Listing 3.10.

Numerous other combination assignment operators exist to provide similar functionality. You can also use the assignment operator with subtraction, multiplication, division and the remainder operators (Listing 3.12 demonstrates).

Listing 3.12: Other Assignment Operator Examples

```c
x -= 2;
x /= 2;
x *= 2;
x %= 2;
```

**Increment and Decrement Operators (++, --)**

C# includes special unary operators for incrementing and decrementing counters. The **increment operator**, ++, increments a variable by one each time it is used. In other words, all of the code lines shown in Listing 3.13 are equivalent.
Listing 3.13: Increment Operator

```csharp
spaceCount = spaceCount + 1;
spaceCount += 1;
spaceCount++;
```

Similarly, you can also decrement a variable by one using the **decrement operator**, `--`. Therefore, all of the code lines shown in Listing 3.14 are also equivalent.

Listing 3.14: Decrement Operator

```csharp
lines = lines - 1;
lines -= 1;
lines--;
```

---

**BEGINNER TOPIC**

A Decrement Example in a Loop

The increment and decrement operators are especially prevalent in loops, such as the `while` loop described later in the chapter. For example, Listing 3.15 uses the decrement operator in order to iterate backward through each letter in the alphabet.

Listing 3.15: Displaying Each Character's Unicode Value in Descending Order

```csharp
char current;
int unicodeValue;

// Set the initial value of current.
current='z';

do
{
    // Retrieve the Unicode value of current.
    unicodeValue = current;
    System.Console.Write("{0}={1}\n", current, unicodeValue);

    // Proceed to the previous Letter in the alphabet;
    current--;
}
while(current>='a');
```

Output 3.9 shows the results of Listing 3.15.

**OUTPUT 3.9**

```
z=122 y=121 x=120 w=119 v=118 u=117 t=116 s=115 r=114
```
The increment and decrement operators are used to control how many times a particular operation is performed. Notice also that in this example, the increment operator is used on a character (char) data type. You can use increment and decrement operators on various data types as long as some meaning is assigned to the concept of the “next” or “previous” value for that data type.

***Production, Beginner Topic ends here***

We saw that the assignment operator first computes the value to be assigned, and then causes the assignment. The result of the assignment operator is the value that was assigned. The increment and decrement operators are similar: They compute the value to be assigned, perform the assignment, and result in a value. It is therefore possible to use the assignment operator with the increment or decrement operator, though doing so carelessly can be extremely confusing. See Listing 3.16 and Output 3.10 for an example.

Listing 3.16: Using the Post-Increment Operator

```csharp
int count = 123;
int result;
result = count++;
System.Console.WriteLine("result = {0} and count = {1}", result, count);
```

**OUTPUT 3.10**

result = 123 and count = 124

You might be surprised that result was assigned the value that was count before count was incremented. Where you place the increment or decrement operator determines whether the assigned value should be the value of the operand before or after the calculation. If you want the value of result to be the value assigned to count, you need to place the operator before the variable being incremented, as shown in Listing 3.17.

Listing 3.17: Using the Pre-Increment Operator

```csharp
int count = 123;
int result;
result = ++count;
System.Console.WriteLine("result = {0} and count = {1}", result, count);
```
Output 3.11 shows the results of Listing 3.17.

Output 3.11

| result = 124 and count = 124 |

In this example, the increment operator appears before the operand, so the result of the expression is the value assigned to the variable after the increment. If count is 123, ++count will assign 124 to count and produce the result 124. By contrast, the postfix increment operator count++ assigns 124 to count and produces the value that count held before the increment: 123. Regardless of whether the operator is postfix or prefix, the variable count will be incremented before the value is produced; the only difference is which value is produced. The difference between prefix and postfix behavior appears in Listing 3.18. The resultant output is shown in Output 3.12.

Listing 3.18: Comparing the Prefix and Postfix Increment Operators

```csharp
class IncrementExample
{
    public static void Main()
    {
        int x = 123;
        // Displays 123, 124, 125.
        System.Console.WriteLine("{0}, {1}, {2}", x++, x++, x);
        // x now contains the value 125.
        // Displays 126, 127, 128
        System.Console.WriteLine("{0}, {1}, {2}", ++x, ++x, x);
        // x now contains the value 128.
    }
}
```

Output 3.12

| 123, 124, 125 |
| 126, 127, 128 |

As Listing 3.18 demonstrates, where the increment and decrement operators appear relative to the operand can affect the result produced by the expression. The result of the prefix operators is the value that the variable had before it was incremented or decremented. The result of the postfix operators is the value that the variable had after it was incremented or decremented. Use caution when embedding these operators in the middle of a statement. When in doubt as to what will happen, use these operators independently, placing them within their own statements. This way, the code is also more
readable and there is no mistaking the intention.

**Language Contrast: C++—Implementation-Defined Behavior**

Earlier we discussed how in C++, the operands in an expression can be evaluated in any order, whereas in C# they are always evaluated left to right. Similarly, in C++ an implementation may legally perform the side effects of increments and decrements in any order. For example, in C++ a call of the form \( M(x++, x++) \) where \( x \) begins as 1 can legally call \( M(1, 2) \) or \( M(2, 1) \) at the whim of the compiler; C# will always call \( M(1, 2) \) because C# makes two guarantees: first, that the arguments to a call are always computed left to right, and second, that the assignment of the incremented value to the variable always happens before the value of the expression is used. C++ makes neither guarantee.

**Guidelines**

- **AVOID** confusing usages of the increment and decrement operators.
- **DO** be cautious when porting code between C, C++, and C# that uses increment and decrement operators; C and C++ implementations need not follow the same rules as C#.

**ADVANCED TOPIC**

**Thread-Safe Incrementing and Decrementing**

In spite of the brevity of the increment and decrement operators, these operators are not atomic. A thread context switch can occur during the execution of the operator and can cause a race condition. You could use a `lock` statement to prevent the race condition. However, for simple increments and decrements, a less expensive alternative is to use the thread-safe `Increment()` and `Decrement()` methods from the `System.Threading.Interlocked` class. These methods rely on processor functions for per-
forming fast thread-safe increments and decrements. See Chapter 19 for more details.

***Production, Advanced Topic ends here***

**Constant Expressions and Constant Locals**

The preceding chapter discussed literal values, or values embedded directly into the code. It is possible to combine multiple literal values in a *constant expression* using operators. By definition, a constant expression is one that the C# compiler can evaluate at compile time (instead of calculating it when the program runs) because it is composed entirely of constant operands. Constant expressions can then be used to initialize constant locals, which allow you to give a name to a constant value (similar to the way local variables allow you to give a name to a storage location). For example, the computation of the number of seconds in a day can be a constant expression that is then used in other expressions by name.

The `const` keyword in Listing 3.19 declares a constant local. Since a constant local is by definition the opposite of a *variable*—“constant” means “not able to vary”—any attempt to modify the value later in the code would result in a compile-time error.

<table>
<thead>
<tr>
<th>Guidelines</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DO NOT</strong> use a constant for any value that can possibly change over time. The value of pi and the number of protons in an atom of gold are constants; the price of gold, the name of your company, and the version number of your program can change.</td>
</tr>
</tbody>
</table>

Note that the expression assigned to `secondsPerWeek` is a constant expression because all the operands in the expression are also constants.

Listing 3.19: Declaring a Constant

```
pickup
```

**Introducing Flow Control**

Later in this chapter is a code listing (Listing 3.43) that shows a simple way to view a
number in its binary form. Even such a simple program, however, cannot be written without using control flow statements. Such statements control the execution path of the program. This section discusses how to change the order of statement execution based on conditional checks. Later on, you will learn how to execute statement groups repeatedly through loop constructs.

A summary of the control flow statements appears in Table 3.1. Note that the General Syntax Structure column indicates common statement use, not the complete lexical structure.

An embedded-statement in Table 3.1 may be any statement other than a labeled statement or a declaration, but it is typically a block statement.

Each C# control flow statement in Table 3.1 appears in the tic-tac-toe program and is available in Appendix B and for download with the rest of the source code listings from the book. The program displays the tic-tac-toe board, prompts each player, and updates with each move.

The remainder of this chapter looks at each statement in more detail. After covering the if statement, it introduces code blocks, scope, Boolean expressions, and bitwise operators before continuing with the remaining control flow statements. Readers who find the table familiar because of C#'s similarities to other languages can jump ahead to the section titled C# Preprocessor Directives or skip to the Summary section at the end of the chapter.

