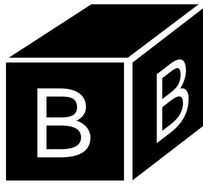


Chapter *3*

Modules and Interfaces

The BlackBox framework consists of tools to help the software designer write application programs. Typical applications present dialog boxes and menu options to the user. When the user enters data into a dialog box and clicks a button with the mouse or makes a menu selection, a program that the software designer wrote is activated. The framework maintains the necessary connections between the programs and the actions of the user. To accomplish the required connections the BlackBox framework uses three collections:

- Modules
- Classes
- Procedures

Each of these collections groups various items together into a single unit. This chapter shows how to organize simple procedures into modules. Later chapters show how to use classes.

Modules

The *module* is the outermost collection of classes, procedures, and data. Every Component Pascal program you write must be contained within a module, which is also known as a *compilation unit*. Figure 3.1 shows one possible organization of a module. It is a collection of some data, a procedure, and a class. A procedure groups data and program statements together.

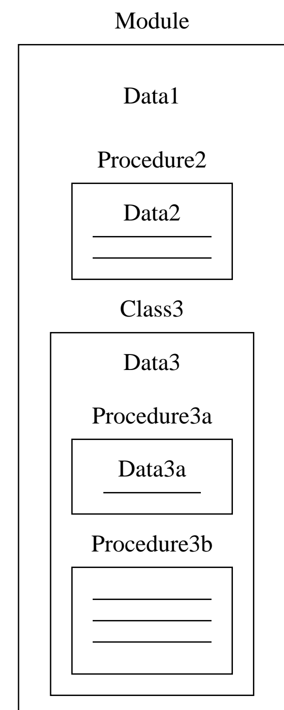


Figure 3.1
A module containing a procedure and an object.

In Figure 3.1, the word Data1 represents data that is contained in the module but is not contained in a procedure. Data that is not contained in a procedure or a class is called *global* data in contrast to *local* data, which is. Data3 is local to Class3, and Data3a is local to Procedure3a.

The two lines in Procedure2 represent program statements. The program statements are grouped so they can be executed as if they were a single statement. A class can collect several procedures together as well as data. Many combinations of collections are possible. Figure 3.1 shows that some procedures may have data while others may not. You can also put procedures inside other procedures, but we will not have occasion to do so in this book. Putting procedures inside a class is the organization required for the design technique called *object-oriented programming* (OOP). The latter part of this book describes principles of OOP.

Interfaces

The BlackBox framework is a collection of modules that you will use to write your programs. Your programs will consist of procedures that are contained within a module that you will write. The modules of the framework all fit together and cooperate to provide services for the programmer's module. A programmer thinks of a particular module in the framework as providing a service for him in much the same way that a professional, say an attorney, provides a service for her customers. The terminology from the commercial world is often carried over to computer science. In the same way that the attorney provides a service to her clients, a module in the framework is a *server* that provides a service to the programmer's module, which is the *client*.

The client/server view

Figure 3.2 shows the relationship between the client module, which you will write, and the server module, which is provided by the framework. The interaction between the two modules is governed by specific rules or protocols that are defined by the server module in its interface. The *interface* of a module is a list of all the items that are exported by the module. Its purpose is to describe the rules that a client module must follow to use its services. You should develop the skill of reading the interface of a module to determine the rules to be followed when requesting its services.

The interface and its purpose

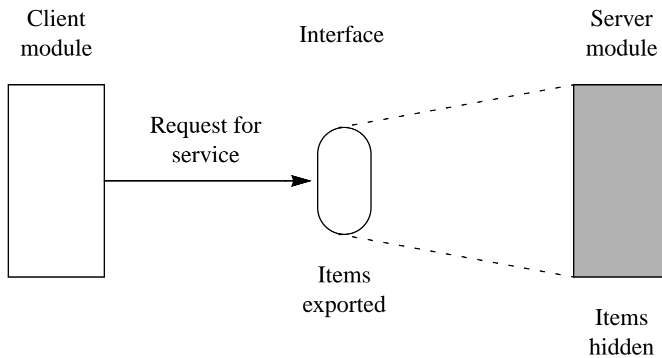


Figure 3.2
The interface between a client module and a server module.

The server module can export many kinds of items including data, procedures, and classes. The client module has access to the items exported by the server. Any items that are not exported by the server are hidden from the client and are not accessible. If the server is written well, knowledge about the items that are hidden will not be needed anyway by the client to perform its task. The hiding of detail is the essence of *abstraction*, and is an important idea in software design. The server module is darkly shaded in Figure 3.2 to indicate that its details cannot be seen by the client. This concept is so important that the representation of a server module as a "black box" whose details are hidden is the inspiration for the name of the BlackBox framework.

The essence of abstraction

If you want to use a module but you are not sure of the exported items, the framework provides a convenient way for you to view the module's interface. You simply type the name of the module, highlight it in a stretch of text, and select Info→Interface from the menu bar. For example, Figure 3.3 shows how you could view the interface of a module named StdLog, which you will use for your first program. The

name of the module, StdLog, has been typed in the Log and selected. With this stretch of text in the focus window, Info→Client Interface is selected from the menu bar. The result is a new window with the text shown in Figure 3.4.

