Section 1

The essence of COBOL programming

The best way to learn COBOL programming is to start doing it, and that's the approach the chapters in this section take. So in chapter 1, you'll learn how to write simple interactive programs that don't require input or output files. And in chapter 2, you'll learn how to enter, compile, and test COBOL programs using Micro Focus Personal COBOL. When you're done, you'll be able to develop interactive programs of your own.

Then, in chapter 3, you'll learn how to develop COBOL programs that prepare reports from the data in files. This type of program gets you started working with files, which COBOL is specifically designed to do. Since report-preparation programs are common to all business systems, all programmers need to know how to develop them.

Because "structured methods" are essential to productive programming on the job, the next two chapters show you how to use the best of these methods. In chapter 4, you'll learn how to design, code, and test a program using structured methods. In chapter 5, you'll learn additional features for coding structured COBOL programs.

When you complete this section, you'll have the essential skills that you need for designing, coding, and testing every program you develop. You'll also have a clear view of what COBOL programming is and what you have to do to become proficient at it. Then, you can add to your skills by reading the other chapters in this book.

Mike Murach & Associates

2560 West Shaw Lane, Suite 101
Fresno, CA 93711-2765
(559) 440-9071 • (800) 221-5528

murachbooks@murach.com • www.murach.com

1

Introduction to COBOL programming

The quickest and best way to *learn* COBOL programming is to *do* COBOL programming. That's why this chapter shows you how to code two simple but complete programs. Before you learn those coding skills, though, this chapter introduces you to COBOL.

COBOL platforms, standards, and compilers

COBOL is an acronym that stands for COmmon Business Oriented Language. Starting in 1959, this language was designed by representatives from business, government, and the U.S. Department of Defense. Their goal was to design a programming language that could be used on all business computers.

COBOL platforms

Today, COBOL is available on all of the major computer platforms. In general, you can think of a *platform* as a unique combination of computer hardware and operating system. Figure 1-1, for example, illustrates typical mainframe and PC hardware configurations and describes the operating systems that run on them.

In terms of hardware, all computer systems have the same basic components, including display screens, keyboards, disk drives, printers, processors, and internal storage. It's just the terminology and scale that differ from one computer to the next. On a *mainframe* or *mid-range computer*, for example, the display screen is a *terminal*; the processor is a *central processing unit*, or *CPU*; and the *internal storage* is *main memory*. On a *PC*, the comparable terms are *monitor*, *processor*, and *RAM* (*random-access memory*).

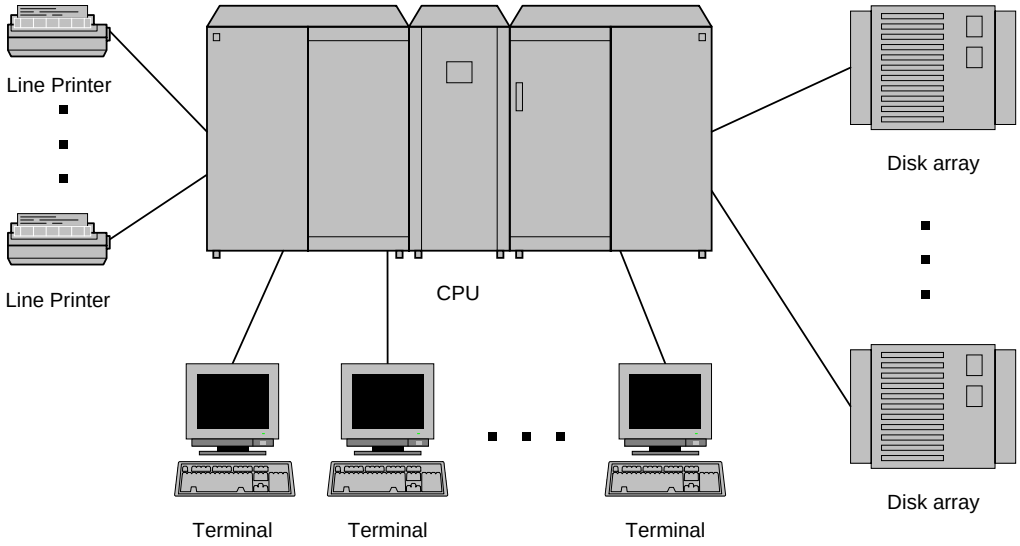
The primary difference between a mainframe computer and a PC is that a mainframe computer serves dozens or hundreds of users from a single processor, while a PC serves only one user. Like a mainframe computer, a mid-range computer serves more than one user, but usually far fewer than a mainframe.

Today, the IBM mainframe computer running under the OS/390 operating system is a widely used COBOL platform. The AS/400 running under the OS/400 operating system and mid-range systems that run under the UNIX operating system are two other common COBOL platforms. And PCs that run under DOS and Windows are two more COBOL platforms.

