Chapter 5

Looping

Chapter Introduction

In this chapter, you will learn about:

- The advantages of looping
- Using a loop control variable
- Nested loops
- Avoiding common loop mistakes
- Using a for loop
- Common loop applications

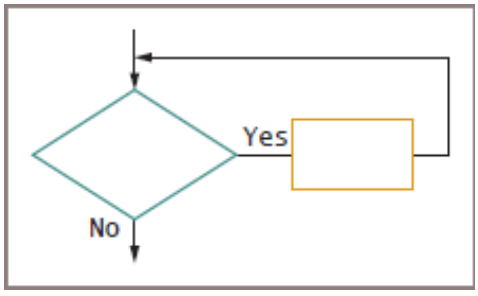
5-1 Understanding the Advantages of Looping

Although making decisions is what makes computers seem intelligent, looping makes computer programming both efficient and worthwhile. When you use a loop, one set of instructions can operate on multiple, separate sets of data. Using fewer instructions results in less time required for design and coding, fewer errors, and shorter compile time.

Recall the loop structure that you learned about in Chapter 3; it looks like Figure 5-1. As long as a Boolean expression remains true, the body of a `while` loop executes.

Figure 5-1

The Loop Structure



You have already learned that many programs use a loop to control repetitive tasks. For

example, Figure 5-2 shows the basic structure of many business programs. After some housekeeping tasks are completed, the detail loop repeats once for every data record that must be processed.

Figure 5-2

The Mainline Logic Common to Many Business Programs

For example, Figure 5-2 might represent the mainline logic of a typical payroll program. The first employee's data would be entered in the `housekeeping()` module, and while the `eof` condition is not met, the `detailLoop()` module would perform such tasks as determining regular and overtime pay and deducting taxes, insurance premiums, charitable contributions, union dues, and other items. Then, after the employee's paycheck is output, the next employee's data would be entered, and the `detailLoop()` module would repeat. The advantage to having a computer produce payroll checks is that the calculation instructions need to be written only once and can be repeated indefinitely.

Watch the video *A Quick Introduction to Loops*.

Two Truths & A Lie

Understanding the Advantages of Looping

1. When you use a loop, you can write one set of instructions that operates on multiple, separate sets of data.

   T F

2. A major advantage of having a computer perform complicated tasks is the ability to repeat them.

   T F

3. A loop is a structure that branches in two logical paths before continuing.

   T F

Chapter 5: Looping: 5-2 Using a Loop Control Variable
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

5-2 Using a Loop Control Variable

You can use a `while` loop to execute a body of statements continuously as long as some condition continues to be true. The body of a loop might contain any number of statements, including method calls, decisions, and other loops. To make a `while` loop end correctly, you should declare a **loop control variable** (A variable that determines whether a loop will continue.) to manage the number of repetitions a loop performs.

Three separate actions should occur:

- The loop control variable is initialized before entering the loop.
- The loop control variable is tested, and if the result is true, the loop body is entered.
- The loop control variable is altered within the body of the loop so that the `while` expression eventually evaluates as false.

If you omit any of these actions or perform them incorrectly, you run the risk of creating an infinite loop. Once your logic enters the body of a structured loop, the entire loop body must execute. Your program can leave a structured loop only at the comparison that tests the loop control variable. Commonly, you can control a loop's repetitions in one of two ways:

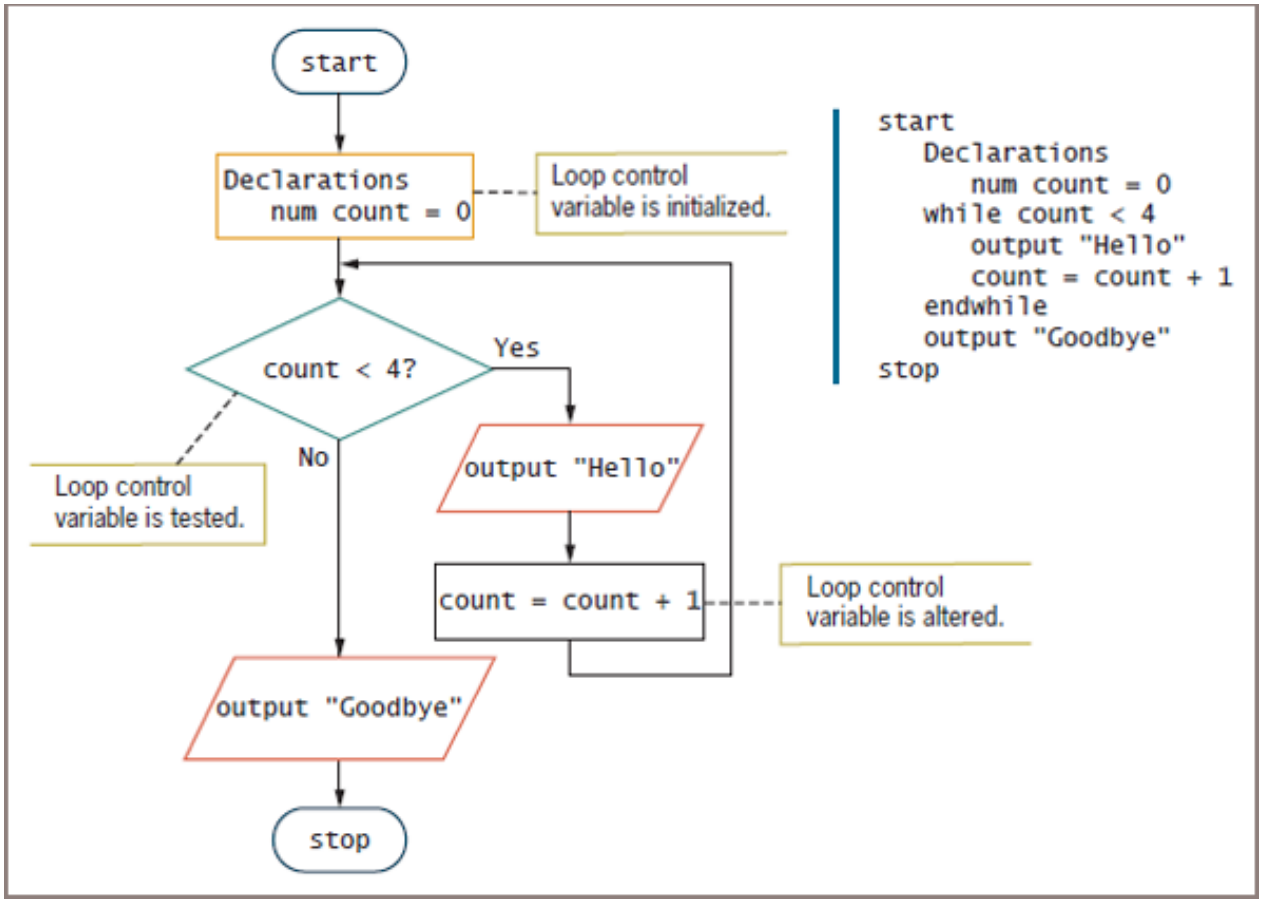
- Use a counter to create a definite, counter-controlled loop.
- Use a sentinel value to create an indefinite loop.

5-2a Using a Definite Loop with a Counter

Figure 5-3 shows a loop that displays *Hello* four times. The variable `count` is the loop control variable. This loop is a **definite loop** (A loop for which the number of repetitions is a predetermined value.) because it executes a definite, predetermined number of times—in this case, four. The loop is a **counted loop** (A counted loop, or counter-controlled loop, is a loop whose repetitions are managed by a counter.) , or **counter-controlled loop** (A counted loop, or counter-controlled loop, is a loop whose repetitions are managed by a counter.) , because the program keeps track of the number of loop repetitions by counting them.

Figure 5-3

A Counted `while` Loop That Outputs *Hello* Four Times



The loop in Figure 5-3 executes as follows:

- The loop control variable, `count`, is initialized to 0.
- The `while` expression compares `count` to 4.
- The value of `count` is less than 4, and so the loop body executes. The loop body shown

in Figure 5-3 consists of two statements that display *Hello* and then add 1 to `count`.

- The next time `count` is evaluated, its value is 1, which is still less than 4, so the loop body executes again. *Hello* is displayed a second time and `count` becomes 2, *Hello* is displayed a third time and `count` becomes 3, then *Hello* is displayed a fourth time and `count` becomes 4. Now when the expression `count < 4?` evaluates, it is `false`, so the loop ends.

Within a loop's body, you can change the value of the loop control variable in a number of ways. For example:

- You might simply assign a new value to the loop control variable.
- You might retrieve a new value from an input device.
- You might **increment** (To change a variable by adding a constant value to it, frequently 1.) , or increase, the loop control variable, as in the logic in Figure 5-3.
- You might reduce, or **decrement** (To change a variable by decreasing it by a constant value, frequently 1.) , the loop control variable. For example, the loop in Figure 5-3 could be rewritten so that `count` is initialized to 4, and reduced by 1 on each pass through the loop. The loop should then continue while `count` remains greater than 0.

The terms *increment* and *decrement* usually refer to small changes; often the value used to increase or decrease the loop control variable is 1. However, loops are also controlled by adding or subtracting values other than 1. For example, to display company profits at five-year intervals for the next 50 years, you would want to add 5 to a loop control variable during each iteration.

Because you frequently need to increment a variable, many programming languages contain a shortcut operator for incrementing. You will learn about these shortcut operators when you study a programming language that uses them.

Watch the video *Looping*.

The looping logic shown in Figure 5-3 uses a counter. A **counter** (Any numeric variable used to count the number of times an event has occurred.) is any numeric variable you use to count the number of times an event has occurred. In everyday life, people usually count things starting with 1. Many programmers prefer starting their counted loops with a

variable containing 0 for two reasons:

- In many computer applications, numbering starts with 0 because of the 0-and-1 nature of computer circuitry.
- When you learn about arrays in Chapter 6, you will discover that array manipulation naturally lends itself to 0-based loops.