<table>
<thead>
<tr>
<th>Statement</th>
<th>General Syntax Structure</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>if statement</td>
<td>if(boolean-expression) &lt;br&gt; embedded-statement</td>
<td>if (input == &quot;quit&quot;)&lt;br&gt; {&lt;br&gt; System.Console.WriteLine(&lt;br&gt; &quot;Game end&quot;);&lt;br&gt; return; }</td>
</tr>
<tr>
<td></td>
<td>if(boolean-expression) &lt;br&gt; embedded-statement else &lt;br&gt; embedded-statement</td>
<td>if (input == &quot;quit&quot;)&lt;br&gt; {&lt;br&gt; System.Console.WriteLine(&lt;br&gt; &quot;Game end&quot;);&lt;br&gt; return; }&lt;br&gt; else&lt;br&gt; GetNextMove();</td>
</tr>
</tbody>
</table>

---

3. Known as noughts and crosses to readers outside the United States.
<table>
<thead>
<tr>
<th>Statement Type</th>
<th>Code Example</th>
</tr>
</thead>
</table>
| while statement | ```
while(boolean-expression)
    {
        embedded-statement
    }
``` |
| do while statement | ```
do
    {
        embedded-statement
        do
            while(boolean-expression);
    }
``` |
| for statement | ```
for(for-initializer;
    boolean-expression;
    for-iterator)
    {
        embedded-statement
    }
``` |
| foreach statement | ```
foreach(type identifier in expression)
    {
        embedded-statement
    }
``` |
| continue statement | ```
continue;
``` |
The remainder of this chapter looks at each statement in more detail. After covering the if statement, it introduces code blocks, scope, Boolean expressions, and bitwise operators before continuing with the remaining control flow statements. Readers who find the table familiar because of C#'s similarities to other languages can jump ahead to the section titled C# Preprocessor Directives or skip to the Summary section at the end of the chapter.

### if Statement

The if statement is one of the most common statements in C#. It evaluates a **Boolean expression** (an expression that results in either `true` or `false`) called the **condition**. If the condition is `true`, the **consequence statement** is executed. An if statement may optionally have an else clause that contains an **alternative statement** to be executed if the condition is `false`. The general form is as follows:

```
if (condition)
  consequence-statement
else
  alternative-statement
```

---

<table>
<thead>
<tr>
<th>switch statement</th>
<th>switch(governing-type-expression)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{ ... case const-expression:</td>
</tr>
<tr>
<td></td>
<td>statement-list jump-statement</td>
</tr>
<tr>
<td></td>
<td>default: statement-list jump-statement</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

| break statement  | break;                            |

| goto statement   | goto identifier;                  |
|                  | goto case const-expression;       |
|                  | goto default;                     |

```
switch(input)
{
  case "exit":
  case "quit":
    System.Console.WriteLine(  "Exiting app...." );
    break;
  case "restart":
    Reset();
    goto case "start";
  case "start":
    GetMove();
    break;
  default:
    System.Console.WriteLine(  input);  break;
}
```

---
In Listing 3.20, if the user enters 1, the program displays "Play against computer selected.". Otherwise, it displays "Play against another player.".

**Nested if**

Sometimes code requires multiple if statements. The code in Listing 3.21 first determines whether the user has chosen to exit by entering a number less than or equal to 0; if not, it checks whether the user knows the maximum number of turns in tic-tac-toe.

Listing 3.21: Nested if Statements

class TicTacToeTrivia
{
    static void Main()
    {
        int input; // Declare a variable to store the input.

        System.Console.Write(
            "What is the maximum number of turns in tic-tac-toe? (Enter 0 to exit): ");

        // int.Parse() converts the ReadLine() // return to an int data type.
```csharp
input = int.Parse(System.Console.ReadLine());

if (input <= 0)
    // Input is less than or equal to 0.
    System.Console.WriteLine("Exiting...");
else
    if (input < 9)
        // Input is less than 9.
        System.Console.WriteLine("Tic-tac-toe has more than {0}" + " maximum turns.", input);
    else
        if (input > 9)
            // Input is greater than 9.
            System.Console.WriteLine("Tic-tac-toe has fewer than {0}" + " maximum turns.", input);
        else
            // Input equals 9.
            System.Console.WriteLine("Correct, tic-tac-toe " + " has a max. of 9 turns.");
```
Although the latter format is more common, in each situation use the format that results in the clearest code.

Each if statement listing above omits the use of braces. However, as discussed next, this is not in accordance with the guidelines, which advocate the use of code blocks except, perhaps, in the simplest of single-line scenarios.

### Code Blocks ({})

In the previous if statement examples, only one statement follows if and else: a single System.Console.WriteLine(), similar to Listing 3.23.

#### Listing 3.23: if Statement with No Code Block

if(input < 9)
    System.Console.WriteLine(“Exiting”);

With curly braces, however, we can combine statements into a single statement called a **block statement** or **code block**, allowing the grouping of multiple statements into a single statement that is the consequence. Take, for example, the highlighted code block in the radius calculation in Listing 3.24.

#### Listing 3.24: if Statement Followed by a Code Block

```csharp
class CircleAreaCalculator
{
    static void Main()
    {
        double radius; // Declare a variable to store the radius.
        double area; // Declare a variable to store the area.
    }
}
```
System.Console.Write("Enter the radius of the circle: ");

// double.Parse converts the ReadLine()
// return to a double.
radius = double.Parse(System.Console.ReadLine());

if(radius>=0)
{
    // Calculate the area of the circle.
    area = 3.14*radius*radius;
    System.Console.WriteLine("The area of the circle is: {0}", area);
}
else
{
    System.Console.WriteLine("{0} is not a valid radius.", radius);
}

Output 3.14 shows the results of Listing 3.24.

In this example, the if statement checks whether the radius is positive. If so, the area of
the circle is calculated and displayed; otherwise, an invalid radius message is displayed.

Notice that in this example, two statements follow the first if. However, these two
statements appear within curly braces. The curly braces combine the statements into a
code block, which is itself a single statement.

If you omit the curly braces that create a code block in Listing 3.24, only the statement
immediately following the Boolean expression executes conditionally. Subsequent
statements will execute regardless of the if statement’s Boolean expression. The invalid
code is shown in Listing 3.25.

Listing 3.25: Relying on Indentation, Resulting in Invalid Code

if(radius>=0)
    area = 3.14 * radius *radius;
System.Console.WriteLine(
    "The area of the circle is: {0}", area);
In C#, indentation is for code readability only. The compiler ignores it, and therefore, the previous code is semantically equivalent to Listing 3.26.

Listing 3.26: Semantically Equivalent to Listing 3.25

```csharp
if (radius >= 0)
{
    area = 3.14 * radius * radius;
}
System.Console.WriteLine(
    "The area of the circle is: {0}", area);
```

Programmers should take great care to avoid subtle bugs such as this, perhaps even going so far as to always include a code block after a control flow statement, even if there is only one statement. In fact, the guideline is to avoid omitting braces, except possibly for the simplest of single-line if statements.

Although unusual, it is possible to have a code block that is not lexically a direct part of a control flow statement. In other words, placing curly braces on their own (without a conditional or loop, for example) is legal syntax.

<table>
<thead>
<tr>
<th>Guidelines</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVOID omitting braces, except for the simplest of single-line if statements.</td>
</tr>
</tbody>
</table>

**ADVANCED TOPIC**

Math Constants
In Listing 3.25 and Listing 3.26, the value of pi as 3.14 was hardcoded—a crude approximation at best. There are much more accurate definitions for pi and E in the System.Math class. Instead of hardcoding a value, code should use System.Math.PI and System.Math.E.

***Production, Advanced Topic ends here***

Code Blocks, Scopes, and Declaration Spaces
Code blocks are often referred to as “scopes,” but the two terms are not exactly inter-
The scope of a named thing is the region of source code in which it is legal to refer to the thing by its unqualified name. The scope of a local variable is exactly the text of the code block that encloses it, which explains why it is common to refer to code blocks as “scopes.”

Scopes are often confused with declaration spaces. A declaration space is a logical container of named things in which two things may not have the same name. A code block not only defines a scope, it also defines a local variable declaration space; it is illegal for two local variable declarations with the same name to appear in the same declaration space. Similarly, it is not possible to declare two methods with the signature of `Main()` within the same class. (Though the rule is relaxed somewhat for methods; two methods may have the same name in a declaration space provided that they have different signatures.) A code block not only defines a scope, it also defines a local variable declaration space. That is to say, within a block a local can be mentioned by name and must be the unique thing that is declared with that name in the block. Outside the declaring block there is no way to refer to a local by its name; the local is said to be “out of scope” outside the block.

In short: A scope is used to determine what thing a name refers to; a declaration space determines when two things declared with the same name conflict with each other. In Listing 3.27, declaring the local variable `message` inside the block statement embedded in the `if` statement restricts its scope to the block statement only; the local is “out of scope” when its name is used later on in the method. To avoid the error, you must declare the variable outside the `if` statement.

Listing 3.27: Variables Inaccessible outside Their Scope

```csharp
class Program
{
    static void Main(string[] args)
    {
        int playerCount;
        System.Console.Write(
            "Enter the number of players (1 or 2):\n";
        playerCount = int.Parse(System.Console.ReadLine());
        if (playerCount != 1 && playerCount != 2)
        {
            string message = 
                "You entered an invalid number of players.\n"
        }
        else
        {
            // ...
        }
        // Error: message is not in scope.
        System.Console.WriteLine(message);
    }
}
```
Output 3.15 shows the results of Listing 3.27.

**Output 3.15**

```text
...
...
...
...
...

