# Lesson 4
# Control Statements: Part I

Assoc. Prof. Dr. Marenglen Biba

## OBJECTIVES

In this Chapter you'll learn:

- To use basic problem-solving techniques.
- To develop algorithms through the process of top-down, stepwise refinement using pseudocode.
- To use the `if` and `if...else` selection statements to choose among alternative actions.
- To use the `while` repetition statement to execute statements in a program repeatedly.
- To use counter-controlled repetition and sentinel-controlled repetition.
- To use the compound assignment, increment and decrement operators.
- The portability of primitive data types.

# 4.1 Introduction

- Before writing a program to solve a problem, have a thorough understanding of the problem and a carefully planned approach to solving it.
- Understand the types of building blocks that are available and employ proven program-construction techniques.
- This chapter introduces
  - The `if`, `if`…`else` and `while` statements
  - Compound assignment, increment and decrement operators
  - Portability of Java's primitive types

# 4.2 Algorithms

- Any computing problem can be solved 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 execute and
  - the order in which these actions execute
- The "rise-and-shine algorithm" followed by one 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; (6) carpool to work.
- Suppose that the same steps are performed 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.
- Specifying the order in which statements (actions) execute in a program is called program control.

# 4.3 Pseudocode

- Pseudocode is an informal language that helps you develop algorithms without having to worry about the strict details of Java language syntax.
- Particularly useful for developing algorithms that will be converted to structured portions of Java programs.
- Similar to everyday English.
- Helps you "think out" a program before attempting to write it in a programming language, such as Java.
- You can type pseudocode conveniently, using any text-editor program.
- Carefully prepared pseudocode can easily be converted to a corresponding Java program.
- Pseudocode normally describes only statements representing the actions that occur after you convert a program from pseudocode to Java and the program is run on a computer.
  - e.g., input, output or calculations.

# 4.4 Control Structures

- Sequential execution: Statements in a program execute one after the other in the order in which they are written.
- Transfer of control: Various Java statements, enable you to specify that the next statement to execute is not necessarily the next one in sequence.
- Bohm and Jacopini
  - Demonstrated that programs could be written *without any* `goto` *statements.*
  - All programs can be written in terms of only three control structures—the sequence structure, the selection structure and the repetition structure.
- When we introduce Java's control structure implementations, we'll refer to them in the terminology of the *Java Language Specification as "control statements."*

# 4.4 Control Structures (Cont.)

- Sequence structure
  - Built into Java.
  - Unless directed otherwise, the computer executes Java statements one after the other in the order in which they're written.
  - The activity diagram in Fig. 4.1 illustrates a typical sequence structure in which two calculations are performed in order.
  - Java lets you have as many actions as you want in a sequence structure.
  - Anywhere a single action may be placed, we may place several actions in sequence.

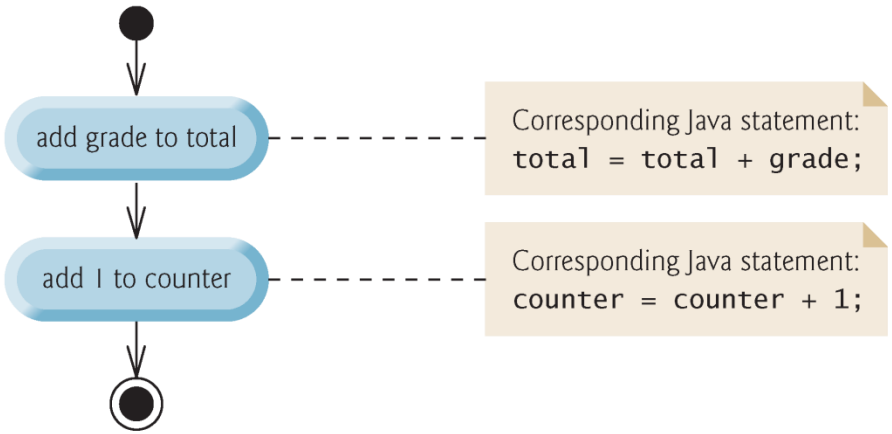


**Fig. 4.1** | Sequence structure activity diagram.

# 4.4 Control Structures (Cont.)

- Three types of selection statements.
- `if` statement:
  - Performs an action, if a condition is true; skips it, if false.
  - Single-selection statement—selects or ignores a single action (or group of actions).
- `if...else` statement:
  - Performs an action if a condition is true and performs a different action if the condition is false.
  - Double-selection statement—selects between two different actions (or groups of actions).
- `switch` statement
  - Performs one of several actions, based on the value of an expression.
  - Multiple-selection statement—selects among many different actions (or groups of actions).