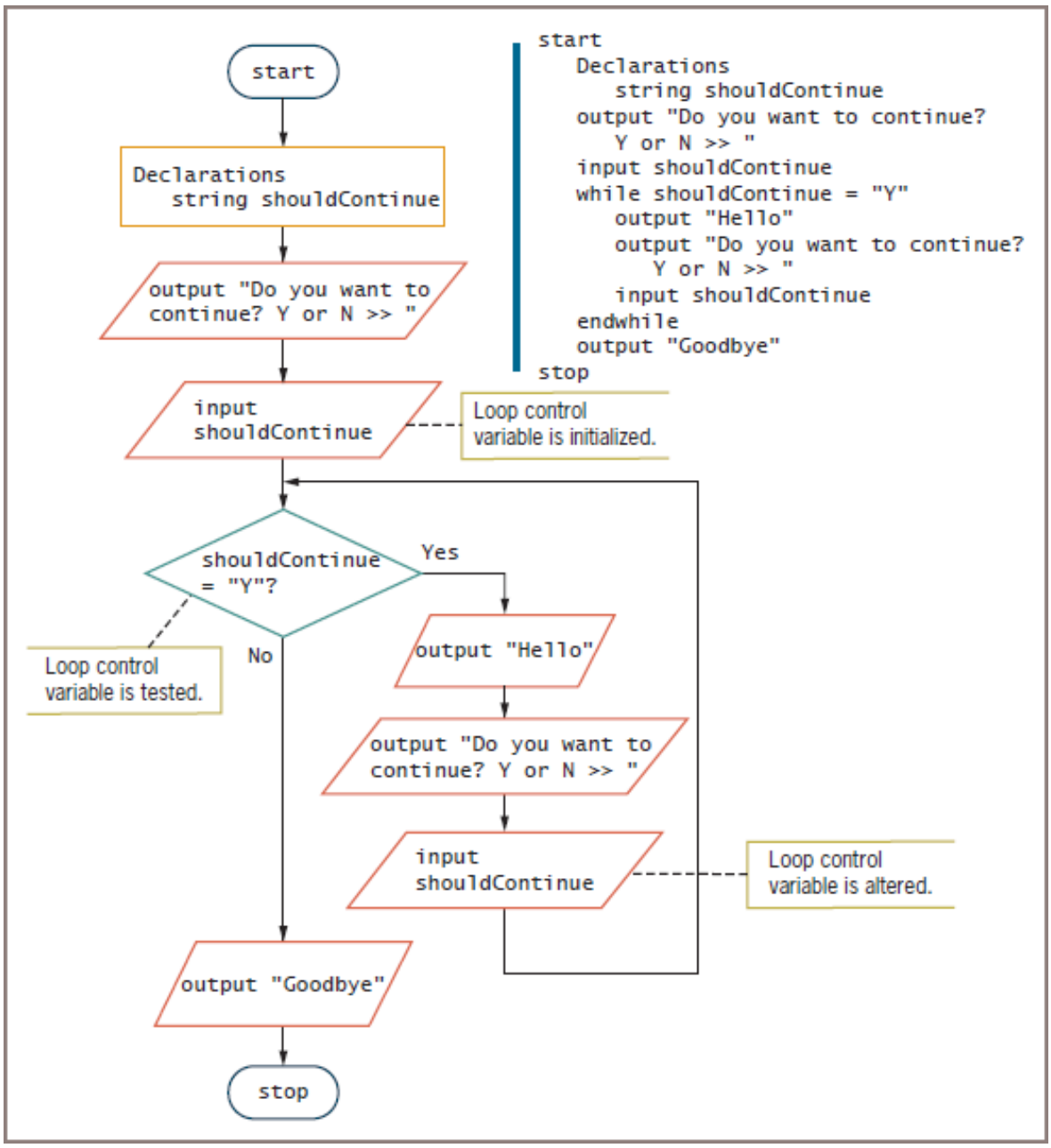
5-2b Using an Indefinite Loop with a Sentinel Value

Often, the value of a loop control variable is not altered by arithmetic, but instead is altered by user input. For example, perhaps you want to keep performing some task while the user indicates a desire to continue. In that case, you do not know when you write the program whether the loop will be executed two times, 200 times, or at all. This type of loop is an **indefinite loop** (A loop for which the number of executions cannot be predicted when the program is written.) .

Consider an interactive program that displays *Hello* repeatedly as long as the user wants to continue. The loop is indefinite because each time the program executes, the loop might be performed a different number of times. The program appears in Figure 5-4.

Figure 5-4

An Indefinite `while` Loop That Displays *Hello* As Long As the User Wants to Continue



In the program in Figure 5-4, the loop control variable is `shouldContinue`. The program executes as follows:

- The first `input shouldContinue` statement in the application in Figure 5-4 is a priming input statement. In this statement, the loop control variable is initialized by the user's first response.
- The `while` expression compares the loop control variable to the sentinel value *Y*.
- If the user has entered *Y*, then *Hello* is output and the user is asked whether the program should continue. In this step, the value of `shouldContinue` might change.
- At any point, if the user enters any value other than *Y*, the loop ends. In most programming languages, comparisons are case sensitive, so any entry other than *Y*, including *y*, will end the loop.

Figure 5-5 shows how the program might look when it is executed at the command line and in a GUI environment. The screens in Figure 5-5 show programs that perform exactly the same tasks using different environments. In each environment, the user can continue choosing to see *Hello* messages, or can choose to quit the program and display *Goodbye*.

Figure 5-5

Typical Executions of the Program in Figure 5-4 in Two Environments

5-2c Understanding the Loop in a Program's Mainline Logic

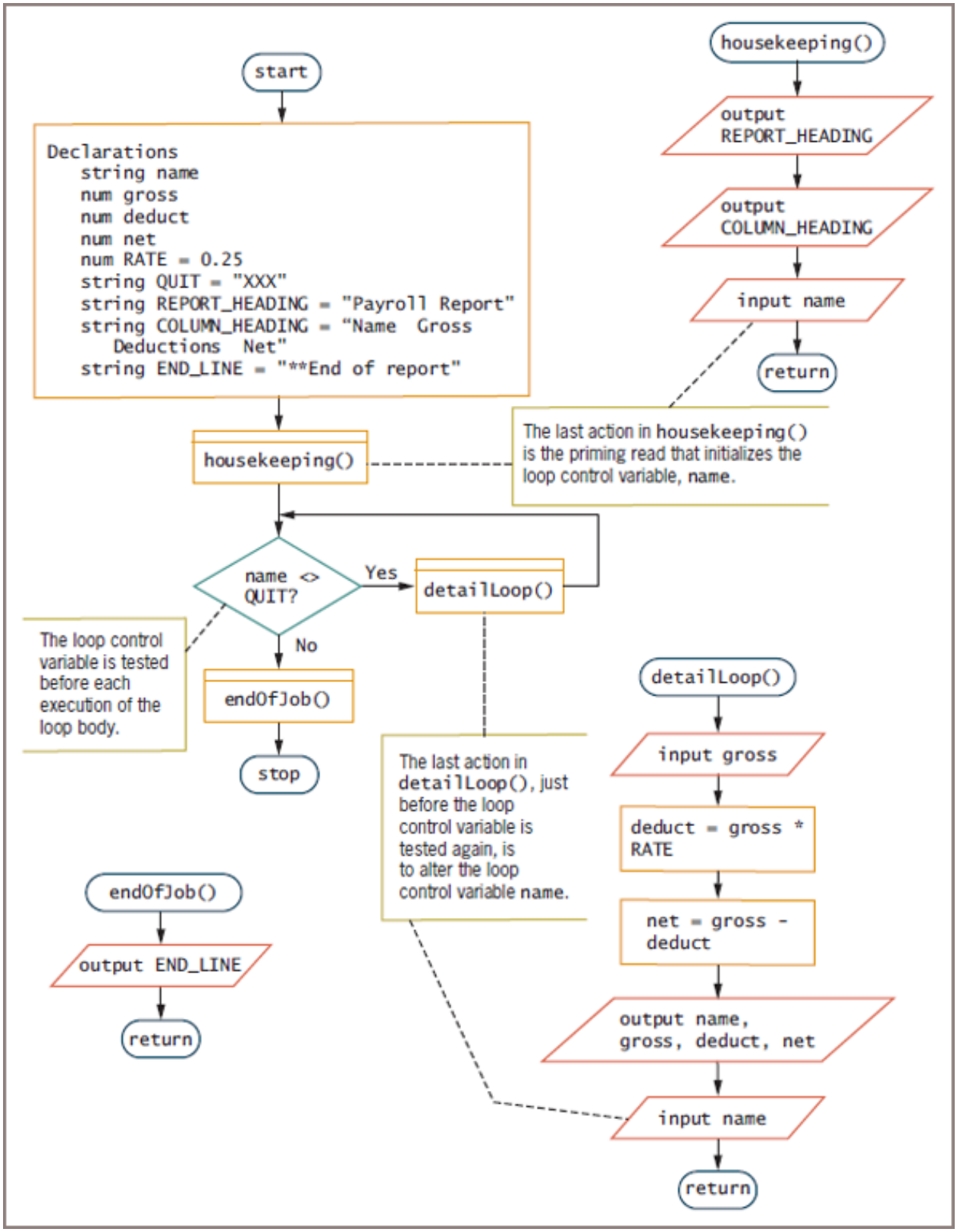
The flowchart and pseudocode segments in Figure 5-4 contain three steps that should occur in every properly functioning loop:

1. You must provide a starting value for the variable that will control the loop.
2. You must test the loop control variable to determine whether the loop body executes.
3. Within the loop, you must alter the loop control variable.

In Chapter 2 you learned that the mainline logic of many business programs follows a standard outline that consists of housekeeping tasks, a loop that repeats, and finishing tasks. The three crucial steps that occur in any loop also occur in standard mainline logic. Figure 5-6 shows the flowchart for the mainline logic of the payroll program that you saw in Figure 2-8. In Figure 5-6, the three loop-controlling steps are highlighted. In this case, the three steps—initializing, testing, and altering the loop control variable—are in different modules. However, the steps all occur in the correct places, showing that the mainline logic uses a standard and correct loop.

Figure 5-6

A Payroll Program Showing How the Loop Control Variable is Used



Two Truths & A Lie

Using a Loop Control Variable

1. To make a `while` loop execute correctly, a loop control variable must be

set to 0 before entering the loop.

T F

2. To make a `while` loop execute correctly, a loop control variable should be tested before entering the loop body.

   T F

3. To make a `while` loop execute correctly, the body of the loop must take some action that alters the value of the loop control variable.

   T F

5-3 Nested Loops

Program logic gets more complicated when you must use loops within loops, or **nested loops** (A loop structure within another loop structure; nesting loops are loops within loops.) . When one loop appears inside another, the loop that contains the other loop is called the **outer loop** (The loop that contains a nested loop.) , and the loop that is contained is called the **inner loop** (When loops are nested, the loop that is contained within the other loop.) . You need to create nested loops when the values of two or more variables repeat to produce every combination of values. Usually, when you create nested loops, each loop has its own loop control variable.

For example, suppose you want to write a program that produces quiz answer sheets like the ones shown in Figure 5-7. Each answer sheet has a unique heading followed by five parts with three questions in each part, and you want a fill-in-the-blank line for each question. You could write a program that uses 63 separate output statements to produce three sheets (each sheet contains 21 printed lines), but it is more efficient to use nested loops.

