

Chapter 2

Applications and Data

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 begins with an overview of creating an application class with a `main()` method. The `class` header and `endClass` statements are introduced. Next, the conventions for choosing identifiers are presented, along with the naming and alignment conventions for methods. The use of string and numeric constants is discussed, along with the conventions for each. Variables are introduced, and the concept of variable declaration is explained. The data types `num` and `string` are illustrated using examples. A discussion on initializing a variable follows, followed by a brief introduction to named constants. Arithmetic operators and the assignment statement are discussed, with an explanation of the rules of precedence. A section on good program design practices follows, including use of temporary variables, prompt messages, and echoing input. The chapter concludes with an introduction to structures.

Chapter Objectives

In this chapter, your students will learn about:

- Creating an application class with a `main()` method
- Using literals, variables, and named constants
- Assigning values to variables
- Arithmetic operations
- Features of good program design
- Structuring and creating a complete program

Teaching Tips

Creating an Application Class with a `main()` method

1. Explain that an application is a program that executes to accomplish some task. Define the term **method** as a set of statements that performs some task or group of tasks. Point out that in most object-oriented programming languages, if a class contains only one method that executes, the method is named `main()`.

Teaching Tip

Explain that although every application is a class, not every class is an application. Chapters 2–5 in this book will deal with application classes, and Chapter 6 will cover classes that are not applications themselves but are used by applications.

2. Use Figure 2-1 on page 32 to show the flowchart and pseudocode for a simple application class that prints the word “Hello.” Explain the use of the `class` header and `endClass` statements, noting that there is only one header and `endClass` statement for each class.
3. Define the term **identifier** as the name of a programming object. Explain that identifiers are simply names for classes, methods, and variables.
4. Discuss the rules for identifier names that are used in this book. Point out to students that a programmer learns the rules for creating identifiers in each programming language they use. Define the term **keyword**, and explain that keywords differ by language.

Understanding the `main()` Method

1. Explain that classes that contain a `main()` method are executable programs. Describe the conventions associated with methods, including naming and alignment. Note that the method’s executable statements are placed between the method header and return statement.

Understanding How Programming Languages Reflect Logic

1. Use Figures 2-2, 2-3, and 2-4 to show differences and similarities of the `Hello` class when implemented in Java, C#, and Visual Basic.

Using Literals, Variables, and Named Constants

1. Explain that data can be input using different types of hardware devices. Define **interactive programs** and **batch programs**, and highlight the differences between them. Note that when data items are input, they are stored in memory variables where they can be processed and converted to output.

Understanding Unnamed, Literal Constants, and their Data Types

1. Introduce the two types of data: text and numeric. Define a numeric constant, noting that it consists of digits and no quotation marks.
2. Define a string constant, explaining that string constants are enclosed in quotation marks and may contain letters, digits, and other special characters. Explain that both string constants and numeric constants are known as unnamed constants, since they do not have identifiers like variables do.

Working with Variables

1. Introduce variables, the named memory locations whose contents can vary or differ over time. Explain that before a variable can be used in a program, it must be declared. Explain that variable declarations include data type and identifier. Use Figure 2-5 on page 37 to illustrate declaration statements.

Teaching Tip	Explain that some programming languages require all variables to be declared at the beginning of a class or method, and others allow declarations to occur anywhere before the variable is first used. This book (course) follows the former convention.
---------------------	--

2. Explain that an item's data type defines the values that can be held by the item, what operations can be performed on it, and how it is stored in memory. Note that this book (course) uses two data types: `num` and `string`.

Teaching Tip	Use the following example to illustrate the concept of data type. Suppose an unknown object is placed on a table. If the person who placed the object tells you it is food, for example, or a tool, then that gives you an idea of what operations can be performed on it. For example, if the item is food, you might infer that you could eat it, cut it, or store it.
---------------------	--

3. Describe what is meant by initializing a variable, that is, to assign a starting value to the variable. Note that a variable can be left uninitialized, but until it is given a value, it is said to contain an unknown value called garbage.

Naming Variables

1. Explain that variable naming follows the same general rules as naming classes and methods, and that the interpreter associates variable names with specific memory addresses.

