Chapter 2

Elements of High-Quality Programs

Chapter Introduction

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

2-1 Declaring and Using Variables and Constants

As you learned in Chapter 1, data items include all the text, numbers, and other information that are processed by a computer. When you input data items into a computer, they are stored in variables in memory where they can be processed and converted to information that is output.

When you write programs, you work with data in three different forms: literals (or unnamed constants), variables, and named constants.

2-1a Understanding Unnamed, Literal Constants and their Data Types

All programming languages support two broad data types; **numeric** (Numeric describes data that consists of numbers.) describes data that consists of numbers and **string** (Describes data that is nonnumeric.) describes data that is nonnumeric. Most programming languages support several additional data types, including multiple types for numeric values of different sizes and with and without decimal places. Languages such as C++, C#, Visual Basic, and Java distinguish between **integer** (A whole number.) (whole number) numeric variables and **floating-point** (A fractional, numeric variable that contains a decimal point.) (fractional) numeric variables that contain a decimal point. (Floating-point numbers are also called **real numbers** (Floating-point numbers.) .) Thus, in some languages, the values 4 and 4.3 would be stored in different types of numeric variables. Additionally, many languages allow you to distinguish between smaller and larger values that occupy different numbers of bytes in memory. You will learn more about these specialized data types when you study a programming language, but this book uses the two broadest types: numeric and string.

When you use a specific numeric value, such as 43, within a program, you write it using the digits and no quotation marks. A specific numeric value is often called a **numeric constant** (A specific numeric value.) (or **literal numeric constant** (A specific numeric value.)) because it does not change—a 43 always has the value 43. When you store a numeric value in computer memory, additional characters such as dollar signs and commas are not input or stored. Those characters can be added to output for readability, but they are not part of the number.

A specific text value, or string of characters, such as “Amanda”, is a **string constant** (A specific group of characters enclosed within quotation marks.) (or **literal string constant** (A specific group of characters enclosed within quotation marks.)). String constants, unlike numeric constants, appear within quotation marks in computer programs. String values are also called **alphanumeric values** (The set of values that include alphabetic characters, numbers, and punctuation.) because they can contain alphabetic characters as well as numbers and other characters. For example, “$3,215.99 U.S.”, including the dollar sign, comma, period, letters, and numbers, is a string. Although strings can contain numbers, numeric values cannot contain alphabetic characters. The numeric constant 43 and the string constant “Amanda” are examples of **unnamed constant** (A literal numeric or string value.) —they do not have identifiers like variables do.

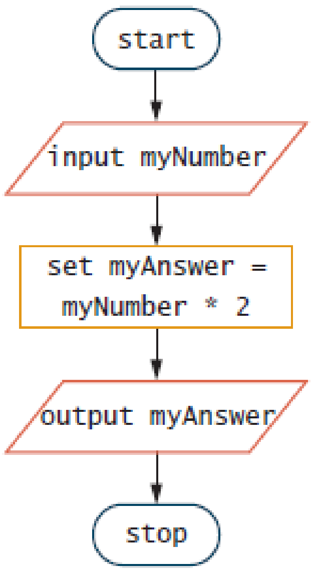
Watch the video *Declaring Variables and Constants.*

2-1b Working with Variables

Variables are named memory locations whose contents can vary or differ over time. For example, in the number-doubling program in Figure 2-1, `myNumber` and `myAnswer` are variables. At any moment in time, a variable holds just one value. Sometimes, `myNumber` holds 2 and `myAnswer` holds 4; at other times, `myNumber` holds 6 and `myAnswer` holds 12. The ability of memory variables to change in value is what makes computers and programming worthwhile. Because one memory location can be used repeatedly with different values, you can write program instructions once and then use them for thousands of separate calculations. *One* set of payroll instructions at your company produces each employee paycheck, and *one* set of instructions at your electric company produces each household's bill.

Figure 2-1

Flowchart and Pseudocode for the Number-Doubling Program



```
start
   input myNumber
   set myAnswer = myNumber * 2
   output myAnswer
stop
```

In most programming languages, before you can use any variable, you must include a declaration for it. A **declaration** (A statement that names a variable and its data type.) is a statement that provides a data type and an identifier for a variable. An **identifier** (A program component's name.) is a program component's name. A data item's **data type** (The characteristic of a variable that describes the kind of values the variable can hold and the types of operations that can be performed with it.) is a classification that describes the following:

- 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

As mentioned earlier, most programming languages support several data types, but in this book, only two data types will be used: `num` and `string`.

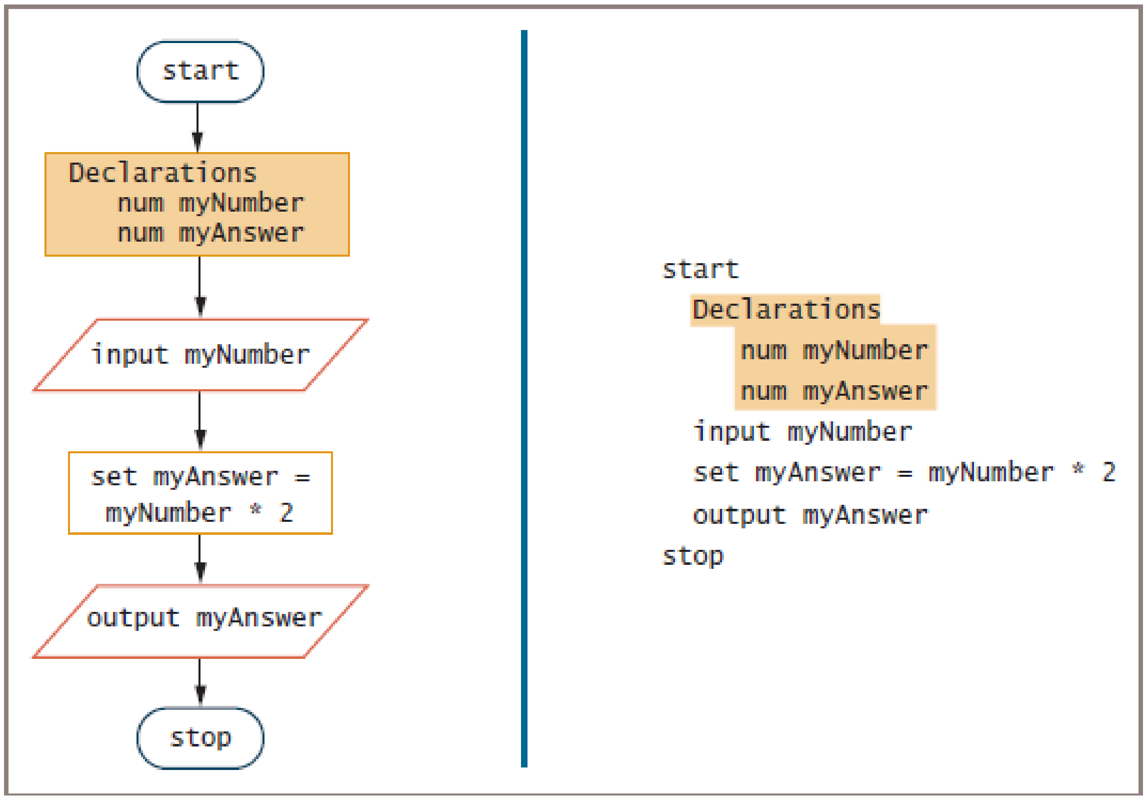
When you declare a variable, you provide both a data type and an identifier. Optionally, you can declare a starting value for any variable. Declaring a starting value is known as **initializing a variable** (The act of assigning the first value to a variable, often at the same time the variable is created.) . For example, each of the following statements is a valid declaration. Two of the statements include initializations, and two do not:

```
num mySalary
num yourSalary = 14.55
string myName
string yourName = "Juanita"
```

Figure 2-2 shows the number-doubling program from Figure 2-1 with the added declarations shaded. Variables must be declared before they are used in a program for the first time. Some languages require all variables to be declared at the beginning of the program; others allow variables to be declared anywhere as long as they are declared before their first use. This book will follow the convention of declaring all variables together.

Figure 2-2

Flowchart and Pseudocode of Number-Doubling Program with Variable Declarations



```
start
   Declarations
      num myNumber
      num myAnswer
   input myNumber
   set myAnswer = myNumber * 2
   output myAnswer
stop
```

In many programming languages, if you declare a variable and do not initialize it, the variable contains an unknown value until it is assigned a value. A variable's unknown value commonly is called **garbage** (Describes the unknown value stored in an unassigned variable.) . Although some languages use a default value for some variables (such as assigning 0 to any unassigned numeric variable), this book will assume that an unassigned variable holds garbage. In many languages it is illegal to use a garbage-holding variable in an arithmetic statement or to display it as output. Even if you work with a language that allows you to display garbage, it serves no purpose to do so and constitutes a logical error.

When you create a variable without assigning it an initial value (as with `myNumber` and `myAnswer` in Figure 2-2), your intention is to assign a value later—for example, by receiving one as input or placing the result of a calculation there.

2-1c Naming Variables

The number-doubling example in Figure 2-2 requires two variables: `myNumber` and `myAnswer`. Alternatively, these variables could be named `userEntry` and `programSolution`, or `inputValue` and `twiceTheValue`. As a programmer, you choose reasonable and descriptive names for your variables. The language interpreter then associates the names you choose with specific memory addresses.

