

LESSON 03

Software Engineering with Control Structures

Our C# programs so far have been simple. All we have learned to do is execute one statement after another in order. We have not had any choices. If we lived 'life like that, we would get up every day, get dressed, and have breakfast no matter how we felt. In reality, we make decisions among alternatives. If we are very sick we might stay in bed and not get dressed. (If we are very lucky, someone might bring us breakfast in bed.) We might not be hungry one morning, so we would get up, get dressed, but skip breakfast. Here is a description of our morning, with decisions:

```
if (I feel ill)
    stay in bed;
else {
    get up;
    get dressed;
    if (I feel hungry)
        eat breakfast;
}
```

In this "program," what I do depends on whether "I feel ill" is true or false. We will see in this chapter how to write C# expressions that are either true or false, and how to write C# statements that allow us to choose among alternatives based on the truth or falsity of a test expression.

Making choices gives us more flexibility, but we need even more control. For example, if I am thirsty, I might drink a glass of water, but one glass of water might not be enough. What I really want to do is to keep drinking water as long as I am still thirsty. I need to be able to repeat an action. The kind of program I want is

```
while (I feel thirsty)
    drink a glass of water;
```

We will see in this chapter how to write C# statements that allow us to repeat steps in our program.

We think of C# as flowing from one statement to the next as it executes our program. The if and while statements allow us to specify how C# should flow through our program as it executes its statements.

Controlling the flow of execution gives us flexibility as to which statements we execute, but we also need some choices about the type of data we use. So far, we have declared variables only of type int, representing whole numbers. In this chapter we will introduce the type double to represent decimal numbers.

LESSON 03

With the if-else and while statements and the type double, we have the language support to create more complex programs,¹ but how do we use these tools to solve problems? In this chapter we introduce a systematic process we can use to develop problem solutions.

■ RELATIONAL OPERATORS AND EXPRESSIONS

Arithmetic operators take numeric operands and give numeric results. For example, the value of 3+4 is an integer, 7. By contrast, an expression such as 3<4, stating that 3 is less than 4, gives the value true, and the expression 7<2 gives the value false. Type bool, named for the British mathematician and logician, George Boole (1815-1864), provides two values, true and false, which we use to express the value of relational and logical expressions.

C# provides relational and equality operators, listed in Figure 3.1, which take two operands of a primitive type and produce a bool result

Symbol	Meaning	Example	
<	less than	31 < 25	false
<=	less than or equal	464 <= 7213	true
>	greater than	-98 > -12	false
>=	greater than or equal	9 >= 99	false
==	equal	9 == 12 + 12	false
!=	not equal	292 != 377	true

FIGURE 3.1 C# relational and equality operators



The operators <=, >=, ==, and != are two-character operators which must be together, without any spaces between the two characters. The expression 3 <= 4 is fine, but 3 < = 4 will give an error. (The compiler thinks we want the '<' operator and cannot figure out why we did not give a correct right-hand operand.) ■

We can mix arithmetic operators and relational operators in the same expression, as in

```
643 < 350 + 450
```

which evaluates to true. We can omit parentheses because C# uses precedence rules, as we saw in Section 2.3. Arithmetic operators all have higher precedence than relational

LESSON 03



operators, so C# adds $350 + 450$ giving 800, and then determines that 643 is less than 800. We could have written the expression using parentheses, as in

```
643 < (350 + 450)
```

but in this case we can omit the parentheses and let C# use the precedence rules to evaluate the expression.² Some programmers prefer to include parentheses for clarity, even when they are not necessary.

We can use variables in relational expressions, and can declare variables of type `bool`. For example, if `x` is an integer variable, the expression

```
x < 3
```

is true if the value of `x` is less than 3, and false otherwise. The expression

```
x == 3
```

evaluates to true if `x` equals 3, and to false otherwise



Be careful not to confuse the equality operator, `==`, with the assignment operator, `=`. If `x` has the value 12, then `x == 3` evaluates to false, but `x = 3` assigns the value 3 to `x`, changing it from 12. ■