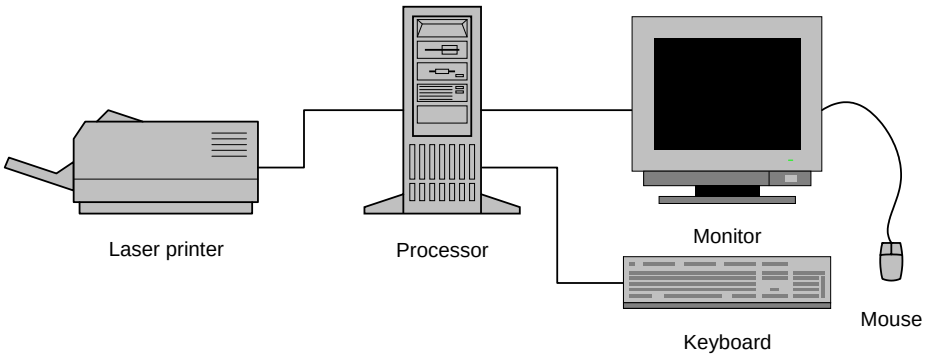
With this book, you'll learn how to develop COBOL programs on your own PC because that's the easiest way to get started. However, you're also going to learn how to write COBOL programs for IBM mainframes because that's by far the most popular COBOL platform. You should also be able to transfer the skills you learn in this book to any other COBOL platform.

Keep in mind, though, that writing COBOL programs for mainframes is by far the largest job market. Today, billions of lines of COBOL code are in use on this platform. During the 1990's, tens of billions of dollars were spent fixing these programs so they would work correctly when the year changed from 1999 to 2000. And many billions more will be spent to keep these programs up-to-date in the next 20 years.

A typical mainframe configuration



A typical PC configuration



Description

- The most widely-used COBOL *platform* is the *IBM mainframe* running under the OS/390 or MVS operating system. IBM AS/400 systems that run under the OS/400 operating system and *mid-range computers* that run under the UNIX operating system are two other common COBOL platforms. A *PC* running under the Windows operating system is another COBOL platform.
- The biggest difference between a mainframe and a PC is that a mainframe serves dozens or even hundreds of users from a single processor, while a PC serves only one user. A mid-range system usually serves many users, but not as many as a mainframe.

Figure 1-1 COBOL platforms

COBOL standards and compilers

In an effort to make COBOL work the same way on all computer platforms, several sets of *COBOL standards* have been published since 1968. These are summarized in figure 1-2. The first three sets were developed by the American National Standards Institute (ANSI), and the latest set is being developed by ANSI/ISO (International Standards Organization).

The first set of specifications, referred to as COBOL-68, got the process of standardization going, but had some serious limitations. In particular, these standards didn't provide for the use of indexed files. In contrast, the COBOL-74 standards provided language for most of the capabilities that were in common use, thus reducing COBOL variation from one computer system to another.

The next set of standards, called *COBOL-85*, added language that makes it easier to adhere to the principles of structured programming. Then, in 1989, an addendum to the COBOL-85 standards introduced intrinsic functions. In this book, you'll learn how to use all of the COBOL-85 features because that's the standard that's in common use today.

Right now, we're awaiting the *COBOL-2000* standards, which are scheduled for release in 2002. The primary enhancement of these standards is that they will provide for object-oriented programming (see chapter 22). Otherwise, the improvements are expected to be minimal.

With all this standardization, you may think that the COBOL language is the same on all computer systems. But it's not, for two major reasons. First, the COBOL standards still don't provide effective language for displaying data on the interactive screens of computer users and for receiving data that has been entered through their keyboards. Second, the COBOL standards still don't provide language for working with databases, even though databases are commonly used on all types of computer systems.

Before a COBOL program can be run on a specific computer system, the COBOL language must be *compiled* into the machine language that will run on that computer. This is done by a *COBOL compiler*, a program that's written specifically for that computer. For a mainframe, the computer manufacturer develops the compiler. For a PC, independent software companies develop the compilers. For a mid-range computer or a UNIX-based system, either the manufacturer or an independent company develops the compiler.

In figure 1-2, you can see the four compilers that are in common use on IBM mainframes today. In addition, you can see some of the Merant compilers for other platforms. Merant is a leading supplier of tools that let you offload mainframe program development to a PC.

When the standards that a compiler is based upon don't provide for a needed capability, the developer of the compiler can add non-standard *extensions* to the language. The resulting compiler can still be called a standard compiler. To be a COBOL programmer, then, you need to know the standard language as well as the extensions that apply to the compiler you're using. In this book, you'll learn the compiler specifics for Micro Focus Personal COBOL, as well as the specifics for the three IBM mainframe compilers that are based on the 1985 standards.

COBOL standards

Year	Description
1968	Although these standards were limited, they showed that a language could be standardized from one type of computer to another.
1974	These standards were much more complete than the 1968 standards so they led to a high degree of COBOL standardization from one computer to another.
1985	These standards provided enhanced language for developing structured programs. A 1989 addendum added intrinsic functions to these standards.
2000	These standards will add a number of features to the previous standards, including object-oriented language. They are scheduled for publication in 2002.