Every computer programming language has its own set of rules for creating identifiers. Most languages allow letters and digits within identifiers. Some languages allow hyphens in variable names, such as `hourly-wage`, and some allow underscores, as in `hourly_wage`. Some languages allow dollar signs or other special characters in variable names (for example, `hourly$`); others allow foreign-alphabet characters, such as π or Ω. Each programming language has a few (perhaps 100 to 200) reserved **keywords** (The limited word set that is reserved in a language.) that are not allowed as variable names because they are part of the language's syntax. When you learn a programming language, you will learn its list of keywords.

Different languages put different limits on the length of variable names, although in general, the length of identifiers in newer languages is virtually unlimited. In many languages, identifiers are case sensitive, so `HoUrLyWaGe`, `hourlywage`, and `hourlyWage` are three separate variable names. Programmers use multiple conventions for naming variables, often depending on the programming language or standards implemented by their employers. Common conventions include the following:

- **Camel casing** (A naming convention in which the initial letter is lowercase, multiple-word names are run together, and each new word within the name begins with an uppercase letter.) is the convention in which the variable starts with a lowercase letter and any subsequent word begins with an uppercase letter, such as `hourlyWage`. The variable names in this book are shown using camel casing.

- **Pascal casing** (A naming convention in which the initial letter is uppercase, multiple-word names are run together, and each new word within the name begins with an uppercase letter.) is a convention in which the first letter of a variable name is uppercase, as in `HourlyWage`.

- **Hungarian notation** (A variable-naming convention in which a variable's data type or other information is stored as part of its name.) is a convention in which a variable's data type is part of the identifier—for example, `numHourlyWage` or `stringLastName`.

Adopting a naming convention for variables and using it consistently will help make your programs easier to read and understand.

Even though every language has its own rules for naming variables, you should not concern yourself with the specific syntax of any particular computer language when designing the logic of a program. The logic, after all, works with any language. The variable names used throughout this book follow only three rules:

1. *Variable names must be one word*. The name can contain letters, digits, hyphens, underscores, or any other characters you choose, with the exception of spaces. Therefore, `r` is a legal variable name, as are `rate` and `interestRate`. The variable name `interest rate` is not allowed because of the space.

2. *Variable names must start with a letter*. Some programming languages allow variable names to start with a nonalphabetic character such as an underscore. Almost all programming languages prohibit variable names that start with a digit. This book follows the most common convention of starting variable names with a letter.

When you write a program using an editor that is packaged with a compiler in an IDE, the compiler may display variable names in a different color from the rest of the program. This visual aid helps your variable names stand out from words that are part of the programming language.

3. *Variable names should have some appropriate meaning*. This is not a formal rule of any programming language. When computing an interest rate in a program, the computer does not care if you call the variable `g`, `u84`, or `fred`. As long as the correct numeric result is placed in the variable, its actual name doesn't matter. However, it's much easier to follow the logic of a statement like `set interestEarned = initialInvestment * interestRate` than a statement like `set f = i * r` or `set someBanana = j89 * myFriendLinda`. When a program requires changes, which could be months or years after you write the original version, you and your fellow programmers will appreciate clear, descriptive variable names in place of cryptic identifiers. Later in this chapter, you will learn more about selecting good identifiers.

Notice that the flowchart in Figure 2-2 follows the preceding rules for variables: Both variable names, `myNumber` and `myAnswer`, are single words without embedded spaces, and they have appropriate meanings. Some programmers name variables after friends or create puns with them, but computer professionals consider such behavior unprofessional and amateurish.

2-1d Assigning Values to Variables

When you create a flowchart or pseudocode for a program that doubles numbers, you can include a statement such as the following:

```
set myAnswer = myNumber * 2
```

Such a statement is an **assignment statement** (A statement that stores the result of any value on its right side to the named location on its left side.) . This statement incorporates two actions. First, the computer calculates the arithmetic value of `myNumber * 2`. Second, the computed value is stored in the `myAnswer` memory location.

The equal sign is the **assignment operator** (The equal sign; it always requires the name of a memory location on its left side.) . The assignment operator is an example of a **binary operator** (An operator that requires two operands—one on each side.) , meaning it requires two operands—one on each side. The assignment operator always operates from right to left, which means that it has **right-associativity** (Descriptions of operators that evaluate the expression to the right first.) or **right-to-left associativity** (Descriptions of operators that evaluate the expression to the right first.) . This means that the value of the expression to the right of the assignment operator is evaluated first, and then the result is assigned to the operand on the left. The operand to the right of an assignment operator can be a value, a formula, a named constant, or a variable. The operand to the left of an assignment operator must be a name that represents a memory address—the name of the location where the result will be stored.

For example, if you have declared two numeric variables named `someNumber` and `someOtherNumber`, then each of the following is a valid assignment statement:

```
set someNumber = 2
set someNumber = 3 + 7
set someOtherNumber = someNumber
set someOtherNumber = someNumber * 5
```

In each case, the expression to the right of the assignment operator is evaluated and stored at the location referenced on the left side. The result to the left of an assignment operator is called an **lvalue** (The memory address identifier to the left of an assignment operator.) . The *l* is for left. Lvalues are always memory address identifiers.

The following statements, however, are *not* valid:

```
set 2 + 4 = someNumber
set someOtherNumber * 10 = someNumber
```

Don't Do It
The operand to the left of an assignment operator must represent a memory address.

In each of these cases, the value to the left of the assignment operator is not a memory address, so the statements are invalid.

When you write pseudocode or draw a flowchart, it might help you to use the word *set* in assignment statements, as shown in these examples, to emphasize that the left-side value is being set. However, in most programming languages, the word *set* is not used, and assignment statements take the following simpler form:

```
someNumber = 2
someOtherNumber = someNumber
```

Because the abbreviated form is how assignments appear in most languages, it is used for the rest of this book.

2-1e Understanding the Data Types of Variables

Computers handle string data differently from the way they handle numeric data. You may have experienced these differences if you have used application software such as spreadsheets or database programs. For example, in a spreadsheet, you cannot sum a column of words. Similarly, every programming language requires that you specify the correct type for each variable, and that you use each type appropriately.

- A **numeric variable** (A variable that holds numeric values.) is one that can hold digits and have mathematical operations performed on it. In this book, all numeric variables can hold a decimal point and a sign indicating positive or negative; some programming languages provide specialized numeric types for these options. In the statement `myAnswer = myNumber * 2`, both `myAnswer` and `myNumber` are numeric variables; that is, their intended contents are numeric values, such as 6 and 3, 14.8 and 7.4, or –18 and –9.
- A **string variable** (A variable that can hold text that includes letters, digits, and special characters such as punctuation marks.) can hold text, such as letters of the alphabet, and other special characters, such as punctuation marks. If a working program contains the statement `lastName = "Lincoln"`, then `lastName` is a string variable. A string variable also can hold digits either with or without other characters. For example, “235 Main Street” and “86” are both strings. A string like “86” is stored differently than the numeric value 86, and you cannot perform arithmetic with the string.

Type-safety (The feature of programming languages that prevents assigning values of an incorrect data type.) is the feature of programming languages that prevents assigning values of an incorrect data type. You can assign data to a variable only if it is the correct type. If you declare `taxRate` as a numeric variable and `inventoryItem` as a string, then the following statements are valid:

```
taxRate = 2.5
inventoryItem = "monitor"
```

The following are invalid because the type of data being assigned does not match the variable type:

```
taxRate = "2.5"
inventoryItem = 2.5
```

Don't Do It
If `taxRate` is numeric and `inventoryItem` is a string, then these assignments are invalid.

Watch the video *Understanding Data Types*.

2-1f Declaring Named Constants

Besides variables, most programming languages allow you to create named constants. A **named constant** (A named memory location, similar to a variable, except its value never changes during the execution of a program. Conventionally, constants are named using all capital letters.) is similar to a variable, except it can be assigned a value only once. You use a named constant when you want to assign a useful name for a value that will never be changed during a program's execution. Using named constants makes your programs easier to understand by eliminating magic numbers. A **magic number** (An unnamed numeric constant.) is an unnamed constant, like 0.06, whose purpose is not immediately apparent.

For example, if a program uses a sales tax rate of 6 percent, you might want to declare a named constant as follows:

```
num SALES_TAX_RATE = 0.06
```

After `SALES_TAX_RATE` is declared, the following statements have identical meaning:

```
taxAmount = price * 0.06
taxAmount = price * SALES_TAX_RATE
```

The way in which named constants are declared differs among programming languages. This book follows the convention of using all uppercase letters in constant identifiers, and using underscores to separate words for readability. Using these conventions makes named constants easier to recognize. In many languages a constant must be assigned its value when it is declared, but in some languages a constant can be assigned its value later. In both cases, however, a constant's value cannot be changed after the first assignment. This book follows the convention of initializing all constants when they are declared.

