2b

Control Statements: Part 1

OBJECTIVES

In this lecture you will learn:

- To use basic problem-solving techniques.
- To develop algorithms through the process of top-down, stepwise refinement.
- To use the `if` and `if…else` selection statements to choose between alternative actions.
- To use the `while` repetition statement to execute statements in an application repeatedly.
- To use counter-controlled repetition and sentinel-controlled repetition.
- To use the increment, decrement and compound assignment operators.

Outline

Outline

5.2 Algorithms

- Computers solve problems by executing a series of actions in a specific order.
- An algorithm is a **procedure** for solving a problem, in terms of:
  - the **actions** to be executed and
  - the **order** in which these actions are executed

5.2 Algorithms (Cont)

- Consider the “rise-and-shine algorithm” followed by one junior executive for getting out of bed and going to work:
- (1) get out of bed, (2) take off pajamas, (3) take a shower, (4) get dressed, (5) eat breakfast and (6) carpool to work.

5.2 Algorithms (Cont)

- In a slightly different order:
- (1) get out of bed, (2) take off pajamas, (3) get dressed, (4) take a shower, (5) eat breakfast, (6) carpool to work.
- In this case, the junior executive shows up for work soaking wet.
- The order in which statements execute in a program is called **program control**.

5.3 Pseudocode

- **Pseudocode** is similar to every-day English, but it is not an actual computer programming language.
- It helps you “think out” an application before attempting to write it.
- A carefully prepared pseudocode application can easily be converted to a corresponding C# application.

5.4 Control Structures

- Normally, statements are executed one after the other in **sequential execution**.
- Various C# statements enable you to specify the next statement to execute. This is called **transfer of control**.
- Structured applications are clearer, easier to debug and modify, and more likely to be bug free.

5.4 Control Structures (Cont)

- An activity diagram models the **workflow** of a software system (Fig. 5.1).
- Activity diagrams are composed of symbols such as **action-state symbols**, **diamonds**, **small circles** and **notes**.
- **Transition arrows** represent the flow of the activity.
- The **solid circle** represents the activity’s **initial state**.
- The **solid circle surrounded by a hollow circle** represents the **final state**.

5.4 Control Structures (Cont)

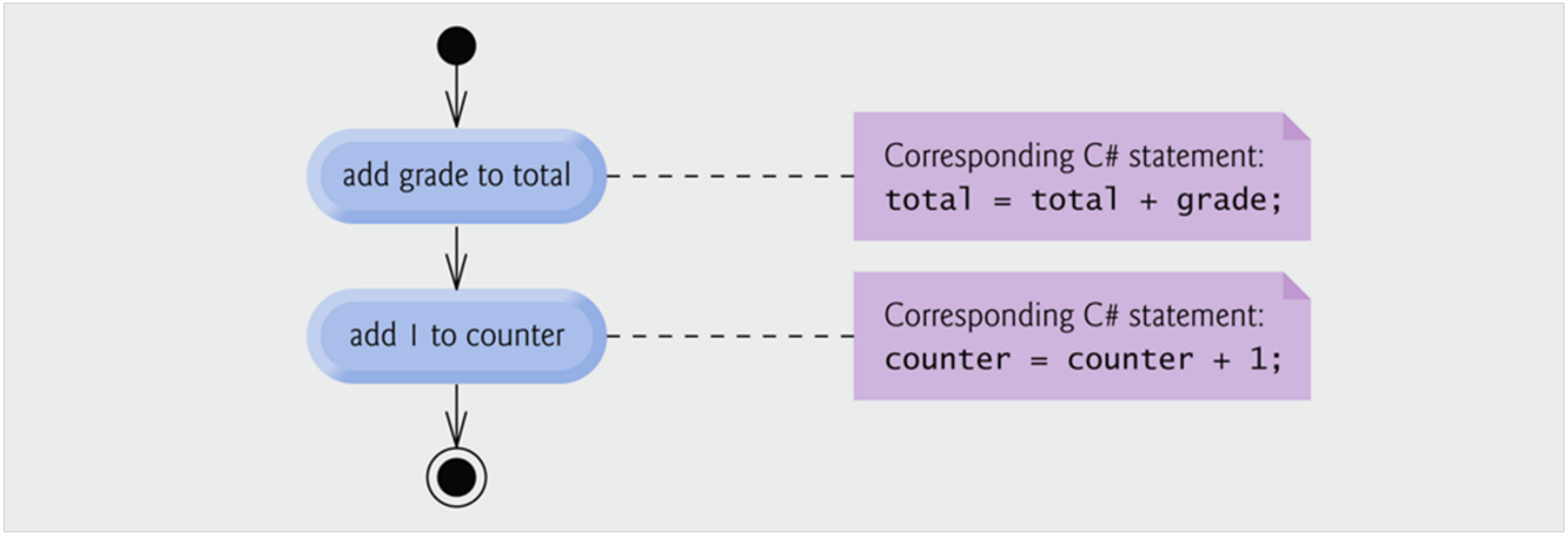

Fig. 5.1 | if else double-selection statement UML activity diagram.

5.4 Control Structures (Cont)

- **Single-entry/single-exit control statements** make it easy to build applications.
- Control statements are "attached" to one another by connecting the exit point of one to the entry point of the next.
- This procedure is called **control-statement stacking**.
- **Control-statement nesting** allows a control statement to appear inside another control statement.