COBOL compilers for IBM mainframes

Compiler	Standard	Description
OS/VS COBOL	1974	This compiler has been phased out of most shops.
VS COBOL II	1985	This compiler is still used in many COBOL shops.
COBOL for MVS & VM	1985	This compiler runs under the MVS and VM operating systems.
COBOL for OS/390 & VM	1985 plus 1989 functions	This compiler runs under the OS/390 and VM operating systems.

COBOL compilers developed by Merant

Compiler	Description
Micro Focus Personal COBOL	An inexpensive COBOL compiler that is an excellent tool for learning COBOL on a PC. Developed by Micro Focus, now part of Merant.
Net Express	A development environment that takes core business processes written in COBOL and extends them to the Web and other distributed platforms.
Mainframe Express	A complete workbench for developing COBOL programs for mainframes. By setting an option, the compiler can emulate any mainframe compiler.
Object COBOL Developer Suite for UNIX	Provides an integrated environment for developing client/server and standalone applications for UNIX platforms and the leading relational databases.

COBOL compilers

- Before a COBOL program can be run on a computer, the COBOL language must be converted to machine language. This is called *compiling* a program, and it's done by a program called a *COBOL compiler*.
- Some COBOL compilers are based on the 1974 standards, some on the 1985 standards, and some on the new 2000 standards.
- If a compiler meets the specifications of a specific standard (like COBOL-85), it is called a standard COBOL compiler, even if it has non-standard *extensions.*

Figure 1-2 COBOL standards and compilers

An interactive COBOL program

With that as background, you're now going to learn how to write a simple interactive program. Before you see this program, though, you should understand the difference between an interactive and a batch program. In addition, you should understand the difference between the interactive program presented here and interactive programs developed for business.

An *interactive program* interacts with the user by displaying information on the screen and accepting data in response. This type of program is easy to start with because it doesn't require the use of files. That way, you can focus on the overall structure, content, and logic of the program rather than on the statements for handling files. Then, in chapter 3, you'll see a simple batch program that works with files. In contrast to an interactive program, a *batch program* runs without interacting with the user and typically processes the data in one or more disk files.

When you develop professional interactive programs, you probably won't use the techniques illustrated in this chapter. Instead, you'll use non-standard extensions. If you're developing programs for an IBM mainframe, for example, you'll probably use CICS (Customer Information Control System) as shown in chapter 19. And if you're developing programs for other platforms, you'll use non-standard COBOL that's designed for those platforms as shown in chapter 17. So keep in mind as you read this chapter that the programs are for training purposes only.

An interactive session

Figure 1-3 presents an interactive session as it's displayed on a PC monitor. Here, the shaded data is data that has been entered by the computer user. All of the other data is displayed by the COBOL program.

As you can see, this program calculates and displays the sales tax for the amount of a purchase. The sales tax rate in this case is 7.85%. When the user enters 100.00, for example, the program displays the number 7.85. And when the user enters 10.00, the program displays the number .79, which is the sales tax rounded to the nearest penny. If the user continues to enter numbers, the program will continue to display the sales tax for each number.

To end the session, the user enters 0 instead of another number. Then, after it displays END OF SESSION, the program ends.

An interactive session on a monitor or terminal

```
-----------------------------------------
TO END THE PROGRAM, ENTER 0.
TO CALCULATE SALES TAX, ENTER THE SALES AMOUNT.
100.00
SALES TAX =     7.85
-----------------------------------------
TO END THE PROGRAM, ENTER 0.
TO CALCULATE SALES TAX, ENTER THE SALES AMOUNT.
10.00
SALES TAX =      .79
-----------------------------------------
TO END THE PROGRAM, ENTER 0.
TO CALCULATE SALES TAX, ENTER THE SALES AMOUNT.
29.99
SALES TAX =     2.35
-----------------------------------------
TO END THE PROGRAM, ENTER 0.
TO CALCULATE SALES TAX, ENTER THE SALES AMOUNT.
0
END OF SESSION.
```

Description

- The shaded numbers above are the entries made by the computer user. All of the other characters are displayed by the COBOL program.
- When the user enters a sales amount other than zero, the program calculates the sales tax, displays the result, and asks for the next amount.
- When the user enters a zero, the program displays END OF SESSION and ends.

Figure 1-3 An interactive session for calculating sales tax

The COBOL code

Figure 1-4 presents the complete COBOL code for the interactive program. This code will run on any standard COBOL compiler. In the rest of this chapter, you'll learn how this program works.

For now, though, please notice that the program is divided into four divisions: the Identification Division, the Environment Division, the Data Division, and the Procedure Division. You'll learn more about the code in each of these divisions later in this chapter and throughout this book. For now, just realize that the division headers are required in every COBOL program.

Please notice also how the periods are used throughout the program. As you can see, they are used at the ends of division and section headers. They are used at the ends of paragraph names like 000-CALCULATE-SALES-TAX and PROGRAM-ID. They are used at the ends of the data descriptions in the Working-Storage Section of the Data Division. And they are used at the ends of the statements in the Procedure Division.

