4 More Control Structures and Types

In Chapters 2 and 3 we covered the essential concepts needed to start solving problems with C#. With the types int and double, the assignment, and if-else and while statements we can solve complex problems. But this is a bare bones set of tools. A good carpenter can do some fine construction with a few tools, but will be much more productive with a variety of tools to make the job easier. In this chapter, we will add some tools to build C# programs more easily.

Logical operators make it easier to use tests that are more complex, such as

```
I am hungry and I am thirsty
```

which is true if I am hungry and I am thirsty are both true.

The if-else statement lets us choose between two alternatives. Nested if statements and switch statements expand our choices to multiple alternatives. The while statement repeats while a condition is true. The for statement and the do statement make it easier to handle repetition.

We use the int and double types most frequently, but C# provides additional primitive types, including long, float, decimal, and char, which can sometimes be useful.

In addition to the statements and types in the C# language, C# libraries provide a wide range of methods to create user interfaces, connect across a network, and generally make C# a more powerful tool. In this chapter we look at the collection of methods in the Math library, which allow us to find powers, roots, maxima, minima, and evaluate natural logarithms and other functions.

Our problemsolving case study uses these topics to design a program to

convert lengths between the English and Metric systems. We use this case study to introduce good software engineering techniques.

OBJECTIVES

- Use logical operators
- Learn useful selection and repetition statements
- Use additional primitive types and operators
- Use the Math library
- Build software using an iterative development cycle

4.1 THE AND, OR, AND NOT OPERATORS

The C# conditional operators express the familiar AND and OR operations, which we can use to write conditions such as

```
John's age is greater than 20 AND John's age is less than 35.
John's height is greater than 78.5 OR John's weight is
greater than 300.
```

Figure 4.1 shows the C# symbols for the conditional operators.

Note that the operands of the conditional operators have type bool. The expression age > 20 is either true or false, and so is age < 35.

The && and || operators use two-character symbols that must be typed without any space between them. Using & & instead of && would give an error.

Conditional AND

The conditional AND expression (age > 20) && (age < 35) will be true only when both of its operands are true, and false otherwise. If the variable age has the value 25, both operands are true, and the whole && expression is true. If age has the value 17, the first operand, age > 20 is false, and the whole && expression is false. Figure 4.2 shows some sample evaluations of a conditional AND expression, illustrating how the value of an && expression depends on the values of its arguments.

Note that when the first operand is false, as it is when age is 17, we know that the conditional AND is false without even checking the value of the second operand.

FIGURE 4.1 Conditional operators

Symbol	*Meaning*	*Example*
`&&`	`conditional AND`	`(age > 20) && (age < 35)`
`||`	`conditional OR`	`(height > 78.5) || (weight > 300)`

FIGURE 4.2 Evaluating an example of a conditional AND expression

age	*age > 20*	*age < 35*	*age > 20 && age < 35*
10	false	true	false
25	true	true	true
40	true	false	false

Conditional OR

The conditional OR expression (height > 78.5) || (weight > 300) is true if either one of its operands is true, or if both are true. If height has the value 72 and weight has the value 310, then the first operand is false and the second operand is true, so the || expression is true. Figure 4.3 shows some sample evaluations of a conditional OR expression, illustrating how the value of an || expression depends on the values of its arguments.

Logical Complement

C# uses the symbol ! for the logical complement, or NOT, operator, which has only one operand. The logical complement negates the value of its operand, as Figure 4.4 shows.

If the bool variable, on, has the value true, then !on is false, but if on is false, then !on is true. Example 4.1 allows the user to enter different values of height and weight and displays the value of conditional AND, conditional OR, and logical complement expressions.

To make our programs easier to read, we abstracted some input and output methods in an IO class. We used several of these methods in Chapter 3 examples. We include the code for the IO class that contains these methods in an addendum at the end of this chapter. Using the IO class is just a convenience. With a loss of readability, we could copy the code for each method instead.

