Chapter 4
Control Statements: Part I; Assignment, ++ and -- Operators

Java How to Program, 11/e, Global Edition

Questions? E-mail paul.deitel@deitel.com

OBJECTIVES

In this chapter you'll:

- Learn basic problem-solving techniques.
- Develop algorithms through the process of top-down, stepwise refinement.
- Use the `if` and `if...else` selection statements to choose between alternative actions.

OBJECTIVES (cont.)

- Use the `while` iteration statement to execute statements in a program repeatedly.
- Use counter-controlled iteration and sentinel-controlled iteration.
- Use the compound assignment operator and the increment and decrement operators.
- Learn about the portability of primitive data types.

OUTLINE

OUTLINE (cont.)

OUTLINE (cont.)

OUTLINE (cont.)

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.
- In this chapter we discuss
  - Java's `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 iteration 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 in Java

- 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.

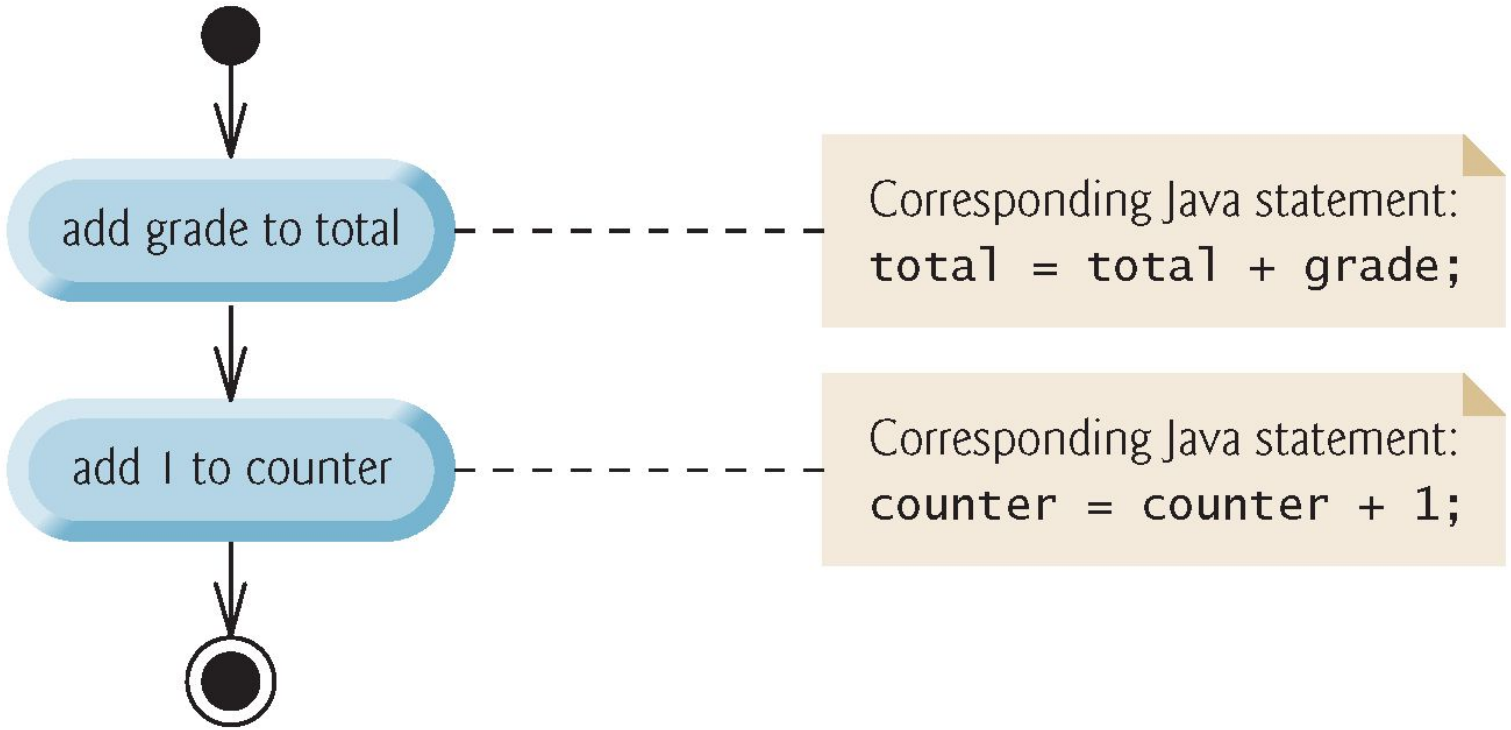


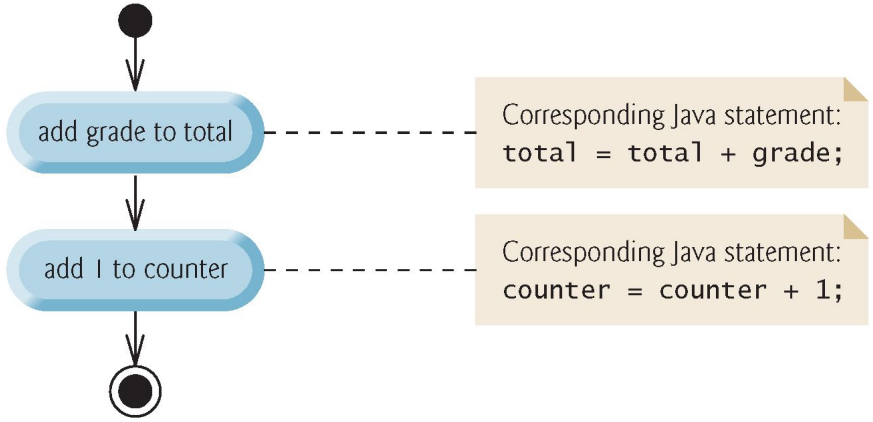
Fig. 4.1 | Sequence-structure activity diagram.

\

4.4 Control Structures (Cont.)

- UML activity diagram
- Models the workflow (also called the activity) of a portion of a software system.
- May include a portion of an algorithm, like the sequence structure in Fig. 4.1.
- Composed of symbols
  - action-state symbols (rectangles with their left and right sides replaced with outward arcs)
  - diamonds
  - small circles
- Symbols connected by transition arrows, which represent the flow of the activity—the order in which the actions should occur.
- Help you develop and represent algorithms.
- Clearly show how control structures operate.

4.4 Control Structures (Cont.)



- Sequence-structure activity diagram in Fig. 4.1.
- Two action states that represent actions to perform.
- Each contains an action expression that specifies a particular action to perform.
- Arrows represent transitions (order in which the actions represented by the action states occur).
- Solid circle at the top represents the initial state—the beginning of the workflow before the program performs the modeled actions.
- Solid circle surrounded by a hollow circle at the bottom represents the final state—the end of the workflow after the program performs its actions.

4.4 Control Structures (Cont.)

- UML notes
  - Like comments in Java.
  - Rectangles with the upper-right corners folded over.
  - Dotted line connects each note with the element it describes.
  - Activity diagrams normally do not show the corresponding Java code. We do this here to illustrate how the diagram relates to Java code.
- More information on the UML
  - see our optional case study (Chapters 33–34)
  - visit `www.uml.org`

4.4 Control Structures (Cont.)

Selection Statements in Java

- 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.)

Iteration Statements in Java

- Four iteration statements (also called iteration statements or 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.
- Chapter 7 presents the enhanced `for` statement.
- `if`, `else`, `switch`, `while`, `do` and `for` are keywords.
  - Appendix C: Complete list of Java keywords.

4.4 Control Structures (Cont.)

Summary of Control Statements in Java

- Every program is formed by combining the sequence statement, selection statements (four types) and iteration statements (three types) as appropriate for the algorithm the program implements.
- Can model each control statement as an activity diagram.
  - Initial state and a final state represent a control statement's entry point and exit point, respectively.
  - Single-entry/single-exit control statements
  - Control-statement stacking—connect the exit point of one to the entry point of the next.
  - Control-statement nesting—a control statement inside another.

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.

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

UML Activity Diagram for an `if` Statement

- Figure 4.2 `if` statement UML activity diagram.
- Diamond, or decision symbol, indicates that a decision is to be made.
- Workflow continues along a path determined by the symbol's guard conditions, which can be true or false.
- Each transition arrow emerging from a decision symbol has a guard condition (in square brackets next to the arrow).
- If a guard condition is true, the workflow enters the action state to which the transition arrow points.

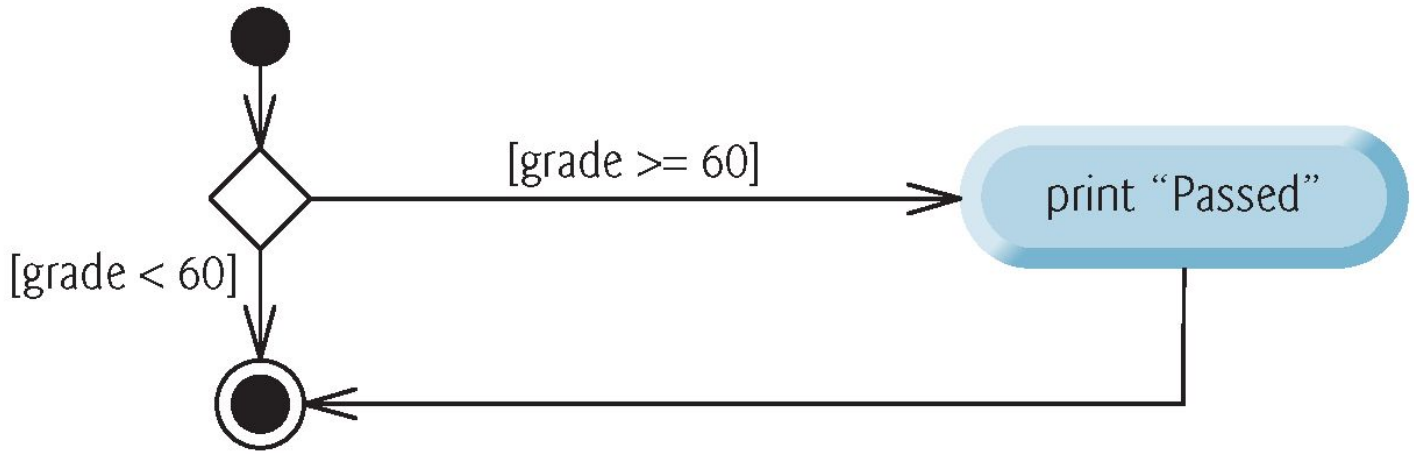


Fig. 4.2 | `if` single-selection statement UML activity diagram.

