Programming Logic and Design

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

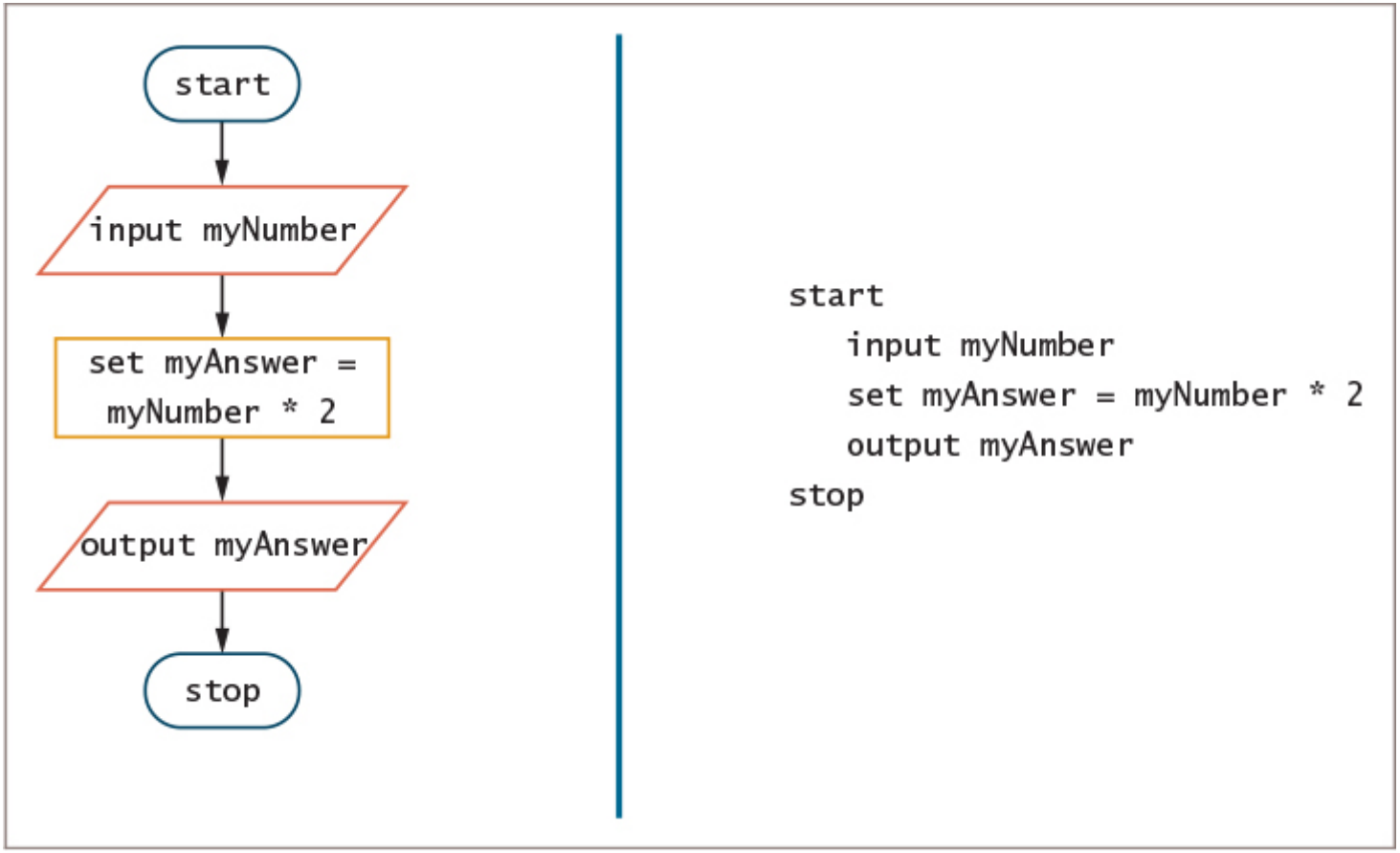
Declaring and Using Variables and Constants

- **Understanding Unnamed, Literal Constants and their Data Types**
  - Data types
    - **Numeric** consists of numbers
    - **String** is anything not used in math
  - Different forms
    - **Integers** and **floating-point (**also called **real)** numbers
    - **Numeric Constant (Literal numeric constant)**
    - **String constants (alphanumeric vakues)**
    - **Unnamed constants**

Working with Variables

- Named memory locations
- Contents can vary or differ over time
- **Declaration**
  - Statement that provides a data type and an identifier for a variable
- **Identifier**
  - Variable's name

Working with Variables (continued)