FIGURE 4.3 Evaluating an example of a conditional OR expression

height	*weight*	*height > 78.5*	*weight > 300*	*(height>78.5) || (weight>300)*
62	125	false	false	false
80	250	true	false	true
72	310	false	true	true
80	325	true	true	true

FIGURE 4.4 Evaluating a logical complement expression

A	*!A*
true	false
false	true

When using the .NET Framework SDK, we compile Measure.cs using the command

```
csc /r:IO.dll Measure.cs
```

where the /r option indicates that Measure.cs references the file IO.dll containing the compiled code for the IO class. When using Visual C#, we first create a new project, then click on *Project, Add, Add Existing Item* to add Measure.cs to the project. We click on *Project, Add Reference* to add the IO.dll file containing the IO library.

EXAMPLE 4.1 Measure.cs

```
/*  Evaluates AND, OR, and NOT expressions
 *  involving height and weight measurements.
 */

using System;
public class Measure {

     /* Inputs a height and a weight.
      * Computes boolean-valued expressions.
      */
  public static void Main() {
    double height =
      IO.GetDouble('Enter a height in inches: ');      // Note 1
    double weight = IO.GetDouble("Enter a weight in pounds: ');
    Console.Write("height < 65 && weight < 130 is ');  // Note 2
    Console.WriteLine((height < 65) && (weight  < 130));
    Console.Write("height < 65 || weight  < 130 is ");
    Console.WriteLine((height < 65) || (weight  < 130));
    bool heavy = (weight > 250);                       // Note 3
    Console.WriteLine("heavy is " + heavy);
    Console.WriteLine("!heavy is " + !heavy);
  }
}
```

First run

Enter a height in inches: 68
Enter a weight in pounds: 125
height < 65 && weight < 130 is False
height < 65 || weight < 130 is True
heavy is False
!heavy is True

Second run

Enter a height in inches: 70
Enter a weight in pounds: 260
height < 65 && weight < 130 is False
height < 65 || weight < 130 is False
heavy is True
!heavy is False

Note 1:

```
double height = \
  IO.GetDouble('Enter a height in inches: ');
```

We can enter a decimal number such as 62.5 or an integer such as 68, which C# will convert to a double. See the addendum at the end of this

chapter for the code for the IO.getDouble method.

Note 2: `Console.Write('height < 65 && weight < 130 is ');`

This statement displays the description of the expression, and the next statement displays the value of the expression.

Note 3: `bool heavy = (weight > 250);`

The variable heavy holds the value of the condition, weight > 250. This is often how we use the NOT operator. We call these conditions **flags**, meaning that they send a signal. In this example, the flag heavy signals that the weight is over 250.

Short-Circuiting

Both A and B must be true in order for A && B to be true. If Ais false, C# knows that A && B is false without even evaluating B. Thus when A is false, C# does not bother to evaluate B, so we do not care what the value of B is. We say that C# short-circuits the evaluation by not evaluating the second argument when it already knows the value of the expression.[1] In general, Figure 4.5 shows how the value of a conditional AND expression, A && B, depends on the value of its operands, A and B.

FIGURE 4.5 Evaluating a conditional AND expression

A	*B*	*A && B*
true	true	true
true	false	false
false	(don't care)	false

FIGURE 4.6 Evaluating a conditional OR expression

A	*B*	*A || B*
true	(don't care)	true
false	true	true
false	false	false

Either A or B, or both, must be true for A || B to be true. If A is true, C# knows that A | | B is true without even evaluating B. Thus when A is true, C# does not bother to evaluate B, so we do not care what the value of B is. As with

the conditional AND operator, C# short-circuits the evaluation by not evaluating the second argument when it already knows the value of the expression. In general, Figure 4.6 shows how the value of a conditional OR expression, A || B, depends on the value of its operands, A and B.