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.

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.

Good Programming Practice 4.1

Indent both body statements of an `if...else` statement. Most IDEs do this for you.

Good Programming Practice 4.2

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

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

UML Activity Diagram for an `if...else` Statement

- Figure 4.3 illustrates the flow of control in the `if...else` statement.
- The symbols in the UML activity diagram (besides the initial state, transition arrows and final state) represent action states and decisions.

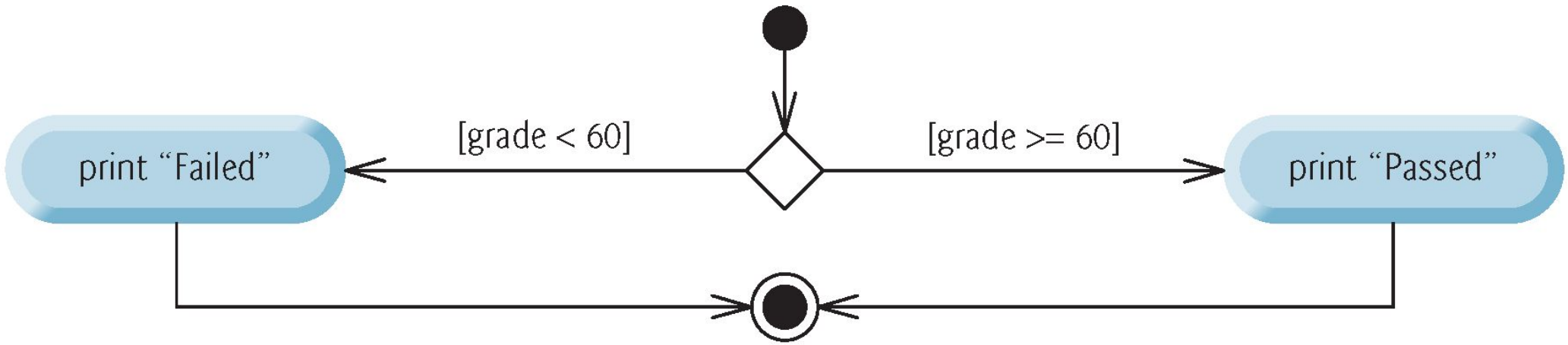


Fig. 4.3 | `if...else` double-selection statement UML activity diagram.

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

Nested `if...else` Statements

- A program 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"
```

Error-Prevention Tip 4.1

In a nested `if...else` statement, ensure that you test for all possible cases.

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");
         }
      }
   }
}
```

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

- 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");
}
```

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

Dangling-`else` Problem

- Throughout the text, we always enclose control statement bodies in braces ({ and }). This avoids a logic error called the "dangling-else" problem. We investigate this problem in –.

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

Blocks

- 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 (such as the body of a method) form a block.
- A block can be placed anywhere in a method 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.");
}
```

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

- *Syntax errors* (e.g., a missing brace) are caught by the compiler.
- A logic error (e.g., when both braces in a block are left out of the program) has its effect at execution time.
- A fatal logic error causes a program to fail and terminate prematurely.
- A nonfatal logic error allows a program to continue executing but causes it to produce incorrect results.

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

- Just as a block can be placed anywhere a single statement can be placed, it's also possible to have an empty statement.
- The empty statement is represented by placing a semicolon (`;`) where a statement would normally be.

Common Programming Error 4.1

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).

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

Conditional operator (`?:`)

- 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`.

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

- 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.

Error-Prevention Tip 4.2

Use expressions of the same type for the second and third operands of the `?:` operator to avoid subtle errors.

4.7 `Student` Class: Nested `if...else` Statement

Class `Student`

- Class `Student` (Fig. 4.4) stores a student's name and average and provides methods for manipulating these values.
- The class contains:
  - instance variable `name` of type `String` to store a `Student`'s name
  - instance variable `average` of type `double` to store a `Student`'s average in a course
  - a constructor that initializes the `name` and `average`
  - methods `setName` and `getName` to set and get the `Student`'s name
  - methods `setAverage` and `getAverage` to *set* and *get* the `Student`'s average
  - method `getLetterGrade`, which uses nested `if...else` statements to determine the `Student`'s letter grade based on the `Student`'s average

```
1  // Fig. 4.4: Student.java
2  // Student class that stores a student name and average.
3  public class Student {
4     private String name;
5     private double average;
6
7     // constructor initializes instance variables
8     public Student(String name, double average) {
9        this.name = name;
10
11       // validate that average is > 0.0 and <= 100.0; otherwise,
12       // keep instance variable average's default value (0.0)
13       if (average > 0.0) {
14          if (average <= 100.0) {
15             this.average = average; // assign to instance variable
16          }
17       }
18    }
```

Fig. 4.4 | `Student` class that stores a student name and average. (Part 1 of 4.)

```
19
20      // sets the Student's name
21      public void setName(String name) {
22         this.name = name;
23      }
24
25      // retrieves the Student's name
26      public String getName() {
27         return name;
28      }
29
```

Fig. 4.4 | `Student` class that stores a student name and average. (Part 2 of 4.)

```
30      // sets the Student's average
31      public void setAverage(double studentAverage) {
32         // validate that average is > 0.0 and <= 100.0; otherwise,
33         // keep instance variable average's current value
34         if (average > 0.0) {
35            if (average <= 100.0) {
36               this.average = average; // assign to instance variable
37            }
38         }
39      }
40
41      // retrieves the Student's average
42      public double getAverage() {
43         return average;
44      }
45
```

Fig. 4.4 | `Student` class that stores a student name and average. (Part 3 of 4.)

```
46      // determines and returns the Student's letter grade
47      public String getLetterGrade() {
48         String letterGrade = ""; // initialized to empty String
49
50         if (average >= 90.0) {
51            letterGrade = "A";
52         }
53         else if (average >= 80.0) {
54            letterGrade = "B";
55         }
56         else if (average >= 70.0) {
57            letterGrade = "C";
58         }
59         else if (average >= 60.0) {
60            letterGrade = "D";
61         }
62         else {
63            letterGrade = "F";
64         }
65
66         return letterGrade;
67      }
68   }
```

Fig. 4.4 | `Student` class that stores a student name and average. (Part 4 of 4.)

4.7 `Student` Class: Nested `if...else` Statement (Cont.)

- The constructor and method `setAverage` each use *nested* `if` *statements* to *validate* the value used to set the `average`—these statements ensure that the value is greater than `0.0` *and* less than or equal to `100.0`; otherwise, `average`'s value is left *unchanged*.
- Each if statement contains a *simple* condition. If the condition in line 13 is *true*, only then will the condition in line 14 be tested, and *only* if the conditions in both line 13 *and* line 14 are *true* will the statement in line 15 execute.

Software Engineering Observation 4.1

Recall from Chapter 3 that you should not call methods from constructors (we'll explain why in Chapter 10, Object-Oriented Programming: Polymorphism and Interfaces). For this reason, there is duplicated validation code in lines 13–17 and 34–38 of Fig. 4.4 and in subsequent examples.

4.7 `Student` Class: Nested `if...else` Statement (Cont.)

Class StudentTest

- To demonstrate the nested if…else statements in class `Student`'s `getLetterGrade` method, class `StudentTest`'s main method creates two `Student` objects then displays each `Student`'s name and letter grade by calling the objects' `getName` and `getLetterGrade` methods, respectively.

```
1  // Fig. 4.5: StudentTest.java
2  // Create and test Student objects.
3  public class StudentTest {
4     public static void main(String[] args) {
5        Student account1 = new Student("Jane Green", 93.5);
6        Student account2 = new Student("John Blue", 72.75);
7
8        System.out.printf("%s's letter grade is: %s%n",
9           account1.getName(), account1.getLetterGrade());
10       System.out.printf("%s's letter grade is: %s%n",
11          account2.getName(), account2.getLetterGrade());
12    }
13 }
```

```
Jane Green's letter grade is: A
John Blue's letter grade is: C
```

Fig. 4.5 | Create and test `Student` objects.

4.8 `while` Iteration Statement

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

4.8 `while` Iteration Statement (Cont.)

- Example of Java's `while` iteration 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 `product` becomes 243, `product <= 100` becomes false.
- Iteration terminates. The final value of `product` is 243.
- Program execution continues with the next statement after the `while` statement.

Common Programming Error 4.2

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).

4.8 `while` Iteration Statement (Cont.)

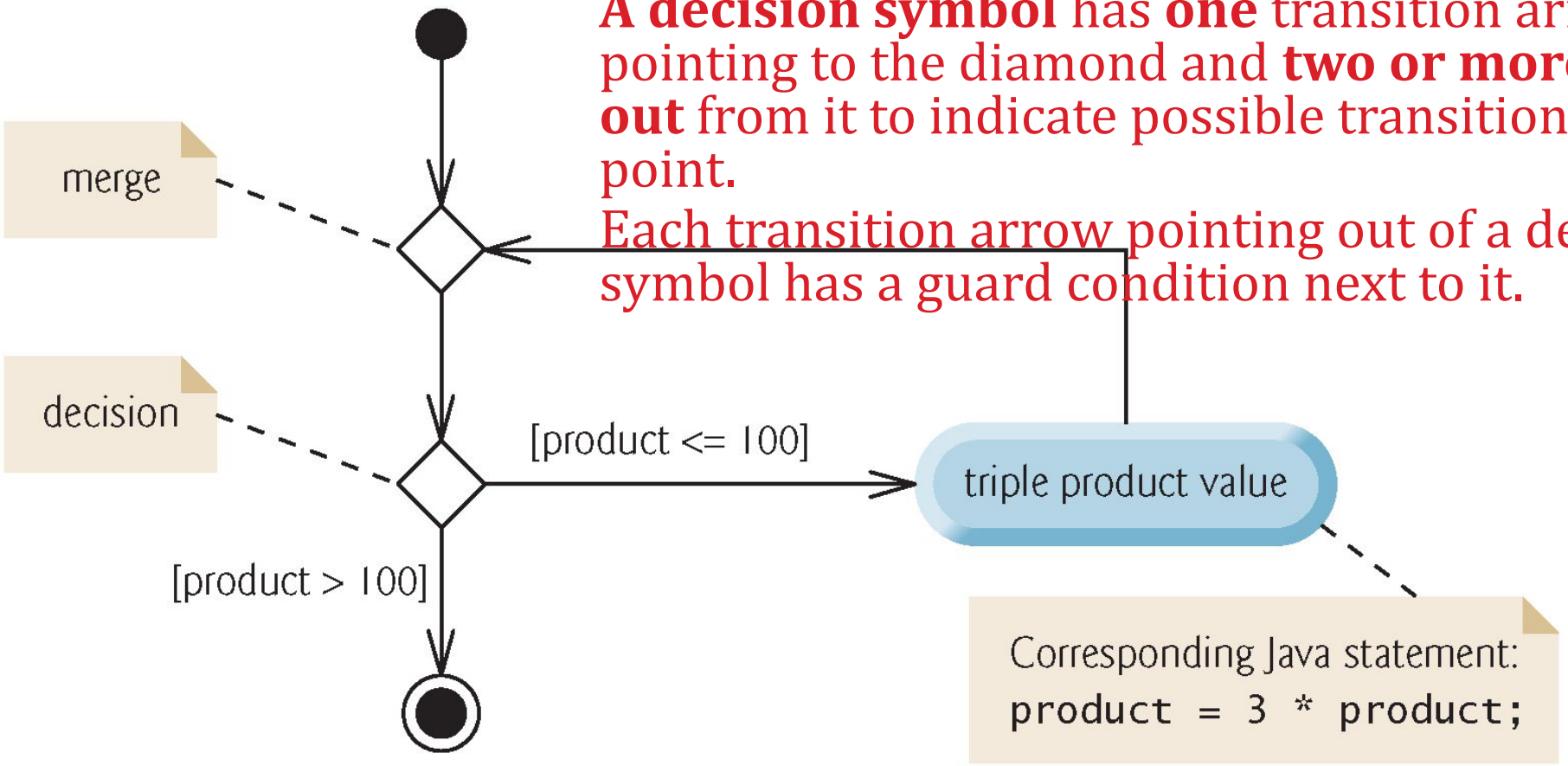
UML Activity Diagram for a `while` Statement