LESSON 03



EXAMPLE 3.1 ■ Relational.cs

```
/* Use relational expressions and boolean variables
*/

using System;
public class Relational {
    public static void Main( ) {
        int i = 3;
        bool result;                                // Note 1

        result = (32 > 87);                          // Note 2
        Console.WriteLine(" (32 > 87) is {0}", result);
        result = (-20 == -20);                       // Note 3
        Console.WriteLine(" (-20 == -20) is {0}", result);
        result = -20 == -20;                         // Note 4
        Console.WriteLine(" -20 == -20 is {0}", result);
        result = -20 == -10 - 10;                   // Note 5
        Console.WriteLine(" -20 == -10 - 10 is {0}", result);
        Console.WriteLine(" 16 <= 54 is {0}", 16 <= 54); // Note 6
        Console.WriteLine(" i != 3 is {0}", i != 3);    // Note 7
    }
}
```



```
(32 > 87) is False
(-20 == -20) is True
-20 == -20 is True
-20 == -10 - 10 is True
16 <= 54 is True
i != 3 is False
```

Note 1:bool result;

We can declare variables of type bool, which will have the values True or False.

Note 2:result = (32 > 87);

For clarity we use parentheses, but we could have omitted them because the greater than operator, >, has higher precedence than the assignment operator, =. The value of the bool variable result is false, a literal of the bool type

LESSON 03



Note 3: `result = (-20 == -20);`

We could omit the parentheses because the equality operator, `==`, has higher precedence than the assignment operator, `=`

Note 4: `result = -20 == -20;`

We do not need parentheses, because `==` has higher precedence than `=`.

Note 5: `result = -20 == -10 - 10;`

This expression uses the equality operator, the subtraction operator, and the negation operator. Again, we do not need parentheses

Note 6: `Console.WriteLine(" 16 <= 54 is {0}", 16 <= 54);`

We can use a relational expression in a `WriteLine` statement without assigning it to a variable. C# will evaluate the expression and display its value. Here we do not need to enclose `16<=54` in parentheses

Note 7: `Console.WriteLine(" i != 3 is {0}", i != 3);`

Here we used a variable, `i`, in a relational expression, `i != 3`. Because `i` has the value 3, the value of this expression is false

if AND if-else STATEMENTS

We are now ready to make choices about which statements to execute. Three C# statements permit us to make choices. We cover the `if` and `if-else` statements in this section. We cover the `switch` statement, which allows a choice among multiple alternatives, in the next chapter

The if Statement

The `if` statement is essential because

- It allows us to make choices
- It allows us to solve more complex problems

The `if` statement has the pattern

```
if (condition)
    if_true_statement
```

as in the example

```
if (x > 2)
    y = x + 17;
```

LESSON 03



The condition is an expression such as $x > 2$ that evaluates to true or false. The `if_true_statement` is a C# statement such as `y = x + 17`. If the condition is true, then execute the `if_true_statement`, but if the condition is false, skip the `if_true_statement` and go on to the next line of the program. In this example, if `x` happened to have the value 5, we would assign `y` the value 22, but if `x` had the value 1, we would skip the statement `y = x + 17`.

EXAMPLE 3.2 ■ Overtime.cs

```
/* Uses the if statement */

using System;
public class Overtime {
    public static void Main( ) {
        Console.Write("Enter the hours worked this week: ");
        int hours = int.Parse(Console.ReadLine( ));
        if (hours > 40) // Note 1
            Console.WriteLine("You worked overtime this week");
        Console.WriteLine
            ("You worked {0} hours", hours); // Note 2
    }
}
```

First run

```
Enter the hours worked this week: 76 You worked overtime this week You worked 76
hours
```

Second run

```
Enter the hours worked this week: 8 You worked 8 hours
```




Note 1: `if (hours > 40) Console.WriteLine("You worked overtime this week");`

The condition, `hours > 40`, is true if the number we enter is greater than 40, in which case C# executes the `WriteLine` statement to display the message, `You worked overtime this week`. If the number we enter is not greater than 40, then C# skips this `WriteLine` statement.

Note 2: `Console.WriteLine ("You worked {0} hours", hours);`

No matter what number we enter, C# executes this `WriteLine` statement, displaying the value we entered.

LESSON 03

 **Style** Indent all lines after the first to show that these lines are part of the if statement, and to make it easier to read.

```
Do
    if (myItem > 10)
        Console.WriteLine("Greater than ten");
Do Not
    if (myItem > 10)
        Console.WriteLine("Greater than ten");
```

Control Flow

Control flow refers to the order in which the processor executes the statements in a program. For example, the processor executes the three statements

```
int item1 = 25;
int item2 = 12;
item2 = item1 + 15;
```

one after the other. These three statements are in a sequence. We call this type of control flow, executing one statement after another in sequence, the **sequence** control structure. We can visualize the sequence structure in Figure 3.2, in which we write each statement inside a box and use directed lines to show the flow of control from one statement to the next

The if statement allows us to make a choice about the control flow.