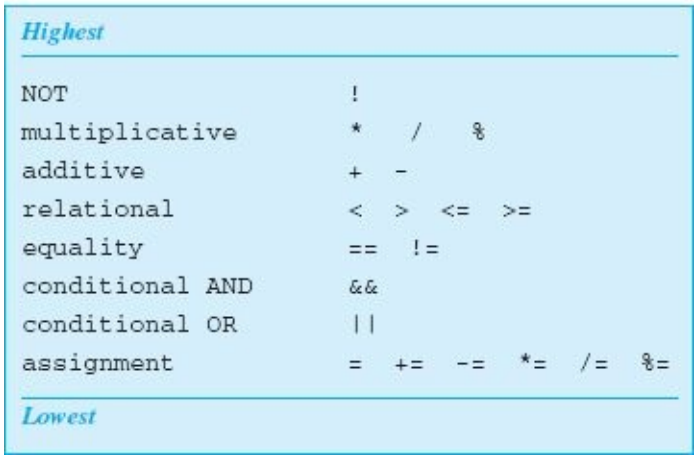
Operator Precedence

The conditional AND and conditional OR operators have lower precedence[2] than the relational and equality operators, as shown in Figure 4.7, where we show operators of equal precedence on the same line.

Remember that C# follows precedence rules in evaluating expressions, with the higher precedence operators getting their arguments first. In the expression

```
(age > 20) && (age < 35)
```

FIGURE 4.7 Operator precedence



we can omit the parentheses, writing it as

```
age > 20  && age < 35
```

The < and > operators have higher precedence than the && operator, so C# will first evaluate age > 20. If age > 20 is false, C# will short-circuit the evaluation, knowing that the result of the && operation must be false. If age > 20 is true, then C# will evaluate age < 35 to determine the value of the && expression.

Similarly, we can omit the parentheses in

```
(height > 78.5) || (weight > 300)
```

writing it as

```
height > 78.5 || weight > 300
```

The operator > has higher precedence than ||, so C# will evaluate height > 78.5 first. If height > 78.5, then the result of the OR operation, ||, will be true, and C# will short-circuit the evaluation. If height > 78.5 is false, C# will evaluate weight > 300 to determine the value of the || expression.

The logical complement operator has higher precedence than the arithmetic, relational, equality, and conditional operators. We must include parentheses in the expression

```
!(x < 10)
```

because we want C# to evaluate the relational expression x < 10, and then apply the NOT operator, !. If we write

```
!x < 10
```

then, because ! has higher precedence than <, C# will try to evaluate !x which will give an error, because the variable x is an integer, and NOT operates on bool values, not on integers.

Style

Use parentheses, even when not necessary, if they help to make the steps in the evaluation of the expression clearer, but omit them if they add too much clutter, making the expression harder to read. In Example 4.1, we chose to include the parentheses for clarity. Some programmers find it helpful to use parentheses always.

To show the use of the conditional AND in a program, we modify Example 3.6, which computes the sum of test scores, to add only those scores between zero and 100.

EXAMPLE 4.2 Zero100.cs

```
/*  Uses a while loop to compute the sum of
 *  valid test scores.
 */
```

```
using System;
public class Zero100  {

     /* Input:   The new score and the total so far
      * Output:  The updated total
      */
  public static int UpdateTotal(int newValue, int subTotal) {
    subTotal += newValue;
    return subTotal;
  }

     /* Sums scores between 0 and 100.
      * Exits when the user enters a value
      * out of that range.
      */
  public static void Main() {
    int total = 0;                          // Sum of scores
    int score =
      IO.GetInt("Enter the first score (0-100), -1 to quit: ");
    while (score >= 0  && score <= 100) {              // Note 1
      total = UpdateTotal(score, total);
      score =  IO.GetInt
               ("Enter the next score (0-100), -1 to quit: ");
    }
    Console.WriteLine("The total is {0}", total);
  }
}
```

Run

```
Enter the first score (0-100), -1 to quit: 72
Enter the next score (0-100), -1 to quit: 65
Enter the next score (0-100), -1 to quit: 122
The total is 137
```

Note 1: `while (score >= 0  && score <= 100) {`

We check that the score is between zero and 100 before adding it to the total.

