Programming Logic and Design
Ninth Edition

Chapter 2
Elements of High-Quality Programs

Objectives

In this chapter, you will learn about:

- Declaring and using variables and constants
- Performing arithmetic operations
- The advantages of modularization
- Modularizing a program
- Hierarchy charts
- Features of good program design

Declaring and Using Variables and Constants

- **Understanding Data Types**
  - Data type describes:
    - What values can be held by the item
    - How the item is stored in memory
    - What operations can be performed on the item
  - All programming languages support these data types:
    - **Numeric** consists of numbers that can be used in math
    - **String** is anything not used in math

Understanding Unnamed, Literal Constants

- **There are two types of constants**
  - **Numeric constant** (or **literal numeric constant**)
    - Contains numbers only
    - Number does not change
  - **String constant** (or **literal string constant**)
    - Also known as **Alphanumeric values**
    - Can contain both alphabetic characters and numbers
    - Strings are enclosed in quotation marks

Working with Variables

- Variable are named memory locations
- Contents can vary or differ over time
- **Declaration** is a statement that provides a variable's:
  - Data type
  - Identifier (variable’s name)
  - Optionally, an initial value

Working with Variables (continued)

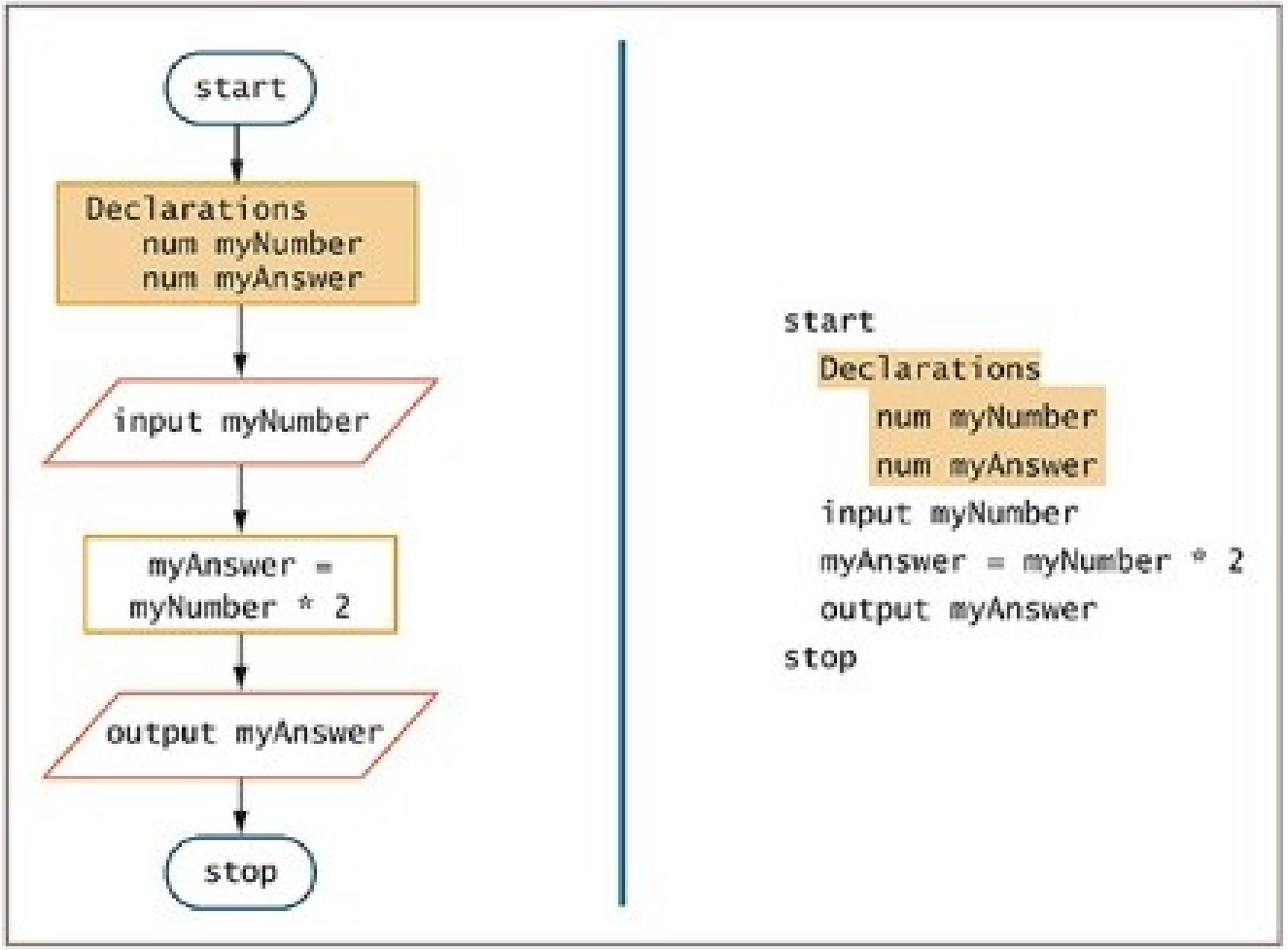

Figure 2-2 Flowchart and pseudocode of number-doubling program with variable declarations

Understanding a Declaration's Data Type

- **Numeric variable**
  - Holds digits
  - Can perform mathematical operations on it
- **String variable**
  - Can hold text
  - Letters of the alphabet
  - Special characters such as punctuation marks
- **Type-safety**
  - Prevents assigning values of an incorrect data type

Understanding a Declaration's Identifier

- An **identifier** is a variable's name
- Programmer chooses reasonable and descriptive names for variables
- Programming languages have rules for creating identifiers
  - Most languages allow letters and digits
  - Some languages allow hyphens
  - Reserved **keywords** are not allowed

Understanding a Declaration's Identifier (continued -1)

- Variable names are case sensitive
- Variable names:
  - Must be one word
  - Must start with a letter
  - Should have some appropriate meaning

Understanding a Declaration's Identifier (continued -2)