When a computer runs a COBOL program, it starts with the first executable statement in the Procedure Division. The computer then executes the statements that follow in sequence, unless this sequence is changed by a Perform or Perform Until statement. In this figure, the first statement is a Perform Until statement, which performs procedure 100-CALCULATE-ONE-SALES-TAX.

In this program, procedure 000 performs procedure 100 until the value in the field named END-OF-SESSION-SWITCH equals Y. Since this field starts with a value of N in the Working-Storage Section, the Perform Until statement always performs the 100-CALCULATE-ONE-SALES-TAX paragraph the first time through the code.

When a Perform Until statement performs another paragraph, the computer executes all of the statements in that paragraph, from the first to the last. When it's finished, it returns to the Perform Until statement, which is executed again. This continues until the condition in the Perform Until statement becomes true. The program then executes the statements that follow the Perform Until statement. In this program, the first statement that follows displays this message: END OF SESSION. The next statement ends the program.

If you have any experience with other programming languages, you should be able to understand how this code works without further help. In the Working-Storage Section, you can see the definitions of the fields that are used by the program. In the Procedure Division, you can see the statements that use these fields.

If, on the other hand, you don't have any programming experience, the pages that follow present everything you need to know for writing a program like this. As you read on, please refer back to the program in figure 1-4 whenever you need to see how the parts of the program fit together.

The interactive program

```
 IDENTIFICATION DIVISION.
*
 PROGRAM-ID. CALC1000.
*
 ENVIRONMENT DIVISION.
*
 INPUT-OUTPUT SECTION.
*
 DATA DIVISION.
*
 FILE SECTION.
*
 WORKING-STORAGE SECTION.
*
 77  END-OF-SESSION-SWITCH       PIC X       VALUE "N".
 77  SALES-AMOUNT                PIC 9(5)V99.
 77  SALES-TAX                   PIC Z,ZZZ.99.
*
 PROCEDURE DIVISION.
*
 000-CALCULATE-SALES-TAX.
*
     PERFORM 100-CALCULATE-ONE-SALES-TAX
         UNTIL END-OF-SESSION-SWITCH = "Y".
     DISPLAY "END OF SESSION.".
     STOP RUN.
*
 100-CALCULATE-ONE-SALES-TAX.
*
     DISPLAY "-------------------------------------------".
     DISPLAY "TO END PROGRAM, ENTER 0.".
     DISPLAY "TO CALCULATE SALES TAX, ENTER THE SALES AMOUNT.".
     ACCEPT SALES-AMOUNT.
     IF SALES-AMOUNT = ZERO
         MOVE "Y" TO END-OF-SESSION-SWITCH
     ELSE
         COMPUTE SALES-TAX ROUNDED =
             SALES-AMOUNT * .0785
         DISPLAY "SALES TAX = " SALES-TAX.
```

Identification Division

Environment Division

Data Division

Procedure Division

Description

- The Identification Division identifies the program by giving the program name.
- The Environment Division includes the Input-Output Section, which identifies the input and output files used by the program.
- The Data Division includes the File Section and the Working-Storage Section. The File Section describes the files identified in the Input-Output Section, and the Working-Storage Section defines other data items used by the program.
- The Procedure Division contains the program logic. It is typically divided into procedures that contain the statements that do the functions of the program.

Figure 1-4 The COBOL code for the sales tax program

Basic coding rules

Figure 1-5 presents some of the basic coding rules for COBOL programs. To start, you should realize that the first six columns (or positions) in each coding line are left blank when you're using a COBOL compiler. Then, when you compile the program later on, the compiler adds sequence numbers in these positions.

The seventh column in each coding line can be used to identify a line as a *comment*. If there is an asterisk in column 7 (the *indicator column*), this means that the rest of the line is ignored by the compiler. If the rest of the line is left blank, the line can be referred to as a *blank comment*, and blank comments can be used to provide vertical spacing in the COBOL code. In this program, that's the only way that comments are used, but you'll soon see other uses for them.

Columns 8 through 11 in each coding line are referred to as the *A margin*, and columns 12 through 72 as the *B margin*. This is significant because some coding elements have to start in the A margin, and some have to start in the B margin. Although you can start an element anywhere in the A or B margin when that's required, it's customary to start A margin elements in column 8 and B margin elements in column 12.

The last eight positions in each coding line (73-80) aren't used by modern COBOL compilers. These positions originally were used for the name of the program. That was back when COBOL programs were punched into 80-column cards so they could be read by the card reader of a computer system. In the COBOL-2000 standards, the limitation of 80 positions per line will be dropped.

Although the code in this program is in all uppercase (capital) letters, COBOL isn't case sensitive. As a result, you can code a program in capital letters, lowercase letters, or any combination of the two. In practice, though, you usually use one or the other throughout since that's the most efficient way to type code. On mainframes, it's customary to use all capitals; on PCs, it's customary to use all lowercase letters.

When you need to use quotation marks in a coding entry, you can use double quotes (") or single quotes ('). On mainframes, single quotes (or apostrophes) are normally used, but that can be changed by a compiler option. On mid-range computers and PCs, double quotes are normally used. In this book, double quotation marks are used in all the programs.