Using an OR instead of an AND in Example 4.2 would not give the desired result. The expression (score >= 0)||(score <= 100) is true whenever either the score is greater than zero or less than 100, so any score makes it true.

Combining AND with OR

We can use both the && and || operators in the same expression, as in:

```
age > 50 && (height > 78.5 || height < 60)
```

where we need the parentheses because the AND operator has higher precedence than the OR operator. Without parentheses, as in

```
age > 50 && height > 78.5 || height < 60
```

C# will evaluate the expression as if we had written it as:

```
(age > 50 && height > 78.5) || height < 60
```

which is not what we intended.

A Little Extra: De Morgan's Laws

Two useful rules for working with logical expression are named after a British mathematician, Augustus De Morgan (1806-1871). They state

```
!(A && B) = !A || !B
```

and

```
!(A || B) = !A && !B
```

The BIG Picture

The &&, ||, and ! operators allow us to use logical expressions in our programs. Conditional AND and OR short-circuit, only evaluating their second arguments when necessary to determine the value of the expression.

Test Your Understanding

1. For each expression, find values for x and y that make it true.

a. (x == 2) && (y > 4)

b. (x <= 5) || (y >= 5)

c. x > 10 || y != 5

d. x > 10 && y < x + 4

2. For each expression in question 1, find values for x and y that make it false.

3. For each expression in question 1, find values for x that allow C# to short-circuit the evaluation and not evaluate the right-hand argument. What is the value of the conditional expression?

4. For each expression in question 1, find values for x that require C# to evaluate the right-hand argument.

5. For each expression, find a value for x that makes it true.

a. !(x == 5)
b. ! (x <= 10)
c. !(x > 10 && x < 50)
d. !(x == 5 || x > 8)

6. For each expression in question 5, find a value for x that makes it false.

7. Omit any unnecessary parentheses from the following expressions.

a. ((a > 1) || (c == 5))
b. ((x < (y+5)) && (y > 2))
c. !((x >2)||(y != 8))

4.2 NESTED ifS AND THE switch STATEMENT

With the if-else statement, we can choose between two alternatives. In this section we show two ways to choose between multiple alternatives, nested if statements and the switch statement.

Nested if Statements

Suppose we grade test scores so that 60-79 earns a C, 80-89 earns a B, and 90-100 earns an A. Given a test score between 60 and 100, we can determine the grade by first checking if the score is between 60 and 79 or higher, using the if-else statement of Figure 4.8.

This if-else statement only chooses between the two alternatives, grades C and B or better. To choose between the three alternatives, grades A, B, or C, we nest another if-else statement as the body of the else-part of our original if-else statement.

The code in Figure 4.9 has a problem. If we assume that a score is always between 60 and 100, then the code does what we expect, but let us trace the code if the score has a value of 40. Then the first test, score >=60 && score < 80, fails, so we execute the else-part, which is a nested if-else statement. Its condition, score >= 80 && score < 90, also fails, so we execute the else-part, which indicates that a score of 40 receives an A grade, not what we expect.

We can improve the code of Figure 4.9 by nesting an if statement in the last else-part to check that the score is really between 90 and 100, as shown in Figure 4.10.

FIGURE 4.8 if-else statement to choose between two alternatives

```
if (score >= 60 && score < 80)
  Console.WriteLine("Score " + score + " receives a C");
else
  Console.WriteLine("Score " + score + " receives a B or an A");
```

FIGURE 4.9 Nested if-else statement to choose among three alternatives

```
if (score >= 60 && score < 80)
  Console.WriteLine("Score " + score + " receives a C");
else if (score >=80 && score < 90)
   Console.WriteLine("Score " + score + " receives a B");
else
   Console.WriteLine("Score " + score + "receives an A");
```