# 4.4 Control Structures (Cont.)

- Three repetition statements (also called looping statements)
  - Perform statements repeatedly while a loop-continuation condition remains true.
- `while` and `for` statements perform the action(s) in their bodies zero or more times
  - if the loop-continuation condition is initially false, the body will not execute.
- The `do`…`while` statement performs the action(s) in its body one or more times.
- `if`, `else`, `switch`, `while`, `do` and `for` are keywords.
  - Appendix C: Complete list of Java keywords.

# 4.5 `if` Single-Selection Statement

- Pseudocode

  *If student's grade is greater than or equal to 60*
  *Print "Passed"*

- If the condition is false, the Print statement is ignored, and the next pseudocode statement in order is performed.
- Indentation
  - Optional, but recommended
  - Emphasizes the inherent structure of structured programs
- The preceding pseudocode *If* in Java:

```
if ( studentGrade >= 60 )
   System.out.println( "Passed" );
```

- Corresponds closely to the pseudocode.

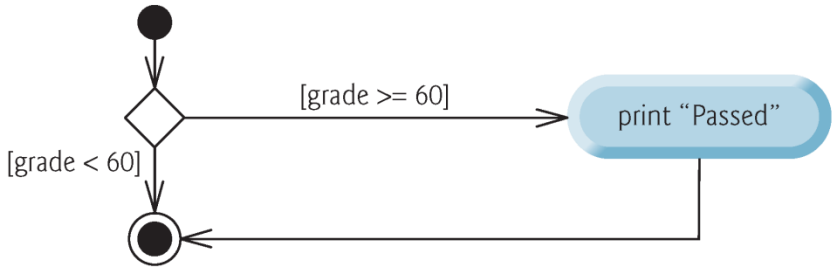


**Fig. 4.2** | `if` single-selection statement UML activity diagram.

# 4.6 `if...else` Double-Selection Statement

- `if...else` double-selection statement—specify an action to perform when the condition is true and a different action when the condition is false.
- Pseudocode

  *If student's grade is greater than or equal to 60*
  *Print "Passed"*
  *Else*
  *Print "Failed"*

- The preceding *If...Else pseudocode statement in Java:*

```
if ( grade >= 60 )
   System.out.println( "Passed" );
else
   System.out.println( "Failed" );
```

- Note that the body of the `else` is also indented.

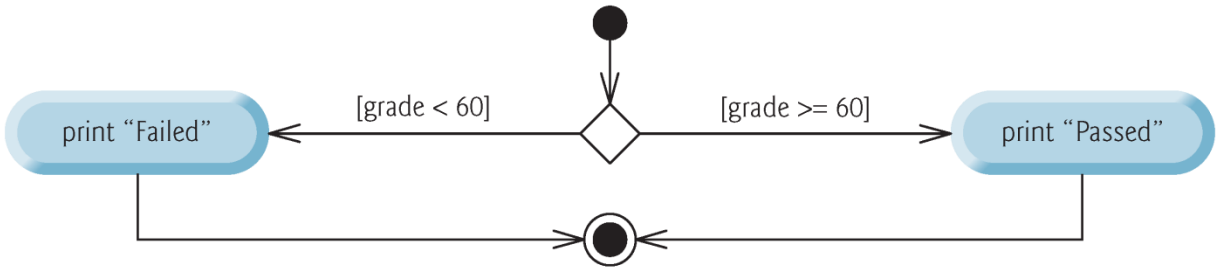


**Fig. 4.3** | `if`...`else` double-selection statement UML activity diagram.

# 4.6 `if...else` Double-Selection Statement (Cont.)

- Conditional operator (`?:`)—shorthand `if...else`.
- Ternary operator (takes three operands)
- Operands and `?:` form a conditional expression
- Operand to the left of the `?` is a `boolean` expression—evaluates to a `boolean` value (`true` or `false`)
- Second operand (between the `?` and `:`) is the value if the `boolean` expression is `true`
- Third operand (to the right of the `:`) is the value if the `boolean` expression evaluates to `false`.
- Example:

```
System.out.println(
   studentGrade >= 60 ? "Passed" : "Failed" );
```

- Evaluates to the string `"Passed"` if the `boolean` expression `studentGrade >= 60` is true and to the string `"Failed"` if it is false.

