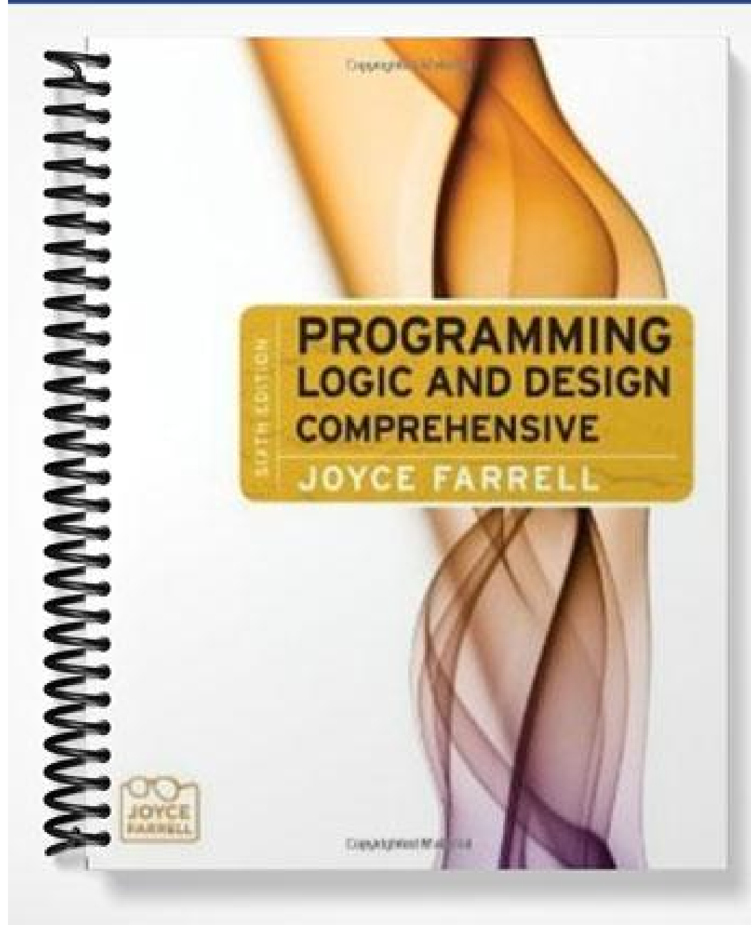
SOLUTIONS MANUAL



Chapter 2

Working with Data, Creating Modules, and Designing High-Quality Programs

At a Glance

Instructor's Manual Table of Contents

- Overview
- Objectives
- Teaching Tips
- Quick Quizzes
- Class Discussion Topics
- Additional Projects
- Additional Resources
- Key Terms

Lecture Notes

Overview

Chapter 2 provides an introduction to declaring and assigning values to variables and constants. Students will learn about the benefits of modularization and how to modularize a program. The most common mainline logic for a program is introduced. Students will learn about hierarchy charts. The chapter concludes with a section on the features of good program design.

Chapter Objectives

In this chapter, students will learn about:

- Declaring and using variables and constants
- Assigning values to variables
- The advantages of modularization
- Modularizing a program
- The most common configuration for mainline logic
- Hierarchy charts
- Some features of good program design

Teaching Tips

Declaring and Using Variables and Constants

1. Remind students how variables are used in a computer program. Introduce the three different forms of data used in a program: variables, literals, and named constants.

Working with Variables

1. Review the example of using a variable in Figure 2-1. Define the terms **declaration**, **identifier**, and **data type** and explain how these concepts apply to variables in a program. Note that you can declare a starting or initial value for a variable. This is known as **initializing the variable**. If a variable is not initialized, it has an unknown value, referred to as **garbage**.

Teaching Tip	Emphasize the importance of properly initializing variables before using them in a program. Using a variable containing a garbage value will result in a logic error.

Naming Variables

1. Note that each computer programming language has its own rules about variable naming. However, there are common rules that should be applied (listed on page 45). Introduce the term **camel casing** and provide several examples.