When you declare a named constant, program maintenance becomes easier. For example, if the value of the sales tax changes from 0.06 to 0.07 in the future, and you have declared a named constant `SALES_TAX_RATE`, you only need to change the value assigned to the named constant at the beginning of the program, then retranslate the program into machine language, and all references to `SALES_TAX_RATE` are automatically updated. If you used the unnamed literal 0.06 instead, you would have to search for every instance of the value and replace it with the new one. Additionally, if the literal 0.06 was used in other calculations within the program (for example, as a discount rate), you would have to carefully select which instances of the value to alter, and you would be likely to make a mistake.

Sometimes, using unnamed literal constants is appropriate in a program, especially if their meaning is clear to most readers. For example, in a program that calculates half of a value by dividing by two, you might choose to use the unnamed literal 2 instead of incurring the extra time and memory costs of creating a named constant `HALF` and assigning 2 to it. Extra costs that result from adding variables or instructions to a program are known as **overhead** (All the resources and time required by an operation.) .

Two Truths & A Lie

Declaring and Using Variables and Constants

1. A variable's data type describes the kind of values the variable can hold and the types of operations that can be performed with it.

   T F

2. If `name` is a string variable, then the statement `set name = "Ed"` is valid.

   T F

3. The operand to the right of an assignment operator must be a name that represents a memory address.

T F

Chapter 2: Elements of High-Quality Programs: 2-2 Performing Arithmetic Operations
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)
© 2013 ,

2-2 Performing Arithmetic Operations

Most programming languages use the following standard arithmetic operators:

+ (plus sign)—addition

– (minus sign)—subtraction

* (asterisk)—multiplication

/ (slash)—division

Many languages also support additional operators that calculate the remainder after division, raise a number to a higher power, manipulate individual bits stored within a value, and perform other operations.

Each of the standard arithmetic operators is a binary operator; that is, each requires an expression on both sides. For example, the following statement adds two test scores and assigns the sum to a variable named `totalScore`:

```
totalScore = test1 + test2
```

The following adds 10 to `totalScore` and stores the result in `totalScore`:

```
totalScore = totalScore + 10
```

In other words, this example increases the value of `totalScore`. This last example looks odd in algebra because it might appear that the value of `totalScore` and `totalScore` plus 10 are equivalent. You must remember that the equal sign is the assignment operator, and that the statement is actually taking the original value of `totalScore`, adding 10 to it, and assigning the result to the memory address on the left of the operator, which is `totalScore`.

In programming languages, you can combine arithmetic statements. When you do, every operator follows **rules of precedence** (rules of precedence dictate the order in which operations in the same statement are carried out.) (also called the **order of operations** (Describes the rules of precedence.)) that dictate the order in which operations in the same statement are carried out. The rules of precedence for the basic arithmetic statements are as follows:

- Expressions within parentheses are evaluated first. If there are multiple sets of parentheses, the expression within the innermost parentheses is evaluated first.
- Multiplication and division are evaluated next, from left to right.
- Addition and subtraction are evaluated next, from left to right.

The assignment operator has a very low precedence. Therefore, in a statement such as `d = e * f + g`, the operations on the right of the assignment operator are always performed before the final assignment to the variable on the left.

When you learn a specific programming language, you will learn about all the operators that are used in that language. Many programming language books contain a table that specifies the relative precedence of every operator used in the language.

For example, consider the following two arithmetic statements:

```
firstAnswer = 2 + 3 * 4
secondAnswer = (2 + 3) * 4
```

After these statements execute, the value of `firstAnswer` is 14. According to the rules of precedence, multiplication is carried out before addition, so 3 is multiplied by 4, giving 12, and then 2 and 12 are added, and 14 is assigned to `firstAnswer`. The value of `secondAnswer`, however, is 20, because the parentheses force the contained addition operation to be performed first. The 2 and 3 are added, producing 5, and then 5 is multiplied by 4, producing 20.

Forgetting about the rules of arithmetic precedence, or forgetting to add parentheses when you need them, can cause logical errors that are difficult to find in programs. For example, the following statement might appear to average two test scores:

```
average = score1 + score2 / 2
```

However, it does not. Because division has a higher precedence than addition, the preceding statement takes half of `score2`, adds it to `score1`, and stores the result in `average`. The correct statement is:

```
average = (score1 + score2) / 2
```

You are free to add parentheses even when you don’t need them to force a different order of operations; sometimes you use them just to make your intentions clearer. For example, the following statements operate identically:

In both cases, `price` is multiplied by `TAX_RATE` first, then it is added to `price`, and finally the result is stored in `totalPriceWithTax`. Because multiplication occurs before addition on the right side of the assignment operator, both statements are the same. However, if you feel that the statement with the parentheses makes your intentions clearer to someone reading your program, then you should use them.

All the arithmetic operators have **left-to-right associativity** (Describes operators that evaluate the expression to the left first.) . This means that operations with the same precedence take place from left to right. Consider the following statement:

Multiplication and division have higher precedence than addition or subtraction, so the multiplication and division are carried out from left to right as follows:

`c` is multiplied by `d`, and the result is divided by `e`, giving a new result.

Therefore, the statement becomes:

Then, addition and subtraction are carried out from left to right as follows:

`a` and `b` are added, the temporary result is added, and then `f` is subtracted. The final result is then assigned to `answer`.

Another way to say this is that the following two statements are equivalent:

Table 2-1 summarizes the precedence and associativity of the five most frequently used operators.

Table 2-1

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

Watch the video *Arithmetic Operator Precedence*.

Two Truths & A Lie

Performing Arithmetic Operations

1. Parentheses have higher precedence than any of the common arithmetic operators.

   T F

2. Operations in arithmetic statements occur from left to right in the order in which they appear.

   T F

3. The following adds 5 to a variable named `points`:

   T F

Chapter 2: Elements of High-Quality Programs: 2-3 Understanding the Advantages of Modularization
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)
© 2013 ,

2-3 Understanding the Advantages of Modularization

Programmers seldom write programs as one long series of steps. Instead, they break down their programming problems into smaller units and tackle one cohesive task at a time. These smaller units are **modules** (A small program unit used with other modules to make a program. Programmers also refer to modules as subroutines, procedures, functions, and methods.) . Programmers also refer to them as **subroutines** (A small program unit used with other modules to make a program. Programmers also refer to modules as subroutines, procedures, functions, and methods.) , **procedures** (A small program unit used with other modules to make a program. Programmers also refer to modules as subroutines, procedures, functions, and methods.) , **functions** (A small program unit used with other modules to make a program. Programmers also refer to modules as subroutines, procedures, functions, and methods.) , or **methods** (A small program unit used with other modules to make a program. Programmers also refer to modules as subroutines, procedures, functions, and methods.) ; the name usually reflects the programming language being used. For example, Visual Basic programmers use *procedure* (or *subprocedure*). C and C++ programmers call their modules *functions*, whereas C#, Java, and other object-oriented language programmers are more likely to use *method*. Programmers in COBOL, RPG, and BASIC (all older languages) are most likely to use *subroutine*.

You can learn about modules that receive and return data in Chapter 9 of the comprehensive version of this book.

A main program executes a module by calling it. To **call a module** (To use a module's name to invoke it, causing it to execute.) is to use its name to invoke the module, causing it to execute. When the module's tasks are complete, control returns to the spot from which the module was called in the main program. When you access a module, the action is similar to putting a DVD player on pause. You abandon your primary action (watching a video), take care of some other task (for example, making a sandwich), and then return to the main task exactly where you left off.

The process of breaking down a large program into modules is **modularization** (The process of breaking down a program into modules.) ; computer scientists also call it **functional decomposition** (Functional decomposition is the act of reducing a large program into more manageable modules.) . You are never required to modularize a large program to make it run on a computer, but there are at least three reasons for doing so:

- Modularization provides abstraction.
- Modularization allows multiple programmers to work on a problem.
- Modularization allows you to reuse work more easily.

2-3a Modularization Provides Abstraction

One reason that modularized programs are easier to understand is that they enable a programmer to see the "big picture." **Abstraction** (The process of paying attention to important properties while ignoring nonessential details.) is the process of paying attention to important properties while ignoring nonessential details. Abstraction is selective ignorance. Life would be tedious without abstraction. For example, you can create a list of things to accomplish today:

Without abstraction, the list of chores would begin:

You might list a dozen more steps before you finish the laundry and move on to the second chore on your original list. If you had to consider every small, low-level detail of every task in your day, you would probably never make it out of bed in the morning. Using a higher-level, more abstract list makes your day manageable. Abstraction makes complex tasks look simple.

Abstract artists create paintings in which they see only the big picture—color and form—and ignore the details. Abstraction has a similar meaning among programmers.

Likewise, some level of abstraction occurs in every computer program. Fifty years ago, a programmer had to understand the low-level circuitry instructions the computer used. But now, newer high-level programming languages allow you to use English-like vocabulary in which one broad statement corresponds to dozens of machine instructions. No matter which high-level programming language you use, if you display a message on the monitor, you are never required to understand how a monitor works to create each pixel on the screen. You write an instruction like `output message` and the details of the hardware operations are handled for you by the operating system.

Modules provide another way to achieve abstraction. For example, a payroll program can call a module named `computeFederalWithholdingTax()`. When you call this module from your program, you use one statement; the module itself might contain dozens of statements. You can write the mathematical details of the module later, someone else can write them, or you can purchase them from an outside source. When you plan your main payroll program, your only concern is that a federal withholding tax will have to be calculated; you save the details for later.