5.5 `if` Single-Selection Statement

if grade is greater than or equal to 60
display "Passed"

- The preceding pseudocode *if* statement may be written in C# as:

```
if ( grade >= 60 )
    Console.WriteLine( "Passed" );
```

- Note that the C# code corresponds closely to the pseudocode.

5.5 `if` Single-Selection Statement (Cont.)

- Figure 5.2 illustrates the single-selection `if` statement.
- The diamond, or **decision symbol** indicates that a decision is to be made.
- The workflow will continue along a path determined by the symbol's associated **guard conditions**.

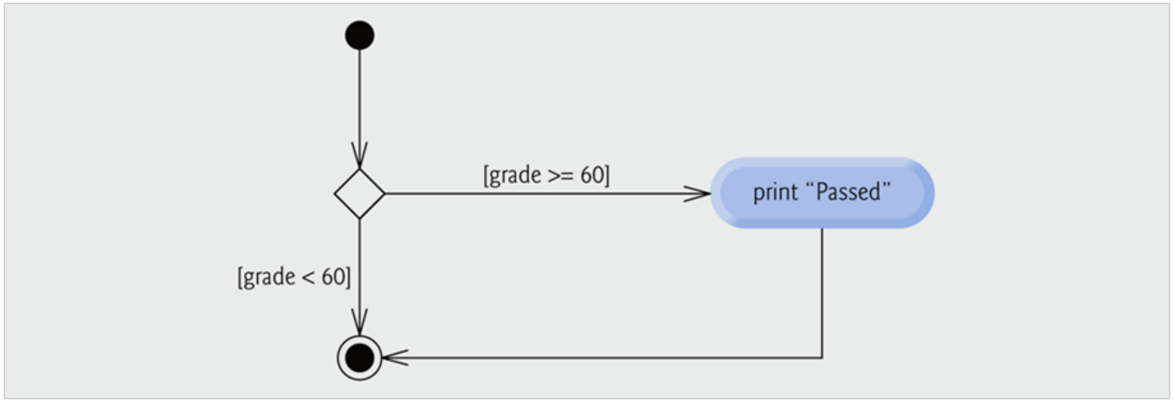

Fig. 5.2 | if single-selection statement UML activity diagram.

5.6 `if...else` Double-Selection Statement

- The `if...else` double-selection statement allows you to specify an action to perform when the condition is true and a different action when the condition is false:

if grade is greater than or equal to 60
display "Passed"
else
display "Failed"

- The preceding *if…else* pseudocode statement can be written in C# as

```
if ( grade >= 60 )
    Console.WriteLine( "Passed" );
 else
    Console.WriteLine( "Failed" );
```

5.6 `if…else` Double-Selection Statement (Cont.)

Good Programming Practice 5.1

Indent both body statements of an `if…else` statement.

Good Programming Practice 5.2

If there are several levels of indentation, each level should be indented the same addi-tional amount of space.

5.6 `if...else` Double-Selection Statement (Cont.)

- Figure 5.3 illustrates the flow of control in the `if…else` statement.
- The symbols in the UML activity diagram represent action states and a decision.

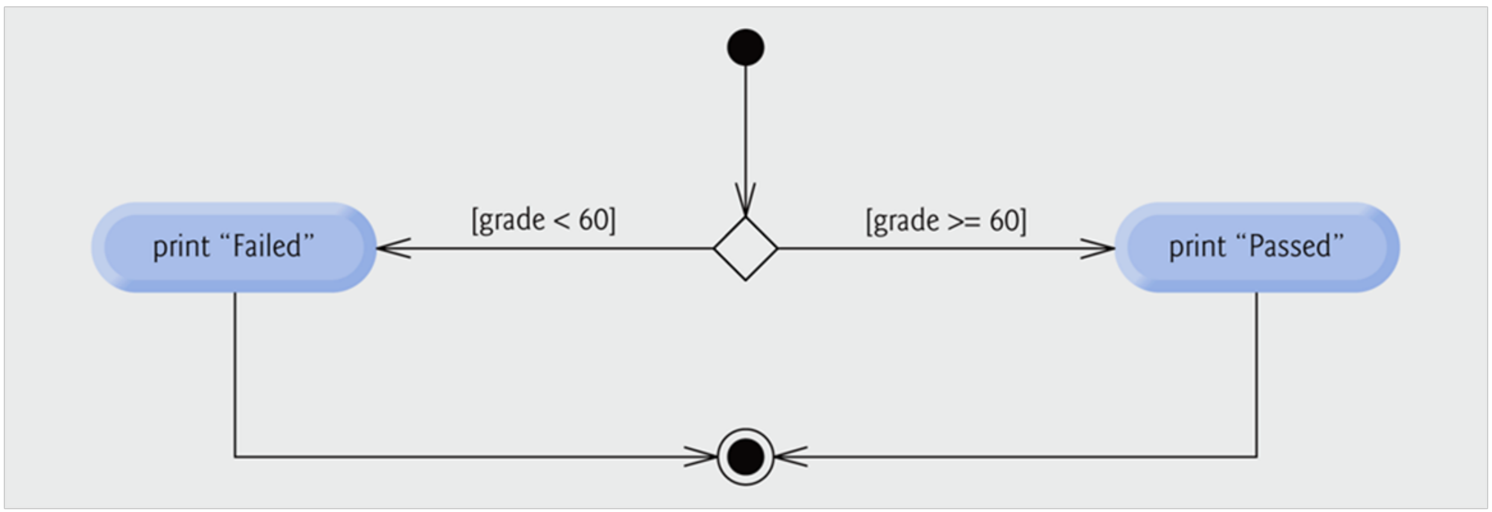

Fig. 5.3 | `if…else` double-selection statement UML activity diagram.