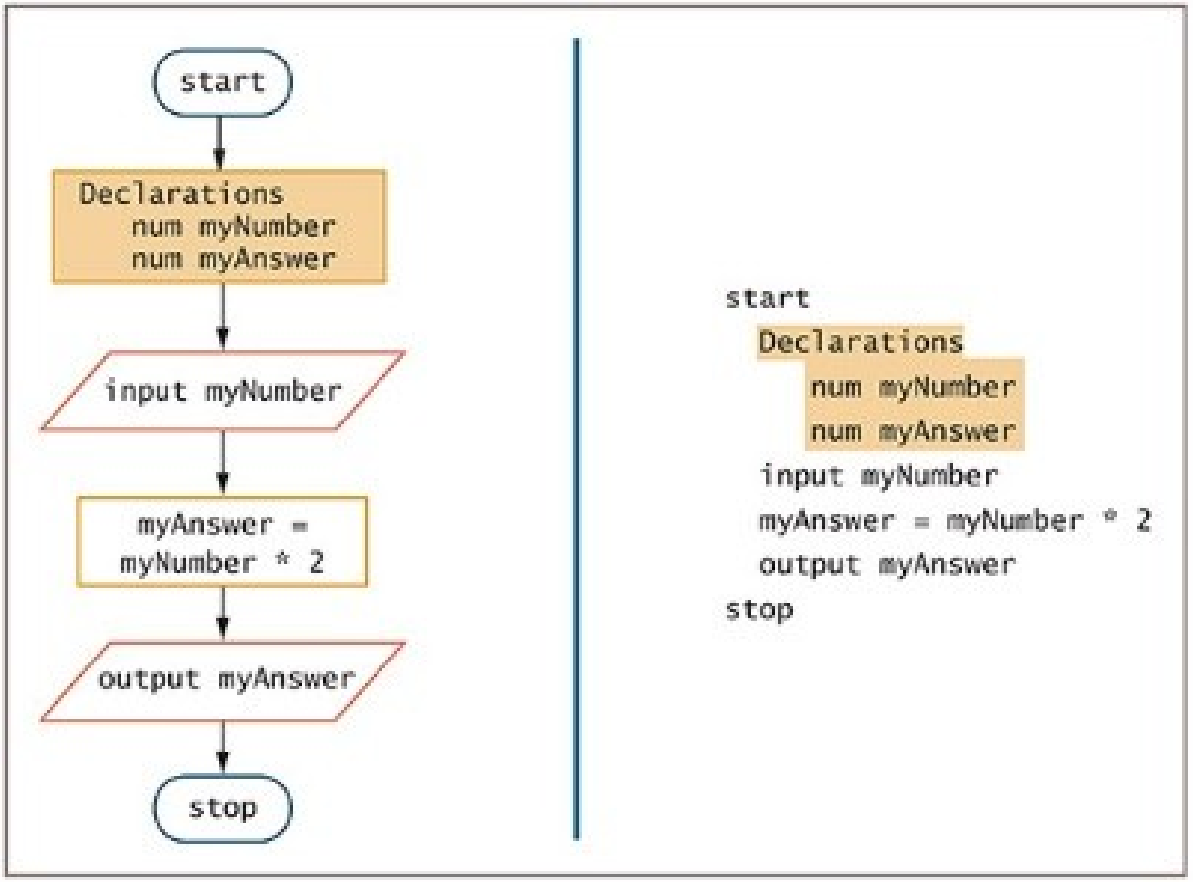

Figure 2-2 Flowchart and pseudocode of number-doubling program with variable declarations

Variable Naming Conventions

- **Camel casing**
  - Variable names have a “hump” in the middle such as `hourlyWage`
- **Pascal casing**
  - Variable names have the first letter in each word in uppercase such as `HourlyWage`
- **Hungarian notation**
  - A form of camel casing in which the data type is part of the name such as `numHourlyWage`

Variable Naming Conventions (continued)

- **Snake casing**
  - Parts of variable names are separated by underscores such as `hourly_wage`
- **Mixed case with underscores**
  - Similar to snake casing, but new words start with a uppercase letter such as `Hourly_Wage`
- **Kebob case**
  - Parts of variable names are separated by dashes such as `hourly-wage`

Assigning Values to Variables

- **Assignment statement**
  - `set myAnswer = myNumber * 2`
- **Assignment operator**
  - Equal sign
  - A **binary operator**, meaning it requires two operands—one on each side
  - Always operates from right to left, which means that it has **right-associativity** or **right-to-left associativity**
  - The result to the left of an assignment operator is called an **lvalue**

Initializing a Variable

- **Initializing the variable -** declare a starting value
  - num yourSalary = 14.55
  - string yourName = “Janita”
- **Garbage** – a variable’s unknown value
- Variables must be declared before they are used in the program

Declaring Named Constants

- **Named constant**
  - Similar to a variable
  - Can be assigned a value only once
  - Assign a useful name to a value that will never be changed during a program's execution
- **Magic number**
  - Unnamed constant
  - Use `taxAmount = price * SALES_TAX_AMOUNT` instead of `taxAmount = price * .06`

Performing Arithmetic Operations

- Standard arithmetic operators:
  - + (plus sign)—addition
  - − (minus sign)—subtraction
  - * (asterisk)—multiplication
  - / (slash)—division

Performing Arithmetic Operations (continued -1)

- **Rules of precedence**
  - Also called the **order of operations**
  - Dictate the order in which operations in the same statement are carried out
  - Expressions within parentheses are evaluated first
  - All the arithmetic operators have **left-to-right associativity**
  - Multiplication and division are evaluated next
    - From left to right
  - Addition and subtraction are evaluated next
    - From left to right

Performing Arithmetic Operations (continued -2)

QUICK REFERENCE 2-2 Precedence and Associativity of Five Common Operators

Operator symbol	Operator name	Precedence (compared to other operators in this table)	Associativity
=	Assignment	Lowest	Right-to-left
+	Addition	Medium	Left-to-right
-	Subtraction	Medium	Left-to-right
*	Multiplication	Highest	Left-to-right
/	Division	Highest	Left-to-right

The Integer Data Type

- Dividing an integer by another integer is a special case
  - Dividing two integers results in an integer, and any fractional part of the result is lost
  - The decimal portion of the result is cut off, or truncated
- A **remainder operator (**called the modulo operator or the modulus operator) contains the remainder of a division operation
  - 24 Mod 10 is 4
  - Because when 24 is divided by 10, 4 is the remainder

Understanding the Advantages of Modularization

- **Modules**
  - Subunit of programming problem
  - Also called **subroutines**, **procedures**, **functions**, or **methods**
  - To **call a module** is to use its name to invoke the module, causing it to execute
- **Modularization**
  - Breaking down a large program into modules
  - Called **functional decomposition**

Modularization Provides Abstraction

- **Abstraction**
  - Paying attention to important properties while ignoring nonessential details
  - Selective ignorance
- Newer high-level programming languages
  - Use English-like vocabulary
  - One broad statement corresponds to dozens of machine instructions
- Modules provide another way to achieve abstraction

Modularization Allows Multiple Programmers to Work on a Problem

- Easier to divide the task among various people
- Rarely does a single programmer write a commercial program
  - Professional software developers can write new programs quickly by dividing large programs into modules
  - Assign each module to an individual programmer or team

Modularization Allows You to Reuse Work

- **Reusability**
  - Feature of modular programs
  - Allows individual modules to be used in a variety of applications
  - Many real-world examples of reusability
- **Reliability**
  - Assures that a module has been tested and proven to function correctly

Modularizing a Program

- **Main program**
  - Basic steps (**mainline logic**) of the program
- Include in a module
  - **Module header**
  - **Module body**
  - **Module return statement**
- Naming a module
  - Similar to naming a variable
  - Module names are followed by a set of parentheses

Modularizing a Program (continued -1)

- When a main program wants to use a module
  - "Calls" the module's name
- Flowchart
  - Symbol used to call a module is a rectangle with a bar across the top
  - Place the name of the module you are calling inside the rectangle
  - Draw each module separately with its own sentinel symbols