2-3b Modularization Allows Multiple Programmers to Work on a Problem

When you dissect any large task into modules, you gain the ability to more easily divide the task among various people. Rarely does a single programmer write a commercial program that you buy. Consider any word-processing, spreadsheet, or database program you have used. Each program has so many options, and responds to user selections in so many possible ways, that it would take years for a single programmer to write all the instructions. Professional software developers can write new programs in weeks or months, instead of years, by dividing large programs into modules and assigning each module to an individual programmer or team.

2-3c Modularization Allows You to Reuse Work

If a module is useful and well written, you may want to use it more than once within a program or in other programs. For example, a routine that verifies the validity of dates is useful in many programs written for a business. (For example, a month value is valid if it is not lower than 1 or higher than 12, a day value is valid if it is not lower than 1 or higher than 31 if the month is 1, and so on.) If a computerized personnel file contains each employee's birth date, hire date, last promotion date, and termination date, the date-validation module can be used four times with each employee record. Other programs in an organization can also use the module; these programs might ship customer orders, plan employees' birthday parties, or calculate when loan payments should be made. If you write the date-checking instructions so they are entangled with other statements in a program, they are difficult to extract and reuse. On the other hand, if you place the instructions in their own module, the unit is easy to use and portable to other applications. The feature of modular programs that allows individual modules to be used in a variety of applications is **reusability** (The feature of modular programs that allows individual modules to be used in a variety of applications.) .

You can find many real-world examples of reusability. When you build a house, you don't invent plumbing and heating systems; you incorporate systems with proven designs. This certainly reduces the time and effort it takes to build a house. The plumbing and electrical systems you choose are in service in other houses, so they have been tested under a variety of circumstances, increasing their reliability. **Reliability** (The feature of modular programs that ensures a module has been tested and proven to function correctly.) is the feature of programs that assures you a module has been proven to function correctly. Reliable software saves time and money. If you create the functional components of your programs as stand-alone modules and test them in your current programs, much of the work will already be done when you use the modules in future applications.

Two Truths & A Lie

Understanding the Advantages of Modularization

1. Modularization eliminates abstraction, a feature that makes programs more confusing.

   T F

2. Modularization makes it easier for multiple programmers to work on a problem.

   T F

3. Modularization allows you to reuse work more easily.

   T F

2-4 Modularizing a Program

Most programs consist of a **main program** (A program that runs from start to stop and calls other modules; also called a *main program method*.) , which contains the basic steps, or the **mainline logic** (The overall logic of the main program from beginning to end.) , of the program. The main program then accesses modules that provide more refined details.

When you create a module, you include the following:

- A header—The **module header** (The module header includes the module identifier and possibly other necessary identifying information.) includes the module identifier and possibly other necessary identifying information.
- A body—The **module body** (The module body contains all the statements in the module.) contains all the statements in the module.
- A `return` statement—The **module return statement** (The module 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.) marks the end of the module and identifies the point at which control returns to the program or module that called the

module. In most programming languages, if you do not include a `return` statement at the end of a module, the logic will still return. However, this book follows the convention of explicitly including a `return` statement with every module.

Naming a module is similar to naming a variable. The rules for naming modules are slightly different in every programming language, but in this text, module names follow the same general rules used for variable identifiers:

- Module names must be one word and start with a letter.
- Module names should have some meaning.

Although it is not a requirement of any programming language, it frequently makes sense to use a verb as all or part of a module's name, because modules perform some action. Typical module names begin with action words such as `get`, `calculate`, and `display`. When you program in visual languages that use screen components such as buttons and text boxes, the module names frequently contain verbs representing user actions, such as `click` or `drag`.

Additionally, in this text, module names are followed by a set of parentheses. This will help you distinguish module names from variable names. This style corresponds to the way modules are named in many programming languages, such as Java, C++, and C#.

As you learn more about modules in specific programming languages, you will find that you sometimes place variable names within the parentheses of module names. Any variables enclosed in the parentheses contain information you want to send to the module. For now, the parentheses at the end of module names will be empty in this book.

When a main program wants to use a module, it calls the module. A module can call another module, and the called module can call another. The number of chained calls is limited only by the amount of memory available on your computer. In this book, the flowchart symbol used to call a module is a rectangle with a bar across the top. You place the name of the module you are calling inside the rectangle.

Some programmers use a rectangle with stripes down each side to represent a module in a flowchart, and this book uses that convention if a module is external to a program. For example, prewritten, built-in modules that generate random numbers, compute standard trigonometric functions, and sort values often are external to your programs. However, if the module is being created as part of the program, the book uses a rectangle with a single stripe across the top.

In a flowchart, you draw each module separately with its own sentinel symbols. The beginning sentinel contains the name of the module. This name must be identical to the name used in the calling program. The ending sentinel contains `return`, which indicates that when the module ends, the logical progression of statements will exit the module and return to the calling program. Similarly, in pseudocode, you start each module with its name and end with a `return` statement; the module name and `return` statements are vertically aligned and all the module statements are indented between them.

For example, consider the program in Figure 2-3, which does not contain any modules. It accepts a customer's name and balance due as input and produces a bill. At the top of the bill, the company's name and address are displayed on three lines, which are followed by the customer's name and balance due. To display the company name and address, you can simply include three `output` statements in the mainline logic of a program, as shown in Figure 2-3, or you can modularize the program by creating both the mainline logic and a `displayAddressInfo()` module, as shown in Figure 2-4.

Figure 2-3

Program that Produces a Bill Using Only Main Program

Figure 2-4

Program that Produces a Bill Using Main Program that Calls `displayaddressinfo()` Module

When the `displayAddressInfo()` module is called in Figure 2-4, logic transfers from the main program to the `displayAddressInfo()` module, as shown by the large red arrow in both the flowchart and the pseudocode. There, each module statement executes in turn before logical control is transferred back to the main program, where it continues with the statement that follows the module call, as shown by the large blue arrow. Programmers say the statements that are contained in a module have been **encapsulated** (encapsulated is the act of containing a task's instructions in a module.) .

Neither of the programs in Figures 2-3 and 2-4 is superior to the other in terms of functionality; both perform exactly the same tasks in the same order. However, you may prefer the modularized version of the program for at least two reasons:

- First, the main program remains short and easy to follow because it contains just one statement to call the module, rather than three separate `output` statements to perform the work of the module.
- Second, a module is easily reusable. After you create the address information module, you can use it in any application that needs the company's name and address. In other words, you do the work once, and then you can use the module many times.

A potential drawback to creating modules and moving between them is the overhead incurred. The computer keeps track of the correct memory address to which it should return after executing a module by recording the memory address in a location known as the **stack** (A memory location that holds the memory addresses to which method calls should return.) . This process requires a small amount of computer time and resources. In most cases, the advantage to creating modules far outweighs the small amount of overhead required.

Determining when to modularize a program does not depend on a fixed set of rules; it requires experience and insight. Programmers do follow some guidelines when deciding how far to break down modules or how much to put in each of them. Some companies may have arbitrary rules, such as "a module's instructions should never take more than a page," or "a module should never have more than 30 statements," or "never have a module with only one statement." Rather than use such arbitrary rules, a better policy is to place together statements that contribute to one specific task. The more the statements contribute to the same job, the greater the **functional cohesion** (The extent to which all operations in a method contribute to the performance of only one task.) of the module. A module that checks the validity of a date variable's value, or one that asks a user for a value and accepts it as input, is considered cohesive. A module that checks date validity, deducts insurance premiums, and computes federal withholding tax for an employee would be less cohesive.

Chapter 9 of the comprehensive version of this book provides more information on designing modules for high cohesion. It also explores the topic of *coupling*, which is a measure of how much modules depend on each other.

Watch the video *Modularizing a Program*.

2-4a Declaring Variables and Constants within Modules

You can place any statements within modules, including input, processing, and output statements. You also can include variable and constant declarations within modules. For example, you might decide to modify the billing program in Figure 2-4 so it looks like the one in Figure 2-5. In this version of the program, three named constants that hold the three lines of company data are declared within the `displayAddressInfo()` module. (See shading.)

Figure 2-5

The Billing Program with Constants Declared within the Module

The variables and constants declared in a module are usable only within the module. Programmers say the data items are **visible** (A characteristic of data items that means they "can be seen" only within the method in which they are declared.) only within the module in which they are declared. That means the program only recognizes them there. Programmers say that variables and constants declared within a module are **in scope** (The characteristic of variables and constants declared within a method that apply only within that method.) only within that module. Programmers also say that variables and constants are **local** (Describes variables that are declared within the method that uses them.) to the module in which they are declared. In other words, when the strings `LINE1, LINE2,` and `LINE3` are declared in the `displayAddressInfo()` module in Figure 2-5, they are not recognized and cannot be used by the main program.

One of the motivations for creating modules is that separate modules are easily reusable in multiple programs. If the `displayAddressInfo()` module will be used by several programs within the organization, it makes sense that the definitions for its variables and constants must come with it. This makes the modules more **portable** (Describes a module that can more easily be reused in multiple programs.) ; that is, they are self-contained units that are easily transported.