- The UML activity diagram in Fig. 4.6 illustrates the flow of control in the preceding `while` statement.
- The UML represents both the merge symbol and the decision symbol as diamonds.
- The merge symbol joins two flows of activity into one.

4.8 `while` Iteration Statement (Cont.)

- The decision and merge symbols can be distinguished by the number of "incoming" and "outgoing" transition arrows.
  - **A decision symbol** has
    - one transition arrow pointing to the diamond and
    - two or more pointing out from it to indicate possible transitions from that point.
    - Each transition arrow pointing out of a decision symbol has a guard condition next to it.
  - **A merge symbol** has
    - two or more transition arrows pointing to the diamond and
    - only one pointing from the diamond, to indicate multiple activity flows merging to continue the activity.
    - None of the transition arrows associated with a merge symbol has a guard condition.

4.8 while Iteration Statement (Cont.)

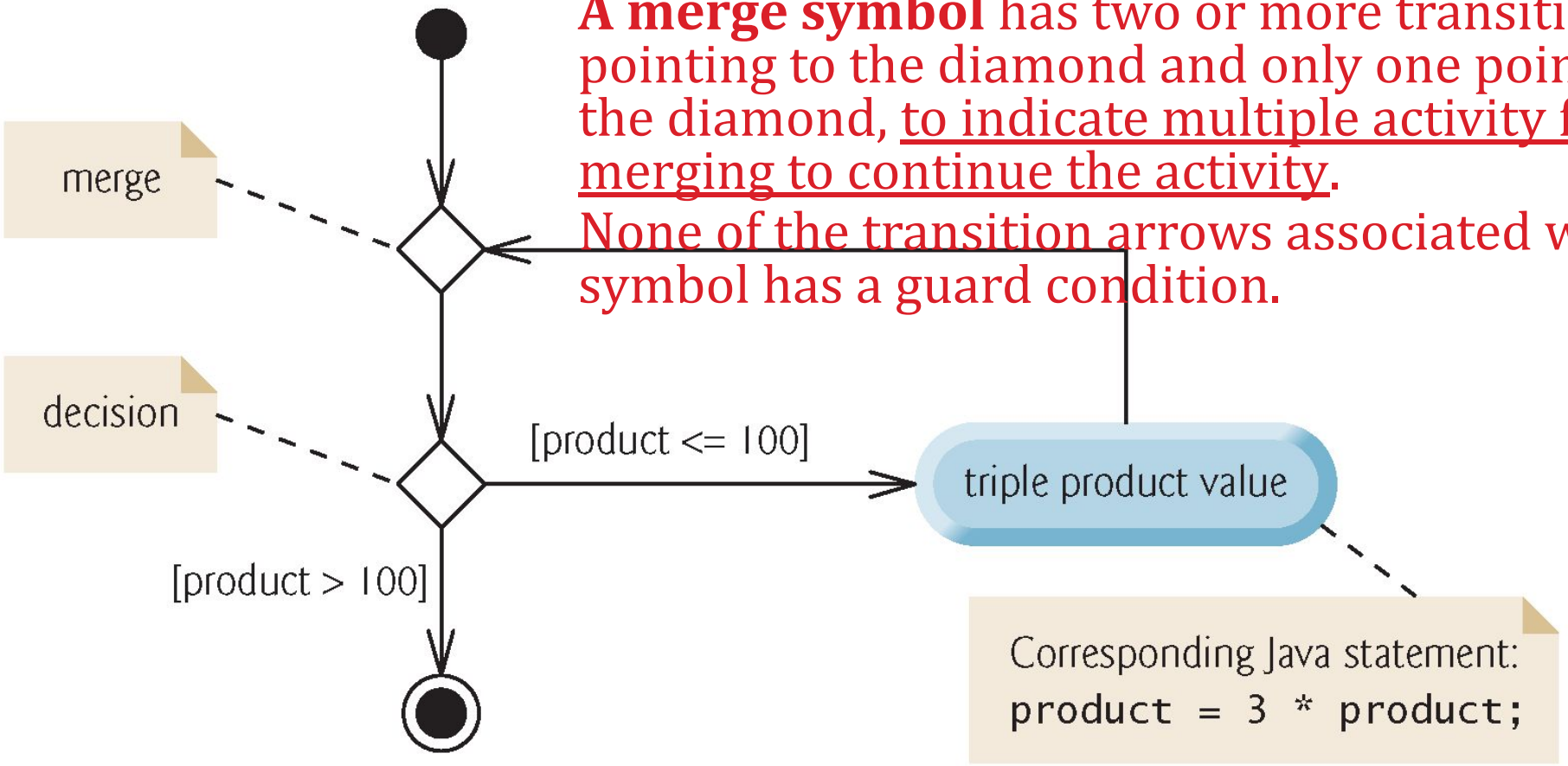


A decision symbol has **one** transition arrow pointing to the diamond and **two or more pointing out** from it to indicate possible transitions from that point.

Each transition arrow pointing out of a decision symbol has a guard condition next to it.

Fig. 4.6 | `while` iteration statement UML activity diagram.

4.8 `while` Iteration Statement (Cont.)



A merge symbol has two or more transition arrows pointing to the diamond and only one pointing from the diamond, to indicate multiple activity flows merging to continue the activity.

None of the transition arrows associated with a merge symbol has a guard condition.

Fig. 4.6 | `while` iteration statement UML activity diagram.

4.9 Formulating Algorithms: Counter-Controlled Iteration

- *A class of ten students took a quiz. The grades (integers in the range 0-100) 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.

4.9 Formulating Algorithms: Counter-Controlled Iteration

Pseudocode Algorithm with Counter-Controlled Iteration

- Use counter-controlled iteration 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 iteration** is often called **definite** iteration, because the number of iterations is known *before* the loop begins executing.

Software Engineering Observation 4.2

Experience has shown that the most difficult part of solving a problem on a computer is developing the algorithm for the solution. Once a correct algorithm has been specified, producing a working Java program from it is usually straightforward.

4.9 Formulating Algorithms: Counter-Controlled Iteration (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. 4.7 | Pseudocode algorithm that uses counter-controlled iteration to solve the class-average problem.

```
1  // Fig. 4.8: ClassAverage.java
2  // Solving the class-average problem using counter-controlled iteration.
3  import java.util.Scanner; // program uses class Scanner
4
5  public class ClassAverage {
6     public static void main(String[] args) {
7        // create Scanner to obtain input from command window
8        Scanner input = new Scanner(System.in);
9
10       // initialization phase
11       int total = 0;  // initialize sum of grades entered by the user
12       int gradeCounter = 1; // initialize # of grade to be entered next
13
14       // processing phase uses counter-controlled iteration
15       while (gradeCounter <= 10) { // loop 10 times
16          System.out.print("Enter grade: "); // prompt
17          int grade = input.nextInt(); // input next grade
18          total = total + grade; // add grade to total
19          gradeCounter = gradeCounter + 1; // increment counter by 1
20       }
```

Fig. 4.8 | Solving the class-average problem using counter-controlled iteration. (Part 1 of 3.)

```
21
22          // termination phase
23          int average = total / 10; // integer division yields integer result
24
25          // display total and average of grades
26          System.out.printf("%nTotal of all 10 grades is %d%n", total);
27          System.out.printf("Class average is %d%n", average);
28       }
29    }
```

Fig. 4.8 | Solving the class-average problem using counter-controlled iteration. (Part 2 of 3.)

```
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. 4.8 | Solving the class-average problem using counter-controlled iteration. (Part 3 of 3.)

4.9 Formulating Algorithms: Counter-Controlled Iteration (Cont.)

Local Variables in Method `main`

- Variables declared in a method body are **local variables** and can be used only from the line of their declaration to the closing right brace of the method declaration.
- A local variable's declaration must appear *before* the variable is used in that method.
- A local variable cannot be accessed outside the method in which it's declared.

Common Programming Error 4.3

Using the value of a local variable before it's initialized results in a compilation error. All local variables must be initialized before their values are used in expressions.

Error-Prevention Tip 4.3

Initialize each total and counter, 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 when to use 0 and when to use 1).

4.9 Formulating Algorithms: Counter-Controlled Iteration (Cont.)

Notes on Integer Division and Truncation

- The program's output indicates that the sum of the grade values in the sample execution is 846, which, when divided by 10, should yield the floating-point number 84.6.
- The result of the calculation `total / 10` (Fig. 4.8) is the integer 84, because `total` and `10` are both integers.
- Dividing two integers results in integer division—any fractional part of the calculation is truncated (i.e., *lost*).

Common Programming Error 4.4

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

4.9 Formulating Algorithms: Counter-Controlled Iteration (Cont.)

A Note About Arithmetic Overflow

- In Fig. 4.8, the following statement added each `grade` entered by the user to the `total`