Modularizing a Program (continued -2)

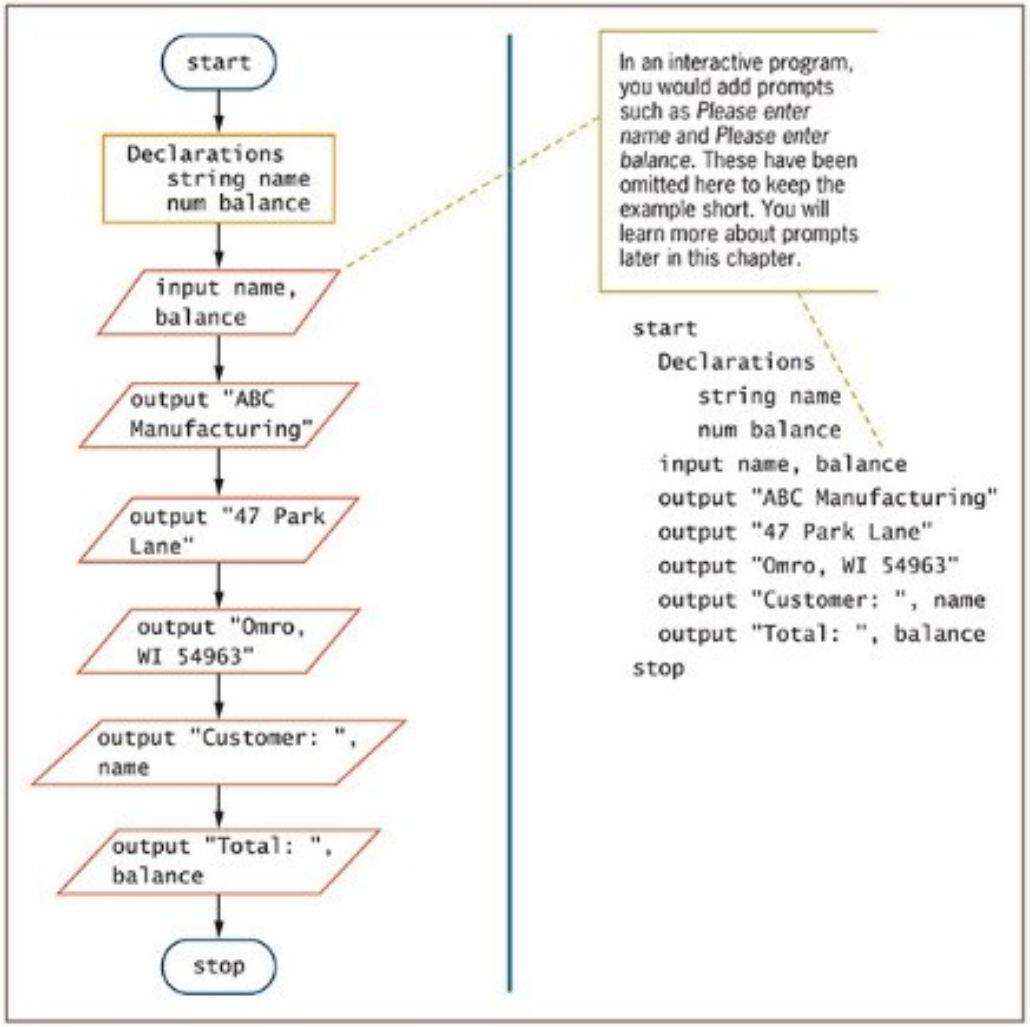

Figure 2-3 Program that produces a bill using only main program

Modularizing a Program (continued -3)

- Statements taken out of a main program and put into a module have been **encapsulated**
- Main program becomes shorter and easier to understand
- Modules are reusable
- When statements contribute to the same job, we get greater **functional cohesion**

Modularizing a Program (continued -4)

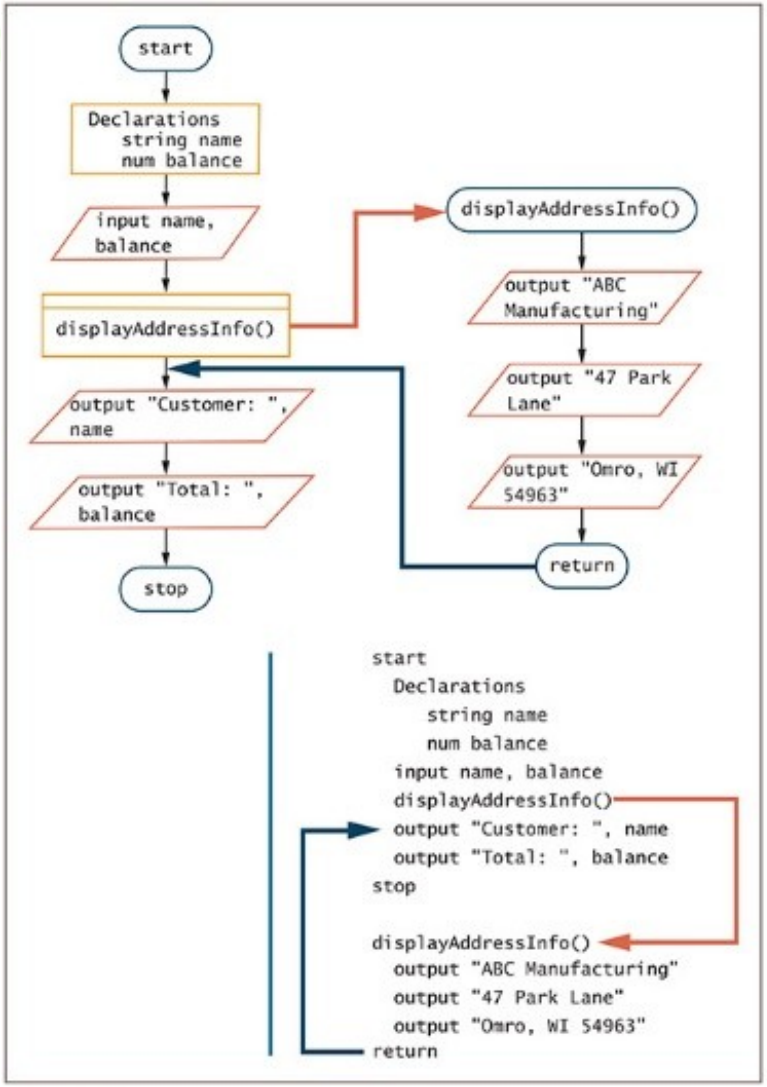

Figure 2-4 Program that produces a bill using main program that calls `displayAddressInfo()` module

Declaring Variables and Constants within Modules

- Place any statements within modules
  - Input, processing, and output statements
  - Variable and constant declarations
- Variables and constants declared in a module are usable only within the module
  - **Visible**
  - **In scope**, also called **local**
- **Portable**
  - Self-contained units that are easily transported

Declaring Variables and Constants within Modules (continued -1)

- **Global** variables and constants
  - Declared at the **program level**
  - Visible to and usable in all the modules called by the program
  - Many programmers avoid global variables to minimize errors