# 4.6 `if...else` Double-Selection Statement (Cont.)

- Can test multiple cases by placing `if...else` statements inside other `if...else` statements to create nested *`if...else` statements*.
- Pseudocode:

```
If student's grade is greater than or equal to 90
   Print "A"
else
   If student's grade is greater than or equal to 80
      Print "B"
   else
      If student's grade is greater than or equal to 70
         Print "C"
      else
         If student's grade is greater than or equal to 60
            Print "D"
         else
            Print "F"
```

# 4.6 `if...else` Double-Selection Statement (Cont.)

- This pseudocode may be written in Java as

```
if ( studentGrade >= 90 )
   System.out.println( "A" );
else
   if ( studentGrade >= 80 )
      System.out.println( "B" );
   else
      if ( studentGrade >= 70 )
         System.out.println( "C" );
      else
         if ( studentGrade >= 60 )
            System.out.println( "D" );
         else
            System.out.println( "F" );
```

- If `studentGrade` >= 90, the first four conditions will be true, but only the statement in the `if` part of the first `if`…`else` statement will execute. After that, the `else` part of the "outermost" `if`…`else` statement is skipped.

# 4.6 `if...else` Double-Selection Statement (Cont.)

- Most Java programmers prefer to write the preceding nested `if...else` statement as

```
if ( studentGrade >= 90 )
   System.out.println( "A" );
else if ( studentGrade >= 80 )
   System.out.println( "B" );
else if ( studentGrade >= 70 )
   System.out.println( "C" );
else if ( studentGrade >= 60 )
   System.out.println( "D" );
else
   System.out.println( "F" );
```

- The two forms are identical except for the spacing and indentation, which the compiler ignores.

# 4.6 `if...else` Double-Selection Statement (Cont.)

- The Java compiler always associates an `else` with the immediately preceding `if` unless told to do otherwise by the placement of braces (`{` and `}`).
- Referred to as the dangling-`else` problem.
- The following code is not what it appears:

```
if ( x > 5 )
   if ( y > 5 )
      System.out.println( "x and y are > 5" );
else
   System.out.println( "x is <= 5" );
```

- Beware! This nested `if...else` statement does not execute as it appears: x maybe > than 5!

# 4.6 `if...else` Double-Selection Statement (Cont.)

- To force the nested `if...else` statement to execute as it was originally intended, we must write it as follows:

```
if ( x > 5 )
{
   if ( y > 5 )
      System.out.println( "x and y are > 5" );
}
else
   System.out.println( "x is <= 5" );
```

- The braces indicate that the second `if` is in the body of the first and that the `else` is associated with the *first* `if`.

# 4.6 `if...else` Double-Selection Statement (Cont.)

- The `if` statement normally expects only one statement in its body.
- To include several statements in the body of an `if` (or the body of an `else` for an `if...else` statement), enclose the statements in braces.
- Statements contained in a pair of braces form a block.
- A block can be placed anywhere that a single statement can be placed.

- Example: A block in the `else` part of an `if...else` statement:

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

# 4.7 while Repetition Statement

- Repetition statement—repeats an action while a condition remains true.
- Pseudocode

  *While there are more items on my shopping list*
  *Purchase next item and cross it off my list*

- The repetition statement's body may be a single statement or a block.
- Eventually, the condition will become false. At this point, the repetition terminates, and the first statement after the repetition statement executes.

# 4.7 while Repetition Statement (Cont.)

- Example of Java's `while` repetition statement: find the first power of 3 larger than 100. Assume `int` variable `product` is initialized to `3`.

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

- Each iteration multiplies `product` by 3, so `product` takes on the values 9, 27, 81 and 243 successively.
- When variable `product` becomes 243, the `while`-statement condition—`product <= 100`—becomes false.
- Repetition terminates. The final value of `product` is 243.
- Program execution continues with the next statement after the `while` statement.

