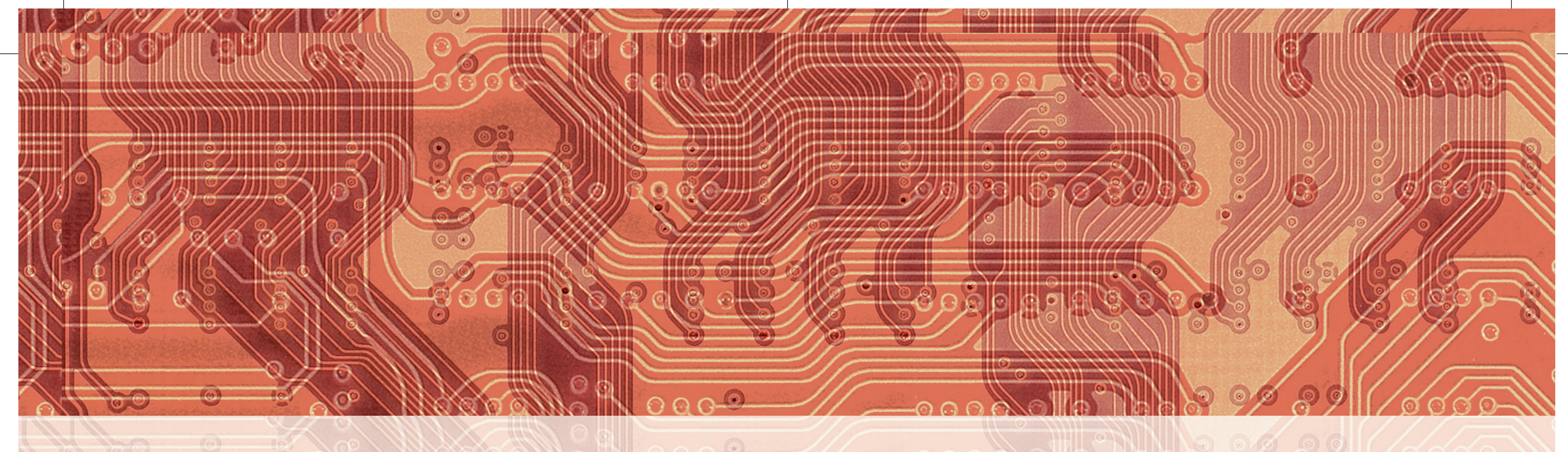


Programming in Ada

Online module to accompany *Invitation to Computer Science, 7th Edition* ISBN-10: 1305075773; ISBN-13: 9781305075771 (Cengage Learning, 2016).

1. Introduction to Ada
- **1.1** A Simple Ada Program
- **1.2** Creating and Running an Ada Program

2. Virtual Data Storage

3. Statement Types
- **3.1** Input/Output Statements
- **3.2** The Assignment Statement
- **3.3** Control Statements

4. Another Example

5. Managing Complexity
- **5.1** Divide and Conquer
- **5.2** Using Functions/Procedures
- **5.3** Writing Functions/Procedures
- **5.4** An Ada Feature: User-Defined Subtypes

6. Object-Oriented Programming
- **6.1** What Is It?
- **6.2** Ada and OOP
- **6.3** One More Example
- **6.4** What Have We Gained?

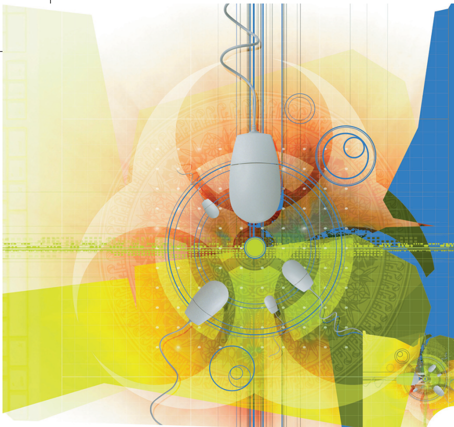
7. Graphical Programming
- **7.1** Graphics Hardware
- **7.2** Graphics Software

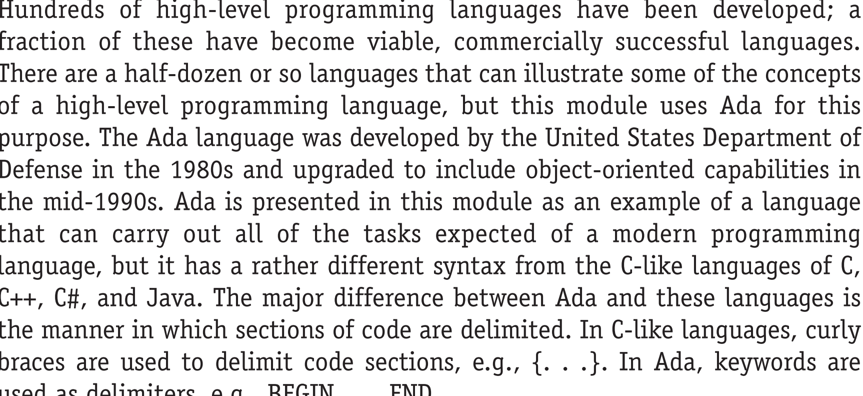
8. Conclusion

EXERCISES

ANSWERS TO PRACTICE PROBLEMS

©2016 Course Technology, a part of Cengage Learning.





1 Introduction to Ada

Hundreds of high-level programming languages have been developed; a fraction of these have become viable, commercially successful languages. There are a half-dozen or so languages that can illustrate some of the concepts of a high-level programming language, but this module uses Ada for this purpose. The Ada language was developed by the United States Department of Defense in the 1980s and upgraded to include object-oriented capabilities in the mid-1990s. Ada is presented in this module as an example of a language that can carry out all of the tasks expected of a modern programming language, but it has a rather different syntax from the C-like languages of C, C++, C#, and Java. The major difference between Ada and these languages is the manner in which sections of code are delimited. In C-like languages, curly braces are used to delimit code sections, e.g., {. . .}. In Ada, keywords are used as delimiters, e.g., BEGIN . . . END.

Our intent here is not to make you an expert programmer—any more than our purpose in Chapter 4 was to make you an expert circuit designer. Indeed, there is much about the language that we will not even discuss. You will, however, get a sense of what programming in a high-level language is like, and perhaps you will see why some people think it is one of the most fascinating of human endeavors.

1.1 *A Simple Ada Program*

Figure 1 shows a simple but complete Ada program. Even if you know nothing about the Ada language, it is not hard to get the general drift of what the program is doing.

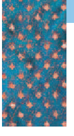


FIGURE 1

A Simple Ada Program

```
-- Computes and outputs travel time
-- for a given speed and distance
-- Written by J. Q. Programmer, 6/15/16

WITH TEXT_IO;

PROCEDURE TravelPlanner IS
  PACKAGE INT_IO IS NEW TEXT_IO.INTEGER_IO(INTEGER);
  PACKAGE FLO_IO IS NEW TEXT_IO.FLOAT_IO(FLOAT);
```

©2016 Course Technology, a part of Cengage Learning.

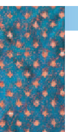


FIGURE 1
A Simple Ada Program
(continued)

```
  speed : INTEGER;       -- rate of travel
  distance : FLOAT;      -- miles to travel
  time : FLOAT;          -- time needed for this travel

BEGIN

  TEXT_IO.PUT("Enter your speed in mph: ");
  INT_IO.GET(speed);
  TEXT_IO.PUT("Enter your distance in miles: ");
  FLO_IO.GET(distance);

  time := distance / FLOAT(speed);

  TEXT_IO.PUT("At ");
  INT_IO.PUT(speed);
  TEXT_IO.PUT(" mph, ");
  TEXT_IO.PUT("it will take ");
  TEXT_IO.NEW_LINE;
  FLO_IO.PUT(time);
  TEXT_IO.PUT(" hours to travel ");
  FLO_IO.PUT(distance);
  TEXT_IO.PUT(" miles.");
  TEXT_IO.NEW_LINE;
END TravelPlanner;
```

Someone running this program (the "user") could have the following dialogue with the program, where boldface indicates what the user types:

```
Enter your speed in mph: 58
Enter your distance in miles: 657.5
At           58 mph, it will take
 1.13362E+01 hours to travel  6.57500E+02 miles.
```

The general form of a typical Ada program is shown in Figure 2. To compare our simple example program with this form, we have reproduced the example program in Figure 3 with a number in front of each line. The numbers are there for reference purposes only; they are *not* part of the program.

Lines 1–3 in the program of Figure 3 are Ada **comments**. Anything appearing on a line after the double dash (--) is ignored by the compiler; it is treated as a comment in the assembly language programs of Chapter 6.

FIGURE 2
The Overall Form of a Typical Ada Package Body Program

```
prologue comment         [optional]
with clauses
procedure name
   functions/procedures  [optional]
   declaratives
begin
  code
end name
```


©2016 Course Technology, a part of Cengage Learning.

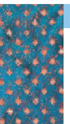


FIGURE 3

The Program of Figure 1 (line numbers added for reference)

```
1.  -- Computes and outputs travel time
2.  -- for a given speed and distance
3.  -- Written by J. Q. Programmer, 6/15/16
4.
5.  WITH TEXT_IO;
    -- USE TEXT_IO;
6.
7.  PROCEDURE TravelPlanner IS
8.    PACKAGE INT_IO IS NEW TEXT_IO.INTEGER_IO(INTEGER);
9.    PACKAGE FLO_IO IS NEW TEXT_IO.FLOAT_IO(FLOAT);
10.
11.   speed : INTEGER;        -- rate of travel
12.   distance : FLOAT;       -- miles to travel
13.   time : FLOAT;           -- time needed for this travel
14.
15. BEGIN
16.
17.   TEXT_IO.PUT("Enter your speed in mph: ");
18.   INT_IO.GET(speed);
19.   TEXT_IO.PUT("Enter your distance in miles: ");
20.   FLO_IO.GET(distance);
21.
22.   time := distance / FLOAT(speed);
23.
24.   TEXT_IO.PUT("At ");
25.   INT_IO.PUT(speed);
26.   TEXT_IO.PUT(" mph, ");
27.   TEXT_IO.PUT("it will take ");
28.   TEXT_IO.NEW_LINE;
29.   FLO_IO.PUT(time);
30.   TEXT_IO.PUT(" hours to travel ");
31.   FLO_IO.PUT(distance);
32.   TEXT_IO.PUT(" miles.");
33.   TEXT_IO.NEW_LINE;
34. END TravelPlanner;
```

Although the computer ignores comments, they are important to include in a program because they give information to the human readers of the code. Every high-level language has some facility for including comments, because understanding code that someone else has written (or understanding your own code after a period of time has passed) is very difficult without the notes and explanations that comments provide. Comments are one way to *document* a computer program to make it more understandable. The comments in the program of Figure 3 describe what the program does plus tell who wrote the program and when. These three comment lines together make up the program's **prologue comment** (the introductory comment that comes first). According to the general form of Figure 2, the prologue comment is optional, but providing it is always a good idea. It's almost like the headline in a newspaper, giving the big picture up front.

©2016 Course Technology, a part of Cengage Learning.

History of Ada

Ada is probably the most systematically developed programming language ever. In the mid-1970s, the United States Department of Defense (DoD) set about trying to solve the problems created by using hundreds of different programming languages for defense system components. Integration was difficult, and reliability was low. Building on the work begun by the Army, Navy, and Air Force, a working group laid out the first informal requirements for a common programming language. This set of requirements was known as Strawman. More complete and stringent requirements followed, known successively as Woodenman (1975) and Tinman (1976). The working group evaluated twenty-three existing programming languages against the Tinman requirements. As none was found satisfactory, it was decided to develop a new programming language, and in 1977 the working group issued requests for proposals to be evaluated against the latest specifications, known as Ironman. Four designs were evaluated in 1978, and two of these were selected to compete against the final set of specifications, named Steelman. The winning language was submitted by Cii-Honeywell Bull, led by the Frenchman Dr. Jean Ichbiah. For his role in developing this new language, he was later awarded membership in the Legion of Honor by the President of France.

The name Ada was chosen, of course, in honor of Lady Ada Augusta Byron Lovelace, who worked with Charles Babbage in the 1800s to help "program" his Analytic Engine (see Chapter 1). The Military Standard reference manual for Ada was approved in 1980 on Ada Lovelace's birthday, December 10.

Between 1987 and 1997, the DoD required the use of Ada for projects with significant new code. Although this standard is no longer in place, Ada is still used to develop highly reliable software.

Blank lines in Ada programs are ignored and are used, like comments, to make the program more readable by human beings. In our example program, we've used blank lines (lines 4, 6, 10, 14, 16, 21, 23) to separate sections of the program, visually indicating groups of statements that are related.

Before looking at the details of the remaining code line by line, it is important to know that Ada programs are constructed from a collection of packages. Each package consists of two sets of code, an optional specification, and a body. The code shown in Figures 1 and 3 has only a body, but it is still part of a package. When referencing one package from within another package—for example, the TEXT_IO package from within the package being written (the *TravelPlanner* program)—it is necessary to first identify that package. This is accomplished by adding a *with* clause to the package being written (line 5). The core Ada language does not provide a way to get data into a program or for a program to display results. The TEXT_IO package contains code for these purposes. Line 5 tells the compiler to look in the TEXT_IO package for the definition of any names not specifically defined within the program. In this program, *GET*, *PUT*, and *NEW_LINE* (used to read input data from the keyboard, write output to the screen, and start a new output line, respectively) obtain their meaning from the TEXT_IO package. One way to reference these code segments is to prefix the name with the name of its package, e.g., line 17:

```
TEXT_IO.PUT("Enter your speed in mph: ");
```

This is the method we will use in this module. Another alternative is to add a *use* clause to the code (see the unnumbered line below line 5 in Figure 3 that is commented out). If this line is in the code, then line 17 can be written as:

©2016 Course Technology, a part of Cengage Learning.

```
PUT("Enter your speed in mph: ");
```

without the qualification prefix TEXT_IO.

The eventual effect of the *with* clause is that the linker includes object code from this package. In addition to TEXT_IO, Ada has many other code packages, such as mathematical and graphics packages, and therefore many other *with* clauses are possible. *With* clauses are optional, but it would be a trivial program indeed that did not need input data or produce output results, so virtually every Ada program has at least the *with* clause shown in our example.

Names in Ada are *not* case sensitive and can be written in uppercase, lowercase, or mixed case. The code in this module is written in the style of the *Reference Manual for the Ada Programming Language.*[1] This style can be recognized by noting that many items are typed in all uppercase letters, and that the underscore character is often used as a "word separator," e.g., TEXT_IO.

Now back to the line-by-line code analysis. Line 7 begins the body part of the Ada program. The package body begins with a PROCEDURE statement, which includes a name—in this case, *TravelPlanner*. The body code concludes with an END statement, line 34, which includes the same name, *TravelPlanner*.

Lines 8 through 13 constitute the declarative portion of the package. Lines 8 and 9 are a special form of declaration. Ada is a **strongly-typed language**, which means the compiler will not allow you to mix up integers (numbers with no decimal point), floating-point numbers (numbers with decimal points), and strings (such as "abc") in the same statement. A ramification of that requirement is that each type of data must have its own separate mechanism for input and output. Lines 8 and 9 deal with that problem for the TravelPlanner program. There are several "kinds" (think "sizes") of integers and several "kinds" of floating-point numbers. Line 8 makes a special package named INT_IO for input/output of integers as used in the TravelPlanner program, and line 9 makes a package named FLO_IO for floating-point numbers as used in the TravelPlanner program. Because there is only one kind of string, a special package is not needed. Any I/O operation involving integers must be prefixed by INT_IO, any operation involving floats must be prefixed with FLO_IO, and any I/O operation involving strings must be prefixed by TEXT_IO, as in lines 17–20 and 24–33.

Lines 11 through 13 are statements that declare the names and data types for the quantities to be used in the program. Descriptive names—*speed*, *distance*, and *time*—are used for these quantities to help document their purpose in the program, and comments provide further clarification. Note that the data type designation (INTEGER, FLOAT) appears after the name, as opposed to C-like languages, where the data type designation comes first.

After all this setup, the executable portion of the package body lies between the BEGIN at line 15 and the END statement at line 34. For want of a

[1]*Reference Manual for the Ada Programming Language* (ANSI/MIL-STD-1815A-1983), Springer-Verlag, New York, 1983, ISBN:0-387-90887-0. Ada is a registered trademark of the U.S. Government.

©2016 Course Technology, a part of Cengage Learning.

better term, we'll call this portion the "main program code," even though the entire Ada program is "code." Line 17 outputs a string (TEXT_IO.PUT(. . .)) as a prompt to the user to enter a value. Line 18 (INT_IO.GET(. . .)) gathers the integer input for *speed* that is typed in by the user on the keyboard. Lines 19 and 20 do a similar job for the floating-point value of *distance*.

Line 22 is a replacement statement used to compute the value for *time*. Two important features of this statement are that the replacement operator is := (as opposed to = in many languages), and that strong typing requires that the integer value of speed be converted to type FLOAT before the division operation will be allowed to take place.

Lines 24 through 33 create the output display on the console screen. Lines 28 and 33 create the line breaks in the output.

Line 34 signals the end of the source code for the package.

Each line of code within the structural markers (PROCEDURE, BEGIN, END) must end with a semicolon. The semicolon requirement is a bit of a pain in the neck, but the Ada compiler generates one or more error messages if you omit the semicolon, so after the first few hundred times this happens, you tend to remember to put it in.

Ada, along with every other programming language, has very specific rules of **syntax**—the correct form for each component of the language. Having a semicolon at the end of every executable statement is an Ada syntax rule. Any violation of the syntax rules generates an error message from the compiler because the compiler does not recognize or know how to translate the offending code. In the case of a missing semicolon, the compiler cannot tell where the instruction ends. The syntax rules for a programming language are often defined by a formal grammar, much as correct English syntax is defined by rules of grammar.

Ada is a **free-format language**, which means that it does not matter where things are placed on a line. For example, we could have written

```
          time      :=
    distance                 /
                                       FLOAT(speed);
```

although this is clearly harder to read. The free-format characteristic explains why a semicolon is needed to mark the end of an instruction, which might be spread over several lines.

1.2 *Creating and Running an Ada Program*

Creating and running an Ada program is basically a three-step process. The first step is to type the program into a text editor. When you are finished, you save the file, giving it a name with the extension *.adb*. The extension "adb" (shorthand for Ada body) is used since this is a package body. Specification files would have an extension *.ads*. So the file for Figure 1 could be named

TravelPlanner.adb

As the second step, the program in the *.adb* file must be prepared for execution. This step has three substeps: compilation (turn the source file into an

©2016 Course Technology, a part of Cengage Learning.

object file), binding (associate address values with symbolic names), and linking (connect the object code just created with any other object code needed), resulting in an executable file. In our example, the result is a file called

TravelPlanner.exe

The third step operates on the *.exe* file and loads and executes the program. Depending on your system, you may have to type operating system commands for the last two steps.

Another approach is to do all of your work in an **Integrated Development Environment**, or **IDE**. The IDE lets the programmer perform a number of tasks within the shell of a single application program, rather than having to use a separate program for each task. A modern programming IDE provides a text editor, a file manager, a compiler, a linker and loader, and tools for debugging, all within this one piece of software. The IDE usually has a GUI interface with menu choices for the different tasks. This can significantly speed up program development.

This Ada exercise is just a beginning. In the rest of this module, we'll examine the features of the language that will enable you to write your own Ada programs to carry out more sophisticated tasks.

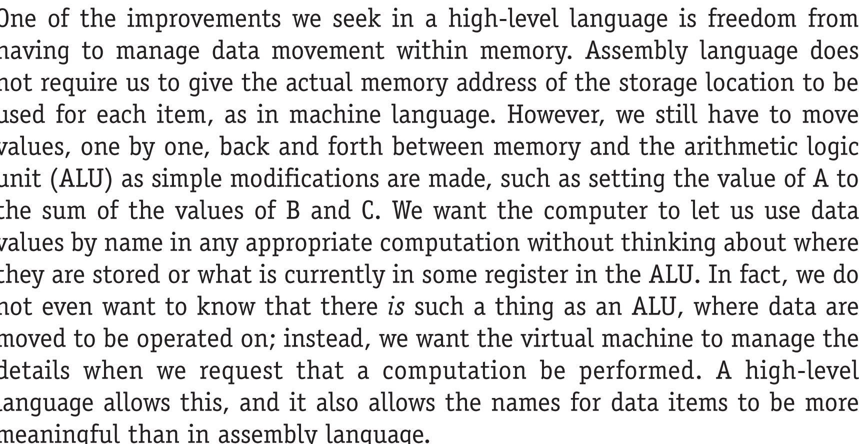
Ada Compiler and Graphics Library

You can download the free open-source GNAT Ada95 command-line compiler that is part of the GNU Compiler Collection from

www.gnu.org/software/gnat/gnat.html

There are versions that run on Linux, Mac OS X, and Windows systems.
The graphics library used in this chapter is AdaGraph, available for free download from

http://www.filewatcher.com/m/adagraph.zip.103876-0.html



2 Virtual Data Storage

One of the improvements we seek in a high-level language is freedom from having to manage data movement within memory. Assembly language does not require us to give the actual memory address of the storage location to be used for each item, as in machine language. However, we still have to move values, one by one, back and forth between memory and the arithmetic logic unit (ALU) as simple modifications are made, such as setting the value of A to the sum of the values of B and C. We want the computer to let us use data values by name in any appropriate computation without thinking about where they are stored or what is currently in some register in the ALU. In fact, we do not even want to know that there *is* such a thing as an ALU, where data are moved to be operated on; instead, we want the virtual machine to manage the details when we request that a computation be performed. A high-level language allows this, and it also allows the names for data items to be more meaningful than in assembly language.

Names in a programming language are called **identifiers**. Each language has its own specific rules for what a legal identifier can look like. In Ada an identifier

©2016 Course Technology, a part of Cengage Learning.

can be any combination of letters, digits, and the underscore symbol (_), as long as it starts with a letter. An additional restriction is that an identifier cannot be one of the few **reserved words**, such as BEGIN, INTEGER, FLOAT, and so forth, that have a special meaning in Ada and that you would not be likely to use anyway. The three integers *B*, *C*, and *A* in our assembly language program can therefore have more descriptive names, such as *subTotal*, *tax*, and *finalTotal*. The use of descriptive identifiers is one of the greatest aids to human understanding of a program. Identifiers can be almost arbitrarily long, so be sure to use a meaningful identifier such as *finalTotal* instead of something like *A*; the improved readability is well worth the extra typing time. Ada is *not* a **case-sensitive** language, which means that uppercase letters are treated the same as lowercase letters. Thus, *FinalTotal*, *Finaltotal*, and *finalTotal* are all the same identifier.

CAPITALIZATION OF IDENTIFIERS

There are two standard capitalization patterns for identifiers, particularly "multiple word" identifiers:

camel case: First word begins with a lowercase letter, additional words begin with uppercase letters (*finalTotal*)

Pascal case: All words begin with an uppercase letter (*FinalTotal*)

As mentioned earlier, the Ada code in this chapter will follow the formatting of the Ada LRM (Language Reference Manual) when referring to packages that are part of the Ada system. For other identifiers, the code in this chapter uses the following convention (examples included):

Simple variables – camel case: *speed*, *time*, *finalTotal*

Function names – camel case: *myFunction*, *getInput*

Class names – Pascal case: *MyClass*

Object names – camel case: *myObject*

The underscore character is not used in programmer-defined identifiers; it is used in "standard" Ada such as `TEXT_IO.PUT(. . .)`. Occasionally, we'll use single capital letters for identifiers in quick code fragments.

Data that a program uses can come in two varieties. Some quantities are fixed throughout the duration of the program, and their values are known ahead of time. These quantities are called **constants**. An example of a constant is the integer value 2. Another is an approximation to π, say 3.1416. The integer 2 is a constant that we don't have to name by an identifier, nor do we have to build the value 2 in memory manually by the equivalent of a .DATA pseudo-op. We can just use the symbol "2" in any program statement. When "2" is first encountered in a program statement, the binary representation of the integer 2 is automatically generated and stored in a memory location. Likewise, we can use "3.1416" for the real number value 3.1416, but if we are really using this number as an approximation to π, it is more informative to use the identifier *pi*.

Some quantities used in a program have values that change as the program executes, or values that are not known ahead of time but must be obtained from the computer user (or from a data file previously prepared

©2016 Course Technology, a part of Cengage Learning.

by the user) as the program runs. These quantities are called **variables**. For example, in a program doing computations with circles (where we might use the constant *pi*), we might need to obtain from the user or a data file the radius of the circle. This variable can be given the identifier *radius*.