__Program.cs(18,26): error CS0103: The name 'message' does not exist in the current context__
```

The declaration space throughout which a local’s name must be unique includes all the child code blocks textually enclosed within the block that originally declared the local. The C# compiler prevents the name of a local variable declared immediately within a method code block (or as a parameter) from being reused within a child code block. In Listing 3.27, because `args` and `playerCount` are declared within the method code block, they cannot be declared again anywhere within the method.

The name `message` refers to this local variable throughout the scope of the local variable: that is, the block immediately enclosing the declaration. Similarly, `playerCount` refers to the same variable throughout the block containing the declaration, including within both of the child blocks that are the consequence and alternative of the `if` statement.

---

**Language Contrast: C++—Local Variable Scope**

In C++, a local variable declared in a block is in scope from the point of the declaration statement through the end of the block; an attempt to refer to the local variable before its declaration will fail to find the local because the local is not in scope. If there is another thing with that name “in scope,” the C++ language will resolve the name to that thing, which might not be what you intended. In C#, the rule is subtly different; a local is in scope throughout the entire block in which it is declared, but it is illegal to refer to the local before its declaration. That is, the attempt to find the local succeeds and the usage is then treated as an error. This is just one of C#'s many rules that attempt to prevent errors common in C++ programs.
Boolean Expressions

The parenthesized condition of the if statement is a **Boolean expression**. In Listing 3.28, the condition is highlighted.

Listing 3.28: Boolean Expression

```csharp
if(input < 9)
{
   // Input is Less than 9.
   System.Console.WriteLine("Tic-tac-toe has more than {0}" +
   " maximum turns.", input);
}
```

Boolean expressions appear within many control flow statements. Their key characteristic is that they always evaluate to true or false. For input<9 to be allowed as a Boolean expression, it must result in a `bool`. The compiler disallows `x=42`, for example, because it assigns `x` and results in the value that was assigned, instead of checking whether the value of the variable is 42.

**Language Contrast: C++—Mistakenly Using `=` in Place of `==`**

C# eliminates a coding error common in C and C++. In C++, Listing 3.29 is allowed.

Listing 3.29: C++, But Not C#, Allows Assignment As a Condition

```csharp
if(input=9)   // Allowed in C++, not in C#.
    System.Console.WriteLine("Correct, tic-tac-toe has a maximum of 9 turns.");
```

Although at a glance this appears to check whether `input` equals 9, Chapter 1 showed that `=` represents the assignment operator, not a check for equality. The return from the assignment operator is the value assigned to the variable—in this case, 9. However, 9 is an `int`, and as such it does
not qualify as a Boolean expression and is not allowed by the C# compiler. The C and C++ languages treat integers that are nonzero as true, and integers that are zero as false. C#, by contrast, requires that the condition actually be of a Boolean type; integers are not allowed.

**Relational and Equality Operators**

Relational and equality operators determine whether a value is greater than, less than, or equal to another value. Table 3.2 lists all the relational and equality operators. All are binary operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>input&lt;9;</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>input&gt;9;</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
<td>input&lt;=9;</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
<td>input&gt;=9;</td>
</tr>
<tr>
<td>==</td>
<td>Equality operator</td>
<td>input==9;</td>
</tr>
<tr>
<td>!=</td>
<td>Inequality operator</td>
<td>input!=9;</td>
</tr>
</tbody>
</table>

The C# syntax for equality uses `==`, just as many other programming languages do. For example, to determine whether `input` equals 9 you use `input==9`. The equality operator uses two equal signs to distinguish it from the assignment operator, `=`. The exclamation point signifies NOT in C#, so to test for inequality you use the inequality operator, `!=`.

Relational and equality operators always produce a `bool` value, as shown in Listing 3.30.
In the tic-tac-toe program (see Appendix B), you use the equality operator to determine whether a user has quit. The Boolean expression of Listing 3.31 includes an OR (||) logical operator, which the next section discusses in detail.

**Logical Boolean Operators**

The logical operators have Boolean operands and produce a Boolean result. Logical operators allow you to combine multiple Boolean expressions to form more complex Boolean expressions. The logical operators are |, ||, &, &&, and ^, corresponding to OR, AND, and exclusive OR. The | and & versions of OR and AND are only rarely used for Boolean logic, for reasons which we discuss below.

**OR Operator (||)**

In Listing 3.31, if the user enters quit or presses the Enter key without typing in a value, it is assumed that she wants to exit the program. To enable two ways for the user to resign, you use the logical OR operator, ||.

The || operator evaluates Boolean expressions and results in a true value if either operand is true (see Listing 3.32).

It is not necessary to evaluate both sides of an OR expression because if either operand is true, the result is known to be true regardless of the value of the other operand. Like all operators in C#, the left operand is evaluated before the right one, so if the left portion of the expression evaluates to true, the right portion is ignored. In the example above, if hourOfDay has the value 33, (hourOfDay > 23) will evaluate to true and the OR operator will ignore the second half of the expression, short-circuiting it.
circuiting an expression also occurs with the Boolean AND operator. (Note that the parentheses are not necessary here; the logical operators are of higher precedence than the relational operators. However, it is clearer to the novice reader to parenthesize the subexpressions for clarity.)

*** Please ensure short-circuiting ends up in the index ***

AND Operator (&&)
The Boolean AND operator, &&, evaluates to true only if both operands evaluate to true. If either operand is false, the result will be false. Listing 3.33 writes a message if the given variable is both greater than 10 and less than 24. Similarly to the OR operator, the AND operator will not always evaluate the right side of the expression. If the left operand is determined to be false, the overall result will be false regardless of the value of the right operand, so the runtime skips evaluating the right operand.

Listing 3.33: Using the AND Operator

```csharp
if ((10 < hourOfDay) && (hourOfDay < 24))
    System.Console.WriteLine("Hi-Ho, Hi-Ho, it's off to work we go.");
```

Exclusive OR Operator (^)
The caret symbol, ^, is the “exclusive OR” (XOR) operator. When applied to two Boolean operands, the XOR operator returns true only if exactly one of the operands is true, as shown in Table 3.3.

Table 3.3: Conditional Values for the XOR Operator

<table>
<thead>
<tr>
<th>Left Operand</th>
<th>Right Operand</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

Unlike the Boolean AND and Boolean OR operators, the Boolean XOR operator does

---

4. The typical hours that programmers work.
not short-circuit: It always checks both operands, because the result cannot be determined unless the values of both operands are known. Note that the XOR operator is exactly the same as the Boolean inequality operator.

**Logical Negation Operator (!)**

The logical negation operator, or **NOT operator**, !, inverts a `bool` value. This operator is a unary operator, meaning it requires only one operand. Listing 3.34 demonstrates how it works, and Output 3.16 shows the results.

**Listing 3.34: Using the Logical Negation Operator**

```csharp
bool valid = false;
bool result = !valid;
// Displays "result = True".
System.Console.WriteLine("result = {0}", result);
```

**Output 3.16**

```
result= True
```

To begin, `valid` is set to `false`. You then use the negation operator on `valid` and assign the value to `result`.

**Conditional Operator (?:)**

In place of an `if-else` statement used to select one of two values, you can use the **conditional operator**. The conditional operator uses both a question mark and a colon; the general format is as follows:

```
c condition ? consequence : alternative
```

The conditional operator is a “ternary” operator because it has three operands: `condition`, `consequence`, and `alternative`. (As it is the only ternary operator in C#, it is often called “the ternary operator,” but it is clearer to refer to it by its name than by the number of operands it takes.) Like the logical operators, the conditional operator uses a form of short-circuiting. If the condition evaluates to `true`, the conditional operator evaluates only `consequence`. If the conditional evaluates to `false`, it evaluates only `alternative`. The result of the operator is the evaluated expression.

Listing 3.35 is an example of how to use the conditional operator. The full listing of this program appears in Appendix B.
The program swaps the current player. To do this, it checks whether the current value is 2. This is the conditional portion of the conditional expression. If the result of the condition is true, the conditional operator results in the “consequence” value 1. Otherwise, it results in the “alternative” value 2. Unlike an if statement, the result of the conditional operator must be assigned (or passed as a parameter). It cannot appear as an entire statement on its own.

Guidelines

CONSIDER using an if/else statement instead of an overly complicated conditional expression.

The C# language requires that the consequence and alternative expressions in a conditional operator be consistently typed, and that the consistent type be determined without examination of the surrounding context of the expression. For example, \( f ? \text{"abc" : 123} \) is not a legal conditional expression because the consequence and alternative are a string and a number, neither of which is convertible to the other. Even if you say object result = \( f ? \text{"abc" : 123} \); the C# compiler will still flag this expression as illegal because the type that is consistent with both expressions (that is, object) is found
outside the conditional expression.

**Null Coalescing Operator (??)**

The null coalescing operator is a concise way to express “if this value is null then use this other value.” Its form is:

```
expression1 ?? expression2;
```

The null coalescing operator also uses a form of short-circuiting. If `expression1` is not null, its value is the result of the operation and the other expression is not evaluated. If `expression1` does evaluate to null, the value of `expression2` is the result of the operator. Unlike the conditional operator, the null coalescing operator is a binary operator.

Listing 3.36 is an example of how to use the null coalescing operator.

Listing 3.36: Null Coalescing Operator

```c
string fileName = GetFileName();
// ...
string fullName = fileName ?? "default.txt";
// ...
```

In this listing, we use the null coalescing operator to set `fullName` to "default.txt" if `fileName` is null. If `fileName` is not null, `fullName` is simply assigned the value of `fileName`.

The coalescing operator “chains” nicely; an expression of the form `x ?? y ?? z` results in `x` if `x` is not null; otherwise, it results in `y` if `y` is not null; otherwise, it results in `z`. That is, it goes from left to right and picks out the first non-null expression, or uses the last expression if all the previous expressions were null.

The null coalescing operator was added to C# in version 2.0 along with nullable value types; the null coalescing operator works on both operands of nullable value types and reference types.

**Bitwise Operators (<<, >>, |, &, ^, ~)**

An additional set of operators that is common to virtually all programming languages is the set of operators for manipulating values in their binary formats: the bit operators.

---

**BEGINNER TOPIC**

**Bits and Bytes**

All values within a computer are represented in a binary format of 1s and 0s, called bi-
Binary digits (bits). Bits are grouped together in sets of eight, called bytes. In a byte, each successive bit corresponds to a value of 2 raised to a power, starting from $2^0$ on the right, to $2^7$ on the left, as shown in Figure 3.1.

Figure 3.1: Corresponding Placeholder Values

In many scenarios, particularly when dealing with low-level or system services, information is retrieved as binary data. In order to manipulate these devices and services, you need to perform manipulations of binary data.

As shown in Figure 3.2, each box corresponds to a value of 2 raised to the power shown. The value of the byte (8-bit number) is the sum of the powers of 2 of all of the eight bits that are set to 1.

Figure 3.2: Calculating the Value of an Unsigned Byte

The binary translation just described is significantly different for signed numbers. Signed numbers (long, short, int) are represented using a “twos complement” notation. This is so that addition continues to work when adding a negative number to a positive number as though both were positive operands. With this notation, negative numbers behave differently than positive numbers. Negative numbers are identified by a 1 in the leftmost location. If the leftmost location contains a 1, you add the locations with 0s rather than the locations with 1s. Each location corresponds to the negative power of 2 value. Furthermore, from the result, it is also necessary to subtract 1. This is demonstrated in Figure 3.3.

Figure 3.3: Calculating the Value of a Signed Byte

Therefore, 1111 1111 1111 1111 corresponds to $-1$ and 1111 1111 1111 1001 holds the value $-7$. 1000 0000 0000 0000 corresponds to the lowest negative value that a 16-bit integer can hold.

***Production, Beginner Topic ends here***
**Shift Operators (<<, >>, <<=, >>=)**

Sometimes you want to shift the binary value of a number to the right or left. In executing a left shift, all bits in a number’s binary representation are shifted to the left by the number of locations specified by the operand on the right of the shift operator. Zeroes are then used to backfill the locations on the right side of the binary number. A right-shift operator does almost the same thing in the opposite direction. However, if the number is a negative value of a signed type, the values used to backfill the left side of the binary number are ones and not zeroes. The shift operators are `>>` and `<<`, the right-shift and left-shift operators, respectively. In addition, there are combined shift and assignment operators, `<<=` and `>>=`.

Consider the following example. Suppose you had the `int` value `-7`, which would have a binary representation of `1111 1111 1111 1111 1111 1111 1111 1001`. In Listing 3.37, you right-shift the binary representation of the number `-7` by two locations.

**Listing 3.37: Using the Right-Shift Operator**

```csharp
int x;
x = (-7 >> 2); // 11111111111111111111111111111001 becomes  
    // 11111111111111111111111111111110
    // Write out "x is -2."
System.Console.WriteLine("x = {0}.", x);
```

**Output 3.17** shows the results of Listing 3.37.

**Output 3.17**

```
x=-2
```

Because of the right shift, the value of the bit in the rightmost location has “dropped off” the edge and the negative bit indicator on the left shifts by two locations to be replaced with 1s. The result is `-2`.

Although legend has it that `x << 2` is faster than `x * 4`, do not use bit shift operators for multiplication or division. This might have been true in certain C compilers in the 1970s, but modern compilers and modern microprocessors are perfectly capable of optimizing arithmetic. Using shifting for multiplication or division is confusing and frequently leads to errors when code maintainers forget that the shift operators are lower precedence than the arithmetic operators.
**Bitwise Operators (&, |, ^)**

In some instances, you might need to perform logical operations, such as AND, OR, and XOR, on a bit-by-bit basis for two operands. You do this via the & |, and ^ operators, respectively.

---

**BEGINNER TOPIC**

**Logical Operators Explained**

If you have two numbers, as shown in Figure 3.4, the bitwise operations will compare the values of the locations beginning at the leftmost significant value and continuing right until the end. The value of “1” in a location is treated as “true,” and the value of “0” in a location is treated as “false.”

Figure 3.4: The Numbers 12 and 7 Represented in Binary

Therefore, the bitwise AND of the two values in Figure 3.4 would be the bit-by-bit comparison of bits in the first operand (12) with the bits in the second operand (7), resulting in the binary value 000000100, which is 4. Alternatively, a bitwise OR of the two values would produce 00001111, the binary equivalent of 15. The XOR result would be 00001011, or decimal 11.

***Production, Beginner Topic ends here***

Listing 3.38 demonstrates how to use these bitwise operators. The results of Listing 3.38 appear in Output 3.18.

Listing 3.38: Using Bitwise Operators

```csharp
using System;

class Program
{
    static void Main()
    {
        byte and, or, xor;
        and = 12 & 7; // and = 4
        or = 12 | 7;  // or = 15
        xor = 12 ^ 7; // xor = 11
        System.Console.WriteLine(
            "and = {0} \nor = {1}\nxor = {2}",
            and, or, xor);
    }
}
```

**OUTPUT 3.18**

and=4
or=15
xor=11
In Listing 3.38, the value 7 is the **mask**; it is used to expose or eliminate specific bits within the first operand using the particular operator expression. Note that, unlike the AND (&&) operator, the & operator always evaluates *both* sides even if the left portion is false. Similarly, the | version of the OR operator is *not* “short-circuiting.” It always evaluates both operands even if the left operand is true. The bit versions of the AND and OR operators, therefore, are not short-circuiting.

In order to convert a number to its binary representation, you need to iterate across each bit in a number. Listing 3.39 is an example of a program that converts an integer to a string of its binary representation. The results of Listing 3.39 appear in Output 3.19.

Listing 3.39: Getting a String Representation of a Binary Display

```csharp
public class BinaryConverter
{
    public static void Main()
    {
        const int size = 64;
        ulong value;
        char bit;

        System.Console.Write("Enter an integer:");
        // Use long.Parse() so as to support negative numbers
        // Assumes unchecked assignment to ulong.
        value = (ulong)long.Parse(System.Console.ReadLine());

        // Set initial mask to 100....
        ulong mask = 1UL << size - 1;
        for (int count = 0; count < size; count++)
        {
            bit = ((mask & value) != 0) ? '1': '0';
            System.Console.Write(bit);
            // Shift mask one location over to the right
            mask >>= 1;
        }
        System.Console.WriteLine();
    }
}
```

**Output 3.19**

```
Enter an integer: 42
0000000000000000000000000000000000000000000000000000000000101010
```

Notice that within each iteration of the for loop (discussed later in this chapter), you use the right-shift assignment operator to create a mask corresponding to each bit position in value. By using the & bit operator to mask a particular bit, you can determine whether the bit is set. If the mask test produces a nonzero result, you write 1 to the console; oth-
erwise, \( \theta \) is written. In this way, you create output describing the binary value of an unsigned long.

Note also that the parentheses in \((\text{mask} \& \text{value}) \neq \theta\) are necessary because inequality is higher precedence than the AND operator; without the explicit parentheses this would be equivalent to \(\text{mask} \& (\text{value} \neq \theta)\), which does not make any sense; the left side of the \& is a ulong and the right side is a bool.

**Bitwise Compound Assignment Operators (\&=, |=, ^=)**

Not surprisingly, you can combine these bitwise operators with assignment operators as follows: \&=, |=, and ^=. As a result, you could take a variable, OR it with a number, and assign the result back to the original variable, which Listing 3.40 demonstrates.

**Listing 3.40: Using Logical Assignment Operators**