Figure 3.3
Highlighting a stretch of text to access the interface of a module.

You can always tell when you are inspecting an interface by the first word DEFINITION in the listing. The items listed between DEFINITION and END are the items exported by the module. This module exports many items, six of which are shown here—the procedures Bool, Char, Int, Ln, Real, and String. Your first program will use procedure String from module StdLog.

```
DEFINITION StdLog;

   PROCEDURE Bool (x: BOOLEAN);
   PROCEDURE Char (ch: CHAR);
   PROCEDURE Int (i: LONGINT);
   PROCEDURE Ln;
   PROCEDURE Real (x: REAL);
   PROCEDURE String (IN str: ARRAY OF CHAR);

END StdLog.
```

Figure 3.4
The interface of the StdLog module. Not all exported items are shown.

Compilers

A computer can directly execute statements only if they are written in the language that the machine can understand. Languages for machines are written in a complex code that is difficult for humans to read or write. The code is called machine language. So a Component Pascal statement must first be translated to machine language before executing. The function of the *compiler* is to perform the translation from a program written in Component Pascal to machine language. The compiler also generates the interface between the program and the rest of the framework. Running a program is a three-step process:

The compiler as a translator

- Write the program in Component Pascal, called the *source program*.
- Invoke the compiler to translate, or compile, the source program from Component Pascal to machine language. The machine language version is called the *object program* and is stored in the code file. The interface is stored in the symbol file.

- Execute the object program.

If you want to execute a program that was previously compiled, you do not need to translate it again. You can simply execute the object program directly. If you ever delete the object program from your disk you can always get it back from the source program by compiling again. But the translation can only go one way. If you ever delete the source program you cannot recover it from the object program.

The Component Pascal compiler is software, not hardware. It is a program that is stored in a file on your disk. In the BlackBox framework, the compiler is located in the development subsystem, abbreviated Dev. Like all programs, the compiler has input, does processing, and produces output. Figure 3.5 shows that the input to the compiler is the source program and the output is the object program and the interface.

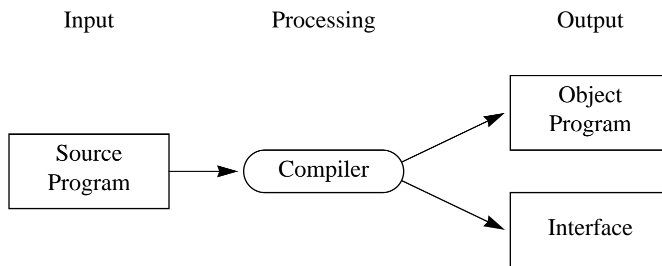


Figure 3.5
The compiler as a program.

When you write the source program, it will be saved in a file on disk just like any other document would be. The text files you wrote in the first chapter were saved in files stored in the Docu folder. You should save your Component Pascal source programs in the Mod folder. When you invoke the compiler, it will produce the code file for the object program, and the framework will save it in the Code folder. The compiler will also produce the interface, which the framework will save in the Sym folder. Both the object program and the interface are created automatically by the compiler from your source program. Most other programming languages require the programmer to write not only the source program, but the interface as well. If you have used such a language before, having the interface produced automatically might take some getting used to. The way in which BlackBox manages the interfaces of the modules is a major benefit over other development systems.

Programs

Figure 3.6 is a program that outputs a message to the Log. To run this program, you should first select File→New and type the listing in the untitled document window as it is shown in the figure. Be particularly careful about the punctuation marks. There are several differences in the program that you should type compared to the program in Figure 3.6.

Every module must have a name. In Figure 3.6 the name of the module is Hw90Pr0380. When you develop software for a large project that requires many modules, it is important to have a consistent naming convention for your modules and your files. In the process of studying from this book, you will be writing many

modules and so will need a consistent naming system. The guidelines for naming the modules in this chapter are a system that is appropriate for programs that are written as assignments in a class. Your instructor may have different guidelines for you to follow.

```
MODULE Hw99Pr0380;
(* Stan Warford *)
(* June 12, 2002 *)

    IMPORT StdLog;

    PROCEDURE PrintAddress*;
    BEGIN
        StdLog.String("Mr. K. Kong"); StdLog.Ln;
        StdLog.String("Empire State Building"); StdLog.Ln;
        StdLog.String("350 Fifth Avenue"); StdLog.Ln;
        StdLog.String("New York, NY 10118-0110"); StdLog.Ln
    END PrintAddress;

END Hw99Pr0380.
```

Figure 3.6
Sending output to the Log.