FIGURE 4.10 Improved version of Figure 4.9

```
if (score >= 60 && score < 80)
  Console.WriteLine("Score " + score + " receives a C");
else if (score >=80 && score < 90)
   Console.WriteLine("Score " + score + " receives a B");
else if (score >= 90 && score <= 100)
   Console.WriteLine("Score " + score + "receives an A");
```

We see that using nested if-else statements allows us, in this example, tochoose among three alternatives:

```
scores between 60 and 79
scores between 80 and 89
scores between 90 and 100
```

Figure 4.10 illustrates the style for nested if-else statements to choose from multiple alternatives. Figure 4.11 shows the general pattern.

```
if ( Is it the first alternative? ){
    First alternative code
} else if ( Is it the second alternative? ) {
    Second alternative code
}
  ...
}else if ( Is it the last alternative? ) {
     Last alternative code
}else {
     Code when none of the above alternatives is true
}
```

FIGURE 4.11 Choosing from multiple alternatives

FIGURE 4.12 Flow chart for nested if-else statements

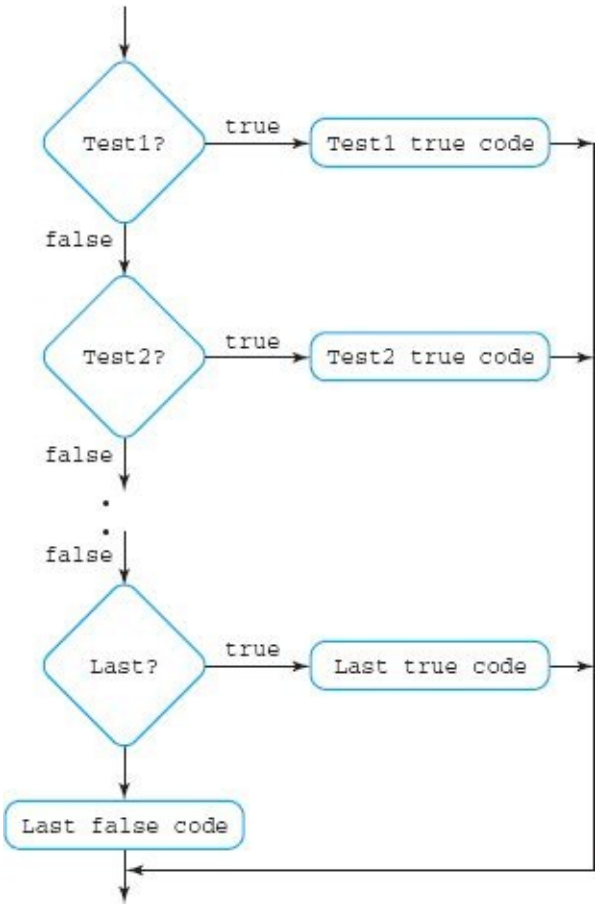


Figure 4.12 shows the flow chart for the nested if-else statement with the optional else-part at the end.

If you use code like that in Figure 4.9, having a final else with no nested if, then be sure that the code in the final else does handle everything else, that is, every case that does not come under one of the tested alternatives in the preceding if statements.

Pairing else with if

Without an additional rule, we cannot always determine how to read nested if statements. For example, contrast Figure 4.13 with Figure 4.14.

In Figure 4.13 we would like to pair else with the first if, but in Figure 4.14 we would like to pair the else with the second if. Unfortunately, aligning the else under the first if in Figure 4.13 will not achieve the outcome we want. As we know, C# does not consider spacing significant, so both examples will produce the same result. C# uses the rule that pairs an else with the nearest if. Figure 4.14 is the correct version, and would be correct even if we typed the else under the first if, as in Figure 4.13.

```
if (score >= 60)
  if (score >= 80)
    Console.WriteLine("You got a B or an A");
else
  Console.WriteLine("You got a D or an F"); // Wrong pairing
```

FIGURE 4.13 Incorrect attempt to pair an else with an if

```
if (score >= 60)
  if (score >= 80)
    Console.WriteLine("You got a B or an A");
  else
    Console.WriteLine("You got a C");   // Correct pairing
```

FIGURE 4.14 Corrected pairing of else and if

Both Figures 4.13 and 4.14 are if statements with nested if-else statements. What we tried to do in Figure 4.13 was write an if-else statement whose if-part contained a nested if statement. To do that, we need to enclose the nested if statement in braces, as in Figure 4.15.

