2c

Control Statements: Part 2

OBJECTIVES

In this lecture you will learn:

- The essentials of counter-controlled repetition.
- To use the `for` and `do…while` repetition statements to execute statements in an application repeatedly.
- To understand multiple selection using the `switch` selection statement.
- To use the `break` and `continue` program-control statements to alter the flow of control.
- To use the logical operators to form complex conditional expressions in control statements.

Outline

6.2 Essentials of Counter-Controlled Repetition

- Counter-controlled repetition requires
  - a **control variable** (or loop counter)
  - the **initial value** of the control variable
  - the **increment** (or **decrement**) by which the control variable is modified each **iteration of the loop**
  - the **loop-continuation condition**

- The application of Fig. 6.1 uses a loop to display the numbers from 1 through 10.

```
// Fig. 6.1: WhileCounter.cs
// Counter-controlled repetition with the while repetition statement.
using System;

public class WhileCounter
{
   public static void Main( string[] args )
   {
      int counter = 1; // declare and initialize control variable

      while ( counter <= 10 ) // loop-continuation condition
      {
         Console.Write( "{0}  ", counter );
         ++counter; // increment control variable
      } // end while

      Console.WriteLine(); // output a newline
   } // end Main
} // end class WhileCounter
```

```
1  2  3  4  5  6  7  8  9  10
```

Declaring the control variable's initial value

The loop-continuation condition.

The increment value for each iteration.

Fig. 6.1 | Counter-controlled repetition with the `while` repetition statement

Common Programming Error 6.1

Because floating-point values may be approximate, controlling loops with floating-point variables may result in imprecise counter values and inaccurate termination tests.

Error-Prevention Tip 6.1

Control counting loops with integers.

Good Programming Practice 6.1

Place blank lines above and below repetition and selection control statements, and indent the statement bodies to enhance readability.

6.2 Essentials of Counter-Controlled Repetition (Cont.)

- The application could be more concise by incrementing `counter` in the `while` condition:

```
int counter = 0;

while ( ++counter <= 10 ) // loop-continuation condition
   Console.Write( "{0}  ", counter ):
```

Software Engineering Observation 6.1

“Keep it simple” is good advice for most of the code you’ll write.

- The `for` **repetition statement** specifies the elements of counter-controlled-repetition in a single line of code.
- Figure 6.2 reimplements the application using the `for` statement.


```
// Fig. 6.2: ForCounter.cs
// Counter-controlled repetition with the for repetition statement.
using System;

public class ForCounter
{
   public static void Main( string[] args )
   {
      // for statement header includes initialization,
      // loop-continuation condition and increment
      for ( int counter = 1; counter <= 10; counter++ )
         Console.Write( "{0}  ", counter );

      Console.WriteLine(); // output a newline
   } // end Main
} // end class ForCounter
```

The `for` statement header controls the counter, the loop-continuation test, and incrementing the control variable.

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 6.2 | Counter-controlled repetition with the `for` repetition statement.

Common Programming Error 6.2

Using an incorrect relational operator or an incorrect final value of a loop counter in the loop-continuation condition of a repetition statement often causes an off-by-one error.

Good Programming Practice 6.2

Using the final value in the condition of a `while` or `for` statement with the <= relational operator helps avoid off-by-one errors. Many programmers prefer so-called zero-based counting, in which, to count 10 times, `counter` would be initialized to zero and the loop-continuation test would be `counter < 10`.

6.3 for Repetition Statement (Cont.)

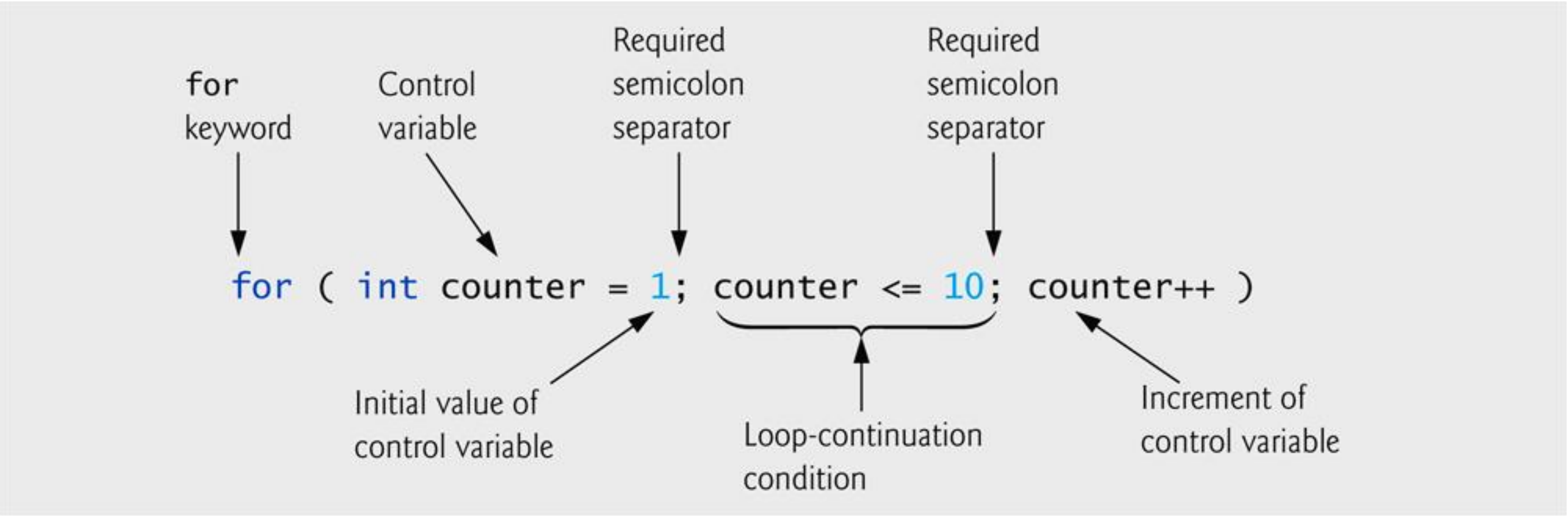

Fig. 6.3 | `for` statement header components.

6.3 `for` Repetition Statement (Cont.)

Common Programming Error 6.3

Using commas instead of the two required semicolons in a *`for`* header is a syntax error.

- A `for` statement usually can be represented with an equivalent `while` statement:

```
initialization;
while ( loopContinuationCondition )
{
    statement
    increment;
}
```

6.3 `for` Repetition Statement (Cont.)

- If the *initialization* expression declares the control variable, the control variable will not exist outside the `for` statement.
- This restriction is known as the variable's **scope**.
- Similarly, a local variable can be used only in the method that declares the variable and only from the point of declaration.

Common Programming Error 6.4

When a `for` statement's control variable is declared in the initialization section of the `for`'s header, using the control variable after the `for`'s body is a compilation error.