In Example 3.2, if the hours worked is greater than 40, we print a message about overtime, otherwise we skip this message. We use a diamond shape to represent a decision based on the truth or falsity of a condition. One arrow, called the true branch, shows what comes next if the condition is true. Another arrow, called the false branch, shows

FIGURE 3.2 The sequence control flow

LESSON 03

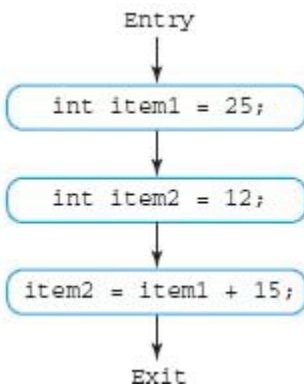
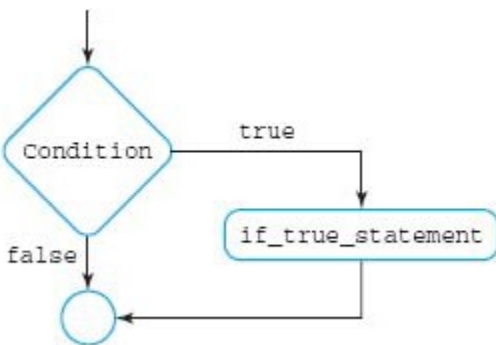


FIGURE 3.3 Control flow for the if statement



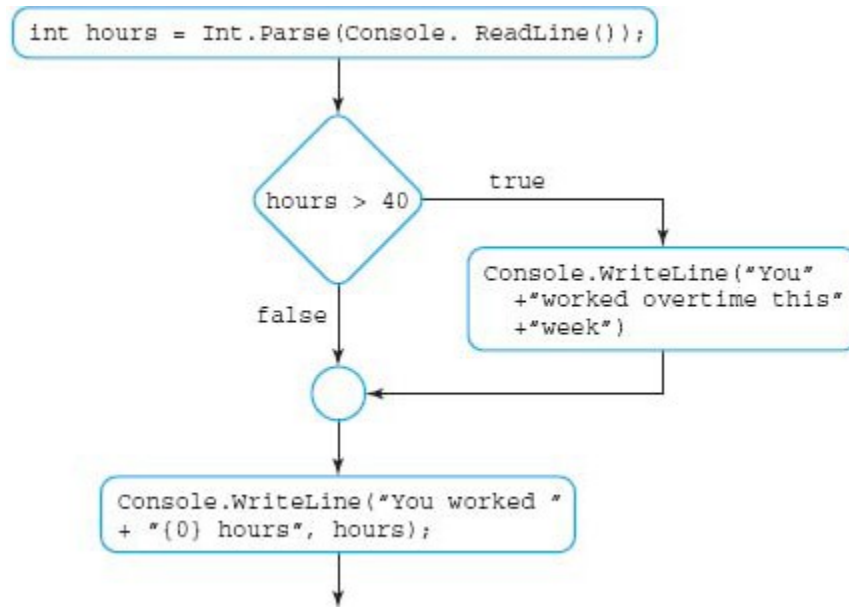
what comes next if the condition is false. Figure 3.3 shows the control flow for an if statement. When the condition is true, C# will execute an additional statement. Figure 3.4 shows the control flow for the program of Example 3.2

The if-else Statement

The if statement allows us to choose to execute a statement or not to execute it depending on the value of a test expression. With the if-else statement we can choose between two alternatives, executing one when the test condition is true and the other when the test condition is false

LESSON 03

FIGURE 3.4 Control flow for Example 3.2



The if-else statement has the form

```
if (condition)
    if_true_statement
else
    if_false_statement
```

For example,

```
if (x <= 20)
    x += 5;
else
    x += 2;
```

If x is less than or equal to 20, then we add 5 to it, otherwise we add 2 to it. The if-else statement gives us a choice between two alternatives. We choose `if_true_statement` if the condition is true and `if_false_statement` if the condition is false