Teaching Tip	Initiate a class discussion on choosing good variable names.

Understanding Unnamed, Literal Constants and their Data Types

1. Explain that the computer data can be divided into text and numeric data. Specific numbers are called **numeric constants**, while specific text values are called **string constants**. Values such as the number 43 or the character string "Amanda" are called **unnamed constants**.

Understanding the Data Types of Variables

1. Describe the following data types:
   a. **Numeric variable**
   b. **String variable**

Teaching Tip	Note that most programming languages provide additional types such as date/time and Boolean (true/false).

2. Review the assignments of values to variables on pages 46-47.

Declaring Named Constants

1. Discuss the benefits of using **named constants** over **magic numbers**. Explain how to declare a named constant.

Assigning Values to Variables

1. Introduce the concept of an **assignment statement** along with the **assignment operator**. Review the examples of assignment statements on pages 48-49.

Performing Arithmetic Operations

1. Note that most programming languages use standard arithmetic operators: +, -, *, and /.

2. Remind students that the **rules of precedence** (**order of operations**) from standard mathematics also apply in computer programs. All arithmetic operators have **left-to-right associativity**. The order of precedence and associativity are shown in Table 2-1.

Quick Quiz 1

1. Declaring a starting value is known as ____ the variable.
   Answer: initializing

2. A variable's unknown value is commonly called ____.
   Answer: garbage

3. (True/False) Variable names should have some appropriate meaning.
   Answer: True

4. A(n) ____ variable can hold text, such as letters of the alphabet, and other special characters, such as punctuation marks.
   Answer: string

5. The equal sign is the ____ operator.
   Answer: assignment

Understanding the Advantages of Modularization

1. Define the term **modules** and mention the synonyms **subroutines**, **procedures**, **functions**, and **methods**.

Modularization Provides Abstraction

1. Describe the benefits of **abstraction** that modularization provides. An example is shown on page 53.

Teaching Tip	Assign students to read the following article on abstraction: http://en.wikipedia.org/wiki/Abstraction_(computer_science).

Modularization Allows Multiple Programmers to Work on a Problem

1. Note that modularization facilitates the development of programs by a team of programmers working at the same time.

Modularization Allows You to Reuse Your Work

1. Discuss the benefits of **reusability** and note that using reusable modules leads to **reliable** programs.

Modularizing a Program

1. Describe the structure of a modular program. In such a program, a **main program** provides the **mainline logic** and accesses the modules.

2. Review the three parts of a module:
   a. **Header**
   b. **Body**
   c. **Return statement**

3. Note that module names should following similar conventions to variable names and that module names are commonly followed by a set of parentheses.

4. Describe the flowchart and pseudocode representations of a module, using Figures 2-3 and 2-4 as an example.

5. Introduce the term **functional cohesion** and explain how this applies to selecting the particular program statements that make up a module.

Teaching Tip	Assign students to read the following article on functional cohesion: http://en.wikipedia.org/wiki/Cohesion_(computer_science).

Declaring Variables and Constants within Modules

1. Explain that when a variable or constant is declared within a module, it is only **visible** within the module. Other terms that describe this are **in scope** and **local**. Note that this behavior helps to make modules **portable**.

2. Note that variables can also be **global** when declared at the **program level**.

Understanding the Most Common Configuration for Mainline Logic

1. Review the four main parts of the mainline logic for a procedural program (shown in Figure 2-6):
   a. Declarations
   b. **Housekeeping**
   c. **Detail loop**
   d. **End-of-job**

2. Introduce the sample payroll report shown in Figure 2-7. The next two figures show how to create this report. A flowchart for the program logic is presented in Figure 2-7 and the pseudocode in Figure 2-8.

Creating Hierarchy Charts

1. Introduce the idea of creating a hierarchy chart that shows which program modules call other modules. Examples are shown in Figures 2-10 and 2-11.