Declaring Variables and Constants within Modules

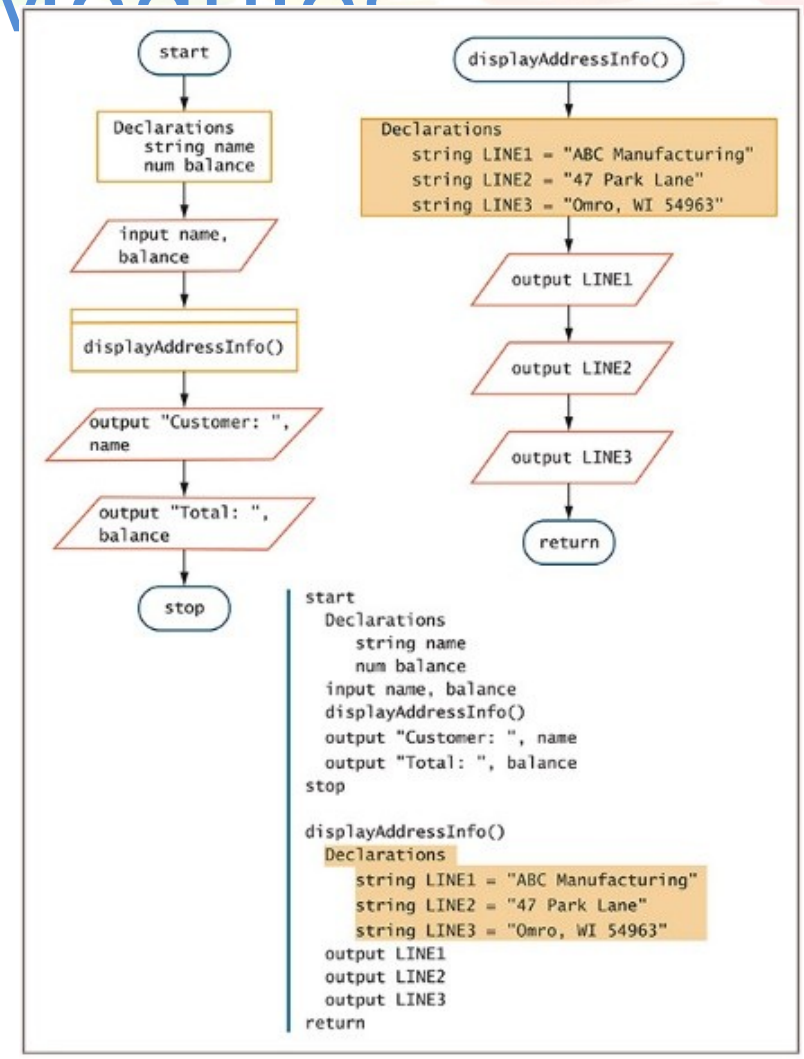

Figure 2-5 The billing program with constants declared within the module

Understanding the Most Common Configuration for Mainline Logic

- Mainline logic of almost every procedural computer program follows a general structure
  - Declarations for global variables and constants
  - **Housekeeping tasks -** steps you must perform at the beginning of a program to get ready for the rest of the program
  - **Detail loop tasks -** do the core work of the program
  - **End-of-job tasks -** steps you take at the end of the program to finish the application

Understanding the Most Common Configuration for Mainline Logic

- A **loop** is a repetition of a series of steps
  - Avoid an **infinite loop** (repeating flow of logic that never ends)
- **Making a decision**
  - Testing a value
  - **Decision symbol:** Diamond shape
- **Dummy value**
  - Data-entry value that the user will never need
  - **Sentinel value**
- **eof** (“end of file”)
  - Marker at the end of a file that automatically acts as a sentinel

Understanding the Most Common Configuration for Mainline Logic

(continued -1)

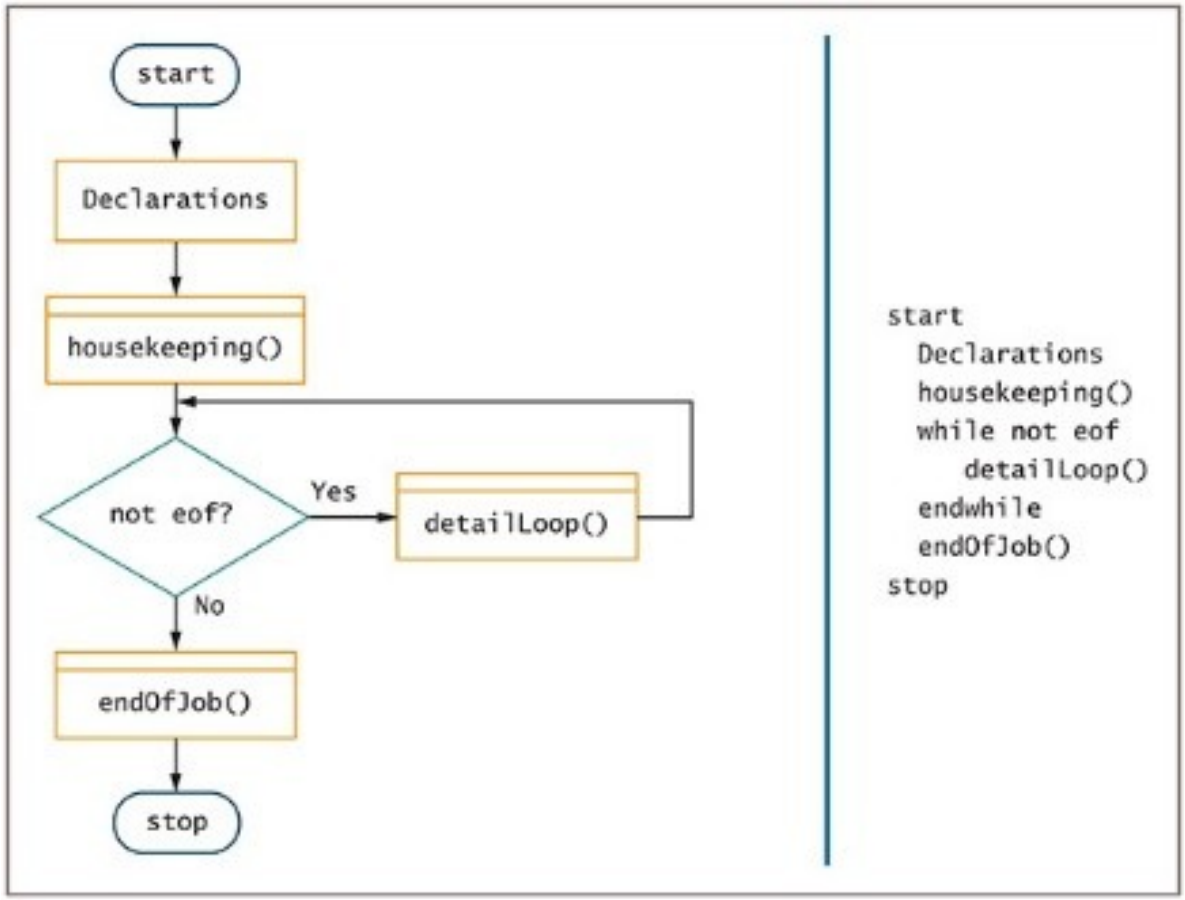

Figure 2-6 Flowchart and pseudocode of mainline logic for a typical procedural program