Identifiers for variables serve the same purpose in program statements as pronouns do in ordinary English statements. The English statement "He will be home today" has specific meaning only when we plug in the value for which "He" stands. Similarly, a program statement such as

```
time := distance/FLOAT(speed);
```

becomes an actual computation only when numeric values have been stored in the memory locations referenced by the *distance* and *speed* identifiers.

We know that all data are represented internally in binary form. In Chapter 4 we noted that any one sequence of binary digits can be interpreted as a whole number, a negative number, a real number (one containing a decimal point, such as –17.5 or 28.342), or as a letter of the alphabet. Ada requires the following information about each variable in the program:

- What identifier we want to use for it (its name)
- What **data type** it represents (e.g., an integer or a letter of the alphabet)

The data type determines how many bytes will be needed to store the variable—that is, how many memory cells are to be considered as one **memory location** referenced by one identifier—and also how the string of bits in that memory location is to be interpreted. Ada provides several "primitive" data types that represent a single unit of information, as shown in Figure 4.

The way to give the necessary information within an Ada program is to declare each variable. A **variable declaration** consists of a list of one or more identifiers of the same data type followed by that data type. Our sample program used three declaration statements:

```
speed : INTEGER;       -- rate of travel
distance : FLOAT;      -- miles to travel
time : FLOAT;          -- time needed for this
                       -- travel
```

but these could have been combined into two:

```
speed : INTEGER;             -- rate of travel
distance, time : FLOAT;      -- miles to travel and
                             -- time needed for this
                             -- travel
```

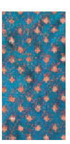


FIGURE 4
Some of the Ada Primitive Data Types

INTEGER	An integer quantity
FLOAT	A real number
CHARACTER	A character (a single keyboard character, such as 'a')

©2016 Course Technology, a part of Cengage Learning.

Where do the variable declarations go? All variable declarations are collected together in the declarative portion of the package, above the executable main program code. This guarantees that a variable will be declared before it can be used. It also gives the reader of the code quick information about the data that the program will be using.

What about the constant *pi*? We want to assign the fixed value 3.1416 to the *pi* identifier. Constant declarations are just like variable declarations, with the addition of the keyword **constant** and the assignment of the fixed value to the constant identifier.

```
pi : constant := 3.1416;
```

Note that the type of the variable *pi* is inferred from the way the number 3.1416 is written. This is a FLOAT number, so *pi* inherits type FLOAT.

Some programmers use all uppercase letters to denote constant identifiers, but the compiler identifies a constant quantity only by the presence of **constant** in the declaration. Once a quantity has been declared as a constant, any attempt later in the program to change its value generates an error message from the compiler.

In addition to variables of a primitive data type that hold only one unit of information, it is possible to declare a whole collection of related variables at one time. This allows storage to be set aside as needed to contain each of the values in this collection. For example, suppose we want to record the number of "hits" on a Web site for each month of the year. The value for each month is a single integer. We want a collection of 12 such integers, ordered in a particular way. An **array** groups together a collection of memory locations, all storing data of the same type. The following statement declares an array:

```
hits : array(0..11) of INTEGER;
```

The 12 individual array elements are numbered from *hits(0)* to *hits(11)*. (Notice that this Ada array counts from 0 up to 11.)

Thus, we use *hits(0)* to refer to the first entry in *hits*, which represents the number of visits to the Web site during the first month of the year, January. Continuing this numbering scheme, *hits(2)* refers to the number of visits during March, and *hits(11)* to the number of visits during December. In this way we use one declaration to set up 12 separate (but related) integer storage locations. Figure 5 illustrates this array.

Here is an example of the power of a high-level language. In assembly language we can name only individual memory locations—that is, individual items of data—but in Ada we can also assign a name to an entire collection of related data items. An array thus enables us to talk about an entire table of values, or the individual elements making up that table. If we are writing Ada programs to implement the data cleanup algorithms of Chapter 3, we can use an array of integers to store the 10 data items.

Ada gives the programmer a bit more flexibility in declaring arrays than most other programming languages. The *hits* array could also be declared as follows in Ada:

```
hits : array(1..12) of INTEGER;
```

This version counts from 1 up to 12. The ".." sequence specifies the beginning and ending values for the range of the array indexing.

©2016 Course Technology, a part of Cengage Learning.

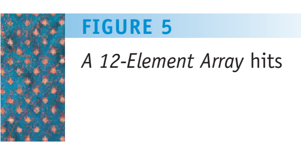
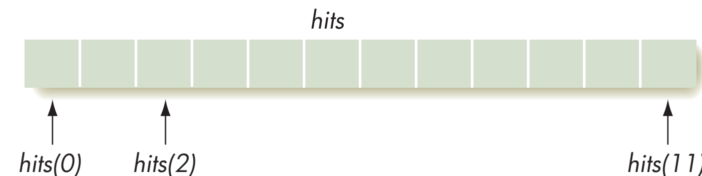


FIGURE 5

A 12-Element Array hits



PRACTICE PROBLEMS

1. Which of the following are legitimate Ada identifiers?

   `martinBradley` `C3P_OH` `Amy3` `3Right` `constant`

2. Write a declaration statement for an Ada program that uses one integer quantity called *number*.
3. Write an Ada statement that declares a type FLOAT constant called *taxRate* that has the value 5.5.
4. Using the *hits* array of Figure 5, how do you reference the number of hits on the Web page for August?

3 Statement Types

Now that we can reserve memory for data items by simply naming what we want to store and describing its data type, we will examine additional kinds of programming instructions (statements) that Ada provides. These statements enable us to manipulate the data items and do something useful with them. The instructions in Ada, or indeed in any high-level language, are designed as components for algorithmic problem solving, rather than as one-to-one translations of the underlying machine language instruction set of the computer. Thus they allow the programmer to work at a higher level of abstraction. In this section we examine three types of high-level programming language statements. They are consistent with the pseudocode operations we described in Chapter 2 (see Figure 2.9).

Input/output statements make up one type of statement. An **input statement** collects a value from the user for a variable within the program. In our TravelPlanner program, we need input statements to get the specific values of the speed and distance that are to be used in the computation. An **output statement** writes a message or the value of a program variable to the user's screen. Once the TravelPlanner program computes the time required to travel the given distance at the given speed, the output statement displays that value on the screen, along with other information about what that value means.

©2016 Course Technology, a part of Cengage Learning.

Another type of statement is the **assignment statement**, which assigns a value to a program variable. This is similar to what an input statement does, except that the value is not collected directly from the user, but is computed by the program. In pseudocode we called this a "computation operation."

Control statements, the third type of statement, affect the order in which instructions are executed. A program executes one instruction or program statement at a time. Without directions to the contrary, instructions are executed sequentially, from first to last in the program. (In Chapter 2 we called this a straight-line algorithm.) Imagine beside each program statement a light bulb that lights up while that statement is being executed; you would see a ripple of lights from the top to the bottom of the program. Sometimes, however, we want to interrupt this sequential progression and jump around in the program (which is accomplished by the instructions JUMP, JUMPGT, and so on, in assembly language). The progression of lights, which may no longer be sequential, illustrates the **flow of control** in the program—that is, the path through the program that is traced by following the currently executing statement. Control statements direct this flow of control.

3.1 *Input/Output Statements*

Remember that the job of an input statement is to collect from the user specific values for variables in the program. In pseudocode, to get the value for *speed* in the TravelPlanner program, we would say something like

Get value for speed

Ada can do this task using a function named *GET*. The input statement is

```
INT_IO.GET(speed);
```

Because all variables must be declared before they can be used, the declaration statement that says *speed* is to be a variable (of data type INTEGER) precedes this input statement.

Let's say that we have written the entire TravelPlanner program and it is now executing. When the preceding input statement is encountered, the program stops and waits for the user to enter a value for *speed* (by typing it at the keyboard, followed by pressing the ENTER key). For example, the user could type

```
58 <ENTER>
```

The *GET* function captures the string consisting of a 5 followed by an 8; this is just a two-character string, similar to the string "ab" consisting of an *a* followed by a *b*. In other words, the two-length string of characters "58" is not the same as the integer numeric value of 58, and we could not do any numerical computations with it. It is necessary to convert the string of numeric characters into an integer. That conversion from string to integer has been planned for in advance and is carried out by the instantiation (line 8 in the TravelPlanner program) of a special form for *GET* called *INT_IO.GET* that reads

©2016 Course Technology, a part of Cengage Learning.

strings and returns integers. If the user enters a decimal number as the input value for *speed*, e.g., 48.7, *INT_IO.GET* will gather the characters 4 and 8 and stop at the decimal point, since it could not be part of an integer. It will return 48 for the value of *speed*. However the .7 is still there as part of the input stream and will be consumed by the next *GET* statement, which is happy with .7 (or 0.7) as a FLOAT, assigns that value to *distance*, and produces an unexpected result for *time*.

In the usual case, the value of *distance* is input using the statement

```
FLO_IO.GET(distance);
```

Note that here the conversion of the string of characters gathered by *GET* is to type FLOAT. It would be acceptable to enter an integer value, say 657, instead of 657.0. The conversion process knows that it can make a FLOAT value from a string of numeric characters that does not contain a decimal point.

After the two input statements, the value of the time can be computed and stored in the memory location referenced by *time*. A pseudocode operation for producing output would be something like

Print the value of time

This could be done by the following statement:

```
FLO_IO.PUT(time);
```

Output in Ada is handled as the opposite of input. A value stored in memory—in this case the value of the variable *time*—is converted into a string and copied to the console (the screen). But we don't want the program to simply print a number with no explanation; we want some words to make the output meaningful.

The form of the output statement for text is

```
TEXT_IO.PUT(string);
```

Literal strings (enclosed in double quotes) are printed out exactly as is. For example,

```
TEXT_IO.PUT("Here's your answer.");
```

prints

```
Here's your answer.
```

The following Ada statement will start a new line in the output display, which is useful for formatting the output to make it easier to read.

```
TEXT_IO.NEW_LINE;
```

A single Ada statement can be spread over multiple lines, but a line break cannot occur in the middle of a literal string. The solution is to make two smaller substrings and join them together (concatenate them), as in

©2016 Course Technology, a part of Cengage Learning.

```
TEXT_IO.PUT("Oh for a sturdy ship to sail, "
                 & "and a star to steer her by.");
```

which has the same effect as if the literal string had all been written on a single line. The & is the Ada **concatenation operator**.

Running the TravelPlanner program with our original data of 58 mph and 657.5 miles resulted in a printed value of *time* of

```
1.13362E+01
```

This is fairly ridiculous output—it does not make sense to display the result in scientific notation. The appearance of numerical output can be controlled, rather than leaving it up to the system to decide, by including additional parameters in the output statement. If only two digits to the right of the decimal point are to be displayed for *time*, the output statement would take the following form.

```
FLO_IO.PUT(time, 5, 2, 0);
```

The parameters for this version of the *PUT* function are

`FLO_IO.PUT`(value, digits before the decimal point, digits after the decimal point, number digits in exponent)

If these parameter values are used in the *PUTs* for *time* and *distance*, the resulting output is:

```
Enter your speed in mph: 58
Enter your distance in miles: 657.5
At          58 mph, it will take
   11.34 hours to travel   657.50 miles.
```

Note that the value of *time* is rounded to 11.34 during the output process. A single parameter in the *INT_IO.PUT* function will control the number of columns for the integer output.

Let's back up a bit and note that we also need to print some text information before the input statement, to alert the user that the program expects some input. A statement such as

```
TEXT_IO.PUT("Enter your speed in mph: ");
```

acts as a user **prompt**. Without a prompt, the user may be unaware that the program is waiting for some input; instead, it may simply seem to the user that the program is "hung up."

Assembling all of these bits and pieces, we can see that

```
TEXT_IO.PUT("Enter your speed in mph: ");
INT_IO.GET(speed);
TEXT_IO.PUT("Enter your distance in miles: ");
FLO_IO.GET(distance);
```

©2016 Course Technology, a part of Cengage Learning.

is a series of prompt, input, prompt, input statements to get the data, and then

```
TEXT_IO.PUT("At ");
INT_IO.PUT(speed);
TEXT_IO.PUT(" mph, ");
TEXT_IO.PUT("it will take ");
TEXT_IO.NEW_LINE;
FLO_IO.PUT(time, 5, 2, 0);
TEXT_IO.PUT(" hours to travel ");
FLO_IO.PUT(distance, 5, 2, 0);
TEXT_IO.PUT(" miles.");
TEXT_IO.NEW_LINE;
```

writes out the computed value of the *time* along with the associated input values in an informative message. In the middle, we need a program statement to compute the value of *time*. We can do this with a single assignment statement; the assignment statement is explained in the next section.

PRACTICE PROBLEMS

1. Write two statements that prompt the user to enter an integer value and store that value in a (previously declared) variable called *quantity*.
2. A program has computed a value for the integer variable *height*. Write output statements that print this variable using six columns and cause successive output to appear on the next line.
3. What appears on the screen after execution of the following statements?

```
TEXT_IO.PUT("This is");
TEXT_IO.PUT("goodbye");
TEXT_IO.NEW_LINE;
```

3.2 *The Assignment Statement*

As we said earlier, an assignment statement assigns a value to a program variable. This is accomplished by evaluating some expression and then writing the resulting value in the memory location referenced by the program variable. The general pseudocode operation

Set the value of “variable” to “arithmetic expression”

has as its Ada equivalent

variable := expression;

©2016 Course Technology, a part of Cengage Learning.

The expression on the right is evaluated, and the result is then written into the memory location named on the left. For example, suppose that *A*, *B*, and *C* have all been declared as integer variables in some program. The assignment statements

```
B := 2;
C := 5;
```

result in *B* taking on the value 2 and *C* taking on the value 5. After execution of

```
A := B + C;
```

A has the value that is the sum of the current values of *B* and *C*. Assignment is a destructive operation, so whatever *A*'s previous value was, it is gone. Note that this one assignment statement says to add the values of *B* and *C* and assign the result to *A*. This one high-level language statement is equivalent to three assembly language statements needed to do this same task (LOAD B, ADD C, STORE A). A high-level language program thus packs more power per line than an assembly language program. To state it another way, whereas a single assembly language instruction is equivalent to a single machine language instruction, a single Ada instruction is usually equivalent to many assembly language instructions or machine language instructions, and it allows us to think at a higher level of problem solving.

In the assignment statement, the expression on the right is evaluated first. Only then is the value of the variable on the left changed. This means that an assignment statement like

```
A := A + 1;
```

makes sense. If *A* has the value 7 before this statement is executed, then the expression evaluates to

7 + 1, or 8

and 8 then becomes the new value of *A*.

All four basic arithmetic operations can be done in Ada, where they are denoted by

+ Addition
- Subtraction
* Multiplication
/ Division

For the most part, this is standard mathematical notation, rather than the somewhat verbose assembly language op code mnemonics such as SUBTRACT. The reason a special symbol is used for multiplication is that × would be confused with *x*, an identifier, · (a multiplication dot) doesn't appear on the keyboard, and juxtaposition—writing *AB* for *A***B*—would look like a single identifier named *AB*.

We do have to pay some attention to data types. Ada is so strongly typed that you cannot, for example, mix types in an arithmetic expression. Of the three expressions below

7.0/2 7/2.0 7.0/2.0

only the last one is acceptable. The first two will result in compiler errors.

©2016 Course Technology, a part of Cengage Learning.

C8814_chapter_ada.qxd 1/17/15 5:27 PM Page 18

However, if the two values being divided are both integers, the result is an integer value; if the division doesn't "come out even," the integer value is obtained by truncating the answer to an integer quotient. Thus,

```
7/2
```

results in the value 3. Think of grade-school long division of integers:

```
   3
2)7
  6
  1
```

Here the quotient is 3 and the remainder is 1. Ada also provides an operation, with the name **mod**, to obtain the integer remainder. Using this operation,

```
7 mod 2
```

results in the value 1. If the values are stored in type INTEGER variables, the same thing happens. For example,

```
numerator : INTEGER;
denominator : INTEGER;

numerator := 7;
denominator := 2;
TEXT_IO.PUT("The result of ");
INT_IO.PUT(numerator, 1);
TEXT_IO.PUT("/");
INT_IO.PUT(denominator, 1);
TEXT_IO.PUT(" is ");
INT_IO.PUT(numerator / denominator, 1);
```

produces the output

```
The result of 7/2 is 3
```

Automatic type conversion, or **type casting** as it is called in most languages, does not take place in Ada. To solve this problem, Ada provides functions that convert types explicitly. For example

```
time := distance / FLOAT(speed);
```

Here, the *FLOAT* function takes in an INTEGER quantity (*speed*) and returns the equivalent FLOAT value so that the division operation involves two FLOAT quantities and avoids a compiler error. There is a corresponding function named *INTEGER* that takes in a FLOAT value, rounds it to an integer, and

©2016 Course Technology, a part of Cengage Learning.

returns the resulting value. For example, with *a* declared as an INTEGER, after execution of the statement

```
a := INTEGER(71.6);
```

the value in *a* would be 72.

Data types also play a role in assignment statements. Suppose the expression in an assignment statement evaluates to a real number (a floating-point number) and is then assigned to an identifier that has been declared as an integer, or vice versa. In either case Ada will produce a compiler error, and again, to avoid this, an explicit type conversion must be done by using the *INTEGER* or *FLOAT* function.

You should only assign an expression that has a character value to a variable that has been declared to be type CHARACTER. Suppose that *letter* is a variable of type CHARACTER. Then

```
letter := 'm';
```

is a legitimate assignment statement, giving *letter* the value of the character 'm'. Note that single quotation marks are used here, as opposed to the double quotation marks that enclose a literal string. The assignment

```
letter := '4';
```

is also acceptable; the single quotes around the 4 mean that it is being treated as just another character on the keyboard, not as the integer 4.

PRACTICE PROBLEMS

1. *newNumber* and *next* are integer variables in an Ada program. Write a statement to assign the value of *newNumber* to *next*.
2. The goal is to compute the *average* when the following statements are executed (*total* and *number* are type INTEGER, and *average* is type FLOAT). Would this code compile in Ada? If not, how should it be written? What is the expected output value?

```
total := 277;
number := 5;
average := total/number;
```

3.3 *Control Statements*

We mentioned earlier that sequential flow of control is the default; that is, a program executes instructions sequentially from first to last. The flowchart in Figure 6 illustrates this, where S1, S2, . . ., Sk are program instructions (i.e., program statements).

©2016 Course Technology, a part of Cengage Learning.

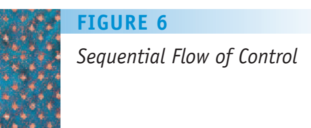
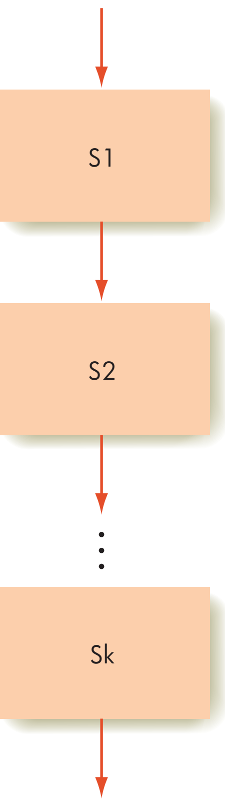


FIGURE 6
Sequential Flow of Control



As stated in Chapter 2, no matter how complicated the task to be done, only three types of control mechanisms are needed:

1. **Sequential:** Instructions are executed in order.
2. **Conditional:** Which instruction executes next depends on some condition.
3. **Looping:** A group of instructions may be executed many times.

Sequential flow of control, the default, is what occurs if the program does not contain any instances of the other two control structures. In the TravelPlanner program, for example, instructions are executed sequentially, beginning with the input statements, next the computation, and finally the output statement.

In Chapter 2 we introduced pseudocode notation for conditional operations and looping. In Chapter 6 we learned how to write somewhat laborious assembly language code to implement conditional operations and looping. Now we'll see how Ada provides instructions that directly carry out these control structure mechanisms—more evidence of the power of high-level language instructions. We can think in a pseudocode algorithm design mode, as we did in Chapter 2, and then translate that pseudocode directly into Ada code.

Conditional flow of control begins with the evaluation of a **Boolean condition**, also called a **Boolean expression**, which can be either true or false. We discussed these "true/false conditions" in Chapter 2, and we also

©2016 Course Technology, a part of Cengage Learning.

encountered Boolean expressions in Chapter 4, where they were used to design circuits. A Boolean condition often involves comparing the values of two expressions and determining whether they are equal, whether the first is greater than the second, and so on. Again assuming that *A*, *B*, and *C* are integer variables in a program, the following are legitimate Boolean conditions:

```
A  =  0          (Does A currently have the value 0?)
B  <  (A + C)    (Is the current value of B less than the sum of the
                 current values of A and C?)
A  /=  B         (Does A currently have a different value than B?)
```

If the current values of *A*, *B*, and *C* are 2, 5, and 7, respectively, then the first condition is false (*A* does not have the value zero), the second condition is true (5 is less than 2 plus 7), and the third condition is true (*A* and *B* do not have equal values).

Comparisons need not be numeric. They can also be done between variables of type CHARACTER, where the "ordering" is the usual alphabetic ordering. If *initial* is a value of type CHARACTER with a current value of 'D', then

```
initial = 'F'
```

is false because *initial* does not have the value 'F', and

```
initial < 'P'
```

is true because 'D' precedes 'P' in the alphabet (or, more precisely, because the binary code for 'D' is numerically less than the binary code for 'P'). Note that the comparisons are case sensitive, so 'F' is not equal to 'f', but 'F' is less than 'f'.

Figure 7 shows the comparison operations available in Ada. Boolean conditions can be built up using the Boolean operators AND, OR, and NOT. Truth tables for these operators were given in Chapter 4 (Figures 4.12–4.14). The only new thing is the symbols that Ada uses for these operators, shown in Figure 8.

A conditional statement relies on the value of a Boolean condition (true or false) to decide which programming statement to execute next. If the condition is true, one statement is executed next, but if the condition is false, a different statement is executed next. Control is therefore no longer in a

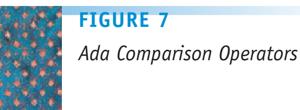


FIGURE 7

Ada Comparison Operators

COMPARISON	SYMBOL	EXAMPLE	EXAMPLE RESULT
the same value as	=	2 = 5	false
less than	<	2 < 5	true
less than or equal to	<=	5 <= 5	true
greater than	>	2 > 5	false
greater than or equal to	>=	2 >= 5	false
not the same value as	/=	2 /= 5	true

©2016 Course Technology, a part of Cengage Learning.

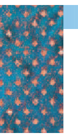


FIGURE 8
Ada Boolean Operators

OPERATOR	SYMBOL	EXAMPLE	EXAMPLE RESULT
AND	and	(2 < 5) and (2 > 7)	false
OR	or	(2 < 5) or (2 > 7)	true
NOT	not	not (2 = 5)	true

straight-line (sequential) flow, but hops to one place or to another. Figure 9 illustrates this situation. If the condition is true, the statement S1 is executed (and statement S2 is not); if the condition is false, the statement S2 is executed (and statement S1 is not). In either case, the flow of control then continues on to statement S3. We saw this same scenario when we discussed pseudocode conditional statements in Chapter 2 (Figure 2.4).

The Ada instruction that carries out conditional flow of control is called an **if-else** statement. It has the following form (note that the words *if*, *then*, *else*, and *end if* are lowercase).

```
if Boolean condition
 then
        S1;
 else
        S2;
end if;
```

On the next page is a simple if-else statement, where we assume that *A, B,* and *C* are integer variables.

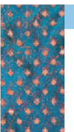
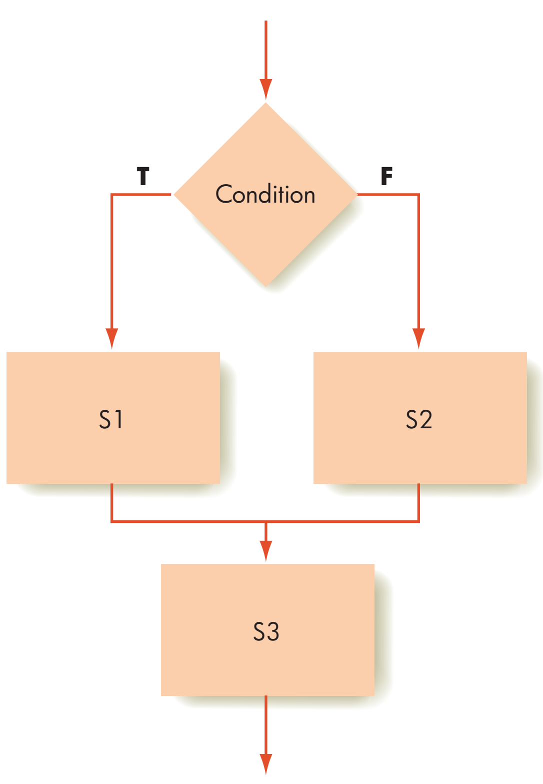


FIGURE 9
Conditional Flow of Control (if-else)



©2016 Course Technology, a part of Cengage Learning.

```
if B < (A + C)

 then
     A := 2*A;
 else
     A := 3*A;
end if
```

Note that in the Ada syntax for the *if* statement, the "parts" of the statement are delimited by reserved words rather than by curly braces as in C-like languages.