## Common Programming Error 4.3

*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*** *(the loop never terminates).*

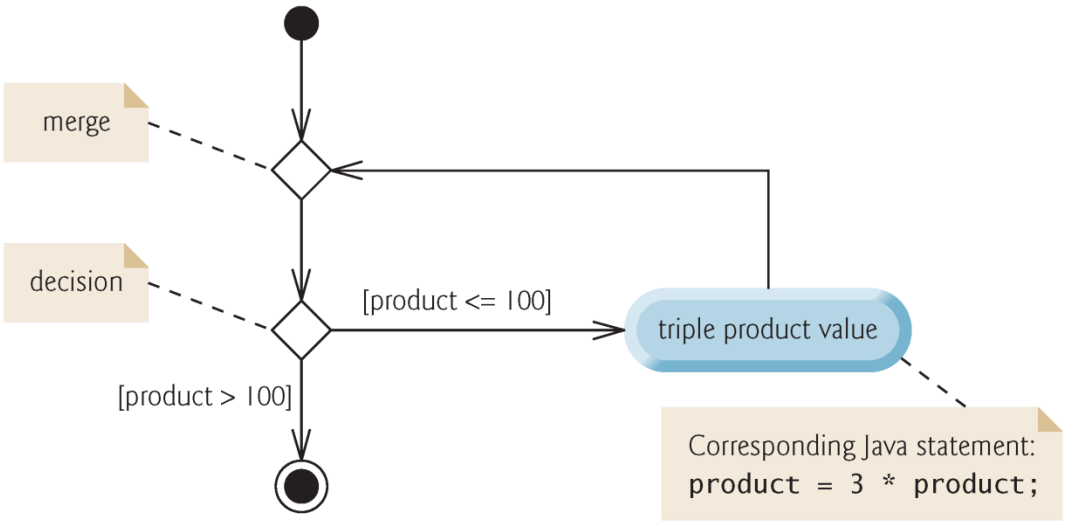


**Fig. 4.4** | `while` repetition statement UML activity diagram.

# 4.8 Formulating Algorithms: Counter-Controlled Repetition

- *A class of ten 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 class average is equal to the sum of the grades divided by the number of students.
- The algorithm for solving this problem on a computer must input each grade, keep track of the total of all grades input, perform the averaging calculation and print the result.
- Use counter-controlled repetition to input the grades one at a time.
- A variable called a counter (or control variable) 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.

# 4.8 Formulating Algorithms: Counter-Controlled Repetition (Cont.)

- A total is a variable used to accumulate the sum of several values.
- A counter is a variable used to count.
- Variables used to store totals are normally initialized to zero before being used in a program.

```
1  Set total to zero
2  Set grade counter to one
3
4  While grade counter is less than or equal to ten
5      Prompt the user to enter the next grade
6      Input the next grade
7      Add the grade into the total
8      Add one to the grade counter
9
10 Set the class average to the total divided by ten
11 Print the class average
```

**Fig. 4.5** | Pseudocode algorithm that uses counter-controlled repetition to solve the class-average problem.

```
1  // Fig. 4.6: GradeBook.java
2  // GradeBook class that solves class-average problem using
3  // counter-controlled repetition.
4  import java.util.Scanner; // program uses class Scanner
5
6  public class GradeBook
7  {
8     private String courseName; // name of course this GradeBook represents
9
10    // constructor initializes courseName
11    public GradeBook( String name )
12    {
13       courseName = name; // initializes courseName
14    } // end constructor
15
16    // method to set the course name
17    public void setCourseName( String name )
18    {
19       courseName = name; // store the course name
20    } // end method setCourseName
21
```

**Fig. 4.6** | Counter-controlled repetition: class-average problem. (Part 1 of 3.)

```
22      // method to retrieve the course name
23      public String getCourseName()
24      {
25         return courseName;
26      } // end method getCourseName
27
28      // display a welcome message to the GradeBook user
29      public void displayMessage()
30      {
31         // getCourseName gets the name of the course
32         System.out.printf( "Welcome to the grade book for\n%s!\n\n",
33            getCourseName() );
34      } // end method displayMessage
35
36      // determine class average based on 10 grades entered by user
37      public void determineClassAverage()
38      {
39         // create Scanner to obtain input from command window
40         Scanner input = new Scanner( System.in );
41
42         int total; // sum of grades entered by user
43         int gradeCounter; // number of the grade to be entered next
44         int grade; // grade value entered by user
45         int average; // average of grades
```