Features of Good Program Design

1. Note that the final section of the chapter outlines many of the features of good program design.

Using Program Comments

1. Introduce the idea of using **program comments** to provide documentation for the program. Some examples are shown in Figure 2-12. Note that in a flowchart, **annotation symbols** can be used to represent comments, as shown in Figure 2-13.

Teaching Tip	Most programming languages provide conventions on where to place comments. Additionally, some languages provide the ability to generate program documentation from comments.

Choosing Identifiers

1. Stress the importance of using good identifiers for variables, constants, and modules. Guidelines for good identifier names are provided on pages 71-72.

Designing Clear Statements

1. Note that it is important to create clear statements in a program by:
    a. Avoiding confusing line breaks
    b. Using **temporary variables** to clarify long statements

Writing Clear Prompts and Echoing Input

1. Define the term prompt and review the examples of prompts on page 74. Additional examples are provided in Figures 2-15 and 2-16.

2. Explain how to **echo** user input to confirm the user’s entry. An example is seen in Figure 2-17.

Maintaining Good Programming Habits

1. Remind students that the best programming results are achieved by following the steps outlined in the previous chapter.

Quick Quiz 2

1. What are some of the other names for modules?
   Answer: subroutines, procedures, functions, or methods

2. The feature of modular programs that allows individual modules to be used in a variety of applications is known as ____.
   Answer: reusability

3. Programmers say the data items are ____ only within the module in which they are declared.
   Answer: visible or in scope

4. (True/False) A hierarchy chart tells you what tasks are to be performed *within* a module, *when* the modules are called, *how* a module executes, and *why* they are called.
   Answer: False

5. When program input should be retrieved from a user, you almost always want to provide a(n) ____ for the user.
   Answer: prompt

Class Discussion Topics

1. Discuss how to determine which data type to choose for a variable.

2. Ask students whether they can think of additional benefits of modularization that are not covered in the chapter.

Additional Projects

1. Create either pseudocode or a flowchart for a program that does the following:
   Prompt the user to enter a sales tax rate. Prompt the user to enter a price. Calculate and output the amount of tax for the item and the total price with tax.

2. Create either pseudocode or a flowchart for a program that does the following:
   Prompt the user to enter two times and then calculate and print the difference between them in minutes.

Additional Resources

1. Article on naming conventions:
   http://en.wikipedia.org/wiki/Naming_conventions_(programming)

2. Some examples for students to practice the order of operations: www.mathgoodies.com/lessons/vol7/order_operations.html

3. Information on modular programming: http://en.wikipedia.org/wiki/Modular_programming

4. Article on scope: http://en.wikipedia.org/wiki/Scope_(programming)

5. Good programming practices: www.kmoser.com/articles/Good_Programming_Practices.php

Key Terms

- **Abstraction** – the process of paying attention to important properties while ignoring nonessential details.
- **Alphanumeric values** – can contain alphabetic characters, numbers, and punctuation.
- **Annotation symbol** – contains information that expands on what appears in another flowchart symbol; it is most often represented by a three-sided box that is connected to the step it references by a dashed line.
- **Assignment operator** – the equal sign; it is used to assign a value to the variable or constant on its left.
- **Assignment statement** – assigns a value from the right of an assignment operator to the variable or constant on the left of the assignment operator.
- **Binary operator** – an operator that requires two operands—one on each side.
- **Camel casing** – the format for naming variables in which the initial letter is lowercase, multiple-word variable names are run together, and each new word within the variable name begins with an uppercase letter.
- **Data dictionary** – a list of every variable name used in a program, along with its type, size, and description.
- **Data type** – a classification that describes what values can be assigned, how the variable is stored, and what types of operations can be performed with the variable.
- **Declaration** – a statement that provides a data type and an identifier for a variable.
- **Detail loop tasks** – include the steps that are repeated for each set of input data.
- **Echoing input** – the act of repeating input back to a user either in a subsequent prompt or in output.
- **Encapsulation** – the act of containing a task's instructions in a module.
- **End-of-job tasks** – hold the steps you take at the end of the program to finish the application.
- **External documentation** – documentation that is outside a coded program.
- **Floating-point** – number is a number with decimal places.
- **Functional cohesion** – a measure of the degree to which all the module statements contribute to the same task.
- **Functional decomposition** – the act of reducing a large program into more manageable modules.
- **Garbage** – describes the unknown value stored in an unassigned variable.
- **Global** – describes variables that are known to an entire program.