Remember the rule: Pair an else with the nearest preceding if. Trace each branch of nested if statements, checking carefully to determine which values of the data will cause execution to flow to that branch.

Example 4.3 uses nested if-else statements to assign letter grades to test scores. We use the sentinel idea introduced in Example 3.6, where the user terminates the input of scores with a negative value.

```
if (score >= 60) {
 if (score >= 80)
  Console.WriteLine("You got a B or an A");
}else                                  // Paired to first 'if'
  Console.WriteLine("You got a D or an F");
```

FIGURE 4.15 Figure 4.13 rewritten as an if-else with nested if

EXAMPLE 4.3 ■ Score.cs

```
/*  Uses nested if-else statements to choose among alternatives
 *  to assign letter grades to test scores.
 */

using System;
public class Score {

     /*  Inputs scores, assigning grades, until the user
      * enters a negative value.
      */
  public static void Main() {
    int score = IO.GetInt("Enter a test score or -1 to quit: ");
    while (score >= 0) {
      if (score < 50)                                    // Note 1
        Console.WriteLine("Score {0} receives an F", score);
      else if (score < 60)
        Console.WriteLine("Score {0} receives a D", score);
      else if (score < 80)
        Console.WriteLine("Score {0} receives a C", score);
      else if (score < 90)
        Console.WriteLine("Score {0} receives a B", score);
      else if (score <= 100)
        Console.WriteLine("Score {0} receives an A', score);
      else                                               // Note 2
        Console.WriteLine
             ("Score can't be greater than 100, try again");
      score = IO.GetInt('Enter a test score or -1 to quit: ");
    }
  }
}
```

Run

```
Enter a test score or -1 to quit: 67
Score 67 receives a C
Enter a test score or -1 to quit: 82
Score 82 receives a B
Enter a test score or -1 to quit: 35
Score 35 receives an F
Enter a test score or -1 to quit: 98
Score 98 receives an A
Enter a test score or -1 to quit: 58
Score 58 receives a D
Enter a test score or -1 to quit: 123
Score can't be greater than 100, try again
Enter a test score or -1 to quit: -1
```

Note 1: `if (score < 50) {`

We extend the if-else statement from Figure 4.10 to interpret all test score values. Counting out-of-range values, we choose among six alternatives.

Note 2: `else`

This else-part catches any score not included by any previous alternative. Because the score has to be nonnegative for the while test to not fail, the only values not included are those greater than 100.

Be sure to test each alternative of nested if-else statements, as we did in Example 4.3.

The switch Statement

Choosing among six alternatives is stretching the use of nested if-else statements. The efficiency of this construction declines as we add more alternatives. For example, to interpret a score of 98, Example 4.3 tests five conditions, the first four of which fail. The switch statement allows us to check a large number of alternatives more efficiently.

A switch statement chooses alternatives based on the value of a variable. In this section, we use an int variable in our switch statement. In Section 4.4 we introduce character variables, which may also be used to indicate choices in a switch statement. We may also use the string type or other integer types to indicate switch choices.

The switch statement has the form

```
switch (test_expression)  {
  case expression1:
                    statement1;
  case expression2:
                    statement2;
   .....
  default:
                    default_statement;
 }
```

We can use a switch statement to replace the nested if-else statements of Example 4.3. Computing score/10 will give a number from 0 to 10 because each score is between 0 and 100. For example, 87/10 is 8 and 35/10 is 3. We can assign score/10 to a variable mark, as in int mark = score/10;

and use mark in the switch statement of Figure 4.16 to determine the grade for that score.