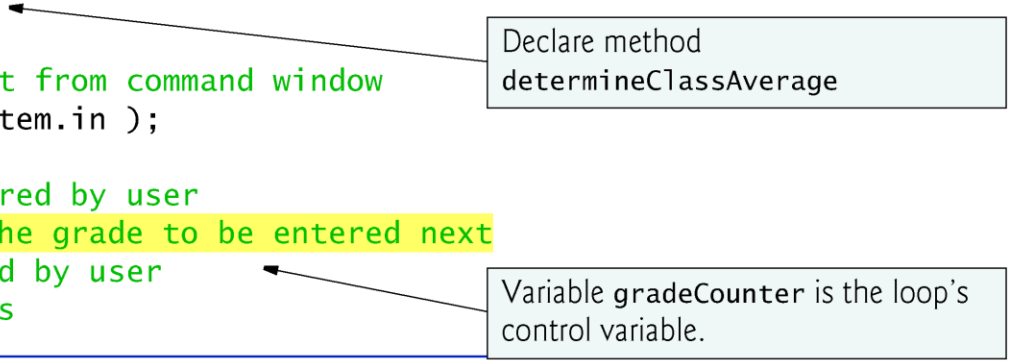


**Fig. 4.6** | Counter-controlled repetition: class-average problem. (Part 2 of 3.)

```
46
47          // initialization phase
48          total = 0; // initialize total
49          gradeCounter = 1; // initialize loop counter
50
51          // processing phase
52          while ( gradeCounter <= 10 ) // loop 10 times
53          {
54             System.out.print( "Enter grade: " ); // prompt
55             grade = input.nextInt(); // input next grade
56             total = total + grade; // add grade to total
57             gradeCounter = gradeCounter + 1; // increment counter by 1
58          } // end while
59
60          // termination phase
61          average = total / 10; // integer division yields integer result
62
63          // display total and average of grades
64          System.out.printf( "\nTotal of all 10 grades is %d\n", total );
65          System.out.printf( "Class average is %d\n", average );
66       } // end method determineClassAverage
67    } // end class GradeBook
```

Initializes `gradeCounter` to `1`; indicates first grade about to be input

Increments `gradeCounter`

Calculates average with integer arithmetic

**Fig. 4.6** | Counter-controlled repetition: class-average problem. (Part 3 of 3.)

```
1  // Fig. 4.7: GradeBookTest.java
2  // Create GradeBook object and invoke its determineClassAverage method.
3
4  public class GradeBookTest
5  {
6     public static void main( String[] args )
7     {
8        // create GradeBook object myGradeBook and
9        // pass course name to constructor
10       GradeBook myGradeBook = new GradeBook(
11          "CS101 Introduction to Java Programming" );
12
13       myGradeBook.displayMessage(); // display welcome message
14       myGradeBook.determineClassAverage(); // find average of 10 grades
15    } // end main
16 } // end class GradeBookTest
```

**Fig. 4.7** | `GradeBookTest` class creates an object of class `GradeBook` (Fig. 4.6) and invokes its `determineClassAverage` method. (Part 1 of 2.)

```
Welcome to the grade book for
CS101 Introduction to Java Programming!

Enter grade: 67
Enter grade: 78
Enter grade: 89
Enter grade: 67
Enter grade: 87
Enter grade: 98
Enter grade: 93
Enter grade: 85
Enter grade: 82
Enter grade: 100

Total of all 10 grades is 846
Class average is 84
```

**Fig. 4.7** | `GradeBookTest` class creates an object of class `GradeBook` (Fig. 4.6) and invokes its `determineClassAverage` method. (Part 2 of 2.)

# 4.9 Formulating Algorithms: Sentinel-Controlled Repetition

- *Develop a class-averaging program that processes grades for an arbitrary number of students each time it is run.*
- Sentinel-controlled repetition is often called indefinite repetition because the number of repetitions is not known before the loop begins executing.
- A special value called a sentinel value (also called a signal value, a dummy value or a flag value) can be used to indicate “end of data entry.”
- A sentinel value must be chosen that cannot be confused with an acceptable input value.

# 4.9 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

- Top-down, stepwise refinement
- Begin with a pseudocode representation of the top—a single statement that conveys the overall function of the program:
  - *Determine the class average for the quiz*
- The top is a *complete representation of a program*. Rarely conveys sufficient detail from which to write a Java program.
- Divide the top into a series of smaller tasks and list these in the order in which they'll be performed.
- First refinement:
  - *Initialize variables*
    *Input, sum and count the quiz grades*
    *Calculate and print the class average*
- This refinement uses only the sequence structure—the steps listed should execute in order, one after the other.