6.3 `for` Repetition Statement (Cont.)

- All three expressions in a for header are optional.
  - Omitting the *loopContinuationCondition* creates an infinite loop.
  - Omitting the *initialization* expression can be done if the control variable is initialized before the loop.
  - Omitting the *increment* expression can be done if the application calculates the increment with statements in the loop’s body or if no increment is needed.

6.3 `for` Repetition Statement (Cont.)

Performance Tip 6.1

There is a slight performance advantage to using the prefix increment operator, but if you choose the postfix increment operator, optimizing compilers will generate MSIL code that uses the more efficient form.

Good Programming Practice 6.3

In many cases, the prefix and postfix increment operators are both used to add 1 to a variable in a statement by itself. In these cases, the effect is exactly the same, except that the prefix increment operator has a slight performance advantage.

6.3 `for` Repetition Statement (Cont.)

Common Programming Error 6.5

Placing a semicolon immediately to the right of the right parenthesis of a `for` header makes that `for`'s body an empty statement. This is normally a logic error.

Error-Prevention Tip 6.2

Infinite loops occur when the loop-continuation condition never becomes false. Ensure that the control variable is incremented (or decremented) during each iteration of the loop. In a sentinel-controlled loop, ensure that the sentinel value is eventually input.

6.3 `for` Repetition Statement (Cont.)

- The initialization, loop-continuation condition and increment portions of a `for` statement can contain arithmetic expressions.

```
for ( int j = x; j <= 4 * x * y; j += y / x )
```

- If the loop-continuation condition is initially `false`, the application does not execute the `for` statement's body.

6.3 `for` Repetition Statement (Cont.)

Error-Prevention Tip 6.3

Although the value of the control variable can be changed in the body of a `for` loop, avoid doing so, because this practice can lead to subtle errors.

Common Programming Error 6.6

Not using the proper relational operator in the loop-continuation condition of a loop that counts downward (e.g., using `i <= 1` instead of `i >= 1` in a loop counting down to 1) is a logic error.

6.3 `for` Repetition Statement (Cont.)

- The `for` statement's UML activity diagram (Fig. 6.4) is similar to that of the `while` statement.

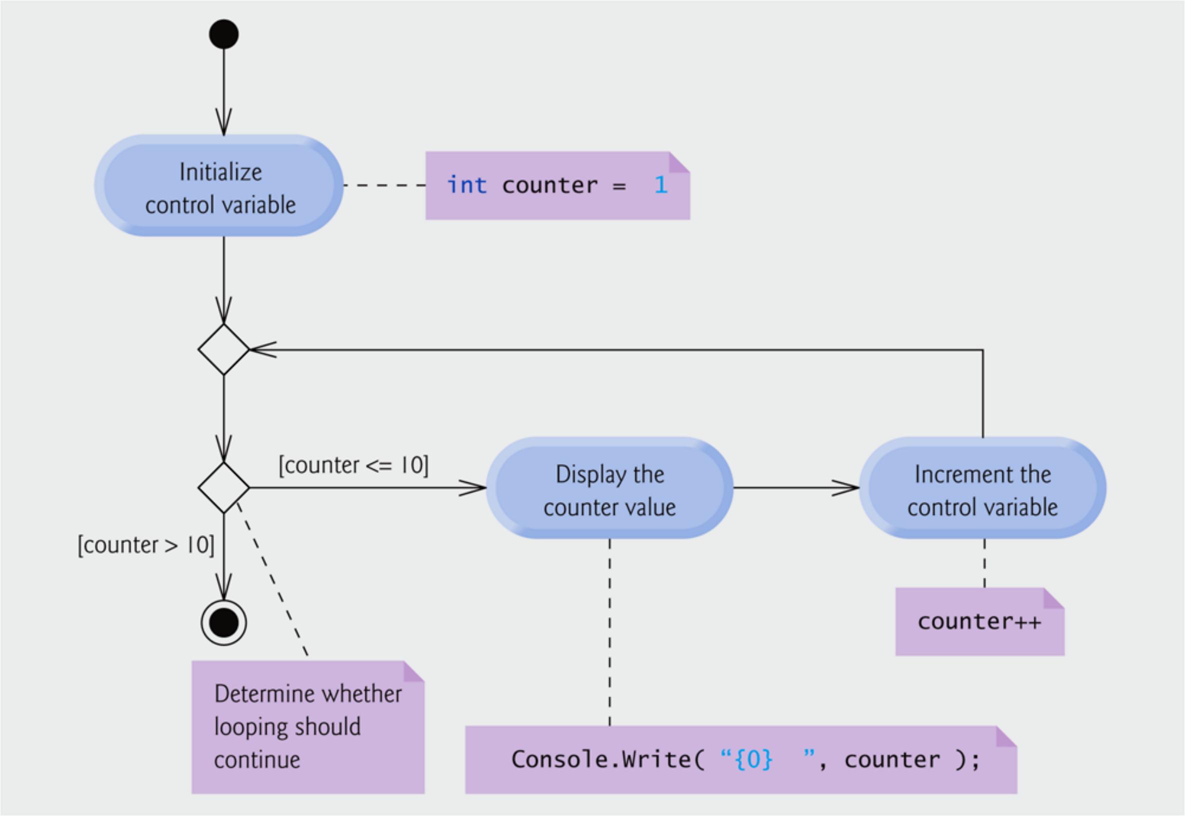

Fig. 6.4 | UML activity diagram for the `for` statement in Fig. 6.2.

Outline

- The application in Fig. 6.5 uses a `for` statement to sum the even integers from 2 to 20.

Sum.cs

```
// Fig. 6.5: Sum.cs
// Summing integers with the for statement.
using System;

public class Sum
{
   public static void Main( string[] args )
   {
      int total = 0; // initialize total

      // total even integers from 2 through 20
      for ( int number = 2; number <= 20; number += 2 )
         total += number;

      Console.WriteLine( "Sum is {0}", total ); // display results
   } // end Main
} // end class Sum
```

```
Sum is 110
```

Fig. 6.5 | Summing integers with the `for` statement.

6.4 Examples Using the `for` Statement (Cont.)

- Comma-separated lists that enable you to use multiple initialization expressions or multiple increment expressions:

```
for ( int number = 2; number <= 20; total += number,
    number += 2 )
     ; // empty statement
```

6.4 Examples Using the `for` Statement (Cont.)

Good Programming Practice 6.4

Limit the size of control-statement headers to a single line if possible.

Good Programming Practice 6.5

Place only expressions involving the control variables in the initialization and increment sections of a `for` statement. Manipulations of other variables should appear either before the loop or in the body of the loop.

6.4 Examples Using the `for` Statement (Cont.)

- Consider the following problem:

A person invests $1,000 in a savings account yielding 5% interest, compounded yearly. Calculate and display the amount of money in the account at the end of each year for 10 years.

$$a = p\,(1 + r)^n$$

p is the original amount invested (i.e., the principal)
r is the annual interest rate (use 0.05 for 5%)
n is the number of years
a is the amount on deposit at the end of the nth year.