In Figure 4.16, C# evaluates the variable mark, jumping directly to one of 12 cases, depending on the value of mark. We specify each case with a case label such as case 5: which is made up of the word *case* followed by the number 5, followed by a colon. The label marks the place in the code to jump to when the switch variable value matches that case label. If mark is 5, C# executes the code following the label case 5:, which displays the grade of D; the break statement then causes a jump out of the switch statement, to the code following the closing brace, }.

If mark is 10, then C# jumps to the code at the label case 10:, which displays an A and breaks to the end of the switch. If mark is any integer other than 0 through 10, then C# jumps to the default case and displays an error message. The default case is optional. Had we omitted the default case in Figure 4.16, then C# would simply do nothing if the variable mark had any value other than 0 through 10. Note that several labels can refer to the same code, as for example, case 6 and case 7, which both label the statement that displays a C.

FIGURE 4.16 An example of a switch statement

```
switch(mark) {
  case 0:
  case 1:
  case 2:
  case 3:
  case 4:  Console.WriteLine("F");
           break;
  case 5:  Console.WriteLine("D");
           break;
  case 6:
  case 7:  Console.WriteLine("C");
           break;
  case 8:  Console.WriteLine("B");
           break;
  case 9:
  case 10: Console.WriteLine("A");
           break;
  default: Console.WriteLine("Incorrect score");
           break;
}
```

We must include the break statement after each case. C# does not allow code to “fall through” to the code for the next case. However, as in Figure 4.16,

several case labels may mark the same location.

Example 4.4 illustrates a good use for switch statements, which is to provide a menu to input choices from the user. We convert our programs from Examples 3.6 and 3.7 to methods to use as two of the menu choices.[3]

EXAMPLE 4.4 ■ UserMenu.cs

```
/*  Uses a switch statement to input a user's selection of
 *  an alternative. Uses code from Examples 3.6 and 3.7.
 */

using System;
public class UserMenu {

    //  Sums scores until user enters -1
  public static void Sum() {                          // Note 1
    int total = 0;             // Sum of scores
    int score =
      IO.GetInt("Enter the first score, -1 to quit: ');
    while (score >= 0) {
      total += score;
      score = IO.GetInt("Enter the next score, -1 to quit");
    }
    Console.WriteLine("The total is {0}", total);
  }

     /* Inputs prices until the users enters -1.
      * Outputs the maximum of all prices entered.
      */
  public static void Max() {                          // Note 2
    double maxSoFar = 0;   // Max of prices entered
    double price =
      IO.GetDouble('Enter the first price, -1 to quit: ");
    while (price >=0 ) {
      if (price > maxSoFar)
        maxSoFar = price;
      price =
        IO.GetDouble('Enter the next price, -1 to quit: ");
    }
    Console.WriteLine("The maximum is {0:C}", maxSoFar);
  }
```

[3] Section 2.4 introduced methods.

```
   // Loops until user chooses 4
  public static void Main() {
    int choice =
      IO.GetInt("Choose: 1-Sum, 2-Max, 3-To Do, or 4-Quit: ");
    while (choice != 4) {                                  // Note 3
      switch (choice) {
        case 1:
                Sum();                                     // Note 4
                break;                                     // Note 5
        case 2:
                Max();
                break;
        case 3:
                Console.WriteLine("Fill in code here");// Note 6
                break;
      }                                                    // Note 7
      choice =
        IO.GetInt("Choose: 1-Sum, 2-Max, 3-To Do, 4-Quit: ");
    }
  }
}
```

Run

Choose: 1-Sum, 2-Max, 3-To Do, or 4-Quit: 3
Fill in code here
Choose: 1-Sum, 2-Max, 3-To Do, 4-Quit: 4

Note 1: `public static void Sum() {`

This is the code from Example 3.6 to sum test scores entered by the user. We put it in a method, Sum, to call if the user selects choice 1.