# 4.9 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

- Second refinement: commit to specific variables.
- The pseudocode statement

  *Initialize variables*

- can be refined as follows:

  *Initialize total to zero*
  *Initialize counter to zero*

# 4.9 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

- The pseudocode statement

  *Input, sum and count the quiz grades*

- requires a repetition structure that successively inputs each grade.
- We do not know in advance how many grades are to be processed, so we'll use sentinel-controlled repetition.
- The second refinement of the preceding pseudocode statement is then

  *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)*

# 4.9 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

- The pseudocode statement

  *Calculate and print the class average*

- can be refined as follows:

  *If the counter is not equal to zero*
  *Set the average to the total divided by the counter*
  *Print the average*
  *else*
  *Print "No grades were entered"*

- Test for the possibility of division by zero—a logic error that, if undetected, would cause the program to fail or produce invalid output.

```
1  Initialize total to zero
2  Initialize counter to zero
3
4  Prompt the user to enter the first grade
5  Input the first grade (possibly the sentinel)
6
7  While the user has not yet entered the sentinel
8      Add this grade into the running total
9      Add one to the grade counter
10     Prompt the user to enter the next grade
11     Input the next grade (possibly the sentinel)
12
13 If the counter is not equal to zero
14     Set the average to the total divided by the counter
15     Print the average
16 else
17     Print "No grades were entered"
```

**Fig. 4.8** | Class-average problem pseudocode algorithm with sentinel-controlled repetition.

```
1  // Fig. 4.9: GradeBook.java
2  // GradeBook class that solves class-average program using
3  // sentinel-controlled repetition.
4  import java.util.Scanner; // program uses class Scanner
5
6  public class GradeBook
7  {
8     private String courseName; // name of course this GradeBook represents
9
10    // constructor initializes courseName
11    public GradeBook( String name )
12    {
13       courseName = name; // initializes courseName
14    } // end constructor
15
16    // method to set the course name
17    public void setCourseName( String name )
18    {
19       courseName = name; // store the course name
20    } // end method setCourseName
21
```

**Fig. 4.9** | Sentinel-controlled repetition: Class-average problem. (Part 1 of 4.)

```
22      // method to retrieve the course name
23      public String getCourseName()
24      {
25         return courseName;
26      } // end method getCourseName
27
28      // display a welcome message to the GradeBook user
29      public void displayMessage()
30      {
31         // getCourseName gets the name of the course
32         System.out.printf( "Welcome to the grade book for\n%s!\n\n",
33            getCourseName() );
34      } // end method displayMessage
35
36      // determine the average of an arbitrary number of grades
37      public void determineClassAverage()
38      {
39         // create Scanner to obtain input from command window
40         Scanner input = new Scanner( System.in );
41
42         int total; // sum of grades
43         int gradeCounter; // number of grades entered
44         int grade; // grade value
45         double average; // number with decimal point for average
```

Declare method `determineClassAverage`

Will calculate and store a floating-point average

**Fig. 4.9** | Sentinel-controlled repetition: Class-average problem. (Part 2 of 4.)

```
46
47          // initialization phase
48          total = 0; // initialize total
49          gradeCounter = 0; // initialize loop counter
50
51          // processing phase
52          // prompt for input and read grade from user
53          System.out.print( "Enter grade or -1 to quit: " );
54          grade = input.nextInt();
55
56          // loop until sentinel value read from user
57          while ( grade != -1 )
58          {
59             total = total + grade; // add grade to total
60             gradeCounter = gradeCounter + 1; // increment counter
61
62             // prompt for input and read next grade from user
63             System.out.print( "Enter grade or -1 to quit: " );
64             grade = input.nextInt();
65          } // end while
66
```

Initializes `gradeCounter` to 0; no grades have been input yet and first grade could be the sentinel value

Obtain first grade before the loop in sentinel-controlled repetition

Obtain subsequent grades at the end of the loop in sentinel-controlled repetition

**Fig. 4.9** | Sentinel-controlled repetition: Class-average problem. (Part 3 of 4.)

```
67          // termination phase
68          // if user entered at least one grade...
69          if ( gradeCounter != 0 )
70          {
71             // calculate average of all grades entered
72             average = (double) total / gradeCounter;
73
74             // display total and average (with two digits of precision)
75             System.out.printf( "\nTotal of the %d grades entered is %d\n",
76                gradeCounter, total );
77             System.out.printf( "Class average is %.2f\n", average );
78          } // end if
79          else // no grades were entered, so output appropriate message
80             System.out.println( "No grades were entered" );
81       } // end method determineClassAverage
82    } // end class GradeBook
```