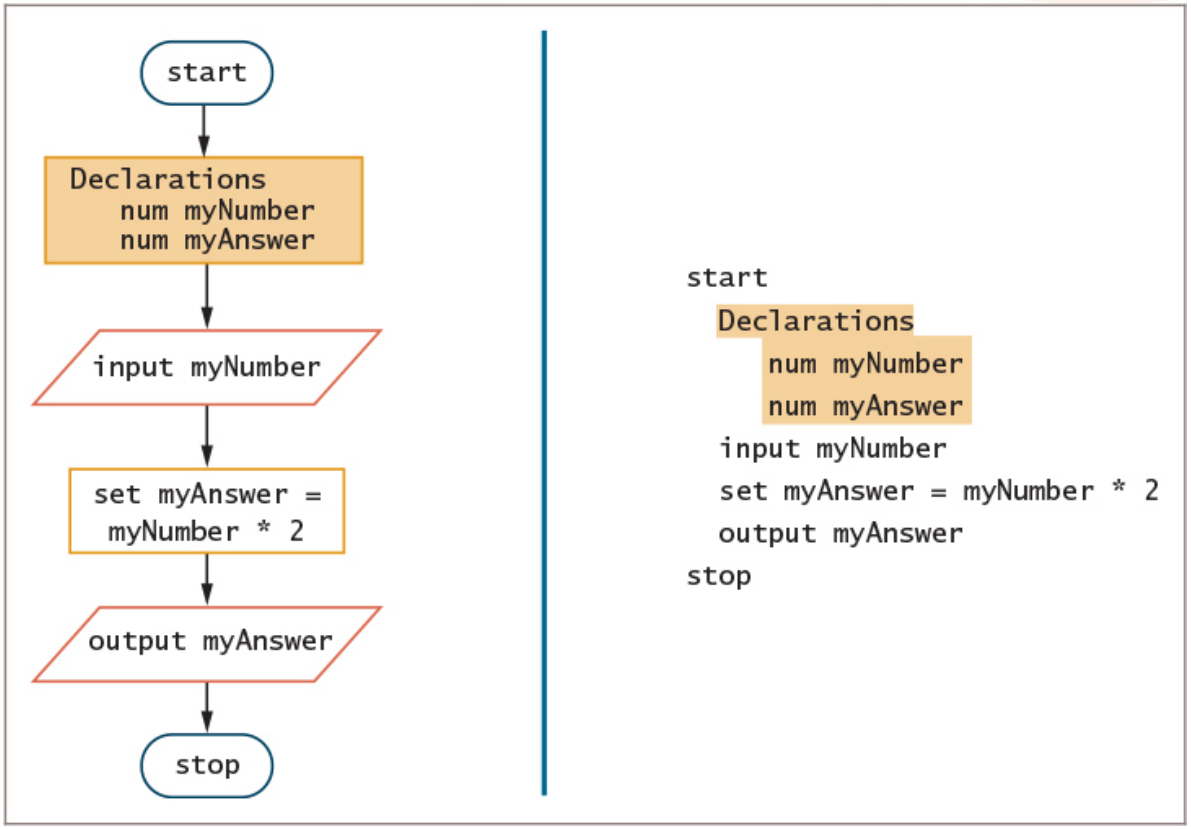


Flowchart and pseudocode for the number-doubling program

Working with Variables (continued)

- A **declaration** is a statement that provides a data type and an identifier for a variable
- An **identifier** is a program component's name
- **Data type**
  - Classification that describes:
    - What values can be held by the item
    - How the item is stored in computer memory
    - What operations can be performed on the data item
- **Initializing the variable -** Declare a starting value
- **Garbage** – a variable's unknown value

Working with Variables (continued)



Flowchart and pseudocode of number-doubling program with variable declarations

Naming Variables

- 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
- Variable names are case sensitive

Naming Variables (continued)

- **Camel casing**
  - Variable names such as `hourlyWage` have a "hump" in the middle
- Be descriptive
  - Must be one word
  - Must start with a letter
  - Should have some appropriate meaning

Naming Variables (continued)

QUICK REFERENCE 2-1 Variable Naming Conventions

Convention for naming variables	Examples	Languages where commonly used
Camel casing is the convention in which the variable starts with a lowercase letter and any subsequent word begins with an uppercase letter. It is sometimes called **lower camel casing** to emphasize the difference from Pascal casing.	`hourlyWage` `lastName`	Java, C#
Pascal casing is a convention in which the first letter of a variable name is uppercase. It is sometimes called **upper camel casing** to distinguish it from lower camel casing.	`HourlyWage` `LastName`	Visual Basic
Hungarian notation is a form of camel casing in which a variable's data type is part of the identifier.	`numHourlyWage` `stringLastName`	C for Windows API programming
Snake casing is a convention in which parts of a variable name are separated by underscores.	`hourly_wage` `last_name`	C, C++, Python, Ruby
Mixed case with underscores is a variable naming convention similar to snake casing, but new words start with an uppercase letter.	`Hourly_Wage` `Last_Name`	Ada
Kebob case is sometimes used as the name for the style that uses dashes to separate parts of a variable name. The name derives from the fact that the words look like pieces of food on a skewer.	`hourly-wage` `last-name`	Lisp (with lowercase letters), COBOL (with uppercase letters)

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**

Understanding the Data Types of Variables

- **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

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)

- **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)