- **Hierarchy chart** – a diagram that illustrates modules' relationships to each other.
- **Housekeeping tasks** – include steps you must perform at the beginning of a program to get ready for the rest of the program.
- **Hungarian notation** – a variable-naming convention in which a variable's data type or other information is stored as part of its name.
- **Identifier** – a variable's name.
- **In scope** – describes the state of data that is visible.
- **Initializing a variable** – the act of assigning its first value, often at the same time the variable is created.
- **Integer** – a whole number.
- **Internal documentation** – documentation within a coded program.
- **Keywords** – comprise the limited word set that is reserved in a language.
- **Left-to-right associativity** – describes operators that evaluate the expression to the left first.
- **Local** – describes variables that are declared within the module that uses them.
- **lvalue** – the memory address identifier to the left of an assignment operator.
- **Magic number** – an unnamed constant whose purpose is not immediately apparent.
- **Main program** – runs from start to stop and calls other modules.
- **Mainline logic** – the logic that appears in a program's main module; it calls other modules.
- **Making declarations** or **declaring variables** – describes the process of naming variables and assigning a data type to them.
- **Mnemonic** – a memory device; variable identifiers act as mnemonics for hard-to-remember memory addresses.
- **Modularization** – the process of breaking down a program into modules.
- **Module's body** – contains all the statements in the module.
- **Module's header** – includes the module identifier and possibly other necessary identifying information.
- **Module's return statement** – marks the end of the module and identifies the point at which control returns to the program or module that called the module.
- **Modules** – small program units that you can use together to make a program. Programmers also refer to modules as **subroutines**, **procedures**, **functions**, or **methods**.
- **Named constant** – similar to a variable, except that its value cannot change after the first assignment.
- **Numeric constant** (**literal numeric constant**) – a specific numeric value.
- **Numeric variable** – one that can hold digits, have mathematical operations performed on it, and usually can hold a decimal point and a sign indicating positive or negative.
- **Order of operations** – describes the rules of precedence.
- **Overhead** – describes the extra resources a task requires.
- **Pascal casing** – the format for naming variables in which the initial letter is uppercase, multiple-word variable names are run together, and each new word within the variable name begins with an uppercase letter.
- **Portable** – module that can more easily be reused in multiple programs.
- **Program comments** – written explanations that are not part of the program logic but that serve as documentation for those reading the program.
- **Program level** – where global variables are declared.
- **Prompt** – a message that is displayed on a monitor to ask the user for a response and perhaps explain how that response should be formatted.

- **Real numbers** – floating-point numbers.
- **Reliability** – the feature of modular programs that assures you a module has been tested and proven to function correctly.
- **Reusability** – the feature of modular programs that allows individual modules to be used in a variety of applications.
- **Right-associativity** and **right-to-left associativity** – describe operators that evaluate the expression to the right first.
- **Rules of precedence** – dictate the order in which operations in the same statement are carried out.
- **Self-documenting** –contain meaningful data and module names that describe the programs' purpose.
- **Stack** – a memory location in which the computer keeps track of the correct memory address to which it should return after executing a module.
- **String constant** (**literal string constant**) – a specific group of characters enclosed within quotation marks.
- **String variable** – can hold text that includes letters, digits, and special characters such as punctuation marks.
- **Temporary variable** (**work variable**) – a working variable that you use to hold intermediate results during a program's execution.
- **Unnamed constant** – a literal numeric or string value.
- **Visible** – describes the state of data items when a module can recognize them.