Test for the possibility of division by zero

Use cast operator to force a floating-point average calculation

**Fig. 4.9** | Sentinel-controlled repetition: Class-average problem. (Part 4 of 4.)

# 4.9 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

- Integer division yields an integer result.
- To perform a floating-point calculation with integers, temporarily treat these values as floating-point numbers for use in the calculation.
- The unary cast operator `(double)` creates a temporary floating-point copy of its operand.
- Cast operator performs explicit conversion (or type cast).
- The value stored in the operand is unchanged.
- Promotion (or implicit conversion) performed on operands.
- In an expression containing values of the types `int` and `double`, the `int` values are promoted to `double` values for use in the expression.

# 4.9 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

- Cast operators are available for any type.
- Cast operator formed by placing parentheses around the name of a type.
- The operator is a unary operator (i.e., an operator that takes only one operand).

```
1  // Fig. 4.10: GradeBookTest.java
2  // Create GradeBook object and invoke its determineClassAverage method.
3
4  public class GradeBookTest
5  {
6     public static void main( String[] args )
7     {
8        // create GradeBook object myGradeBook and
9        // pass course name to constructor
10       GradeBook myGradeBook = new GradeBook(
11          "CS101 Introduction to Java Programming" );
12
13       myGradeBook.displayMessage(); // display welcome message
14       myGradeBook.determineClassAverage(); // find average of grades
15    } // end main
16 } // end class GradeBookTest
```

**Fig. 4.10** | `GradeBookTest` class creates an object of class `GradeBook` (Fig. 4.9) and invokes its `determineClassAverage` method. (Part 1 of 2.)

```
Welcome to the grade book for
CS101 Introduction to Java Programming!

Enter grade or -1 to quit: 97
Enter grade or -1 to quit: 88
Enter grade or -1 to quit: 72
Enter grade or -1 to quit: -1

Total of the 3 grades entered is 257
Class average is 85.67
```

**Fig. 4.10** | `GradeBookTest` class creates an object of class `GradeBook` (Fig. 4.9) and invokes its `determineClassAverage` method. (Part 2 of 2.)

# 4.10 Formulating Algorithms: Nested Control Statements

- This case study examines nesting one control statement within another.
- A college offers a course that prepares students for the state licensing exam for real estate brokers. Last year, ten of the students who completed this course took the exam. The college wants to know how well its students did on the exam. You've been asked to write a program to summarize the results. You've been given a list of these 10 students. Next to each name is written a 1 if the student passed the exam or a 2 if the student failed.

# 4.10 Formulating Algorithms: Nested Control Statements (Cont.)

- This case study examines nesting one control statement within another.
- Your program should analyze the results of the exam as follows:
  - Input each test result (i.e., a 1 or a 2). Display the message "Enter result" on the screen each time the program requests another test result.
  - Count the number of test results of each type.
  - Display a summary of the test results, indicating the number of students who passed and the number who failed.
  - If more than eight students passed the exam, print the message "Bonus to instructor!"

```
1  Initialize passes to zero
2  Initialize failures to zero
3  Initialize student counter to one
4
5  While student counter is less than or equal to 10
6      Prompt the user to enter the next exam result
7      Input the next exam result
8
9      If the student passed
10         Add one to passes
11     Else
12         Add one to failures
13
14     Add one to student counter
15
16 Print the number of passes
17 Print the number of failures
18
19 If more than eight students passed
20     Print "Bonus to instructor!"
```

**Fig. 4.11** | Pseudocode for examination-results problem.

```
1  // Fig. 4.12: Analysis.java
2  // Analysis of examination results.
3  import java.util.Scanner; // class uses class Scanner
4
5  public class Analysis
6  {
7     public static void main( String[] args )
8     {
9        // create Scanner to obtain input from command window
10       Scanner input = new Scanner( System.in );
11
12       // initializing variables in declarations
13       int passes = 0; // number of passes
14       int failures = 0; // number of failures
15       int studentCounter = 1; // student counter
16       int result; // one exam result (obtains value from user)
17
18       // process 10 students using counter-controlled loop
19       while ( studentCounter <= 10 )
20       {
21          // prompt user for input and obtain value from user
22          System.out.print( "Enter result (1 = pass, 2 = fail): " );
23          result = input.nextInt();
```