Understanding a Variable's Data Type

1. Explain that a numeric variable can hold digits and have mathematical operations performed on it. Note that it can also hold a decimal point and a plus or minus sign.
2. Explain that a string variable can hold text, digits, and other special characters and that you cannot perform mathematical operations on it. Note that a variable must be assigned data in a format that matches its declared type. Use the `taxRate` and `inventoryItem` examples on page 39 to illustrate this concept.

Declaring Named Constants

1. Discuss named constants and their conventions. Explain that a named constant is similar to a variable except that it can be assigned a value only once. Explain that a benefit of using a named constant instead of an unnamed constant is that if a change is necessary, you need to make the change only once, and it applies throughout the program.
2. Note that this book uses the convention of all caps, with underscores separating words for named constants.

Assigning Values to Variables

1. Introduce the concept of the assignment statement and the equal sign (=) as the assignment operator. Stress that the expression to the right is always evaluated before the assignment to the left. Explain that operators that work this way are said to have right-associativity, or right-to-left associativity.

Teaching Tip	After explaining the correct use of the assignment statement, use the following example of an incorrect statement using the assignment operator: $a+c=b-d$. Explain that this would only be correct if $a+c$ was a memory location, which is unlikely to be correct in any programming language because it contains the plus sign (+), a mathematical operator.
---------------------	--

Performing Arithmetic Operations

1. Explain that most programming languages use four standard arithmetic operators:
 - + (plus sign), to represent addition
 - (minus sign), to represent subtraction
 - * (asterisk), to represent multiplication
 - / (slash), to represent division
2. Discuss the concept of rules of precedence, also called order of operations. Explain that these dictate the order in which arithmetic operations are carried out. Explain that expressions within parentheses are evaluated first, followed by multiplication and division (from left to right), and then addition and subtraction (from left to right). Note that it is acceptable to use extra parentheses to make a mathematical statement appear clearer to the reader.
3. Use Table 2-1 on page 44 as a visual summary of the rules of precedence.

Features of Good Program Design

1. Explain that certain programming practices make programs easier to write and maintain. Review the bulleted list on page 44.

Using Program Comments

1. Explain that program comments are a type of internal documentation, and explain how this differs from external documentation. Explain that comments help other programmers understand a program's function more easily.

Choosing Identifiers

1. Discuss the importance of choosing appropriate identifiers. Note that nouns are generally appropriate for variables and constants, and that verbs or verb-and-noun combinations can be appropriate identifiers for methods. Note that if meaningful names are chosen, the program is said to be self-documenting.
2. Mention that identifiers should be pronounceable and that abbreviation use should be kept to a minimum. Also, note that digits should be avoided because of confusion with some alphabetic characters.

Teaching Tip	Explain to students that use of abbreviations should be kept to a minimum because their meaning is not always clear. The same abbreviation can mean different things to different people, due to differences in their education, culture, life experiences, profession, and other factors.
---------------------	--

3. Explain that it is generally good practice to separate words using the appropriate programming language convention.
4. For status variables, note that a version of the verb "to be," such as "is" or "are," is often a good choice of identifier.
5. Briefly mention that different organizations sometimes adopt different programming conventions, and that it is the programmer's responsibility to learn and use those.

Designing Clear Statements

1. Discuss the benefits of designing clear statements. Instruct the students to avoid confusing line breaks and to use temporary variables to clarify long statements. Use the example in Figure 2-7 on page 48 to illustrate the use of temporary variables.

Writing Clear Prompts and Echoing Input

1. Explain that a prompt is a message that asks the user for a response. Note that a well-written prompt tells the user how the response should be formatted. Explain the practice of echoing input, repeating the user's response back to the user in the next prompt or in output. Describe the main advantage of echoing input: that the user may be more likely to catch an error in the input data. Use the examples in Figures 2-9 and 2-10 to illustrate the use of prompts and echoing input.

Maintaining Good Programming Habits

1. Mention that developing and maintaining good programming habits will benefit the programmer in the long term, making the process of writing and maintaining code easier.