Suppose that when this statement is reached, the values of *A*, *B*, and *C* are 2, 5, and 7, respectively. As we noted before, the condition $B < (A + C)$ is then true, so the statement

```
A := 2*A;
```

is executed, and the value of *A* is changed to 4. However, suppose that when this statement is reached, the values of *A*, *B*, and *C* are 2, 10, and 7, respectively. Then the condition $B < (A + C)$ is false, the statement

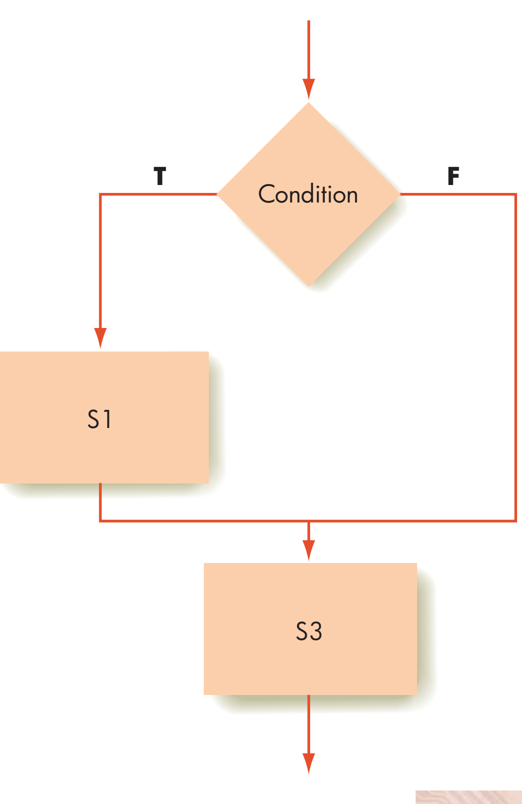
```
A := 3*A;
```

is executed, and the value of *A* is changed to 6.

A variation on the if-else statement is to allow an "empty else" case. Here we want to do something if the condition is true, but if the condition is false, we want to do nothing. Figure 10 illustrates the empty else case. If the condition is true,

FIGURE 10

If-else with Empty Else



©2016 Course Technology, a part of Cengage Learning.

statement S1 is executed, and after that the flow of control continues on to statement S3, but if the condition is false, nothing happens except to move the flow of control directly on to statement S3.

This *if* variation of the if-else statement can be accomplished by omitting the word *else*. This form of the instruction therefore looks like

```
if Boolean condition
   then
      S1
end if;
```

We could write

```
if B < (A + C)
  then
     A := 2*A;
end if;
```

This has the effect of doubling the value of A if the condition is true and of doing nothing if the condition is false.

It is possible to include many statements in either the “then” or the “else” part of the if statement. For example, in

```
if B < (A + C)
  then
   TEXT_IO.PUT("This is the first statement.");
   TEXT_IO.PUT("This is the second statement.");
   TEXT_IO.PUT("This is the third statement.");
end if;
```

all three output statements are executed if the condition is true. The implication is that in Figure 9, S1 or S2 can be a collection of statements, called a **compound statement**. This makes the if-else statement potentially much more powerful and similar to the pseudocode conditional statement in Figure 2.9.

Let’s expand on our TravelPlanner program and give the user of the program a choice of computing the time either as a decimal number (3.75 hours) or as hours and minutes (3 hours, 45 minutes). This situation is ideal for a conditional statement. Depending on what the user wants to do, the program does one of two tasks. For either task, the program still needs information about the speed and distance. The program must also collect information to indicate which task the user wishes to perform. We need an additional variable in the program to store this information. Let’s use a variable called *choice* of type CHARACTER to collect the user’s choice of which task to perform. We also need two new integer variables to store the values of hours and minutes.

Figure 11 shows the new program, with the three additional declared variables. The condition evaluated at the beginning of the if-else statement tests whether *choice* has the value ‘D’. If so, then the condition is true, and the first group of statements is executed—that is, the time is output in decimal format as we have been doing all along. If *choice* does not have the value ‘D’, then the condition is false. In this event, the second group of statements is executed. Note that because of the way the condition is written, if *choice* does

©2016 Course Technology, a part of Cengage Learning.

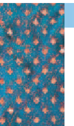


FIGURE 11

The TravelPlanner Program with a Conditional Statement

```
-- Computes and outputs travel time
-- for a given speed and distance
-- Written by J. Q. Programmer, 6/15/16

WITH TEXT_IO;

PROCEDURE TravelPlanner IS
  PACKAGE INT_IO IS NEW TEXT_IO.INTEGER_IO(INTEGER);
  PACKAGE FLO_IO IS NEW TEXT_IO.FLOAT_IO(FLOAT);

  speed : INTEGER;        -- rate of travel
  distance : FLOAT;       -- miles to travel
  time : FLOAT;           -- time needed for this travel
  hours : INTEGER;        -- time for travel in hours
  minutes : INTEGER;      -- leftover time in minutes
  choice : CHARACTER;     -- choice of output as
                          -- decimal hours
                          -- or hours and minutes

BEGIN

  TEXT_IO.PUT("Enter your speed in mph: ");
  INT_IO.GET(speed);
  TEXT_IO.PUT("Enter your distance in miles: ");
  FLO_IO.GET(distance);
  TEXT_IO.PUT("Enter your choice of format" &
              " for time, ");
  TEXT_IO.NEW_LINE;
  TEXT_IO.PUT("decimal hours (D) " &
              "or hours and minutes (M): ");
  TEXT_IO.GET(choice);

  if choice = 'D'
   then
    time := distance / FLOAT(speed);
    TEXT_IO.PUT("At ");
    INT_IO.PUT(speed, 3);
    TEXT_IO.PUT(" mph, ");
    TEXT_IO.PUT("it will take ");
    TEXT_IO.NEW_LINE;
    FLO_IO.PUT(time, 5, 2, 0);
    TEXT_IO.PUT(" hours to travel ");
    FLO_IO.PUT(distance, 5, 2, 0);
    TEXT_IO.PUT(" miles.");
    TEXT_IO.NEW_LINE;
   else
    time := distance / FLOAT(speed);
    hours := INTEGER(time + 0.5) - 1;
    minutes := INTEGER((time - FLOAT(hours)) * 60.0);
    TEXT_IO.PUT("At ");
    INT_IO.PUT(speed, 3);
    TEXT_IO.PUT(" mph, ");
    TEXT_IO.PUT("it will take ");
    TEXT_IO.NEW_LINE;
```

©2016 Course Technology, a part of Cengage Learning.

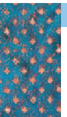


FIGURE 11

The TravelPlanner Program with a Conditional Statement (continued)

```
    INT_IO.PUT(hours, 3);
    TEXT_IO.PUT(" hours and ");
    INT_IO.PUT(minutes, 3);
    TEXT_IO.PUT(" minutes to travel ");
    FLO_IO.PUT(distance, 5, 2, 0);
    TEXT_IO.PUT(" miles.");
    TEXT_IO.NEW_LINE;
  end if;

END TravelPlanner;
```

not have the value 'D', it is assumed that the user wants to compute the time in hours and minutes, even though *choice* may have any other non-D value (including 'd') that the user may have typed in response to the prompt.

To compute hours and minutes (the *else* clause of the if-else statement), time is computed in the usual way, which results in a decimal value. The whole number (integer) part of that decimal is the number of hours needed for the trip. We can get this number by a somewhat complex statement that is shown below.

```
hours := INTEGER(time + 0.5) - 1;
```

The issue is that the Ada conversion function *INTEGER* rounds the decimal value. To get the correct integer component, it is necessary to force the function to round up and then subtract one to obtain the proper value (an example follows shortly).

To find the fractional part of the hour that we dropped, we subtract *hours* from *time*. We multiply this by 60 to turn it into some number of minutes, but this is still a decimal number. We do another explicit type cast to round this to an integer value for *minutes*:

```
minutes := INTEGER((time - FLOAT(hours)) * 60.0);
```

For example, if the user enters data of 50 mph and 475 miles and requests output in hours and minutes, the following table shows the computed values.

Quantity	*Value*
speed	*50*
distance	*475*
time = distance/speed	*9.5*
hours = INTEGER (time + 0.5) - 1	*9*
time – FLOAT(hours)	*0.5*
(time – FLOAT(hours)*)*60*	*30.0*
minutes = INTEGER((time - FLOAT(hours)) * 60.0)	*30*

©2016 Course Technology, a part of Cengage Learning.

Here is the actual program output for this case:

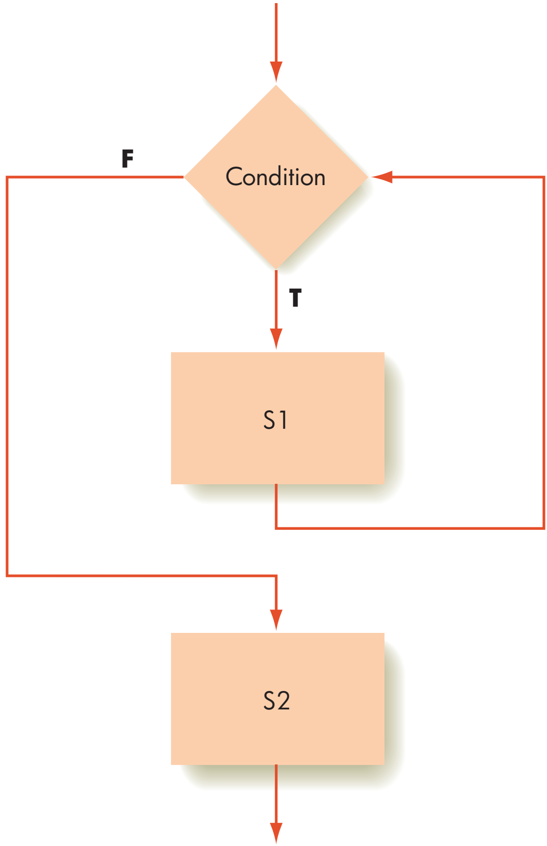
```
Enter your speed in mph: 50
Enter your distance in miles: 475
Enter your choice of format for time,
decimal hours (D) or hours and minutes (M): M
At  50 mph, it will take
  9 hours and  30 minutes to travel   475.00 miles.
```

The two groups of statements in an if-else statement are identified by the enclosing keywords, but in Figure 11 we also indented them to make them easier to pick out when looking at the program. Like comments, indentation is ignored by the computer but is valuable in helping people to more readily understand a program.

Now let's look at the third variation on flow of control, namely looping (iteration). We want to execute the same group of statements (called the **loop body**) repeatedly, depending on the result of a Boolean condition. As long as (while) the condition remains true, the loop body is executed. The condition is tested before each execution of the loop body. When the condition becomes false, the loop body is not executed again, which is usually expressed by saying that the algorithm *exits* the loop. To ensure that the algorithm ultimately exits the loop, the condition must be such that its truth value can be affected by what happens when the loop body is executed. Figure 12 illustrates the

FIGURE 12

While Loop



©2016 Course Technology, a part of Cengage Learning.

while loop. The loop body is statement S1 (which can be a compound statement), and S1 is executed *while* the condition is true. Once the condition is false, the flow of control moves on to statement S2. If the condition is false when it is first evaluated, then the body of the loop is never executed at all. We saw this same scenario when we discussed pseudocode looping statements in Chapter 2 (Figure 2.6).

Ada uses a variation of its *loop* statement to achieve the *while* iteration scheme. The statement to implement this type of looping has the form shown below.

```
while Boolean condition
  loop
    S1
  end loop;
```

Again, S1 can be a compound statement. For example, suppose we want to write a program to add up a number of nonnegative integers that the user supplies and write out the total. We need a variable to hold the total; we'll call this variable *sum,* and make its data type INTEGER. To handle the numbers to be added, we could declare a bunch of integer variables such as *n1, n2, n3,* . . . and do a series of input-and-add statements of the form

```
INT_IO.GET(n1);
sum := sum + n1;
INT_IO.GET(n2);
sum := sum + n2;
```

and so on. There are two problems with this approach. The first is that we may not know ahead of time how many numbers the user wants to add. If we declare variables *n1, n2,* . . ., *n25*, and the user wants to add 26 numbers, the program won't do the job. The second problem is that this approach requires too much effort. Suppose that we know the user wants to add 2000 numbers. We could declare 2000 variables (*n1,* . . ., *n2000*), and we could write the above input-and-add statements 2000 times, but it wouldn't be fun. Nor is it necessary—we are doing a very repetitive task here, and we should be able to use a loop mechanism to simplify the job. (We faced a similar situation in the first pass at a sequential search algorithm, Figure 2.11; our solution there was also to use iteration.)

Even if we use a loop mechanism, we are still adding a succession of values to *sum*. Unless we are sure that the value of *sum* is zero to begin with, we cannot be sure that the answer isn't nonsense. Remember that the identifier *sum* is simply an indirect way to designate a memory location in the computer. That memory location contains a pattern of bits, perhaps left over from whatever was stored there when some previous program was run. We cannot assume that just because this program hasn't used *sum*, its value is zero. (In contrast, the assembly language statement SUM: .DATA 0 reserves a memory location, assigns it the identifier SUM, and fills it with the value zero.) If we want the beginning value of *sum* to be zero, we must use an assignment statement. Using assignment statements to set the values of certain variables before they are used by the program is called **initialization of variables**.

©2016 Course Technology, a part of Cengage Learning.

Now on to the loop mechanism. First, let's note that once a number has been read in and added to *sum*, the program doesn't need to know the value of the number any longer. We can declare just one integer variable called *number* and use it repeatedly to hold the first numerical value, then the second, and so on. The general idea is

```
sum := 0; -- initialize sum
while (there are more numbers to add)
 loop
   INT_IO.GET(number);
   sum := sum + number;
 end loop;
TEXT_IO.PUT("The total is ");
INT_IO.PUT(sum, 3);
TEXT_IO.NEW_LINE;
```

Now we have to figure out what the condition "there are more numbers to add" really means. Because we are adding nonnegative integers, we could ask the user to enter one extra integer that is not part of the legitimate data but is instead a signal that there *are* no more data. Such a value is called a **sentinel value**. For this problem, any negative number would be a good sentinel value. Because the numbers to be added are all nonnegative, the appearance of a negative number signals the end of the legitimate data. We don't want to process the sentinel value (because it is not a legitimate data item); we only want to use it to terminate the looping process. This might suggest the following code:

```
sum := 0;           -- initialize sum
while number >= 0   -- but there is a problem here,
                    -- see following discussion
 loop
   INT_IO.GET(number);
   sum := sum + number;
 end loop;
TEXT_IO.PUT("The total is ");
INT_IO.PUT(sum, 3);
TEXT_IO.NEW_LINE;
```

Here's the problem. How can we test whether *number* is greater than or equal to 0 if we haven't read the value of *number* yet? We need to do a preliminary input for the first value of *number* outside of the loop and then test that value in the loop condition. If it is nonnegative, we want to add it to *sum* and then read the next value and test it. Whenever the value of *number* is negative (including the first value), we want to do nothing with it—that is, we want to avoid executing the loop body. The following statements do this; we've also added instructions to the user.

```
sum := 0; -- initialize sum
TEXT_IO.PUT("Please enter numbers to add; ");
TEXT_IO.NEW_LINE;
TEXT_IO.PUT("terminate with a negative number.");
TEXT_IO.NEW_LINE;
INT_IO.GET(number);
```


©2016 Course Technology, a part of Cengage Learning.

```
while number >= 0
 loop
   sum := sum + number;
   INT_IO.GET(number);
 end loop;
TEXT_IO.PUT("The total is ");
INT_IO.PUT(sum, 3);
TEXT_IO.NEW_LINE;
```

The value of *number* gets changed within the loop body by reading in a new value. The new value is tested, and if it is nonnegative, the loop body executes again, adding the data value to *sum* and reading in a new value for *number*. The loop terminates when a negative value is read in. Remember the requirement that something within the loop body must be able to affect the truth value of the condition. In this case, it is reading in a new value for *number* that has the potential to change the value of the condition from true to false. Without this requirement, the condition, once true, would remain true forever, and the loop body would be endlessly executed. This results in what is called an **infinite loop**. A program that contains an infinite loop will execute forever (or until the programmer gets tired of waiting and interrupts the program, or until the program exceeds some preset time limit).

Here is a sample of the program output.

```
Please enter numbers to add;
terminate with a negative number.
5
6
10
-1
The total is 21
```

The problem we've solved here, adding nonnegative integers until a negative sentinel value occurs, is the same one solved using assembly language in Chapter 6. The preceding Ada code is almost identical to the pseudocode version of the algorithm shown in Figure 6.7. Thanks to the power of the language, the Ada code embodies the algorithm directly, at a high level of thinking, whereas in assembly language this same algorithm had to be translated into the lengthy and awkward code of Figure 6.8.

To process data for a number of different trips in the TravelPlanner program, we could use a while loop. During each pass through the loop, the program computes the time for a given speed and distance. The body of the loop is therefore exactly like our previous code. All we are adding here is the framework that provides looping. To terminate the loop, we could use a sentinel value, as we did for the program above. A negative value for *speed*, for example, is not a valid value and could serve as a sentinel value. Instead of that, let's allow the user to control loop termination by having the program ask the user whether he or she wishes to continue. We'll need a variable to hold the user's response to this question. Of course, the user could answer "N" at the first query, the loop body would never be executed at all, and the program would terminate. Figure 13 shows the complete program.

©2016 Course Technology, a part of Cengage Learning.

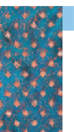


FIGURE 13

The TravelPlanner Program with Looping

```
-- Computes and outputs travel time
-- for a given speed and distance
-- Written by J. Q. Programmer, 6/15/16

WITH TEXT_IO;

PROCEDURE TravelPlanner IS
  PACKAGE INT_IO IS NEW TEXT_IO.INTEGER_IO(INTEGER);
  PACKAGE FLO_IO IS NEW TEXT_IO.FLOAT_IO(FLOAT);

  speed : INTEGER;          -- rate of travel
  distance : FLOAT;         -- miles to travel
  time : FLOAT;             -- time needed for this travel
  hours : INTEGER;          -- time for travel in hours
  minutes : INTEGER;        -- leftover time in minutes
  choice : CHARACTER;       -- choice of output as
                            -- decimal hours
                            -- or hours and minutes
  more : CHARACTER;         -- user's choice to do
                            -- another trip
BEGIN

  TEXT_IO.PUT("Do you want to plan a trip? " &
              "(Y or N): ");
  TEXT_IO.GET(more);

  while more = 'Y'          -- more trips to plan
   loop
    TEXT_IO.PUT("Enter your speed in mph: ");
    INT_IO.GET(speed);
    TEXT_IO.PUT("Enter your distance in miles: ");
    FLO_IO.GET(distance);
    TEXT_IO.PUT("Enter your choice of format" &
                " for time, ");
    TEXT_IO.NEW_LINE;
    TEXT_IO.PUT("decimal hours (D) " &
                "or hours and minutes (M): ");
    TEXT_IO.GET(choice);

    if choice = 'D'
     then
      time := distance / FLOAT(speed);
      TEXT_IO.PUT("At ");
      INT_IO.PUT(speed, 3);
      TEXT_IO.PUT(" mph, ");
      TEXT_IO.PUT("it will take ");
      TEXT_IO.NEW_LINE;
      FLO_IO.PUT(time, 5, 2, 0);
      TEXT_IO.PUT(" hours to travel ");
      FLO_IO.PUT(distance, 5, 2, 0);
      TEXT_IO.PUT(" miles.");
      TEXT_IO.NEW_LINE;
     else
      time := distance / FLOAT(speed);
```

©2016 Course Technology, a part of Cengage Learning.

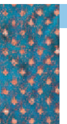


FIGURE 13

The TravelPlanner Program with Looping (continued)

```
        hours := INTEGER(time + 0.5) - 1;
        minutes := INTEGER((time - FLOAT(hours)) * 60.0);
        TEXT_IO.PUT("At ");
        INT_IO.PUT(speed, 3);
        TEXT_IO.PUT(" mph, ");
        TEXT_IO.PUT("it will take ");
        TEXT_IO.NEW_LINE;
        INT_IO.PUT(hours, 3);
        TEXT_IO.PUT(" hours and ");
        INT_IO.PUT(minutes, 3);
        TEXT_IO.PUT(" minutes to travel ");
        FLO_IO.PUT(distance, 5, 2, 0);
        TEXT_IO.PUT(" miles.");
        TEXT_IO.NEW_LINE;
      end if;
      TEXT_IO.PUT("Do you want to plan another trip? " &
                  "(Y or N): ");
      TEXT_IO.GET(more);
     end loop;

  END TravelPlanner;
```

PRACTICE PROBLEMS

Assume all variables have previously been declared.

1. What is the output from the following section of code?

```
number1 := 15;
number2 := 7;
if number1 >= number2
 then
    INT_IO.PUT(2*number1);
 else
    INT_IO.PUT(2*number2);
end if;
```

2. What is the output from the following section of code?

```
scores := 1;
while scores < 20
 loop
     scores := scores + 2;
     INT_IO.PUT(scores);
 end loop;
```

3. What is the output from the following section of code?

```
quotaThisMonth := 7;
quotaLastMonth := quotaThisMonth + 1;
if (quotaThisMonth > quotaLastMonth) or
    (quotaLastMonth >= 8)
```

©2016 Course Technology, a part of Cengage Learning.

PRACTICE PROBLEMS

```
     then
         TEXT_IO.PUT("Yes");
         quotaLastMonth := quotaLastMonth + 1;
     else
         TEXT_IO.PUT("No");
         quotaThisMonth := quotaThisMonth + 1;
    end if;
```

4. How many times is the *PUT* statement executed in the following section of code?

```
left := 10;
right := 20;
while left <= right
 loop
     INT_IO.PUT(left);
     left := left + 2;
 end loop;
```

5. Write an Ada statement that outputs “Equal” if the integer values of *night* and *day* are the same, but otherwise does nothing.

4 Another Example

Let’s briefly review the types of Ada programming statements we’ve learned. We can do input and output—reading values from the user into memory, writing values out of memory for the user to see, being sure to use meaningful variable identifiers to reference memory locations. We can assign values to variables within the program. And we can direct the flow of control by using conditional statements or looping. Although many other statement types are available in Ada, you can do almost everything using only the modest collection of statements we have described. The power lies in how these statements are combined and nested within groups to produce ever more complex courses of action.

For example, suppose we write a program to assist SportsWorld, a company that installs circular swimming pools. In order to estimate their costs for swimming pool covers or for fencing to surround the pool, SportsWorld needs to know the area or circumference of a pool, given its radius. A pseudocode version of the program is shown in Figure 14.

We should be able to translate this pseudocode fairly directly into an Ada package body. Other things we need to add to complete the program are:

- A prologue comment to explain what the program does (optional but always recommended for program documentation)
- A *with* statement to gain access to TEXT_IO and instantiations of IO packages for the data types to be input or output

©2016 Course Technology, a part of Cengage Learning.

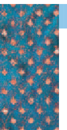


FIGURE 14

A Pseudocode Version of the SportsWorld Program

```
Get value for user's choice about continuing
While user wants to continue, do the following steps
    Get value for pool radius
    Get value for choice of task
    If task choice is circumference
        Compute pool circumference
        Print output
    Else (task choice is area)
        Compute pool area
        Print output
    Get value for user's choice about continuing
Stop
```

- A declaration for the constant value *pi* (3.1416)
- Variable declarations

Figure 15 gives the complete program. Figure 16 shows what actually appears on the screen when this program is executed with some sample data.

One point of interest in this code is that Ada has an exponentiation operator, **. So the area of the circular swimming pool is computed with the following line of code.

```
area := pi * radius ** 2;
```

The exponentiation operator is at the highest precedence level, so the ** operation is carried out (on the variable *radius*) before the * (multiplication) operation.