Chapter 1 described a system where each student in the class is assigned a unique two-digit number. The name of the module in Figure 3.6 is appropriate for a student who has been assigned the number 99. The first part of the name Hw99 consists of the letters Hw, which stands for homework, followed by the assigned student number. The second part of the name Pr0380 assumes that this program is a homework assignment as specified in Chapter 3, Problem 80. You must be careful to distinguish between uppercase and lowercase letters. In the name Hw99Pr0380, H and P are uppercase, while w and r are lowercase. When you type a program for an assignment use your assigned number in place of 99, the chapter from which the assignment is taken in place of 03, and the problem number in place of 80.

After you have entered the text, select File→Save As to save the file as a document (not Ascii or Plain Text). Manipulate the controls of the dialog box so the document will be saved in the folder named Mod within the folder named Hw99 that you previously installed in your BlackBox folder. Name the file Pr0380 when you save it. Note that the first part of the module name Hw99 is the name of your folder contained in the BlackBox folder, while the second half of the module name Pr0380 is the name of the file that is within the Mod folder that is within your folder.

You can see from Appendix A how Figure 3.6 conforms to the structure for a module. The EBNF syntax rule for Module from the appendix is

MODULE Ident “ ; ” [ImportList] DeclSeq [BEGIN StatementSeq] [CLOSE StatementSeq] END Ident “ . ”

In this program Ident is Hw99Pr0380, there is an ImportList containing StdLog, the declaration sequence DeclSeq corresponds to the procedure named PrintAddress with its own BEGIN and END, there is no BEGIN StatementSeq part, and there is no CLOSE StatementSeq part. Notice how the module must terminate with a period.

Comments

The documentation section at the beginning of the module in Figure 3.6 is enclosed in comment brackets, (* and *). The compiler ignores everything between the brackets. The only purpose of the documentation section is to provide information to a human reader. The comments in this module list the programmer's name and the date the program was written. You can write a comment anywhere that a blank space can occur and not affect the program execution. All your modules should contain a documentation section with at least your name and the date you wrote the program. In the BlackBox framework, documentation about how to use a module is placed in the Docu file, which is described later.

Reserved words

Figure 3.6 has five reserved words—MODULE, IMPORT, PROCEDURE, BEGIN, and END. Reserved words have special meaning to the Component Pascal compiler. The reserved word MODULE indicates to the compiler the start of a Component Pascal module. The reserved word IMPORT tells the compiler that another module, StdLog in this case, will be used by this module. PROCEDURE indicates the beginning of a procedure declaration. BEGIN indicates the start of a list of Component Pascal statements, and END indicates the end of the list. Component Pascal has 40 reserved words. They are:

Component Pascal's reserved words

ABSTRACT	ELSIF	LIMITED	RECORD
ARRAY	EMPTY	LOOP	REPEAT
BEGIN	END	MOD	RETURN
BY	EXIT	MODULE	THEN
CASE	EXTENSIBLE	NIL	TO
CLOSE	FOR	OF	TYPE
CONST	IF	OR	UNTIL
DIV	IMPORT	OUT	VAR
DO	IN	POINTER	WHILE
ELSE	IS	PROCEDURE	WITH

These are terminal symbols in the grammar in Appendix A.

Identifiers

The name of the module is Hw99Pr0380 and the name of the procedure in the module is PrintAddress. Both names are Component Pascal identifiers determined arbitrarily by the programmer. You could just as easily call the procedure OutputName instead of PrintAddress. In that case, the first line of the listing after the documentation section would be

```
PROCEDURE OutputName*;
```

Items other than modules and procedures can be named by Component Pascal identifiers. Regardless of the item named, you must follow the rules for devising an identifier. Component Pascal identifiers may contain only letters, digits, and under-

score characters, and they must start with a letter or underscore character. The EBNF syntax rule for Ident from the appendix is

(Letter | “ _ ”) {Letter | “ _ ” | Digit}

An identifier can consist of more than one word, but the words may not be separated by a space. If an identifier contains more than one word you should capitalize the first letter of the word to make it easily readable.

Example 3.1 Here are five legal Component Pascal identifiers:

NewYork   DC9   quantityOnHand   i   hoursWorked

Notice how much easier it is to read the identifier quantityOnHand instead of quantityonhand.

Component Pascal distinguishes between uppercase and lowercase characters in identifiers or reserved words. So, hours and Hours would be detected by the compiler as different identifiers. Reserved words may not be used as Component Pascal identifiers.

Example 3.2 Here are some illegal Component Pascal identifiers:

7Eleven   Tax%   home-Address   TO

The first is illegal because it does not begin with a letter. The second and third have characters other than letters or digits. The last is illegal because it is a reserved word.

Unlike Standard Pascal and Oberon-2, Component Pascal allows the underscore character in its identifiers. The rule to allow the underscore character is provided mainly for compatibility with the widespread Java and C++ programming languages.