- The application shown in Fig. 6.6 uses a loop that performs the calculation for each of the 10 years the money remains on deposit.

Interest.cs

(1 of 2)

```
// Fig. 6.6: Interest.cs
// Compound-interest calculations with for.
using System;

public class Interest
{
   public static void Main( string[] args )
   {
      decimal amount; // amount on deposit at end of each year
      decimal principal = 1000; // initial amount before interest
      double rate = 0.05; // interest rate

      // display headers
      Console.WriteLine( "Year{0,20}", "Amount on deposit" );

      // calculate amount on deposit for each of ten years
```

Format item `{0,20}` indicates that the value output should be displayed with a **field width** of 20.

Fig. 6.6 | Compound-interest calculations with `for`. (Part 1 of 2.)

```
        for ( int year = 1; year <= 10; year++ )
        {
           // calculate new amount for specified year
           amount = principal *
              ( ( decimal ) Math.Pow( 1.0 + rate, year ) );

           // display the year and the amount
           Console.WriteLine( "{0,4}{1,20:C}", year, amount );
        } // end for
     } // end Main
  } // end class Interest
```

Interest.cs


```
Year    Amount on deposit
   1            $1,050.00
   2            $1,102.50
   3            $1,157.63
   4            $1,215.51
   5            $1,276.28
   6            $1,340.10
   7            $1,407.10
   8            $1,477.46
   9            $1,551.33
  10            $1,628.89
```

The `for` statement executes 10 times, varying `year` from `1` to `10` in increments of `1`.

Fig. 6.6 | Compound-interest calculations with `for`. (Part 2 of 2.)

6.4 Examples Using the `for` Statement (Cont.)

- Format item `{0,20}` indicates that the value output should be displayed with a **field width** of 20.
  - To indicate that output should be **left justified**, use a negative field width.

6.4 Examples Using the `for` Statement (Cont.)

- Many classes provide static methods that do not need to be called on objects.
- Call a `static` method by specifying the class name followed by the member access (`.`) operator and the method name:

ClassName.methodName(*arguments*)

- `Console` methods `Write` and `WriteLine` are `static` methods.
- `Math.Pow(`*x*`, `*y*`)` calculates the value of *x* raised to the *y*th power.

6.4 Examples Using the `for` Statement (Cont.)

Good Programming Practice 6.6

Do not use variables of type `double` (or `float`) to perform precise monetary calculations; use type `decimal` instead.

Performance Tip 6.2

In loops, avoid calculations for which the result never changes—such calculations should typically be placed before the loop.

- The do...while repetition statement (Fig. 6.7) is similar to the while statement.

DoWhileTest.cs

```
// Fig. 6.7: DoWhileTest.cs
// do...while repetition statement.
using System;

public class DoWhileTest
{
   public static void Main( string[] args )
   {
      int counter = 1; // initialize counter

      do
      {
         Console.Write( "{0}  ", counter );
         ++counter;
      } while ( counter <= 10 ); // end do...while

      Console.WriteLine(); // outputs a newline
   } // end Main
} // end class DoWhileTest
```

```
1  2  3  4  5  6  7  8  9  10
```

Declaring and initializing the control variable.

The do...while loop outputs the numbers 1–10.

The application evaluates the loop-continuation test at the bottom of the loop.

Fig. 6.7 | do...while repetition statement.

6.5 do...while Repetition Statement (Cont.)

- Figure 6.8 contains the UML activity diagram for the do...while statement.

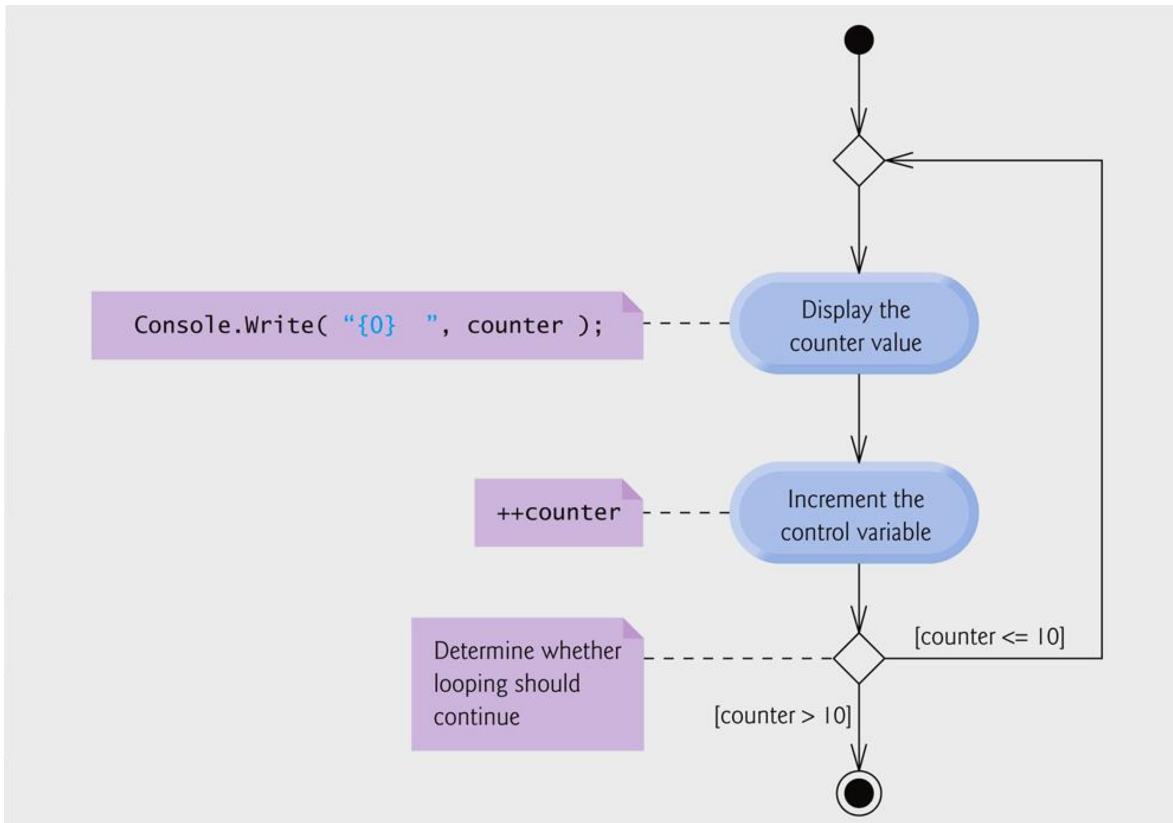

Fig. 6.8 | do...while repetition statement UML activity diagram.

6.5 do…`while` Repetition Statement (Cont.)