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

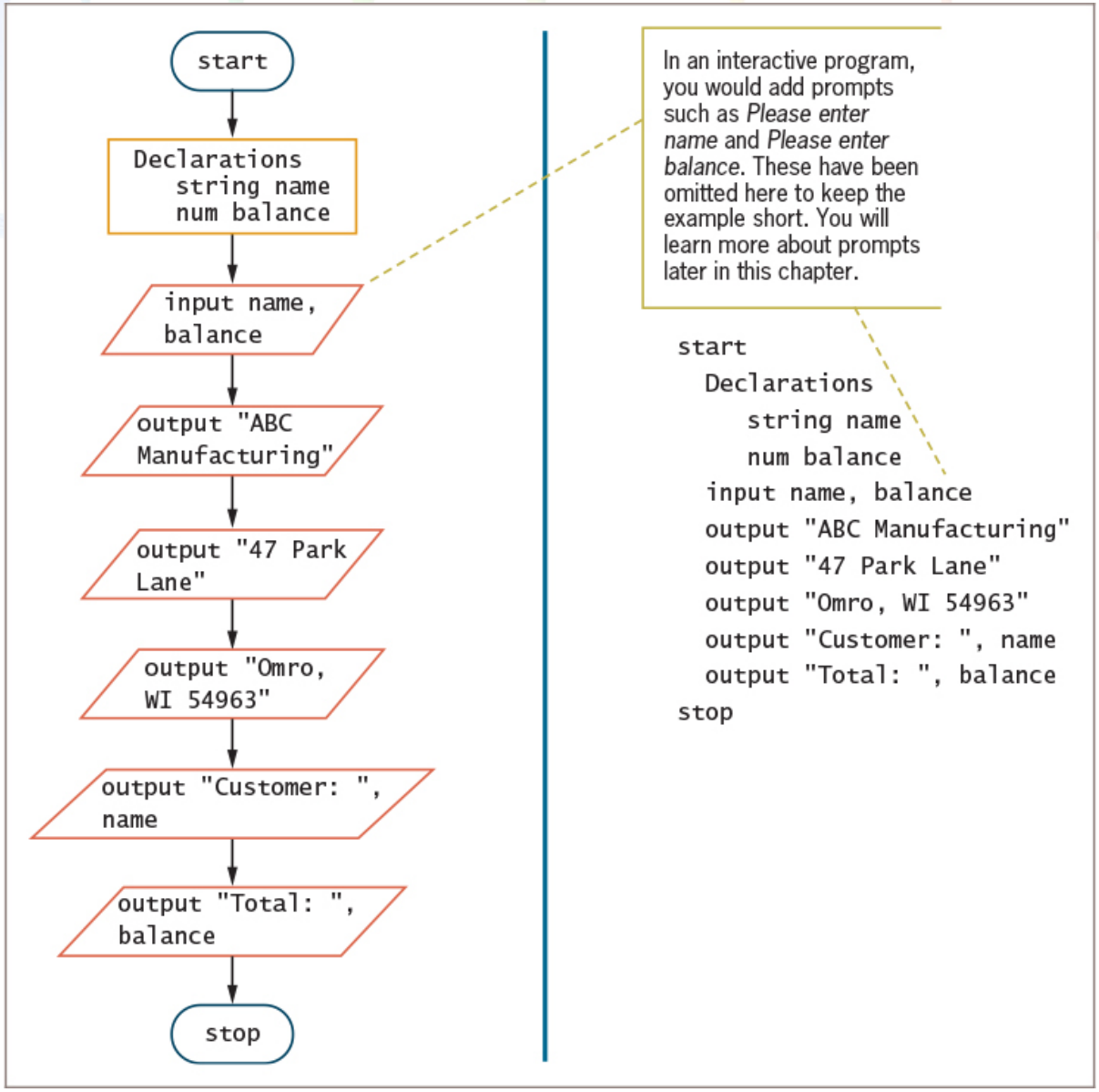
- **Reusability**
  - Feature of modular programs
  - Allows individual modules to be used in a variety of applications
  - Many real-world examples of reusability
- **Reliability**
  - Assures that a module has been tested and proven to function correctly

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)