Example 3.3 Here are some legal Component Pascal identifiers that use the underscore character.

new_york   quantity_on_hand   hours_worked

To write a Component Pascal program you must make up identifiers to name items. You should get in the habit of using mnemonic identifiers, that is, identifiers that remind the human reader about the meaning of the item you are naming. PrintAddress is a good name for the procedure in module Hw99Pr0380 because the program prints an address on the Log. The program would execute exactly the same if you wrote

```
PROCEDURE Xyz*;
```

But that would be horrible style, because the identifier indicates nothing about what the program does. Even worse would be

```
PROCEDURE Payroll*;
```

for this module, because that would indicate to the human reader that the program has something to do with a payroll problem, which it does not. When you use a program from this book as a model for your own program, do not blindly copy the identifiers if they are not appropriate to your problem. Instead, make up your own mnemonic identifiers.

Exporting and importing procedures

In the same way that a large company is subdivided into several departments, a large software project is subdivided into several modules. For a company to function effectively, people within a department must be able to communicate with people in other departments. Similarly, for a large program to function effectively, communication must take place between entities in different modules. It is the responsibility of the software designer to specify how the communication between modules is to take place.

To maintain an orderly flow of work, most companies place some restriction on the lines of communication between departments. For example, a manufacturing worker usually is not allowed to walk in to the legal department and ask the company's attorneys about the latest legal issues the company is dealing with. Indeed, some of the information in the legal department might be privileged information that should be kept hidden from production workers.

In the same way that information is hidden between departments of a company, information can be hidden within a module. In Component Pascal, hiding information within a module is the default rule. That is, all items are hidden within a module and are not accessible to other modules unless the programmer makes them accessible. When a module *exports* an item, it gives permission for another module to use it. If a module wants to use an item that another module has exported, it must *import* the item.

The line

```
IMPORT StdLog;
```

Importing a module

in Figure 3.6 indicates to the compiler that module Hw99Pr0380 wants to import all of the items exported by module StdLog. Module StdLog, whose interface is shown in Figure 3.4, contains a collection of procedures that output messages to the Log.

Exporting a module

The asterisk after the name of the procedure PrintAddress* is called an *export mark*. The asterisk indicates to the compiler that this procedure is to be made available to other modules that want to import it. We will see later that items other than procedures can be exported and imported.

Statements

The statement

```
StdLog.String("Mr. K. Kong")
```

causes the phrase Mr. K. Kong to be printed on the Log. This statement is using the procedure named String from the module named StdLog. If module Hw99Pr0380 had not imported module StdLog, this statement would produce an error. To execute a procedure from an imported module, you must type the name of the module followed by a period followed by the name of the procedure you want to execute.

Following the name of the procedure String are a pair of parentheses ( ). Within the parentheses is the parameter "Mr. K. Kong". The procedure prints the content between the double quote to the Log. It is permitted to use single quotes instead of double quotes. Hence the statement

```
StdLog.String('Mr. K. Kong')
```

would produce the same output to the Log. If you want a double quote to be printed to the Log you must enclose the phrase by single quotes and vice versa.

Example 3.4 The statement

```
StdLog.String('He said, "Hello". How are you?')
```

prints the phrase

```
He said, "Hello". How are you?
```

to the Log. ▮

The statement

```
StdLog.Ln
```

executes the procedure named Ln from the module StdLog. Unlike procedure String from the same module, this procedure has no parameter. Procedure Ln sends the cursor in the Log to the beginning of the next line, which causes the second StdLog.String to place its parameter below the first one.

You can see from Figure 3.4 that module StdLog exports six procedures. Module Hw99Pr0380 uses two of them—String and Ln. There are no parentheses following Ln indicating that this procedure has no parameters. String does have parentheses following it. The item

```
IN str: ARRAY OF CHAR
```

that is contained between parentheses is called the *signature* of procedure String. str is the name of the parameter and ARRAY OF CHAR is its type. str is called the *formal* parameter. It matches the *actual* parameter "Mr. K. Kong" in the procedure call.

Signatures, formal parameters and actual parameters

The meaning of IN in the signature is described later. The parameters in the interface are guidelines for the use of the procedure in the importing module. The type ARRAY OF CHAR in the formal parameter list indicates that the actual parameter must be an array of characters, which is what "Mr. K. Kong" is.

You can get more extensive information about a documented module from the framework by highlighting the module name and selecting Info→Documentation. You should try this now to inspect the documentation of module StdLog.

All the lines between BEGIN and END in module Hw99Pr0380 are called executable statements, because they perform an operation when the program executes. Component Pascal has the following 11 executable statements:

assignment	if	return
case	loop	while
exit	procedure	with
for	repeat	

We will consider them in later chapters. The StdLog.String statement is an example of a procedure statement, or procedure call. IMPORT is an example of a nonexecutable statement. It has an effect during the compile phase as opposed to the execute phase.

You may have noticed in procedure PrintAddress that a semicolon appears after each StdLog statement except for the last one. This production rule from Appendix A for a statement sequence

StatementSeq = Statement {“ ; ” Statement}

shows that semicolons are used to separate two statements. For example, use the semicolon after the first StdLog.String statement to separate it from the following StdLog.Ln statement. Then use the semicolon after the StdLog.Ln statement to separate it from the following StdLog.String. The reserved word END, however, is not a statement. Therefore, you do not need a semicolon separator after the last StdLog.Ln. The general rule to remember is

- Do not place a semicolon before an END.