Declare and initialize counters for passes, failures and students

`while` repetition statement iterates 10 times

**Fig. 4.12** | Nested control structures: Examination-results problem. (Part 1 of 4.)

```
24
25          // if...else nested in while
26          if ( result == 1 )          // if result 1,
27             passes = passes + 1;     // increment passes;
28          else                        // else result is not 1, so
29             failures = failures + 1; // increment failures
30
31          // increment studentCounter so loop eventually terminates
32          studentCounter = studentCounter + 1;
33       } // end while
34
35       // termination phase; prepare and display results
36       System.out.printf( "Passed: %d\nFailed: %d\n", passes, failures );
37
38       // determine whether more than 8 students passed
39       if ( passes > 8 )
40          System.out.println( "Bonus to instructor!" );
41    } // end main
42 } // end class Analysis
```

Increment **passes** or **failures** based on user's input.

**Fig. 4.12** | Nested control structures: Examination-results problem. (Part 2 of 4.)

```
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
Bonus to instructor!
```

**Fig. 4.12** | Nested control structures: Examination-results problem. (Part 3 of 4.)

```
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): 2
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): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 6
Failed: 4
```

**Fig. 4.12** | Nested control structures: Examination-results problem. (Part 4 of 4.)

# 4.11 Compound Assignment Operators

- Compound assignment operators abbreviate assignment expressions.
- Statements like

  *variable* = *variable operator expression*;

  where operator is one of the binary operators +, -, *, / or % can be written in the form

  *variable operator= expression;*
- Example:

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

  can be written with the addition compound assignment operator, +=, as

```
c += 3;
```

- The += operator adds the value of the expression on its right to the value of the variable on its left and stores the result in the variable on the left of the operator.

| 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. 4.13** | Arithmetic compound assignment operators.

# 4.12 Increment and Decrement Operators

- Unary increment operator, `++`, adds one to its operand
- Unary decrement operator, `--`, subtracts one from its operand
- An increment or decrement operator that is prefixed to (placed before) a variable is referred to as the prefix increment or prefix decrement operator, respectively.
- An increment or decrement operator that is postfixed to (placed after) a variable is referred to as the postfix increment or postfix decrement operator, respectively.

| Operator | Operator name | Sample expression | Explanation |
|---|---|---|---|
| `++` | prefix increment | `++a` | Increment `a` by `1`, then use the new value of `a` in the expression in which `a` resides. |
| `++` | postfix increment | `a++` | Use the current value of `a` in the expression in which `a` resides, then increment `a` by `1`. |
| `--` | prefix decrement | `--b` | Decrement `b` by `1`, then use the new value of `b` in the expression in which `b` resides. |
| `--` | postfix decrement | `b--` | Use the current value of `b` in the expression in which `b` resides, then decrement `b` by `1`. |

**Fig. 4.14** | Increment and decrement operators.

```
1  // Fig. 4.15: Increment.java
2  // Prefix increment and postfix increment operators.
3
4  public class Increment
5  {
6     public static void main( String[] args )
7     {
8        int c;
9
10       // demonstrate postfix increment operator
11       c = 5; // assign 5 to c
12       System.out.println( c );   // prints 5
13       System.out.println( c++ ); // prints 5 then postincrements
14       System.out.println( c );   // prints 6
15
16       System.out.println(); // skip a line
17
18       // demonstrate prefix increment operator
19       c = 5; // assign 5 to c
20       System.out.println( c );   // prints 5
21       System.out.println( ++c ); // preincrements then prints 6
22       System.out.println( c );   // prints 6
23    } // end main
24 } // end class Increment
```

Uses current value, then increments c

Increments c then uses new value

**Fig. 4.15** | Preincrementing and postincrementing. (Part 1 of 2.)

```
5
5
6

5
6
6
```

**Fig. 4.15** | Preincrementing and postincrementing. (Part 2 of 2.)

# 4.13 Primitive Types

- Appendix D lists the eight primitive types in Java.
- Java requires all variables to have a type.
- Java is a strongly typed language.
- Primitive types in Java are portable across all platforms that support Java.
- Instance variables of types `char`, `byte`, `short`, `int`, `long`, `float` and `double` are all given the value `0` by default. Instance variables of type `boolean` are given the value `false` by default.
- Reference-type instance variables are initialized by default to the value `null`.

# End of Part I