Note 2: `public static void Max() {`

This is the code from Example 3.7 to find the maximum of the values input by the user.We put it in a method, Max, to call if the user selects choice 2.

Note 3: `while (choice != 4) {`

If choice has the value 1, 2, or 3, then the switch jumps to the code for that case. For a choice of 4, the while condition fails and the program finishes without executing the switch statement. If choice has any other value, then the switch does nothing, and execution continues after the switch by reading the user's next choice.

Note 4: `Sum();`

If choice has the value 1, then call the Sum method to add test scores. We can execute any C# statements, including if-else and while statements as well as method calls, after a case label.

Note 5: `break;`

This break statement causes execution to jump to the end of the switch statement, so the next statement will be the call to display the menu.

Note 6: `Console.WriteLine('Fill in code here');`

We have not really specified what to do if the user selects 3. We want to get the whole program working without adding any more code for additional choices. This illustrates a principle of program design, to solve the problem in stages, getting one stage right before going on to the next. We build a good framework and can fill in the details later.

Note 7: `}`

We do not need a default case in this example. If the user enters any value but 1, 2, 3, or 4, then the switch does nothing, the menu displays again, and C# asks the user for another choice. This is just what we want to happen, so there is no need for a default case.

Example 4.4 is longer than any of the previous examples, but much of it contains code developed in Chapter 3. In addition to illustrating the switch statement, it uses methods to satisfy user requests.

The BIG Picture

Nested if statements work for a choice among a few alternatives. With the switch statement, we can choose among many. switch statements can implement a menu of choices.

✓ Test Your Understanding

8. A charity designates donors who give more than $1,000 as Benefactors, those who give $500-$999 as Patrons, and those who give $100-$499 as Supporters. Write a nested if-else statement that, given the amount of a contribution, outputs the correct designation for that contributor.

9. Write a nested if-else statement that includes the categories from question 8 and identifies donors of $1-$99 as Contributors.

10. What value will the variable x have after executing

```
x = 6;
if (k < 10)
  if (k < 5)
    x = 7;
  else
    x = 8;
```

if k has the value

a. 9

b. 3

c. 11

d. −2

```
x = 6;
if (k < 10)
  if (k < 5)
    x = 7;
else
  x = 8;
```

if k has the value

a. 9

b. 3

c. 11

d. −2

11. What value will the variable x have after executing

```
x = 6;
if (k < 10)  {
   if (k < 5)
       x = 7;
}else
      x = 8;
```

if k has the value

a. 9

b. 3

c. 11

d. −2

12. What value will the variable x have after executing

```
x = 6;
if (k < 10)  {
   if (k < 5)
       x = 7;
}else
      x = 8;
```

if k has the value

a. 9

b. 3

c. 11

d. −2

13. What value will the variable x have after executing

```
x = 5;
switch(k) {
   case 2:
```

```
   case 3:  x = 6;
            break;
   case 5:  x = 7;
            break;
   case 9:  x = 8;
            break;
   default: x = 9;
            break;
}
```

if k has the value

a. 1
b. 3
c. 5
d. 6
e. 9
f. −5
g. 10

14. Answer question 13 for the code

```
x = 5;
switch(k) {
  case 2:
  case 3: x = 6;
          break;
  case 5: x = 7;
          break;
  case 9: x = 8;
          break;
}
```

4.3 THE for AND do LOOPS

In this section we introduce the for and do statements. The for statement makes it easy to repeat a block of code a fixed number of times. The do statement is like the while statement, with the difference that it tests the condition after executing a block of code rather than before. These new statements help us to write programs requiring repetition.

The for Statement

The for statement provides a powerful iteration capability. It works well when we know the number of repetitions. Technically, we could use a while statement instead of a for statement in these cases, but it is much more convenient to say

```
Do this calculation 10 times.
```

than it is to write

```
Declare and initialize a count variable to zero.
while (count < 10) {
    doSomething;
     count ++;
}
```