```
total = total + grade; // add grade to total
```

- Even this simple statement has a *potential* problem—adding the integers could result in a value that's *too large* to store in an `int` variable.
- This is known as arithmetic overflow and causes *undefined behavior*, which can lead to unintended results.

4.9 Formulating Algorithms: Counter-Controlled Iteration (Cont.)

- Figure 2.7's `Addition` program had the same issue when calculating the sum of two `int` values entered by the user:
  - `int sum = number1 + number2;`

4.9 Formulating Algorithms: Counter-Controlled Iteration (Cont.)

- The maximum and minimum values that can be stored in an `int` variable are represented by the constants `MIN_VALUE` and `MAX_VALUE`, respectively, which are defined in class `Integer`.
- There are similar constants for the other integral types and for floating-point types.
- Each primitive type has a corresponding class type in package `java.lang`.

4.9 Formulating Algorithms: Counter-Controlled Iteration (Cont.)

- It's considered a good practice to ensure, *before* you perform arithmetic calculations, that they will *not* overflow.
- The code for doing this is shown on the CERT website `www.securecoding.cert.org`—just search for guideline "NUM00-J."
- The code for doing this is shown on the CERT website:
  - http://www.securecoding.cert.org
- In industrial-strength code, you should perform checks like these for all calculations.

4.9 Formulating Algorithms: Counter-Controlled Iteration (Cont.)

A Deeper Look at Receiving User Input

- Any time a program receives input from the user, various problems might occur. For example, in the following statement we assume that the user will enter an integer grade in the range 0 to 100

```
int grade = input.nextInt(); // input next grade
```

- However, the person entering a grade could enter an integer less than 0, an integer greater than 100, an integer outside the range of values that can be stored in an `int` variable, a number containing a decimal point or a value containing letters or special symbols that's not even an integer.

4.9 Formulating Algorithms: Counter-Controlled Iteration (Cont.)

- To ensure that inputs are valid, industrial-strength programs must test for all possible erroneous cases.
- A program that inputs grades should validate the grades by using range checking to ensure that they are values from 0 to 100.
- You can then ask the user to reenter any value that's out of range.
- If a program requires inputs from a specific set of values (e.g., nonsequential product codes), you can ensure that each input matches a value in the set.

4.10 Formulating Algorithms: Sentinel-Controlled Iteration

- *Develop a class-averaging program that processes grades for an arbitrary number of students each time it is run.*
- Sentinel-controlled iteration is often called indefinite iteration because the number of iterations 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.

4.10 Formulating Algorithms: Sentinel-Controlled Iteration (Cont.)

Developing the Pseudocode Algorithm with Top-Down, Stepwise Refinement: The Top and First Refinement

- 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.

4.10 Formulating Algorithms: Sentinel-Controlled Iteration (Cont.)

- 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.

Software Engineering Observation 4.3

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

Software Engineering Observation 4.4

Many programs can be divided logically into three phases: an initialization phase that initializes the variables; a processing phase that inputs data values and adjusts program variables accordingly; and a termination phase that calculates and outputs the final results.

4.10 Formulating Algorithms: Sentinel-Controlled Iteration (Cont.)

Proceeding to the Second Refinement

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

  Initialize variables

- can be refined as follows:

  Initialize total to zero

  Initialize counter to zero

4.10 Formulating Algorithms: Sentinel-Controlled Iteration (Cont.)

- The pseudocode statement

  Input, sum and count the quiz grades

- requires iteration to successively input each grade.
- We do not know in advance how many grades will be entered, so we'll use sentinel-controlled iteration.

4.10 Formulating Algorithms: Sentinel-Controlled Iteration (Cont.)

- 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)
```

4.10 Formulating Algorithms: Sentinel-Controlled Iteration (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. 4.9 | Class-average pseudocode algorithm with sentinel-controlled iteration.

Error-Prevention Tip 4.4

When performing division (/) or remainder (%) calculations in which the right operand could be zero, test for this and handle it (e.g., display an error message) rather than allowing the error to occur.

Software Engineering Observation 4.5

Terminate the top-down, stepwise refinement process when you've specified the pseudocode algorithm in sufficient detail for you to convert the pseudocode to Java. Normally, implementing the Java program is then straightforward.

Software Engineering Observation 4.6

Some programmers do not use program development tools like pseudocode. They feel that their ultimate goal is to solve the problem on a computer and that writing pseudocode merely delays the production of final outputs. Although this may work for simple and familiar problems, it can lead to serious errors and delays in large, complex projects.

```
1  // Fig. 4.10: ClassAverage.java
2  // Solving the class-average problem using sentinel-controlled iteration.
3  import java.util.Scanner; // program uses class Scanner
4
5  public class ClassAverage {
6     public static void main(String[] args) {
7        // create Scanner to obtain input from command window
8        Scanner input = new Scanner(System.in);
9
10       // initialization phase
11       int total = 0; // initialize sum of grades
12       int gradeCounter = 0; // initialize # of grades entered so far
13
```

Fig. 4.10 | Solving the class-average problem using sentinel-controlled iteration. (Part 1 of 4.)

```
14          // processing phase
15          // prompt for input and read grade from user
16          System.out.print("Enter grade or -1 to quit: ");
17          int grade = input.nextInt();
18
19          // loop until sentinel value read from user
20          while (grade != -1) {
21             total = total + grade; // add grade to total
22             gradeCounter = gradeCounter + 1; // increment counter
23
24             // prompt for input and read next grade from user
25             System.out.print("Enter grade or -1 to quit: ");
26             grade = input.nextInt();
27          }
28
```

Fig. 4.10 | Solving the class-average problem using sentinel-controlled iteration. (Part 2 of 4.)

```
29          // termination phase
30          // if user entered at least one grade...
31          if (gradeCounter != 0) {
32             // use number with decimal point to calculate average of grades
33             double average = (double) total / gradeCounter;
34
35             // display total and average (with two digits of precision)
36             System.out.printf("%nTotal of the %d grades entered is %d%n",
37                gradeCounter, total);
38             System.out.printf("Class average is %.2f%n", average);
39          }
40          else { // no grades were entered, so output appropriate message
41             System.out.println("No grades were entered");
42          }
43       }
44    }
```

Fig. 4.10 | Solving the class-average problem using sentinel-controlled iteration. (Part 3 of 4.)

```
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 | Solving the class-average problem using sentinel-controlled iteration. (Part 4 of 4.)

4.10 Formulating Algorithms: Sentinel-Controlled Iteration (Cont.)

Program Logic for Sentinel-Controlled Iteration vs. Counter-Controlled Iteration

- Program logic for **sentinel-controlled iteration**
  - Reads the first value before reaching the `while`.
  - This value determines whether the program's flow of control should enter the body of the `while`. If the condition of the `while` is false, the user entered the sentinel value, so the body of the `while` does not execute (i.e., no grades were entered).
  - If the condition is true, the body begins execution and processes the input.
  - Then the loop body inputs the next value from the user before the end of the loop.

Good Programming Practice 4.3

In a sentinel-controlled loop, prompts should remind the user of the sentinel.

Common Programming Error 4.5

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.

4.10 Formulating Algorithms: Sentinel-Controlled Iteration (Cont.)

Explicitly and Implicitly Converting Between Primitive Types

- 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.
- Java evaluates only arithmetic expressions in which the operands' types are *identical*.
- 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.

Common Programming Error 4.6

A cast operator can be used to convert between primitive numeric types, such as `int` and `double`, and between related reference types (as we discuss in Chapter 10, Object-Oriented Programming: Polymorphism and Interfaces). Casting to the wrong type may cause compilation errors or runtime errors.

4.10 Formulating Algorithms: Sentinel-Controlled Iteration (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).
- Java also supports unary versions of the plus (+) and minus (–) operators.
- Cast operators associate from right to left; same precedence as other unary operators, such as unary + and unary -.
- This precedence is one level higher than that of the multiplicative operators *, / and %.
- Appendix A: Operator precedence chart

4.10 Formulating Algorithms: Sentinel-Controlled Iteration (Cont.)

Widening casting is done automatically when passing a smaller size type to a larger size type:

```
public static void main(String[] args) {
    int myInt = 9;
    double myDouble = myInt; // Automatic casting: int to double
    System.out.println(myInt);      // Outputs 9
    System.out.println(myDouble);   // Outputs 9.0
}
```

Narrowing casting must be done manually by placing the type in parentheses in front of the value:

```
public static void main(String[] args) {
    double myDouble = 9.78d;
    int myInt = (int) myDouble; // Manual casting: double to int
    System.out.println(myDouble);   // Outputs 9.78
    System.out.println(myInt);      // Outputs 9
}
```

4.10 Formulating Algorithms: Sentinel-Controlled Iteration (Cont.)

Floating-Point Number Precision

- Floating-point numbers are **not** always 100% precise, but they have numerous applications.
- For example, when we speak of a "normal" body temperature of 98.6, we do not need to be precise to a large number of digits.
- Floating-point numbers often arise as a result of division, such as in this example's class-average calculation.
- In conventional arithmetic, when we **divide 10 by 3**, the result is 3.3333333…, with the sequence of 3s repeating infinitely.
- The computer allocates only a fixed amount of space to hold such a value, so clearly the stored floating-point value can be only an **approximation**.

4.10 Formulating Algorithms: Sentinel-Controlled Iteration (Cont.)

- Owing to the imprecise nature of floating-point numbers, type **`double`** is preferred over type **`float`**, because `double` variables can represent floating-point numbers more accurately.
- In some applications, the precision of float and double variables will be inadequate.

- For precise floating-point numbers (such as those required by monetary calculations), Java provides class `BigDecimal` (package `java.math`), which we'll discuss in Chapter 8.

Common Programming Error 4.7

Using floating-point numbers in a manner that assumes they're represented precisely can lead to incorrect results.

4.11 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(for pass)) or a 2(for fail)).
    - 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 "Bonus t
    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.

Error-Prevention Tip 4.5

Initializing local variables when they're declared helps you avoid any compilation errors that might arise from attempts to use uninitialized variables. While Java does not require that local-variable initializations be incorporated into declarations, it does require that local variables have values before they're used in an expression.

```
1  // Fig. 4.12: Analysis.java
2  // Analysis of examination results using nested control statements.
3  import java.util.Scanner; // class uses class Scanner
4
5  public class Analysis {
6     public static void main(String[] args) {
7        // create Scanner to obtain input from command window
8        Scanner input = new Scanner(System.in);
9
10       // initializing variables in declarations
11       int passes = 0;
12       int failures = 0;
13       int studentCounter = 1;
14
```

Fig. 4.12 | Analysis of examination results using nested control statements. (Part 1 of 5.)