Figure 5-7

Quiz Answer Sheets

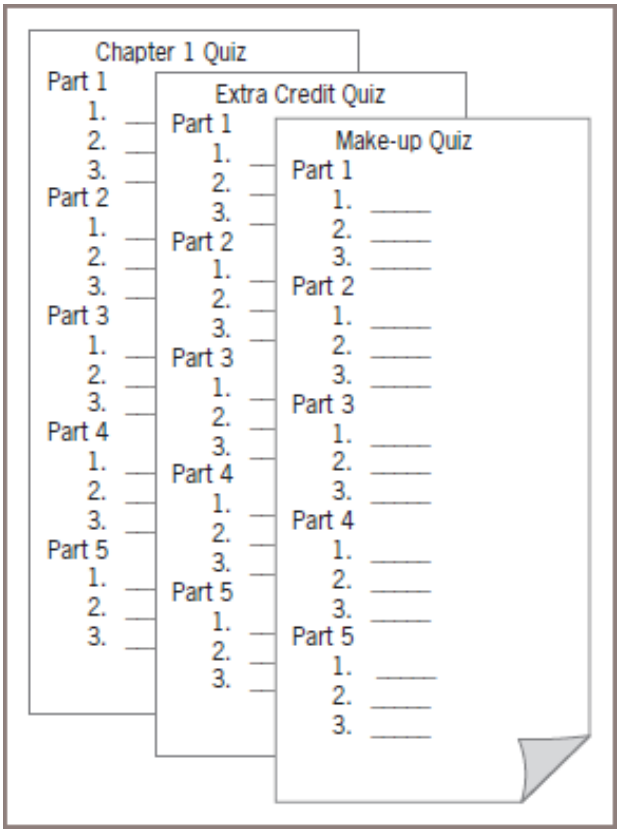
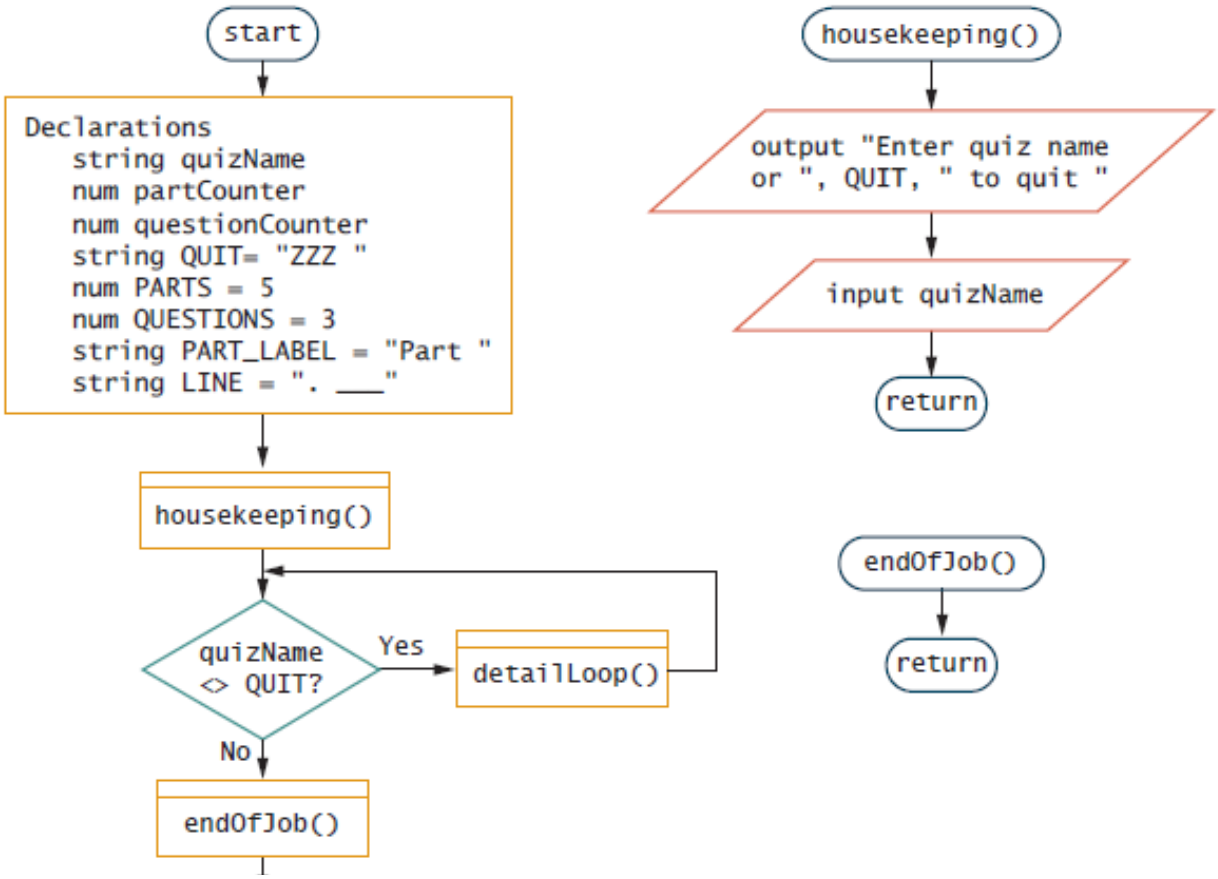
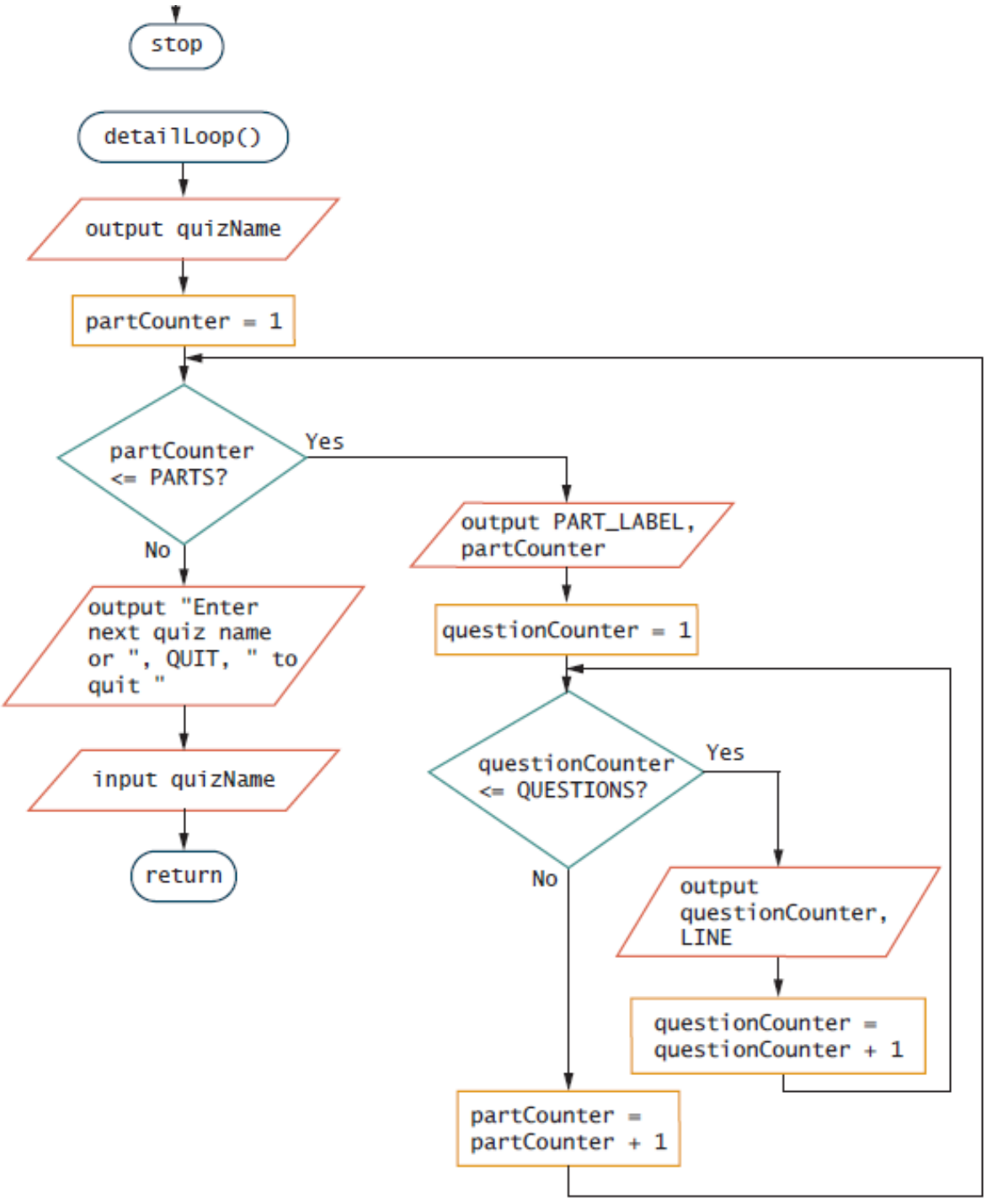


Figure 5-8 shows the logic for the program that produces answer sheets. Three loop control variables are declared for the program:

Figure 5-8

Flowchart and Pseudocode for `answersheet` Program






```
start
   Declarations
      string quizName
      num partCounter
      num questionCounter
      string QUIT = "ZZZ "
      num PARTS = 5
      num QUESTIONS = 3
      string PART_LABEL = "Part "
      string LINE = ". _____"
   housekeeping()
   while quizName <> QUIT
      detailLoop()
   endwhile
   endOfJob()
stop
```

```
housekeeping()
   output "Enter quiz name or ", QUIT, " to quit "
   input quizName
return

detailLoop()
   output quizName
   partCounter = 1
   while partCounter <= PARTS
      output PART_LABEL, partCounter
      questionCounter = 1
      while questionCounter <= QUESTIONS
         output questionCounter, LINE
         questionCounter = questionCounter + 1
      endwhile
      partCounter = partCounter + 1
   endwhile
   output "Enter next quiz name or ", QUIT, " to quit "
   input quizName
return

endOfJob()
return
```

- `quizName` controls the `detailLoop()` module that is called from the mainline logic.
- `partCounter` controls the outer loop within the `detailLoop()` module; it keeps track of the answer sheet parts.
- `questionCounter` controls the inner loop in the `detailLoop()` module; it keeps track of the questions and answer lines within each part section on each answer sheet.

Five named constants are also declared. Three of these constants (`QUIT`, `PARTS`, and `QUESTIONS`) hold the sentinel values for each of the three loops in the program. The other two hold the text that will be output (the word *Part* that precedes each part number, and the period-space-underscore combination that forms a fill-in line for each question).

When the program starts, the `housekeeping()` module executes and the user enters the name to be output at the top of the first quiz. If the user enters the `QUIT` value, the program ends immediately, but if the user enters anything else, such as *Make-up Quiz*, then the `detailLoop()` module executes.

In the `detailLoop()` the quiz name is output at the top of the answer sheet. Then `partCounter` is initialized to 1. The `partCounter` variable is the loop control variable for the outer loop in this module. The outer loop continues while `partCounter` is less than or equal to `PARTS`. The last statement in the outer loop adds 1 to `partCounter`. In other words, the outer loop will execute when `partCounter` is 1, 2, 3, 4, and 5.