Besides local variables and constants, you can create global variables and constants. **Global** (Describes variables that are known to an entire program.) variables and constants are known to the entire program; they are said to be declared at the **program level** (The level at which global variables are declared.) . That means they are visible to and usable in all the modules called by the program. The opposite is not true—variables and constants declared within a module are not usable elsewhere; they are visible only to that module. (For example, in Figure 2-5, the main program variables `name` and `balance` are global variables, although in this case they are not used in any modules.) For the most part, this book will use only global variables and constants so that the examples are easier to follow and you can concentrate on the main logic.

Many programmers do not approve of using global variables and constants. They are used here so you can more easily understand modularization before you learn the techniques of sending local variables from one module to another. Chapter 9 of the comprehensive version of this book will describe how you can make every variable local.

2-4b Understanding the Most Common Configuration for Mainline Logic

In Chapter 1, you learned that a procedural program contains procedures that follow one another in sequence. The mainline logic of almost every procedural computer program can follow a general structure that consists of three distinct parts:

1. **Housekeeping tasks** (Tasks that must be performed at the beginning of a program to prepare for the rest of the program.) include any steps you must perform at the beginning of a program to get ready for the rest of the program. They can include tasks such as variable and constant declarations, displaying instructions to users, displaying report headings, opening any files the program requires, and inputting the first piece of data.

Inputting the first data item is always part of the housekeeping module. You will learn the theory behind this practice in Chapter 3. Chapter 7 covers file handling, including what it means to open and close a file.

2. **Detail loop tasks** (The steps that are repeated for each set of input data.) do the core work of the program. When a program processes many records, detail loop tasks execute repeatedly for each set of input data until there are no more. For example, in a payroll program, the same set of calculations is executed repeatedly until a check has been produced for each employee.

3. **End-of-job tasks** (End-of-job tasks hold the steps you take at the end of the program to finish the application.) are the steps you take at the end of the program to finish the application. You can call these finish-up or clean-up tasks. They might include displaying totals or other final messages and closing any open files.

Figure 2-6 shows the relationship of these three typical program parts. Notice how the `housekeeping()` and `endOfJob()` tasks are executed just once, but the `detailLoop()` tasks repeat as long as the `eof` condition has not been met. The flowchart uses a flowline to show how the `detailLoop()` module repeats; the pseudocode uses the words `while` and `endwhile` to contain statements that execute in a loop. You will learn more about the `while` and `endwhile` terms in subsequent chapters; for now, understand that they are a way of expressing repeated actions.

Figure 2-6

Flowchart and Pseudocode of Mainline Logic for a Typical Procedural Program

Many everyday tasks follow the three-module format just described. For example, a candy factory opens in the morning, and the machines are started and filled with ingredients. These housekeeping tasks occur just once at the start of the day. Then, repeatedly during the day, candy is manufactured. This process might take many steps, each of which occurs many times. These are the steps in the detail loop. Then, at the end of the day, the machines are cleaned and shut down. These are the end-of-job tasks.

Not all programs take the format of the logic shown in Figure 2-6, but many do. Keep this general configuration in mind as you think about how you might organize many programs. For example, Figure 2-7 shows a sample payroll report for a small company. A user enters employee names until there are no more to enter, at which point the user enters *XXX*. As long as the entered name is not *XXX*, the user enters the employee's weekly gross pay. Deductions are computed as a flat 25 percent of the gross pay, and the statistics for each employee are output. The user enters another name, and as long as it is not *XXX*, the process continues. Examine the logic in Figure 2-8 to identify the components in the housekeeping, detail loop, and end-of-job tasks. You will learn more about the payroll report program in the next few chapters. For now, concentrate on the big picture of how a typical application works.

Figure 2-7

Sample Payroll Report

Figure 2-8

Logic for Payroll Report

Two Truths & A Lie

Modularizing a Program

1. A calling program calls a module’s name when it wants to use the module.

   T F

2. Whenever a main program calls a module, the logic transfers to the module; when the module ends, the program ends.

   T F

3. Housekeeping tasks include any steps you must perform just once at the beginning of a program to get ready for the rest of the program.

   T F

2-5 Creating Hierarchy Charts

You may have seen hierarchy charts for organizations, such as the one in Figure 2-9. The chart shows who reports to whom, not when or how often they report.

Figure 2-9

An Organizational Hierarchy Chart

When a program has several modules calling other modules, programmers often use a program **hierarchy chart** (A diagram that illustrates modules' relationships to each other.) that operates in a similar manner to show the overall picture of how modules are related to one another. A hierarchy chart does not tell you what tasks are to be performed *within* a module, *when* the modules are called, *how* a module executes, or *why* they are called—that information is in the flowchart or pseudocode. A hierarchy chart tells you only *which* modules exist within a program and *which* modules call others. The hierarchy chart for the program in Figure 2-8 looks like Figure 2-10. It shows that the main module calls three others—`housekeeping()`, `detailLoop()`, and `endOfJob()`.

Figure 2-10

Hierarchy Chart of Payroll Report Program in Figure 2-8

Figure 2-11 shows an example of a hierarchy chart for the billing program of a mail-order company. The hierarchy chart is for a more complicated program, but like the payroll report chart in Figure 2-10, it supplies module names and a general overview of the tasks to be performed, without specifying any details.

Figure 2-11

Billing Program Hierarchy Chart

Because program modules are reusable, a specific module may be called from several locations within a program. For example, in the billing program hierarchy chart in Figure 2-11, you can see that the `printCustomerData()` module is used twice. By convention, you blacken a corner of each box that represents a module used more than once. This action alerts readers that any change to this module could have consequences in multiple locations.

A hierarchy chart can be both a planning tool for developing the overall relationship of program modules before you write them and a documentation tool to help others see how modules are related after a program is written. For example, if a tax law changes, a programmer might be asked to rewrite the `calculateTax()` module in the billing program diagrammed in Figure 2-11. As the programmer changes the `calculateTax()` module, the hierarchy chart shows other dependent modules that might be affected. A hierarchy chart is useful for getting the big picture in a complex program.

Hierarchy charts are used in procedural programming, but other types of diagrams frequently are used in object-oriented environments. Chapter 13 of the comprehensive edition of this book describes the Unified Modeling Language, which is a set of diagrams you use to describe a system.

Two Truths & A Lie

Creating Hierarchy Charts

1. You can use a hierarchy chart to illustrate modules' relationships.

   T F

2. A hierarchy chart tells you what tasks are to be performed within a module.

   T F

3. A hierarchy chart tells you only which modules call other modules.

   T F

2-6 Features of Good Program Design

As your programs become larger and more complicated, the need for good planning and design increases. Think of an application you use, such as a word processor or a spreadsheet. The number and variety of user options are staggering. Not only would it be impossible for a single programmer to write such an application, but without thorough planning and design, the components would never work together properly. Ideally, each program module you design needs to work well as a stand-alone component and as an element of larger systems. Just as a house with poor plumbing or a car with bad brakes is fatally flawed, a computer-based application can be highly functional only if each component is designed well. Walking through your program's logic on paper (called deskchecking, as you learned in Chapter 1) is an important step to achieving superior programs. Additionally, you can implement several design features while creating programs that are easier to write and maintain. To create good programs, you should do the following:

- Use program comments where appropriate.
- Choose identifiers thoughtfully.
- 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.

2-6a Using Program Comments

When you write programs, you often might want to insert program comments. **Program comment** (A nonexecuting statement that programmers place within code to explain program statements in English. See also *internal program documentation*.) are written explanations that are not part of the program logic but that serve as documentation for readers of the program. In other words, they are nonexecuting statements that help readers understand programming statements. Readers might include users who help you test the program and other programmers who might have to modify your programs in the future. Even you, as the program's author, will appreciate comments when you make future modifications and forget why you constructed a statement in a certain way.

The syntax used to create program comments differs among programming languages. This book starts comments in pseudocode with two front slashes. For example, Figure 2-12 contains comments that explain the origins and purposes of variables in a real estate program.

Figure 2-12

Pseudocode that Declares Variables and Includes Comments

Program comments are a type of **internal documentation** (Documentation within a program. See also *program comments*.) . This term distinguishes them from supporting documents outside the program, which are called **external documentation** (All the external material that programmers develop to support a program; contrast with *program comments*, which are internal program documentation.) . Appendix D discusses other types of documentation.

In a flowchart, you can use an annotation symbol to hold information that expands on what is stored within another flowchart symbol. An **annotation symbol** (A flowchart symbol used to hold comments; it is most often represented by a three-sided box connected with a dashed line to the step it explains.) is most often represented by a three-sided box that is connected to the step it references by a dashed line. Annotation symbols are used to hold comments or sometimes statements that are too long to fit neatly into a flowchart symbol. For example, Figure 2-13 shows how a programmer might use some annotation symbols in a flowchart for a payroll program.

Figure 2-13

Flowchart that Includes Annotation Symbols

You probably will use comments in your coded programs more frequently

than you use them in pseudocode or flowcharts. For one thing, flowcharts and pseudocode are more English-like than the code in some languages, so your statements might be less cryptic. Also, your comments will remain in the program as part of the program documentation, but your planning tools are likely to be discarded once the program goes into production.