```
15          // process 10 students using counter-controlled loop
16          while (studentCounter <= 10) {
17             // prompt user for input and obtain value from user
18             System.out.print("Enter result (1 = pass, 2 = fail): ");
19             int result = input.nextInt();
20
21             // if...else is nested in the while statement
22             if (result == 1) {
23                passes = passes + 1;
24             }
25             else {
26                failures = failures + 1;
27             }
28
29             // increment studentCounter so loop eventually terminates
30             studentCounter = studentCounter + 1;
31          }
```

Fig. 4.12 | Analysis of examination results using nested control statements. (Part 2 of 5.)

```
32
33          // termination phase; prepare and display results
34          System.out.printf("Passed: %d%nFailed: %d%n", passes, failures);
35
36          // determine whether more than 8 students passed
37          if (passes > 8) {
38             System.out.println("Bonus to instructor!");
39          }
40       }
41    }
```

Fig. 4.12 | Analysis of examination results using nested control statements. (Part 3 of 5.)

```
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 | Analysis of examination results using nested control statements. (Part 4 of 5.)

```
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 | Analysis of examination results using nested control statements. (Part 5 of 5.)

4.12 Compound Assignment Operators

- Compound assignment operators abbreviate assignment expressions.
- 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.
- Statements like

variable = variable operator expression;

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

variable operator= expression;

4.12 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. 4.13 | Arithmetic compound assignment operators.

4.13 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.

4.13 Increment and Decrement Operators

Operator	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.

4.12 Increment and Decrement Operators (Cont.)

- Using the prefix increment (or decrement) operator to add (or subtract) 1 from a variable is known as preincrementing (or predecrementing) the variable.
- Preincrementing (or predecrementing) a variable causes the variable to be incremented (decremented) by 1; then the new value is used in the expression in which it appears.
- Using the postfix increment (or decrement) operator to add (or subtract) 1 from a variable is known as postincrementing (or postdecrementing) the variable.
- This causes the current value of the variable to be used in the expression in which it appears; then the variable's value is incremented (decremented) by 1.

Good Programming Practice 4.4

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

```
1  // Fig. 4.15: Increment.java
2  // Prefix increment and postfix increment operators.
3
4  public class Increment {
5     public static void main(String[] args) {
6        // demonstrate postfix increment operator
7        int c = 5;
8        System.out.printf("c before postincrement: %d%n", c); // prints 5
9        System.out.printf("    postincrementing c: %d%n", c++); // prints 5
10       System.out.printf(" c after postincrement: %d%n", c); // prints 6
11
12       System.out.println(); // skip a line
13
14       // demonstrate prefix increment operator
15       c = 5;
16       System.out.printf(" c before preincrement: %d%n", c); // prints 5
17       System.out.printf("     preincrementing c: %d%n", ++c); // prints 6
18       System.out.printf("  c after preincrement: %d%n", c); // prints 6
19    }
20 }
```

Fig. 4.15 | Prefix increment and postfix increment operators. (Part 1 of 2.)

```
c before postincrement: 5
    postincrementing c: 5
 c after postincrement: 6