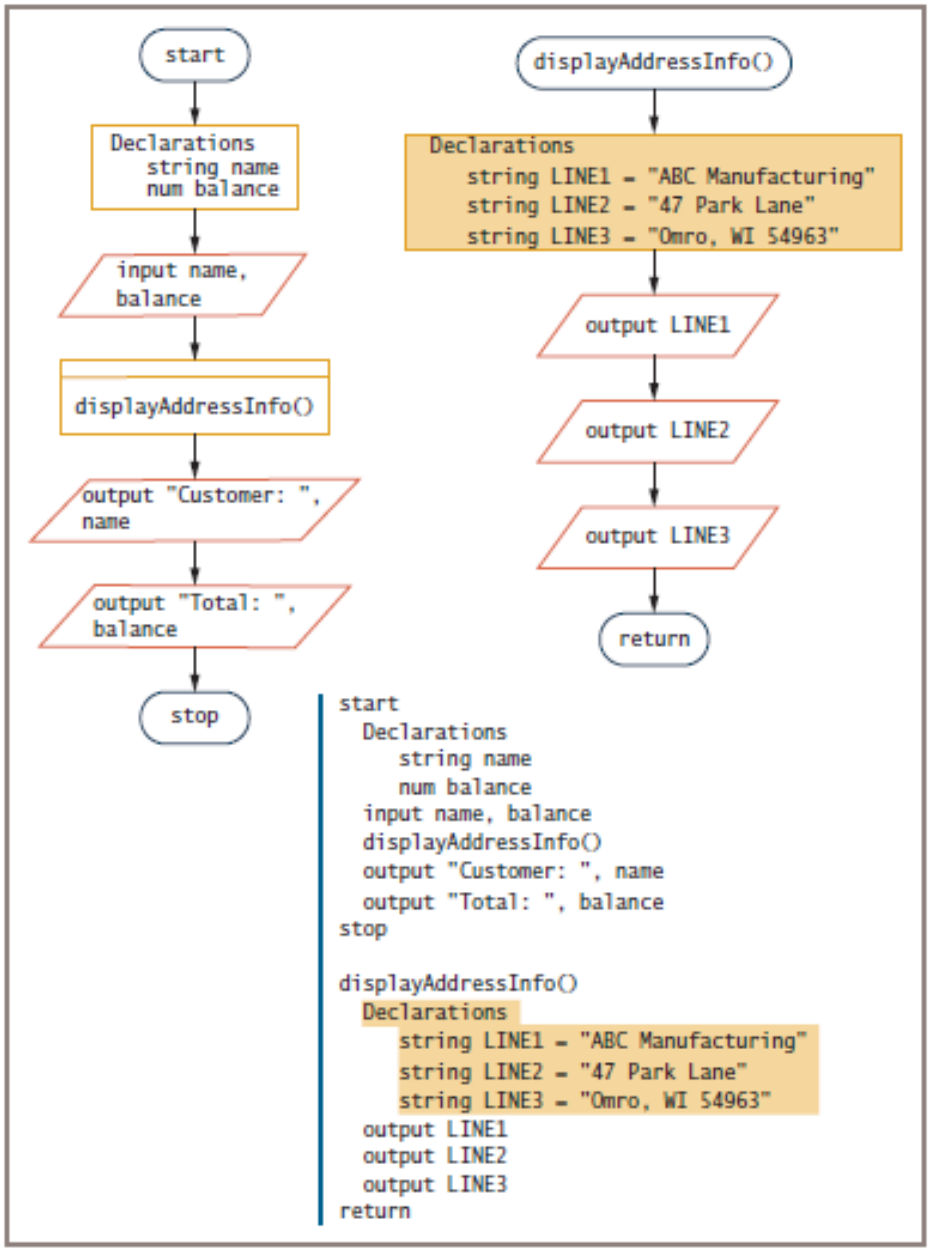
- When a main program wants to use a module
  - "Calls" the module's name
- Flowchart
  - Symbol used to call a module is a rectangle with a bar across the top
  - Place the name of the module you are calling inside the rectangle
  - Draw each module separately with its own sentinel symbols



Program that produces a bill using only main program

Modularizing a Program (continued)

- 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**



The billing program with constants declared within the 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

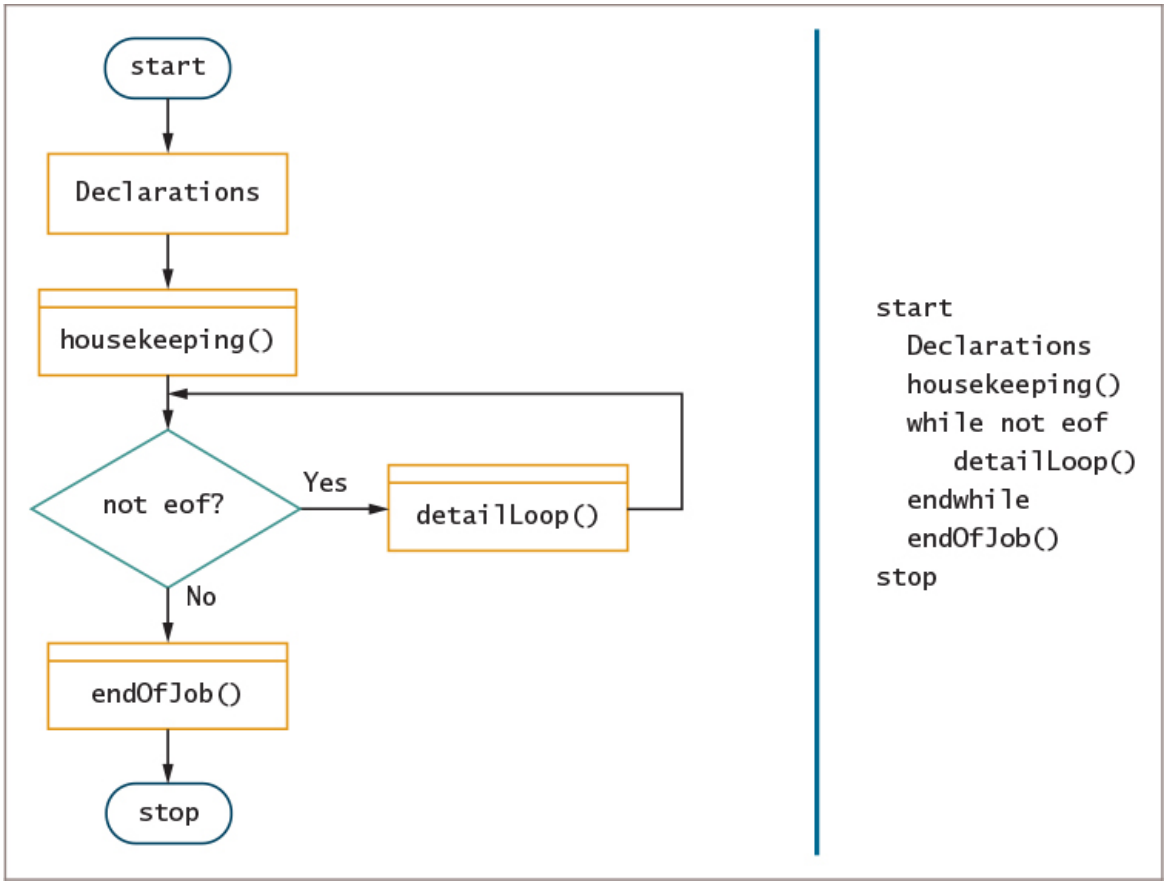
Declaring Variables and Constants within Modules (continued)