Understanding the Most Common Configuration for Mainline Logic

(continued -2)

Payroll Report

Name	Gross	Deductions	Net
Andrews	1000.00	250.00	750.00
Brown	1400.00	350.00	1050.00
Carter	1275.00	318.75	956.25
Young	1100.00	275.00	825.00

***End of report

Figure 2-7 Sample payroll report

Understanding the Most Common Configuration for Mainline Logic (continued -3)

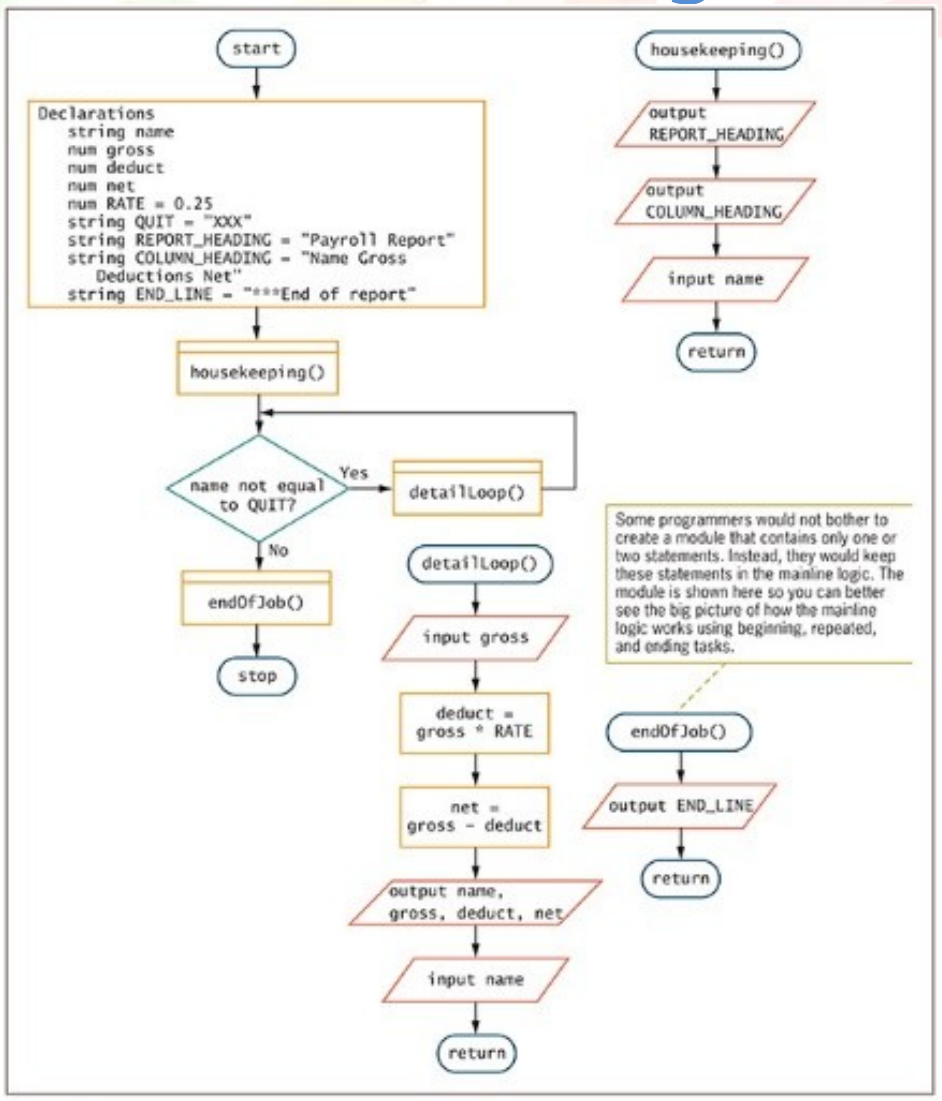

Figure 2-8 Logic for payroll report

Creating Hierarchy Charts

- **Hierarchy chart**
  - Shows the overall picture of how modules are related to one another
  - Tells you which modules exist within a program and which modules call others
  - Specific module may be called from several locations within a program
- Planning tool
  - Develop the overall relationship of program modules before you write them
- Documentation tool

Creating Hierarchy Charts

(continued -1)

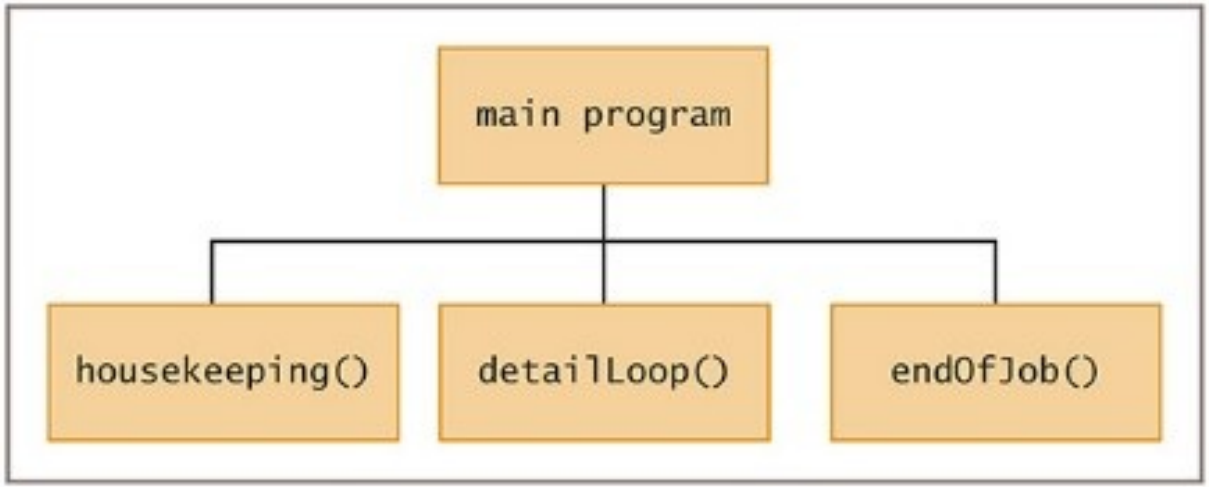

Figure 2-10 Hierarchy chart of payroll report program in Figure 2-8

Creating Hierarchy Charts

(continued -2)