LESSON 03



EXAMPLE 3.3 RentalCost.cs

```
/* Computes the cost of a car rental.
 */

using System;

public class RentalCost {

    /* Cost is $30 per day for the first three days
     * and $20 for each additional day.
     * Input: number of days
     * Output: cost of rental
     */
    public static int Cost(int days) {
        int pay;
        if (days <= 3)
            pay = 30*days;           // Note 1
        else
            pay = 90 + 20*(days - 3); // Note 2
        return pay;
    }

    // Enters number of days. Calls the cost method.
    public static void Main() {
        Console.WriteLine("Enter the number of rental days: ");
        int days = int.Parse(Console.ReadLine( ));
        Console.WriteLine("The rental cost is {0:C}", Cost(days));
    }
}
```

First run

Enter the number of rental days: 7 The rental cost is \$170

Second run

Enter the number of rental days: 2 The rental cost is \$60

Note 1:if (days <= 3) pay = 30*days;

If we rent for up to three days, the cost is \$30 times the number of days

Note 2:pay = 90 + 20*(days - 3);

If we rent for more than three days, the cost is \$90 for the first three days plus \$20 for each additional day

Figure 3.5 shows the flow chart for the if-else statement

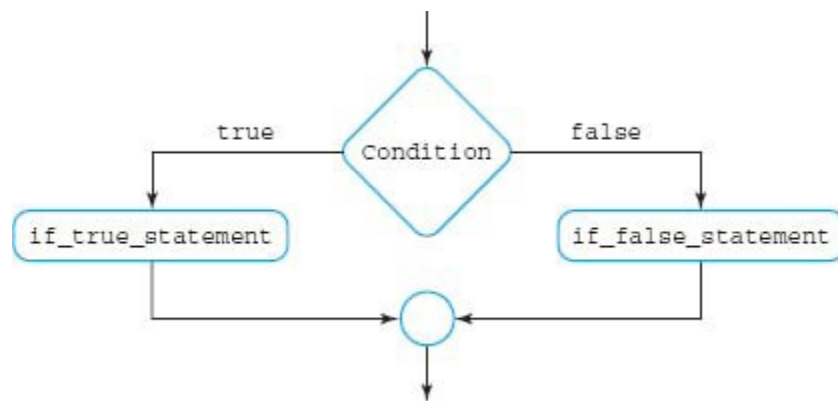
LESSON 03

Blocks

We can group a sequence of statements inside curly braces to form a **block**, as in

```
{  
  x = 5;  
  y = -8;  
  z = x * y;  
}
```

FIGURE 3.5 Flow chart for the if-else statement



We can use a block as a statement in an if or an if-else statement, as in

```
if (y > 5) {  
  x = 5;  
  y = -8;  
  z = x * y;  
}
```

By using a block, we can perform more than one action if the test condition is true. In this example, if *y* is greater than 5, we want to set *x*, *y*, and *z* to new values



Do not forget to enclose in curly braces the statements that you want to execute if a condition is true. Just indenting them, as in

LESSON 03



```
if (y > 5)
    x = 5;
    y = -8;
    z = x * y;
```

will not group the three statements together. We indent to make the program easier to read; indenting does not affect the meaning of the program. Without the braces, C# will interpret the code as

```
if (y > 5)
    x = 5;
y = -8;
z = x * y;
```

If y is greater than 5, then C# will set x to 5. Whether or not y is greater than 5, C# will always set y to -8, and z to x*y. This is quite a different result than we would get if we grouped the three statements in a block, and changed the values of x, y, and z only if the condition is true.



Style

Use a consistent style for blocks to help you match the opening brace, { , with the closing brace , } . One choice is to put the left brace on the same line as the if or else, and to align the right brace with the if or else, as in

```
if (x < 10){
    y = 5;
    z = 8;
}
else {
    y = 9;
    z = -2;
}
```

Using this style, we can match the keyword if with the closing brace, } , to keep our code neatly organized. Another choice is to align the left brace with the if or else, as in

```
if (x < 10)
{
    y = 5;
    z = 8;
}
else
{
    y = 9;
    z = -2;
}
```

LESSON 03



Either of these styles allows us to add or delete lines within a block without having to change the braces. The latter style makes it easier to match opening with closing braces, but uses an extra line to separate the opening brace from the code. We could make the code more compact by putting the braces on the same line as the code, but this is harder to read and modify, and is not recommended