5.6 `if...else` Double-Selection Statement (Cont.)

- The **conditional operator** (**`?:`**) can be used in place of an `if`…`else` statement.

```
Console.WriteLine( grade >= 60 ? "Passed" :
  "Failed" );
```

  – The first operand is a **boolean** expression that evaluates to **`true`** or **`false`**.

  – The second operand is the value if the expression is `true`

  – The third operand is the value if the expression is `false`. `

5.6 `if...else` Double-Selection Statement (Cont.)

Good Programming Practice 5.3

Conditional expressions are more difficult to read than `if...else` statements and should be used to replace only simple if else statements that choose between two values.

Good Programming Practice 5.4

When a conditional expression is inside a larger expression, it's good practice to parenthesize the conditional expression for clarity. Adding parentheses may also prevent operator precedence problems that could cause syntax errors.

5.6 `if...else` Double-Selection Statement (Cont.)

- An application can test multiple cases with **nested `if...else` statements**:

if grade is greater than or equal to 90
display "A"
else
if grade is greater than or equal to 80
display "B"
else
if grade is greater than or equal to 70
display "C"
else
if grade is greater than or equal to 60
display "D"
else
display "F"

5.6 if...else Double-Selection Statement (Cont.)

- This pseudocode may be written in C# as

```
if ( grade >= 90 )
     Console.WriteLine( "A" );
  else
     if ( grade >= 80 )
        Console.WriteLine( "B" );
     else
        if ( grade >= 70 )
           Console.WriteLine( "C" );
        else
           if ( grade >= 60 )
              Console.WriteLine( "D" );
           else
              Console.WriteLine( "F" );
```

5.6 `if…else` Double-Selection Statement (Cont.)

- Most C# programmers prefer to use `else if`:

```
if ( grade >= 90 )
   Console.WriteLine( "A" );
else if ( grade >= 80 )
   Console.WriteLine( "B" );
else if ( grade >= 70 )
   Console.WriteLine( "C" );
else if ( grade >= 60 )
   Console.WriteLine( "D" );
else
   Console.WriteLine( "F" );
```

5.6 `if...else` Double-Selection Statement (Cont.)

- The C# compiler always associates an `else` with the immediately preceding `if` unless told to do otherwise by the placement of braces (`{` and `}`).
- This behavior can lead to what is referred to as the **dangling-`else` problem**.

5.6 if…else Double-Selection Statement (Cont.)

```
if ( x > 5 )
     if ( y > 5 )
        Console.WriteLine( "x and y are > 5" );
  else
     Console.WriteLine( "x is <= 5" );
```

- The compiler actually interprets the statement as:

```
if ( x > 5 )
     if ( y > 5 )
        Console.WriteLine( "x and y are > 5" );
     else
        Console.WriteLine( "x is <= 5" );
```

5.6 if…else Double-Selection Statement (Cont.)

- To have the nested if else statement execute as it was intended, write it as follows:

```
if ( x > 5 )
{
    if ( y > 5 )
       Console.WriteLine( "x and y are > 5" );
}
else
    Console.WriteLine( "x is <= 5" );
```

5.6 if…else Double-Selection Statement (Cont.)

- A set of statements contained within a pair of braces ({ and }) is called a **block**:

```
if ( grade >= 60 )
    Console.WriteLine( "Passed" );
else
{
    Console.WriteLine( "Failed" );
    Console.WriteLine( "You must take this course
      again." );
}
```

- Without the braces, the last statement would execute regardless of whether the grade was less than 60.

5.6 `if…else` Double-Selection Statement (Cont.)

- A **logic error** has its effect at execution time.
  - A **fatal logic error** causes an application to fail and terminate prematurely.
  - A **nonfatal logic error** allows an application to continue executing.

5.6 `if...else` Double-Selection Statement (Cont.)

Common Programming Error 5.1

Forgetting braces that delimit a block can lead to syntax errors or logic errors in an application.

Good Programming Practice 5.5

Always using braces in an `if...else` (or other) statement helps prevent their accidental omission. Some programmers type the beginning and ending braces of blocks before typing the individual statements within them.

5.6 `if...else` Double-Selection Statement (Cont.)

Common Programming Error 5.2

Placing a semicolon after the condition in an `if` or `if…else` statement leads to a logic error in single-selection `if` statements and a syntax error in double-selection `if…else` statements (when the `if`-part contains an actual body statement).

5.7 while Repetition Statement

- A **repetition statement** allows you to specify that an application should repeat an action:

while there are more items on my shopping list
put next item in cart and cross it off my list

- As an example of C#'s **while repetition statement**, consider a code segment designed to find the first power of 3 larger than 100:

```
int product = 3;
while ( product <= 100 )
    product = 3 * product;
```

5.7 `while` Repetition Statement (Cont.)

Common Programming Error 5.3

Not providing in the body of a `while` statement an action that eventually causes the condition in the `while` to become false normally results in a logic error called an infinite loop, in which the loop never terminates.

5.7 `while` Repetition Statement (Cont.)