```csharp
byte and = 12, or = 12, xor = 12;
and &= 7;  // and = 4
or |= 7;   // or = 15
xor ^= 7;  // xor = 11
System.Console.WriteLine(
    "and = \{0\}\nor = \{1\}\nxor = \{2\}",
    and, or, xor);
```

The results of Listing 3.40 appear in Output 3.20.

**OUTPUT 3.20**

```
and=4
or=15
xor=11
```

Combining a bitmap with a mask using something like \(\text{fields} \&= \text{mask}\) clears the bits in \(\text{fields}\) that are not set in the \(\text{mask}\). The opposite, \(\text{fields} \&= \sim\text{mask}\), clears out the bits in \(\text{fields}\) that are set in \(\text{mask}\).

**Bitwise Complement Operator (\~)**

The **bitwise complement operator** takes the complement of each bit in the operand, where the operand can be an int, uint, long, or ulong. \(~1\), therefore, returns the value with binary notation 1111 1111 1111 1111 1111 1111 1111 1110, and \(~(1<<31)\) returns the number with binary notation 0111 1111 1111 1111 1111 1111 1111 1111.
Control Flow Statements, Continued

Now that we’ve described Boolean expressions in more detail we can more clearly describe the control flow statements supported by C#. Many of these statements will be familiar to experienced programmers, so you can skim this section looking for details specific to C#. Note in particular the `foreach` loop, as this may be new to many programmers.

The `while` and `do/while` Loops

Thus far you have learned how to write programs that do something only once. However, computers can easily perform similar operations multiple times. In order to do this, you need to create an instruction loop. The first instruction loop we will discuss is the `while` loop, because it is the simplest conditional loop. The general form of the `while` statement is as follows:

```
while (condition)
  statement
```

The computer will repeatedly execute the statement that is the “body” of the loop as long as the condition (which must be a Boolean expression) evaluates to `true`. If the condition evaluates to `false`, code execution skips the body and executes the code following the loop statement. Note that `statement` will continue to execute even if it causes the condition to become `false`. It isn’t until the condition is reevaluated “at the top of the loop” that the loop exits. The Fibonacci calculator shown in Listing 3.41 demonstrates the `while` loop.

Listing 3.41: while Loop Example

```csharp
class FibonacciCalculator
{
    static void Main()
    {
        decimal current;
        decimal previous;
        decimal temp;
        decimal input;

        System.Console.Write("Enter a positive integer:");

        // decimal.Parse convert the ReadLine to a decimal.
        input = decimal.Parse(System.Console.ReadLine());

        // Initialize current and previous to 1, the first
        // two numbers in the Fibonacci series.
        current = previous = 1;

        // While the current Fibonacci number in the series is
```
A Fibonacci number is a member of the Fibonacci series, which includes all numbers that are the sum of the previous two numbers in the series, beginning with 1 and 1. In Listing 3.41, you prompt the user for an integer. Then you use a while loop to find the first Fibonacci number that is greater than the number the user entered.

BEGINNER TOPIC

When to Use a while Loop

The remainder of this chapter considers other statements that cause a block of code to execute repeatedly. The term loop body refers to the statement (frequently a code block) that is to be executed within the while statement, since the code is executed in a “loop” until the exit condition is achieved. It is important to understand which loop construct to select. You use a while construct to iterate while the condition evaluates to true. A for loop is used most appropriately whenever the number of repetitions is known, such as counting from 0 to n. A do/while is similar to a while loop, except that it will always execute the loop body at least once.

***Production, Beginner Topic ends here***

The do/while loop is very similar to the while loop except that a do/while loop is preferred when the number of repetitions is from 1 to n and n is not known when iterating begins. This pattern frequently occurs when prompting a user for input. Listing 3.42 is taken from the tic-tac-toe program.

Listing 3.42: do/while Loop Example

```csharp
// Repeatedly request player to move until he
// enter a valid position on the board.
bool valid;
```
do
{
  valid = false;

  // Request a move from the current player.
  System.Console.Write("Player \{0\}: Enter move:", currentPlayer);
  input = System.Console.ReadLine();

  // Check the current player's input.
  // ...
}

while (!valid);

In Listing 3.42, you always initialize valid to false at the beginning of each iteration, or loop repetition. Next, you prompt and retrieve the number the user input. Although not shown here, you then check whether the input was correct, and if it was, you assign valid equal to true. Since the code uses a do/while statement rather than a while statement, the user will be prompted for input at least once.

The general form of the do/while loop is as follows:

do
  statement
while (condition);

As with all the control flow statements, a code block is generally used as the single statement in order to allow multiple statements to be executed as the loop body. However, any single statement except for a labeled statement or a local variable declaration can be used.

The for Loop

The for loop iterates a code block until a specified condition is reached. In that way, it is very similar to the while loop. The difference is that the for loop has built-in syntax for initializing, incrementing, and testing the value of a counter, known as the loop variable. Because there is a specific location in the loop syntax for an increment operation, the increment and decrement operators are frequently used as part of a for loop.

Listing 3.43 shows the for loop used to display an integer in binary form. The results of this listing appear in Output 3.21.

Listing 3.43: Using the for Loop

public class BinaryConverter
{
  public static void Main()
  {
    const int size = 64;
ulong value;
char bit;

System.Console.Write("Enter an integer: ");
// Use Long.Parse() so as to support negative numbers
// Assumes unchecked assignment to ulong.
value = (ulong)long.Parse(System.Console.ReadLine());

// Set initial mask to 100....
ulong mask = 1UL << size - 1;
for (int count = 0; count < size; count++)
{
    bit = ((mask & value) > 0) ? '1': '0';
    System.Console.Write(bit);
    // Shift mask one location over to the right
    mask >>= 1;
}

Listing 3.43 performs a bit mask 64 times, once for each bit in the number. The three parts of the for loop header first declare and initialize the variable count, then describe the condition that must be met for the loop body to be executed, and finally describe the operation that updates the loop variable. The general form of the for loop is as follows:

    for (initial ; condition ; loop)
        statement

Here is a breakdown of the for loop.
• The initial section performs operations that precede the first iteration. In Listing 3.43, it declares and initializes the variable count. The initial expression does not have to be a declaration of a new variable (though it frequently is). It is possible, for example, to declare the variable beforehand and simply initialize it in the for loop, or to skip the initialization section entirely by leaving it blank. Variables declared here are in scope throughout the header and body of the for statement.

• The condition portion of the for loop specifies an end condition. The loop exits when this condition is false exactly like the while loop does. The for loop will execute the body only as long as the condition evaluates to true. In the preceding example, the loop exits when count is greater than or equal to 64.

• The loop expression executes after each iteration. In the preceding example, count++ executes after the right shift of the mask (mask >>= 1), but before the condition is evaluated. During the sixty-fourth iteration, count is incremented to 64, causing the condition to become false, and therefore terminating the loop.

• The statement portion of the for loop is the “loop body” code that executes while the conditional expression remains true.

If you wrote out each for loop execution step in pseudocode without using a for loop expression, it would look like this:

1. Declare and initialize count to 0.

2. If count is less than 64, continue to step 3; otherwise, go to step 7.

3. Calculate bit and display it.

4. Shift the mask.

5. Increment count by one.

6. Jump back to line 2.

7. Continue the execution of the program after the loop.

The for statement doesn’t require any of the elements in its header. for(;;){ ... } is perfectly valid; although there still needs to be a means to escape from the loop to avoid
executing infinitely. (If the condition is missing, it is assumed to be the constant \texttt{true}.)

The initial and loop expressions have an unusual syntax to support loops that require
multiple loop variables, as shown in Listing 3.44.

Listing 3.44: \texttt{for} Loop Using Multiple Expressions

\begin{verbatim}
for(int x=0, y=5; ((x<=5) && (y>=0)); y--, x++)
{
    System.Console.Write("{0}{1}{2}
    \t",
    x, (x>y? '>': '<'), y);
}
\end{verbatim}

The results of Listing 3.44 appear in Output 3.22.

\begin{verbatim}
0<5 1<4 2<3 3>2 4>1 5>0
\end{verbatim}

Here the initialization clause contains a complex declaration that declares and initializes
two loop variables, but this is at least similar to a declaration statement that declares
multiple local variables. The loop clause is quite unusual, as it can consist of a comma-
separated list of expressions, not just a single expression.

\begin{Verbatim}
Guidelines

CONSIDER refactoring the method to make the control flow easier to understand if you find yourself writing \texttt{for} loops with complex conditionals and multiple loop variables.
\end{Verbatim}

The \texttt{for} loop is little more than a more convenient way to write a \texttt{while} loop; you can always rewrite a \texttt{for} loop like this:

```
// initial;
while(condition)
{
    statement;
    loop;
```
DO use the for loop when the number of loop iterations is known in advance and the "counter" that gives the number of iterations executed is needed in the loop.
DO use the while loop when the number of loop iterations is not known in advance and a counter is not needed.

The foreach Loop

The last loop statement in the C# language is foreach. The foreach loop iterates through a collection of items, setting a loop variable to represent each item in turn. In the body of the loop, operations may be performed on the item. A nice property of the foreach loop is that every item is iterated over exactly once; it is not possible to accidentally miscount and iterate past the end of the collection as can happen with other loops.

The general form of the foreach statement is as follows:

    foreach(type variable in collection)
    statement

Here is a breakdown of the foreach statement.

- type is used to declare the data type of the variable for each item within the collection. It may be var, in which case the compiler infers the type of the item from the type of the collection.
- variable is a read-only variable into which the foreach loop will automatically assign the next item within the collection. The scope of the variable is limited to the body of the loop.
- collection is an expression, such as an array, representing any number of items.
- statement is the loop body that executes for each iteration of the loop.

Consider the foreach loop in the context of the simple example shown in Listing 3.45.
class TicTacToe  // Declares the TicTacToe class.
{
    static void Main() // Declares the entry point of the program.
    {
        // Hardcode initial board as follows
        // 1 | 2 | 3
        // 4 | 5 | 6
        // 7 | 8 | 9
        char[] cells = {
            '1', '2', '3', '4', '5', '6', '7', '8', '9'
        };

        System.Console.Write(
            "The available moves are as follows: ");

        // Write out the initial available moves
        foreach (char cell in cells)
        {
            if (cell != 'O' && cell != 'X')
            {
                System.Console.Write("{0} ", cell);
            }
        }
    }
}

Output 3.23 shows the results of Listing 3.45.

OUTPUT 3.23

The available moves are as follows: 1 2 3 4 5 6 7 8 9

When the execution engine reaches the foreach statement, it assigns to the variable cell the first item in the cells array—in this case, the value '1'. It then executes the code within the block that makes up the foreach loop body. The if statement determines whether the value of cell is 'O' or 'X'. If it is neither, the value of cell is written out to the console. The next iteration then assigns the next array value to cell, and so on.

It is important to note that the compiler prevents modification of the variable (cell) during the execution of a foreach loop. Also, the loop variable has a subtly different behavior in C# 5 than it did in previous versions; the difference is only apparent when the loop body contains a lambda expression or anonymous method that uses the loop varia-
BEGINNER TOPIC

Where the switch Statement Is More Appropriate

Sometimes you might compare the same value in several continuous if statements, as shown with the input variable in Listing 3.46.

Listing 3.46: Checking the Player’s Input with an if Statement

```
bool valid = false;

// Check the current player's input.
if ( input == "1" ) ||
( input == "2" ) ||
( input == "3" ) ||
( input == "4" ) ||
( input == "5" ) ||
( input == "6" ) ||
( input == "7" ) ||
( input == "8" ) ||
( input == "9" ) )
{
    // Save/move as the player directed.
    valid = true;
}
else if ( input == "" ) || ( input == "quit" )
{
    valid = true;
}
else
{
    System.Console.WriteLine(
        "\nERROR: Enter a value from 1-9. "
        + "Push ENTER to quit");
}
```

This code validates the text entered to ensure that it is a valid tic-tac-toe move. If the value of input were 9, for example, the program would have to perform nine different evaluations. It would be preferable to jump to the correct code after only one evaluation. To enable this, you use a switch statement.
**The switch Statement**

A switch statement is simpler to understand than a complex if statement when you have a value that must be compared against may different constant values. The switch statement looks like this:

```
switch(expression)
{
    case constant:
        statements
    default:
        statements
}
```

Here is a breakdown of the switch statement.

- **expression** is the value that is being compared against the different constants. The type of this expression determines the “governing type” of the switch. Allowable governing data types are bool, sbyte, byte, short, ushort, int, uint, long, ulong, char, any enum type (covered in Chapter 8), the corresponding nullable types of each of those value types, and string.

- **constant** is any constant expression compatible with the governing type.

- A group of one or more case labels (or the default label) followed by a group of one or more statements is called a **switch section**. The pattern above has two switch sections; Listing 3.47 shows a switch statement with three switch sections.

- **statements** is one or more statements to be executed when the expression equals one of the constant values mentioned in a label in the switch section. The end point of the group of statements must not be reachable. Typically the last statement is a jump statement such as a break, return, or goto statement.

---

**Guidelines**

DO NOT use continue as the jump statement that exits a switch section. This is legal when the switch is inside a loop, but it is easy to become confused about the meaning of break in a later switch section.
A `switch` statement should have at least one switch section; `switch(x){}` is legal but will generate a warning. Also, earlier the guideline was to avoid omitting braces in general. One exception is to omit braces for `case` and `break` statements because they serve to indicate the beginning and end of a block.

Listing 3.47, with a `switch` statement, is semantically equivalent to the series of `if` statements in Listing 3.46.

Listing 3.47: Replacing the `if` Statement with a `switch` Statement

```csharp
static bool ValidateAndMove(
    int[] playerPositions, int currentPlayer, string input)
{
    bool valid = false;

    // Check the current player’s input.
    switch (input)
    {
    case "1":
    case "2":
    case "3":
    case "4":
    case "5":
    case "6":
    case "7":
    case "8":
    case "9":
        // Save/move as the player directed.
        ...
        valid = true;
        break;
    case "":
    case "quit":
        valid = true;
        break;
    default:
        // If none of the other case statements
        // is encountered then the text is invalid.
        System.Console.WriteLine(
            "ERROR: Enter a value from 1-9."
            + " Push ENTER to quit");
        break;
    }

    return valid;
}
```

In Listing 3.47, `input` is the test expression. Since `input` is a string, the governing type is `string`. If the value of `input` is one of the strings 1, 2, 3, 4, 5, 6, 7, 8, or 9, the move is valid and you change the appropriate cell to match that of the current user’s token (X or O).
Once execution encounters a break statement, control leaves the switch statement.

The next switch section describes how to handle the empty string or the string quit; it sets valid to true if input equals either value. The default switch section is executed if no other switch section had a case label that matched the test expression.

---

**Language Contrast: C++—switch Statement Fall-Through**

In C++, if a switch section does not end with a jump statement, control “falls through” to the next switch section, executing its code. Because unintended fall-through is a common error in C++, C# does not allow control to accidentally fall through from one switch section to the next. The C# designers believed it was better to prevent this common source of bugs and encourage better code readability than to match the potentially confusing C++ behavior. If you do want one switch section to execute the statements of another switch section, you may do so explicitly with a goto statement, as demonstrated later in this chapter.

---

There are several things to note about the switch statement.

- A switch statement with no switch sections will generate a compiler warning, but the statement will still compile.
- Switch sections can appear in any order; the default section does not have to appear last. In fact, the default switch section does not have to appear at all; it is optional.
- The C# language requires that the end point of every switch section, including the last section, be unreachable. This means that switch sections usually end with a break, return, or goto.

**Jump Statements**

It is possible to alter the execution path of a loop. In fact, with jump statements, it is possible to escape out of the loop or to skip the remaining portion of an iteration and begin with the next iteration, even when the loop condition remains true. This section consid-
ers some of the ways to jump the execution path from one location to another.

The break Statement

To escape out of a loop or a switch statement, C# uses a break statement. Whenever the break statement is encountered, control immediately leaves the loop or switch. Listing 3.48 examines the foreach loop from the tic-tac-toe program.

Listing 3.48: Using break to Escape Once a Winner Is Found

class TicTacToe  // Declares the TicTacToe class.
{
    static void Main() // Declares the entry point of the program.
    {
        int winner=0;
        // Stores locations each player has moved.
        int[] playerPositions = {0,0};

        // Hardcoded board position
        // X | 2 | 0
        // 2 | 0 | 6
        // 6 | X | X
        playerPositions[0] = 449;
        playerPositions[1] = 28;

        // Determine if there is a winner
        int[] winningMasks = {
            7, 56, 448, 73, 146, 292, 84, 273
        };

        // Iterate through each winning mask to determine
        // if there is a winner.
        foreach (int mask in winningMasks)
        {
            if ((mask & playerPositions[0]) == mask)
            {
                winner = 1;
                break;
            }
            else if ((mask & playerPositions[1]) == mask)
            {
                winner = 2;
                break;
            }
        }

        System.Console.WriteLine(
            "Player {0} was the winner", winner);
    }
}
Output 3.24 shows the results of Listing 3.48.

**OUTPUT 3.24**

Player 1 was the winner

Listing 3.48 uses a break statement when a player holds a winning position. The break statement forces its enclosing loop (or a switch statement) to cease execution, and control moves to the next line outside the loop. For this listing, if the bit comparison returns true (if the board holds a winning position), the break statement causes control to jump and display the winner.

**BEGINNER TOPIC**

**Bitwise Operators for Positions**

The tic-tac-toe example (the full listing is available in Appendix B) uses the bitwise operators to determine which player wins the game. First, the code saves the positions of each player into a bitmap called playerPositions. (It uses an array so that the positions for both players can be saved.)

To begin, both playerPositions are 0. As each player moves, the bit corresponding to the move is set. If, for example, the player selects cell 3, shifter is set to 3 – 1. The code subtracts 1 because C# is zero-based and you need to adjust for 0 as the first position instead of 1. Next, the code sets position, the bit corresponding to cell 3, using the shift operator \(0000000000000001 \ll \text{shifter}\), where shifter now has a value of 2. Lastly, it sets playerPositions for the current player (subtracting 1 again to shift to zero-based) to \(00000000000000100\). Listing 3.49 uses |\= so that previous moves are combined with the current move.

**Listing 3.49: Setting the Bit That Corresponds to Each Player’s Move**

```csharp
int shifter; // The number of places to shift
            // over in order to set a bit.

int position; // The bit which is to be set.