Syntax errors

How to compile a Component Pascal program

Now that the program has been written and saved in the Mod folder, it is time to attempt a compile. With the source window in focus select Dev→Compile And Unload to translate the program. If your program has no errors, the compiler will create the object program and the interface and store them on your disk. But how does the compiler determine where to store them? It examines the name of the module, scanning it from left to right. It assumes the first letter of the module name is the first letter of the project folder. Then it assumes that every letter and digit after the first character up to but not including the first uppercase letter is also part of the project folder. The rest of the module name is taken to be the name of the file where the object program and the interface are to be stored. The object file is stored in the Code folder and the interface is stored in the Sym folder.

Example 3.5 The module in Figure 3.6 is named Hw99Pr0380. The programmer saved it in the file BlackBox/Hw99/Mod/Pr0380. When this module was compiled by selecting Dev→Compile And Unload, the compiler scanned the name Hw99Pr0380 from left to right until it reached the uppercase P. It determined that the project folder was Hw99 and that the name of the file is Pr0380. Therefore, it stored the object file in BlackBox/Hw99/Code/Pr0380 and the interface in BlackBox/Hw99/Sym/Pr0380. In the end there were three files, all named Pr0380, in three different folders—the source file in the Mod folder, the object file in the Code folder, and the interface in the Sym folder. ▮

When you try to execute a program, two types of errors are possible:

- Syntax error—The program does not compile.
- Logical error—The program compiles but produces incorrect results.

The production rules of the grammar can only indicate possible sources of syntax errors, not logical errors. Remember from Chapter 2 that they are not even perfect at specifying all the syntax rules of the Component Pascal language.

Figure 3.7 illustrates a syntax error. When a program does not compile, no object program can be generated, and it is impossible to test for logical errors. Errors, whether syntax or logical, are called bugs. Getting the errors out of your program is called debugging. Can you spot the bug in Figure 3.7?

```
MODULE Hw99Pr0381;
(* Stan Warford *)
(* June 12, 2002 *)

   IMPORT StdLog;

   PROCEDURE PrintAddress*
   BEGIN
      StdLog.String("Mr. K. Kong"); StdLog.Ln;
      StdLog.String("Empire State Building"); StdLog.Ln;
      StdLog.String("350 Fifth Avenue"); StdLog.Ln;
      StdLog.String("New York, NY 10118-0110"); StdLog.Ln
   END PrintAddress;

END Hw99Pr0381.
```

Figure 3.7
This procedure has a bug.

If you try to compile this module the following error message will be printed on the Log:

```
compiling "Hw99Pr0381"
one error detected
```

The compiler will also place a marker symbol in the window of your source text that indicates where it detected the error and will give a more detailed description of the error. Clicking the error marker causes the marker to expand to reveal an error message. On MSWindows, the error message is also displayed at the bottom of the win-

dow. You can then figure out what the error was and make the correction with your text editor. After you make any changes you should select File→Save to save your changed file then Dev→Compile And Unload again. It is not necessary to remove the error marker symbols in your text before attempting another compile. The compiler automatically removes all error markers before it attempts a translation. Repeat the correction process until you get a successful compilation. When you succeed, a message on the Log will inform you of the translation it made.

Documentation files

Now that you have a program written and translated, you need to provide a way for the user to execute your program. In the BlackBox framework, user documentation is stored in the Docu folder. Figure 3.8 shows the documentation for module Hw99Pr0380.

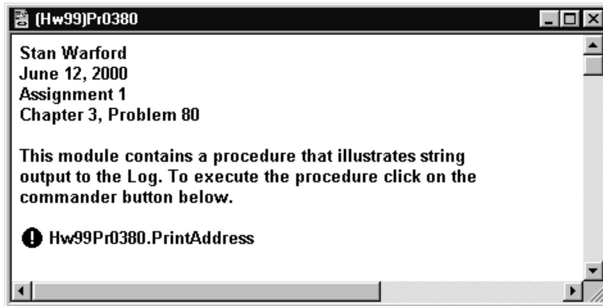


Figure 3.8
Documentation for module Hw99Pr0380, It is stored as file Pr0380 in the Docu folder.

To create user documentation you select File→New and enter a description of the program and instructions for the user on how to use it. The document in Figure 3.8 is not compiled. So, the comments contained in it do not need to be enclosed in comment brackets (* and *) as are the comments in the source listing.

Do not use comment brackets in your Docu file.

The instructions for the user shown in the figure include a commander button that the user should click to execute the program. Many of the programs you will write with this book can be conveniently executed by providing the user with a *commander button* in the user documentation. To insert the commander button select Tools→Insert Commander with your insertion cursor in your documentation window at the location where you want the button. (Be careful to *not* select the option Controls→Insert Command Button, which sounds similar but is quite different.)