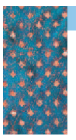


FIGURE 15

The SportsWorld Program

```
-- This program helps SportsWorld estimate costs
-- for pool covers and pool fencing by computing
-- the area or circumference of a circle
-- with a given radius.
-- Any number of circles can be processed.
-- Written by M. Phelps, 10/05/16

WITH TEXT_IO;

PROCEDURE SportsWorld IS
  PACKAGE FLO_IO IS NEW TEXT_IO.FLOAT_IO(FLOAT);

      pi : constant := 3.1416;    -- value of pi
      radius : FLOAT;             -- radius of a pool - given
      circumference : FLOAT;      -- circumference of a pool -
                                  -- computed
      area : FLOAT;               -- area of a pool -
                                  -- computed
      taskToDo : CHARACTER;       -- holds user choice to
                                  -- compute circumference
                                  -- or area
```

©2016 Course Technology, a part of Cengage Learning.

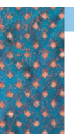


FIGURE 15

The SportsWorld Program (continued)

```
      more : CHARACTER;          -- controls loop for
                                 -- processing more pools
BEGIN
  TEXT_IO.PUT("Do you want to process a pool? (Y or N): ");
  TEXT_IO.GET(more);

  while more = 'Y'     -- more circles to process
   loop
    TEXT_IO.NEW_LINE;
    TEXT_IO.PUT("Enter the value of the radius of a " &
                "pool: ");
    FLO_IO.GET(radius);
                       -- See what user wants to compute
    TEXT_IO.PUT("Enter your choice of task.");
    TEXT_IO.NEW_LINE;
    TEXT_IO.PUT("C to compute circumference, " &
                "A to compute area: ");
    TEXT_IO.GET(taskToDo);

    if taskToDo = 'C'   -- compute circumference
     then
      circumference := 2.0 * pi * radius;
      TEXT_IO.PUT("The circumference for a pool " &
                  "of radius ");
      FLO_IO.PUT(radius, 3, 2, 0);
      TEXT_IO.PUT(" is ");
      FLO_IO.PUT(circumference, 3, 2, 0);
      TEXT_IO.NEW_LINE;
     else               -- compute area
      area := pi * radius ** 2;
      TEXT_IO.PUT("The area for a pool " &
                  "of radius ");
      FLO_IO.PUT(radius, 3, 2, 0);
      TEXT_IO.PUT(" is ");
      FLO_IO.PUT(area, 3, 2, 0);
      TEXT_IO.NEW_LINE;
    end if;
    TEXT_IO.NEW_LINE;
    TEXT_IO.PUT("Do you want to process more pools? " &
                "(Y or N): ");
    TEXT_IO.GET(more);
   end loop;              -- end of while loop
                          -- finish up
   TEXT_IO.NEW_LINE;
   TEXT_IO.PUT("Program will now terminate.");

END SportsWorld;
```


©2016 Course Technology, a part of Cengage Learning.

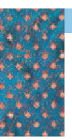


FIGURE 16

A Sample Session Using the Program of Figure 15

```
Do you want to process a pool? (Y or N): Y

Enter the value of the radius of a pool: 2.7
Enter your choice of task.
C to compute circumference, A to compute area: C
The circumference for a pool of radius    2.70 is  16.96

Do you want to process more pools? (Y or N): Y

Enter the value of the radius of a pool:   2.7
Enter your choice of task.
C to compute circumference, A to compute area: A
The area for a pool of radius   2.70 is  22.90

Do you want to process more pools? (Y or N): Y

Enter the value of the radius of a pool: 14.53
Enter your choice of task.
C to compute circumference, A to compute area: C
The circumference for a pool of radius  14.53 is  91.29

Do you want to process more pools? (Y or N): N

Program will now terminate.
```

PRACTICE PROBLEMS

1. Write a complete Ada program to read in an integer number and write out the square of that number.
2. Write a complete Ada program that asks for the price of an item and the quantity purchased, and writes out the total cost.
3. Write a complete Ada program that asks for a number. If the number is less than 5, it is written out, but if it is greater than or equal to 5, twice that number is written out.
4. Write a complete Ada program that asks the user for a positive integer *n* and then writes out all the numbers from 1 up to and including *n*.

5 Managing Complexity

The programs we have written have been relatively simple. More complex problems require more complex programs to solve them. Although it is fairly easy to understand what is happening in the 40 or so lines of the SportsWorld program, imagine trying to understand a program that is 50,000 lines long. Imagine trying to write such a program! It is not possible to understand—all at once—everything that goes on in a 50,000-line program.

©2016 Course Technology, a part of Cengage Learning.

5.1 *Divide and Conquer*

Writing large programs is an exercise in managing complexity. The solution is a problem-solving approach called **divide and conquer**. Suppose a program is to be written to do a certain task; let's call it task T. Suppose further that we can divide this task into smaller tasks, say A, B, C, and D, such that, if we can do those four tasks in the right order, we can do task T. Then our high-level understanding of the problem need only be concerned with *what* A, B, C, and D do and how they must work together to accomplish T. We do not, at this stage, need to understand *how* A, B, C, and D can be done. Figure 17(a), an example of a **structure chart** or **structure diagram**, illustrates this situation. Task T is composed in some way of subtasks A, B, C, and D. Later we can turn our attention to, say, subtask A and see if it too can be decomposed into smaller subtasks, as in Figure 17(b). In this way, we continue to break the task down into smaller and smaller pieces, finally arriving at subtasks that are simple enough that it is easy to write the code to carry them out. By *dividing* the problem into small pieces, we can *conquer* the complexity that is overwhelming if we look at the problem as a whole.

Divide and conquer is a problem-solving approach and not just a computer programming technique. Outlining a term paper into major and minor topics is a divide-and-conquer approach to writing the paper. Doing a Form 1040 Individual Tax Return for the Internal Revenue Service can involve the subtasks of completing Schedules A, B, C, D, and so on and then reassembling the results. Designing a house can be broken down into subtasks of designing floor plans, wiring, plumbing, and the like. Large companies organize their management responsibilities using a divide-and-conquer approach; what we have called structure charts become, in the business world, organization charts.

How is the divide-and-conquer problem-solving approach reflected in the resulting computer program? If we think about the problem in terms of subtasks, then the program should show that same structure; that is, part of the code should do subtask A, part should do subtask B, and so on. We divide the code into *modules* or *subprograms*, each of which does some part of the overall task. Then we empower these modules to work together to solve the original problem.

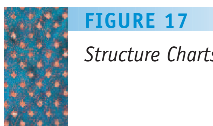
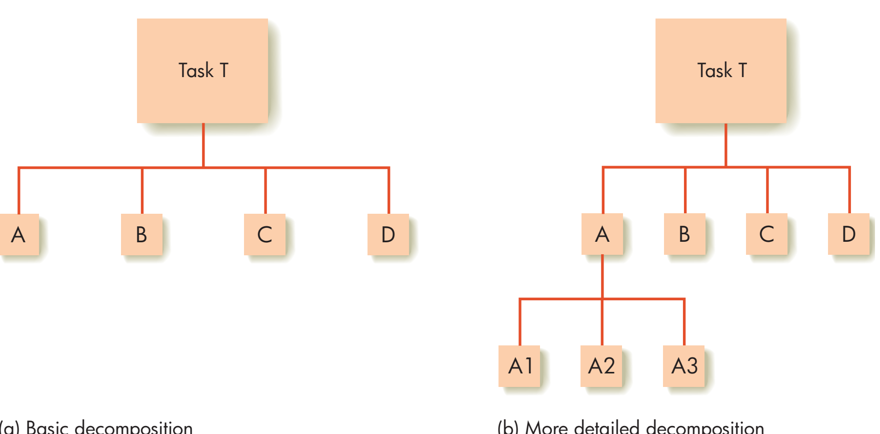


FIGURE 17

Structure Charts



©2016 Course Technology, a part of Cengage Learning.

5.2 *Using Functions/Procedures*

In Ada, modules of code are called either **functions** or **procedures**. These are the optional functions/procedures listed before the BEGIN keyword in the Ada program outline of Figure 2. One feature of the Ada language that differs from C-like languages (C, C++, C#, Java) is that functions and procedures are **nested** within one another. In Figure 2, the code for any optional functions/procedures is written inside the main procedure. And any of these functions or procedures could themselves contain code for other functions or procedures.

Each function/procedure in a program should do one and only one subtask. The distinction between a function and a procedure is the following: A function performs a subtask of computing one and only one value (similar to a mathematical function) and returning that single value for use by the rest of the program. A procedure carries out more general subtasks that may include returning multiple results (through a different mechanism than a function uses) for use by the rest of the program. For the moment, we'll continue to use the non-Ada term "module" so we don't have to worry just yet whether a module is an Ada function or an Ada procedure.

The executable part of a package body lies between the BEGIN and END statements; we are calling this the "main program code." When modules are used, the main program code can consist primarily of invoking these modules of code in the correct order. Let's review the main program code of the SportsWorld program (Figure 15) with an eye to further subdividing the task. In the main program code, there is a loop that does some operations as long as the user wants. What gets done? Input is obtained from the user about the radius of the circle and the choice of task to be done (compute circumference or compute area). Then the circumference or the area gets computed and written out. We've identified three subtasks, as shown in the structure chart of Figure 18.

We can visualize the main program code at a pseudocode level, as shown in Figure 19. This divide-and-conquer approach to solving the problem can (and should) be planned first in pseudocode, without regard to the details of the programming language to be used. If the three subtasks (input, circumference, area) can all be done, then arranging them within the structure of Figure 19 solves the problem. We can write a module for each of the subtasks. Although we now know what form the main program code will take, we have pushed the

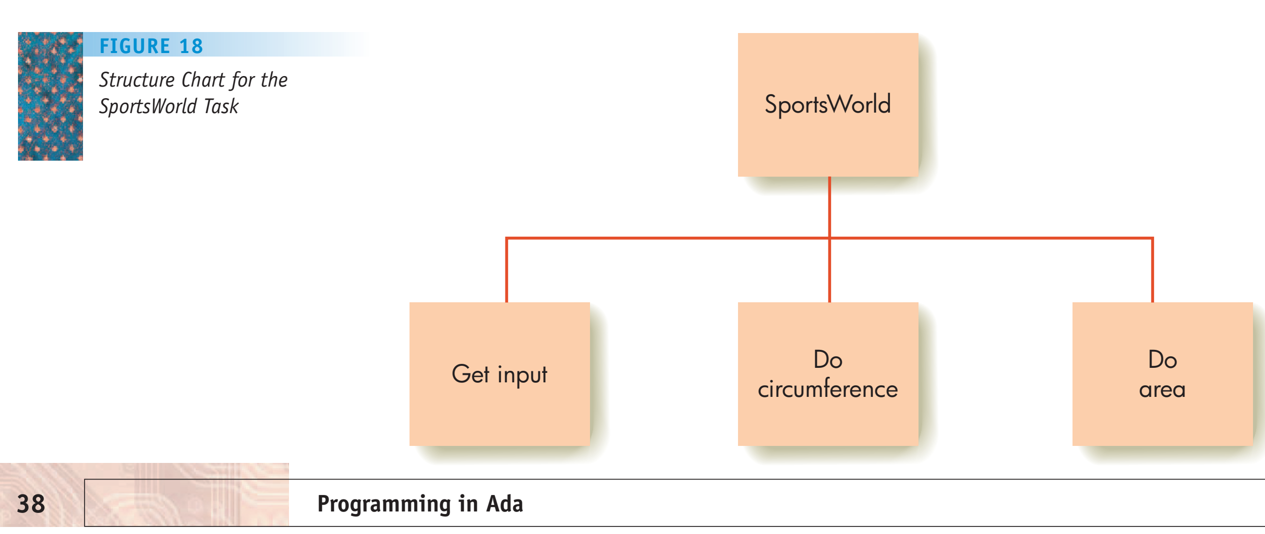


FIGURE 18
Structure Chart for the SportsWorld Task

©2016 Course Technology, a part of Cengage Learning.

FIGURE 19

A High-Level Modular View of the SportsWorld Program

```
Get value for user's choice about continuing
While the user wants to continue
  Do the input subtask
  If (Task = 'C') then
    do the circumference subtask
  else
    do the area subtask
Get value for user's choice about continuing
```

details of how to do each of the subtasks off into the other modules. Execution of the main program code starts after the BEGIN statement. Every time the flow of control reaches the equivalent of a "do subtask" instruction, it transfers execution to the appropriate module code. When execution of the module code is complete, flow of control returns to the main program code and picks up where it left off.

Before we look at the details of how to write a module, we need to examine the mechanism that allows the modules to work with each other and with the main program code. This mechanism consists of passing information about various quantities in the program back and forth between the modules and the main program code. Because each module is doing only one subtask of the entire task, it does not need to know the values of all variables in the program. It only needs to know the values of the variables with which its particular subtask is concerned. Allowing a module access to only pertinent variables prevents that module from inadvertently changing a value it has no business changing.

When the main program code wants a module to be executed, it gives the name of the module (which is an ordinary Ada identifier) and also a list of the identifiers for variables pertinent to that module. This is called an **argument list**. In our SportsWorld program, let's name the three modules *getInput*, *doCircumference*, and *doArea* (names that are descriptive of the subtasks these modules carry out). The *getInput* module collects the values for the variables *radius* and *taskToDo*. The main program code invokes the *getInput* module with the statement

```
getInput(radius, taskToDo);
```

which takes the place of the "Do the input subtask" line in Figure 19. When this statement is reached, control passes to the *getInput* module. After execution of this module, control returns to the main program code, and the variables *radius* and *taskToDo* have the values obtained for them within *getInput*.

The *doCircumference* module computes and writes out the value of the circumference, and, in order to do that, it needs to know the radius. Therefore, the variable *radius* is a legitimate argument for this module. The main program code contains the statement

```
doCircumference(radius);
```

in place of the "do the circumference subtask" line in Figure 19. When this statement is reached, the variable *radius* conveys the value of the radius to the

©2016 Course Technology, a part of Cengage Learning.

doCircumference module, which computes and writes out the circumference. The variable *circumference*, then, is also a variable of interest to the *doCircumference* module, but it is of interest to this module alone, in the sense that *doCircumference* does the computation and writes out the result. No other use is made of the circumference in the entire program, so no other module, nor the main program code, has anything to do with *circumference*. So now the variable *circumference* will be declared (and can be used) only within the *doCircumference* module; it will be **local** to that module. Any module can have its own **local constants** and **local variables**, declared within and known only to that module.

The *doCircumference* module also needs to know the value of the constant *pi*. We could declare *pi* as a constant local to *doCircumference,* but *doArea* needs the same constant, so we will place the declaration for *pi* above the code for any module. This will make *pi* a **global constant** whose value is known everywhere. The value of a constant cannot be changed, so there is no reason to prevent any function/procedure from having access to its value.

The *doArea* module computes and writes out the area and needs to know the radius, so the line "do the area subtask" in Figure 19 is replaced by

```
doArea(radius);
```

Within *doArea*, *area* is a local variable.

Now we can write the main program code of the modularized version of the SportsWorld program, shown in Figure 20. The main program code is a direct translation of Figure 19. If, in starting from scratch to write this program, we had taken a divide-and-conquer approach, broken the original problem down into three subtasks, and come up with the outline of Figure 19, it would have been easy to get from there to Figure 20. The only additional task would have been determining the variables needed.

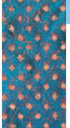


FIGURE 20

The Main Program Code in a Modularized Version of the SportsWorld Program

```
BEGIN
  TEXT_IO.PUT("Do you want to process a pool? (Y or N): ");
  TEXT_IO.GET(more);

  while more = 'Y'      -- more circles to process
   loop
    getInput(radius, taskToDo);
    if taskToDo = 'C'   -- compute circumference
     then
      doCircumference(radius);
     else               -- compute area
      doArea(radius);
    end if;
    TEXT_IO.NEW_LINE;
    TEXT_IO.PUT("Do you want to process more pools? (Y or N): ");
    TEXT_IO.GET(more);
   end loop;              -- end of while loop
                          -- finish up
   TEXT_IO.NEW_LINE;
   TEXT_IO.PUT("Program will now terminate.");

END SportsWorld;
```

©2016 Course Technology, a part of Cengage Learning.

At a glance, the main program code in Figure 20 does not look a great deal different from our former main program code. However, it is conceptually quite different; the subtasks of getting the input values, computing and writing out the circumference, and computing and writing out the area have been relegated to modules. The details (such as the formulas for computing circumference and area) are now hidden and have been replaced by module invocations. If these subtasks had required many lines of code, our new main program code would indeed be shorter—and easier to understand—than before.

5.3 *Writing Functions/Procedures*

Now we know how the main program code can invoke a module. (In fact, using the same process, any function/procedure can invoke another function/procedure. A function can even invoke itself.) It is time to see how to write the code for these functions/procedures. The general outline for an Ada function or an Ada procedure is shown in Figure 21.

The header (shown as the first two lines in Figure 21) consists of four parts:

- The keyword FUNCTION or PROCEDURE
- The function or procedure identifier
- A parameter list
- A return indicator (for a function)

The **return indicator** for a function indicates the data type of the one and only value computed and returned by the function. None of the three modules in the SportsWorld program does something as simple as computing and returning a single value: The *getInput* module has to prompt for and collect several input values and make these results available to the main program code. The *doCircumference* and *doArea* modules both compute single values, but they don't return them to the main program code; they write them as output. All three modules are Ada procedures.

The function or procedure identifier can be any legitimate Ada identifier. The parameters in the **parameter list** correspond to the arguments in the statement that invokes this function or procedure; that is, the first parameter in the list matches the first argument given in the statement that invokes the function or procedure, the second parameter matches the second argument, and so on. It is through this correspondence between parameters and arguments that information (data) flows from the main program code to other modules, and vice versa. The data type of each parameter must be given as

FIGURE 21
The Outline for an Ada Function/Procedure

```
FUNCTION function name (parameter list)
  return type is
local declarations   [optional]
BEGIN
     function body
     return . . .
END name
```

```
PROCEDURE procedure name
  (parameter list) is
local declarations   [optional]
BEGIN
     procedure body
END name
```


©2016 Course Technology, a part of Cengage Learning.

part of the parameter list, and it must match the data type of the corresponding argument. For example, because the *getInput* procedure is invoked with the two arguments *radius* and *taskToDo*, the parameter list for the *getInput* header has two parameters, the first of type FLOAT and the second of type CHARACTER. Parameters may have, but do not have to have, the same identifiers as the corresponding arguments; arguments and parameters correspond by virtue of their respective positions in the argument list and the parameter list, regardless of the identifiers used. For the *getInput* procedure, we choose the parameter identifiers *radius* and *taskToDo*, matching the argument identifiers. No semicolon is used at the end of a procedure/function header; the delimiter is the word "IS".

One additional aspect of the parameter list in the header concerns the use the module will make of each parameter. Consider the statement that invokes the module; an argument in the invoking statement carries a data value to the corresponding parameter in the header. If the value is one that the module must know to do its job but should not change, then the argument is **passed by value**. The module receives a copy of the data value for its use but can make no changes to that value. If, however, the value passed to the module is one that the module should change, and the main program code should know the new value, then the argument is **passed by reference**. The module receives access to the memory location where the value is stored, and any changes it makes to the value are seen by the main program code after control returns there.

By default, arguments in Ada are passed by value (the default can also be denoted by the keyword ***in*** next to the corresponding parameter names), which protects them from change by the module. An *in* parameter cannot appear on the left side of an assignment statement. Explicit action must be taken by the programmer to pass an argument by reference; specifically, the keywords ***out*** or ***in out*** must appear in front of the corresponding parameter data type in the module parameter list. An *out* argument does not have a value before the module invocation. Its corresponding *out* parameter is used solely to obtain a value within the module, and, without any further use being made of it within the module, the new value is sent back to the argument in the invoking statement. An *in out* parameter receives a value that can be both used and modified within the module, and the modified value is then sent back to the invoking statement.

How do we decide whether to pass an argument by value or by reference? If the main program code needs to obtain a new value back from a module when execution of that module terminates, then the argument must be passed by reference (by inserting the *out* or *in out* into the parameter list). Otherwise, the argument should be passed by value, the default arrangement (either use *in* or don't indicate anything, since *in* is the default).

In the *getInput* procedure, both *radius* and *taskToDo* have no assigned values when passed into the procedure. The task of the *getInput* procedure is to obtain values for these variables from the user that the main program code will get back when *getInput* terminates, so both of these arguments are passed by reference (using *out*). The header for the *getInput* procedure is shown below, along with the invoking statement from the main program code. Note that the parameters *radius* and *taskToDo* are in the right order, have been given the correct data types, and are marked for passing by reference. Also remember that, although the arguments are named *radius* and *taskToDo* because those are the variable identifiers used in the main program code, the

©2016 Course Technology, a part of Cengage Learning.

parameters could have different identifiers, and it is the parameter identifiers that are used within the body of the procedure.

```
-- header
PROCEDURE getInput(radius : out FLOAT;
                              taskToDo : out
                              CHARACTER) IS

-- invocation
getInput(radius, taskToDo);
```

The body of the *getInput* procedure comes from the corresponding part of Figure 15. If we hadn't already written this code, we could have done a pseudocode plan first. The complete procedure appears in Figure 22, where a comment has been added to document the purpose of the procedure.

The *doCircumference* procedure needs to know the value of *radius* but does not change that value. Therefore, *radius* is passed by value. Why is the distinction between arguments passed by value and those passed by reference important? If modules are to effect any changes at all, then clearly reference parameters are necessary, but why not just make everything a reference parameter? Suppose that in this example *radius* is made a reference parameter. If an instruction within *doCircumference* were to inadvertently change the value of *radius*, then that new value would be returned to the main program code, and any subsequent calculations using this value (there are none in this example) would be in error. Making *radius* a value parameter prevents this. How could one possibly write a program statement that changes the value of a variable inadvertently? In something as short and simple as our example, this probably would not happen, but in a more complicated program, it might. Distinguishing between passing by value and passing by reference is just a further step in controlling a module's access to data values, to limit the damage the module might do. The code for the *doCircumference* procedure appears in Figure 23.

The *doArea* procedure is very similar. Let's reassemble everything and give the complete modularized version of the program. In Figure 24, only the main program code needs to know the value of *more*. No other procedure needs

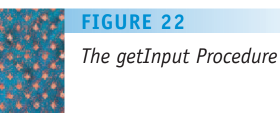


FIGURE 22

The getInput Procedure

```
PROCEDURE getInput(radius : out FLOAT; taskToDo : out
  CHARACTER) IS
  -- gets radius and choice of task from the user
  BEGIN
    TEXT_IO.NEW_LINE;
    TEXT_IO.PUT("Enter the value of the radius of a " &
                "pool: ");
    FLO_IO.GET(radius);
                       -- See what user wants to compute
    TEXT_IO.PUT("Enter your choice of task.");
    TEXT_IO.NEW_LINE;
    TEXT_IO.PUT("C to compute circumference, " &
                "A to compute area: ");
    TEXT_IO.GET(taskToDo);
  END getInput;
```


©2016 Course Technology, a part of Cengage Learning.

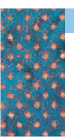


FIGURE 23

The doCircumference Procedure

```
PROCEDURE doCircumference(radius : in FLOAT) IS
-- computes and writes out the circumference of
-- a circle with given radius

  circumference : FLOAT;     -- circumference of a pool -
                             -- computed
 BEGIN
   circumference := 2.0 * pi * radius;
   TEXT_IO.PUT("The circumference for a pool " &
               "of radius ");
   FLO_IO.PUT(radius, 3, 2, 0);
   TEXT_IO.PUT(" is ");
   FLO_IO.PUT(circumference, 3, 2, 0);
   TEXT_IO.NEW_LINE;
 END doCircumference;
```

access to this value, so this variable is never passed as an argument. In Ada, the code for optional procedures (or functions) is nested within other procedures or functions. In Figure 24, the code for each of the three procedures *getInput*, *doCircumference*, and *doArea* is nested within the declarative portion of the main procedure. The main procedure header

```
PROCEDURE SportsWorld IS
```

also follows the form for a procedure header. In other words, the main procedure truly is an Ada procedure. It has an empty parameter list because it is the starting point for the program, and there's no other place that could pass argument values to it.