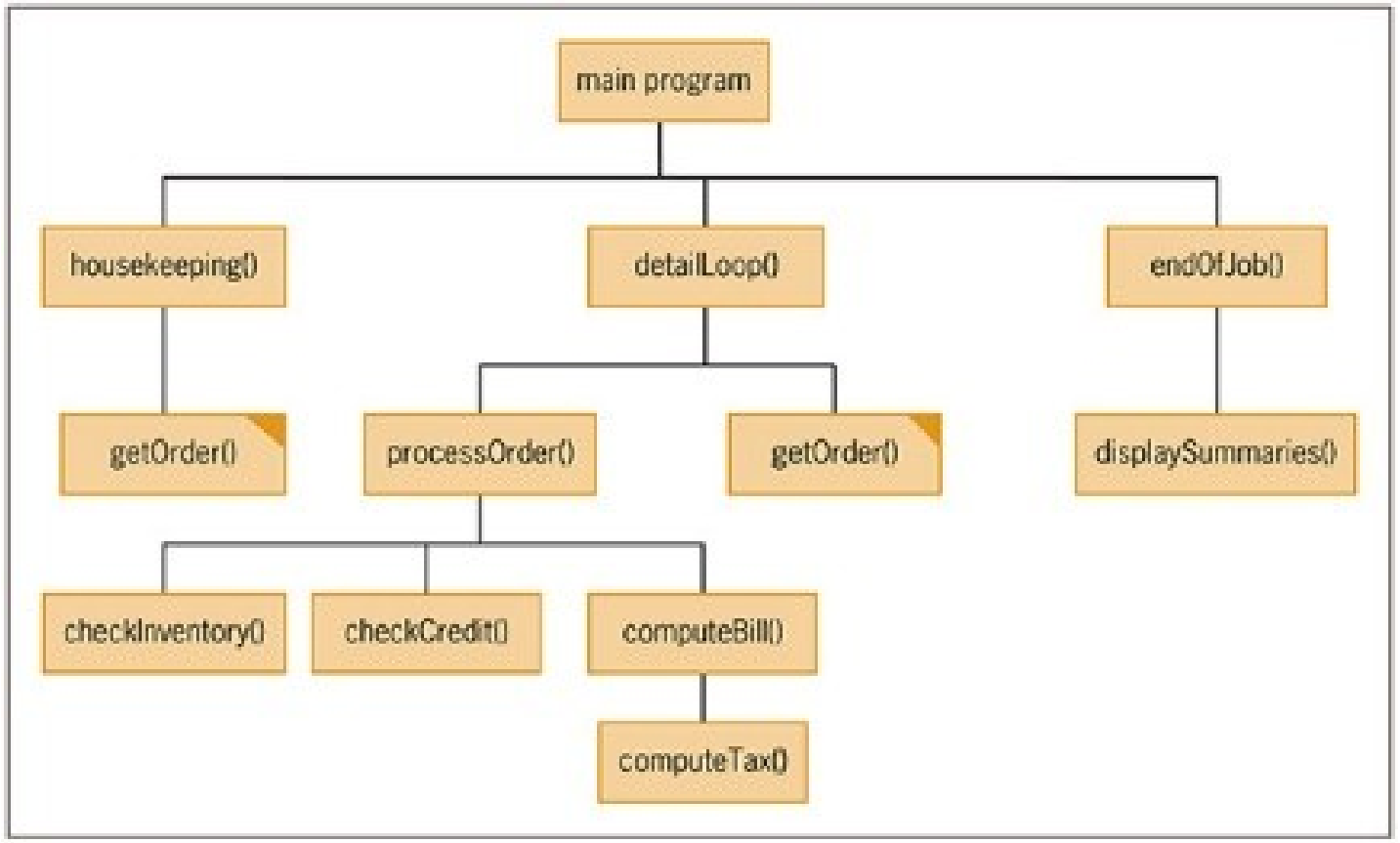

Figure 2-11 Billing program hierarchy chart

Features of Good Program Design

- **Use program comments** where appropriate
- Identifiers should be chosen carefully
- Strive to design clear statements within your programs and modules
- Write clear prompts and echo input
- Continue to maintain good programming habits as you develop your programming skills

Using Program Comments

- **Program comments**
  - Written explanations of programming statements
  - Not part of the program logic
  - Serve as internal documentation for the program
- Syntax used differs among programming languages
- Flowchart
  - Use an **annotation symbol** to hold information that expands on what is stored within another flowchart symbol

Using Program Comments

(continued -1)

Examples of declarations:

num sqFeet // sqFeet is an estimate provided by the seller of the property

num pricePerFoot // pricePerFoot is determined by current market conditions

num lotPremium // lotPremium depends on amenities such as whether lot is waterfront

Using Program Comments

(continued -2)