Error-Prevention Tip 6.4

Always include braces in a do…`while` statement, even if they are not necessary. This helps eliminate ambiguity between `while` statements and do…`while` statements containing only one statement.

6.6 `switch` Multiple-Selection Statement

- The **`switch` multiple-selection** statement performs different actions based on the value of an expression.
- Each action is associated with the value of a **constant integral expression** or a **constant string expression** that the expression may assume.

- Figure 6.9 contains an enhanced version of the `GradeBook` class.
- A `switch` statement determines whether each grade is an A, B, C, D or F.

```
// Fig. 6.9: GradeBook.cs
// GradeBook class uses switch statement to count letter grades.
using System;

public class GradeBook
{
   private int total; // sum of grades
   private int gradeCounter; // number of grades entered
   private int aCount; // count of A grades
   private int bCount; // count of B grades
   private int cCount; // count of C grades
   private int dCount; // count of D grades
   private int fCount; // count of F grades

   // automatic property CourseName
   public string CourseName { get; set; }
```

Keeping track of the sum of the grades and the number of grades entered, for averaging.

Counter variables for each grade category.

Fig. 6.9 | GradeBook class that uses a switch statement to count A, B, C, D and F grades. (Part 1 of 6)

Outline

```

   // constructor initializes automatic property CourseName;
   // int instance variables are initialized to 0 by default
   public GradeBook( string name )
   {
      CourseName = name; // set CourseName to name
   } // end constructor

   // display a welcome message to the GradeBook user
   public void DisplayMessage()
   {
      // CourseName gets the name of the course
      Console.WriteLine( "Welcome to the grade book for\n{0}!\n",
         CourseName );
   } // end method DisplayMessage

   // input arbitrary number of grades from user
   public void InputGrades()
   {
      int grade; // grade entered by user
      string input; // text entered by the user

      Console.WriteLine( "{0}\n{1}",
```

GradeBook.cs

(2 of 6)

Fig. 6.9 | GradeBook class that uses a switch statement to count A, B, C, D and F grades. (Part 2 of 6)

```
            "Enter the integer grades in the range 0-100.",
            "Type <Ctrl> z and press Enter to terminate input:" );

         input = Console.ReadLine(); // read user input

         // loop until user enters the end-of-file indicator (<Ctrl> z)
         while ( input != null )
         {
            grade = Convert.ToInt32( input ); // read grade off user input
            total += grade; // add grade to total
            ++gradeCounter; // increment number of grades

            // call method to increment appropriate counter
            IncrementLetterGradeCounter( grade );

            input = Console.ReadLine(); // read user input
         } // end while
      } // end method InputGrades

```

GradeBook.cs

(3 of 6)

*<**Ctrl**> **z*** is the Windows key sequence for typing the **end-of-file indicator**.

Fig. 6.9 | GradeBook class that uses a switch statement to count A, B, C, D and F grades. (Part 3 of 6)

```
// add 1 to appropriate counter for specified grade
private void IncrementLetterGradeCounter( int grade )
{
   // determine which grade was entered
   switch ( grade / 10 )
   {
      case 9: // grade was in the 90s
      case 10: // grade was 100
         ++aCount; // increment aCount
         break; // necessary to exit switch
      case 8: // grade was between 80 and 89
         ++bCount; // increment bCount
         break; // exit switch
      case 7: // grade was between 70 and 79
         ++cCount; // increment cCount
         break; // exit switch
      case 6: // grade was between 60 and 69
         ++dCount; // increment dCount
         break; // exit switch
      default: // grade was less than 60
         ++fCount; // increment fCount
         break; // exit switch
   } // end switch
} // end method IncrementLetterGradeCounter
```

The expression following keyword `switch` is called the **switch expression**.

The application attempts to match the value of the `switch` expression with one of the `case` labels.

The **`break` statement** causes program control to proceed with the first statement after the `switch`.

If no match occurs, the statements after the `default` label execute.

Fig. 6.9 | GradeBook class that uses a switch statement to count A, B, C, D and F grades. (Part 4 of 6)

Outline

GradeBook.cs

(5 of 6)

```
   // display a report based on the grades entered by the user
   public void DisplayGradeReport()
   {
      Console.WriteLine( "\nGrade Report:" );

      // if user entered at least one grade...
      if ( gradeCounter != 0 )
      {
         // calculate average of all grades entered
         double average = ( double ) total / gradeCounter;

         // output summary of results
         Console.WriteLine( "Total of the {0} grades entered is {1}",
            gradeCounter, total );
         Console.WriteLine( "Class average is {0:F}", average );
         Console.WriteLine( "{0}A: {1}\nB: {2}\nC: {3}\nD: {4}\nF: {5}",
```

Fig. 6.9 | GradeBook class that uses a switch statement to count A, B, C, D and F grades. (Part 5 of 6)

Outline

GradeBook.cs

(6 of 6)

```
             "Number of students who received each grade:\n",
             aCount, // display number of A grades
             bCount, // display number of B grades
             cCount, // display number of C grades
             dCount, // display number of D grades
             fCount ); // display number of F grades
       } // end if
       else // no grades were entered, so output appropriate message
          Console.WriteLine( "No grades were entered" );
    } // end method DisplayGradeReport
} // end class GradeBook
```

Fig. 6.9 | GradeBook class that uses a switch statement to count A, B, C, D and F grades. (Part 6 of 6)

6.6 `switch` Multiple-Selection Statement (Cont.)

- The expression following keyword `switch` is called the **`switch` expression**.
- The application attempts to match the value of the `switch` expression with one of the `case` labels.
- You are required to include a statement that terminates the `case`, such as a `break`, a `return` or a `throw`.
- The **`break` statement** causes program control to proceed with the first statement after the `switch`.
- If no match occurs, the statements after the `default` label execute.

Common Programming Error 6.7

Forgetting a `break` statement when one is needed in a `switch` is a compilation error.

- The `GradeBookTest` application (Fig. 6.10) uses class `GradeBook` to process a set of grades.

GradeBookTest.cs

(1 of 2)

```
// Fig. 6.10: GradeBookTest.cs
// Create GradeBook object, input grades and display grade report.

public class GradeBookTest
{
   public static void Main( string[] args )
   {
      // create GradeBook object myGradeBook and
      // pass course name to constructor
      GradeBook myGradeBook = new GradeBook(
         "CS101 Introduction to C# Programming" );

      myGradeBook.DisplayMessage(); // display welcome message
      myGradeBook.InputGrades(); // read grades from user
      myGradeBook.DisplayGradeReport(); // display report based on grades
   } // end Main
} // end class GradeBookTest
```

`InputGrades` reads a set of grades from the user.

`DisplayGradeReport` outputs a report based on the grades entered.