Quick Quiz 1

1. True or False: Rules for identifiers may differ by programming language.
Answer: True
2. Which of the following may be part of a literal numeric constant?
 - a. Letters of the alphabet
 - b. Parentheses
 - c. Digits
 - d. Punctuation marksAnswer: C
3. A(n) ____ can be assigned a value only once.
Answer: named constant
4. True or False: A variable can be assigned the value of another variable as long as it has the same data type.
Answer: True
5. True or False: A string constant may not contain digits.
Answer: False
6. Giving a variable a starting value is called ____ the variable.
 - a. beginning
 - b. clearing
 - c. declaring
 - d. initializingAnswer: D

7. The purpose of a temporary variable is to:
 - a. break up complex calculations to create clearer code.
 - b. save memory by using fewer memory locations.
 - c. output the variable's value to the user.
 - d. aid in identifying syntax errors.

Answer: A

An Introduction to Structure

1. Briefly define and describe the term **structure**. Explain that every method within every application can be constructed using three basic structures: sequence, selection, or loop. These structures will be covered in much greater detail in Chapters 3 and 4.
2. Using the leftmost image of Figure 2-11 on page 51, introduce the sequence structure.
3. Inform the students that there is no limit on the number of events that a sequence structure may have; however, there is no chance to branch off and skip any of the events. Emphasize the “straight-through” aspect of this structure, with no branches.

Teaching Tip	A sequence structure can be compared to a person's life span. Although a variety of activities can occur, each age level can only be visited once. Within the sequence structure, other structures can occur. A person may take different career paths or even repeat the same activities over and over, but the sequence of a person's life is still linear at the highest level.
---------------------	--

4. Using the center image of Figure 2-11 on page 51, introduce the selection structure. Explain that this structure involves making a decision and following one of two branches of logic based on this decision.
5. Using the rightmost image of Figure 2-11 on page 51, introduce the loop structure, noting that it offers the ability to perform repetitive processing with the same code.

Quick Quiz 2

1. What structure is used to implement repetition?
Answer: the loop structure
2. What structure is used to execute events in a “straight-through” sequential manner?
Answer: the sequence structure
3. What structure involves making a single decision?
Answer: the selection structure

4. True or False: Every method contains at least one of the three types of structures.

Answer: True

Class Discussion Topics

1. Ask students to use the Web to find some examples of echoing input in the applications they encounter. In addition, they should find examples of where the user input is not echoed and discuss whether or not this is appropriate for those situations.
2. Discuss the need for clarity and order when designing program structures. An example might be the act of getting a glass of milk from the refrigerator. What are the specific steps that must be taken, and in what order must they be taken? Discuss the amount of detail involved. (Example: You must open the refrigerator before you grab the milk container. You must get the glass before you pour the milk. You had better ensure that the glass has been placed right-side up before you pour the milk!) There are decisions to be made that will influence the actions to follow. (Example: If the milk carton is not empty, pour milk into glass. If the milk carton is empty after you fill the glass, do not return it to the refrigerator.)

This type of exercise helps students understand what an algorithm is, helps them appreciate the need for careful analysis of the problem to be solved, and helps them recognize that much detail can be involved in designing the solution.

3. By now, the students are aware of the existence of many programming languages and that each may have different conventions, keywords, and syntax. Ask students why they think these language differences exist. Do they think it would be better to have one universal programming language? Discuss how programming has evolved over time and why different applications may be better suited to one language than another. One reason is that with a large number of programmers working in different regions of the world and on different projects, individual conventions and usage will naturally differ.

Additional Projects

1. Have the students select one or two methods to represent any task and identify what types of structures will be needed to implement these methods. Ask the students to create flowchart diagrams of these methods.
2. Have the students interview a programmer who has been working for at least 20 years. Here are some questions that might be appropriate:
 - a. How many languages do you know?
 - b. What changes in programming methodologies have you seen over your career?
 - c. Did you have to change your programming style from unstructured to structured? If so, what do you see as the advantages and disadvantages of structured programming?
 - d. What changes do you think a new programmer entering this field will be likely to see in the near future?