 c before preincrement: 5
     preincrementing c: 6
  c after preincrement: 6
```

Fig. 4.15 | Prefix increment and postfix increment operators. (Part 2 of 2.)

Common Programming Error 4.8

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.

Error-Prevention Tip 4.6

Refer to the operator precedence and associativity chart (Appendix A) when writing expressions containing many operators. Confirm that the operators in the expression are performed in the order you expect. If you're uncertain about the order of evaluation in a complex expression, break the expression into smaller statements or use parentheses to force the order of evaluation, exactly as you'd do in an algebraic expression. Be sure to observe that some operators such as assignment (=) associate right to left rather than left to right.

Operators						Associativity	Type
`++`	`--`					right to left	unary postfix
`++`	`--`	`+`	`-`	(*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. 4.16 | Precedence and associativity of the operators discussed so far.

4.14 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.
- 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`.

Portability Tip 4.1

The primitive types in Java are portable across all computer platforms that support Java.

4.15 (Optional) GUI and Graphics Case Study: Creating Simple Drawings

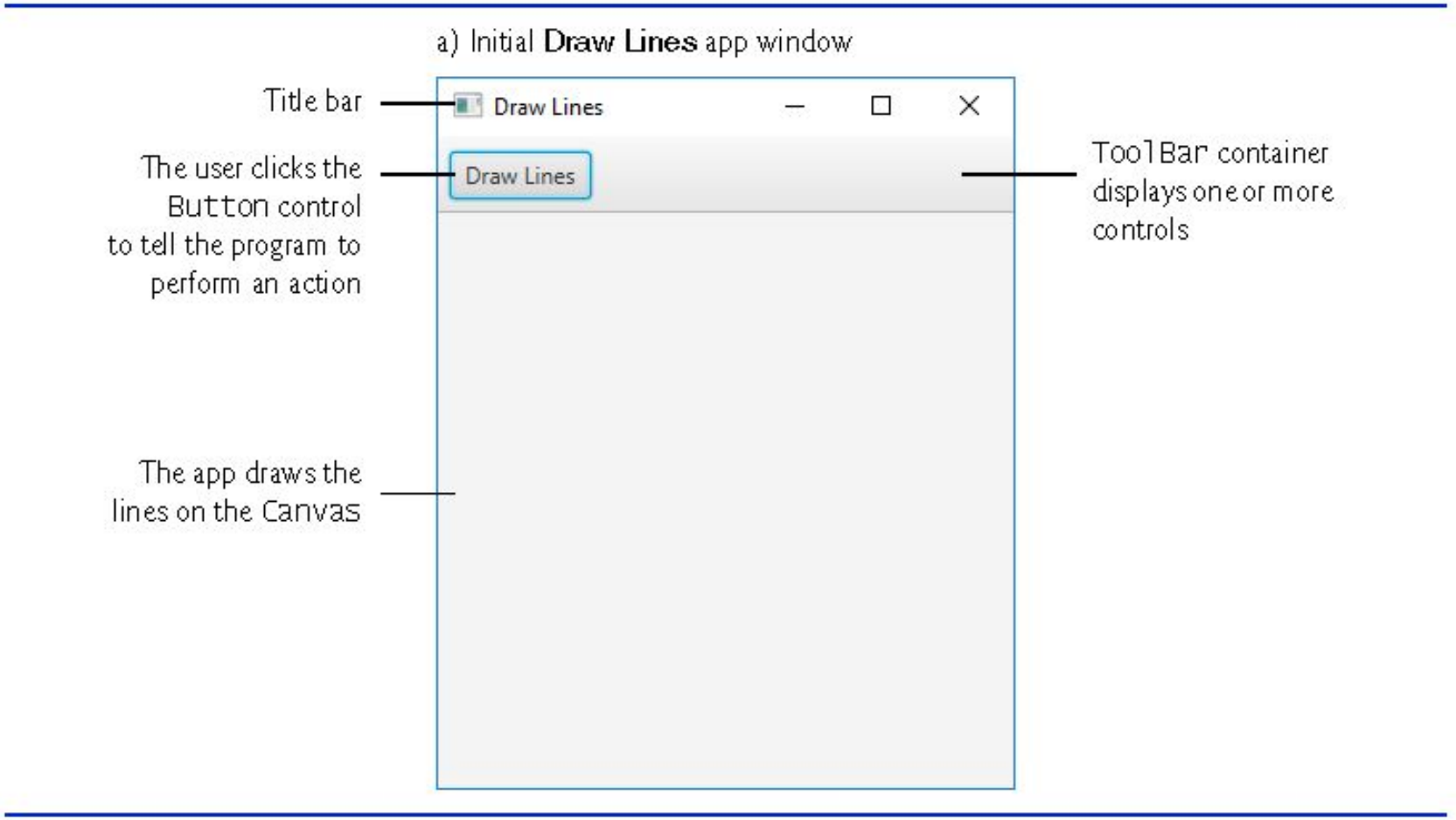


Fig. 4.17 | **Draw Lines** app in action. (Part 1 of 2.)

4.15 (Optional) GUI and Graphics Case Study: Creating Simple Drawings

b) **Draw Lines** app window after the user clicks the **Draw Lines** `Button`

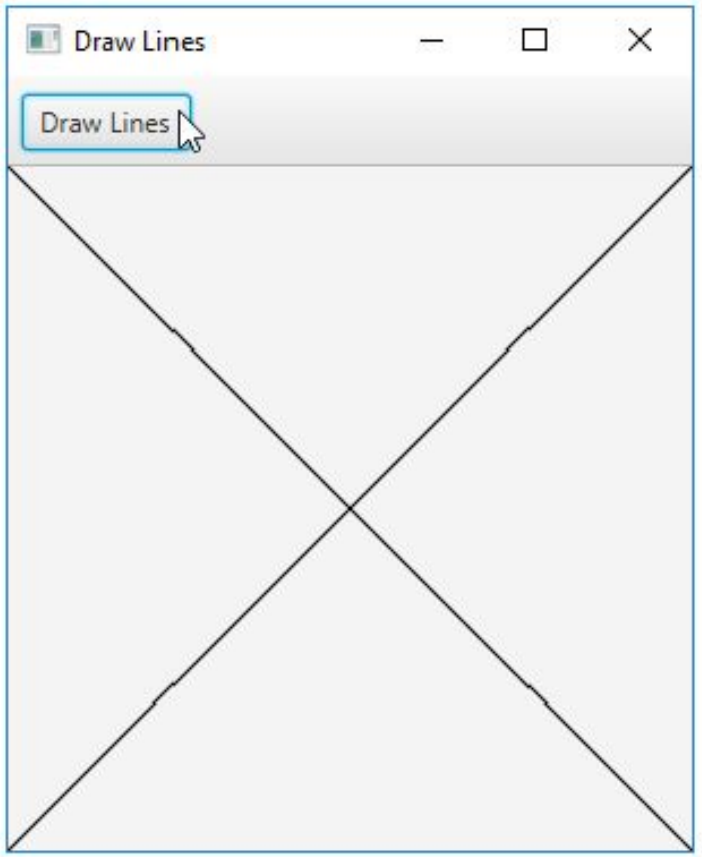


Fig. 4.17 | **Draw Lines** app in action. (Part 2 of 2.)

Look-and-Feel Observation 4.1

All the areas in a `BorderPane` are optional: If the top or bottom area is empty, the left, center and right areas expand vertically to fill that area. If the left or right area is empty, the center expands horizontally to fill that area.

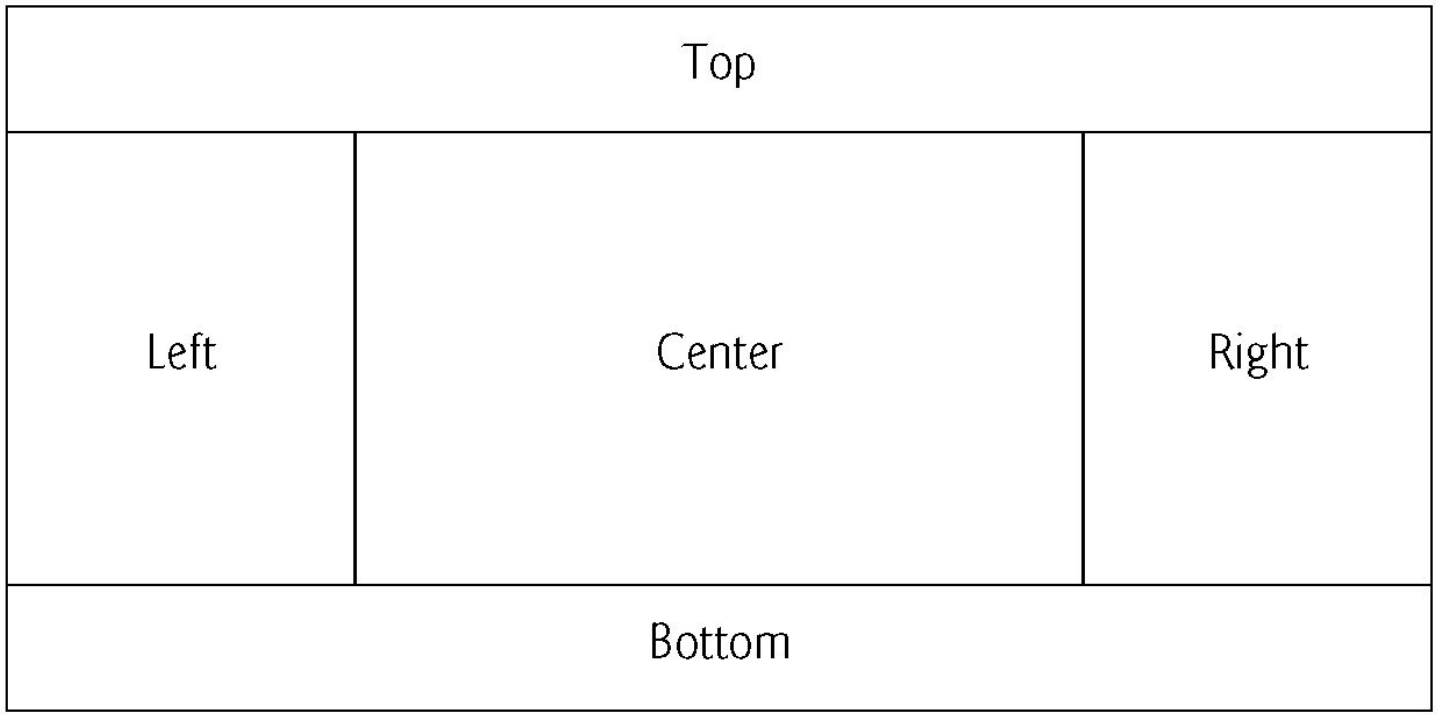


Fig. 4.18 | BorderPane's five areas.

Look-and-Feel Observation 4.2

`ToolBar`s typically organize multiple controls at a layout's edges, such as in a `BorderPane`'s top, right, bottom or left areas.

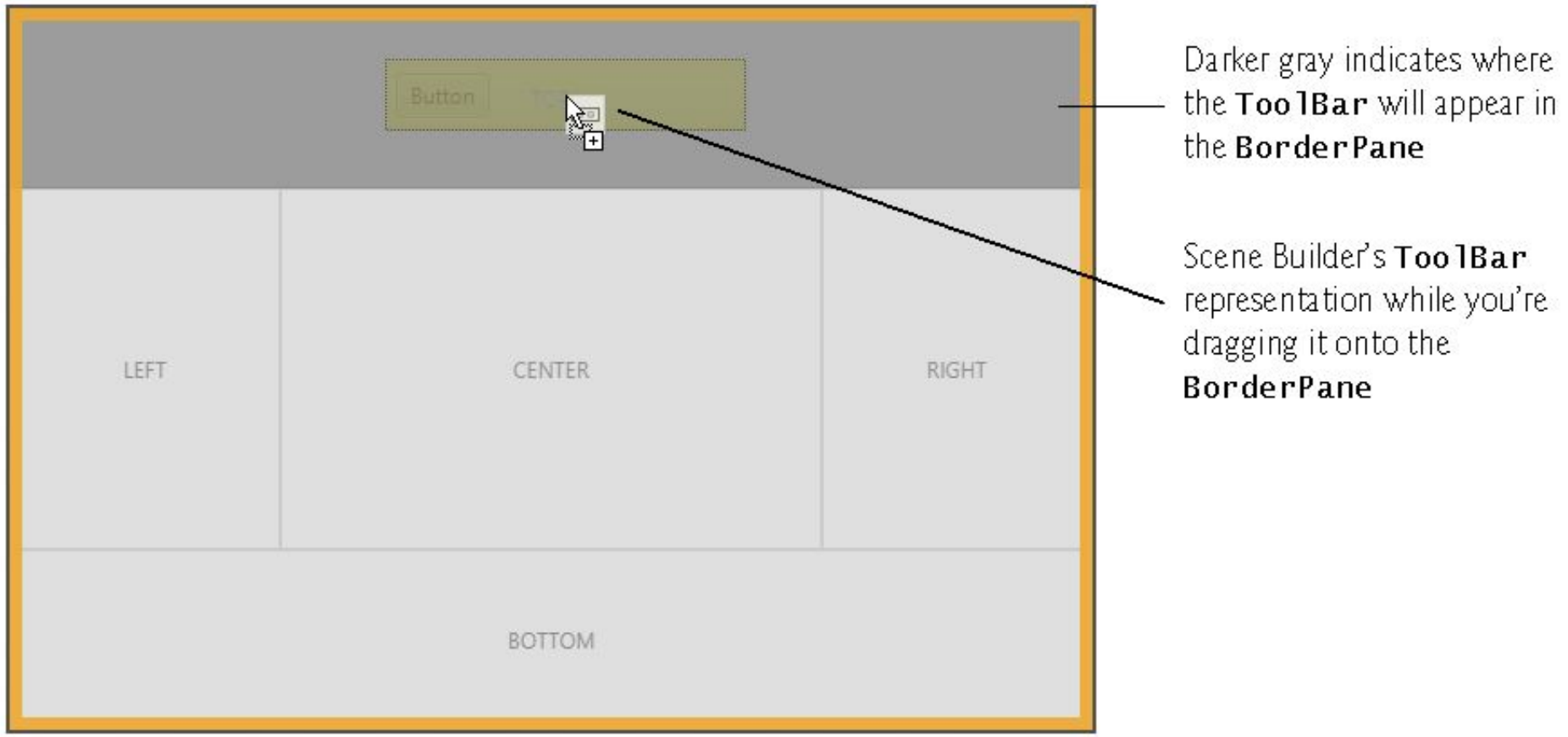


Fig. 4.19 | Dragging a `ToolBar` onto a `BorderPane`.

Fig. 4.20 | Initial **ToolBar** containing a **Button**—by default the **ToolBar** is the full width of the **BorderPane**'s top area.

Fig. 4.21 | `ToolBar` containing the **`Button`** with its updated text.

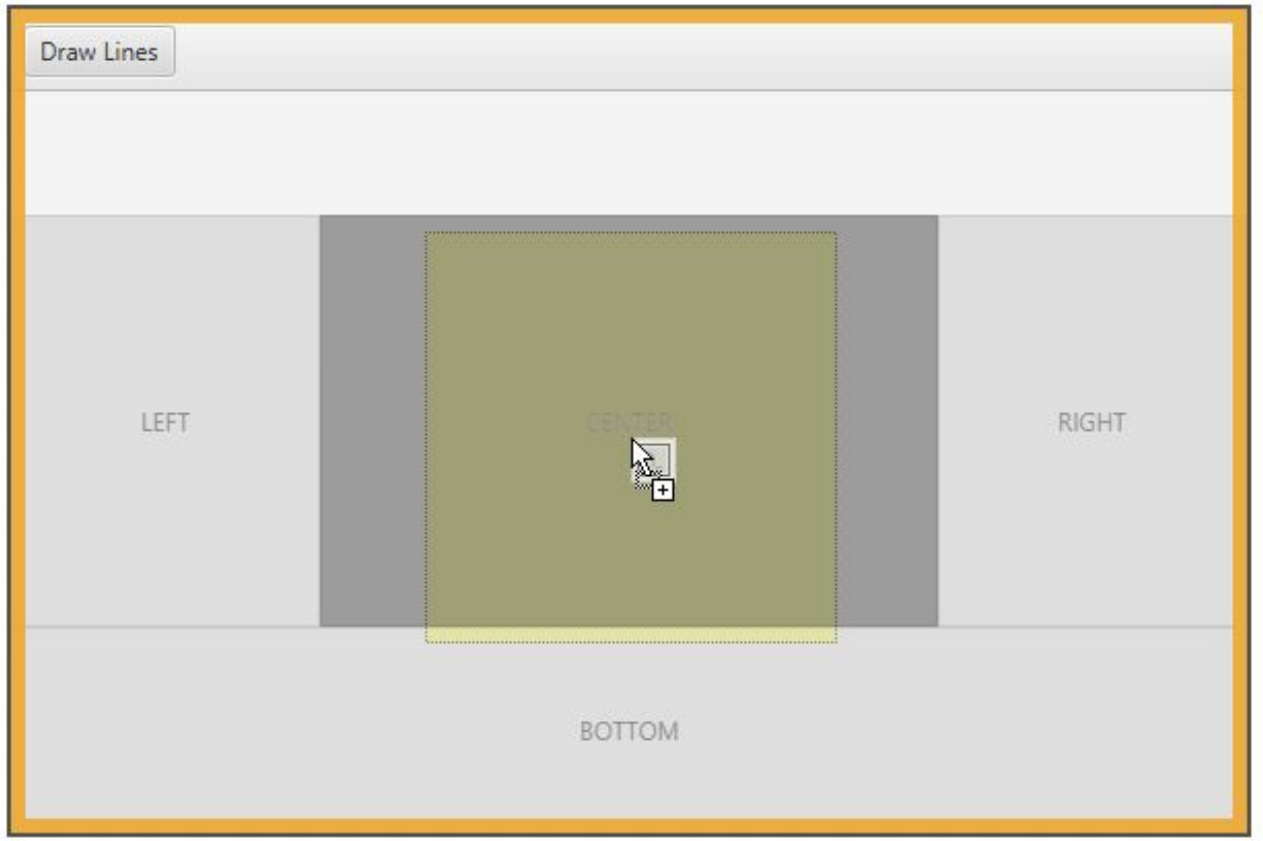


Fig. 4.22 | Dragging and dropping the **Canvas** onto the **BorderPane**'s center area.

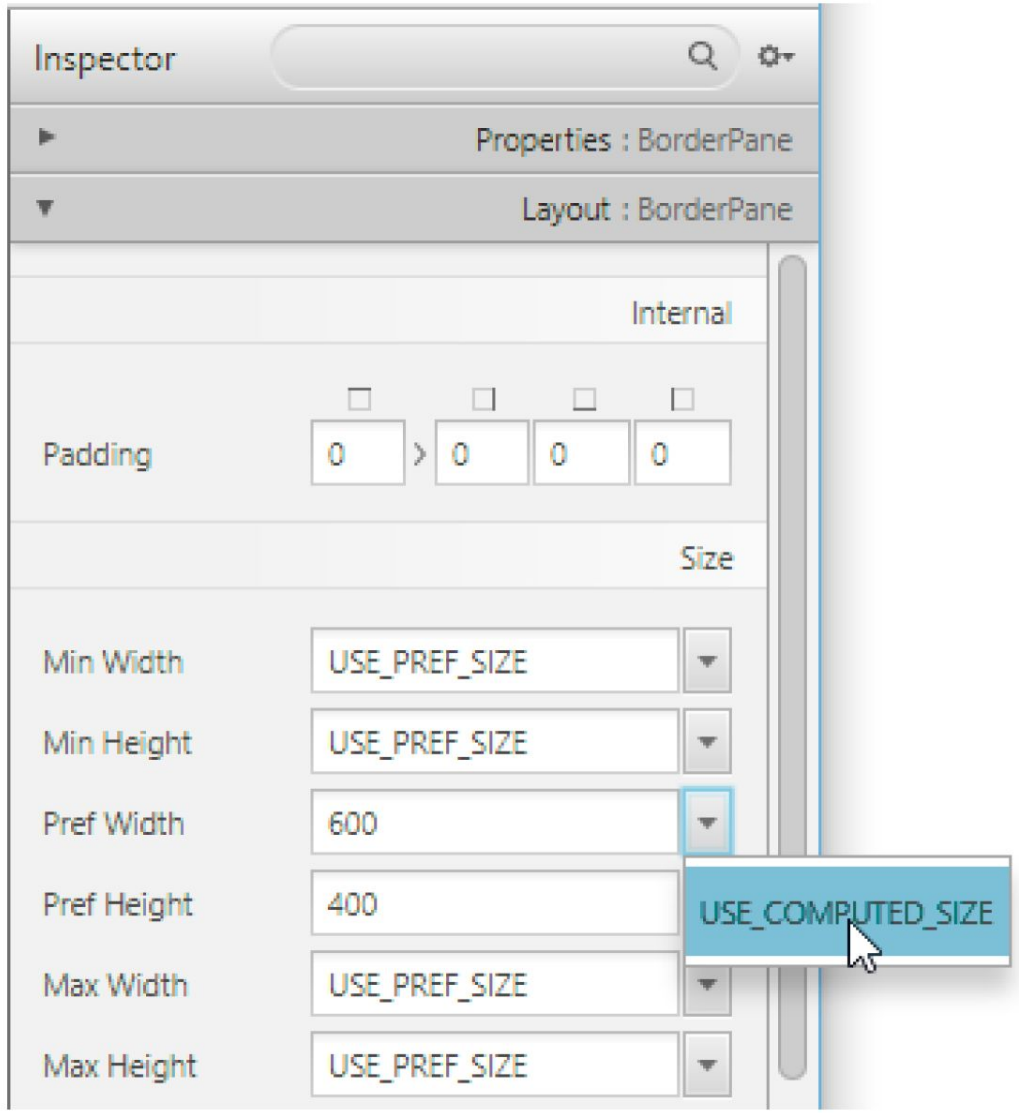


Fig. 4.23 | Selecting `USE_COMPUTED_SIZE` for the `BorderPane`'s **Pref Width** property.

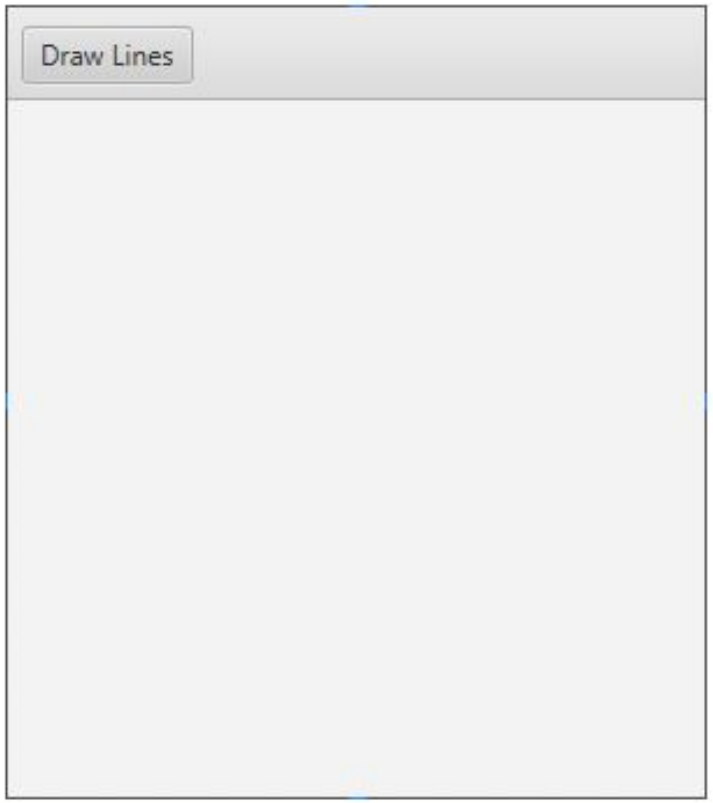


Fig. 4.24 | Completed GUI in Scene Builder's content panel.

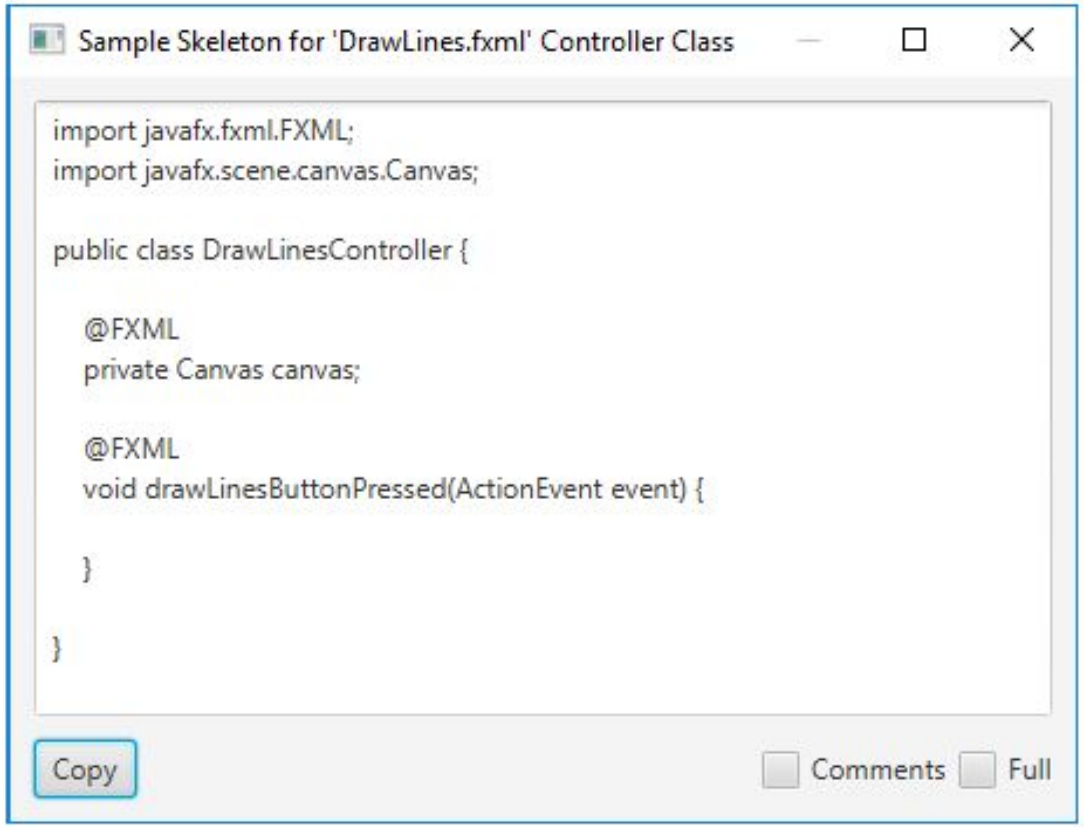


Fig. 4.25 | Skeleton controller-class code generated by Scene Builder.