To separate two coding elements like the words in a Procedure Division statement, you use one or more spaces. This means that you can use spacing to align or indent coding elements. In the Working-Storage Section, for example, you can see how extra spacing is used to align the Pic clauses. In the Procedure Division, you can see how extra spacing is used to indent the Until clause to show that it's part of the Perform statement that starts on the line above.

The A and B margins of a COBOL program

```
      11                                                  77       8
1     678  12                                             23       0
           A   B
       *
        WORKING-STORAGE SECTION.
       *
        77  END-OF-SESSION-SWITCH       PIC X       VALUE "N".
        77  SALES-AMOUNT                PIC 9(5)V99.
        77  SALES-TAX                   PIC Z,ZZZ.99.
       *
        PROCEDURE DIVISION.
       *
        000-CALCULATE-SALES-TAX.
       *
            PERFORM 100-CALCULATE-ONE-SALES-TAX
                UNTIL END-OF-SESSION-SWITCH = "Y".
            DISPLAY "END OF SESSION.".
            STOP RUN.
```

The components of each line of code

Columns	Purpose	Remarks
1-6	Sequence	A sequence number is added to each coding line when the program is compiled. As a result, the programmer doesn't enter anything in these positions.
7	Indicator	If you code an asterisk in this column, the entire line is treated as a *comment*, which means it's ignored by the compiler. You can also use a slash (/) in this column to force the program listing to start on a new page when it is compiled or a hyphen (-) to continue the code from the previous line (see chapter 6).
8-11	A margin	Some coding elements (like division names, section names, procedure names, 77 level numbers, and 01 level numbers) have to start in this margin.
12-72	B margin	Coding lines that don't start in the A margin have to start in this margin.
73-80	Identification	These positions originally were used to identify a program, but they're not used today.

Coding rules

- You can use capital or lowercase letters when you code a COBOL program since the compilers treat them the same.
- When quotation marks are used, double quotes (") are required by most compilers. On IBM mainframes, though, single quotes (') are commonly used, although this can be changed to double quotes by a compiler option.
- One space is treated the same as any number of spaces in sequence. As a result, you can code more than one space whenever you want to indent or align portions of code.

Figure 1-5 Basic coding rules

How to code the Identification Division

Figure 1-6 shows how to code the Identification Division. The only required lines in this division are the division header and the Program-ID paragraph followed by the program name. Often, though, the Program-ID paragraph will be followed by a series of comments like those shown in this figure. These comments give more information about the program.

In standard COBOL, the program name can be up to 30 characters long, so a name like

```
CALCULATE-SALES-TAX
```

is a legal name. Many COBOL compilers, though, require names that are more restrictive than that. On an IBM mainframe, for example, you should keep the program name to eight characters or less using the rules presented in this figure. If you don't, the compiler will convert your name to one that does obey these rules. That's why the program in this example is named CALC1000.

In a mainframe COBOL shop, you usually are given the program name when a program is assigned to you so you don't have to create your own name. In addition, you usually are given specifications for what information you should provide through comments in the Identification Division.

Incidentally, the term *syntax* refers to the structure of a language. So the syntax at the top of this figure gives the structure that the Identification Division requires. In a syntax summary like this, the capital letters represent the COBOL words that are required. The lowercase letters represent entries made by the programmer. In the figures that follow, you'll learn the other conventions that are used in syntax summaries.

How to code the Environment Division

The Environment Division is used to identify any disk files that are used by the program. Since the sales tax program in this chapter doesn't use any, you don't have to write any code for this section. However, you still need to include the Environment Division header and the Input-Output Section paragraph that are shown in figure 1-4. In chapter 3, you'll learn how to develop programs that use disk files.

The syntax of the Identification Division

```
IDENTIFICATION DIVISION.
PROGRAM-ID. program-name.
```

An Identification Division with the minimum code

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    CALC1000.
```

An Identification Division that contains comments