- The UML activity diagram in Fig. 5.4 illustrates the flow of control that corresponds to the preceding `while` statement.
- The UML's **merge symbol** joins two flows of activity into one.

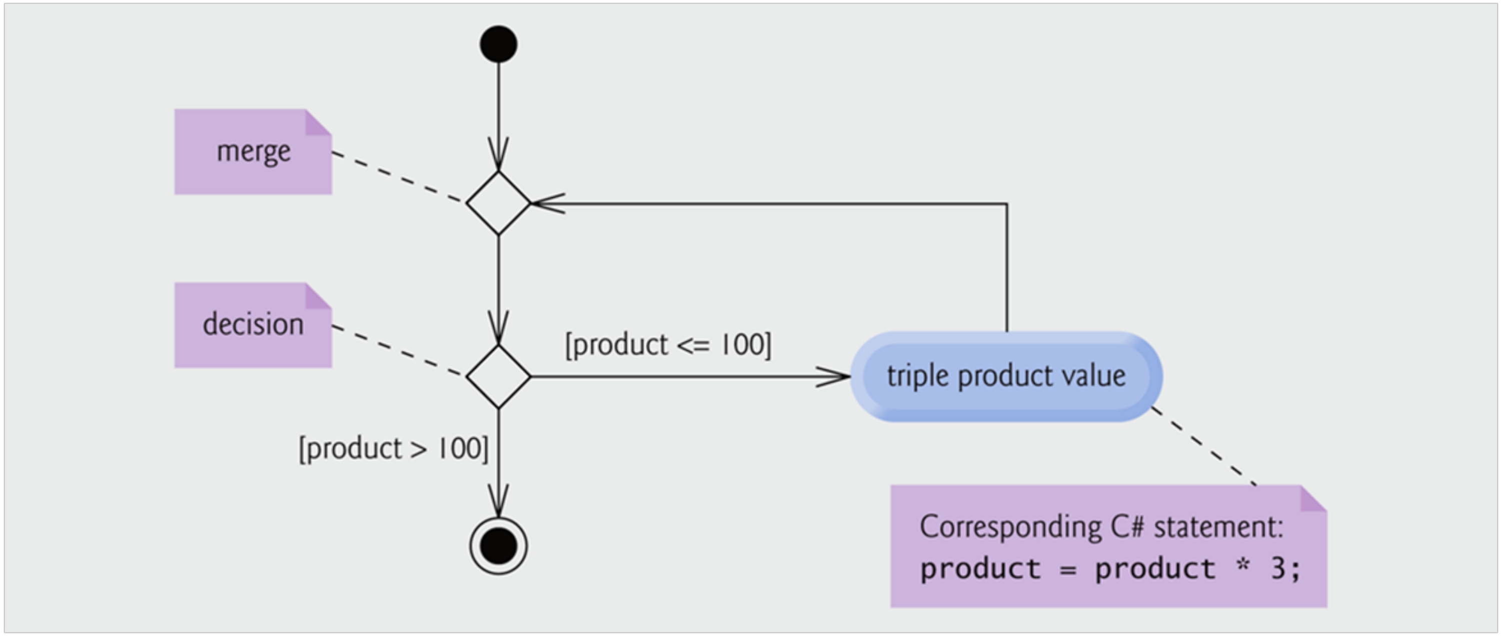

Fig. 5.4 | while repetition statement UML activity diagram.

5.8 Formulating Algorithms: Counter-Controlled Repetition

- Consider the following problem statement:

A class of 10 students took a quiz. The grades (integers in the range 0 to 100) for this quiz are available to you. Determine the class average on the quiz.

- The algorithm must input each grade, keep track of the total of all grades input, perform the averaging calculation and display the result.

5.8 Formulating Algorithms: Counter-Controlled Repetition (Cont.)

- We use **counter-controlled repetition** to input the grades one at a time.
- A variable called a **counter** controls the number of times a set of statements will execute.
- Counter-controlled repetition is often called **definite repetition**, because the number of repetitions is known before the loop begins executing.

Software Engineering Observation 5.1

Experience has shown that the most difficult part of solving a problem on a computer is developing the algorithm for the solution.

5.8 Formulating Algorithms: Counter-Controlled Repetition (Cont.)

- Figure 5.5 expresses the algorithm in pseudocode.

```
set total to zero
set grade counter to one

while grade counter is less than or equal to 10
    prompt the user to enter the next grade
    input the next grade
    add the grade into the total
    add one to the grade counter

set the class average to the total divided by 10
display the class average
```

Fig. 5.5 | Pseudocode algorithm that uses counter-controlled repetition to solve the class-average problem.

- Class `GradeBook` (Fig. 5.6) implements the grades algorithm.

GradeBook.cs

(1 of 3)

```
// Fig. 5.6: GradeBook.cs
// GradeBook class that solves class-average problem using
// counter-controlled repetition.
using System;

public class GradeBook
{
   // autoimplemented property CourseName
   public string CourseName { get; set; }

   // constructor initializes CourseName property
   public GradeBook( string name )
   {
      CourseName = name; // set CourseName to name
   } // end constructor
```

Fig. 5.6 | `GradeBook` class that solves the class-average problem using counter-controlled repetition. (Part 1 of 3.)