- **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

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 (continued)

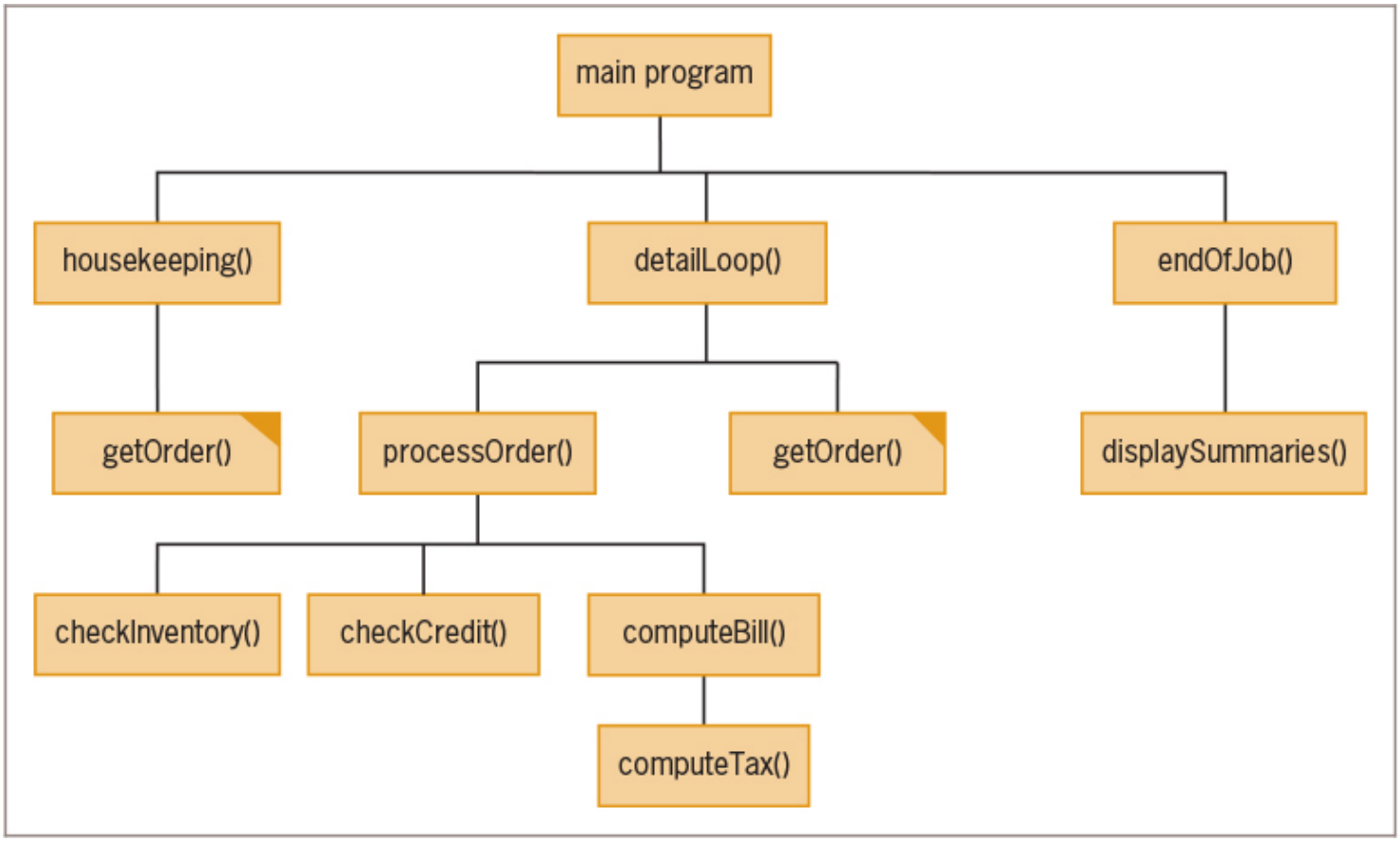


Flowchart and pseudocode of mainline logic for a typical procedural program

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)