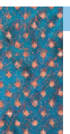


FIGURE 24

The Complete Modularized SportsWorld Program

```
-- This program helps SportsWorld estimate costs
-- for pool covers and pool fencing by computing
-- the area or circumference of a circle
-- with a given radius.
-- Any number of circles can be processed.
-- Written by M. Phelps, 10/05/16

WITH TEXT_IO;

PROCEDURE SportsWorld IS
 PACKAGE FLO_IO IS NEW TEXT_IO.FLOAT_IO(FLOAT);

      pi : constant := 3.1416;   -- value of pi

 PROCEDURE getInput(radius : out FLOAT; taskToDo : out
   CHARACTER) IS
  -- gets radius and choice of task from the user
  BEGIN
    TEXT_IO.NEW_LINE;
    TEXT_IO.PUT("Enter the value of the radius of a " &
                "pool: ");
```

©2016 Course Technology, a part of Cengage Learning.

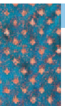


FIGURE 24

The Complete Modularized SportsWorld Program (continued)

```
     FLO_IO.GET(radius);
                           -- See what user wants to compute
     TEXT_IO.PUT("Enter your choice of task.");
     TEXT_IO.NEW_LINE;
     TEXT_IO.PUT("C to compute circumference, " &
                 "A to compute area: ");
     TEXT_IO.GET(taskToDo);
   END getInput;

  PROCEDURE doCircumference(radius : in FLOAT) IS
  -- computes and writes out the circumference of
  -- a circle with given radius

    circumference : FLOAT;     -- circumference of a pool -
                               -- computed
   BEGIN
     circumference := 2.0 * pi * radius;
     TEXT_IO.PUT("The circumference for a pool " &
                 "of radius ");
     FLO_IO.PUT(radius, 3, 2, 0);
     TEXT_IO.PUT(" is ");
     FLO_IO.PUT(circumference, 3, 2, 0);
     TEXT_IO.NEW_LINE;
   END doCircumference;

  PROCEDURE doArea(radius : in FLOAT) IS
  -- computes and writes out the area of
  -- a circle with given radius

    area : FLOAT;              -- area of a pool -
                               -- computed

   BEGIN
     area := pi * radius ** 2;
     TEXT_IO.PUT("The area for a pool " &
                 "of radius ");
     FLO_IO.PUT(radius, 3, 2, 0);
     TEXT_IO.PUT(" is ");
     FLO_IO.PUT(area, 3, 2, 0);
     TEXT_IO.NEW_LINE;
   END doArea;

       radius : FLOAT;            -- radius of a pool - given
       taskToDo : CHARACTER;      -- holds user choice to
                                  -- compute circumference
                                  -- or area
       more : CHARACTER;          -- controls loop for
                                  -- processing more pools
 BEGIN
   TEXT_IO.PUT("Do you want to process a pool? (Y or N): ");
   TEXT_IO.GET(more);
```


©2016 Course Technology, a part of Cengage Learning.

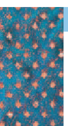


FIGURE 24

The Complete Modularized SportsWorld Program (continued)

```
   while more = 'Y'     -- more circles to process
    loop
     getInput(radius, taskToDo);
     if taskToDo = 'C'   -- compute circumference
      then
       doCircumference(radius);
      else               -- compute area
       doArea(radius);
     end if;
     TEXT_IO.NEW_LINE;
     TEXT_IO.PUT("Do you want to process more pools?" &
                 " (Y or N): ");
     TEXT_IO.GET(more);
    end loop;              -- end of while loop
                           -- finish up
    TEXT_IO.NEW_LINE;
    TEXT_IO.PUT("Program will now terminate.");

END SportsWorld;
```

Because it seems to have been a lot of effort to arrive at this complete, modularized version of our SportsWorld program (which, after all, does the same thing as the program in Figure 15), let's review why this effort is worthwhile.

The modularized version of the program is compartmentalized in two ways. First, it is compartmentalized with respect to task. The major task is accomplished by a series of subtasks, and the work for each subtask takes place within a separate module. This leaves the main program code free of details and consisting primarily of invoking the appropriate procedures at the appropriate points. As an analogy, think of the president of a company calling on various assistants to carry out tasks as needed. The president does not need to know *how* a task is done, only the name of the person responsible for carrying it out. Second, the program is compartmentalized with respect to data, in the sense that the data values known to the various modules are controlled by parameter lists and by the use of value instead of reference parameters where appropriate. In our analogy, the president gives each assistant the information he or she needs to do the assigned task, and expects relevant information to be returned—but not all assistants know all information.

This compartmentalization is useful in many ways. It is useful when we *plan the solution* to a problem, because it allows us to use a divide-and-conquer approach. We can think about the problem in terms of subtasks. This makes it easier for us to understand how to achieve a solution to a large and complex problem. It is also useful when we *code the solution* to a problem, because it allows us to concentrate on writing one section of the code at a time. We can write a module and then fit it into the program, so that the program gradually expands rather than having to be written all at once. Developing a large software project is a team effort, and different parts of the team can be writing different modules at the same time. It is useful when we *test the program*, because we can test one new module at a time as the program grows, and any errors are localized to the module being added. (The main program code can be tested early by writing appropriate headers but empty bodies for the remaining modules.) Compartmentalization is useful when we *modify the program*, because changes tend to be within certain subtasks and

©2016 Course Technology, a part of Cengage Learning.

hence within certain modules in the code. And finally it is useful for anyone (including the programmer) who wants to *read* the resulting program. The overall idea of how the program works, without the details, can be gleaned from reading the main program code; if and when the details become important, the appropriate code for the other modules can be consulted. In other words, modularizing a program is useful for its

- Planning
- Coding
- Testing
- Modifying
- Reading

A special type of Ada module can be written to compute a single value as its subtask. In Ada this is called a **function**. For example, the *doCircumference* procedure does everything connected with the circumference, both calculating the value and writing it out. We can write a *doCircumference* function that only computes the value of the circumference and then returns that value to the main program code, which writes it out. Instead of the word PROCEDURE, the keyword FUNCTION is used in the header. Also the keyword RETURN is added, along with the data type of the value to be returned. In addition, a function must contain a return statement, which consists of the keyword **return** followed by an expression for the value to be returned. (This explains why we have always written the main module as a procedure; it is never invoked anywhere else in the program and does not return a value.) A function may need data values passed into it to compute the value it returns, but it should not do anything besides this one computation—in particular, it should not change the data values it receives—so all arguments to the function should be passed by value.

The code for this new *doCircumference* function would be simply

```
FUNCTION doCircumference(radius : in FLOAT) RETURN
   FLOAT IS
-- computes the circumference of a circle with
-- given radius
 BEGIN
   return 2.0 * pi * radius;
 END doCircumference;
```

A function is invoked wherever the returned value is to be used, rather than in a separate statement. For example, the statement

```
FLO_IO.PUT(doCircumference(radius), 3, 2, 0);
```

invokes the *doCircumference* function by giving its name and argument, and this invocation actually becomes the value returned by the *doCircumference* function, which is then written out.

Figure 25 shows a third version of the SportsWorld program using functions for *doCircumference* and *doArea*.

An Ada package will often be split into separately compiled files. For the program of Figure 25 there could be a package called *SportsWorldFunctions*

©2016 Course Technology, a part of Cengage Learning.

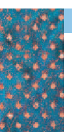


FIGURE 25

The SportsWorld Program Using Functions

```
-- This program helps SportsWorld estimate costs
-- for pool covers and pool fencing by computing
-- the area or circumference of a circle
-- with a given radius.
-- Any number of circles can be processed.
-- Written by M. Phelps, 10/05/16

WITH TEXT_IO;

PROCEDURE SportsWorld IS
 PACKAGE FLO_IO IS NEW TEXT_IO.FLOAT_IO(FLOAT);

      pi : constant := 3.1416;    -- value of pi

 PROCEDURE getInput(radius : out FLOAT; taskToDo : out
 CHARACTER) IS
  -- gets radius and choice of task from the user
  BEGIN
    TEXT_IO.NEW_LINE;
    TEXT_IO.PUT("Enter the value of the radius of a " &
                "pool: ");
    FLO_IO.GET(radius);
                       -- See what user wants to compute
    TEXT_IO.PUT("Enter your choice of task.");
    TEXT_IO.NEW_LINE;
    TEXT_IO.PUT("C to compute circumference, " &
                "A to compute area: ");
    TEXT_IO.GET(taskToDo);
  END getInput;

 FUNCTION doCircumference(radius : in FLOAT) RETURN FLOAT IS
 -- computes the circumference of a circle with given radius
  BEGIN
    return 2.0 * pi * radius;
  END doCircumference;

 FUNCTION doArea(radius : in FLOAT) RETURN FLOAT IS
 -- computes the area of a circle with given radius
  BEGIN
    return pi * radius ** 2;
  END doArea;
      radius : FLOAT;             -- radius of a pool - given
      taskToDo : CHARACTER;       -- holds user choice to
                                  -- compute circumference
                                  -- or area
      more : CHARACTER;           -- controls loop for
                                  -- processing more pools
BEGIN
  TEXT_IO.PUT("Do you want to process a pool? (Y or N): ");
  TEXT_IO.GET(more);
```


©2016 Course Technology, a part of Cengage Learning.

C8814_chapter_ada.qxd 1/17/15 5:27 PM Page 49

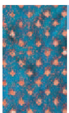


FIGURE 25

The SportsWorld Program Using Functions (continued)

```
   while more = 'Y'       -- more circles to process
    loop
     getInput(radius, taskToDo);
     if taskToDo = 'C'    -- compute circumference
      then
       TEXT_IO.PUT("The circumference for a pool " &
                   "of radius ");
       FLO_IO.PUT(radius, 3, 2, 0);
       TEXT_IO.PUT(" is ");
       FLO_IO.PUT(doCircumference(radius), 3, 2, 0);
       TEXT_IO.NEW_LINE;
      else                -- compute area
       TEXT_IO.PUT("The area for a pool " &
                   "of radius ");
       FLO_IO.PUT(radius, 3, 2, 0);
       TEXT_IO.PUT(" is ");
       FLO_IO.PUT(doArea(radius), 3, 2, 0);
       TEXT_IO.NEW_LINE;
     end if;
     TEXT_IO.NEW_LINE;
     TEXT_IO.PUT("Do you want to process more pools? (Y or N): ");
     TEXT_IO.GET(more);
    end loop;             -- end of while loop
                          -- finish up
    TEXT_IO.NEW_LINE;
    TEXT_IO.PUT("Program will now terminate.");

 END SportsWorld;
```

and a package called *SportsWorld*. The *SportsWorldFunctions* package would be divided into the package specification and the package body. The package specification would show the function/procedure headers, thereby giving information about the capabilities of the package without the details of how these capabilities are implemented. The implementations would occur in the *SportsWorldFunction* package body. The *SportsWorld* package would consist primarily of the variable declarations for *radius*, *taskToDo*, and *more*, the main program code, with function and procedure invocations as before, plus the appropriate WITH statement to give definitions to these functions:

```
WITH SportsWorldFunctions;  -- gain access to the
                            -- functions
```

The separation of the functions into a separate package allows these functions to be used in other programs in addition to *SportsWorld*. The only thing other programs need to see is the package specification for the functions. There is no need to make the source code for the functions (in the package body) available to the other program. Also, if a new and better implementation for the functions is created, no change needs to be made to the code for a program that uses these functions, because the specification does not change.

Figure 26 summarizes several sets of terms introduced in this section.

©2016 Course Technology, a part of Cengage Learning.

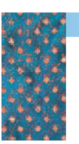


FIGURE 26

Some Ada Terminology

Term	Meaning	Term	Meaning
Local variable	Declared and known only within a function or procedure.	Global constant	Declared before any function or procedure and known everywhere.
Argument passed by value	Function or procedure receives a copy of the value and can make no changes in the value. Designate corresponding parameter using *in*.	Argument passed by reference	Function or procedure gains access to memory location where the value is stored; changes made to the value persist after control returns to the invoking module. Designate corresponding parameter using *out* or *in out*.
Function	Computes a single value and returns it via a return statement. Invocation is within another statement.	Procedure	Performs a task, perhaps returns multiple values. Invocation is a complete Ada statement.

PRACTICE PROBLEMS

1. What is the output of the following Ada program?

```
WITH TEXT_IO;

PROCEDURE PracticeProblem IS
 PACKAGE INT_IO IS NEW TEXT_IO.INTEGER_IO (INTEGER);

PROCEDURE doIt(number : in out INTEGER) IS
 BEGIN
   number := number + 4;
 END doIt;

    number : INTEGER;

BEGIN
 number := 7;
 doIt(number);
 TEXT_IO.put("Answer: ");
 INT_IO.PUT(number);
 TEXT_IO.NEW_LINE;
END PracticeProblem;
```

2. Ada will not allow this code to compile. Why?

```
WITH TEXT_IO;

PROCEDURE PracticeProblem IS
 PACKAGE INT_IO IS NEW TEXT_IO.INTEGER_IO(INTEGER);

PROCEDURE doIt(number : INTEGER) IS
 BEGIN
   number := number + 4;
 END doIt;

    number : INTEGER;
```

©2016 Course Technology, a part of Cengage Learning.

PRACTICE PROBLEMS

```
BEGIN
 number := 7;
 doIt(number);
 TEXT_IO.PUT("Answer: ");
 INT_IO.PUT(number);
 TEXT_IO.NEW_LINE;
END PracticeProblem;
```

3. Write an Ada procedure that performs an input task for the main program code, collecting two integer values *one* and *two* from the user.

4. Suppose a function called *tax* gets a value *subTotal* from the main program code, multiplies it by a constant tax rate called *rate*, and returns the resulting tax value. All quantities are type FLOAT.
   a. Write the function header.
   b. Write the return statement in the function body.
   c. Write the statement in the main program code that writes out the tax.

5.4 *An Ada Feature: User-Defined Subtypes*

We've already mentioned that Ada is a strongly-typed language and that if you want to multiply, say, a FLOAT quantity by an INTEGER quantity, you have to do an explicit type cast to one of the two operands. Ada has another feature that most languages lack—you can define your own subtypes of the Ada standard data types, and strong typing will apply to these new types as well. In Figure 27, a new version of the SportsWorld program, note the sections of code in boldface type. In previous versions, radius, circumference, and area have all been declared as type FLOAT. This version recognizes that radius and circumference are measured in feet, while area has units of square feet. The first two boldface lines in Figure 27 define two new types based on FLOAT. One is called *Feet*, the other *SquareFeet*. The strong typing in Ada requires that appropriate versions of the I/O package be used for variables declared of these new types, so the next two boldface lines create the appropriate versions of *TEXT_IO.FLOAT_IO*. These are variations of line 9 in Figure 3. Then these types are used in the declaratives for radius and circumference (type *Feet*), and area (type *SquareFeet*). For the most part, the remainder of the code is unchanged except for the I/O statements, which are also shown in bold. The key to this example is the following line.

```
area := pi * SquareFeet(radius ** 2);
```

The variable radius has type *Feet*. When it is squared, the units must be converted to square feet. The "conversion" is carried out by the type casting operation performed by SquareFeet( ). As a "pure" number, *pi* has no units. It is a constant and

©2016 Course Technology, a part of Cengage Learning.

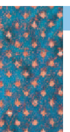


FIGURE 27

The SportsWorld Program with Defined Subtypes

```
-- This program helps SportsWorld estimate costs
-- for pool covers and pool fencing by computing
-- the area or circumference of a circle
-- with a given radius.
-- Any number of circles can be processed.
-- Written by M. Phelps, 10/05/16

WITH TEXT_IO;

PROCEDURE SportsWorld IS

   TYPE Feet is NEW FLOAT;        -- new data types
   TYPE SquareFeet is NEW FLOAT;
   PACKAGE FLO_IO_Feet IS NEW TEXT_IO.FLOAT_IO(Feet);
   PACKAGE FLO_IO_SquareFeet IS NEW
     TEXT_IO.FLOAT_IO(SquareFeet);

      pi : constant := 3.1416;    -- value of pi
      radius : Feet;              -- radius of a pool - given
      circumference : Feet;       -- circumference of a pool -
                                  -- computed
      area : SquareFeet;          -- area of a pool -
                                  -- computed
      taskToDo : CHARACTER;       -- holds user choice to
                                  -- compute circumference
                                  -- or area
      more : CHARACTER;           -- controls loop for
                                  -- processing more pools

BEGIN
  TEXT_IO.PUT("Do you want to process a pool? (Y or N): ");
  TEXT_IO.GET(more);

  while more = 'Y'     -- more circles to process
   loop
    TEXT_IO.NEW_LINE;
    TEXT_IO.PUT("Enter the value of the radius of a " &
                "pool: ");
    FLO_IO_Feet.GET(radius);
                       -- See what user wants to compute
    TEXT_IO.PUT("Enter your choice of task.");
    TEXT_IO.NEW_LINE;
    TEXT_IO.PUT("C to compute circumference, " &
                "A to compute area: ");
    TEXT_IO.GET(taskToDo);

    if taskToDo = 'C'   -- compute circumference
     then
```

©2016 Course Technology, a part of Cengage Learning.

FIGURE 27

The SportsWorld Program with Defined Subtypes (continued)

```
        circumference := 2.0 * pi * radius;
        TEXT_IO.PUT("The circumference for a pool " &
                    "of radius ");
        FLO_IO_Feet.PUT(radius, 3, 2, 0);
        TEXT_IO.PUT(" is ");
        FLO_IO_Feet.PUT(circumference, 3, 2, 0);
        TEXT_IO.NEW_LINE;
       else                  -- compute area
        area := pi * SquareFeet(radius ** 2);
        TEXT_IO.PUT("The area for a pool " &
                    "of radius ");
        FLO_IO_Feet.PUT(radius, 3, 2, 0);
        TEXT_IO.PUT(" is ");
        FLO_IO_SquareFeet.PUT(area, 3, 2, 0);
        TEXT_IO.NEW_LINE;
      end if;
      TEXT_IO.NEW_LINE;
      TEXT_IO.PUT("Do you want to process more pools? (Y or N): ");
      TEXT_IO.GET(more);
     end loop;               – end of while loop
                             – finish up
     TEXT_IO.NEW_LINE;
     TEXT_IO.PUT("Program will now terminate.");

  END SportsWorld;
```

When It Absolutely, Positively Has To Be Right

We have seen that Ada has some rather rigid syntax requirements—for example, the strict typing that does not allow "mixed mode" arithmetic (between types INTEGER and FLOAT, for example, or between types *Feet* and *SquareFeet*), and the requirement that arguments to functions and procedures be regulated by their use as specified by the *in*, *out*, and *in out* parameter markers. The result is that it can take many trials to get a successful compilation of a program, but that once it compiles, the chances for error are reduced. This makes Ada the language of choice for safety-critical software such as high-speed-train control, air-traffic control, nuclear reactor monitoring, and so forth.

The European Space Agency launched the Ariane 5 rocket for the first time on June 4, 1996, only to see it explode less than 40 seconds after takeoff. The failure was traced to the Inertial Reference System software, which calculates angles and velocities that are ultimately used to execute the flight program. This software was carried over almost intact from the earlier Ariane 4, but did not take into account the greater horizontal velocity of the Ariane 5 flight path. Specifically, during data conversion from a 64-bit floating point to a 16-bit signed integer, an arithmetic overflow occurred because the floating-point number was too large to fit into 16 bits. This ultimately led to a shutdown of the entire system and the subsequent explosion. For efficiency reasons, the "exception" that Ada automatically raised because of this situation—which should have been addressed by writing code called an exception handler—had been disabled. The specific line of code used was:

```
pragma suppress(numeric_error,
    horizontal_veloc_bias);
```

We won't explain this in detail, but it basically assures the compiler that the condition raising this exception will never occur, so the compiler need not check for it. The Ariane 5 stands as one of the more spectacular software failures on record.

©2016 Course Technology, a part of Cengage Learning.

has inherited its data type from the numeric value 3.1416—presumably FLOAT, which is compatible with the *SquareFeet* type, so the whole right-side expression is now type *SquareFeet*. That is what is expected by the assignment operation, since area is of type *SquareFeet*.

The point is that if a line of code like

```
area := circumference;
```

were to appear in this program, it would cause an error at compile time because of the data type mismatch. Using user-defined types allows many checks to be made at compile time so that the resulting code is as correct as possible when execution occurs.

6 Object-Oriented Programming

6.1 *What Is It?*

The divide-and-conquer approach to programming is a traditional approach. The focus is on the overall task to be done: how to break it down into subtasks, and how to write algorithms for these subtasks that are carried out by communicating modules (in the case of Ada, by functions and procedures). The program can be thought of as a giant statement executor designed to carry out the major task, even though the main program code may simply call on, in turn, the various other modules that do the subtask work.

Object-oriented programming (OOP) takes a somewhat different approach. A program is considered a simulation of some part of the world that is the domain of interest. "Objects" populate this domain. Objects in a banking system, for example, might be savings accounts, checking accounts, and loans. Objects in a company personnel system might be employees. Objects in a medical office might be patients and doctors. Each object is an example drawn from a class of similar objects. The savings account "class" in a bank has certain properties associated with it, such as name, Social Security number, account type, and account balance. Each individual savings account at the bank is an example of (an object of) the savings account class, and each has specific values for these common properties; that is, each savings account has a specific value for the name of the account holder, a specific value for the account balance, and so forth. Each object of a class therefore has its own data values.

So far, this is similar to the idea of a data type in Ada; in the SportsWorld program, *radius*, *circumference*, and *area* are all examples (objects) from the data type (class) FLOAT; the class has one property (a numeric quantity), and each object has its own specific value for that property. However, in object-oriented programming, a class also has one or more subtasks associated with it, and all objects from that class can perform those subtasks. In carrying out a subtask, each object can be thought of as providing some service. A savings account, for example, can compute compound interest due on the balance. When an object-oriented program is executed, the program generates requests for services that go to the various objects. The objects respond by performing the requested service—that is, carrying out the subtask. Thus, a program that

©2016 Course Technology, a part of Cengage Learning.

is using the savings account class might request a particular savings account object to perform the service of computing interest due on its account balance. An object always knows its own data values and may use them in performing the requested service.

There are three terms often associated with object-oriented programming, as illustrated in Figure 28. The first term is **encapsulation**. Each class has its own program module to perform each of its subtasks. Any user of the class (which might be some other program) can ask an object of that class to invoke the appropriate module and thereby perform the subtask service. The class user needs to know what services objects of the class can provide and how to request an object to perform any such service. The details of the module code belong to the class itself, and this code may be modified in any manner, as long as the way the user interacts with the class remains unchanged. (In the savings account example, the details of the algorithm used to compute interest due belong only to the class, and need not be known by any user of the class. If the bank wants to change how it computes interest, only the code for the interest module in the savings account class needs to be modified; any programs that use the services of the savings account class can remain unchanged.) Furthermore, the class properties represent data values that will exist as part of each object of the class. A class therefore consists of two components, its subtask modules and its properties, and both components are encapsulated—bundled—with the class.

A second term associated with object-oriented programming is **inheritance**. Once a class A of objects is defined, a class B of objects can be defined as a "subclass" of A. Every object of class B is also an object of class A; this is sometimes called an "is a" relationship. Objects in the B class will "inherit" all of the properties and be able to perform all the services of objects in A, but they may also be given some special property or ability. The benefit is that class B does not have to be built from the ground up, but rather can take advantage of the fact that class A already exists. In the banking example, a senior citizens savings account would be a subclass of the savings account class. Any senior citizens savings account object is also a savings account object, but may have special properties or be able to provide special services.

The third term is **polymorphism**. *Poly* means "many." Objects of different classes may provide services that should logically have the same name because they do roughly the same thing, but the details differ. In the banking

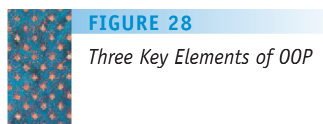
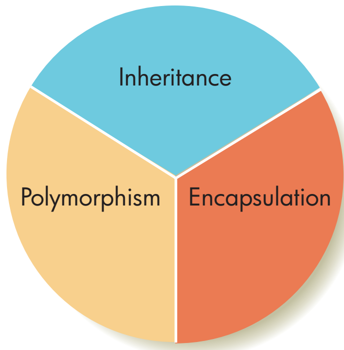


FIGURE 28

Three Key Elements of OOP



©2016 Course Technology, a part of Cengage Learning.

example, both savings account objects and checking account objects should provide a "compute interest" service, but the details of how interest is computed differ in these two cases. Thus, one name, the name of the service to be performed, has several meanings, depending on the class of the object providing the service. It may even be the case that more than one service with the same name exists for the same class, although there must be some way to tell which service is meant when it is invoked by an object of that class.