```
 IDENTIFICATION DIVISION.
*
 PROGRAM-ID.       CALC1000.
*PROGRAMMER.       MIKE MURACH.
*COMPLETION-DATE.  MARCH 28, 2000.
*REMARKS.          THIS IS A SIMPLE INTERACTIVE PROGRAM THAT'S
*                  DESIGNED TO ILLUSTRATE THE USE OF COBOL.
*                  IT CALCULATES THE SALES TAX ON AN AMOUNT
*                  THAT HAS BEEN ENTERED BY THE USER.
```

The rules for forming a program name in standard COBOL

- Use letters, the digits 0 through 9, and the hyphen.
- Don’t start or end the name with a hyphen.
- Use a maximum of 30 characters.

The rules for forming a program name on a mainframe compiler

- Start the name with a letter.
- Use letters and digits only.
- Use a maximum of 8 characters.

Typical comment entries in the Identification Division

- Who created the program and when it was completed.
- Who maintained the program and when that maintenance was completed.
- The purpose of the program.
- Any notes that will make the program easier to understand.

Figure 1-6 How to code the Identification Division

How to code the Working-Storage Section

Figure 1-7 gives the basic rules for coding the entries in the Working-Storage Section of the Data Division. When there are only a few entries, you can code them using 77 level entries as shown in this figure. To do that, you start by coding the number 77 in the A margin. Then, you code a *data name* starting in the B margin, usually in column 12. This is the name that will be used in the Procedure Division to refer to the *data item* that you're defining.

After the data name, you code a Picture clause that defines the format of the data item. Last, you can code a Value clause that gives a starting value to the item. One of the most important coding rules is that the Value clause should be consistent with the Picture clause. If it isn't, an error may occur when the program is compiled.

If your program requires more than just a few data items, you probably won't use 77 level numbers. Instead, you'll want to group related items to make them easier to find. You'll learn how to do that later in this chapter, after you learn how to create data names and code Picture and Value clauses.

How to create data names

Figure 1-7 gives the rules for forming data names. In brief, use letters, numbers, and hyphens with a maximum of 30 characters in each name. If you follow these rules, your names will be acceptable to the COBOL compiler.

However, it's also important to create data names that are easy to remember and understand. That's why long names like SALES-AMOUNT and SALES-TAX are better than short names like A1 and B2. Since a typical COBOL program contains dozens of data items, this is a critical factor for efficient programming. And the more data names a program uses, the more important this is.

Incidentally, one naming rule that isn't included in the summaries in this chapter is that a name created by a programmer can't be the same as a COBOL *reserved word*. Reserved words are those that are part of the COBOL language, like DATA, DIVISION, SECTION, WORKING-STORAGE, DISPLAY, ACCEPT, and PERFORM. If you follow our naming recommendations, though, you won't accidentally use a reserved word as one of your names.

Although the COBOL standards refer to *data items* and *data names*, you also can refer to a data item as a *variable* and to its name as a *variable name*. In addition, you can refer to a variable as a *field* because your data definitions are actually defining fields within internal storage. These terms are commonly used in other programming languages. In this book, we'll use all of these terms because a COBOL programmer should be able to use them all.

The Working-Storage Section of the interactive program