Including program comments is not necessary to create a working program, but comments can help you to remember the purpose of variables or to explain complicated calculations. Some students do not like to include comments in their programs because it takes time to type them and they aren't part of the "real" program, but the programs you write in the future probably will require some comments. When you acquire your first programming job and modify a program written by another programmer, you will appreciate well-placed comments that explain complicated sections of the code.

A drawback to comments is that they must be kept current as a program is modified. Outdated comments can provide misleading information about a program's status.

Chapter 2: Elements of High-Quality Programs: 2-6b Choosing Identifiers
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

2-6b Choosing Identifiers

The selection of good identifiers is an often-overlooked element in program design. When you write programs, you choose identifiers for variables, constants, and modules. You learned the rules for naming variables and modules earlier in this chapter: Each must be a single word with no embedded spaces and must start with a letter. Those simple rules provide a lot of leeway in naming program elements, but not all identifiers are equally good. Choosing good identifiers simplifies your programming job and makes it easier for others to understand your work.

Some general guidelines include the following:

- Although not required in any programming language, it usually makes sense to give a variable or constant a name that is a noun (or a combination of an adjective and noun) because it represents a thing. Similarly, it makes sense to give a module an identifier that is a verb, or a combined verb and noun, because a module takes action.

- Use meaningful names. Creating a data item named `someData` or a module named `firstModule()` makes a program cryptic. Not only will others find it hard to read your programs, but you will forget the purpose of these identifiers even within your own programs. All programmers occasionally use short, nondescriptive names such as `x` or `temp` in a quick program; however, in most cases, data and module names should be meaningful. Programmers refer to programs that contain meaningful names as **self-documenting** (Describes programs that contain meaningful and descriptive data, method, and class names.) . This means that even without further documentation, the program code explains itself to readers.

- Use pronounceable names. A variable name like `pzf` is neither pronounceable nor meaningful. A name that looks meaningful when you write it might not be as meaningful when someone else reads it; for instance, `preparead()` might mean "Prepare ad" to you, but "Prep a read" to others. Look at your names critically to make sure they can be pronounced. Very standard abbreviations do not have to be pronounceable. For example, most businesspeople would interpret `ssn` as a Social Security number.

- Don't forget that not all programmers share your culture. An abbreviation whose meaning seems obvious to you might be cryptic to someone in a different part of the world, or even a different part of your country. For example, you might name a variable `roi` to hold a value for *return on investment*, but a French-speaking person might interpret the meaning as *king*.

- Be judicious in your use of abbreviations. You can save a few keystrokes when creating a module called `getStat()`, but is the module's purpose to find the state in which a city is located, input some statistics, or determine the status of some variables? Similarly, is a variable named `fn` meant to hold a first name, file number, or something else? Abbreviations can also confuse people in different lines of work: AKA might suggest a sorority (Alpha Kappa Alpha) to a college administrator, a registry (American Kennel Association) to a dog breeder, or an alias (also known as) to a police detective.

To save typing time when you develop a program, you can use a short name like `efn`. After the program operates correctly, you can use a text editor's Search and Replace feature to replace your coded name with a more meaningful name such as `employeeFirstName`.

Many IDEs support an automatic statement-completion feature that saves typing time. After the first time you use a name like `employeeFirstName`, you need to type only the first few letters before the compiler editor offers a list of available names from which to choose. The list is constructed from all the names you have used that begin with the same characters.

- Usually, avoid digits in a name. Zeroes get confused with the letter *O*, and the lowercase letter *l* is misread as the numeral 1. Of course, use your judgment: `budgetFor2014` probably will not be misinterpreted.

- Use the system your language allows to separate words in long, multiword variable names. For example, if the programming language you use allows dashes or underscores, then use a module name like `initialize-data()` or `initialize_data()`, which is easier to read than `initializedata()`. Another option is to use camel casing to create an identifier such as `initializeData()`. If you use a language that is case sensitive, it is legal but confusing to use variable names that differ only in case. For example, if a single program contains `empName, EmpName,` and `Empname`, confusion is sure to follow.

- Consider including a form of the verb *to be*, such as *is* or *are*, in names for variables that are intended to hold a status. For example, use `isFinished` as a string variable that holds a *Y* or *N* to indicate whether a file is exhausted. The shorter name `finished` is more likely to be confused with a module that executes when a program is done. (Many languages support a Boolean data type, which you assign to variables meant to hold only true or false. Using a form of *to be* in identifiers for Boolean variables is appropriate.)

- Many programmers follow the convention of naming constants using all uppercase letters, inserting underscores between words for readability. In this chapter you saw examples such as `SALES_TAX_RATE`.

- Organizations sometimes enforce different rules for programmers to follow when naming program components. It is your responsibility to find out the conventions used in your organization and to adhere to them.

Programmers sometimes create a **data dictionary** (A list of every variable name used in a program, along with its type, size, and description.) , which is a list of every variable name used in a program, along with its type, size, and description. When a data dictionary is created, it becomes part of the program documentation.

When you begin to write programs, the process of determining what data variables, constants, and modules you need and what to name them all might seem overwhelming. The design process is crucial, however. When you acquire your first professional programming assignment, the design process might very well be completed already. Most likely, your first assignment will be to write or modify one small member module of a much larger application. The more the original programmers stuck to naming guidelines, the better the original design was, and the easier your job of modification will be.

2-6c Designing Clear Statements

In addition to using program comments and selecting good identifiers, you can use the following tactics to contribute to the clarity of the statements within your programs:

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

Avoiding Confusing Line Breaks

Some older programming languages require that program statements be placed in specific columns. Most modern programming languages are free-form; you can arrange your lines of code any way you see fit. As in real life, with freedom comes responsibility; when you have flexibility in arranging your lines of code, you must take care to make sure your meaning is clear. With free-form code, programmers are allowed to place two or three statements on a line, or, conversely, to spread a single statement across multiple lines. Both make programs harder to read. All the pseudocode examples in this book use appropriate, clear spacing and line breaks.

Using Temporary Variables to Clarify Long Statements

When you need several mathematical operations to determine a result, consider using a series of temporary variables to hold intermediate results. A **temporary variable** (A working variable that holds intermediate results during a program's execution.) (or a **work variable** (A working variable that holds intermediate results during a program's execution.)) is not used for input or output, but instead is just a working variable that you use during a program's execution. For example, Figure 2-14 shows two ways to calculate a value for a real estate `salespersonCommission` variable. Each module achieves the same result—the salesperson's commission is based on the square feet multiplied by the price per square foot, plus any premium for a lot with special features, such as a wooded or waterfront lot. However, the second example uses two temporary variables: `basePropertyPrice` and `totalSalePrice`. When the computation is broken down into less complicated, individual steps, it is easier to see how the total price is calculated. In calculations with even more computation steps, performing the arithmetic in stages would become increasingly helpful.

Figure 2-14

Two Ways of Achieving the Same `salespersoncommission` Result

Programmers might say using temporary variables, like the second example in Figure 2-14, is *cheap*. When executing a lengthy arithmetic statement, even if you don't explicitly name temporary variables, the programming language compiler creates them behind the scenes (although without descriptive names), so declaring them yourself does not cost much in terms of program execution time.

2-6d Writing Clear Prompts and Echoing Input

When program input should be retrieved from a user, you almost always want to provide a prompt for the user. A **prompt** (A message that is displayed on a monitor, asking the user for a response.) 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. Prompts are used both in command-line and GUI interactive programs.

For example, suppose a program asks a user to enter a catalog number for an item the user is ordering. The following prompt is not very helpful:

The following prompt is more helpful:

The following prompt is even more helpful:

When program input comes from a stored file instead of a user, prompts are not needed. However, when a program expects a user response, prompts are valuable. For example, Figure 2-15 shows the flowchart and pseudocode for the beginning of the bill-producing program shown earlier in this chapter. If the input was coming from a data file, no prompt would be required, and the logic might look like the logic in Figure 2-15.

Figure 2-15

Beginning of a Program that Accepts a Name and Balance as Input

However, if the input was coming from a user, including prompts would be helpful. You could supply a single prompt such as *Please enter a customer's name and balance due*, but inserting more requests into a prompt generally makes it less likely that the user can remember to enter all the parts or enter them in the correct order. It is almost always best to include a separate prompt for each item to be entered. Figure 2-16 shows an example.

Figure 2-16

Beginning of a Program that Accepts a Name and Balance as Input and Uses a Separate Prompt for Each Item

Users also find it helpful when you echo their input. **Echoing input** (The act of repeating input back to a user either in a subsequent prompt or in output.) is the act of repeating input back to a user either in a subsequent prompt or in output. For example, Figure 2-17 shows how the second prompt in Figure 2-16 can be improved by echoing the user's first piece of input data in the second prompt. When a user runs the program that is started in Figure 2-17 and enters *Green* for the customer name, the second prompt will not be *Please enter balance due*. Instead, it will be *Please enter balance due for Green*. For example, if a clerk was about to enter the balance for the wrong customer, the mention of *Green* might be enough to alert the clerk to the potential error.

Figure 2-17

Beginning of a Program that Accepts a Customer's Name and Uses it in the Second Prompt

Notice the space before the quotation mark in the prompt that asks the user for a balance due. The space will appear between *for* and the last name.