Billing program hierarchy chart

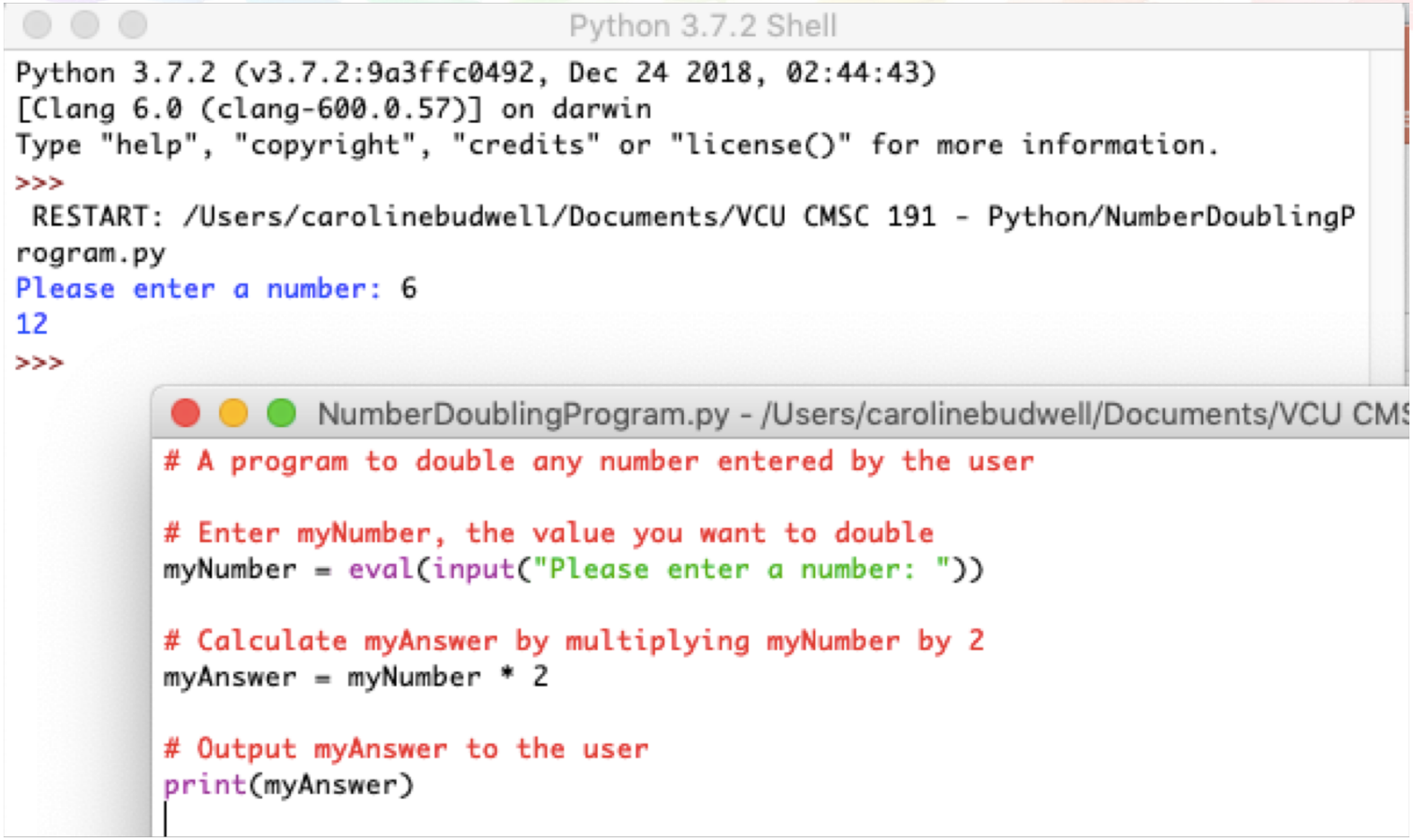
Features of Good Program Design

- **Provide program comments where appropriate**
- Choose good identifiers for program elements
- Design clear statements within your program and modules
- Write clear prompts to the user and echo their input back to them
- Continue to maintain good programming habits

