Chapter 2 (2nd part Module)

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

At a Glance

- Overview
- Objectives
- Quick Quizzes
- Additional Resources
- Key Terms
Overview

Chapter 2 (2nd part) - 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.

Chapter Objectives

In this chapter, students will learn about:

- The advantages of modularization
- Modularizing a program
- The most common configuration for mainline logic
  Hierarchy charts

Understanding the Advantages of Modularization

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

Modularization Provides Abstraction

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

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. Understand the benefits of **reusability** and note that using reusable modules leads to **reliable** programs.
Modularizing a Program

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

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

Declaring Variables and Constants within Modules

1. 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. Understand 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. Review 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. Understand 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.

Quick Quiz

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

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


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

Key Terms

- **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.
- **Local** – describes variables that are declared within the module that uses them.
- **Ivalue** – 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.