// int.Parse() converts "input" to an integer.
// "int.Parse(input) - 1" because arrays
// are zero-based.
shifter = int.Parse(input) - 1;

// Shift mask of 0000000000000001
// over by cellLocations.
position = 1 << shifter;
```
Later in the program, you can iterate over each mask corresponding to winning positions on the board to determine whether the current player has a winning position, as shown in Listing 3.48.

***Production, Beginner Topic ends here***

**The continue Statement**

You might have a block containing a series of statements within a loop. If you determine that some conditions warrant executing only a portion of these statements for some iterations, you can use the continue statement to jump to the end of the current iteration and begin the next iteration. The continue statement exits the current iteration (regardless of whether additional statements remain) and jumps to the loop condition. At that point, if the loop conditional is still true, the loop will continue execution.

Listing 3.50 uses the continue statement so that only the letters of the domain portion of an email are displayed. Output 3.25 shows the results of Listing 3.50.

Listing 3.50: Determining the Domain of an Email Address
```csharp
class EmailDomain
{
    static void Main()
    {
        string email;
        bool insideDomain = false;
        System.Console.WriteLine("Enter an email address: ");
        email = System.Console.ReadLine();
        System.Console.Write("The email domain is: ");
        // Iterate through each letter in the email address.
        foreach (char letter in email)
        {
            if (!insideDomain)
            {
                if (letter == '@')
                {
                    insideDomain = true;
                }
            }
```
continue;
}

System.Console.Write(letter);
}

In Listing 3.50, if you are not yet inside the domain portion of the email address, you can use a continue statement to move control to the end of the loop, and process the next character in the email address.

You can almost always use an if statement in place of a continue statement, and this is usually more readable. The problem with the continue statement is that it provides multiple flows of control within a single iteration, and this compromises readability. In Listing 3.51, the sample has been rewritten, replacing the continue statement with the if/else construct to demonstrate a more readable version that does not use the continue statement.

Listing 3.51: Replacing a continue with an if Statement

```csharp
foreach (char letter in email)
{
    if (insideDomain)
    {
        System.Console.Write(letter);
    }
    else
    {
        if (letter == '@')
        {
            insideDomain = true;
        }
    }
}
```

The goto Statement

Early programming languages lacked the relatively sophisticated “structured” control flows that modern languages such as C# have as a matter of course, and instead relied upon simple conditional branching (if) and unconditional branching (goto) statements.
for most of their control flow needs. The resultant programs were often hard to understand. The continued existence of a goto statement within C# seems like an anachronism to many experienced programmers. However, C# supports goto, and it is the only method for supporting fall-through within a switch statement. In Listing 3.52, if the /out option is set, code execution jumps to the default case using the goto statement; similarly for /f.

Listing 3.52: Demonstrating a switch with goto Statements

```csharp
// ...
static void Main(string[] args)
{
    bool isOutputSet = false;
    bool isFiltered = false;

    foreach (string option in args)
    {
        switch (option)
        {
            case "/out":
                isOutputSet = true;
                isFiltered = false;
                goto default;
            case "/f":
                isFiltered = true;
                isRecursive = false;
                goto default;
            default:
                if (isRecursive)
                {
                    // Recurse down the hierarchy
                    // ...
                }
                else if (isFiltered)
                {
                    // Add option to list of filters.
                    // ...
                }
                break;
        }
    }
    // ...
}
```

Output 3.26 shows how to execute the code shown in Listing 3.52.
To branch to a switch section label other than the default label, you can use the syntax `goto` case constant; where constant is the constant associated with the case label you wish to branch to. To branch to a statement that is not associated with a switch section, precede the target statement with any identifier followed by a colon; you can then use that identifier with the `goto` statement. For example, you could have a labeled statement `myLabel : Console.WriteLine();` and then the statement `goto myLabel;` would branch to the labeled statement. Fortunately, C# prevents using `goto` to branch into a code block; it may only be used to branch within a code block or to an enclosing code block. By making these restrictions, C# avoids most of the serious `goto` abuses possible in other languages.

In spite of the improvements, using `goto` is generally considered to be inelegant, difficult to understand, and symptomatic of poorly structured code. If you need to execute a section of code multiple times or under different circumstances, either use a loop or extract code to a method of its own.

Guidelines

AVOID using `goto`.

C# Preprocessor Directives

Control flow statements evaluate expressions at runtime. In contrast, the C# preprocessor is invoked during compilation. The preprocessor commands are directives to the C# compiler, specifying the sections of code to compile or identifying how to handle specific errors and warnings within the code. C# preprocessor commands can also provide directives to C# editors regarding the organization of code.


**Language Contrast: C++—Preprocessing**

Languages such as C and C++ use a preprocessor to perform actions on the code based on special tokens. Preprocessor directives generally tell the compiler how to compile the code in a file and do not participate in the compilation process itself. In contrast, the C# compiler handles “preprocessor” directives as part of the regular lexical analysis of the source code. As a result, C# does not support preprocessor macros beyond defining a constant. In fact, the term preprocessor is generally a misnomer for C#.

Each preprocessor directive begins with a hash symbol (#), and all preprocessor directives must appear on one line. A newline rather than a semicolon indicates the end of the directive.

A list of each preprocessor directive appears in Table 3.4.

**Table 3.4: Preprocessor Directives**

<table>
<thead>
<tr>
<th>Statement or Expression</th>
<th>General Syntax Structure</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>#if directive</td>
<td>#if preprocessor-expression code #endif</td>
<td>#if CSHARP2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Console.Clear();</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#endif</td>
</tr>
<tr>
<td>#elif directive</td>
<td>#if preprocessor-expression1 code</td>
<td>#if LINUX</td>
</tr>
<tr>
<td></td>
<td>#elif preprocessor-expression2 code</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#elif WINDOWS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#endif</td>
</tr>
<tr>
<td>#else directive</td>
<td>#if code #else code #endif</td>
<td>#if CSHARP1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#else</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#endif</td>
</tr>
<tr>
<td>#define directive</td>
<td>#define conditional-symbol</td>
<td>#define CSHARP2</td>
</tr>
</tbody>
</table>
### Preprocessor Directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Symbol/Message</th>
<th>Symbol/Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>#undef</td>
<td>#undef conditional-symbol</td>
<td>#undef CSHARP2</td>
</tr>
<tr>
<td>#error</td>
<td>#error preproc-message</td>
<td>#error Buggy implementation</td>
</tr>
<tr>
<td>#warning</td>
<td>#warning preproc-message</td>
<td>#warning Needs code review</td>
</tr>
<tr>
<td>#pragma</td>
<td>#pragma warning</td>
<td>#pragma warning disable 1030</td>
</tr>
<tr>
<td>#line</td>
<td>#line org-line new-line</td>
<td>#line 467</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;TicTacToe.cs&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#line default</td>
</tr>
<tr>
<td>#region</td>
<td>#region pre-proc-message code</td>
<td>#region Methods</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#endregion</td>
</tr>
</tbody>
</table>

**Excluding and Including Code (#if, #elif, #else, #endif)**

Perhaps the most common use of preprocessor directives is in controlling when and how code is included. For example, to write code that could be compiled by both C# 2.0 and later compilers and the prior version 1.2 compilers, you use a preprocessor directive to exclude C# 2.0-specific code when compiling with a 1.2 compiler. You can see this in the tic-tac-toe example and in Listing 3.53.

Listing 3.53: Excluding C# 2.0 Code from a C# 1.x Compiler

```csharp
#if CSHARP2
    System.Console.Clear();
#endif
```

In this case, you call the `System.Console.Clear()` method, which is available only in 2.0 CLI and later versions. Using the `#if` and `#endif` preprocessor directives, this line of code will be compiled only if the preprocessor symbol `CSHARP2` is defined.

Another use of the preprocessor directive would be to handle differences among platforms, such as surrounding Windows- and Linux-specific APIs with `WINDOWS` and `LINUX` `#if` directives. Developers often use these directives in place of multiline comments (`/!*...*/`) because they are easier to remove by defining the appropriate symbol or via a search and replace. A final common use of the directives is for debugging. If you sur-
round code with an #if DEBUG, you will remove the code from a release build on most IDEs. The IDEs define the DEBUG symbol by default in a debug compile and RELEASE by default for release builds.

To handle an else-if condition, you can use the #elif directive within the #if directive, instead of creating two entirely separate #if blocks, as shown in Listing 3.54.