In Figure 5-8, some output (the user prompt) would be sent to one output device, such as a monitor. Other output (the quiz sheet) would be sent to another

output device, such as a printer. The statements needed to send output to separate devices differs among languages. Chapter 7 provides more details.

The `endOfJob()` module is included in the program in Figure 5-8, even though it contains no statements, so that the mainline logic contains all the parts you have learned. An empty module that acts as a placeholder is called a **stub** (A method without statements that is used as a placeholder.) .

In the outer loop in the `detailLoop()` module in Figure 5-8, the word *Part* and the current `partCounter` value are output. Then the following steps execute:

- The loop control variable for the inner loop is initialized by setting `questionCounter` to 1.
- The loop control variable `questionCounter` is evaluated by comparing it to `QUESTIONS`, and while `questionCounter` does not exceed `QUESTIONS`, the loop body executes: The value of `questionCounter` is output, followed by a period and a fill-in-the-blank line.
- At the end of the loop body, the loop control variable is altered by adding 1 to `questionCounter` and the `questionCounter` comparison is made again.

In other words, when `partCounter` is 1, the part heading is output and underscore lines are output for questions 1, 2, and 3. Then `partCounter` becomes 2, the part heading is output, and underscore lines are created for another set of questions 1, 2, and 3. Then `partCounter` becomes 3, 4, and 5 in turn, and three underscore lines are created for each part.

In the program in Figure 5-8, it is important that `questionCounter` is reset to 1 within the outer loop, just before entering the inner loop. If this step was omitted, Part 1 would contain questions 1, 2, and 3, but subsequent parts would be empty.

Watch the video *Nested Loops*.

Two Truths & A Lie

Nested Loops

1. When one loop is nested inside another, the loop that contains the other loop is called the outer loop.

   T F

2. You need to create nested loops when the values of two or more variables repeat to produce every combination of values.

   T F

3. The number of times a loop executes always depends on a constant.

   T F

5-4 Avoiding Common Loop Mistakes

Programmers make the following common mistakes with loops:

- Neglecting to initialize the loop control variable
- Neglecting to alter the loop control variable
- Using the wrong comparison with the loop control variable
- Including statements inside the loop that belong outside the loop

The following sections explain these common mistakes in more detail.

5-4a Mistake: Neglecting to Initialize the Loop Control Variable

Failing to initialize a loop's control variable is a mistake. For example, consider the program in Figure 5-9. It prompts the user for a name, and while the value of `name` continues not to be the sentinel value *ZZZ*, it outputs a greeting that uses the name and asks for the next name. This program works correctly.

Figure 5-9

Correct Logic for Greeting Program

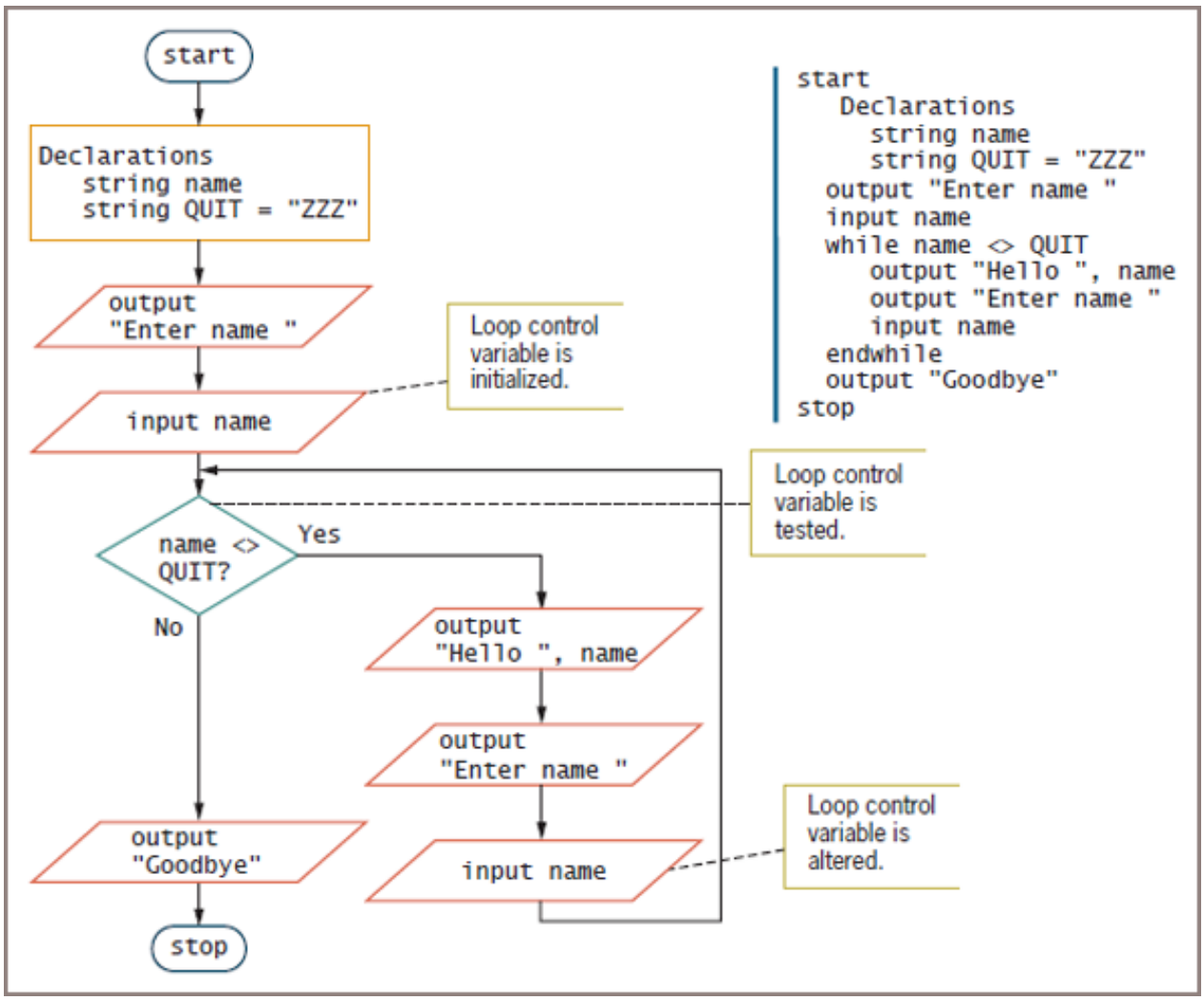
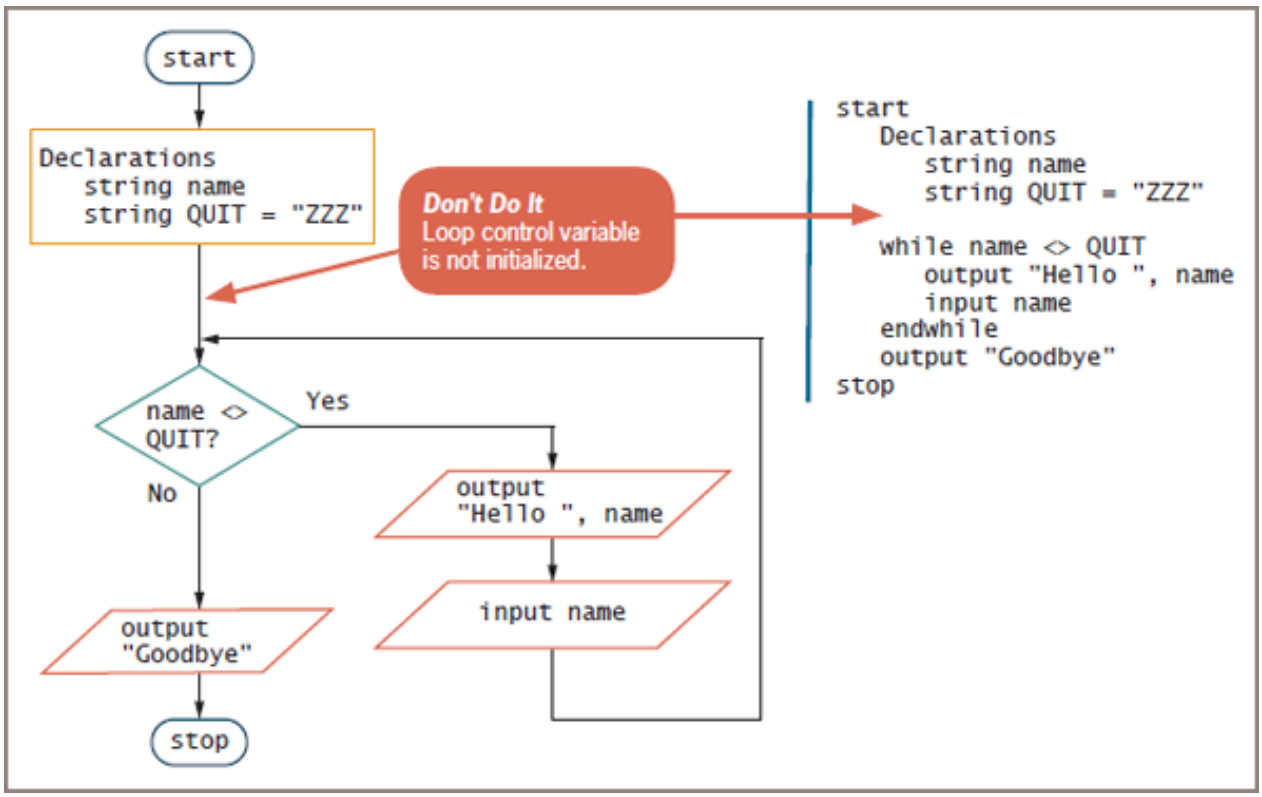


Figure 5-10 shows an incorrect program in which the loop control variable is not assigned a starting value. If the `name` variable is not set to a starting value, then when the `eof` condition is tested, there is no way to predict whether it will be true. If the user does not enter a value for `name`, the garbage value originally held by that variable might or might not be *ZZZ*. So, one of two scenarios follows:

Figure 5-10

Incorrect Logic for Greeting Program because the Loop Control Variable Initialization is Missing