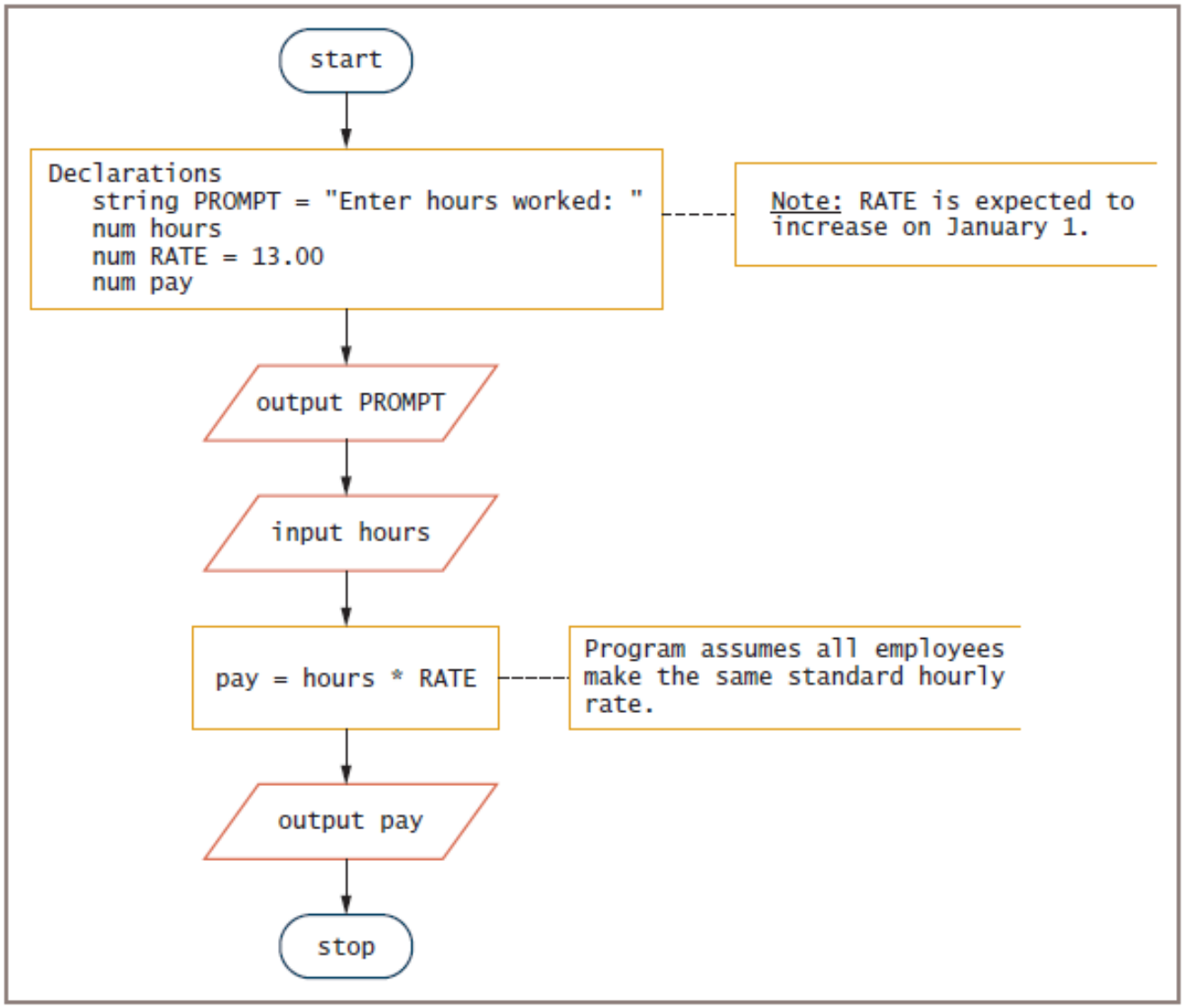
Using Program Comments

- **Program comments**
  - Written explanations of programming statements
  - Not part of the program logic
  - Serve as documentation for readers of 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)



Comments that describe what the code is doing. Note that they ignored when the program is run.



Flowchart that includes annotation symbols

Features of Good Program Design

- Providing program comments where appropriate
- **Choosing good identifiers for program element**s
- Design clear statements within your program and modules
- Write clear prompts to the user and echo their input back to them
- Continue to maintain good programming habits

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**

Features of Good Program Design

- Providing program comments where appropriate
- Choosing good identifiers for program elements
- **Design clear statements within your program and modules**
- Write clear prompts to the user and echo their input back to them
- Continue to maintain good programming habits

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
salespersonCommission = (sqFeet * pricePerFoot + lotPremium) * commissionRate