```
 WORKING-STORAGE SECTION.
*
 77  END-OF-SESSION-SWITCH       PIC X       VALUE "N".
 77  SALES-AMOUNT                PIC 9(5)V99.
 77  SALES-TAX                   PIC Z,ZZZ.99.
```

Coding rules

- Code the level number (77) in the A margin, and code the data name, its Picture (PIC) clause, and its Value clause (if any) in the B margin.
- Code the Value clause so it is consistent with the Picture clause for each data item (see figure 1-9 for more information).
- Code a period at the end of each data item.
- You can use other level numbers when you want to group the items in working storage (see figure 1-10 for more information).

The rules for forming a data name

- Use letters, the digits 0 through 9, and hyphens only.
- Don’t start or end the name with a hyphen.
- Use a maximum of 30 characters.
- Use at least one letter in the name.

Figure 1-7 How to code the Working-Storage Section

How to code Picture clauses

The Picture clause is used to define the data items that a program requires. When you code this clause, you normally code the abbreviation Pic followed by the characters that define the data item. In figure 1-8, you can learn how to code the pictures for the three types of data items you'll use the most.

When you code the characters in a picture, a number in parentheses means that the character is repeated that number of times. As a result, X(3) is equivalent to XXX, S9(3)V9(2) is equivalent to S999V99, and Z(4).9(2) is equivalent to ZZZZ.99.

When you define an *alphanumeric item*, you use X's to indicate the number of characters that can be stored in the item. Each of these characters means that one letter, digit, or special character can be stored in the item. Thus, a data item that's defined as X(3) can store values like 123, ABC, X2$, or just Y. If an alphanumeric item contains fewer characters than the picture provides for, unused positions to the right are filled with spaces. Thus, the data for the third example of an alphanumeric item is followed by five spaces.

When you define a *numeric item*, you use 9's to indicate the number of digits that can be stored in the item. You can also code a leading S to indicate that the item can have a plus or minus sign, and you can code one V to indicate where the decimal point is assumed to be. Thus, a data item that's defined as S999V99 can store values like +.05 and –999.95. If a numeric item contains fewer digits that the picture provides for, the unused positions to the left are set to zeros. As you will see later, it is the numeric items that you use in arithmetic operations.

When you define a *numeric edited item*, you use characters that make the data in the item easier to read. For instance, you can code a Z when you want a zero to the left of a number to be changed to a space, and you can use a comma or decimal point when you want to insert a comma or decimal point into a number. Then, when the program moves a numeric item to a numeric edited item, the data is converted to the more readable form. You'll learn more about this in figure 1-14.

When you code the pictures for the data items, you are actually defining the internal storage fields that are going to be used when the program is run. In general, each character or digit that you define in a Pic clause requires one *byte* of internal storage so an alphanumeric field that's defined as X(20) requires 20 bytes of storage, and a numeric edited field that's defined as ZZ,ZZZ.99- requires 10 bytes of storage.

For a numeric item, the V just marks the position of the decimal point so it doesn't require a storage byte. Similarly, the S indicates that the data item can include a sign, which is usually carried in the rightmost byte of the field along with the rightmost digit, so it doesn't require a separate storage byte. As a result, a numeric item that's defined as S999V99 usually requires just 5 bytes of internal storage. For more information about how data is defined and stored, please refer to chapter 6.

Some of the characters that can be used in Picture clauses

Item type	Characters	Meaning	Examples
Alphanumeric	X	Any character	PIC X PIC XXX PIC X(3)
Numeric	9	Digit	PIC 99
	S	Sign	PIC S999
	V	Assumed decimal point	PIC S9(5)V99
Numeric edited	9	Digit	PIC 99
	Z	Zero suppressed digit	PIC ZZ9
	,	Inserted comma	PIC ZZZ,ZZZ
	.	Inserted decimal point	PIC ZZ,ZZZ.99
	-	Minus sign if negative	PIC ZZZ,ZZZ-

Examples of Picture clauses

Alphanumeric items

Value represented	Picture	Data in storage
Y	X	Y
OFF	XXX	OFF
714 Main Street	X(20)	714 Main Street

Numeric items

Value represented	Picture	Data in storage	Sign
-26	999V99	02600	(no sign)
+12.50	999V99	01250	(no sign)
+.23	S9(5)V99	0000023	+
-10682.35	S9(5)V99	1068235	-

Numeric edited items

Value represented	Picture	Data in storage
0	Z(4)	(spaces)
0	ZZZ9	0
87	ZZZ9	87
+2,319	ZZ,ZZZ-	2,319
-338	ZZ,ZZZ-	338-
+5,933	Z,ZZZ.99-	5,933.00
-.05	Z,ZZZ.99-	.05-

Description

- The Picture clause (PIC) defines the format of the data that can be stored in the field.
- When coding a Picture clause, a number in parentheses means that the preceding character is repeated that number of times.
- When data is stored in an alphanumeric item, unused positions to the right are set to spaces. When data is stored in a numeric item, unused positions to the left are set to zeros.

Figure 1-8 How to code Picture clauses

How to code Value clauses

Figure 1-9 shows how to use a Value clause to assign a starting value to an alphanumeric or numeric data item. One way to do that is to code a *literal* in the Value clause. If the data item is defined as alphanumeric, you can code an *alphanumeric literal* in the Value clause by enclosing the characters in quotation marks. If the data item is defined as numeric, you can code a *numeric literal* in the Value clause by using the digits, a leading plus or minus sign, and a decimal point.

Another way to assign a value to a data item is to code a *figurative constant* in the Value clause. Although COBOL provides for a number of these, the two you'll use the most are ZERO (or ZEROS or ZEROES), which can be used to assign a value of zero to a numeric item, and SPACE (or SPACES), which can be used to assign all spaces to an alphanumeric item.

When you code a Value clause, it should be consistent with the data type that's defined by the Picture clause. For instance, a Value clause for a numeric item must contain a numeric value. Although it's okay to define a value that is shorter than the maximum entry for a data item, you can't define a value that is too large to be stored in the data item.

The use of literals in Value clauses

Type	Characters	Meaning	Examples
Non-numeric literal	Any	Any character	`VALUE "Y"` `VALUE "END OF SESSION"`

Numeric literal	0-9
	+ or -
	.
		Digit
		Leading sign
		Decimal point
			`VALUE 100`
			`VALUE -100`
			`VALUE +123.55`

The use of figurative constants in Value clauses

Type	Constant	Meaning	Examples

Numeric	`ZERO`
	`ZEROS`
	`ZEROES`
		Zero value	`VALUE ZERO`
			`VALUE ZEROS`
			`VALUE ZEROES`

Non-numeric	`SPACE` `SPACES`	All spaces	`VALUE SPACE` `VALUE SPACES`

Examples of data entries with consistent Picture and Value clauses

Alphanumeric items

```
77  CUSTOMER-ADDRESS     PIC X(20)        VALUE "213 W. Palm Street".
77  END-OF-FILE-SWITCH   PIC X            VALUE "N".
77  SEPARATOR-LINE       PIC X(20)        VALUE "--------------------".
77  BLANK-LINE           PIC X(30)        VALUE SPACE.
```

Numeric items