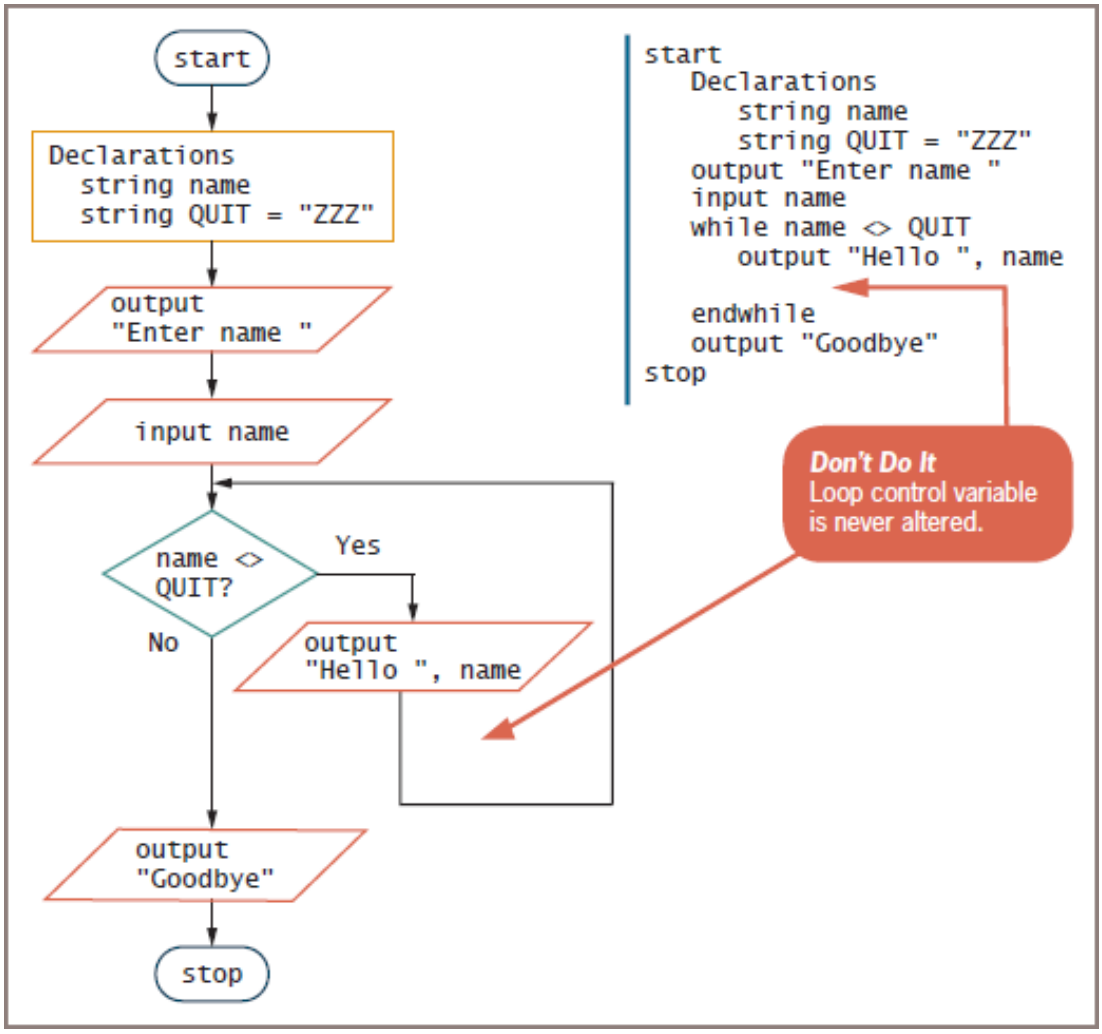
- Most likely, the uninitialized value of `name` is not *ZZZ*, so the first greeting output will include garbage—for example, *Hello 12BGr5*.
- By a remote chance, the uninitialized value of `name` *is ZZZ*, so the program ends immediately before the user can enter any names.

5-4b Mistake: Neglecting to Alter the Loop Control Variable

Different sorts of errors will occur if you fail to alter a loop control variable within the loop. For example, in the program in Figure 5-9 that accepts and displays names, you create such an error if you don't accept names within the loop. Figure 5-11 shows the resulting incorrect logic.

Figure 5-11

Incorrect Logic for Greeting Program because the Loop Control Variable is Not Altered



If you remove the `input name` instruction from the end of the loop in the program, no name is ever entered after the first one. For example, assume that when the program starts, the user enters *Fred*. The name will be compared to the sentinel value, and the loop will be entered. After a greeting is output for Fred, no new name is entered, so when the logic returns to the loop-controlling question, the `name` will still not be *ZZZ*, and greetings for Fred will continue to be output infinitely. You never want to create a loop that cannot terminate.

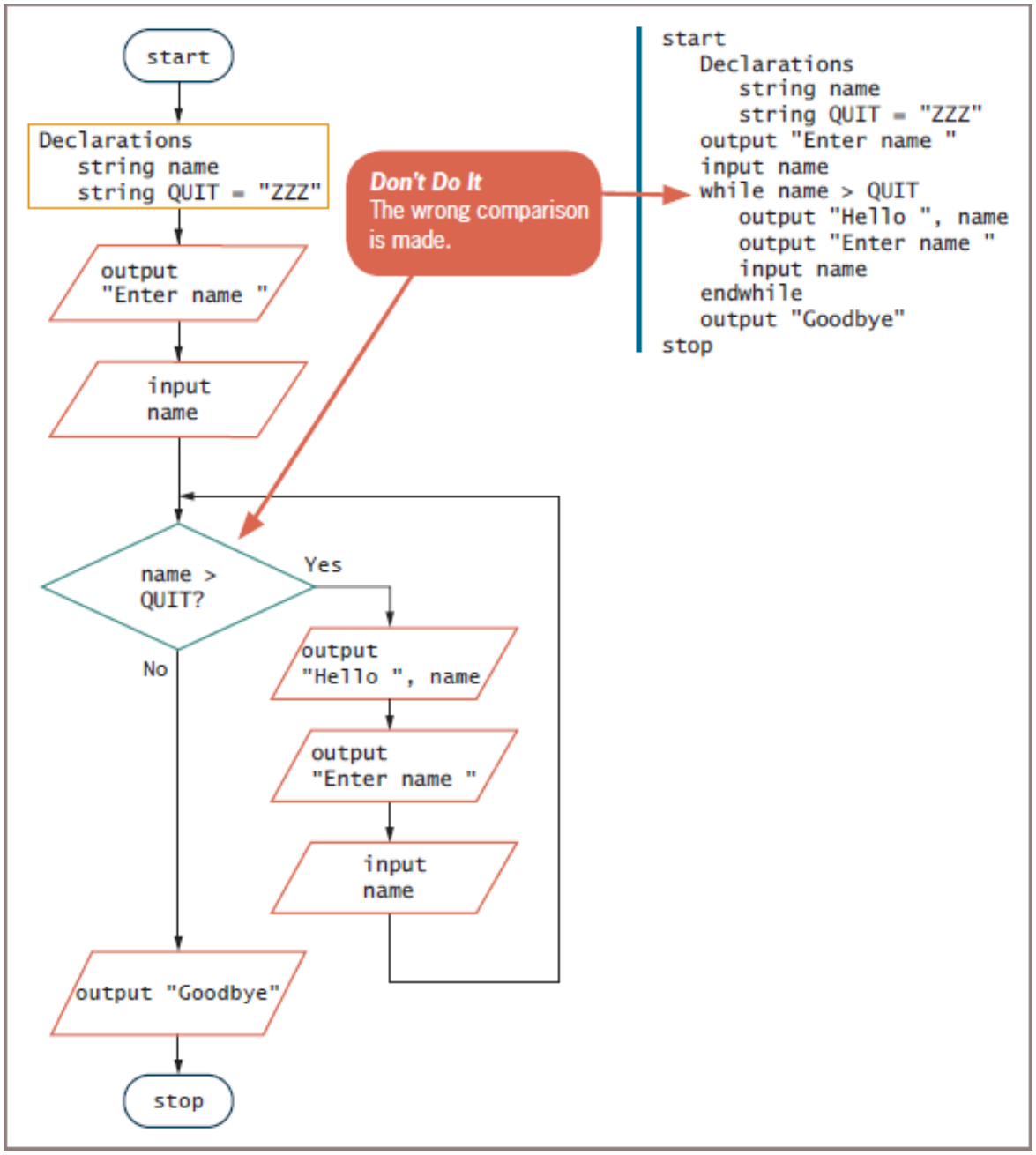
Chapter 5: Looping: 5-4c Mistake: Using the Wrong Comparison with the Loop Control Variable
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

5-4c Mistake: Using the Wrong Comparison with the Loop Control Variable

Programmers must be careful to use the correct comparison in the statement that controls a loop. A comparison is correct only when the appropriate operands and operator are used. For example, although only one keystroke differs between the original greeting program in Figure 5-9 and the one in Figure 5-12, the original program correctly produces named greetings and the second one does not.

Figure 5-12

Incorrect Logic for Greeting Program because the Wrong Test is Made with the Loop Control Variable



In Figure 5-12, a greater-than comparison (>) is made instead of a not-equal-to (<>) comparison. Suppose that when the program executes, the user enters *Fred* as the first name. In most programming languages, when the comparison between *Fred* and *ZZZ* is made, the values are compared alphabetically. *Fred* is not greater than *ZZZ*, so the loop is never entered, and the program ends.

Using the wrong comparison can have serious effects. For example, in a counted loop, if you use <= instead of < to compare a counter to a sentinel value, the program will perform one loop execution too many. If the loop only displays greetings, the error might not be critical, but if such an error occurred in a loan company application, each customer might be charged a month's additional interest. If the error occurred in an airline's application, it might overbook a flight. If the error occurred in a pharmacy's drug-dispensing application,