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

// determine class average based on 10 grades entered by user
public void DetermineClassAverage()
{
   int total; // sum of the grades entered by user
   int gradeCounter; // number of the grade to be entered next
   int grade; // grade value entered by the user
   int average; // average of the grades

   // initialization phase
   total = 0; // initialize the total
   gradeCounter = 1; // initialize the loop counter
```

`GradeBook.cs`

(2 of 3)

Declaring local variables `total`, `gradeCounter`, `grade` and `average` to be of type `int`.

Initializing the counter variable.

Fig. 5.6 | `GradeBook` class that solves the class-average problem using counter-controlled repetition. (Part 2 of 3.)

```

      // processing phase
      while ( gradeCounter <= 10 ) // loop 10 times
      {
         Console.Write( "Enter grade: " ); // prompt the user
         grade = Convert.ToInt32( Console.ReadLine() ); // read grade
         total = total + grade; // add the grade to total
         gradeCounter = gradeCounter + 1; // increment the counter by 1
      } // end while

      // termination phase
      average = total / 10; // integer division yields integer result

      // display total and average of grades
      Console.WriteLine( "\nTotal of all 10 grades is {0}", total );
      Console.WriteLine( "Class average is {0}", average );
   } // end method DetermineClassAverage
} // end class GradeBook
```

`GradeBook.cs`

(3 of 3)

Indicating the condition for the `while` statement to continue looping.

Indicating that the application has processed a grade and is ready to input the next grade from the user.

Calculating `average`.

Fig. 5.6 | `GradeBook` class that solves the class-average problem using counter-controlled repetition. (Part 3 of 3.)

Good Programming Practice 5.6

Separate declarations from other statements in methods with a blank line for readability.

5.8 Formulating Algorithms: Counter-Controlled Repetition (Cont.)

- A variable is **definitely assigned** when it is guaranteed to be assigned a value before it is used.

Common Programming Error 5.4

Using the value of a local variable before it is definitely assigned results in a compilation error. All local variables must be definitely assigned before their values are used in expressions.

Error-Prevention Tip 5.1

Initialize each counter and total, either in its declaration or in an assignment statement. Totals are normally initialized to 0. Counters are normally initialized to 0 or 1, depending on how they’re used (we’ll show examples of each).

- Class `GradeBookTest` (Fig. 5.7) tests an object of class `GradeBook`.

GradeBookTest.cs

(1 of 2)

```
// Fig. 5.7: GradeBookTest.cs
// Create GradeBook object and invoke its DetermineClassAverage method.
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.DetermineClassAverage(); // find average of 10 grades
   } // end Main
} // end class GradeBookTest
```

Calling `myGradeBook`’s `DetermineClassAverage` method to allow the user to enter 10 grades.

Fig. 5.7 | Create `GradeBook` object and invoke its `DetermineClassAverage` method. (Part 1 of 2.)

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

Enter grade: 88
Enter grade: 79
Enter grade: 95
Enter grade: 100
Enter grade: 48
Enter grade: 88
Enter grade: 92
Enter grade: 83
Enter grade: 90
Enter grade: 85

Total of all 10 grades is 848
Class average is 84
```

GradeBookTest.cs

(2 of 2)

Fig. 5.7 | Create `GradeBook` object and invoke its `DetermineClassAverage` method. (Part 2 of 2.)

5.8 Formulating Algorithms: Counter-Controlled Repetition (Cont.)

- Dividing two integers results in **integer division**.
- Any fractional part of the result is lost.

Common Programming Error 5.5

Assuming that integer division rounds can lead to incorrect results. For example, 7÷4, which yields 1.75 in conventional arithmetic, truncates to 1 in integer arithmetic, rather than rounding to 2.

5.9 Formulating Algorithms: Sentinel-Controlled Repetition

- Consider the following problem:

Develop a class-averaging application that processes grades for an arbitrary number of students each time it is run.

- In this example, no indication is given of how many grades the user will enter during the application's execution.

5.9 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

- A **sentinel value** can be entered to indicate "end of data entry." This is called **sentinel-controlled repetition**.
- Sentinel-controlled repetition is often called **indefinite repetition** because the number of repetitions is not known before the loop begins.

Common Programming Error 5.6

Choosing a sentinel value that is also a legitimate data value is a logic error.

5.9 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

- We approach the class-average application with a technique called **top-down, stepwise refinement**.
- The **top** conveys the overall function of the application: *determine the class average for the quiz*
- We divide the top into a series of smaller tasks in the **first refinement**:

initialize variables

input, sum and count the quiz grades

calculate and display the class average

5.9 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

Software Engineering Observation 5.2

Each refinement, as well as the top itself, is a complete specification of the algorithm—only the level of detail varies.

Software Engineering Observation 5.3

Many applications can be divided logically into three phases: an initialization phase that initializes the variables; a processing phase that inputs data values and adjusts application variables accordingly; and a termination phase that calculates and outputs the fi-nal results.

5.9 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

- The **second refinement** specifies individual variables.

initialize variables

- This can be refined as follows:

initialize total to zero

initialize counter to zero

5.9 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

input, sum and count the quiz grades

- The second refinement of the preceding pseudocode state-ment is:

prompt the user to enter the first grade

input the first grade (possibly the sentinel)

while the user has not yet entered the sentinel
    add this grade into the running total
    add one to the grade counter
    prompt the user to enter the next grade
    input the next grade (possibly the sentinel)

5.9 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

calculate and display the class average

- The preceding pseudocode can be refined as follows:

if the counter is not equal to zero
  set the average to the total divided by the counter
  display the average
else
  display "No grades were entered"

- We are careful here to test for the possibility of division by zero.