```
1  // Fig. 4.26: DrawLinesController.java
2  // Using strokeLine to connect the corners of a canvas.
3  import javafx.event.ActionEvent;
4  import javafx.fxml.FXML;
5  import javafx.scene.canvas.Canvas;
6  import javafx.scene.canvas.GraphicsContext;
7
8  public class DrawLinesController {
9     @FXML private Canvas canvas; // used to get the GraphicsContext
10
11    // when user presses Draw Lines button, draw two Lines in the Canvas
12    @FXML
13    void drawLinesButtonPressed(ActionEvent event) {
14       // get the GraphicsContext, which is used to draw on the Canvas
15       GraphicsContext gc = canvas.getGraphicsContext2D();
16
17       // draw line from upper-left to lower-right corner
18       gc.strokeLine(0, 0, canvas.getWidth(), canvas.getHeight());
19
20       // draw line from upper-right to lower-left corner
21       gc.strokeLine(canvas.getWidth(), 0, 0, canvas.getHeight());
22    }
23 }
```

Fig. 4.26 | Using `strokeLine` to connect the corners of a canvas.

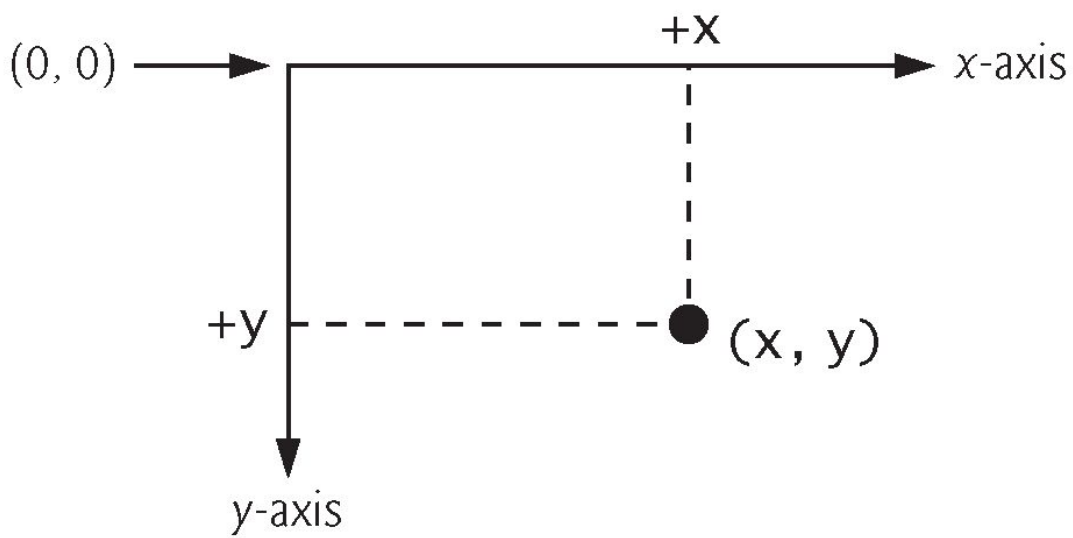


Fig. 4.27 | Java coordinate system. Units are measured in pixels.