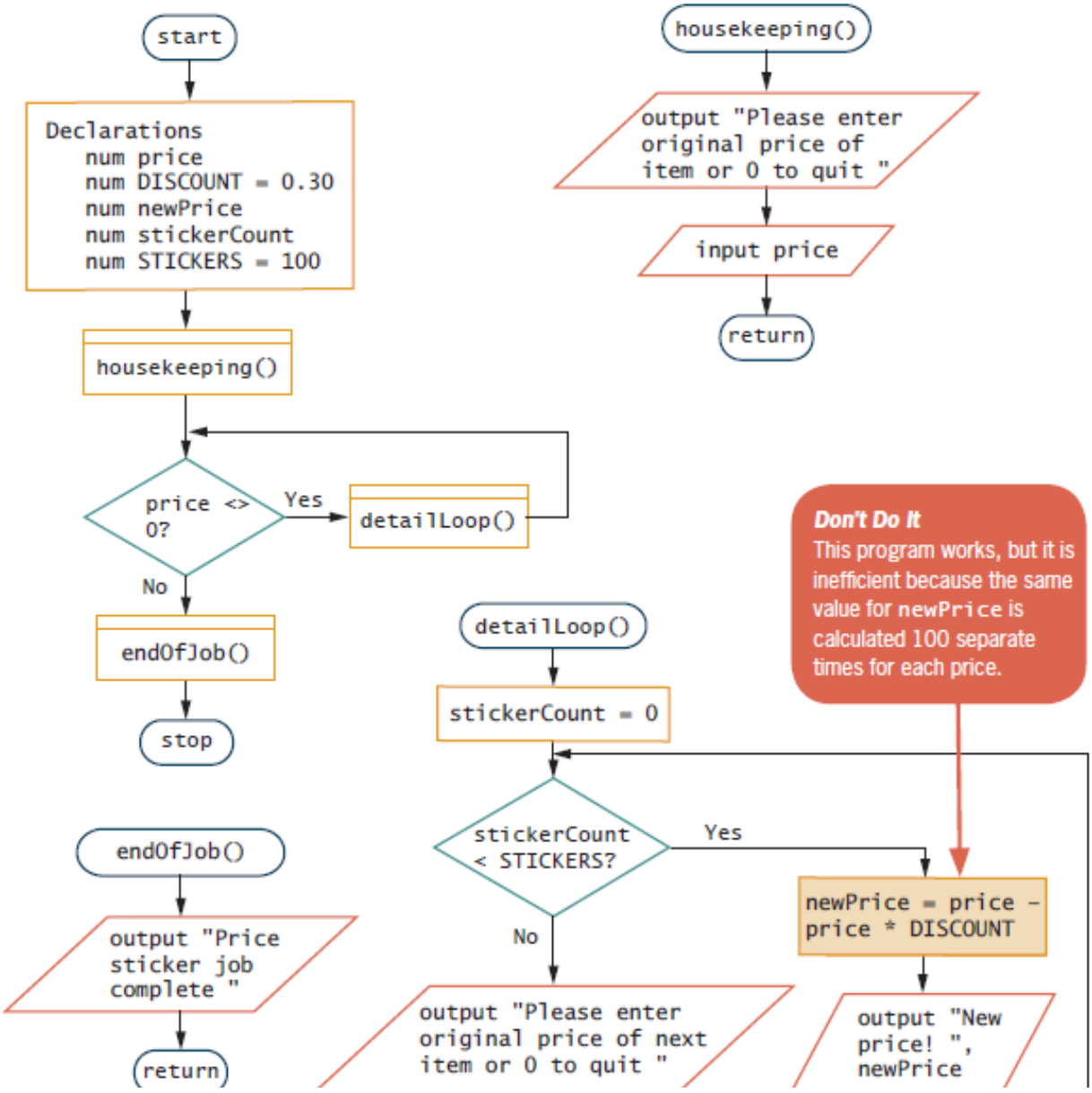
each patient might receive one extra (and possibly harmful) unit of medication.

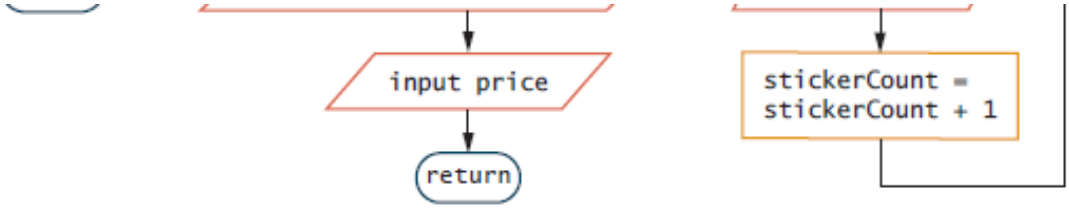
5-4d Mistake: Including Statements inside the Loop That Belong Outside the Loop

Suppose that you write a program for a store manager who wants to discount every item he sells by 30 percent. The manager wants 100 new price label stickers for each item. The user enters a price, the new discounted price is calculated, 100 stickers are printed, and the next price is entered. Figure 5-13 shows a program that performs the job inefficiently because the same value, `newPrice`, is calculated 100 separate times for each `price` that is entered.

Figure 5-13

Inefficient Way to Produce 100 Discount Price Stickers for Differently Priced Items






```
start
   Declarations
      num price
      num DISCOUNT = 0.30
      num newPrice
      num stickerCount
      num STICKERS = 100
   housekeeping()
   while price <> 0
      detailLoop()
   endwhile
   endOfJob()
stop

housekeeping()
   output "Please enter original price of item or 0 to quit "
   input price
return

detailLoop()
   stickerCount = 0
   while stickerCount < STICKERS
      newPrice = price - price * DISCOUNT
      output "New price! ", newPrice
      stickerCount = stickerCount + 1
   endwhile
   output "Please enter original price of
      next item or 0 to quit "
   input price
return

endOfJob()
   output "Price sticker job complete"
return
```

Don't Do It
This program works, but it is inefficient because the same value for `newPrice` is calculated 100 separate times for each price.

Figure 5-14 shows the same program, in which the `newPrice` value that is output on the sticker is calculated only once per new price; the calculation has been moved to a better location. The programs in Figures 5-13 and 5-14 do the same thing, but the second program does it more efficiently. As you become more proficient at programming, you will recognize many opportunities to perform the same tasks in alternate, more elegant, and more efficient ways.

Figure 5-14

Improved Discount Sticker-Making Program

When you describe people or events as elegant, you mean they possess a refined gracefulness. Similarly, programmers use the term *elegant* to describe programs that are well designed and easy to understand and maintain.

Two Truths & A Lie

Avoiding Common Loop Mistakes

1. In a loop, neglecting to initialize the loop control variable is a mistake.

   T F

2. In a loop, neglecting to alter the loop control variable is a mistake.

   T F

3. In a loop, comparing the loop control variable using >= or <= is a mistake.

   T F

Chapter 5: Looping: 5-5 Using a `for` Loop
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

5-5 Using a `for` Loop

Every high-level programming language contains a `while` statement that you can use to code any loop, including both indefinite and definite loops. In addition to the `while` statement, most computer languages support a `for` statement. You usually use the **for statement** (A statement that can be used to code definite loops; also called a *for loop*. The statement contains a loop control variable that it automatically initializes, evaluates, and alters.) , or **for loop** (A statement that can be used to code definite loops; also called a *for loop*. The statement contains a loop control variable that it automatically initializes, evaluates, and alters.) , with definite loops—those that will loop a specific number of times—when you know exactly how many times the loop will repeat. The `for` statement provides you with three actions in one compact statement. In a `for` statement, a loop control variable is:

- Initialized
- Evaluated
- Altered

The `for` statement takes a form similar to the following:

```
for loopControlVariable = initialValue to finalValue step stepValue
   do something
```

The amount by which a `for` loop control variable changes is often called a **step value** (A step value is a number you use to increase a loop control variable on each pass through a loop.) . The step value can be positive or negative; that is, it can increment or decrement.

For example, to display *Hello* four times, you can write either of the sets of statements in Figure 5-15.

Figure 5-15

Comparable `while` and `for` Statements That Each Output *Hello* Four Times

```
count = 0
while count <= 3
   output "Hello"
   count = count + 1
endwhile
```

```
for count = 0 to 3 step 1
   output "Hello"
endfor
```

The code segments in Figure 5-15 each accomplish the same tasks:

- The variable `count` is initialized to 0.
- The `count` variable is compared to the limit value 3; while `count` is less than or equal to 3, the loop body executes.
- As the last statement in the loop execution, the value of `count` increases by 1. After the increase, the comparison to the limit value is made again.

The `for` loop simply expresses the same logic in a more compact form than the `while` statement. You never are required to use a `for` statement for any loop; a `while` statement can always be used instead. However, when a loop's execution is based on a loop control variable progressing from a known starting value to a known ending value in equal steps, the `for` loop provides you with a convenient shorthand. It is easy for others to read, and because the loop control variable's initialization, testing, and alteration are all performed in one location, you are less likely to leave out one of these crucial elements.

Although `for` loops are commonly used to control execution of a block of statements a fixed number of times, the programmer doesn't need to know the starting, ending, or step value for the loop when the program is written. For example, any of the values might be entered by the user, or might be the result of a calculation.

The `for` loop is particularly useful when processing arrays. You will learn about arrays in Chapter 6.

In Java, C++, and C#, a `for` loop that displays 21 values (0 through 20) might look similar to the following:

The three actions (initialization, comparison, and altering of the loop control variable) are separated by semicolons within a set of parentheses that follow the keyword `for`. The expression `count++` adds 1 to `count`. The block of statements that depends on the loop sits between a pair of curly braces.

Both the `while` loop and the `for` loop are examples of pretest loops. In a **pretest loop** (A loop that tests its loop control variable before each iteration, meaning that the loop body might never execute.) , the loop control variable is tested before each iteration. That means the loop body might never execute because the question controlling the loop might be false the first time it is asked. Most languages allow you to use a variation of the looping structure known as a **posttest loop** (A loop that tests its loop control variable after each iteration, meaning that the loop body executes at least one time.) , which tests the loop control variable after each iteration. In a posttest loop, the loop body executes at least one time because the loop control variable is not tested until after one iteration. Appendix F contains information about posttest loops.

Some books and flowchart programs use a symbol that looks like a hexagon to represent a `for` loop in a flowchart. However, no special symbols are needed to express a `for` loop's logic. A `for` loop is simply a code shortcut, so this book uses standard flowchart symbols to represent initializing the loop control variable, testing it, and altering it.

Two Truths & A Lie

Using a `for` Loop

1. The `for` statement provides you with three actions in one compact statement: initializing, evaluating, and altering a loop control variable.

   T F

2. A `for` statement body always executes at least one time.

   T F

3. In most programming languages, you can provide a `for` loop with any step value.

   T F

5-6 Common Loop Applications

Although every computer program is different, many techniques are common to a variety of applications. Loops, for example, are frequently used to accumulate totals and to validate data.

5-6a Using a Loop to Accumulate Totals

Business reports often include totals. The supervisor who requests a list of employees in the company dental plan is often as interested in the number of participating employees as in who they are. When you receive your telephone bill each month, you usually check the total as well as charges for the individual calls.

Assume that a real estate broker wants to see a list of all properties sold in the last month as well as the total value for all the properties. A program might accept sales data that includes the street address of each property sold and its selling price. The data records might be entered by a clerk as each sale is made, and stored in a file until the end of the month; then they can be used in a monthly report. Figure 5-16 shows an example of such a report.

Figure 5-16

Month-end Real Estate Sales Report

To create the sales report, you must output the address and price for each property sold and add its value to an accumulator. An **accumulator** (A variable used to gather or accumulate values.) is a variable that you use to gather or accumulate values, and is very similar to a counter that you use to count loop iterations. However, usually you add just one to a counter, whereas you add some other value to an accumulator. If the real estate broker wants to know how many listings the company holds, you *count* them. When the broker wants to know the total real estate value, you *accumulate* it.

To accumulate total real estate prices, you declare a numeric variable such as `accumPrice` and initialize it to 0. As you get data for each real estate transaction, you output it and add its value to the accumulator `accumPrice`, as shown shaded in Figure 5-17.

Figure 5-17

Flowchart and Pseudocode for Real Estate Sales Report Program

Some programming languages assign 0 to a variable you fail to initialize explicitly, but many do not—when you try to add a value to an uninitialized variable, they either issue an error message or let you incorrectly start with an accumulator that holds garbage. The safest and clearest course of action is to assign the value 0 to accumulators before using them.

After the program in Figure 5-17 gets and displays the last real estate transaction, the user enters the sentinel value and loop execution ends. At that point, the accumulator will hold the grand total of all the real estate values. The program displays the word *Total* and the accumulated value `accumPrice`. Then the program ends.