Let's change analogies from the banking world to something more fanciful, and consider a football team. Every member of the team's backfield is an "object" of the "backfield" class. The quarterback is the only "object" of the "quarterback" class. Each backfield object can perform the service of carrying the ball if he (or she) receives the ball from the quarterback; ball carrying is a subtask of the backfield class. The quarterback who hands the ball off to a backfield object is requesting that the backfield object perform that subtask because it is "public knowledge" that the backfield class carries the ball and that this service is invoked by handing off the ball to a backfield object. The "program" to carry out this subtask is *encapsulated* within the backfield class, in the sense that it may have evolved over the week's practice and may depend on specific knowledge of the opposing team, but at any rate, its details need not be known to other players. *Inheritance* can be illustrated by the halfback subclass within the backfield class. A halfback object can do everything a backfield object can but may also be a pass receiver. And *polymorphism* can be illustrated by the fact that the backfield may invoke a different "program" depending on where on the field the ball is handed off. Of course our analogy is imperfect, because not all human "objects" from the same class behave in precisely the same way—fullbacks sometimes receive passes and so on.

6.2 Ada and OOP

In February 1995, Ada 95, a revision of the original Ada programming language, became the first internationally standardized object-oriented programming language. The new standard, officially ISO/IEC 8652:1995, added many new and important features to the language.[2] While there is no single mechanism for a class in Ada 95, Ada 95 packages can be constructed to provide the analogous behavior. Figure 29 compares standard object-oriented terminology and Ada 95 terminology.

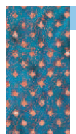


FIGURE 29

Object-Oriented Terminology and Usage: Standard and Ada

Standard Object-Oriented Terminology and Usage	Ada 95 Terminology and Usage
Method	Primitive operation: procedure or function defined after the tagged record in the definition package
Class	Tagged Record Type
Reference to object property: objectName.property	objectName.property
Message (request for service): objectName.method(arguments)	PrimitiveOperation(objectName, arguments)

[2]The newest international Ada standard was adopted in 2012.

©2016 Course Technology, a part of Cengage Learning.

How do these ideas get translated into real programs? Let's rewrite the SportsWorld program one more time, this time using a more object-oriented approach. What are the objects of interest within the scope of this problem? SportsWorld deals with circular swimming pools, but they are basically just circles. In Ada the first step in this process is to create a new type, something called a *tagged record type*. The object-oriented code for SportsWorld will look similar to the previous function-based code, but the tagged record type will allow the type to be extended to cover other object-oriented ideas, e.g., inheritance. So, let's create a *CIRCLE* type, and have the SportsWorld program create objects of (**instances of**) that type. The objects are individual circles. A CIRCLE object has a radius. A CIRCLE object, which knows the value of its own radius, should be able to perform the services of computing its own circumference and its own area. At this point, we have answered the two major questions about our *CIRCLE* type:

- What are the properties common to any object of this type? (In this case, there is a single property—the radius.)
- What are the services that any object of the type should be able to perform? (In this case, it should be able to compute its circumference and compute its area, although as we will see shortly, we will need other services as well.)

Figure 30 shows the complete object-oriented version of SportsWorld, with its new type *CIRCLE*. The type *CIRCLE* has the single property *radius*, and four

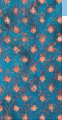


FIGURE 30

An Object-Oriented SportsWorld Program

```
-- This program helps SportsWorld estimate costs
-- for pool covers and pool fencing by computing
-- the area or circumference of a circle
-- with a given radius.
-- Any number of circles can be processed.
-- Written by M. Phelps, 10/05/16

WITH TEXT_IO;

PROCEDURE SportsWorld IS
 PACKAGE FLO_IO IS NEW TEXT_IO.FLOAT_IO(FLOAT);

  pi : constant := 3.1416;   — value of pi

  TYPE CIRCLE IS
    tagged record
       radius:  FLOAT;
    end record;

 PROCEDURE setRadius(obj: in out CIRCLE; value : in FLOAT) IS
  -- sets radius equal to value
  BEGIN
    obj.radius := value;
  END setRadius;

 FUNCTION getRadius(obj: in CIRCLE) RETURN FLOAT IS
  -- returns the value of radius
```


©2016 Course Technology, a part of Cengage Learning.

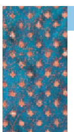


FIGURE 30

An Object-Oriented SportsWorld Program (continued)

```
  BEGIN
    return obj.radius;
  END getRadius;

 FUNCTION doCircumference(obj: in CIRCLE) RETURN FLOAT IS
  -- computes the circumference of a circle with given radius
  BEGIN
    return 2.0 * pi * obj.radius;
  END doCircumference;

 FUNCTION doArea(obj: in CIRCLE) RETURN FLOAT IS
  -- computes the area of a circle with given radius
  BEGIN
    return pi * obj.radius ** 2;
  END doArea;

 PROCEDURE getInput(radius : out FLOAT; taskToDo : out
                   CHARACTER) IS
  -- gets radius and choice of task from the user
  BEGIN
    TEXT_IO.NEW_LINE;
    TEXT_IO.PUT("Enter the value of the radius of a " &
                "pool: ");
    FLO_IO.GET(radius);
                       -- See what user wants to compute
    TEXT_IO.PUT("Enter your choice of task.");
    TEXT_IO.NEW_LINE;
    TEXT_IO.PUT("C to compute circumference, " &
                "A to compute area: ");
    TEXT_IO.GET(taskToDo);
  END getInput;

      newRadius : FLOAT;         -- radius of a pool - given
      taskToDo : CHARACTER;      -- holds user choice to
                                 -- compute circumference
                                 -- or area
      more : CHARACTER;          -- controls loop for
                                 -- processing more pools
      swimmingPool : CIRCLE;     -- instance of type circle
BEGIN
  TEXT_IO.PUT("Do you want to process a pool? (Y or N): ");
  TEXT_IO.GET(more);

  while more = 'Y'      -- more circles to process
   loop
    getInput(newRadius, taskToDo);
    setRadius(swimmingPool, newRadius);
    if taskToDo = 'C'   -- compute circumference
     then
      TEXT_IO.PUT("The circumference for a pool " &
                  "of radius ");
      FLO_IO.PUT(getRadius(swimmingPool), 3, 2, 0);
      TEXT_IO.PUT(" is ");
      FLO_IO.PUT(doCircumference(swimmingPool), 3, 2, 0);
      TEXT_IO.NEW_LINE;
```

©2016 Course Technology, a part of Cengage Learning.

FIGURE 30

An Object-Oriented SportsWorld Program (continued)

```
      else                -- compute area
       TEXT_IO.PUT("The area for a pool " &
                   "of radius ");
       FLO_IO.PUT(getRadius(swimmingPool), 3, 2, 0);
       TEXT_IO.PUT(" is ");
       FLO_IO.PUT(doArea(swimmingPool), 3, 2, 0);
       TEXT_IO.NEW_LINE;
     end if;
     TEXT_IO.NEW_LINE;
     TEXT_IO.PUT("Do you want to process more pools" &
                 " (Y or N): ");
     TEXT_IO.GET(more);
    end loop;             -- end of while loop
                          -- finish up
    TEXT_IO.NEW_LINE;
    TEXT_IO.PUT("Program will now terminate.");

 END SportsWorld;
```

primitive operations, the procedure *setRadius* and the functions *getRadius*, *doCircumference*, and *doArea*. Each of these primitive operations can be recognized as being related to the *CIRCLE* type by the fact that each has a parameter of type *CIRCLE*.

The question is, how does the main program code use the services of the *CIRCLE* type? An object (an instance of the type) has to be declared as being of type *CIRCLE*. The statement

```
swimmingPool : CIRCLE;     -- instance of type
                           -- circle
```

instantiates an object from the *CIRCLE* type and calls it *swimmingPool*. Now, the main program code can ask the *swimmingPool* object to invoke methods from its class. Consider the statements

```
TEXT_IO.PUT("The circumference for a pool " &
            "of radius ");
FLO_IO.PUT(getRadius(swimmingPool), 3, 2, 0);
TEXT_IO.PUT(" is ");
FLO_IO.PUT(doCircumference(swimmingPool), 3, 2, 0);
TEXT_IO.NEW_LINE;
```

These statements ask the *swimmingPool* object first to invoke the *getRadius* function. This function returns the value of the *swimmingPool* radius, which is then written out. Later the *swimmingPool* object invokes the *doCircumference* function. This function returns the value of the circumference of the *swimmingPool* object, which also is then written out. Note that, unlike the functions of Figure 25, the *doCircumference* and *doArea* functions in Figure 30 have no parameter for the value of the radius; as primitive operations of this class, they know at all times the current value of *radius* for the object that invoked them, and it does not have to be passed to them as an argument.

©2016 Course Technology, a part of Cengage Learning.

However, the invoking object (*swimmingPool*) is passed to each of the primitive operations.

These primitive operations are all "public" by default. Public operations can be used anywhere, including within the main program code, and indeed in any Ada program that wants to make use of this type. Think of the *CIRCLE* type as handing out a business card that advertises these services: Hey, you want a CIRCLE object that can find its own area? Find its own circumference? Set the value of its own radius? I'm your type!

The main program code, as before, handles all of the user interaction and now makes use of the *CIRCLE* type. In addition, it uses the *getInput* procedure, which is not a primitive operation of the *CIRCLE* type, it's the same "regular" procedure we've seen before.

Looking at the code for the primitive operations in Figure 30, we see that the *setRadius* procedure uses an assignment statement to change the value of *radius* to whatever quantity is passed to the parameter *value*. The *getRadius* function body is a single return statement. The *doCircumference* and *doArea* functions again consist of single statements that compute and return the proper value.

Among the variable declarations in procedure *SportsWorld*, there is no declaration for a variable called *radius*. There is a declaration for *newRadius*, and *newRadius* receives the value entered by the user in *getInput* for the radius of the circle. Therefore, isn't *newRadius* serving the same purpose as *radius* did in the old program? No—this is rather subtle, so pay attention: While *newRadius* holds the number the user wants for the circle radius, it is not itself the radius of *swimmingPool*. The radius of *swimmingPool* is the single property *radius* conferred on the *swimmingPool* object because it is an instance of the *CIRCLE* type. Only primitive operations of the class can change the properties of an object of that class. The *CIRCLE* type provides the *setRadius* procedure for this purpose. The main program code must ask the *swimmingPool* object to invoke *setRadius* to set the value of its radius equal to the value contained in *newRadius*. The *newRadius* argument corresponds to the *value* parameter in the *setRadius* procedure, which then gets assigned to the *radius* property.

```
setRadius(swimmingPool, newRadius);


PROCEDURE setRadius(obj: in out CIRCLE; value : in
  FLOAT) IS
 -- sets radius equal to value
 BEGIN
   obj.radius := value;
 END setRadius;
```

The *setRadius* primitive operation is a procedure because it contains no return statement. The invocation of this procedure is a complete Ada statement. Notice, however, that the *obj* parameter is an *in out* parameter because a property (the radius) of the corresponding argument (*swimmingPool*) is changed within this procedure.

Finally, the output statements in the main program code that print the values of the circumference and area also have *swimmingPool* invoke the *getRadius* function to return its current *radius* value, so it can be printed as part of the output. We could have used the variable *newRadius* here instead. However, *newRadius* is what we THINK has been used in the computation, whereas *radius* is what has REALLY been used.

©2016 Course Technology, a part of Cengage Learning.

6.3 *One More Example*

The object-oriented version of our SportsWorld program illustrates encapsulation. All data and calculations concerning circles are encapsulated in the *Circle* class. Let's look at one final example that illustrates the other two watchwords of OOP—polymorphism and inheritance.

In Figure 31 the domain of interest is that of geometric shapes. Four different types (classes) are given: *CIRCLE*, *RECTANGLE*, *SQUARE*, and *SQUARE2*. Each type declares its own properties and provides, in the form of procedures or functions, the services that an object of that type can perform. A CIRCLE object has a radius property, whereas a RECTANGLE object has a width property and a height property. Any CIRCLE object can set the value of its radius and can compute its area. A RECTANGLE object can set the value of its width and height and can compute its area. Both SQUARE and SQUARE2 objects have a side property that they can set, but they compute their areas in very different ways, as we will explain shortly.

The main program code creates objects from the various types. Then, for each object, the main program code requests the object to set its dimensions, using the values given, and to compute its area as part of a series of output statements giving information about the object. For example, the statement

```
setRadius(joe, 23.5);
```

instructs the circle named *joe* to invoke the *setRadius* procedure of *joe*'s type, thereby setting *joe*'s radius to 23.5. Figure 32 shows the output (wrapped to fit on the page) after the program in Figure 31 is run.

Here we see polymorphism at work, because there are lots of *doArea* functions; when the program executes, the correct function is used, on the basis of the type to which the object invoking the function belongs. After all, computing the area of a circle is quite different from computing the area of a rectangle.

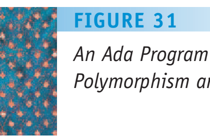


FIGURE 31
An Ada Program with Polymorphism and Inheritance

```
-- This program demonstrates the object-oriented
-- concepts of polymorphism and inheritance
-- Written by M. Phelps, 10/23/16

WITH TEXT_IO;

PROCEDURE Shapes IS
 PACKAGE FLO_IO IS NEW TEXT_IO.FLOAT_IO(FLOAT);

  pi : constant := 3.1416;   -- value of pi
---------------- type for circle -------
  TYPE CIRCLE IS
    tagged record
       radius:  FLOAT;
    end record;

 PROCEDURE setRadius(obj: in out CIRCLE; value : in FLOAT) IS
  -- sets radius equal to value
```


©2016 Course Technology, a part of Cengage Learning.

FIGURE 31

An Ada Program with Polymorphism and Inheritance (continued)

```
    BEGIN
      obj.radius := value;
    END setRadius;

   FUNCTION getRadius(obj: in CIRCLE) RETURN FLOAT IS
    -- returns the value of radius
    BEGIN
      return obj.radius;
    END getRadius;

   FUNCTION doArea(obj: in CIRCLE) RETURN FLOAT IS
    -- computes and returns the area of a circle
    BEGIN
       return pi * obj.radius ** 2;
    END doArea;
---------------- type for rectangle ------
    TYPE RECTANGLE IS
      tagged record
         width :  FLOAT;
         height : FLOAT;
      end record;

   PROCEDURE setWidth(obj : in out RECTANGLE; value : in FLOAT) IS
    -- sets width of the rectangle equal to value
    BEGIN
      obj.width := value;
    END setWidth;

   PROCEDURE setHeight(obj : in out RECTANGLE; value : in FLOAT) IS
    -- sets height of the rectangle equal to value
    BEGIN
      obj.height := value;
    END setHeight;

   FUNCTION getWidth(obj: in RECTANGLE) RETURN FLOAT IS
    -- returns the value of width
    BEGIN
      return obj.width;
    END getWidth;

   FUNCTION getHeight(obj: in RECTANGLE) RETURN FLOAT IS
    -- returns the value of height
    BEGIN
      return obj.height;
    END getHeight;

   FUNCTION doArea(obj : in RECTANGLE) RETURN FLOAT IS
    -- computes and returns the area of the rectangle
    BEGIN
      return obj.width * obj.height;
    END doArea;
```

©2016 Course Technology, a part of Cengage Learning.

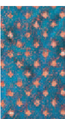


FIGURE 31

An Ada Program with Polymorphism and Inheritance (continued)

```
--------------- type for square -----
  TYPE SQUARE IS
    tagged record
       side :  FLOAT;
    end record;

 PROCEDURE setSide(obj : in out SQUARE; value : in FLOAT) IS
  -- sets side of the square equal to value
  BEGIN
    obj.side := value;
  END setSide;

 FUNCTION getSide(obj: in SQUARE) RETURN FLOAT IS
  -- returns the value of side
  BEGIN
     return obj.side;
  END getSide;

 FUNCTION doArea(obj : in SQUARE) RETURN FLOAT IS
  -- computes and returns the area of the square
  BEGIN
    return obj.side * obj.side;
  END doArea;
---------------- type for square2 -----
   TYPE SQUARE2 IS
    new RECTANGLE with record
       side : FLOAT;
    end record;

 PROCEDURE setSide(obj : in out SQUARE2; value : in FLOAT) IS
  -- sets side of the square equal to value
  -- also sets inherited width and height properties
  BEGIN
    obj.side := value;
    obj.width := value;
    obj.height := value;
  END setSide;

 FUNCTION getSide(obj: in SQUARE2) RETURN FLOAT IS
  -- returns the value of side
  BEGIN
    return obj.side;
  END getSide;

 FUNCTION doArea(obj : in SQUARE2) RETURN FLOAT IS
  -- computes and returns the area of the square
  BEGIN
     return doArea(RECTANGLE(obj));
  END doArea;
------------------- instances ----
  joe : CIRCLE;                -- instance of type circle
  luis : RECTANGLE;            -- instance of type rectangle
  anastasia : SQUARE;          -- instance of type square
  tyler : SQUARE2;             -- instance of type square2,
                                  special rectangle
```


©2016 Course Technology, a part of Cengage Learning.

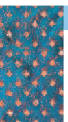


FIGURE 31

An Ada Program with Polymorphism and Inheritance (continued)

```
BEGIN
  setRadius(joe, 23.5);
  TEXT_IO.PUT("The area of a circle with radius ");
  FLO_IO.PUT(getRadius(joe), 3, 2, 0);
  TEXT_IO.PUT(" is ");
  FLO_IO.PUT(doArea(joe), 3, 2, 0);
  TEXT_IO.NEW_LINE;

  setWidth(luis, 12.4);
  setHeight(luis, 18.1);
  TEXT_IO.PUT("The area of a rectangle with " &
                "dimensions ");
  FLO_IO.PUT(getWidth(luis), 3, 2, 0);
  TEXT_IO.PUT(" and ");
  FLO_IO.PUT(getHeight(luis), 3, 2, 0);
  TEXT_IO.PUT(" is ");
  FLO_IO.PUT(doArea(luis), 3, 2, 0);
  TEXT_IO.NEW_LINE;

  setSide(anastasia, 3.0);
  TEXT_IO.PUT("The area of a square with side ");
  FLO_IO.PUT(getSide(anastasia), 3, 2, 0);
  TEXT_IO.PUT(" is ");
  FLO_IO.PUT(doArea(anastasia), 3, 2, 0);
  TEXT_IO.NEW_LINE;

  setSide(tyler, 4.2);
  TEXT_IO.PUT("The area of a square with side ");
  FLO_IO.PUT(getSide(tyler), 3, 2, 0);
  TEXT_IO.PUT(" is ");
  FLO_IO.PUT(doArea(tyler), 3, 2, 0);
  TEXT_IO.NEW_LINE;
                              -- finish up
  TEXT_IO.NEW_LINE;
  TEXT_IO.PUT("Program will now terminate.");

END Shapes;
```

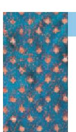


FIGURE 32

Output from the Program of Figure 31

```
The area of a circle with radius  23.50 is 1734.95
The area of a rectangle with dimensions  12.40 and  18.10
  is 224.44
The area of a square with side   3.00 is   9.00
The area of a square with side   4.20 is  17.64
```

The algorithms themselves are straightforward; they employ the usual formulas to compute the area of a circle, rectangle, and square. These functions can use the properties of the objects that invoke them without having the values of those properties passed as arguments.

SQUARE is a stand-alone type with a *side* property and a *doArea* function. The *SQUARE2* type, however, recognizes the fact that squares are special kinds of rectangles. The *SQUARE2* type is a subtype of the *RECTANGLE* class, as is

©2016 Course Technology, a part of Cengage Learning.

indicated by the reference to *RECTANGLE* in the type declaration for *SQUARE2*. It inherits the *width* and *height* properties from the “parent” *RECTANGLE* type. *SQUARE2* also has an additional *side* property of its own, which makes sense for a square but not for an arbitrary rectangle. The *setSide* procedure assigns a value to the *side* property, and also assigns the same value to the *width* and *height* properties that a SQUARE2 object inherits from the parent type. To compute the area, the *doArea* function simply turns the computation over to the *doArea* function inherited from the *RECTANGLE* type. This is invoked in the statement

```
return doArea(RECTANGLE(obj));
```

where the *obj* has been type cast from a SQUARE2 to a RECTANGLE. This is possible because a SQUARE2 object really is a form of rectangle, and this type cast tells the system that the invoking object is a RECTANGLE object, so the *doArea* function from the RECTANGLE type is to be used. Here we see inheritance at work.

Inheritance can be carried through multiple “generations.” We might redesign the program so that there is one “supertype” that is a general *SHAPE* type, of which *CIRCLE* and *RECTANGLE* are subtypes, *SQUARE2* being a subtype of *RECTANGLE* (see Figure 33 for a possible type hierarchy).

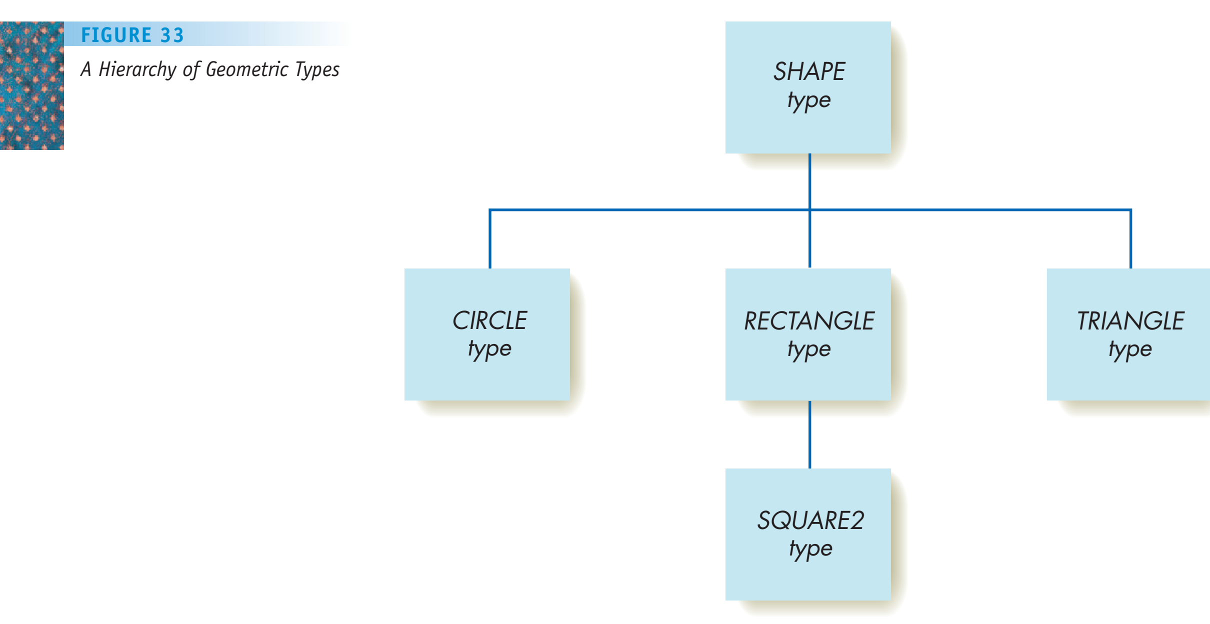
6.4 *What Have We Gained?*

Now that we have some idea of the flavor of object-oriented programming, we should ask what we gain by this approach. There are two major reasons why OOP is a popular way to program:

- Software reuse
- A more natural “worldview”

FIGURE 33

A Hierarchy of Geometric Types



©2016 Course Technology, a part of Cengage Learning.

SOFTWARE REUSE. Manufacturing productivity took a great leap forward when Henry Ford invented the assembly line. Automobiles could be assembled using identical parts so that each car did not have to be treated as a unique creation. Computer scientists are striving to make software development more of an assembly-line operation and less of a handcrafted, start-over-each-time process. Object-oriented programming is a step toward this goal: A useful type that has been implemented and tested becomes a component available for use in future software development. Anyone who wants to write an application program involving circles, for example, can use the already written, tried, and tested *CIRCLE* type. As the "parts list" (the type library) grows, it becomes easier and easier to find a "part" that fits, and less and less time has to be devoted to writing original code. If the type doesn't quite fit, perhaps it can be modified to fit by creating a subtype; this is still less work than starting from scratch. Software reuse implies more than just faster code generation. It also means improvements in *reliability*; these types have already been tested, and if properly used, they will work correctly. And it means improvements in *maintainability*. Thanks to the encapsulation property of object-oriented programming, changes can be made in type implementations without affecting other code, although such change requires retesting the types.