Fig. 6.10 | Create `GradeBook` object, input grades and display grade report. (Part 1 of 2)

```
Welcome to the grade book for
CS101 Introduction to C# Programming!

Enter the integer grades in the range 0-100.
Type <Ctrl> z and press Enter to terminate input:
99
92
45
100
57
63
76
14
92
^Z

Grade Report:
Total of the 9 grades entered is 638
Class average is 70.89
Number of students who received each grade:
A: 4
B: 0
C: 1
D: 1
F: 3
```

GradeBookTest.cs

(2 of 2)

Fig. 6.10 | Create `GradeBook` object, input grades and display grade report. (Part 2 of 2)

6.6 `switch` Multiple-Selection Statement (Cont.)

- Figure 6.11 shows the UML activity diagram for the general `switch` statement.

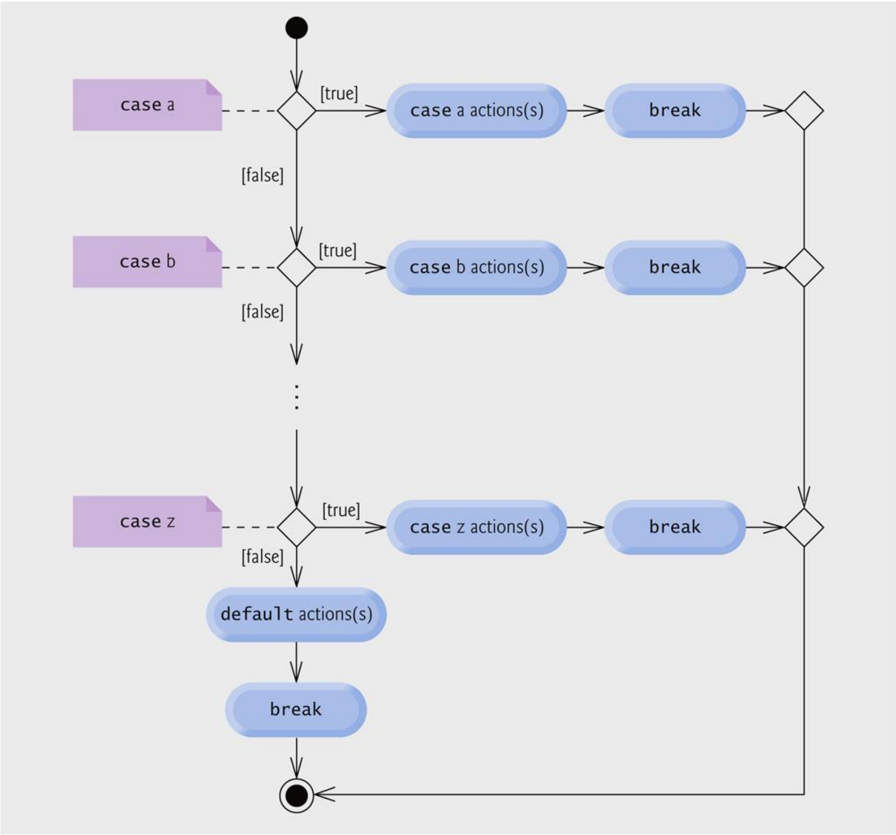

Fig. 6.11 | `switch` multiple-selection statement UML activity diagram with `break` statements.

6.6 `switch` Multiple-Selection Statement (Cont.)

Software Engineering Observation 6.2

Provide a `default` case in `switch` statements. Cases not explicitly tested in a `switch` that lacks a `default` case are ignored. Including a `default` case focuses you on the need to process exceptional conditions.

Good Programming Practice 6.7

Although each `case` and the `default` label in a `switch` can occur in any order, place the `default` label last for clarity.

6.6 `switch` Multiple-Selection Statement (Cont.)

- The expression after each `case` can be only a constant integral expression or a constant string expression.
- You can also use `null` and **character constants** which represent the integer values of characters.
- The expression also can be a **constant** that contains a value which does not change for the entire application.

- The `break` statement causes immediate exit from a statement.
- Figure 6.12 demonstrates a `break` statement exiting a `for`.

BreakTest.cs

```
// Fig. 6.12: BreakTest.cs
// break statement exiting a for statement.
using System;

public class BreakTest
{
   public static void Main( string[] args )
   {
      int count; // control variable also used after loop terminates

      for ( count = 1; count <= 10; count++ ) // loop 10 times
      {
         if ( count == 5 ) // if count is 5,
            break; // terminate loop

         Console.Write( "{0} ", count );
      } // end for

      Console.WriteLine( "\nBroke out of loop at count = {0}", count );
   } // end Main
} // end class BreakTest
```

When `count` is 5, the `break` statement terminates the `for` statement.

```
1 2 3 4
Broke out of loop at count = 5
```

Fig. 6.12 | `break` statement exiting a `for` statement.

6.7 `break` and `continue` Statements (Cont.)

- The `continue` **statement** skips the remaining statements in the loop body and tests whether to proceed with the next iteration of the loop.
- In a `for` statement, the increment expression executes, then the application evaluates the loop-continuation test.

Software Engineering Observation 6.3

Some programmers feel that `break` and `continue` statements violate structured programming, since the same effects are achievable with structured programming techniques.

- Figure 6.13 uses continue in a for statement.

ContinueTest.cs

```
// Fig. 6.13: ContinueTest.cs
// continue statement terminating an iteration of a for statement.
using System;

public class ContinueTest
{
   public static void Main( string[] args )
   {
      for ( int count = 1; count <= 10; count++ ) // loop 10 times
      {
         if ( count == 5 ) // if count is 5,
            continue; // skip remaining code in loop

         Console.Write( "{0} ", count );
      } // end for

      Console.WriteLine( "\nUsed continue to skip displaying 5" );
   } // end Main
} // end class ContinueTest
```

Skipping to the next iteration when `count` is 5.

`Console.Write` skips 5 because of the `continue` statement.

```
1 2 3 4 6 7 8 9 10
Used continue to skip displaying 5
```

Fig. 6.13 | `continue` statement terminating an iteration of a `for` statement.

6.7 `break` and `continue` Statements (Cont.)

Software Engineering Observation 6.4

There is a tension between achieving quality software engineering and achieving the best-performing software. Often, one of these goals is achieved at the expense of the other. For all but the most performance-intensive situations, apply the following rule: First, make your code simple and correct; then make it fast, but only if necessary.

6.8 Logical Operators

Conditional AND (&&) Operator

- The **&&** (**conditional AND**) operator works as follows:

```
if ( gender == "F" && age >= 65 )
   ++seniorFemales;
```

- The combined condition is true if and only if *both* simple conditions are true.
- The combined condition may be more readable with redundant parentheses:

```
( gender == "F" ) && ( age >= 65 )
```

6.8 Logical Operators (Cont.)

- This table (Fig. 6.14) shows all four possible combinations of `false` and `true` values for *expression1* && *expression2*.
- Such tables are called **truth tables**.

expression1	expression2	expression1 && expression2
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`true`

Fig. 6.14 | && (conditional AND) operator truth table.