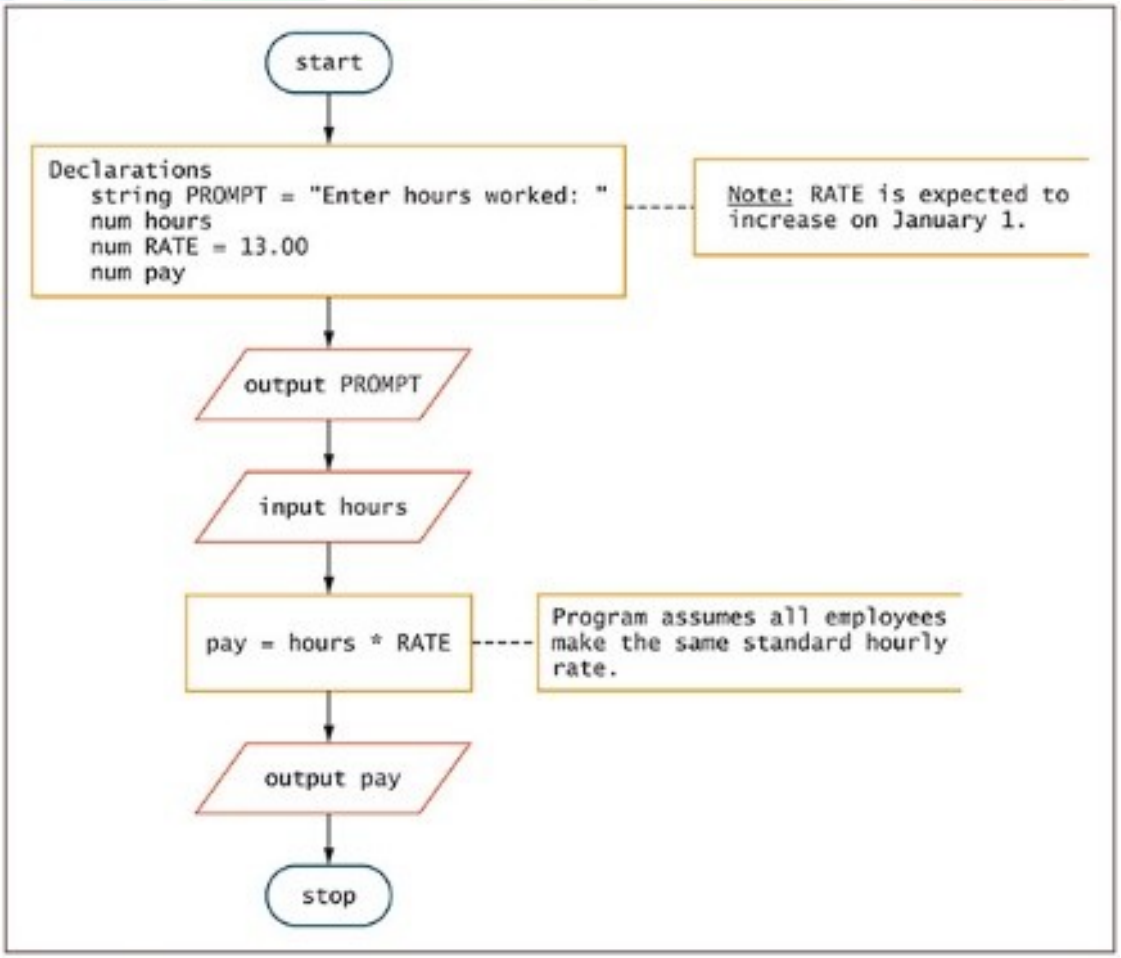

Figure 2-13 Flowchart that includes annotation symbols

Choosing Identifiers

- General guidelines
  - Give a variable or a constant a name that is a noun (because it represents a thing)
  - Give a module an identifier that is a verb (because it performs an action)
  - Use meaningful names
    - **Self-documenting**
  - Use pronounceable names
  - Be judicious in your use of abbreviations
  - Avoid digits in a name

Choosing Identifiers (continued)

- General guidelines (continued)
  - Use the system your language allows to separate words in long, multiword variable names
  - Consider including a form of the verb *to be*
  - Name constants using all uppercase letters separated by underscores (_)
- Programmers create a list of all variables
  - **Data dictionary**

Designing Clear Statements

- Avoid confusing line breaks
- Use temporary variables to clarify long statements

Avoiding Confusing Line Breaks

- Most modern programming languages are free-form
- Make sure your meaning is clear
- Do not combine multiple statements on one line

Using Temporary Variables to Clarify Long Statements

- **Temporary variable**
  - **Work variable**
  - Not used for input or output
  - Working variable that you use during a program's execution
- Consider using a series of temporary variables to hold intermediate results

Using Temporary Variables to Clarify Long Statements (continued)

```
// Using a single statement to compute
commission
   salesCommission = (sqFeet * pricePerFoot +
                              lotPremium) *
commissionRate

// Using multiple statements to compute
commission
   basePropertyPrice = sqFeet * pricePerFoot
   totalSalesPrice = basePropertyPrice +
lotPremium
   salesCommission = totalSalesPrice *
```

Writing Clear Prompts and Echoing Input

- **Prompt**
  - Message displayed on a monitor to ask the user for a response
  - Used both in command-line and GUI interactive programs
- **Echoing input**
  - Repeating input back to a user either in a subsequent prompt or in output

Writing Clear Prompts and Echoing Input (continued -1)

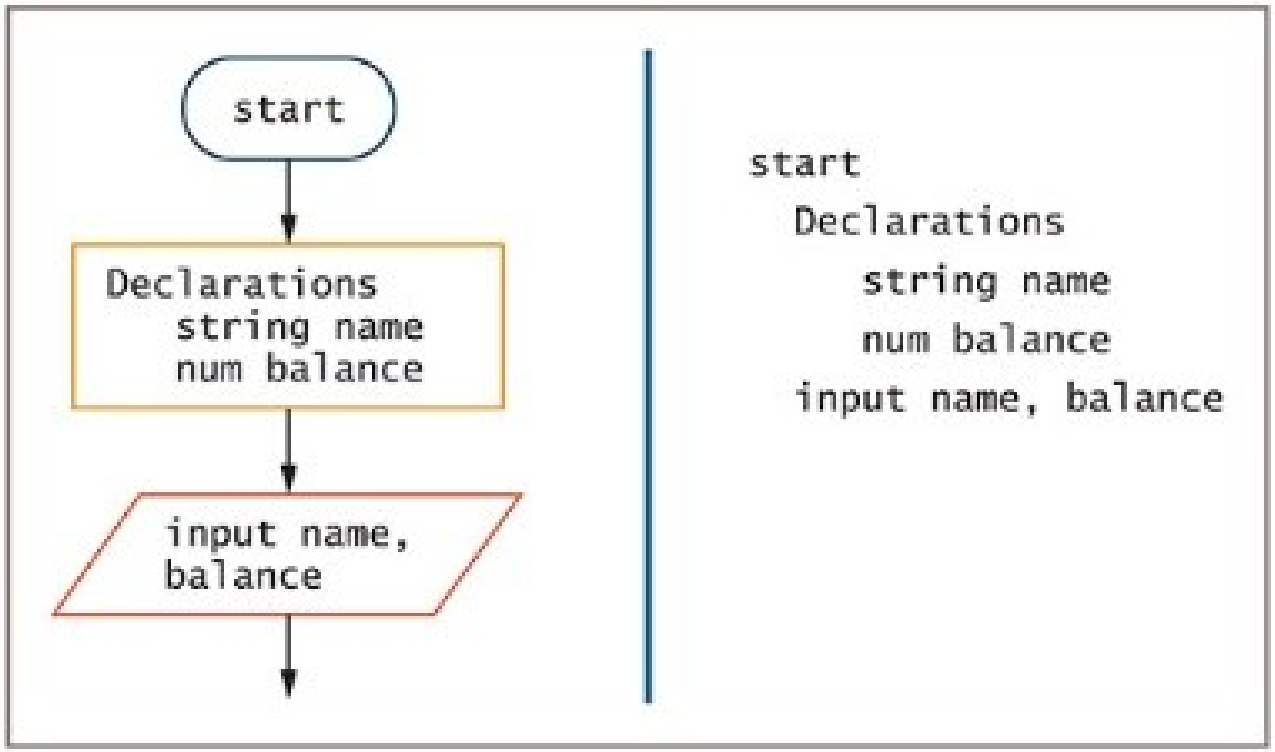

Figure 2-15 Beginning of a program that accepts a name and balance as input

Writing Clear Prompts and Echoing Input (continued -2)