A MORE NATURAL "WORLDVIEW." The traditional view of programming is procedure-oriented, with a focus on tasks, subtasks, and algorithms. But wait—didn't we talk about subtasks in OOP? Haven't we said that computer science is all about algorithms? Does OOP abandon these ideas? Not at all. It is more a question of *when* these ideas come into play. Object-oriented programming recognizes that in the "real world," tasks are done by entities (objects). Object-oriented program design begins by identifying those objects that are important in the domain of the program because their actions contribute to the mix of activities present in the banking enterprise, the medical office, or wherever. Then it is determined what data should be associated with each object and what subtasks the object contributes to this mix. Finally, an algorithm to carry out each subtask must be designed. We saw in the modularized versions of the SportsWorld program in Figures 24 and 25 how the overall algorithm could be broken down into pieces that are isolated within functions and procedures. Object-oriented programming repackages those functions and procedures by encapsulating them within the appropriate type of objects.

Object-oriented programming is an approach that allows the programmer to come closer to modeling or simulating the world as we see it, rather than to mimic the sequential actions of the Von Neumann machine. It provides another buffer between the real world and the machine, another level of abstraction in which the programmer can create a virtual problem solution that is ultimately translated into electronic signals on hardware circuitry.

Finally, we should mention that a graphical user interface, with its windows, icons, buttons, and menus, is an example of object-oriented programming at work. A general button class, for example, can have properties of height, width, location on the screen, text that may appear on the button, and so forth. Each individual button object has specific values for those properties. The button class can perform certain services by responding to messages, which are generated by events (for example, the user clicking the mouse on a button triggers a "mousedown" event). Each particular button object individualizes the code to respond to these messages in unique ways. We will not go into details of how to develop graphical user interfaces in Ada,

©2016 Course Technology, a part of Cengage Learning.

but in the next section you will see a bit of the programming mechanics that can be used to draw the graphics items that make up a visual interface.

PRACTICE PROBLEMS

1. What is the output from the following section of code if it is added to the main program code of the Ada program in Figure 31?

```
one : SQUARE;
setSide(one,10.0);
TEXT_IO.PUT("The area of a square with side ");
FLO_IO.PUT(getSide(one), 3, 2, 0);
TEXT_IO.PUT(" is ");
FLO_IO.PUT(doArea(one), 3, 2, 0);
TEXT_IO.NEW_LINE;
```

2. In the hierarchy of Figure 33, suppose that the *Triangle* type is able to perform a *doArea* function. What two properties should any triangle object have?

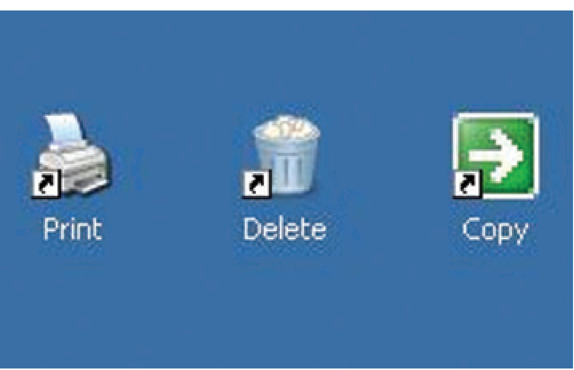
7 Graphical Programming

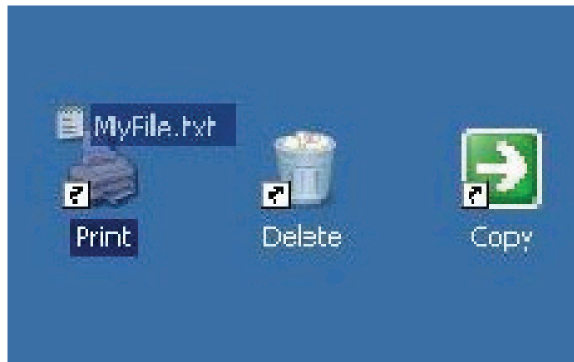
The programs that we have looked at so far all produce *text output*—output composed of the characters {A . . . Z, a . . . z, 0 . . . 9} along with a few punctuation marks. For the first 30 to 35 years of software development, text was virtually the only method of displaying results in human-readable form, and in those early days it was quite common for programs to produce huge stacks of alphanumeric output. These days an alternative form of output—*graphics*—has become much more widely used. With graphics, we are no longer limited to 100 or so printable characters; instead, programmers are free to construct whatever shapes and images they desire.

The intelligent and well-planned use of graphical output can produce some phenomenal improvements in software. We discussed this issue in Chapter 6, where we described the move away from the text-oriented operating systems of the 1970s and 1980s, such as MS-DOS and VMS, to operating systems with more powerful and user-friendly graphical user interfaces (GUIs), such as Windows 7, Windows 8, and Mac OS X. Instead of requiring users to learn dozens of complex text-oriented commands for such things as copying, editing, deleting, moving, and printing files, GUIs can present users with simple and easy-to-understand visual metaphors for these operations. In the first image on the next page, the operating system presents the user with icons for printing, deleting, or copying a file. In the second image on the next page, dragging a file to the printer icon prints the file.

Not only does graphics make it easier to manage the tasks of the operating system, it can help us visualize and make sense of massive amounts of output produced by programs that model complex physical, social, and mathematical systems. (We discuss modeling and visualization in Chapter 13 of the text.) Finally, there are many applications of computers that would simply be impossible

©2016 Course Technology, a part of Cengage Learning.





without the ability to display output visually. Applications such as virtual reality, computer-aided design/computer-aided manufacturing (CAD/CAM), games and entertainment, medical imaging, and computer mapping would not be anywhere near as important as they are without the enormous improvements that have occurred in the areas of graphics and visualization.

So, we know that graphical programming is important. The question is: What features must be added to a programming language like Ada to produce graphical output?

7.1 *Graphics Hardware*

Modern computer terminals use what is called a **bitmapped display**, in which the screen is made up of thousands of individual picture elements, or **pixels**, laid out in a two-dimensional grid. These are the same pixels used in visual images, as discussed in Chapter 4. In fact, the display is simply one large visual image. The number of pixels on the screen varies from system to system; typical values range from 800 × 600 up to 1560 × 1280. Terminals with a high density of pixels are called **high-resolution** terminals. The higher the resolution—that is, the more pixels available in a given amount of space—the sharper the visual image because each individual pixel is smaller. However, if the screen size itself is small, then a high-resolution image can be too tiny to read. A 30″ wide-screen monitor might support a resolution of 2560 × 1600, but that would not be suitable for a laptop screen. In Chapter 4 you learned that a color display requires

©2016 Course Technology, a part of Cengage Learning.

C8814_chapter_ada.qxd 1/17/15 5:27 PM Page 69

24 bits per pixel, with 8 bits used to represent the value of each of the three colors red, green, and blue. The memory that stores the actual screen image is called a **frame buffer**. A high-resolution color display might need a frame buffer with (1560 × 1280) pixels × 24 bits/pixel = 47,923,000 bits, or about 6 MB, of memory for a single image. (One of the problems with graphics is that it requires many times the amount of memory needed for storing text.)

The individual pixels in the display are addressed using a two-dimensional coordinate grid system, the pixel in the upper-left corner being (0, 0). The overall pixel-numbering system is summarized in Figure 34. The specific values for *maxX* and *maxY* in Figure 34 are, as mentioned earlier, system-dependent. (Note that this coordinate system is not the usual mathematical one. Here, the origin is in the upper-left corner, and *y* values are measured downward.)

The terminal hardware displays on the screen the frame buffer value of every individual pixel. For example, if the frame buffer value on a color monitor for position (24, 47) is RGB (0, 0, 0), the hardware sets the color of the pixel located at column 24, row 47 to black, as shown in Figure 35. The operation diagrammed in Figure 35 must be repeated for all of the 500,000 to 2 million pixels on the screen. However, the setting of a pixel is not permanent; on the contrary, its color and intensity fade quickly. Therefore, each pixel must be "repainted" often enough so that our eyes do not detect any "flicker," or change in intensity. This requires the screen to be completely updated, or

FIGURE 34

Pixel-Numbering System in a Bitmapped Display

(0, 0)	(1, 0)	(2, 0)	. . .	(maxX, 0)
(0, 1)	(1, 1)	(2, 1)	. . .	(maxX, 1)
(0, 2)	(1, 2)	(2, 2)	. . .	(maxX, 2)
		⋮		
(0, maxY)	(1, maxY)	(2, maxY)		(maxX, maxY)

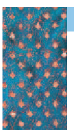
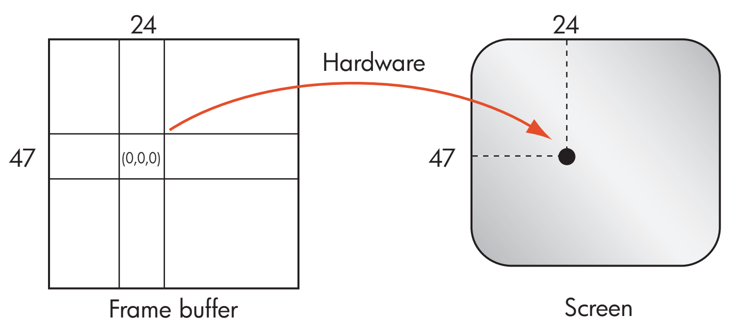


FIGURE 35

Display of Information on the Terminal



©2016 Course Technology, a part of Cengage Learning.

refreshed, 30–50 times per second. By setting various sequences of pixels to different colors, the user can have the screen display any desired shape or image. This is the fundamental way in which graphical output is achieved.

7.2 *Graphics Software*

To control the setting and clearing of pixels, programmers use a collection of procedures that are part of a special software package called a **graphics library**. Virtually all modern programming languages, including Ada, have a graphics library that is available for creating different shapes and images. Typically, an "industrial strength" graphics library includes hundreds of procedures for everything from drawing simple geometric shapes like lines and circles, to creating and selecting colors, to more complex operations such as displaying scrolling windows, pull-down menus, and buttons. We restrict our discussion to a modest set of procedures. Although the set is unrealistically small, it will still give you a good idea of what visual programming is like, and enable you to produce some interesting, nontrivial images on the screen. Again, the graphics library used in the examples that follow is AdaGraph, available for free download from *http://www.filewatcher.com/m/adagraph.zip.103876-0.html*.

To employ the AdaGraph library, all that is needed is a *with* (and optionally a *use*) clause to make the main program code aware that graphics commands are to be used, for example:

```
WITH AdaGraph;
USE AdaGraph;
```

Recall, including the *use* clause just eliminates the need for the prefix qualification *AdaGraph* for references to items in the AdaGraph library. As in the previous examples, the *use* clause will not be included, so that the exact details of what comes from which package will be obvious.

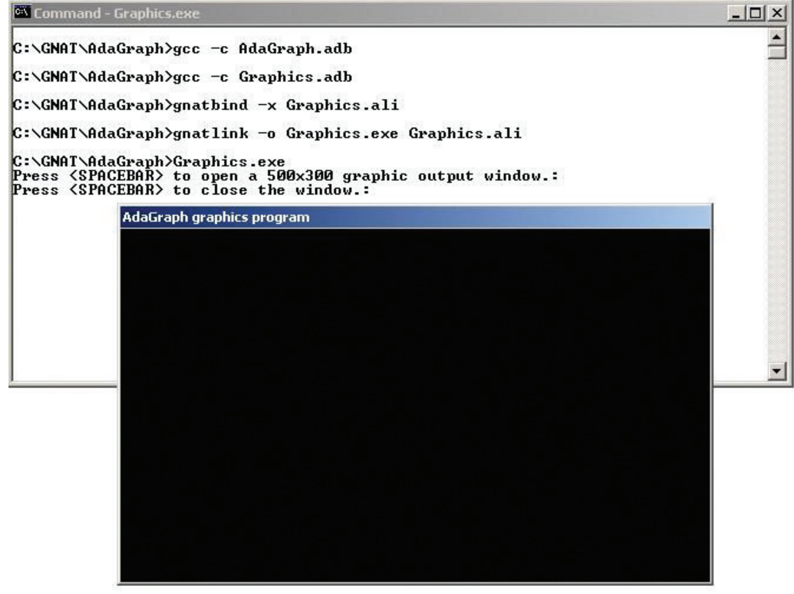
Here is the program code to open and close a window.

```
-- This program demonstrates Ada
-- graphics using the AdaGraph library

WITH TEXT_IO;      -- Ada I/O package

WITH AdaGraph;     -- Ada graphics package

PROCEDURE Graphics IS

   -- Wait for spacebar to be pressed in the
   -- command window
  PROCEDURE WaitForKeypress (message : in STRING) IS
     C : CHARACTER := 'X';

   BEGIN
     TEXT_IO.PUT("Press <SPACEBAR> to " & message
                 & ": ");
     while C /= ' '
      loop
```


©2016 Course Technology, a part of Cengage Learning.

```
          TEXT_IO.Get_Immediate (C);
        end loop;
        TEXT_IO.New_Line;
     END WaitForKeypress;

-- horizontal size
displayWidth  : CONSTANT INTEGER := 500;
-- vertical size
displayHeight : CONSTANT INTEGER := 300;

     maxX      : INTEGER; -- maximum display coordinate
     maxY      : INTEGER;
     maxCharX  : INTEGER; -- maximum character size
     maxCharY  : INTEGER;

   BEGIN

     WaitForKeypress("open a 500x300 graphic output " &
                     "before window.");
     AdaGraph.Create_Sized_Graph_Window
       (displayWidth, displayHeight, maxX, maxY,
        maxCharX, maxCharY);
     AdaGraph.Set_Window_Title("AdaGraph " &
                               "graphics program");

     WaitForKeypress("close the window.");
     AdaGraph.Destroy_Graph_Window;

   END Graphics;
```

The screen display that follows shows the details of compilation, binding, and execution of the Graphics package body.



©2016 Course Technology, a part of Cengage Learning.

Notice that the Command window, where the commands to the Ada compiler are entered, and the graphics display window, where the graphics will be drawn, are separate.

In the Command window, the first command compiles the AdaGraph package. The second command compiles the Graphics package that we just wrote. The third command carries out the binding between the packages. The fourth command links the results of the compilation together to produce an executable (exe) file. Then the program begins to execute the main program code.

```
BEGIN

  WaitForKeypress("open a 500x300 graphic output " &
                  "before window.");
  AdaGraph.Create_Sized_Graph_Window
    (displayWidth, displayHeight, maxX, maxY,
     maxCharX, maxCharY);
  AdaGraph.Set_Window_Title("AdaGraph " &
                            "graphics program");

  WaitForKeypress("close the window.");
  AdaGraph.Destroy_Graph_Window;

END Graphics;
```

The first line of output from the executing program is a prompt to press the Spacebar (*WaitForKeypress*). When the Spacebar is pressed, the graphics window is created and displayed by a call to an AdaGraph library procedure (*Create_Sized_Graph_Window*). The window is then given a title (*Set_Window_Title*). Then the Command window waits for another press of the Spacebar (*WaitForKeypress*) to close the window (*Destroy_Graph_Window*).

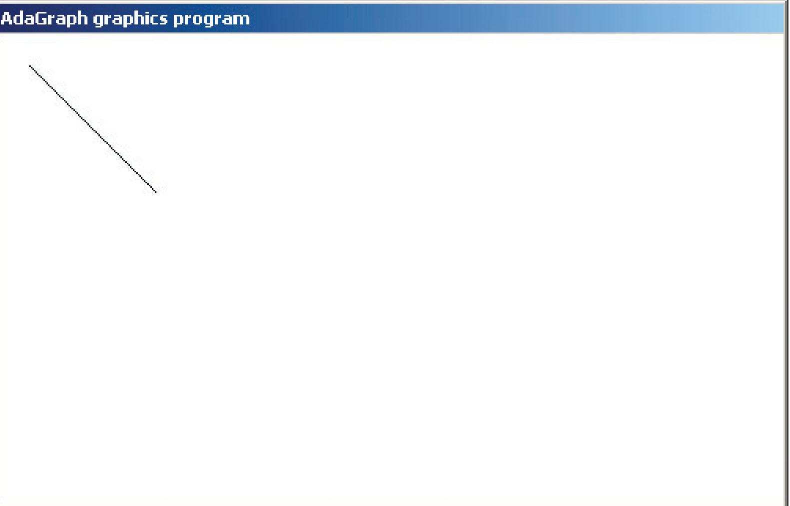
The next task is to see how AdaGraph procedures can be used to draw on the display window.

The code to draw a line from point (20,20) to point (100,100) is shown next. The added code is in bold.

```
BEGIN

  WaitForKeypress("open a 500x300 graphic output " &
                  "before window.");
  AdaGraph.Create_Sized_Graph_Window
    (displayWidth, displayHeight, maxX, maxY,
     maxCharX, maxCharY);
  AdaGraph.Set_Window_Title("AdaGraph " &
                            "graphics program");

  AdaGraph.Clear_Window(AdaGraph.White);
  AdaGraph.Draw_Line(20, 20, 100, 100,
                     AdaGraph.Black);

  WaitForKeypress("close the window.");
  AdaGraph.Destroy_Graph_Window;

END Graphics
```

©2016 Course Technology, a part of Cengage Learning.

The first added line changes the background color of the display to white. The second draws the line. The result of executing the program containing this code is:



The parameter definitions for *Draw_Line* are

Draw_Line (x_1, y_1, x_2, y_2, Color)

where (x_1, y_1) are the coordinates (in pixels) of the start of the line, and (x_2, y_2) are the coordinates of the end of the line. Color is something new to our discussion of Ada. Color is an enumeration type; that is, the colors supported by AdaGraph form a list (an enumeration) of the items (colors) that can be used. Here is the code from the AdaGraph specification that sets up the enumeration:

```
type Color_Type is  (Black, Blue, Green, Cyan,
                     Red, Magenta, Brown,
                     Light_Gray, Dark_Gray,
                     Light_Blue, Light_Green,
                     Light_Cyan, Light_Red,
                     Light_Magenta, Yellow, White);
```

The code to draw a rectangle touching the four points (25, 60), (75, 60), (25, 100), and (75, 100) is:

```
AdaGraph.Draw_Box (25, 60, 75, 100,
                   AdaGraph.Black,
                   AdaGraph.No_Fill);
```

Note that the parameters for *Draw_Box* are

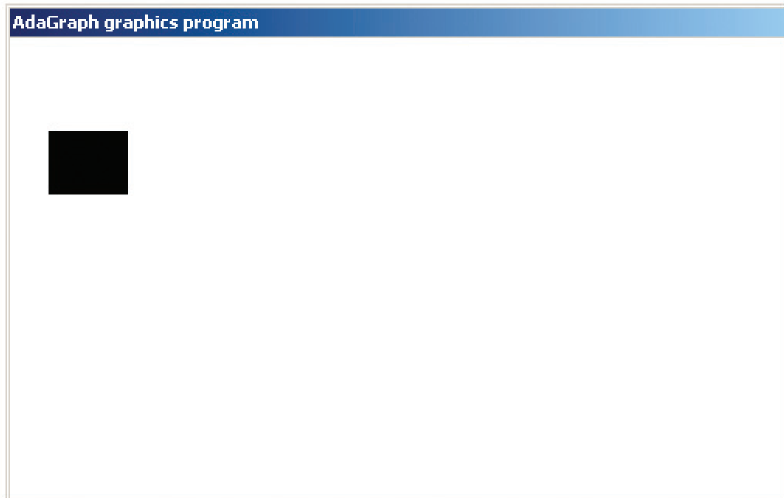
Draw_Box (x_1, y_1, x_2, y_2, Color, Filled)

©2016 Course Technology, a part of Cengage Learning.

C8814_chapter_ada.qxd 1/17/15 5:27 PM Page 74

where the (x, y) pairs are the opposite corners of the rectangle. *Color* is the color of the line and the fill color. *Filled* is another enumeration type with the values *No_Fill* (display on the top) and *Fill* (display on the bottom).

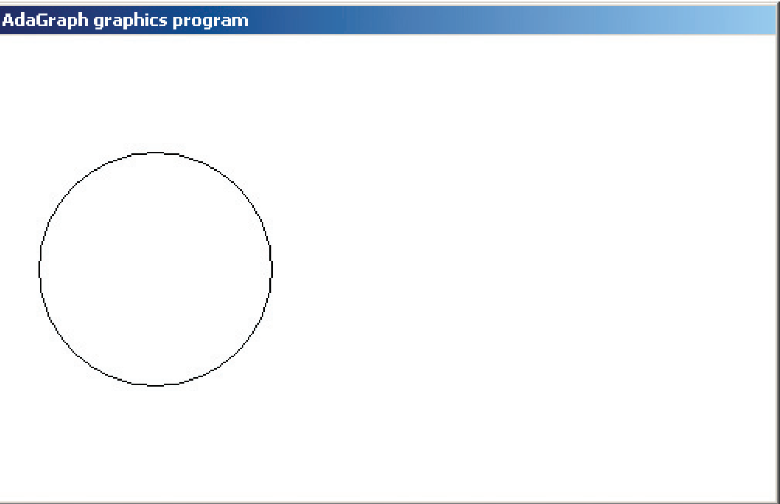
AdaGraph graphics program



In AdaGraph, the *Draw_Circle* procedure is used to draw a circle. Here is the code and, on the next page, the result of drawing a circle with radius 75 pixels centered at the point (100, 150):

```
AdaGraph.Draw_Circle (100, 150, 75,
                      AdaGraph.Black,
                      AdaGraph.No_Fill);
```

©2016 Course Technology, a part of Cengage Learning.



The parameters are:

```
Draw_Circle (x, y, Radius, Color, Filled);
```

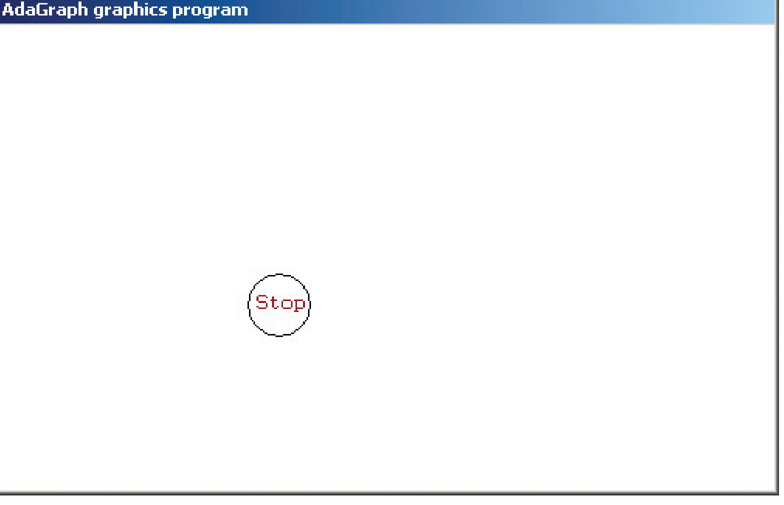
where (x, y) are the coordinates of the center of the circle, and Radius is the radius of the circle. Color and Filled are the same as for the rectangle. An error will occur if the figure being drawn does not fit completely inside the graphics window.

How does AdaGraph provide for text annotations on the screen? There is a procedure named *Display_Text* that will do the job. The parameter list is

```
Display_Text (x, y, string, Color)
```

The *string* is the string to be output beginning at (x, y). Color determines the color of the letters.

Here is an example with the text "Stop" drawn inside of a circle.



©2016 Course Technology, a part of Cengage Learning.

The code to produce this image is

```
AdaGraph.Draw_Circle (180, 180, 20,
                      AdaGraph.Black,
                      AdaGraph.No_Fill);
AdaGraph.Display_Text (165, 172, "Stop",
                       AdaGraph.Red);
```

In summary, we have the following graphics procedures at our disposal.