The commander button

Following the commander button you place the command to be executed. A command is a procedure that is exported by a module. The syntax is identical to that used in a module to execute an imported procedure. Namely, following the commander button you place the name of the module followed by a period followed by the name of the exported procedure.

This documentation should be saved as a file named Pr0380 in BlackBox/Hw99/Docu. We now have four files all named Pr0380 stored in four different folders as shown in Figure 3.9. You write and save the source program in the Mod folder and the documentation in the Docu folder. The BlackBox system creates and stores the object program in the Code folder and the interface in the Sym folder.

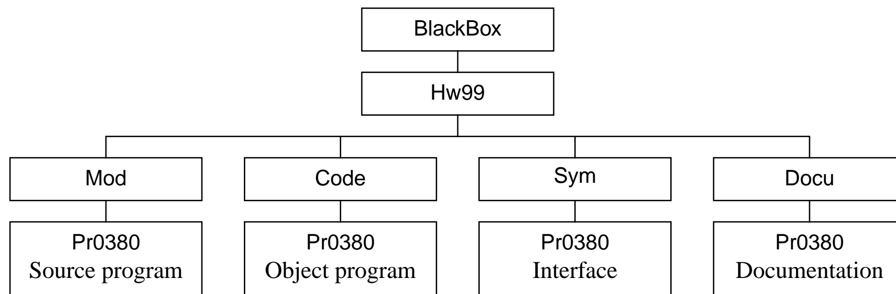


Figure 3.9
The files associated with a BlackBox program.

When you click the commander button the following text should appear on the Log.

```
Mr. K. Kong
Empire State Building
350 Fifth Avenue
New York, NY 10118-0110
```

Even the modules that you write have interfaces. Figure 3.10 shows the interface for module Hw99Pr0380. As the programmer of the module Hw99Pr0380, you do not write the interface. The compiler produces it automatically. The interface in Figure 3.10 was produced by highlighting the text Hw99Pr0380 in some document and selecting Info→Interface (after the module was compiled). You do not need to type the name of the module in the Log to select it. For example, you can select the name of your module in your source code document to bring up its interface. The export marks are omitted in the interface because they would be redundant. Every item listed in the interface is an exported item.

```
DEFINITION Hw99Pr0380;

   PROCEDURE  PrintAddress;

END Hw99Pr0380.
```

Figure 3.10
The interface for the program in Figure 3.6.

Program style

Some computer languages are line-oriented, that is, each statement must be written on a separate line. Listing 3.11 shows that Component Pascal is not line-oriented. The behavior of the object program does not depend on the spacing or indentation style of the source program. The program in Listing 3.11 produces the same output as the one in Listing 3.6.

```
MODULE Hw99Pr0382;
(* Stan Warford *)
(* June 18, 2002 *)

IMPORT StdLog;

PROCEDURE PrintAddress    *;
BEGIN StdLog.String       ("Mr. K. Kong"); StdLog.Ln;
StdLog.String
("Empire State Building");      StdLog.Ln
; StdLog.String    ("350 Fifth Avenue"); StdLog.Ln;
StdLog.String
("New York, NY 10118-0110"); StdLog.Ln
END PrintAddress; END Hw99Pr0382.
```

Figure 3.11
This module compiles without error.

One good habit to cultivate when learning to program is to adhere to a consistent standard of style. You should follow either the style of the programs in this book, the style specified by your instructor or employer, or a consistent style from some other source. The document titled Programming Conventions in the BlackBox on-line help system has detailed guidelines for Component Pascal programming style. The style conventions in this book have only a few differences from the published guidelines. The most noticeable difference is that the guidelines specify that all comments be in italic. Both this book and the published guidelines recommend that all exported procedures in a module be in a bold font.

The importance of good style

The computer does not require such neatness for the program to work. However, just getting the program to work correctly is not sufficient. Good style is necessary because people, as well as computers, must read your programs. You would not write a business letter without the paragraphs indented consistently. Nor should you write a program that way. Although you may want to rebel at first against such seemingly trivial details, you will find in the long run that they are not restrictive at all. In fact just the opposite is true—these rules are liberating.

The situation is similar to that of a new driver on the road for the first time. Think of how many restrictive rules there are—speed limits, yield signs, stop signals, and so on. New drivers may feel hampered and may worry about all the rules they need to remember. But experienced drivers do not even consciously try to remember the rules. They know them subconsciously. What's more, the rules liberate them from fear of an accident. Programming standards will liberate your mind to think constructively. The standards will take care of the details, freeing you to take care of the problem.