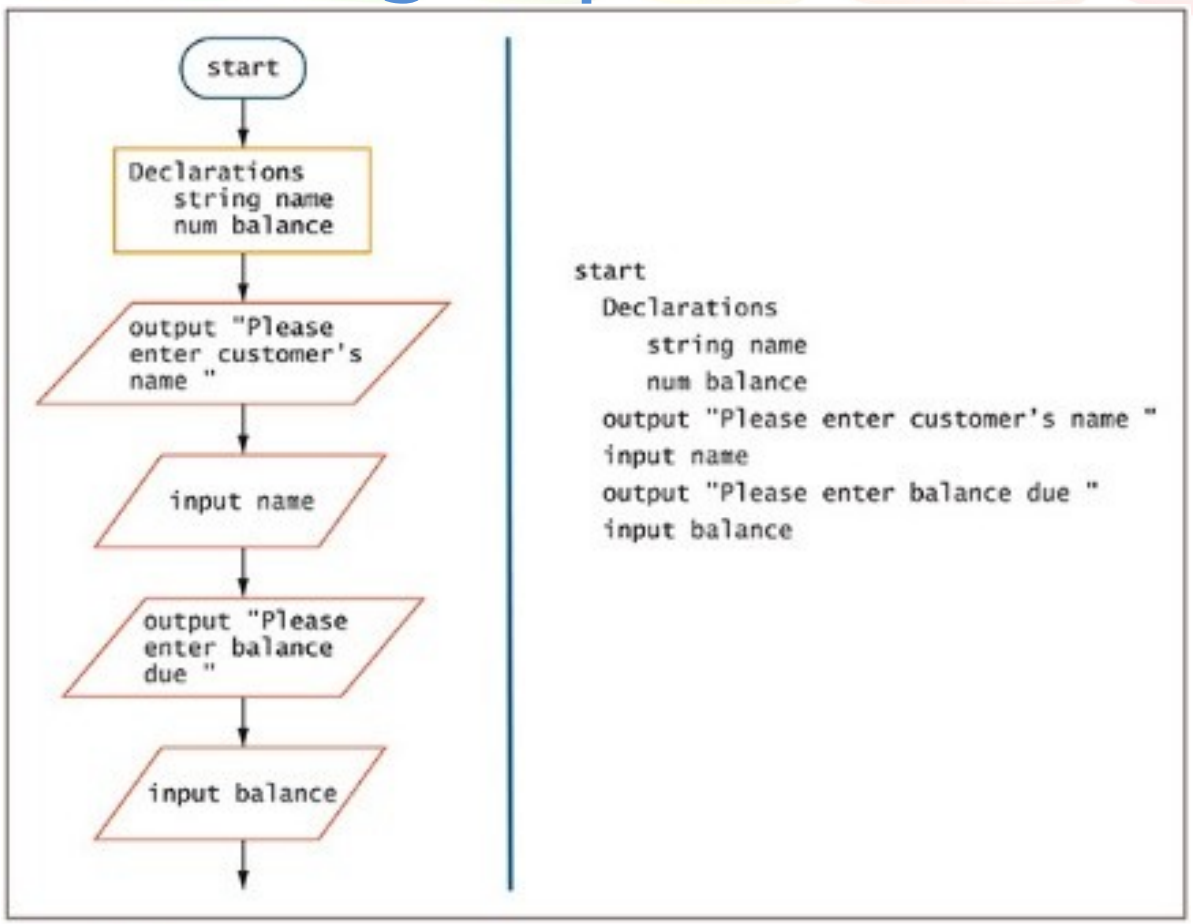

Figure 2-16 Beginning of a program that accepts a name and balance as input and uses a separate prompt for each item

Writing Clear Prompts and Echoing Input (continued -3)

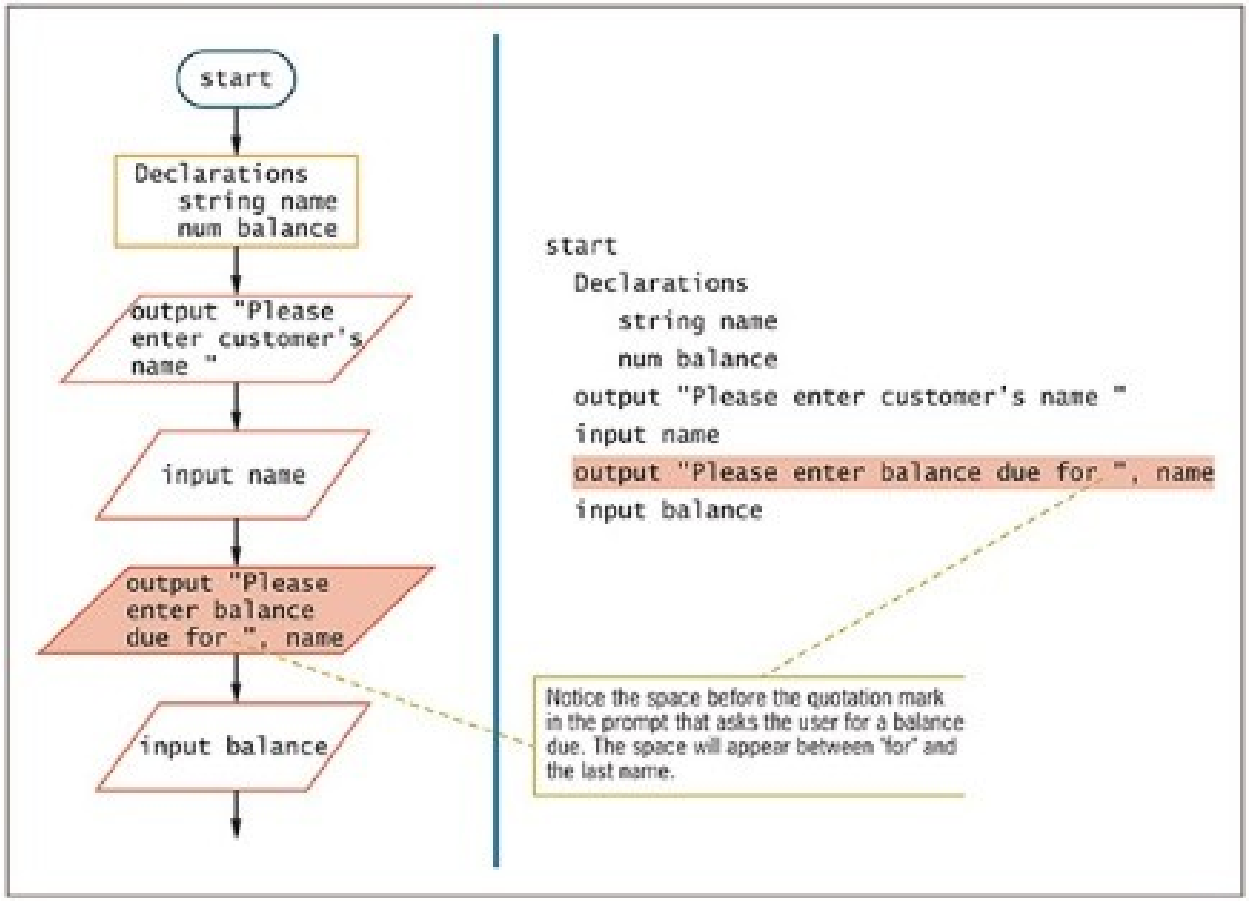

Figure 2-17 Beginning of a program that accepts a customer's name and uses it in the second prompt

Maintaining Good Programming Habits

- Every program you write will be better if you:
  - Plan before you code
  - Maintain the habit of first drawing flowcharts or writing pseudocode
  - Desk-check your program logic on paper
  - Think carefully about the variable and module names you use
  - Design your program statements to be easy to read and use

Summary

- Programs contain literals, variables, and named constants
- Arithmetic follows rules of precedence
- Break down programming problems into modules
  - Include a header, a body, and a `return` statement
- Hierarchy charts show relationship among modules
- As programs become more complicated:
  - Need for good planning and design increases