```
Listing 3.54: Using #if, #elif, and #endif Directives

#if LINUX
...
#elif WINDOWS
...
#endif
```

**Defining Preprocessor Symbols (#define, #undef)**

You can define a preprocessor symbol in two ways. The first is with the #define directive, as shown in Listing 3.55.

```
Listing 3.55: A #define Example

#define CSHARP2
```

The second method uses the define option when compiling for .NET, as shown in Output 3.27.

```
OUTPUT 3.27

>csc.exe /define:CSHARP2 TicTacToe.cs
```

Output 3.28 shows the same functionality using the Mono compiler.

```
OUTPUT 3.28

>mcs.exe -define:CSHARP2 TicTacToe.cs
```

To add multiple definitions, separate them with a semicolon. The advantage of the define compiler option is that no source code changes are required, so you may use the same source files to produce two different binaries.

To undefine a symbol you use the #undef directive in the same way you use #define.
**Emitting Errors and Warnings (#error, #warning)**

Sometimes you may want to flag a potential problem with your code. You do this by inserting `#error` and `#warning` directives to emit an error or warning, respectively. Listing 3.56 uses the tic-tac-toe sample to warn that the code does not yet prevent players from entering the same move multiple times. The results of Listing 3.56 appear in Output 3.29.

Listing 3.56: Defining a Warning with `#warning`

```csharp
#warning "Same move allowed multiple times."
```

**Output 3.29**

```
Performing main compilation...
...
...
\tictactoe.cs(471,16): warning CS1030: #warning: "Same move allowed multiple times."
Build complete – 0 errors, 1 warnings
```

By including the `#warning` directive, you ensure that the compiler will report a warning, as shown in Output 3.29. This particular warning is a way of flagging the fact that there is a potential enhancement or bug within the code. It could be a simple way of reminding the developer of a pending task.

***COMP: Begin C# 2.0 here.***

**Turning Off Warning Messages (#pragma)**

Warnings are helpful because they point to code that could potentially be troublesome. However, sometimes it is preferred to turn off particular warnings explicitly because they can be ignored legitimately. C# 2.0 and later compilers provide the preprocessor `#pragma` directive for just this purpose (see Listing 3.57).

Listing 3.57: Using the Preprocessor `#pragma` Directive to Disable the `#warning` Directive

```
#pragma warning disable 1030
```

Note that warning numbers are prefixed with the letters `CS` in the compiler output. However, this prefix is not used in the `#pragma` warning directive. The number corresponds to the warning error number emitted by the compiler when there is no preprocessor command.

To reenable the warning, `#pragma` supports the `restore` option following the warning, as shown in Listing 3.58.
In combination, these two directives can surround a particular block of code where the warning is explicitly determined to be irrelevant.

Perhaps one of the most common warnings to disable is CS1591, as this appears when you elect to generate XML documentation using the /doc compiler option, but you neglect to document all of the public items within your program.

**nowarn:<warn list> Option**

In addition to the #pragma directive, C# compilers generally support the nowarn:<warn list> option. This achieves the same result as #pragma, except that instead of adding it to the source code, you can insert the command as a compiler option. In addition, the nowarn option affects the entire compilation, and the #pragma option affects only the file in which it appears. Turning off the CS1591 warning, for example, would appear on the command line as shown in Output 3.30.

**Specifying Line Numbers (#line)**

The #line directive controls on which line number the C# compiler reports an error or warning. It is used predominantly by utilities and designers that emit C# code. In Listing 3.59, the actual line numbers within the file appear on the left.

Including the #line directive causes the compiler to report the warning found on line 125 as though it was on line 113, as shown in the compiler error message shown in Output 3.31.
Following the #line directive with default reverses the effect of all prior #line directives and instructs the compiler to report true line numbers rather than the ones designated by previous uses of the #line directive.

**Hints for Visual Editors (#region, #endregion)**

C# contains two preprocessor directives, #region and #endregion, that are useful only within the context of visual code editors. Code editors, such as the one in the Microsoft Visual Studio .NET IDE, can search through source code and find these directives to provide editor features when writing code. C# allows you to declare a region of code using the #region directive. You must pair the #region directive with a matching #endregion directive, both of which may optionally include a descriptive string following the directive. In addition, you may nest regions within one another.

Again, Listing 3.60 shows the tic-tac-toe program as an example.

**Listing 3.60: A #region and #endregion Preprocessor Directive**

```
... #region Display Tic-tac-toe Board
  #if CSHARP2
    System.Console.Clear();
  #endif

  // Display the current board;
  border = 0;  // set the first border (border[0] = "")

  // Display the top line of dashes.
  // ("\n--------\n")
  System.Console.Write(borders[2]);
  foreach (char cell in cells)
  {
    // Write out a cell value and the border that comes after it.
    System.Console.Write("{0} {1}", cell, borders[border]);

    // Increment to the next border;
    border++;

    // Reset border to 0 if it is 3.
    if (border == 3)
    {
      border = 0;
    }
```

```
One example of how these preprocessor directives are used is with Microsoft Visual Studio .NET. Visual Studio .NET examines the code and provides a tree control to open and collapse the code (on the left-hand side of the code editor window) that matches the region demarcated by the #region directives (see Figure 3.5).

Figure 3.5: Collapsed Region in Microsoft Visual Studio .NET

SUMMARY

This chapter began with an introduction to the C# operators related to assignment and arithmetic. Next, you used the operators along with the const keyword to declare constants. Coverage of all the C# operators was not sequential, however. Before discussing the relational and logical comparison operators, the chapter introduced the if statement and the important concepts of code blocks and scope. To close out the coverage of operators we discussed the bitwise operators, especially regarding masks. We also discussed
other control flow statements such as loops, `switch`, and `goto`, and ended the chapter with a discussion of the C# preprocessor directives.

Operator precedence was discussed earlier in the chapter; Table 3.5 summarizes the order of precedence across all operators, including several that are not yet covered.

Table 3.5: Operator Order of Precedence*

<table>
<thead>
<tr>
<th>Category</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td><code>x.y</code> <code>f(x)</code> <code>a[x]</code> <code>x++</code> <code>x--</code> <code>new</code> <code>typeof(T)</code> <code>checked(x)</code> <code>unchecked(x)</code> <code>default(T)</code> <code>delegate()</code> <code>()</code></td>
</tr>
<tr>
<td>Unary</td>
<td><code>+</code> <code>-</code> <code>!</code> <code>~</code> <code>++x</code> <code>--x</code> <code>(T)x</code></td>
</tr>
<tr>
<td>Multiplicative</td>
<td><code>*</code> <code>/</code> <code>%</code></td>
</tr>
<tr>
<td>Additive</td>
<td><code>+</code> <code>-</code></td>
</tr>
<tr>
<td>Shift</td>
<td><code>&lt;&lt;</code> <code>&gt;&gt;</code></td>
</tr>
<tr>
<td>Relational and type testing</td>
<td><code>&lt;</code> <code>&gt;</code> <code>&lt;=</code> <code>&gt;=</code> <code>is</code> <code>as</code></td>
</tr>
<tr>
<td>Equality</td>
<td><code>==</code> <code>!=</code></td>
</tr>
<tr>
<td>Logical AND</td>
<td><code>&amp;</code></td>
</tr>
<tr>
<td>Logical XOR</td>
<td><code>^</code></td>
</tr>
<tr>
<td>Logical OR</td>
<td>`</td>
</tr>
<tr>
<td>Conditional AND</td>
<td><code>&amp;&amp;</code></td>
</tr>
<tr>
<td>Conditional OR</td>
<td>`</td>
</tr>
<tr>
<td>Null coalescing</td>
<td><code>??</code></td>
</tr>
<tr>
<td>Conditional</td>
<td><code>?:</code></td>
</tr>
<tr>
<td>Assignment and lambda</td>
<td><code>*=</code> <code>/=</code> <code>%=</code> <code>+=</code> <code>-=</code> <code>&lt;&lt;=</code> <code>&gt;&gt;=</code> <code>&amp;=</code> <code>^=</code> `</td>
</tr>
</tbody>
</table>
Perhaps one of the best ways to review all of the content covered in Chapters 1–3 is to look at the tic-tac-toe program found in Appendix B. By reviewing the program, you can see one way in which you can combine all that you have learned into a complete program.