Proper procedures

The next two modules introduce the concept of a programmer-defined procedure, which will be discussed in more detail in later chapters. Procedures are useful when your program has a task that it needs to perform more than once. Programmers working with procedures must first define the task in the procedure declaration part

then invoke or call the procedure when the task needs to be executed. The program in Figure 3.12 outputs a pattern on the Log.

```
MODULE Hw99Pr0383;
   IMPORT StdLog;

   PROCEDURE PrintPattern*;
   BEGIN
      StdLog.String("@"); StdLog.Ln;
      StdLog.String("@@"); StdLog.Ln;
      StdLog.String("@@@"); StdLog.Ln;
      StdLog.String("@@@@"); StdLog.Ln;
      StdLog.String("@"); StdLog.Ln;
      StdLog.String("@@"); StdLog.Ln;
      StdLog.String("@@@"); StdLog.Ln;
      StdLog.String("@@@@"); StdLog.Ln;
      StdLog.String("@"); StdLog.Ln;
      StdLog.String("@@"); StdLog.Ln;
      StdLog.String("@@@"); StdLog.Ln;
      StdLog.String("@@@@"); StdLog.Ln
   END PrintPattern;

END Hw99Pr0383.
```

Figure 3.12
This procedure prints three triangles to the Log.

The exported procedure PrintPattern in Figure 3.12 outputs a pattern of asterisks without using another procedure. The pattern is a repetition of three smaller patterns in the shape of a triangle. The module in Figure 3.13, on the other hand, collects the statements that print a single triangle into another procedure that is not exported. The programmer declared the procedure and gave it the name PrintTriangle. She then called the procedure three times to produce the final pattern.

In Figure 3.13, PrintTriangle is an identifier that names the procedure. Figure 3.14 shows that procedures PrintTriangle and PrintPattern are both nested in module Hw99Pr0384. The StdLog.String statements in region (1) belong to procedure PrintTriangle. The procedure call statements in region (2) belong to the procedure PrintPattern.

When PrintPattern is invoked the first statement in region (2) is executed. It is a call to procedure PrintTriangle defined earlier in the listing, and causes execution to jump to the first statement of region (1). The computer then executes all the statements of region (1). After it executes the last statement of region (1), it transfers execution to the statement after the one that made the call in the calling procedure. At this point the first triangle has been printed.

```
MODULE Hw99Pr0384;
   IMPORT StdLog;

   PROCEDURE PrintTriangle;
   BEGIN
      StdLog.String("@"); StdLog.Ln;
      StdLog.String("@@"); StdLog.Ln;
      StdLog.String("@@@"); StdLog.Ln;
      StdLog.String("@@@@"); StdLog.Ln
   END PrintTriangle;

   PROCEDURE PrintPattern*;
   BEGIN
      PrintTriangle;
      PrintTriangle;
      PrintTriangle
   END PrintPattern;

END Hw99Pr0384.
```

Figure 3.13
This procedure prints three triangles to the Log using a procedure.

Next, the computer executes the second statement in region (2), which is another call to procedure PrintTriangle. So, the statements in region (1) execute again. Similarly, the third statement in region (2) makes them execute a third time. In general, a procedure call causes control to jump to the previously defined procedure. After the procedure executes, control returns to the statement after the calling statement. Figure 3.15 shows the order in which statements are executed in a module that has ProcedureP2, which is exported, making two procedure calls to ProcedureP1, which is not exported.

Because PrintTriangle is an identifier, the programmer determined it arbitrarily. The program would produce the exact output if the programmer wrote

```
PROCEDURE WriteTriangle
```

in the definition of the procedure, and then called it with

```
WriteTriangle
```

in the main program. The only requirement is that the name in the procedure definition match the name in the procedure call. Of course, the identifiers you choose for the names of your procedures should be mnemonic.

Procedures are useful when you need to perform the same task at several different points in a program. They are also useful in structuring a program into levels of abstraction, even if the task is only performed once. Although this program uses a procedure to output text to the Log, later programs will use procedures to process data as well.

The StdLog.String and StdLog.Ln statements are procedure calls. They differ from PrintTriangle in two respects. First, they are imported from a framework module as opposed to being user-defined procedures. Their declaration part is hidden