// Using multiple statements to compute commission
basePropertyPrice = sqFeet * pricePerFoot
totalSalePrice = basePropertyPrice + lotPremium
salespersonCommission = totalSalePrice * commissionRate
```

Figure 2-14 Two ways of achieving the same `salespersonCommission` result

Features of Good Program Design

- Providing program comments where appropriate
- Choosing good identifiers for program elements
- Design clear statements within your program and modules
- **Write clear prompts to the user and echo their input back to them**
- Continue to maintain good programming habits

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)

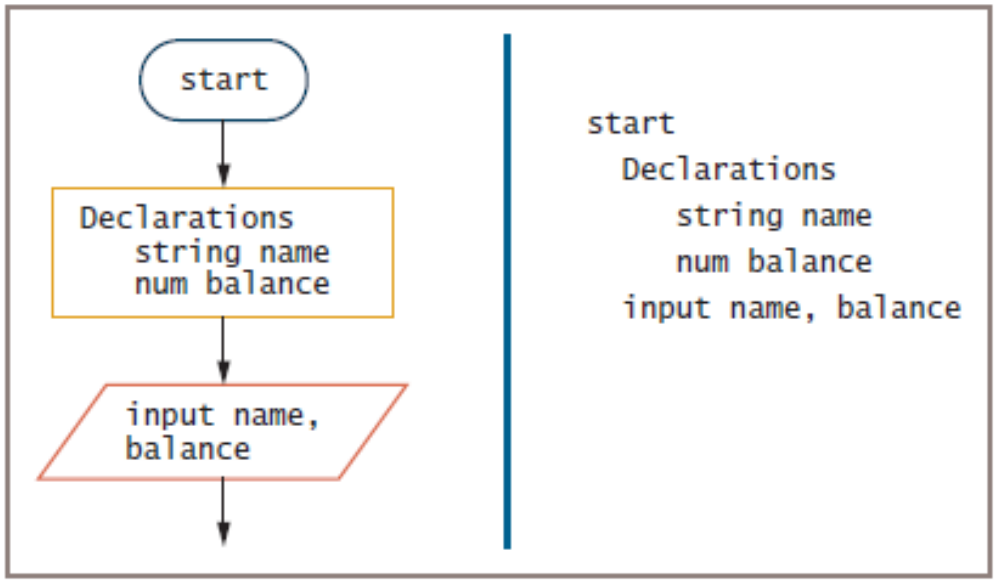


Figure 2-15 Beginning of a program that accepts a name and balance as input

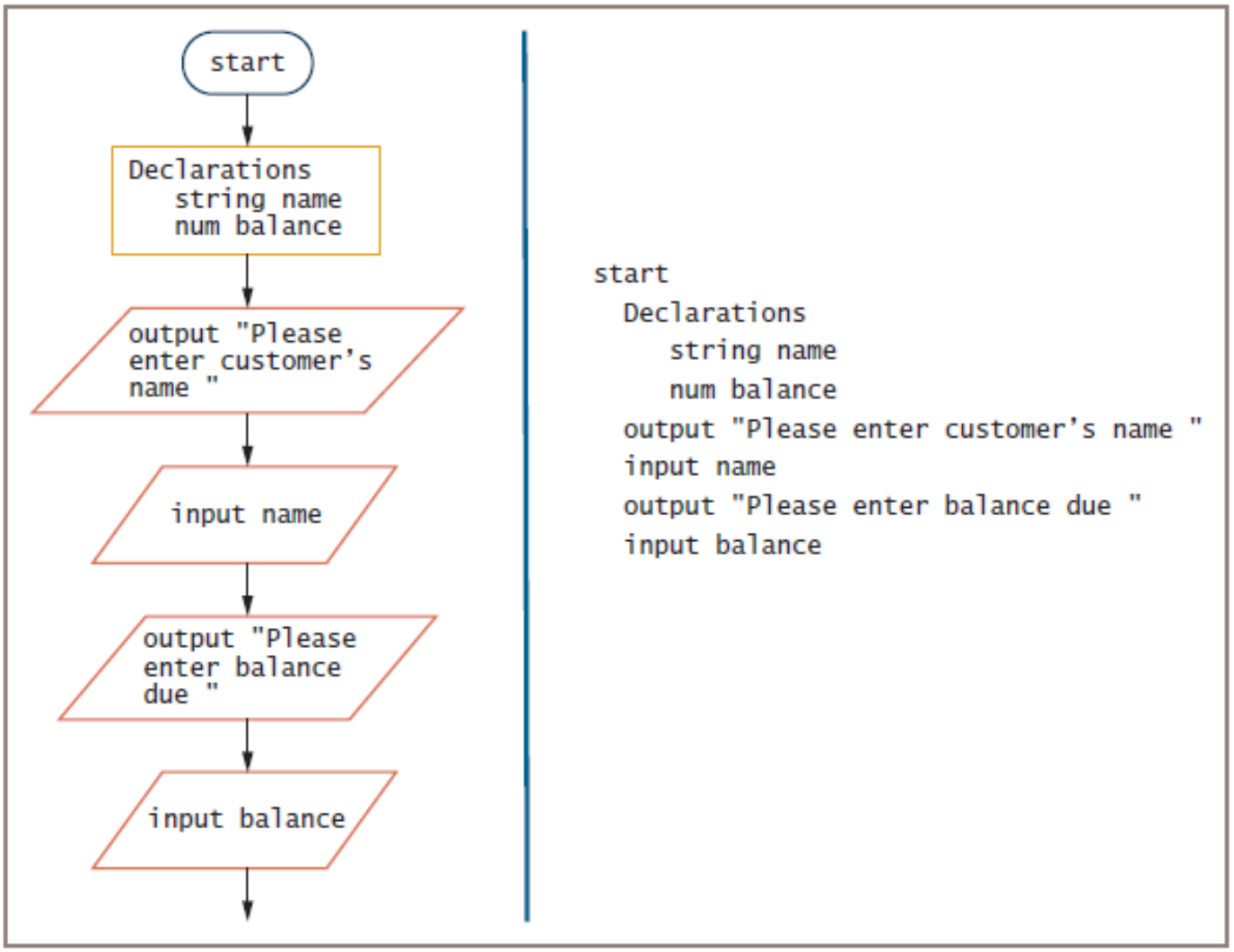


Figure 2-16 Beginning of a program that accepts a name and balance as input and uses a separate prompt for each item

Features of Good Program Design

- Providing program comments where appropriate
- Choosing good identifiers for program elements
- Design clear statements within your program and modules
- Write clear prompts to the user and echo their input back to them
- **Continue to maintain good programming habits**

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