Draw_Line (x_1, y_1, x_2, y_2, Color)
Draw_Box (x_1, y_1, x_2, y_2, Color, Filled)
Draw_Circle (x, y, Radius, Color, Filled)
Display_Text (x, y, string, Color)

Now that a display window and graphics are available, we seem close to producing the elements of a typical GUI. Can we draw a button that acts like a button on a real GUI form—that is, can we write code to sense a mouse click on that button and respond with some action? In particular, we will sense the left mouse button up event. (Looking for the mouse up event gives users a way out if they press the mouse button and decide they don't want to click: namely, they can move the mouse off the button and then release the mouse button.) Here is the full code for the Ada program.

```
-- This program demonstrates capture of a mouse
-- click using the AdaGraph library

WITH TEXT_IO;                 -- Ada I/O package

WITH AdaGraph;               -- Ada graphics package

PROCEDURE Graphics IS
  PACKAGE INT_IO IS NEW TEXT_IO.INTEGER_IO(INTEGER);

  PROCEDURE WaitForKeypress (message : in STRING) IS
   -- Wait for spacebar to be pressed in the
   -- command window
     C : CHARACTER := 'X';
   BEGIN
     TEXT_IO.PUT("Press <SPACEBAR> to " & message
                 & ": ");
     while C /= ' '
      loop
       TEXT_IO.Get_Immediate(C);
     end loop;
     TEXT_IO.New_Line;
  END WaitForKeypress;

  -- horizontal size
  displayWidth  : CONSTANT INTEGER := 500;
  -- vertical size
  displayHeight : CONSTANT INTEGER := 300;
```

©2016 Course Technology, a part of Cengage Learning.

```
  maxX      : INTEGER;   -- maximum display coordinate
  maxY      : INTEGER;
  maxCharX : INTEGER;   -- maximum character size
  maxCharY : INTEGER;

  Mouse    : AdaGraph.Mouse_Type;  -- mouse information
  x        : INTEGER := 0;
  y        : INTEGER := 0;

BEGIN
  WaitForKeypress("open a 500x300 graphic output " &
                "before window.");
  AdaGraph.Create_Sized_Graph_Window
     (displayWidth, displayHeight, maxX, maxY,
      maxCharX, maxCharY);
  AdaGraph.Set_Window_Title("AdaGraph " &
                          "graphics program");

  AdaGraph.Clear_Window(AdaGraph.White);
  AdaGraph.Draw_Box (25, 60, 65, 110,
                   AdaGraph.Black,
                   AdaGraph.No_Fill);
  AdaGraph.Display_Text (30, 75, "Stop",
                       AdaGraph.Red);

  while not AdaGraph.Key_Hit loop
   if AdaGraph.Mouse_Event
    then
     Mouse := AdaGraph.Get_Mouse;
      case Mouse.Event is
        when AdaGraph.None       => null;
        when AdaGraph.Moved      => X :=
          Mouse.X_Pos; Y := Mouse.Y_Pos;
        when AdaGraph.Right_Up   => null;
        when AdaGraph.Left_Down  => null;
        when AdaGraph.Right_Down => null;
        when AdaGraph.Left_Up    =>
          TEXT_IO.NEW_LINE;
          TEXT_IO.PUT(" x="); INT_IO.PUT(x);
          TEXT_IO.PUT(" y="); INT_IO.PUT(y);
          AdaGraph.Display_Text (10, 10, " Inside ",
             AdaGraph.White);
          AdaGraph.Display_Text (10, 10,
             " Outside ", AdaGraph.White);
          if (x >= 25) and (x <=  65) and
             (y >= 60) and (y <= 110)
           then
            TEXT_IO.PUT("  Inside");
            AdaGraph.Display_Text (10, 10,
            " Inside ", AdaGraph.Green);
           else
            TEXT_IO.PUT("  Outside");
```


©2016 Course Technology, a part of Cengage Learning.

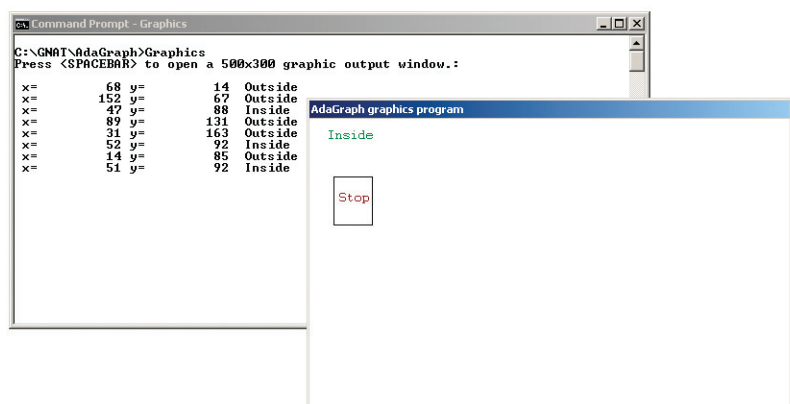
```
              AdaGraph.Display_Text (10, 10,
              " Outside ", AdaGraph.Red);
            end if;
         end case;
        end if;
       end loop;
       AdaGraph.Destroy_Graph_Window;

     END Graphics;
```

Wow! The lines in bold are the keys to the example; the other lines are needed to make the example work.

First there are some declarations to gain access to the mouse properties and store them for use in the decision process. The next set of bold lines should be familiar; they draw the button and caption (rectangle and text).

The *while* statement forms a continuous looping process looking for one of two things: a keystroke (to stop the program) or a mouse event (left button up). The *while* condition looks for the keystroke (*AdaGraph.Key_Hit*). If one occurs, the loop ends, and the window is closed. The *if* condition looks for a mouse event (*AdaGraph.Mouse_Event*). If one occurs, the various options for a mouse event are examined using a *case* statement. The way the *case* statement works is that it looks at the event options by checking a set of *when* clauses. There are only two of the event choices that are of interest here: the event that the mouse moves (then save the x, y position of the mouse pointer), and the event that the left mouse button has come up (then do the testing to see if the mouse pointer is in the rectangle). The bold *if* statement does the checking to see if the (x, y) position of the mouse pointer (at the time of the click) is within the rectangle. Depending on the results of this test, "Inside" or "Outside" is displayed both on the console and on the window. The two *Display_Text* statements with Color set to white are needed to "erase" the previous word from the window before drawing the new word.

The following display in the Command window shows the results of a series of mouse clicks; the final configuration is shown in the graphics window.



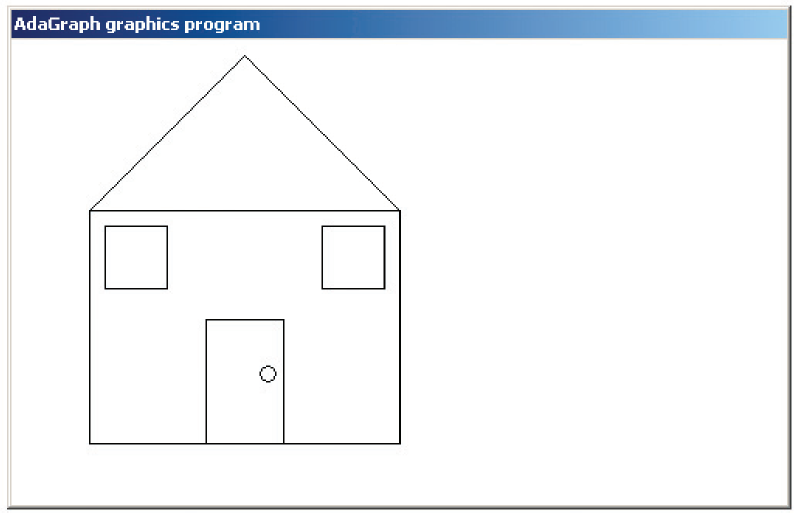
Of course, a real GUI interface would produce much more sophisticated responses to user mouse clicks, but this is the general idea of how **event-driven programming** works.

This brief introduction to graphical programming allows you to produce some interesting images and, even more important, gives you an appreciation for how visually oriented software is developed.

©2016 Course Technology, a part of Cengage Learning.

PRACTICE PROBLEMS

Write the sequence of commands to draw the following "house" in the graphics window.



Create the house using four rectangles (for the base of the house, the door, and the two windows), two line segments (for the roof), and one circle (for the doorknob). Locate the house anywhere you want in the graphics window.

8 Conclusion

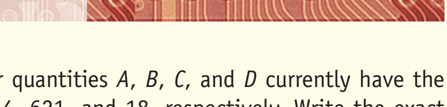
In this module we looked at one representative high-level programming language, Ada. Of course, there is much about this language that has been left unsaid, but we have seen how the use of a high-level language overcomes many of the disadvantages of assembly language programming, creating a more comfortable and useful environment for the programmer. In a high-level language, the programmer need not manage the storage or movement of data values in memory. The programmer can think about the problem at a higher level, can use program instructions that are both more powerful and more natural language–like, and can write a program that is much more portable among various hardware platforms. We also saw how modularization, through the use of functions, procedures, and parameters, allows the program to be more cleanly structured, and how object-oriented programming allows a more intuitive view of the problem solution and provides the possibility for reuse of helpful types. We even had a glimpse of graphical programming.

Ada is not the only high-level language. You might be interested in looking at the other online language modules (Java, Python, C++, and C#). You will find that the Ada syntax (form of the statements) is quite different from

©2016 Course Technology, a part of Cengage Learning.

that of these other languages. Ada derives its syntax from Pascal-like languages where statement blocks are delimited by keywords such as BEGIN .. END and THEN .. ELSE .. END IF, rather than by "curly brackets" { . . . }, and functions/procedures can be nested within other functions/procedures. Still other languages take quite a different approach to problem solving. In Chapter 10 of *Invitation to Computer Science*, we look at some other languages and language approaches and also address the question of why there are so many different programming languages.

©2016 Course Technology, a part of Cengage Learning.



EXERCISES

1. Write an Ada declaration for one real number quantity to be called *rate*.

2. Write an Ada declaration for two integer quantities called *orderOne* and *orderTwo*.

3. Write an Ada declaration for a constant quantity called *evaporationRate*, which is to have the value 6.15.

4. An Ada program needs one constant *stockTime* with a value of 4, one integer variable *inventory*, and one real number variable *sales*. Write the necessary declarations.

5. You want to write an Ada program to compute the average of three quiz grades for a single student. Decide what variables your program needs, and write the necessary declarations.

6. Given the declaration

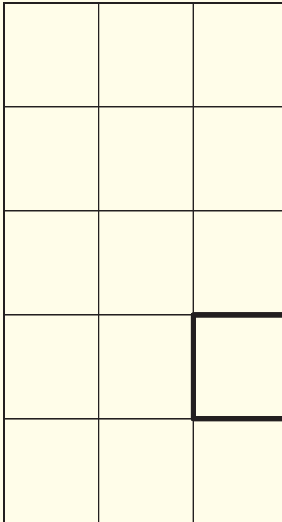
```
list : array(1..10) of INTEGER;
```

   how do you refer to the eighth number in the array?

7. An array declaration such as

```
table : array(0..4, 0..2) of
  INTEGER;
```

   represents a two-dimensional table of values with 5 rows and 3 columns. Rows and columns are numbered in this Ada array starting at 0, not at 1. Given this declaration, how do you refer to the marked cell below?



8. Write Ada statements to prompt for and collect values for the time in hours and minutes (two integer quantities).

9. Say an Ada program computes two integer quantities *inventoryNumber* and *numberOrdered*. Write the output statements that print these two quantities along with appropriate text information.

10. The integer quantities *A*, *B*, *C*, and *D* currently have the values 13, 4, 621, and 18, respectively. Write the exact output generated by the following statements, using b to denote a blank space.

```
INT_IO.PUT(A, 5);
INT_IO.PUT(B, 3);
INT_IO.PUT(C, 4);
INT_IO.PUT(D, 2);
```

11. Write Ada formatting and output statements to generate the following output, assuming that *density* is a type FLOAT variable with the value 63.78.

```
The current density is 63.8,
to within one decimal place.
```

12. What is the output after the following sequence of statements is executed? (Assume that the integer variables *A* and *B* have been declared.)

```
A := 12;
B := 20;
B := B + 1;
A := A + B;
INT_IO.PUT(2 * A);
```

13. Write Ada code that gets the length and width of a rectangle from the user and computes and writes out the area. Assume that the variables have all been declared as type INTEGER.

14. a. In the *SportsWorld* program of Figure 15, the user must respond with "C" to choose the circumference task. In such a situation, it is preferable to accept either uppercase or lowercase letters. Rewrite the condition in the program to allow this.

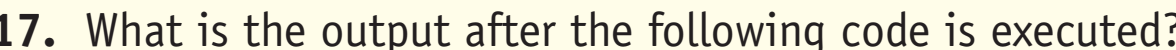
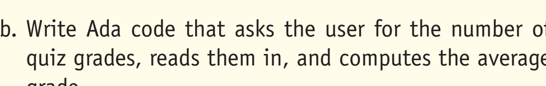
    b. In the *SportsWorld* program, rewrite the condition for continuation of the program to allow either an uppercase or a lowercase response.

15. Write Ada code that gets a single character from the user and writes out a congratulatory message if the character is a vowel (a, e, i, o, or u), but otherwise writes out a "You lose, better luck next time" message.

16. Insert the missing line of code so that the following adds the integers from 1 to 10, inclusive.

```
value := 0;
top := 10;
score := 1;
while score <= top
 loop
  value := value + score;
  -- missing line
 end loop;
TEXT_IO.PUT("value is: ");
INT_IO.PUT(value);
```


©2016 Course Technology, a part of Cengage Learning.

EXERCISES

17. What is the output after the following code is executed?

```
low := 1;
high := 20;
while low <= high
 loop
  INT_IO.PUT(low);
  TEXT_IO.PUT(" ");
  INT_IO.PUT(high);
  TEXT_IO.NEW_LINE;
  low := low + 1;
  high := high - 1;
 end loop;
```

18. Write Ada code that outputs the even integers from 2 through 30, one per line. Use a while loop.

19. In a while loop, the Boolean condition that tests for loop continuation is done at the top of the loop, before each iteration of the loop body. As a consequence, the loop body might not be executed at all. Our pseudocode language of Chapter 2 contains a do-while loop construction in which a test for loop termination occurs at the bottom of the loop rather than at the top, so that the loop body always executes at least once. Ada contains an *exit when* statement that tests for a condition and allows an exit from the loop. The form of the statement is:

```
loop
     S1;
   exit when (Boolean condition);
end loop;
```

where, as usual, S1 can be a compound statement. Write Ada code to add up a number of nonnegative integers that the user supplies and to write out the total. Use a negative value as a sentinel, and assume that the first value is nonnegative. Use an *exit-when* statement.

20. Write Ada code that asks for a duration of time in hours and minutes, and writes out the duration only in minutes.

21. Write Ada code that asks for the user's age in years. If the user is under 35, then quote an insurance rate of $2.23 per $100 for life insurance; otherwise, quote a rate of $4.32.

22. Write Ada code that reads integer values until a 0 value is encountered and then writes out the sum of the positive values read and the sum of the negative values read.

23. Write Ada code that reads in a series of positive integers and writes out the product of all the integers less than 25 and the sum of all the integers greater than or equal to 25. Use 0 as a sentinel value.

24. a. Write Ada code that reads in 10 integer quiz grades and computes the average grade. (*Hint*: Remember the peculiarity of integer division.)

    b. Write Ada code that asks the user for the number of quiz grades, reads them in, and computes the average grade.

25. Write an Ada procedure that receives two integer arguments and writes out their sum and their product.

26. Write an Ada procedure that receives an integer argument representing the number of DVDs rented so far this month and a real number argument representing the sales amount for DVDs sold so far this month. The procedure asks the user for the number of DVDs rented today and the sales amount for DVDs sold today, and then returns the updated figures to the main program code.

27. Write an Ada function that receives three integer arguments and returns the maximum of the three values.

28. Write an Ada function that receives miles driven as a type FLOAT argument and gallons of gas used as a type INTEGER argument, and returns miles per gallon.

29. Write an Ada program that uses an input procedure to get the miles driven (type FLOAT) and the gallons of gas used (type INTEGER), then writes out the miles per gallon, using the function from Exercise 28.

30. Write an Ada program to balance a checkbook. The program needs to get the initial balance, the amounts of deposits, and the amounts of checks. Allow the user to process as many transactions as desired; use separate procedures to handle deposits and checks.

31. Write an Ada program to compute the cost of carpeting three rooms. Make the carpet cost a constant of $8.95 per square yard. Use four separate functions/procedures to collect the dimensions of a room in feet, convert feet into yards, compute the area, and compute the cost per room. The main program code should use a loop to process each of the three rooms, then add the three costs, and write out the total cost. (*Hint*: The function to convert feet into yards must be used twice for each room, with two different arguments. Hence, it does not make sense to try to give the parameter the same name as the argument.)

32. a. Write an Ada *doPerimeter* function for the *Rectangle* class of Figure 31.

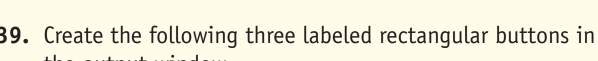
    b. Write Ada code that creates a new Rectangle object called *yuri*, then writes out information about this object and its perimeter using the *doPerimeter* function from part (a).

33. Draw a type hierarchy diagram similar to Figure 33 for the following types: *Student*, *UndergraduateStudent*, *GraduateStudent*, *Sophomore*, *Senior*, *PhDStudent*.

34. Imagine that you are writing a program using an object-oriented programming language. Your program will be used to maintain records for a real estate office. Decide

©2016 Course Technology, a part of Cengage Learning.

C8814_chapter_ada.qxd 1/17/15 5:28 PM Page 83



EXERCISES

on one class in your program and a service that objects of that class might provide.

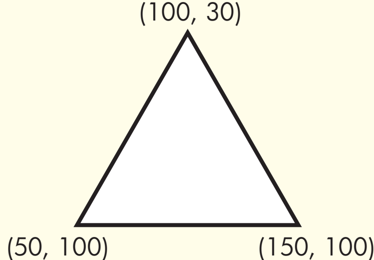
35. Determine the resolution of the screen on your computer (ask your instructor or the local computer center how to do this). Using this information, determine how many bytes of memory are required for the frame buffer to store the following:

a. A black-and-white image (1 bit per pixel)

b. A grayscale image (8 bits per pixel)

c. A color image (24 bits per pixel)

36. Using the *Draw_Line* commands described in Section 7.2, draw an isosceles triangle with the following configuration:



37. Discuss what problem the display hardware might encounter while attempting to execute the following operations, and explain how this problem could be solved.

```
AdaGraph.Draw_Line(1, 1, 4, 5,
  AdaGraph.Black);
```

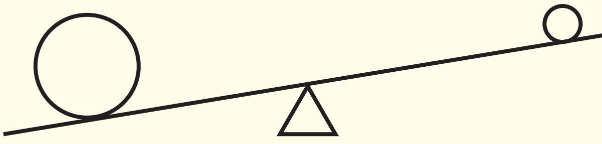
38. Draw a square with sides 100 pixels in length. Then inscribe a circle of radius 50 inside the square.

39. Create the following three labeled rectangular buttons in the output window.

Start | Stop | Pause

Have the space between the Start and Stop buttons be the same as the space between the Stop and Pause buttons.

40. Create the following image of a "teeter-totter":



41. Write a program that inputs the coordinates of three mouse clicks from the user and then draws a triangle in the output window using those three points. Here are some hints. Model the program after the code used to sense "mouse up" in the example with the Stop button.

In the *when* selector for *AdaGraph.Left_Up* consider the following code

```
counter := counter + 1;
if counter = 1
  then
    x1 := X;
    y1 := Y;
end if;
```

There will need to be two more *if* statements to capture the x, y values for the second and third clicks. The drawing code for the triangle will be in the body of the third *if* statement.

©2016 Course Technology, a part of Cengage Learning.

©2016 Course Technology, a part of Cengage Learning.

ANSWERS TO PRACTICE PROBLEMS

Section 2

1. The first three.

   `martinBradley` (camel case)

   `C3P_OH` (acceptable, although not used in this module)

   `Amy3` (Pascal case)

   `3Right` (not acceptable, begins with digit)

   `constant` (not acceptable, Ada reserved word)

2. `number : INTEGER;`
3. `taxRate : constant := 5.5;`
4. `hits(7)`

Section 3.1

1. ```
   TEXT_IO.PUT("Enter quantity: ");
   INT_IO.GET(quantity);
   ```
2. ```
   INT_IO.PUT(height, 6);
   TEXT_IO.NEW_LINE;
   ```
3. `This isgoodbye`

Section 3.2

1. next : = newNumber;
2. This code will not compile in Ada because of the mixed types in the assignment statement. It should be written as

   ```
   average := FLOAT(total/number);
   ```

   The output (with the default system formatting) will be

   ```
   5.50000E+01
   ```

   Note that *total/number* results in integer division. To get a true average, use

   ```
   average := FLOAT(total)/FLOAT(number);
   ```

Section 3.3

1. 30
2. 3
   5
   7
   9
   11
   13
   15
   17
   19
   21
3. Yes
4. 6

©2016 Course Technology, a part of Cengage Learning.

5.

```
if day = night
 then
  TEXT_IO.PUT("Equal");
end if;
```

Section 4 **1.**

```
--program to read an integer and
--write out its square

WITH TEXT_IO;

PROCEDURE PracticeProblem IS
 PACKAGE INT_IO IS NEW TEXT_IO.INTEGER_IO(INTEGER);

 number, square : INTEGER;

BEGIN

 TEXT_IO.PUT("Enter a number: ");
 INT_IO.GET(number);
 square := number ** 2;
 TEXT_IO.PUT("The square is: ");
 INT_IO.PUT(square);
 TEXT_IO.NEW_LINE;
END PracticeProblem;
```

2.

```
--program to compute cost based on price per item
--and quantity purchased

WITH TEXT_IO;

PROCEDURE PracticeProblem IS
 PACKAGE INT_IO IS NEW TEXT_IO.INTEGER_IO(INTEGER);
 PACKAGE FLO_IO IS NEW TEXT_IO.FLOAT_IO(FLOAT);

 quantity : INTEGER;
 price, totalCost : FLOAT;

BEGIN
 TEXT_IO.PUT("Enter the price of the item: ");
 FLO_IO.GET(price);
 TEXT_IO.PUT("Enter the quantity purchased: ");
 INT_IO.GET(quantity);
 totalCost := price * FLOAT(quantity);
 TEXT_IO.PUT("The total cost is: ");
 FLO_IO.PUT(totalCost, 3, 2, 0);
 TEXT_IO.NEW_LINE;
END PracticeProblem;
```

3.

```
--program to test a number relative to 5
--and write out the number or its double

WITH TEXT_IO;
```

©2016 Course Technology, a part of Cengage Learning.

```
PROCEDURE PracticeProblem IS
 PACKAGE INT_IO IS NEW TEXT_IO.INTEGER_IO(INTEGER);

 number : INTEGER;

BEGIN
 TEXT_IO.PUT("Enter a number: ");
 INT_IO.GET(number);
 if number < 5
  then
    TEXT_IO.PUT("The number is: ");
    INT_IO.PUT(number, 4);
  else
    TEXT_IO.PUT("Twice the number is: ");
    INT_IO.PUT(number * 2, 4);
 end if;
 TEXT_IO.NEW_LINE;
END PracticeProblem;
```

4.

```
-- program to collect a number, then write all
-- the values from 1 to that number

WITH TEXT_IO;

PROCEDURE PracticeProblem IS
 PACKAGE INT_IO IS NEW TEXT_IO.INTEGER_IO(INTEGER);

 number : INTEGER;
 i      : INTEGER;

BEGIN
 TEXT_IO.put("Enter a positive integer: ");
 INT_IO.GET(number);
 i := 1;
 while i <= number
  loop
    INT_IO.PUT(i, 4);
    TEXT_IO.NEW_LINE;
    i := i + 1;
  end loop;
END PracticeProblem;
```

Section 5.3

1. 11

2. The Ada compiler message is:

PracticeProblem.adb:10:04: assignment to “IN” mode parameter not allowed

A change is being made to an item passed (by default) by value. Ada will not allow the code to compile without adding *in out* to the parameter list—see Practice Problem 1 above.

3.

```
PROCEDURE getInput(one : out INTEGER; two : out
                   INTEGER) IS
 BEGIN
  TEXT_IO.PUT("Input the value for One: ");
  INT_IO.GET(one);
```


©2016 Course Technology, a part of Cengage Learning.

```
  TEXT_IO.PUT("Input the value for Two: ");
  INT_IO.GET(two);
END getInput;
```

4.

```
a.  FUNCTION tax(subTotal : in FLOAT) RETURN FLOAT IS
b.  return subTotal * rate;
c.  FLO_IO.PUT(tax(subTotal), 3, 2, 0);
```

Section 6.4

1. The area of a square with side 10.00 is 100.00
2. height and base

Section 7.2

```
AdaGraph.Clear_Window (AdaGraph.White);

AdaGraph.Draw_Box (50, 110, 250, 260, AdaGraph.Black,
                   AdaGraph.No_Fill);
AdaGraph.Draw_Box (60, 120, 100, 160, AdaGraph.Black,
                   AdaGraph.No_Fill);
AdaGraph.Draw_Box (200, 120, 240, 160, AdaGraph.Black,
                   AdaGraph.No_Fill);
AdaGraph.Draw_Box (125, 180, 175, 260, AdaGraph.Black,
                   AdaGraph.No_Fill);
AdaGraph.Draw_Line (50, 110, 150, 10, AdaGraph.Black);
AdaGraph.Draw_Line (250, 110, 150, 10, AdaGraph.Black);
AdaGraph.Draw_Circle (165, 215, 5, AdaGraph.Black,
                      AdaGraph.No_Fill);
```


©2016 Course Technology, a part of Cengage Learning.