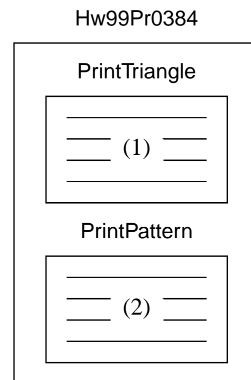
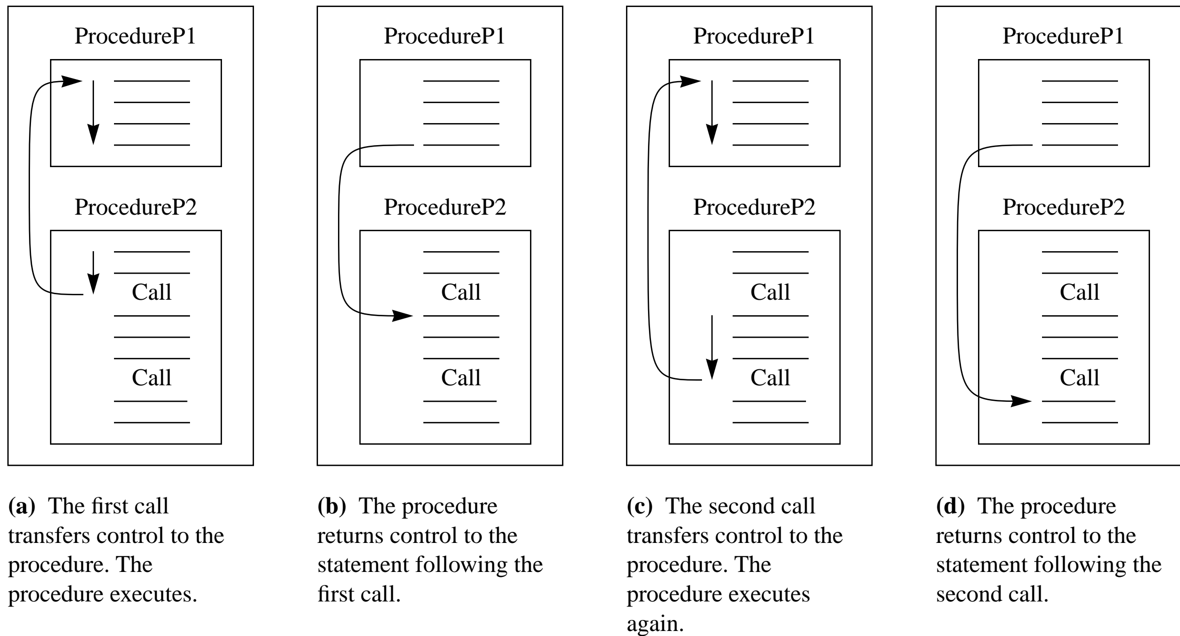


Figure 3.14
Procedures PrintPattern and PrintTriangle nested in module Hw99Pr0384.



(a) The first call transfers control to the procedure. The procedure executes.

(b) The procedure returns control to the statement following the first call.

(c) The second call transfers control to the procedure. The procedure executes again.

(d) The procedure returns control to the statement following the second call.

Figure 3.15 The order of execution when a procedure has two procedure calls.

from the programmer, although their interface is available on-line. This is in contrast to a programmer-defined procedure, which must be declared within the programmer's module.

Second, StdLog.String calls include a string parameter enclosed in parentheses. The declaration part needs this information to do its task. It needs the string to know what to output to the Log. Programmer-defined procedures can also have parameters. Later chapters will explain how to define procedures with parameters.

Figure 3.16 shows the interface for module Hw99Pr0384. Only the exported procedure PrintPattern shows in the interface. Because procedure PrintTriangle is not exported, it does not appear in the interface.

```
DEFINITION Hw99Pr0384;

   PROCEDURE PrintPattern;

END Hw99Pr0384.
```

Figure 3.16 The interface for the module in Listing 3.13.

Exercises

1. Define the following terms.

(a) module **(b)** global data **(c)** local data

2. What is an interface? What is its purpose?

3. What is the essence of abstraction?

4. What is the function of a compiler?

5. State whether each of the following Component Pascal identifiers is valid. For those that are not valid, explain why they are not.

(a) hourlyWage **(b)** last 1 **(c)** WITH
(d) one Name **(e)** 1stOne **(f)** %Profit
(g) Acme-Tool **(h)** A

6. State whether each of the following Component Pascal identifiers is valid. For those that are not valid, explain why they are not.

(a) amountOnHand **(b)** 2Day **(c)** superCallifragillistic
(d) BY **(e)** Soc-Sec-Num **(f)** John Smith
(g) tools/Bolts **(h)** i

7. Find all the syntax errors in the following Component Pascal procedure. Assume that module StdLog has been imported correctly.

```
PROCEDURE PrintString; *
BEGIN
    StdLog.String ('Is this wrong?'); StdLog.Ln;
    StdLog.String ('Maybe it's right'); StdLog.Ln
END;
```

8. Find all the syntax errors in the following Component Pascal procedure. Assume that module StdLog has been imported correctly.

```
PROCEDURE PrintString *;
BEGIN
    StdLog.String (“This can’t be wrong!); StdLog.Ln;
    StdLog.String (“Or can it?”); StdLog.Ln
END;
```

9. Inspect the interface of module TextViews, and answer the following questions about the procedures that are listed in it.

(a) Does procedure SetCtrlDir have a parameter list?
(b) Does procedure Deposit have a parameter list?
(c) What is the signature of procedure SetDir? What is the name of its formal parameter? What is its type?

Problems

10. Write a Component Pascal program to output the following two-line message on the Log:

```
She said, "Hi there.
What's up?"
```

Use two StdLog.String procedure calls for the second line to print both the single and the double quote marks. Test your program by inserting a commander button in a documentation file to execute the procedure.

11. Write a Component Pascal program to output to the Log your name and address suitable for use as a mailing label. Test your program by inserting a commander button in a documentation file to execute the procedure.

12. Using a procedure that is not exported to output a single pattern, write a Component Pascal program to output the following triple pattern on the Log:

```
+
+++
+++++
+++
+
+
+++
+++++
+++
+
+
+++
+++++
+++
+
```

Test your program by inserting a commander button in a documentation file to execute the exported procedure.