6.8 Logical Operators (Cont.)

Conditional OR (||) Operator

- The || (**conditional OR**) operator, as in the following application segment:

```
if ( ( semesterAverage >= 90 ) || ( finalExam >= 90 ) )
   Console.WriteLine ( "Student grade is A" );
```

- The complex condition evaluates to `true` if either or both of the simple conditions are true.
- Figure 6.15 is a truth table for operator conditional OR (||).

expression1	expression2	expression1 || expression2
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`true`

Fig. 6.15 | || (conditional OR) operator truth table.

6.8 Logical Operators (Cont.)

- The parts of an expression containing && or || operators are evaluated only until it is known whether the condition is true or false.

```
( gender == "F" ) && ( age >= 65 )
```

- If `gender` is not equal to `"F"` (i.e., it is certain that the entire expression is `false`) evaluation stops.
- This feature is called **short-circuit evaluation**.

Common Programming Error 6.8

In expressions using operator &&, a condition—which we refer to as the dependent condition—may require another condition to be true for the evaluation of the dependent condition to be meaningful. In this case, the dependent condition should be placed after the other condition, or an error might occur.

6.8 Logical Operators (Cont.)

Boolean Logical AND (&) and Boolean Logical OR (|) Operators

- The **boolean logical AND** (**&**) and **boolean logical inclusive OR** (**|**) operators do not perform short-circuit evaluation.
- This is useful if the right operand has a required **side effect**. For example:

```
( birthday == true ) | ( ++age >= 65 )
```

- This ensures that the condition `++age >= 65` will be evaluated.

Error-Prevention Tip 6.5

For clarity, avoid expressions with side effects in conditions. The side effects may look clever, but they can make it harder to understand code and can lead to subtle logic errors.

6.8 Logical Operators (Cont.)

Boolean Logical Exclusive OR (^)

- A complex condition containing the **boolean logical exclusive OR** (^) operator (also called the **logical XOR operator**) is true *if and only if one of its operands is* `true` *and the other is* `false`*.*
- Figure 6.16 is a truth table for the boolean logical exclusive OR operator (^).

expression1	expression2	expression1 ^ expression2
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`false`

Fig. 6.16 | ^ (boolean logical exclusive OR) operator truth table.

6.8 Logical Operators (Cont.)

Logical Negation (!) Operator

- The **!** (**logical negation**) operator enables you to “reverse” the meaning of a condition.

```
if ( ! ( grade == sentinelValue ) )
   Console.WriteLine( "The next grade is {0}", grade );
```

- In most cases, you can avoid using logical negation by expressing the condition differently:

```
if ( grade != sentinelValue )
    Console.WriteLine( "The next grade is {0}", grade );
```

- Figure 6.17 is a truth table for the logical negation operator.

expression	!expression
`false`	`true`
`true`	`false`

Fig. 6.17 | ! (logical negation) operator truth table.

- Figure 6.18 demonstrates the logical operators and boolean logical operators.

LogicalOperators.cs


```
// Fig. 6.18: LogicalOperators.cs
// Logical operators.
using System;

public class LogicalOperators
{
   public static void Main( string[] args )
   {
      // create truth table for && (conditional AND) operator
      Console.WriteLine( "{0}\n{1}: {2}\n{3}: {4}\n{5}: {6}\n{7}: {8}\n",
         "Conditional AND (&&)", "false && false", ( false && false ),
         "false && true", ( false && true ),
         "true && false", ( true && false ),
         "true && true", ( true && true ) );

```

Producing the truth table for conditional AND.

Fig. 6.18 | Logical operators. (Part 1 of 5.)

```
      // create truth table for || (conditional OR) operator
      Console.WriteLine( "{0}\n{1}: {2}\n{3}: {4}\n{5}: {6}\n{7}: {8}\n",
         "Conditional OR (||)", "false || false", ( false || false ),
         "false || true", ( false || true ),
         "true || false", ( true || false ),
         "true || true", ( true || true ) );

      // create truth table for & (boolean logical AND) operator
      Console.WriteLine( "{0}\n{1}: {2}\n{3}: {4}\n{5}: {6}\n{7}: {8}\n",
         "Boolean logical AND (&)", "false & false", ( false & false ),
         "false & true", ( false & true ),
         "true & false", ( true & false ),
         "true & true", ( true & true ) );

      // create truth table for | (boolean logical inclusive OR) operator
      Console.WriteLine( "{0}\n{1}: {2}\n{3}: {4}\n{5}: {6}\n{7}: {8}\n",
         "Boolean logical inclusive OR (|)",
         "false | false", ( false | false ),
         "false | true", ( false | true ),
         "true | false", ( true | false ),
         "true | true", ( true | true ) );

```


Producing the truth table for conditional OR.

Producing the truth table for boolean logical AND.

Producing the truth table for boolean logical inclusive OR.

Fig. 6.18 | Logical operators. (Part 2 of 5.)

```
        // create truth table for ^ (boolean logical exclusive OR) operator
        Console.WriteLine( "{0}\n{1}: {2}\n{3}: {4}\n{5}: {6}\n{7}: {8}\n",
           "Boolean logical exclusive OR (^)",
           "false ^ false", ( false ^ false ),
           "false ^ true", ( false ^ true ),
           "true ^ false", ( true ^ false ),
           "true ^ true", ( true ^ true ) );

        // create truth table for ! (logical negation) operator
        Console.WriteLine( "{0}\n{1}: {2}\n{3}: {4}",
           "Logical negation (!)", "!false", ( !false ),
           "!true", ( !true ) );
     } // end Main
  } // end class LogicalOperators
```

LogicalOperators.cs

Producing the truth table for boolean logical exclusive OR.

Producing the truth table for logical negation.

Fig. 6.18 | Logical operators. (Part 3 of 5.)

```
Conditional AND (&&)
false && false: False
false && true: False
true && false: False
true && true: True

Conditional OR (||)
false || false: False
false || true: True
true || false: True
true || true: True

Boolean logical AND (&)
false & false: False
false & true: False
true & false: False
true & true: True
```

Fig. 6.18 | Logical operators. (Part 4 of 5.)

LogicalOperators.cs

(5 of 5)

```
Boolean logical inclusive OR (|)
false | false: False
false | true: True
true | false: True
true | true: True

Boolean logical exclusive OR (^)
false ^ false: False
false ^ true: True
true ^ false: True
true ^ true: False

Logical negation (!)
!false: True
!true: False
```

Fig. 6.18 | Logical operators. (Part 5 of 5.)

6.8 Logical Operators (Cont.)

- Figure 6.19 shows the C# operators from highest precedence to lowest.

Operators						Associativity	Type
`.`	`new`	`++`*(postfix)*	`--`*(postfix)*			left to right	highest precedence
`++`	`--`	`+`	`-`	`!`	`(`*type*`)`	right to left	unary prefix
`*`	`/`	`%`				left to right	multiplicative
`+`	`-`					left to right	additive
`<`	`<=`	`>`	`>=`			left to right	relational
`==`	`!=`					left to right	equality

Fig. 6.19 | Precedence/associativity of the operators discussed so far. (Part 1 of 2.)