2-6e Maintaining Good Programming Habits

When you learn a programming language and begin to write lines of program code, it is easy to forget the principles you have learned in this text. Having some programming knowledge and a keyboard at your fingertips can lure you into typing lines of code before you think things through. But every program you write will be better if you plan before you code. Maintaining the habits of first drawing flowcharts or writing pseudocode, as you have learned here, will make your future programming projects go more smoothly. If you desk-check your program logic on paper before coding statements in a programming language, your programs will run correctly sooner. If you think carefully about the variable and module names you choose, and design program statements to be easy to read and use, your programs will be easier to develop and maintain.

Two Truths & A Lie

Features of Good Program Design

1. A program comment 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.

   T F

2. It usually makes sense to give each variable a name that contains a noun and to give each module a name that contains a verb.

   T F

3. Echoing input can help a user to confirm that a data item was entered correctly.

   T F

2-7 Chapter Review

2-7a Chapter Summary

- Programs contain data in three different forms: literals (or unnamed constants), variables, and named constants. Each of these types of data can be numeric or string. Variables are named memory locations, the contents of which can vary. A variable declaration includes a data type and an identifier; optionally, it can include an initialization. Every computer programming language has its own set of rules for naming variables; however, all variable names must be written as one word without embedded spaces and should have appropriate meaning. A named constant is similar to a variable, except it can be assigned a value only once.

- Most programming languages use +, –, *, and / as the four standard arithmetic operators. Every operator follows rules of precedence that dictate the order in which operations in the same statement are carried out; multiplication and division always take precedence over addition and subtraction. The rules of precedence can be overridden using parentheses.

- Programmers break down programming problems into smaller, cohesive units called modules, subroutines, procedures, functions, or methods. To execute a module, you call it from another program or module. Any program can contain an unlimited number of modules, and each module can be called an unlimited number of times. Modularization provides abstraction, allows multiple programmers to work on a problem, and makes it easier for you to reuse work.

- When you create a module, you include a header, a body, and a `return` statement. A program or module calls a module's name to execute it. You can place any statements within modules, including declarations, which are local to the module. Global variables and constants are those that are known to the entire program. The mainline logic of almost every procedural computer program can follow a general structure that consists of three distinct parts: housekeeping tasks, detail loop tasks, and end-of-job tasks.

- A hierarchy chart illustrates modules and their relationships; it indicates which modules exist within a program and which modules call others.

- As programs become larger and more complicated, the need for good planning and design increases. You should use program comments where appropriate. Choose identifiers wisely, strive to design clear statements within your programs and modules, write clear prompts and echo input, and continue to maintain good programming habits as you develop your programming skills.

Chapter Review

2-7b Key Terms

numeric (Numeric describes data that consists of numbers.)

string (Describes data that is nonnumeric.)

integer (A whole number.)

floating-point (A fractional, numeric variable that contains a decimal point.)

real numbers (Floating-point numbers.)

numeric constant (A specific numeric value.)

literal numeric constant (A specific numeric value.)

string constant (A specific group of characters enclosed within quotation marks.)

literal string constant (A specific group of characters enclosed within quotation marks.)

alphanumeric values (The set of values that include alphabetic characters, numbers, and punctuation.)

unnamed constant (A literal numeric or string value.)

declaration (A statement that names a variable and its data type.)

identifier (A program component's name.)

data type (The characteristic of a variable that describes the kind of values the variable can hold and the types of operations that can be performed with it.)

initializing a variable (The act of assigning the first value to a variable, often at the same time the variable is created.)

garbage (Describes the unknown value stored in an unassigned variable.)

keywords (The limited word set that is reserved in a language.)

Camel casing (A naming convention in which the initial letter is lowercase, multiple-word names are run together, and each new word within the name begins with an uppercase letter.)

Pascal casing (A naming convention in which the initial letter is uppercase, multiple-word names are run together, and each new word within the name begins with an uppercase letter.)

Hungarian notation (A variable-naming convention in which a variable's data type or other information is stored as part of its name.)

assignment statement (A statement that stores the result of any value on its right side to the named location on its left side.)

assignment operator (The equal sign; it always requires the name of a memory location on its left side.)

binary operator (An operator that requires two operands—one on each side.)

right-associativity (Descriptions of operators that evaluate the expression to the right first.)

right-to-left associativity (Descriptions of operators that evaluate the expression to the right first.)

lvalue (The memory address identifier to the left of an assignment operator.)

numeric variable (A variable that holds numeric values.)

string variable (A variable that can hold text that includes letters, digits, and special characters such as punctuation marks.)

type-safety (The feature of programming languages that prevents assigning values of an incorrect data type.)

named constant (A named memory location, similar to a variable, except its value never changes during the execution of a program. Conventionally, constants are named using all capital letters.)

magic number (An unnamed numeric constant.)

overhead (All the resources and time required by an operation.)

rules of precedence (rules of precedence dictate the order in which operations in the same statement are carried out.)

order of operations (Describes the rules of precedence.)

left-to-right associativity (Describes operators that evaluate the expression to the left first.)

modules (A small program unit used with other modules to make a program. Programmers also refer to modules as subroutines, procedures, functions, and methods.)

subroutines (A small program unit used with other modules to make a program. Programmers also refer to modules as subroutines, procedures, functions, and methods.)

procedures (A small program unit used with other modules to make a program. Programmers also refer to modules as subroutines, procedures, functions, and methods.)

functions (A small program unit used with other modules to make a program. Programmers also refer to modules as subroutines, procedures, functions, and methods.)

methods (A small program unit used with other modules to make a program. Programmers also refer to modules as subroutines, procedures, functions, and methods.)

call a module (To use a module's name to invoke it, causing it to execute.)

modularization (The process of breaking down a program into modules.)

Functional decomposition (Functional decomposition is the act of reducing a large program into more manageable modules.)

Abstraction (The process of paying attention to important properties while ignoring nonessential details.)

reusability (The feature of modular programs that allows individual modules to be used in a variety of applications.)

Reliability (The feature of modular programs that ensures a module has been tested and proven to function correctly.)

main program (A program that runs from start to stop and calls other modules; also called a *main program method*.)

mainline logic (The overall logic of the main program from beginning to end.)

module header (The module header includes the module identifier and possibly other necessary identifying information.)

module body (The module body contains all the statements in the module.)

module return statement (The module 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.)

encapsulated (encapsulated is the act of containing a task's instructions in a module.)

stack (A memory location that holds the memory addresses to which method calls should return.)

functional cohesion (The extent to which all operations in a method contribute to the performance of only one task.)

visible (A characteristic of data items that means they "can be seen" only within the method in which they are declared.)

in scope (The characteristic of variables and constants declared within a method that apply only within that method.)

local (Describes variables that are declared within the method that uses them.)

portable (Describes a module that can more easily be reused in multiple programs.)

Global (Describes variables that are known to an entire program.)

program level (The level at which global variables are declared.)

Housekeeping tasks (Tasks that must be performed at the beginning of a program to prepare for the rest of the program.)

Detail loop tasks (The steps that are repeated for each set of input data.)

End-of-job tasks (End-of-job tasks hold the steps you take at the end of the program to finish the application.)

hierarchy chart (A diagram that illustrates modules' relationships to each other.)

Program comment (A nonexecuting statement that programmers place within code to explain program statements in English. See also *internal program documentation*.)

internal documentation (Documentation within a program. See also *program comments*.)

external documentation (All the external material that programmers develop to support a program; contrast with *program comments*, which are internal program documentation.)

annotation symbol (A flowchart symbol used to hold comments; it is most often represented by a three-sided box connected with a dashed line to the step it explains.)

self-documenting (Describes programs that contain meaningful and descriptive data, method, and class names.)

data dictionary (A list of every variable name used in a program, along with its type, size, and description.)

temporary variable (A working variable that holds intermediate results during a program's execution.)

work variable (A working variable that holds intermediate results during a program's execution.)

prompt (A message that is displayed on a monitor, asking the user for a response.)

Echoing input (The act of repeating input back to a user either in a subsequent prompt or in output.)

Chapter Review

2-7c Review Questions

1. What does a declaration provide for a variable?
    a. a name
    b. a data type
    c. both of the above
    d. none of the above
2. A variable's data type describes all of the following *except* ____ .
    a. what values the variable can hold
    b. how the variable is stored in memory
    c. what operations can be performed with the variable
    d. the scope of the variable
3. The value stored in an uninitialized variable is ____ .
    a. garbage
    b. null
    c. compost
    d. its identifier
4. The value 3 is a ____ .
    a. numeric variable
    b. numeric constant
    c. string variable

    d. string constant

5. The assignment operator ____ .

    a. is a binary operator

    b. has left-to-right associativity

    c. is most often represented by a colon

    d. two of the above

6. Which of the following is true about arithmetic precedence?

    a. Multiplication has a higher precedence than division.

    b. Operators with the lowest precedence always have left-to-right associativity.

    c. Division has higher precedence than subtraction.

    d. all of the above

7. Which of the following is a term used as a synonym for *module* in some programming languages?

    a. method

    b. procedure

    c. both of these

    d. none of these

8. Which of the following is a reason to use modularization?

    a. Modularization avoids abstraction.

    b. Modularization reduces overhead.

    c. Modularization allows you to more easily reuse your work.

    d. Modularization eliminates the need for syntax.