```
1  // Fig. 4.28: DrawLines.java
2  // Main application class that loads and displays the DrawLines GUI.
3  import javafx.application.Application;
4  import javafx.fxml.FXMLLoader;
5  import javafx.scene.Parent;
6  import javafx.scene.Scene;
7  import javafx.stage.Stage;
8
9  public class DrawLines extends Application {
10    @Override
11    public void start(Stage stage) throws Exception {
12       // loads DrawLines.fxml and configures the DrawLinesController
13       Parent root =
14          FXMLLoader.load(getClass().getResource("DrawLines.fxml"));
15
```

Fig. 4.28 | Main application class that loads and displays the **Draw Lines** GUI. (Part 1 of 2.)

```
16          Scene scene = new Scene(root); // attach scene graph to scene
17          stage.setTitle("Draw Lines"); // displayed in window's title bar
18          stage.setScene(scene); // attach scene to stage
19          stage.show(); // display the stage
20       }
21
22       // application execution starts here
23       public static void main(String[] args) {
24          launch(args); // create a DrawLines object and call its start method
25       }
26    }
```

Fig. 4.28 | Main application class that loads and displays the **Draw Lines** GUI. (Part 2 of 2.)

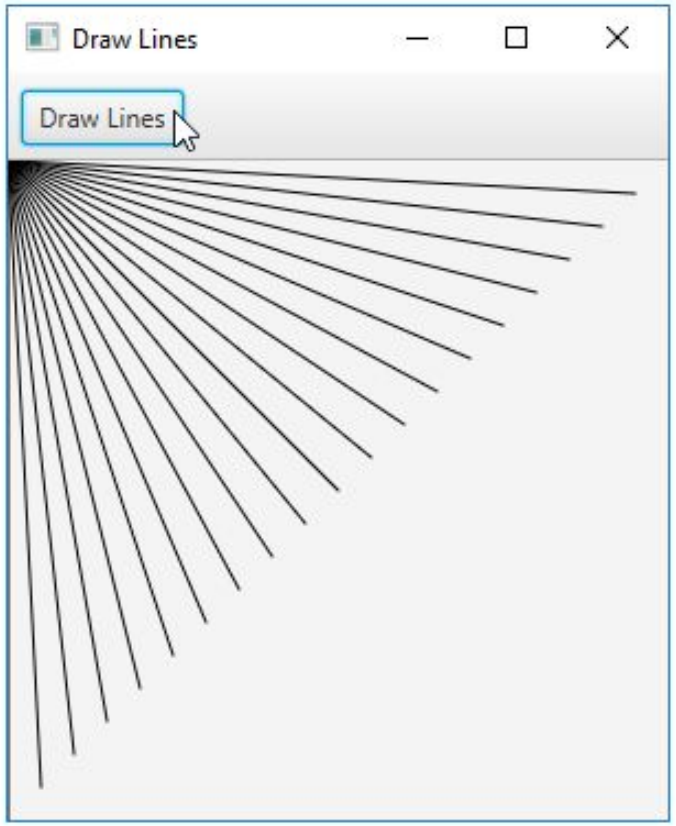


Fig. 4.29 | Lines fanning from a corner.

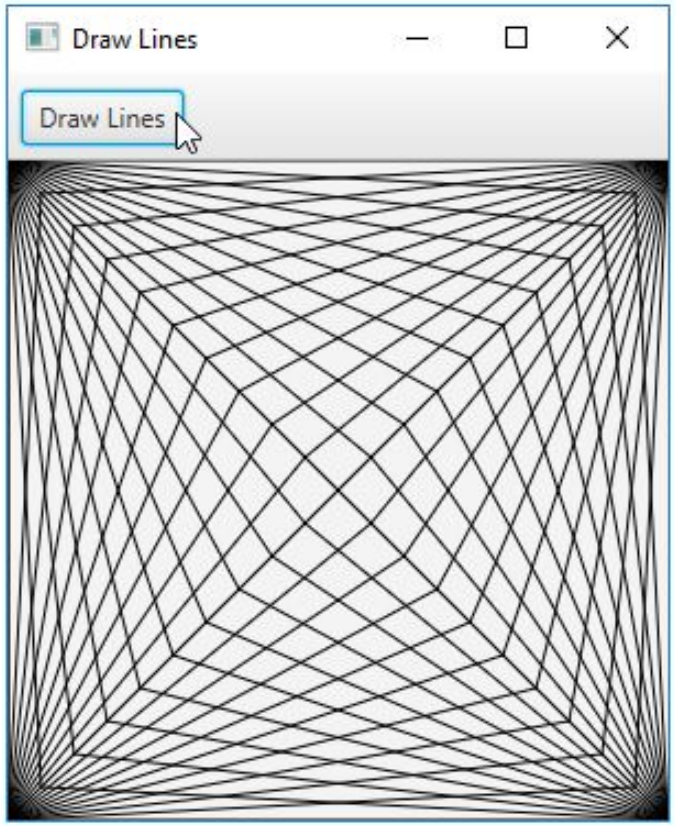


Fig. 4.30 | Lines fanning from all four corners.

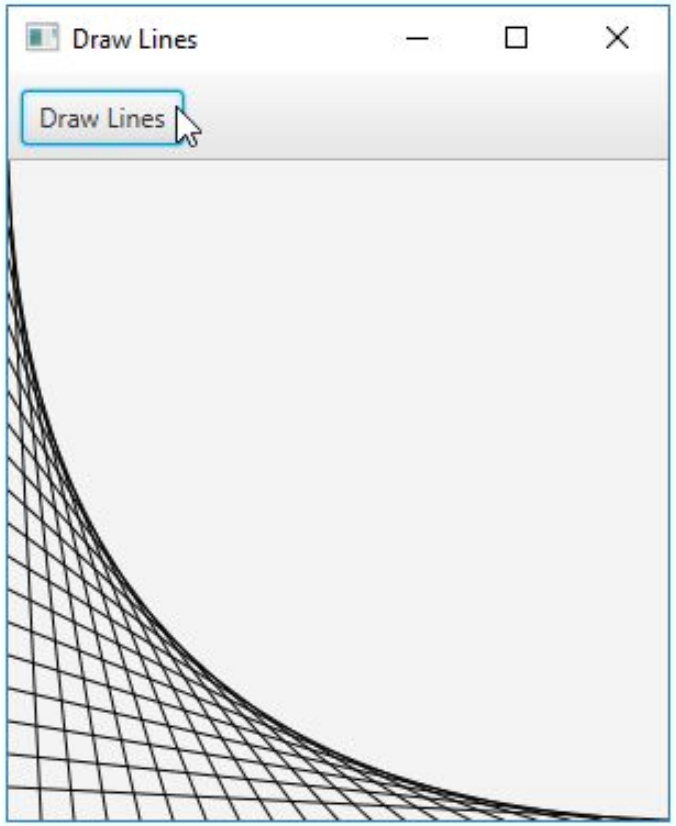


Fig. 4.31 | Line art with loops and `strokeLine`.

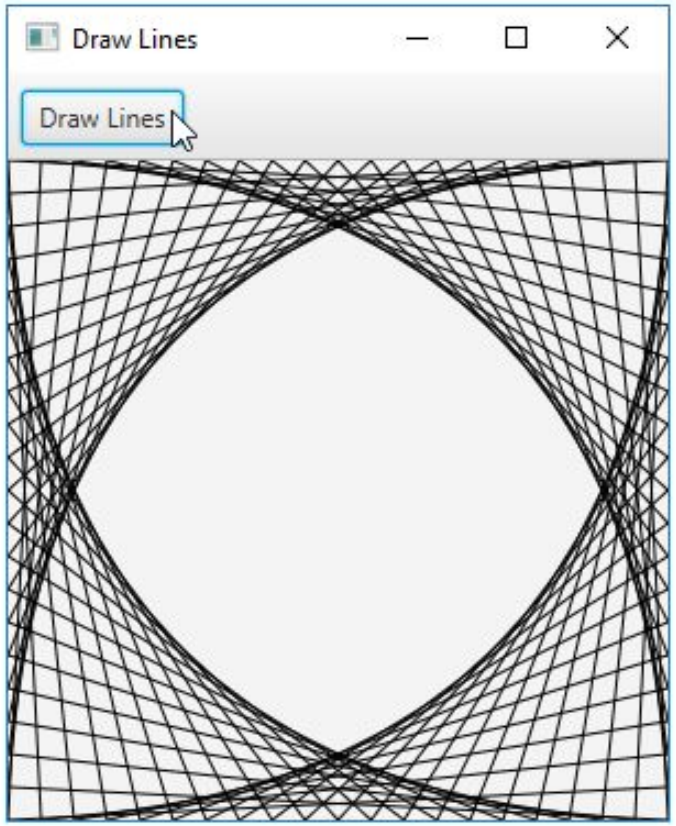


Fig. 4.32 | Line art with loops and `strokeLine`—repeating from all four corners.