Error-Prevention Tip 5.2

When performing division by an expression whose value could be zero, explicitly test for this possibility and handle it appropriately in your application (e.g., by displaying an error mes-sage) rather than allowing the error to occur.

5.9 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

```
initialize total to zero
initialize counter to zero

prompt the user to enter the first grade
input the first grade (possibly the sentinel)

while the user has not yet entered the sentinel
    add this grade into the running total
    add one to the grade counter
    prompt the user to enter the next grade
    input the next grade (possibly the sentinel)

if the counter is not equal to zero
    set the average to the total divided by the counter
    display the average
else
     display "No grades were entered"
```

Fig. 5.8 | Class-average problem pseudocode algorithm with sentinel-controlled

5.9 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

Software Engineering Observation 5.4

Terminate the top-down, stepwise refinement process when you have specified the pseu-docode algorithm in sufficient detail for you to convert the pseudocode to C#.

Software Engineering Observation 5.5

Some experienced programmers write applications without ever using application-development tools like pseudocode. Although this method may work for simple and familiar problems, it can lead to serious errors and delays in large, complex projects.

- Class **GradeBook** now implements the algorithm for sentinel-controlled repetition (Fig. 5.9)

GradeBook.cs


```
// Fig. 5.9: GradeBook.cs
// GradeBook class that solves class-average problem using
// sentinel-controlled repetition.
using System;

public class GradeBook
{
   // autoimplemented property CourseName
   public string CourseName { get; set; }

   // constructor initializes the CourseName property
   public GradeBook( string name )
   {
      CourseName = name; // set CourseName to name
   } // end constructor
```

Fig. 5.9 | GradeBook class that solves the class-average problem using sentinel-controlled repetition. (Part 1 of 4.)

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

   // determine the average of an arbitrary number of grades
   public void DetermineClassAverage()
   {
      int total; // sum of grades
      int gradeCounter; // number of grades entered
      int grade; // grade value
      double average; // number with decimal point for average

      // initialization phase
      total = 0; // initialize total
      gradeCounter = 0; // initialize loop counter

```

`average` is of type `double` so it can store a floating-point number

Initializing the counter variable to `0`.

Fig. 5.9 | `GradeBook` class that solves the class-average problem using sentinel-controlled repetition. (Part 2 of 4.)

GradeBook.cs

(3 of 4)

```
      // processing phase
      // prompt for and read a grade from the user
      Console.Write( "Enter grade or -1 to quit: " );
      grade = Convert.ToInt32( Console.ReadLine() );

      // loop until sentinel value is read from the user
      while ( grade != -1 )
      {
         total = total + grade; // add grade to total
         gradeCounter = gradeCounter + 1; // increment counter

         // prompt for and read the next grade from the user
         Console.Write( "Enter grade or -1 to quit: " );
         grade = Convert.ToInt32( Console.ReadLine() );
      } // end while

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

Reading the first value before entering the `while`.

Obtaining the next value from the user before determining whether to repeat.

Using a cast operator to use a floating-point copy of `total`.

Fig. 5.9 | `GradeBook` class that solves the class-average problem using sentinel-controlled repetition. (Part 3 of 4.)

```

         // display the total and average (with two digits of precision)
         Console.WriteLine( "\nTotal of the {0} grades entered is {1}",
            gradeCounter, total );
         Console.WriteLine( "Class average is {0:F}", average );
      } // end if
      else // no grades were entered, so output error message
         Console.WriteLine( "No grades were entered" );
   } // end method DetermineClassAverage
} // end class GradeBook
```

`GradeBook.cs`

(4 of 4)

Outputting the class average rounded to the nearest hundredth.

Fig. 5.9 | `GradeBook` class that solves the class-average problem using sentinel-controlled repetition. (Part 4 of 4.)

Good Programming Practice 5.7

In a sentinel-controlled loop, the prompts requesting data entry should explicitly remind the user of the sentinel value.

Error-Prevention Tip 5.3

Omitting the braces that delimit a block can lead to logic errors, such as infinite loops. To prevent this problem, some programmers enclose the body of every control statement in braces even if the body contains only a single statement.

5.9 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

- To perform a floating-point calculation with integer values, we temporarily treat these values as floating-point Anumbers.
- A **unary cast operator** such as **`(double)`** performs **explicit conversion**.
- C# performs an operation called **promotion** (or **implicit conversion**) on selected operands for use in the expression.

Common Programming Error 5.7

A cast operator can be used to convert between simple numeric types, such as `int` and `double`, and between related reference types. Casting to the wrong type may cause compilation or runtime errors.