Figure 5-17 highlights the three actions you usually must take with an accumulator:

- Accumulators are initialized to 0.
- Accumulators are altered, usually once for every data set processed.
- At the end of processing, accumulators are output.

After outputting the value of `accumPrice`, new programmers often want to reset it to 0. Their argument is that they are "cleaning up after themselves." Although you can take this step without harming the execution of the program, it serves no useful purpose. You cannot set `accumPrice` to 0 in anticipation of having it ready for the next program, or even for the next time you execute the same program. Variables exist only during an execution of the program, and even if a future application happens to contain a variable named `accumPrice`, the variable will not necessarily occupy the same memory location as this one. Even if you run the same application a second time, the variables might occupy different physical memory locations from those during the first run. At the beginning of any method, it is the programmer's responsibility to initialize all variables that must start with a specific value. There is no benefit to changing a variable's value when it will never be used again during the current execution.

Some business reports are **summary reports** (A report that lists only totals, without individual detail records.) —they contain only totals with no data for individual records. In the example in Figure 5-17, suppose that the broker did not care about details of individual sales, but only about the total for all transactions. You could create a summary report by omitting the step that outputs `address` and `price` from the `createReport()` module. Then you could simply output `accumPrice` at the end of the program.

5-6b Using a Loop to Validate Data

When you ask a user to enter data into a computer program, you have no assurance that the data will be accurate. Incorrect user entries are by far the most common source of computer errors. The programs you write will be improved if you employ **defensive programming** (A technique in which programmers try to prepare for all possible errors before they occur.) , which means trying to prepare for all possible errors before they occur. Loops are frequently used to **validate data** (Ensuring that data falls within an acceptable range.) —that is, to make sure it is meaningful and useful. For example, validation might ensure that a value is the correct data type or that it falls within an acceptable range.

Suppose that part of a program you are writing asks a user to enter a number that represents his or her birth month. If the user types a number lower than 1 or greater than 12, you must take some sort of action. For example:

- You could display an error message and stop the program.
- You could choose to assign a default value for the month (for example, 1) before proceeding.
- You could reprompt the user for valid input.

If you choose this last course of action, you could then take at least two approaches. You could use a selection, and if the month is invalid, you could ask the user to reenter a number, as shown in Figure 5-18.

Figure 5-18

Reprompting a User Once After an Invalid Month is Entered

The problem with the logic in Figure 5-18 is that the user still might not enter valid data on the second attempt. Of course, you could add a third decision, but you still couldn't control what the user enters.

The superior solution is to use a loop to continuously prompt a user for a month until the user enters it correctly. Figure 5-19 shows this approach.

Figure 5-19

Reprompting a User Continuously After an Invalid Month is Entered

Most languages provide a built-in way to check whether an entered value is numeric. When you rely on user input, you frequently accept each piece of input data as a string and then attempt to convert it to a number. The procedure for accomplishing numeric checks varies slightly in different programming languages.

Of course, data validation doesn't prevent all errors; just because a data item is valid does not mean that it is correct. For example, a program can determine that 5 is a valid birth month, but not that your birthday actually falls in month 5. Programmers employ the acronym **GIGO** (Acronym for *garbage in, garbage out*; it means that if input is incorrect, output is worthless.) for *garbage in, garbage out*. It means that if your input is incorrect, your output is worthless.

5-6c Limiting a Reprompting Loop

Reprompting a user is a good way to ensure valid data, but it can be frustrating to a user if it continues indefinitely. For example, suppose the user must enter a valid birth month, but has used another application in which January was month 0, and keeps entering 0 no matter how many times you repeat the prompt. One helpful addition to the program would be to use the limiting values as part of the prompt. In other words, instead of the statement output `"Enter birth month… "`, the following statement might be more useful:

Still, the user might not understand the prompt or not read it carefully, so you might want to employ the tactic used in Figure 5-20, in which the program maintains a count of the number of reprompts. In this example, a constant named `ATTEMPTS` is set to 3. While a count of the user's attempts at correct data entry remains below this limit, and the user enters invalid data, the user continues to be reprompted. If the user exceeds the limited number of allowed attempts, the loop ends. The next action depends on the application. If `count` equals `ATTEMPTS` after the data-entry loop ends, you might want to force the invalid data to a default value. **Forcing** (Forcing a data item means you override incorrect data by setting it to a specific value.) a data item means you override incorrect data by setting the variable to a specific value. For example, you might decide that if a month value does not fall between 1 and 12, you will force the month to 0 or 99, which indicates to users that no valid value exists. In a different application, you might just choose to end the program. In an interactive, Web-based program, you might choose to have a customer service representative start a chat session with the user to offer help.

Figure 5-20

Limiting User Reprompts

Programs that frustrate users can result in lost revenue for a company. For example, if a company's Web site is difficult to navigate, users might give up and not do business with the organization.

5-6d Validating a Data Type

The data you use within computer programs is varied. It stands to reason that validating data requires a variety of methods. For example, some programming languages allow you to check data items to make sure they are the correct data type. Although this technique varies from language to language, you can often make a statement like the one shown in Figure 5-21. In this program segment, `isNumeric()` represents a call to a module; it is used

to check whether the entered employee `salary` falls within the category of numeric data. You check to ensure that a value is numeric for many reasons—an important one is that only numeric values can be used correctly in arithmetic statements. A module such as `isNumeric()` is most often provided with the language translator you use to write your programs. Such a method operates as a black box; in other words, you can use the method's results without understanding its internal statements.

Figure 5-21

Checking Data for Correct Type

Besides allowing you to check whether a value is numeric, some languages contain methods such as `isChar()`, which checks whether a value is a character data type; `isWhitespace()`, which checks whether a value is a nonprinting character, such as a space or tab; and `isUpper()`, which checks whether a value is a capital letter.

In many languages, you accept all user data as a string of characters, and then use built-in methods to attempt to convert the characters to the correct data type for your application. When the conversion methods succeed, you have useful data. When the conversion methods fail because the user has entered the wrong data type, you can take appropriate action, such as issuing an error message, reprompting the user, or forcing the data to a default value.

5-6e Validating Reasonableness and Consistency of Data

Data items can be the correct type and within range, but still be incorrect. You have experienced this problem yourself if anyone has ever misspelled your name or overbilled you. The data might have been the correct type—for example, alphabetic letters were used in your name—but the name itself was incorrect. Many data items cannot be checked for reasonableness; for example, the names Catherine, Katherine, and Kathryn are equally reasonable, but only one spelling is correct for a particular woman.

However, many data items can be checked for reasonableness. If you make a purchase on May 3, 2013, then the payment cannot possibly be due prior to that date. Perhaps within your organization, you cannot make more than $20.00 per hour if you work in Department 12. If your zip code is 90201, your state of residence cannot be New York. If your store's cash on hand was $3000 when it closed on Tuesday, the amount should not be different when the store opens on Wednesday. If a customer's title is *Ms.*, the customer's gender should be *F*. Each of these examples involves comparing two data items for reasonableness or consistency. You should consider making as many such comparisons as possible when writing your own programs.

Frequently, testing for reasonableness and consistency involves using additional data files. For example, to check that a user has entered a valid county of residence for a state, you might use a file that contains every county name within every state in the United States, and check the user's county against those contained in the file.

Good defensive programs try to foresee all possible inconsistencies and errors. The more accurate your data, the more useful information you will produce as output from your programs.

When you become a professional programmer, you want your programs to work correctly as a source of professional pride. On a more basic level, you do not want to be called in to work at 3:00 a.m. when the overnight run of your program fails because of errors you created.

Two Truths & A Lie

Common Loop Applications

1. An accumulator is a variable that you use to gather or accumulate values.

   T F

2. An accumulator typically is initialized to 0.

   T F

3. An accumulator is typically reset to 0 after it is output.

   T F

5-7 Chapter Review

5-7a Chapter Summary

- A loop contains one set of instructions that operates on multiple, separate sets of data.
- Three steps must occur in every loop: You must initialize a loop control variable, compare the variable to some value that controls whether the loop continues or stops, and alter the variable that controls the loop.
- When you must use loops within loops, you use nested loops. When nesting loops, you must maintain two individual loop control variables and alter each at the appropriate time.
- Common mistakes that programmers make when writing loops include neglecting to initialize the loop control variable, neglecting to alter the loop control variable, using the wrong comparison with the loop control variable, and including statements inside the loop that belong outside the loop.
- Most computer languages support a `for` statement or `for` loop that you can use with definite loops when you know how many times a loop will repeat. The `for` statement uses a loop control variable that it automatically initializes, evaluates, and alters.
- Loops are used in many applications—for example, to accumulate totals in business reports. Loops also are used to ensure that user data entries are valid by continuously reprompting the user.

Chapter Review

5-7b Key Terms

loop control variable (A variable that determines whether a loop will continue.)

definite loop (A loop for which the number of repetitions is a predetermined value.)

counted loop (A counted loop, or counter-controlled loop, is a loop whose repetitions are managed by a counter.)

counter-controlled loop (A counted loop, or counter-controlled loop, is a loop whose repetitions are managed by a counter.)

increment (To change a variable by adding a constant value to it, frequently 1.)

decrement (To change a variable by decreasing it by a constant value, frequently 1.)

counter (Any numeric variable used to count the number of times an event has occurred.)

indefinite loop (A loop for which the number of executions cannot be predicted when the program is written.)

nested loops (A loop structure within another loop structure; nesting loops are loops within loops.)

outer loop (The loop that contains a nested loop.)

inner loop (When loops are nested, the loop that is contained within the other loop.)

stub (A method without statements that is used as a placeholder.)

for statement (A statement that can be used to code definite loops; also called a *for loop*. The statement contains a loop control variable that it automatically initializes, evaluates, and alters.)

for loop (A statement that can be used to code definite loops; also called a *for loop*. The statement contains a loop control variable that it automatically initializes, evaluates, and alters.)

step value (A step value is a number you use to increase a loop control variable on each pass through a loop.)

pretest loop (A loop that tests its loop control variable before each iteration, meaning that the loop body might never execute.)

posttest loop (A loop that tests its loop control variable after each iteration, meaning that the loop body executes at least one time.)

accumulator (A variable used to gather or accumulate values.)

summary reports (A report that lists only totals, without individual detail records.)

defensive programming (A technique in which programmers try to prepare for all possible errors before they occur.)

validate data (Ensuring that data falls within an acceptable range.)

GIGO (Acronym for *garbage in, garbage out*; it means that if input is incorrect, output is worthless.)

Forcing (Forcing a data item means you override incorrect data by setting it to a specific value.)

Chapter 5: Looping: 5-7c Review Questions
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

Chapter Review

5-7c Review Questions

1. The structure that allows you to write one set of instructions that operates on multiple, separate sets of data is the ____ .
    a. sequence
    b. selection
    c. loop
    d. case
2. The loop that frequently appears in a program's mainline logic ____ .
    a. always depends on whether a variable equals 0
    b. works correctly based on the same logic as other loops
    c. is an unstructured loop
    d. is an example of an infinite loop
3. Which of the following is *not* a step that must occur with every correctly working loop?

a. Initialize a loop control variable before the loop starts.

b. Set the loop control value equal to a sentinel during each iteration.

c. Compare the loop control value to a sentinel during each iteration.

d. Alter the loop control variable during each iteration.