9. What is the name for the process of paying attention to important properties while ignoring nonessential details?

    a. abstraction

   b. extraction
   c. extinction
   d. modularization

10. Every module has all of the following *except* ____ .
    a. a header
    b. local variables
    c. a body
    d. a `return` statement
11. Programmers say that one module ____ can another, meaning that the first module causes the second module to execute.
    a. declare
    b. define
    c. enact
    d. call
12. The more that a module's statements contribute to the same job, the greater the ____ of the module.
    a. structure
    b. modularity
    c. functional cohesion
    d. size
13. In most modern programming languages, a variable or constant that is declared in a module is ____ in that module.
    a. global
    b. invisible
    c. in scope
    d. undefined

14. Which of the following is *not* a typical housekeeping task?

    a. displaying instructions

    b. printing summaries

    c. opening files

    d. displaying report headings

15. Which module in a typical program will execute the most times?

    a. the housekeeping module

    b. the detail loop

    c. the end-of-job module

    d. It is different in every program.

16. A hierarchy chart tells you _____ .

    a. what tasks are to be performed within each program module

    b. when a module executes

    c. which routines call which other routines

    d. all of the above

17. What are nonexecuting statements that programmers place within code to explain program statements in English?

    a. comments

    b. pseudocode

    c. trivia

    d. user documentation

18. Program comments are _____ .

    a. required to create a runnable program

    b. a form of external documentation

    c. both of the above

   d. none of the above

19. Which of the following is valid advice for naming variables?

    a. To save typing, make most variable names one or two letters.
    b. To avoid conflict with names that others are using, use unusual or unpronounceable names.
    c. To make names easier to read, separate long names by using underscores or capitalization for each new word.
    d. To maintain your independence, shun the conventions of your organization.

20. A message that asks a user for input is a ____ .

    a. comment
    b. prompt
    c. echo
    d. declaration

Chapter Review

2-7d Exercises

1. Explain why each of the following names does or does not seem like a good variable name to you.

    a. `d`
    b. `dsctamt`
    c. `discountAmount`
    d. `discount Amount`

e. `discount`

f. `discountAmountForEachNewCustomer`

g. `discountYear2013`

h. `2013Discountyear`

2. If `productCost` and `productPrice` are numeric variables, and `productName` is a string variable, which of the following statements are valid assignments? If a statement is not valid, explain why not.

    a. `productCost = 100`

    b. `productPrice = productCost`

    c. `productPrice = productName`

    d. `productPrice = "24.95"`

    e. `15.67 = productCost`

    f. `productCost = $1,345.52`

    g. `productCost = productPrice - 10`

    h. `productName = "mouse pad"`

    i. `productCost + 20 = productPrice`

    j. `productName = 3-inch nails`

    k. `productName = 43`

    l. `productName = "44"`

    m. `"99" = productName`

    n. `productName = brush`

    o. `battery = productName`

    p. `productPrice = productPrice`

    q. `productName = productCost`

3. Assume that `income = 8` and `expense = 6`. What is the value of each of the following expressions?

a. `income + expense * 2`

b. `income + 4 – expense / 2`

c. `(income + expense) * 2`

d. `income – 3 * 2 + expense`

e. `4 * ((income – expense) + 2) + 10`

4. Draw a typical hierarchy chart for a program that produces a monthly bill for a cell phone customer. Try to think of at least 10 separate modules that might be included. For example, one module might calculate the charge for daytime phone minutes used.

5. a. Draw the hierarchy chart and then plan the logic for a program needed by the sales manager of The Henry Used Car Dealership. The program will determine the profit on any car sold. Input includes the sale price and actual purchase price for a car. The output is the profit, which is the sale price minus the purchase price. Use three modules. The main program declares global variables and calls housekeeping, detail, and end-of-job modules. The housekeeping module prompts for and accepts a sale price. The detail module prompts for and accepts the purchase price, computes the profit, and displays the result. The end-of-job module displays the message *Thanks for using this program*.

   b. Revise the profit-determining program so that it runs continuously for any number of cars. The detail loop executes continuously while the sale price is not 0; in addition to calculating the profit, it prompts the user for and gets the next sale price. The end-of-job module executes after 0 is entered for the sale price.

6. a. Draw the hierarchy chart and then plan the logic for a program that calculates a person’s body mass index (BMI). BMI is a statistical measure that compares a person’s weight and height. The program uses three modules. The first prompts a user for and accepts the user’s height in inches. The second module accepts the user’s weight in pounds and converts the user’s height to meters and weight to kilograms. Then, it calculates BMI as weight in kilograms times height in meters squared, and displays the results. There are 2.54 centimeters in an inch, 100 centimeters in a meter, 453.59 grams in a pound, and 1,000 grams in a kilogram. Use named constants whenever you think they are appropriate. The last module displays the message *End of job*.

    b. Revise the BMI-determining program to execute continuously until the user enters 0 for the height in inches.

7. Draw the hierarchy chart and design the logic for a program that calculates service charges for Hazel's Housecleaning service. The program contains housekeeping, detail loop, and end-of-job modules. The main program declares any needed global variables and constants and calls the other modules. The housekeeping module displays a prompt for and accepts a customer's last name. While the user does not enter *ZZZZ* for the name, the detail loop accepts the number of bathrooms and the number of other rooms to be cleaned. The service charge is computed as $40 plus $15 for each bathroom and $10 for each of the other rooms. The detail loop also displays the service charge and then prompts the user for the next customer's name. The end-of-job module, which executes after the user enters the sentinel value for the name, displays a message that indicates the program is complete.

8. Draw the hierarchy chart and design the logic for a program that calculates the projected cost of an automobile trip. Assume that the user's car travels 20 miles per gallon of gas. Design a program that prompts the user for a number of miles driven and a current cost per gallon. The program computes and displays the cost of the trip as well as the cost if gas prices rise by 10 percent. The program accepts data continuously until 0 is entered for the number of miles. Use appropriate modules, including one that displays *End of program* when the program is finished.

9.
    a. Draw the hierarchy chart and design the logic for a program needed by the manager of the Stengel County softball team, who wants to compute slugging percentages for his players. A slugging percentage is the total bases earned with base hits divided by the player's number of at-bats. Design a program that prompts the user for a player jersey number, the number of bases earned, and the number of at-bats, and then displays all the data, including the calculated slugging average. The program accepts players continuously until 0 is entered for the jersey number. Use appropriate modules, including one that displays *End of job* after the sentinel is entered for the jersey number.

    b. Modify the slugging percentage program to also calculate a player's on-base percentage. An on-base percentage is calculated by adding a player's hits and walks, and then dividing by the sum of at-bats, walks, and sacrifice flies. Prompt the user for all the additional data needed, and display all the data for each player.

c. Modify the softball program so that it also computes a gross production average (GPA) for each player. A GPA is calculated by multiplying a player's on-base percentage by 1.8, then adding the player's slugging percentage, and then dividing by four.

Find the Bugs

10. Your downloadable files for Chapter 2 include DEBUG02-01.txt, DEBUG02-02.txt, and DEBUG02-03.txt. Each file starts with some comments that describe the problem. Comments are lines that begin with two slashes (//). Following the comments, each file contains pseudocode that has one or more bugs you must find and correct.

Game Zone

11. For games to hold your interest, they almost always include some random, unpredictable behavior. For example, a game in which you shoot asteroids loses some of its fun if the asteroids follow the same, predictable path each time you play. Therefore, generating random values is a key component in creating most interesting computer games. Many programming languages come with a built-in module you can use to generate random numbers. The syntax varies in each language, but it is usually something like the following:

    In this statement, `myRandomNumber` is a numeric variable you have declared and the expression `random(10)` means "call a method that generates and returns a random number between 1 and 10." By convention, in a flowchart, you would place a statement like this in a processing symbol with two vertical stripes at the edges, as shown below.

    Create a flowchart or pseudocode that shows the logic for a program that generates a random number, then asks the user to think of a number between 1 and 10. Then display the randomly generated number so the user can see whether his or her guess was accurate. (In future chapters, you will improve this game so that the user can enter a guess and the program can determine whether the user was correct.)

Create a flowchart or pseudocode that shows the logic for a program that generates a random number, then asks the user to think of a number between 1 and 10. Then display the randomly generated number so the user can see whether his or her guess was accurate. (In future chapters, you will improve this game so that the user can enter a guess and the program can determine whether the user was correct.)

Up for Discussion

12. Many programming style guides are published on the Web. These guides suggest good identifiers, explain standard indentation rules, and identify style issues in specific programming languages. Find style guides for at least two languages (for example, C++, Java, Visual Basic, or C#) and list any differences you notice.
13. What advantages are there to requiring variables to have a data type?
14. As this chapter mentions, some programming languages require that named constants are assigned a value when they are declared; other languages allow a constant's value to be assigned later in a program. Which requirement do you think is better? Why?
15. Would you prefer to write a large program by yourself, or to work on a team in which each programmer produces one or more modules? Why?
16. Extreme programming is a system for rapidly developing software. One of its tenets is that all production code is written by two programmers sitting at one machine. Is this a good idea? Does working this way as a programmer appeal to you? Why or why not?