6.8 Logical Operators (Cont.)

<table>
<tr><th>Operators</th><th>Associativity</th><th>Type</th></tr>
<tr><td>&</td><td>left to right</td><td>boolean logical AND</td></tr>
<tr><td>^</td><td>left to right</td><td>boolean logical exclusive OR</td></tr>
<tr><td>|</td><td>left to right</td><td>boolean logical inclusive OR</td></tr>
<tr><td>&&</td><td>left to right</td><td>conditional AND</td></tr>
<tr><td>||</td><td>left to right</td><td>conditional OR</td></tr>
<tr><td>?:</td><td>right to left</td><td>conditional</td></tr>
<tr><td>= += -= *= /= %=</td><td>right to left</td><td>assignment</td></tr>
</table>

Fig. 6.19 | Precedence/associativity of the operators discussed so far. (Part 2 of 2.)

6.9 Structured-Programming Summary

- Structured programming produces applications that are easier to understand, test, debug, and modify.
- The final state of one control statement is connected to the initial state of the next (Fig. 6.20). We call this “control-statement stacking.”

6.9 Structured-Programming Summary (Cont.)

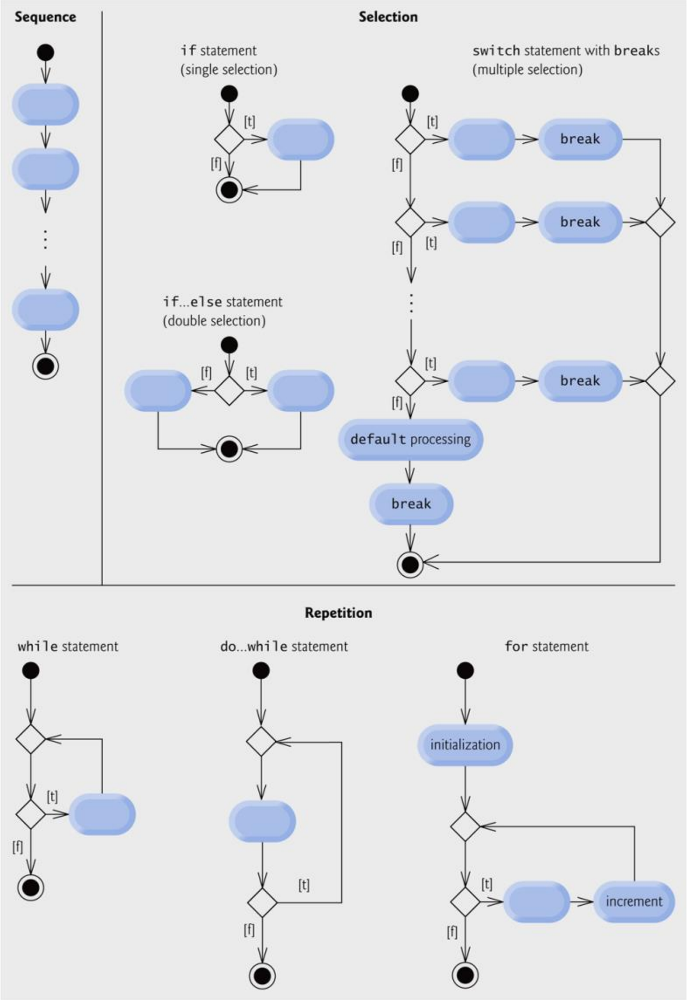

Fig. 6.20 | C#'s single-entry/single-exit sequence, selection and repetition statements.

6.9 Structured-Programming Summary (Cont.)

Rules for forming structured applications	
1	Begin with the simplest activity diagram (Fig. 6.22).
2	Any action state can be replaced by two action states in sequence.
3	Any action state can be replaced by any control statement.
4	Rules 2 and 3 can be applied as often as necessary in any order.

Fig. 6.21 | Rules for forming structured applications.

6.9 Structured-Programming Summary (Cont.)

- We begin with the simplest activity diagram (Fig. 6.22).

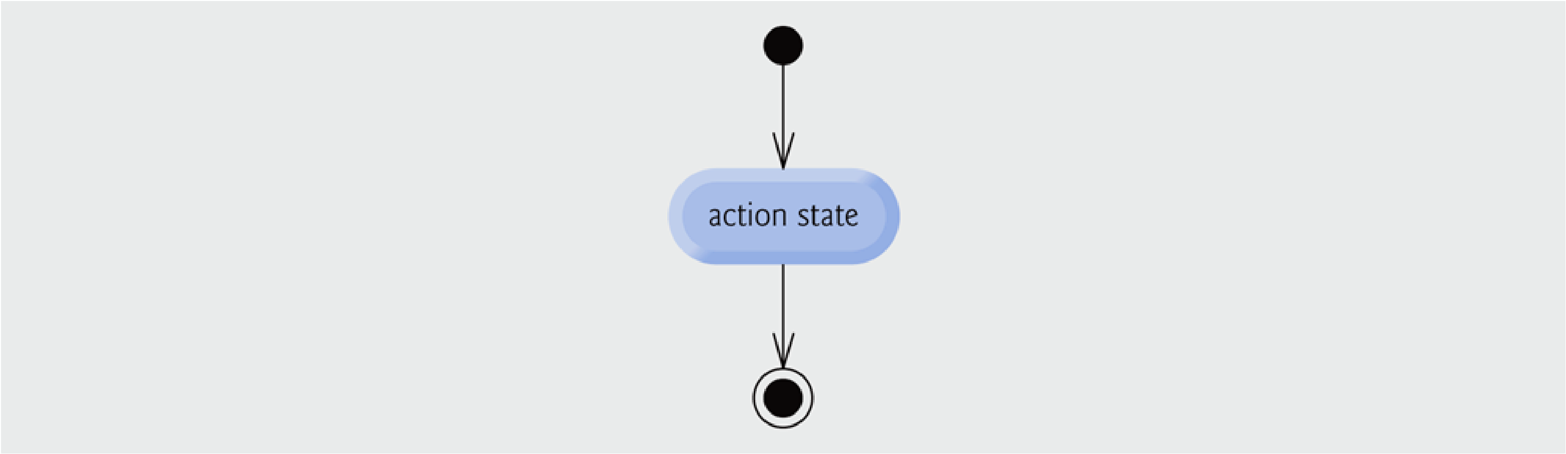

Fig. 6.22 | Simplest activity diagram.

6.9 Structured-Programming Summary (Cont.)

- Rule 2 generates a stack of control statements, so let us call rule 2 the **stacking rule**.
- Repeatedly applying rule 2 results in an activity diagram containing many action states in sequence (Fig. 6.23).