4. The statements executed within a loop are known collectively as the ____ .

a. loop body

b. loop controls

c. sequences

d. sentinels

5. A counter keeps track of ____ .

a. the number of times an event has occurred

b. the number of machine cycles required by a segment of a program

c. the number of loop structures within a program

d. the number of times software has been revised

6. Adding 1 to a variable is also called ____ it.

a. digesting

b. resetting

c. decrementing

d. incrementing

7. Which of the following is a definite loop?

a. a loop that executes as long as a user continues to enter valid data

b. a loop that executes 1000 times

c. both of the above

d. none of the above

8. Which of the following is an indefinite loop?

a. a loop that executes exactly 10 times

b. a loop that follows a prompt that asks a user how many repetitions to make and uses the value to control the loop

c. both of the above

d. none of the above

9. When you decrement a variable, you ____ .

a. set it to 0

b. reduce it by one-tenth

c. subtract a value from it

d. remove it from a program

10. When two loops are nested, the loop that is contained by the other is the ____ loop.

a. captive

b. unstructured

c. inner

d. outer

11. When loops are nested, ____ .

a. they typically share a loop control variable

b. one must end before the other begins

c. both must be the same type—definite or indefinite

d. none of the above

12. Most programmers use a `for` loop ____ .

a. for every loop they write

b. when a loop will not repeat

c. when a loop must repeat many times

d. when they know the exact number of times a loop will repeat

13. A report that lists only totals, with no details about individual records, is a(n) ____ report.

    a. accumulator

    b. final

    c. summary

    d. detailless

14. Typically, the value added to a counter variable is ____ .

    a. 0

    b. 1

    c. 10

    d. different in each iteration

15. Typically, the value added to an accumulator variable is ____ .

    a. 0

    b. 1

    c. the same for each iteration

    d. different in each iteration

16. After an accumulator or counter variable is displayed at the end of a program, it is best to ____ .

    a. delete the variable from the program

    b. reset the variable to 0

    c. subtract 1 from the variable

    d. none of the above

17. When you ____ , you make sure data items are the correct type and fall within the correct range.

    a. validate data

    b. employ offensive programming

c. use object orientation

d. count loop iterations

18. Overriding a user's entered value by setting it to a predetermined value is known as ____ .

    a. forcing

    b. accumulating

    c. validating

    d. pushing

19. To ensure that a user's entry is the correct data type, frequently you ____ .

    a. prompt the user to verify that the type is correct

    b. use a method built into the programming language

    c. include a statement at the beginning of the program that lists the data types allowed

    d. all of the above

20. A variable might hold an incorrect value even when it is ____ .

    a. the correct data type

    b. within a required range

    c. a constant coded by the programmer

    d. all of the above

Chapter Review

5-7d Exercises

1. What is output by each of the pseudocode segments in Figure 5-22?

Figure 5-22

Pseudocode Segments for Exercise 1

a.
```
a = 1
b = 2
c = 5
while a < c
   a = a + 1
   b = b + c
endwhile
output a, b, c
```

b.
```
d = 4
e = 6
f = 7
while d > f
   d = d + 1
   e = e - 1
endwhile
output d, e, f
```

c.
```
g = 4
h = 6
while g < h
   g = g + 1
endwhile
output g, h
```

d.
```
j = 2
k = 5
n = 9
while j < k
   m = 6
   while m < n
      output "Goodbye"
      m = m + 1
   endwhile
   j = j + 1
endwhile
```

e.
```
j = 2
k = 5
m = 6
n = 9
while j < k
   while m < n
      output "Hello"
      m = m + 1
   endwhile
   j = j + 1
endwhile
```

f.
```
p = 2
q = 4
while p < q
   output "Adios"
   r = 1
   while r < q
      output "Adios"
      r = r + 1
   endwhile
   p = p + 1
endwhile
```

2. Design the logic for a program that outputs every number from 1 through 20.

3. Design the logic for a program that outputs every number from 1 through 20 along with its value doubled and tripled.

4. Design the logic for a program that outputs every even number from 2 through 100.

5. Design the logic for a program that outputs numbers in reverse order from 25 down to 0.

6. Design the logic for a program that allows a user to enter a number. Display the sum of every number from 1 through the entered number.

7. a. Design an application for the Homestead Furniture Store that gets sales transaction data, including an account number, customer name, and purchase price. Output the account number and name, then output the customer's payment each month for the next 12 months. Assume that there is no finance charge, that the customer makes no new purchases, and that the customer pays off the balance with equal monthly payments.

b. Modify the Homestead Furniture Store application so it executes continuously for any number of customers until a sentinel value is supplied for the account number.

8. a. Design an application for Domicile Designs that gets sales transaction data, including an account number, customer name, and purchase price. The store charges 1.25 percent interest on the balance due each month. Output the account number and name, then output the customer's projected balance each month for the next 12 months. Assume that when the balance reaches $25 or less, the customer can pay off the account. At the beginning of every month, 1.25 percent interest is added to the balance, and then the customer makes a payment equal to 7 percent of the current balance. Assume that the customer makes no new purchases.

   b. Modify the Domicile Designs application so it executes continuously for any number of customers until a sentinel value is supplied for the account number.

9. Yabe Online Auctions requires its sellers to post items for sale for a six-week period during which the price of any unsold item drops 12 percent each week. For example, an item that costs $10.00 during the first week costs 12 percent less, or $8.80, during the second week. During the third week, the same item is 12 percent less than $8.80, or $7.74. Design an application that allows a user to input prices until an appropriate sentinel value is entered. Program output is the price of each item during each week, one through six.

10. Mr. Roper owns 20 apartment buildings. Each building contains 15 units that he rents for $800 per month each. Design the application that would output 12 payment coupons for each of the 15 apartments in each of the 20 buildings. Each coupon should contain the building number (1 through 20), the apartment number (1 through 15), the month (1 through 12), and the amount of rent due.

11. Design a retirement planning calculator for Skulling Financial Services. Allow a user to enter a number of working years remaining in the user's career and the annual amount of money the user can save. Assume that the user earns three percent simple interest on savings annually. Program output is a schedule that lists each year number in retirement starting with year 0 and the user's savings at the start of that year. Assume that the user spends $50,000 per year in retirement and then earns three percent interest on the remaining balance. End the list after 40 years, or when the user's balance is 0 or less, whichever comes first.

12. Ellison Private Elementary School has three classrooms in each of nine grades, kindergarten (grade 0) through grade 8, and allows parents to pay tuition over the nine-month school year. Design the application that outputs nine tuition payment coupons for each of the 27 classrooms. Each coupon should contain the grade number (0 through 8), the classroom number (1 through 3), the month (1 through 9), and the amount of tuition due. Tuition for kindergarten is $80 per month. Tuition for the other grades is $60 per month times the grade level.

13. a. Design a program for the Hollywood Movie Rating Guide, which can be installed in a kiosk in theaters. Each theater patron enters a value from 0 to 4 indicating the number of stars that the patron awards to the guide's featured movie of the week. If a user enters a star value that does not fall in the correct range, reprompt the user continuously until a correct value is entered. The program executes continuously until the theater manager enters a negative number to quit. At the end of the program, display the average star rating for the movie.

    b. Modify the movie-rating program so that a user gets three tries to enter a valid rating. After three incorrect entries, the program issues an appropriate message and continues with a new user.

14. The Café Noir Coffee Shop wants some market research on its customers. When a customer places an order, a clerk asks for the customer's zip code and age. The clerk enters that data as well as the number of items the customer orders. The program operates continuously until the clerk enters a 0 for zip code at the end of the day. When the clerk enters an invalid zip code (more than 5 digits) or an invalid age (defined as less than 10 or more than 110), the program reprompts the clerk continuously. When the clerk enters fewer than 1 or more than 12 items, the program reprompts the clerk two more times. If the clerk enters a high value on the third attempt, the program accepts the high value, but if the clerk enters a value of less than 1 on the third attempt, an error message is displayed and the order is not counted. At the end of the program, display a count of the number of items ordered by customers from the same zip code as the coffee shop (54984), and a count from other zip codes. Also display the average customer age as well as counts of the number of items ordered by customers under 30 and by customers 30 and older.

Find the Bugs

15. Your downloadable files for Chapter 5 include DEBUG05-01.txt, DEBUG05-02.txt, and DEBUG05-03.txt. Each file starts with some comments that describe the problem. Comments are lines that begin with two slashes (//). Following the comments, each file contains pseudocode that has one or more bugs you must find and correct.

Game Zone

16. In Chapter 2, you learned that in many programming languages you can generate a random number between 1 and a limiting value named `LIMIT` by using a statement similar to `randomNumber = random(LIMIT)`. In Chapter 4, you created the logic for a guessing game in which the application generates a random number and the player tries to guess it. Now, create the guessing game itself. After each guess, display a message indicating whether the player's guess was correct, too high, or too low. When the player eventually guesses the correct number, display a count of the number of guesses that were required.

17. Create the logic for a game that simulates rolling two dice by generating two numbers between 1 and 6 inclusive. The player chooses a number between 2 and 12 (the lowest and highest totals possible for two dice). The player then "rolls" two dice up to three times. If the number chosen by the user comes up, the user wins and the game ends. If the number does not come up within three rolls, the computer wins.

18. Create the logic for the dice game Pig, in which a player can compete with the computer. The object of the game is to be the first to score 100 points. The user and computer take turns "rolling" a pair of dice following these rules:

    - On a turn, each player rolls two dice. If no 1 appears, the dice values are added to a running total for the turn, and the player can choose whether to roll again or pass the turn to the other player. When a player passes, the accumulated turn total is added to the player's game total.
    - If a 1 appears on one of the dice, the player's turn total becomes 0; in other words, nothing more is added to the player's game total for that turn, and it becomes the other player's turn.
    - If a 1 appears on both of the dice, not only is the player's turn over, but

the player's entire accumulated total is reset to 0.

- When the computer does not roll a 1 and can choose whether to roll again, generate a random value from 1 to 2. The computer will then decide to continue when the value is 1 and decide to quit and pass the turn to the player when the value is not 1.

Up for Discussion

19. Suppose you wrote a program that you suspect is in an infinite loop because it keeps running for several minutes with no output and without ending. What would you add to your program to help you discover the origin of the problem?

20. Suppose you know that every employee in your organization has a seven-digit ID number used for logging on to the computer system. A loop would be useful to guess every combination of seven digits in an ID. Are there any circumstances in which you should try to guess another employee's ID number?

21. If every employee in an organization had a seven-digit ID number, guessing all the possible combinations would be a relatively easy programming task. How could you alter the format of employee IDs to make them more difficult to guess?