Outline

GradeBookTest.cs

```
// Fig. 5.10: GradeBookTest.cs
// Create GradeBook object and invoke its DetermineClassAverage method.
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.DetermineClassAverage(); // find average of grades
   } // end Main
} // end class GradeBookTest
```

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

Enter grade or -1 to quit: 96
Enter grade or -1 to quit: 88
Enter grade or -1 to quit: 79
Enter grade or -1 to quit: -1

Total of the 3 grades entered is 263
Class average is 87.67
```

Fig. 5.10 | Create `GradeBook` object and invoke `DetermineClassAverage` method.

5.10 Formulating Algorithms: Nested Control Statements

- Consider the following problem statement:

A college offers a course that prepares students for a state licensing exam. You've been given a list of 10 students. Next to each name is written a 1 if the student passed the exam or a 2 if the student failed.

1. *Count the number of test results of each type.*
2. *Display a summary of the test results.*
3. *If more than eight students passed the exam, display "Raise tuition."*

5.10 Formulating Algorithms: Nested Control Statements (Cont.)

- Proceed with top-down, stepwise refinement:

analyze exam results and decide whether tuition should be raised

- Our first refinement is

initialize variables

input the 10 exam results, and count passes and failures display a summary of the exam results and decide whether tuition should be raised

5.10 Formulating Algorithms: Nested Control Statements (Cont.)

initialize variables

- The preceding pseudocode statement can be refined as follows:

initialize passes to zero

initialize failures to zero

initialize student counter to one

- The next pseudocode statement is more complex.

input the 10 exam results, and count passes and failures

5.10 Formulating Algorithms: Nested Control Statements (Cont.)

- Inside the loop, a double-selection statement will determine whether each exam result is a pass or a failure:

while student counter is less than or equal to 10
    prompt the user to enter the next exam result
    input the next exam result

    if the student passed
        add one to passes
    else
        add one to failures

    add one to student counter

5.10 Formulating Algorithms: Nested Control Statements (Cont.)

- The pseudocode statement

display a summary of the exam results and decide whether tuition should be raised

- can be refined as follows:

display the number of passes

display the number of failures

if more than eight students passed

    display "Raise tuition"

5.5 Formulating Algorithms: Counter-Controlled Repetition (Cont.)

- The complete second refinement of the pseudocode appears in Fig. 5.11.

```
initialize passes to zero
initialize failures to zero
initialize student counter to one

while student counter is less than or equal to 10
    prompt the user to enter the next exam result
    input the next exam result

    if the student passed
        add one to passes
    else
        add one to failures

    add one to student counter

display the number of passes
display the number of failures

if more than eight students passed
    display "Raise tuition"
```

Fig. 5.11 | Pseudocode for the examination-results problem.

- The class that implements the algorithm is shown in Fig. 5.12.

Analysis.cs

(1 of 3)

```
// Fig. 5.12: Analysis.cs
// Analysis of examination results, using nested control statements.
using System;

public class Analysis
{
   public void ProcessExamResults()
   {
      // initialize variables in declarations
      int passes = 0; // number of passes
      int failures = 0; // number of failures
      int studentCounter = 1; // student counter
      int result; // one exam result from user

      // process 10 students using counter-controlled repetition
      while ( studentCounter <= 10 )
```

Declaring the variables used to process the exam results.

Fig. 5.12 | Analysis of examination results, using nested control statements. (Part 1 of 3.)

```
      {
         // prompt user for input and obtain a value from the user
         Console.Write( "Enter result (1 = pass, 2 = fail): " );
         result = Convert.ToInt32( Console.ReadLine() );

         // if...else nested in while
         if ( result == 1 ) // if result 1,
            passes = passes + 1; // increment passes
         else // else result is not 1, so
            failures = failures + 1; // increment failures

         // increment studentCounter so loop eventually terminates
         studentCounter = studentCounter + 1;
      } // end while
```

The `if…else` statement processes each result to increment `passes` or `failures`.

Fig. 5.12 | Analysis of examination results, using nested control statements. (Part 2 of 3.)

```

      // termination phase; prepare and display results
      Console.WriteLine( "Passed: {0}\nFailed: {1}", passes, failures );

      // determine whether more than 8 students passed
      if ( passes > 8 )
         Console.WriteLine( "Raise Tuition" );
   } // end method ProcessExamResults
} // end class Analysis
```

Fig. 5.12 | Analysis of examination results, using nested control statements. (Part 3 of 3.)

Error-Prevention Tip 5.4

Initializing local variables when they’re declared helps you avoid compilation errors that might arise from attempts to use uninitialized data. C# requires that local variables be initialized before their values are used in an expression.

Outline

- Class `AnalysisTest` (Fig. 5.13) processes a set of exam results entered by the user.

`AnalysisTest.cs`

(1 of 2)

```
// Fig. 5.13: AnalysisTest.cs
// Test application for class Analysis.
public class AnalysisTest
{
   public static void Main( string[] args )
   {
      Analysis application = new Analysis(); // create Analysis object
      application.ProcessExamResults(); // call method to process results
   } // end Main
} // end class AnalysisTest
```

Creating an `Analysis` object for grade processing.

Entering and processing grades.

Fig. 5.13 | Test application for class `Analysis`. (Part 1 of 2.)

Outline

AnalysisTest.cs

(2 of 2)

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 9
Failed: 1
Raise Tuition
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 2
Passed: 5
Failed: 5
```

Fig. 5.13 | Test application for class `Analysis`. (Part 2 of 2.)

5.11 Compound Assignment Operators

- C# provides several **compound assignment operators**
- For example, you can abbreviate the statement

```
c = c + 3;
```

- with the **addition compound assignment operator**, +=, as

```
c += 3;
```

5.11 Compound Assignment Operators (Cont.)

- Figure 5.14 explains the arithmetic compound assignment operators.