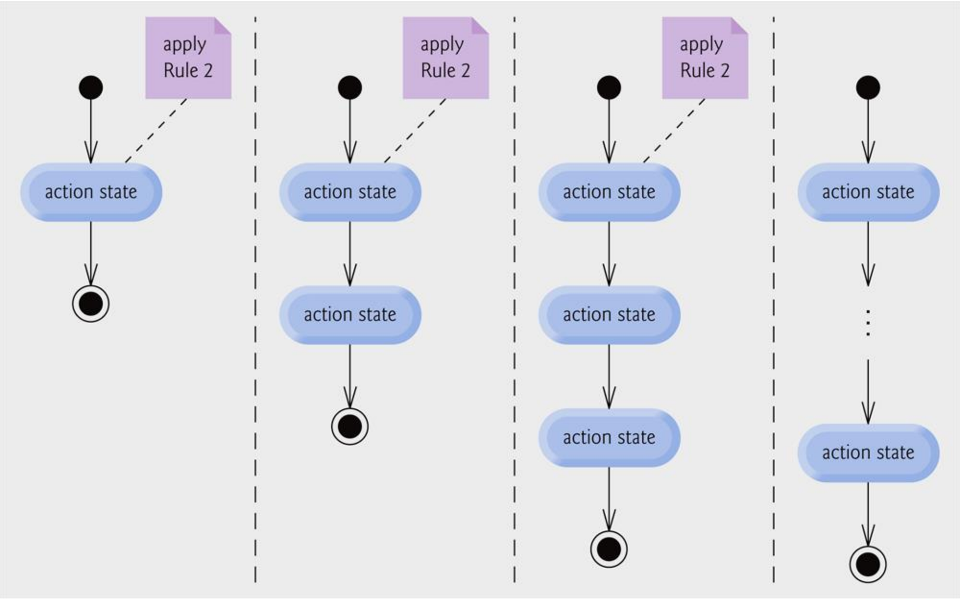

Fig. 6.23 | Repeatedly applying the stacking rule (Rule 2) of Fig. 6.21 to the simplest activity diagram.

6.9 Structured-Programming Summary (Cont.)

- Rule 3 is called the **nesting rule**.
- Repeatedly applying rule 3 results in an activity diagram with neatly **nested control statements**.

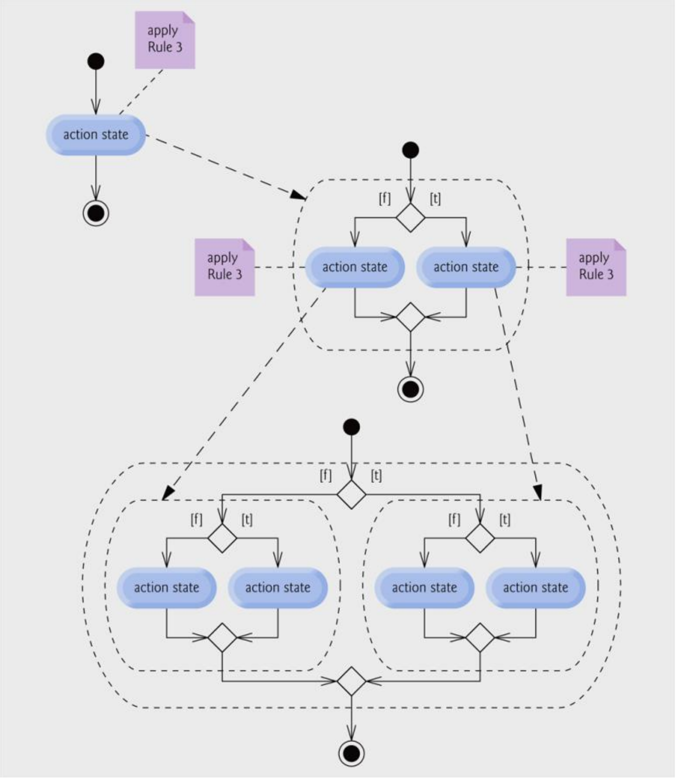

Fig. 6.24 | Repeatedly applying the nesting rule (Rule 3) of Fig. 6.21 to the simplest activity diagram.

6.9 Structured-Programming Summary (Cont.)

- Rule 4 generates larger, more involved and more deeply nested statements.
- If the rules are followed, an "unstructured" activity diagram (like the one in Fig. 6.25) cannot be created.

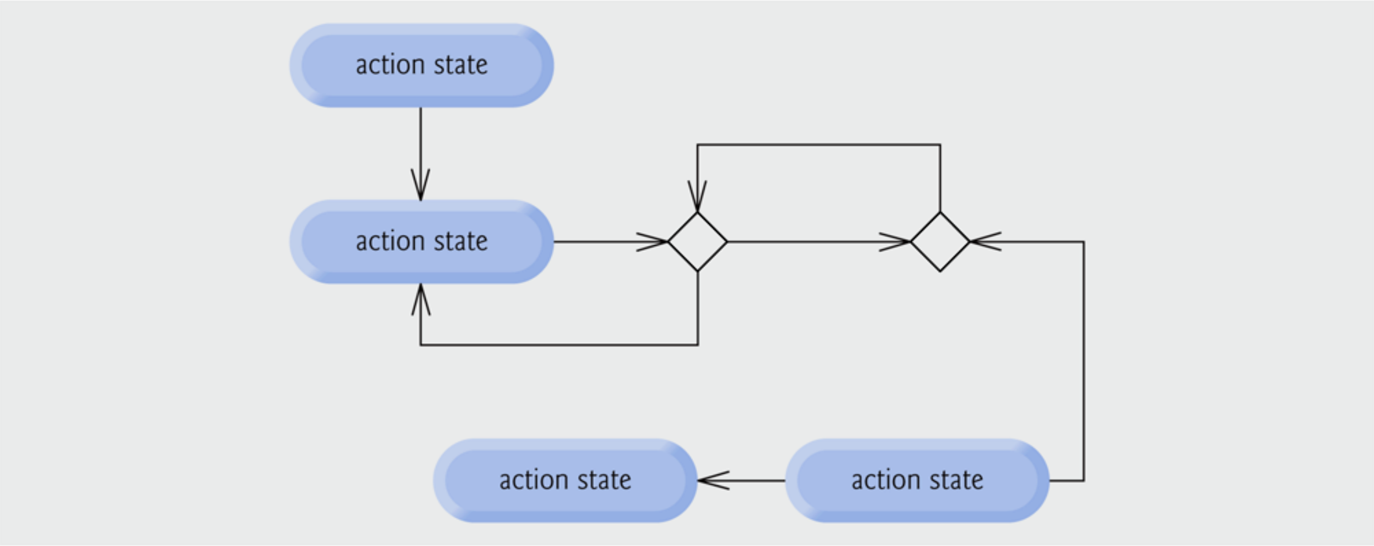

Fig. 6.25 | "Unstructured" activity diagram.