- e. Can you describe your company's process for designing a new software application? Are there design reviews, written specifications, etc.?
- f. Are there peer code reviews before new code is put into production?

To summarize, ask the students if what they heard in the interview changed their perspective of the programming process, and if so, how?

Additional Resources

1. An introduction to the terminology of computing plus some history and a brief look at the structure of a computer program:
www.freenetpages.co.uk/hp/alan.gauld/tutor2/tutwhat.htm
2. Identifier naming conventions for the Java programming language:
<http://www.oracle.com/technetwork/java/codeconventions-135099.html#367>
3. Drawing a structured flowchart:
www.rff.com/structured_flowchart.htm

Key Terms

- **Alphanumeric values** can contain alphabetic characters, numbers, and punctuation.
- An **application** is a program that you execute to accomplish some task.
- The **assignment operator** is the equal sign (=); it is used to assign a value to the variable or constant on its left.
- An **assignment statement** assigns a value from the right of an assignment operator to the variable or constant on the left of the assignment operator.
- **Batch programs** execute on large quantities of data without human intervention for each record; they accept data from a storage device such as a disk.
- A **binary operator** is an operator that requires two operands: one on each side.
- **Camel casing** (sometimes called **lower camel casing**) is 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.
- A **class header** starts a class; it contains the word class and an identifier.
- A **data dictionary** is a list of every variable name used in a program, along with its type, size, and description.
- An item's **data type** describes what values can be held by the item, how the item is stored in computer memory, and what operations can be performed on the data item.
- **Echoing input** is the act of repeating input back to a user either in a subsequent prompt or in output.
- **External documentation** is documentation that is outside a coded program in separate documents.
- A **floating-point** number is a number with decimal places.
- **Garbage** describes the unknown values that reside in variables that have not been initialized.
- An **identifier** is the name of a programming object such as a class, method, or variable.

- To **initialize** a variable is to provide a first value for it.
- An **integer** is a whole number.
- **Interactive programs** execute with frequent intervention from a user with an input device such as a keyboard or a mouse.
- **Internal documentation** is documentation within a coded program that helps explain the meaning and purpose of program elements.
- **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.
- A **loop structure** repeats instructions based on a decision.
- An **lvalue** is the memory address identifier to the left of an assignment operator.
- A **magic number** is an unnamed constant whose purpose is not immediately apparent.
- The **main method** is an application's primary method.
- A **method** is a named set of statements that performs some task or group of tasks within an application.
- A **method header** starts a method. It contains an identifier followed by parentheses.
- A **named constant** is similar to a variable, except that its value cannot change after the first assignment.
- A **numeric constant** (or **literal numeric constant**) is a specific numeric value.
- A **numeric variable** is one that can hold digits, have mathematical operations performed on it, and usually can hold a decimal point and a plus (+) or minus (-) sign, indicating a positive or negative value.
- An **operand** is a value that is manipulated by an operator.
- The **order of operations** describes the rules of precedence.
- **Overhead** describes the extra resources a task requires.
- **Pascal casing** (or **upper camel casing**) is 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.
- A **prompt** is 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** are floating-point numbers.
- **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.
- An **rvalue** is an operand to the right of an assignment operator.
- A **selection structure** contains a decision in which the logic can break in one of two paths.
- **Self-documenting** programs are those that contain meaningful data, method, and class names that describe their purposes.
- A **sequence structure** contains steps that execute in order with no option of branching to skip or repeat any of the tasks.
- A **string constant** (or **literal string constant**) is a specific group of characters enclosed within quotation marks.
- A **string variable** can hold text that includes letters, digits, and special characters such as punctuation marks.
- A **structure** is a basic unit of programming logic.
- A **temporary variable** (or a **work variable**) is a working variable that you use to hold intermediate results during a program's execution.

- **Type casting** is the act of converting data from one type to another.
- An **unnamed constant** is a literal numeric or string value.
- A **variable** is a named memory location with contents that can change.
- A **variable declaration** is a statement that provides a data type and identifier for a variable.
- **White space** describes any character that appears to be empty, such as a space or tab.