Assignment operator	Sample expression	Explanation	Assigns
Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	`10` to `c`
`-=`	`d -= 4`	`d = d - 4`	`1` to `d`
`*=`	`e *= 5`	`e = e * 5`	`20` to `e`
`/=`	`f /= 3`	`f = f / 3`	`2` to `f`
`%=`	`g %= 9`	`g = g % 9`	`3` to `g`

Fig. 5.14 | Arithmetic compound assignment operators.

5.12 Increment and Decrement Operators

- C# provides operators for adding or subtracting 1 from a numeric variable (Fig. 5.15).
  - The unary **increment operator**, **++**
  - The unary **decrement operator**, **--**.

Operator	Called	Sample expression	Explanation
`++`	prefix increment	`++a`	Increments a by `1`, then uses the new value of `a` in the expression.
`++`	postfix increment	`a++`	Uses the current value of `a`, then increments `a` by `1`.
`--`	prefix decrement	`--b`	Decrements `b` by `1`, then uses the new value of `b`.
`--`	postfix decrement	`b--`	Uses the current value of `b`, then decrements `b` by `1`.

Fig. 5.15 | Increment and decrement operators.

5.12 Increment and Decrement Operators (Cont.)

Good Programming Practice 5.8

Unlike binary operators, the unary increment and decrement operators should (by convention) be placed next to their operands, with no intervening spaces.

- Figure 5.16 demonstrates the difference between the prefix increment and postfix increment versions of the `++` increment operator.


```
// Fig. 5.16: Increment.cs
// Prefix increment and postfix increment operators.
using System;

public class Increment
{
   public static void Main( string[] args )
   {
      int c;

      // demonstrate postfix increment operator
      c = 5; // assign 5 to c
```

Fig. 5.16 | Prefix increment and postfix increment operators. (Part 1 of 2.)

Increment.cs


```
      Console.WriteLine( c ); // display 5
      Console.WriteLine( c++ ); // display 5 again, then increment
      Console.WriteLine( c ); // display 6

      Console.WriteLine(); // skip a line

      // demonstrate prefix increment operator
      c = 5; // assign 5 to c
      Console.WriteLine( c ); // display 5
      Console.WriteLine( ++c ); // increment, then display 6
      Console.WriteLine( c ); // display 6 again
   } // end Main
} // end class Increment
```

```
5
5
6

5
6
6
```

Outputting the value before `c`'s value is incremented.

Outputting the value after `c`'s value is incremented.

Fig. 5.16 | Prefix increment and postfix increment operators. (Part 2 of 2.)

5.12 Increment and Decrement Operators (Cont.)

```
passes = passes + 1;
failures = failures + 1;
studentCounter = studentCounter + 1;
```

- This can be written more concisely with compound assignment operators:

```
passes += 1;
failures += 1;
studentCounter += 1;
```

5.12 Increment and Decrement Operators (Cont.)

- This is even more precise with prefix increment operators:

```
++passes;
++failures;
++studentCounter;
```

- Or with postfix increment operators:

```
passes++;
failures++;
studentCounter++;
```

5.12 Increment and Decrement Operators (Cont.)

Common Programming Error 5.8

Attempting to use the increment or decrement operator on an expression other than one to which a value can be assigned is a syntax error. For example, writing `++(x + 1)` is a syntax error, because `(x + 1)` is not a variable.

5.14 (Optional) Software Engineering Case Study: Identifying Class Attributes in the ATM System (Cont.)

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
`?:`	right to left	conditional
`=` `+=` `-=` `*=` `/=` `%=`	right to left	assignment

Fig. 5.17 | Precedence and associativity of the operators discussed so far.

5.13 Simple Types

- The table in Appendix B, Simple Types, lists the 13 **simple types** in C#.
- C# requires all variables to have a type.
- Instance variables of types `char`, `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, and `decimal` are all given the value `0` by default.
- Instance variables of type `bool` are given the value `false` by default.

5.14 Identifying Class Attributes

- A person’s attributes include height, weight and whether the person is left-handed, right-handed or ambidextrous.
- We can identify attributes of the classes in our system by looking for descriptive words and phrases in the requirements document.

5.14 Identifying Class Attributes in the ATM System

Class	Descriptive words and phrases
`ATM`	user is authenticated
`BalanceInquiry`	account number
`Withdrawal`	account number amount
`Deposit`	account number amount
`BankDatabase`	[no descriptive words or phrases]
`Account`	account number PIN balance
`Screen`	[no descriptive words or phrases]
`Keypad`	[no descriptive words or phrases]
`CashDispenser`	begins each day loaded with 500 $20 bills
`DepositSlot`	[no descriptive words or phrases]

Fig. 5.18 | Descriptive words and phrases from the ATM requirements document.

5.14 Identifying Class Attributes in the ATM System (Cont.)

- The class diagram in Fig. 5.19 lists some of the attributes for the classes in our system.
- We list each attribute’s name and type, followed in some cases by an initial value.

Fig. 5.19 | Classes with attributes.

5.14 Identifying Class Attributes in the ATM System (Cont.)

- Consider the `userAuthenticated` attribute of class `ATM`:

`userAuthenticated : bool = false`

- This attribute declaration contains:
  - the **attribute name**, `userAuthenticated`.
  - the **attribute type**, `bool`.
  - an initial value for an attribute, `false`.

Software Engineering Observation 5.6

Early in the design process, classes often lack attributes (and operations). Such classes should not be eliminated, however, because attributes (and operations) may become evident in the later phases of design and implementation.