```
77  INTEREST-RATE        PIC 99V9         VALUE 12.5.
77  UNIT-COST            PIC 99V999       VALUE 6.35.
77  MINIMUM-BALANCE      PIC S9(5)V99     VALUE +1000.
77  GRAND-TOTAL          PIC S9(5)V99     VALUE ZERO.
```

Description

- The Value clause defines the value that is stored in the field when the program starts. As a result, the value should be consistent with the type of item that's defined by the Picture clause.
- In contrast to the rest of the program, the characters between the quotation marks in an alphanumeric literal are case sensitive. So the value of "End of Session" is: End of Session.
- If the Value clause defines a value that is smaller than the field defined by the Picture clause, an alphanumeric field is filled out with spaces on the right; a numeric field is filled out with zeroes on the left.
- If the Value clause defines a value that is larger than can be stored in the field defined by the Picture clause, a compiler error will occur.
- Because a numeric edited item typically receives a value as the result of a Move statement, it usually is not defined with a Value clause. See figure 1-14 for more information on the Move statement.

Figure 1-9 How to code Value clauses

How to code group items

A 77 level number in a data definition means that the item is independent of all other items. As the number of these items increases in a program, it becomes more difficult to find an item. If, for example, a program has 30 independent items, it's hard to find the one you're looking for when you need to check the spelling of its name or the way it's defined.

As some point, then, it makes sense to group related items as shown in figure 1-10. Here, the names used at the 01 level indicate how the subordinate items are related. In this case, the first group contains the fields that are going to receive the data entered by the computer user; the second group contains work fields that are needed by the program.

You also can use group items to show the structure of the data that you're defining. This is illustrated by the field named TODAYS-DATE. Here, the date is made up of three fields that represent the month, day, and year. In chapter 3, you'll see how grouping can be used to show the structure of the fields within a record.

When you group items, the item at the top is called a *group item* and the items that it's made up of are called *elementary items*. In this figure, USER-ENTRIES, WORK-FIELDS, and TODAYS-DATE are group items, and all the others are elementary items.

Because all but the simplest programs require a dozen or more working-storage fields, we recommend that you group the data items in all of your programs. That means that you shouldn't use 77 levels at all. We presented them in the first program only because you're likely to see them in other people's code.

A Working-Storage Section that contains group items

```
 WORKING-STORAGE SECTION.
*
 01  USER-ENTRIES.
*
     05  NUMBER-ENTERED              PIC 9         VALUE 1.
     05  INVESTMENT-AMOUNT           PIC 99999.
     05  NUMBER-OF-YEARS             PIC 99.
     05  YEARLY-INTEREST-RATE        PIC 99V9.
*
 01  WORK-FIELDS.
*
     05  FUTURE-VALUE                PIC 9(7)V99.
     05  YEAR-COUNTER                PIC 99.
     05  EDITED-FUTURE-VALUE         PIC Z,ZZZ,ZZZ.99.
     05  TODAYS-DATE.
         10  TODAYS-MONTH            PIC 99.
         10  TODAYS-DAY              PIC 99.
         10  TODAYS-YEAR             PIC 9(4).
```

Description

- To code group items, you use the level numbers 01 through 49. Typically you will start with 01, and then use multiples of 5, such as 05 and 10. This allows you some room to add other levels later if you need to.
- Level 01 items must begin in the A margin. Other level numbers can begin in either the A or B margin.
- Whenever one data item has higher level numbers beneath it, it is a *group item* and the items beneath it are *elementary items*.
- In the example above, USER-ENTRIES, WORK-FIELDS, and TODAYS-DATE are group items. All of the others are elementary items.
- You can’t code a Picture clause for a group item, and you have to code a Picture clause for an elementary item.
- A group item is always treated as an alphanumeric item, no matter how the elementary items beneath it are defined.
- To make the structure of the data items easy to read and understand, you should align the levels as shown above. However, this indentation isn’t required.

Figure 1-10 How to code group items

How to code the Procedure Division

Figure 1-11 gives the basic rules for coding the Procedure Division. As you can see, the procedure names start in the A margin, and the statements start in the B margin. To make a statement easier to read, you can code extra spaces to indent portions of the statement. This is illustrated by the Perform and If statements.

In the pages that follow, you'll get detailed information about each of the statements used in the Procedure Division of the interactive program. But first, you need to know how to create valid procedure names.

How to create procedure names

Figure 1-11 gives the rules for forming *procedure names*. In brief, use letters, numbers, and hyphens with a maximum of 30 characters in each name, and don't start or end the name with a hyphen. If you follow these rules, your names will be acceptable to the COBOL compiler.

However, it's also important to create procedure names that have meaning rather than names like P5 or MY-PROCEDURE-1. In a program with only two procedures, this doesn't matter much, but the more procedures a program contains, the more it matters.

That's why we recommend that each procedure name consist of a sequence number, a verb, an adjective, and an object. This is illustrated by the procedure names in this figure and in all the programs throughout this book.

In this book, the Procedure Divisions of all the illustrative programs are divided into paragraphs. As a result, you also can refer to a procedure name as a *paragraph name*. You should think of the code within each paragraph, though, as a *procedure* that represents one functional module. You'll understand this better when you learn how the Perform and Perform Until statements work.

The Procedure Division of the interactive program

```
 PROCEDURE DIVISION.
*
 000-CALCULATE-SALES-TAX.
*
     PERFORM 100-CALCULATE-ONE-SALES-TAX
         UNTIL END-OF-SESSION-SWITCH = "Y".
     DISPLAY "END OF SESSION.".
     STOP RUN.
*
 100-CALCULATE-ONE-SALES-TAX.
*
     DISPLAY "-------------------------------------------".
     DISPLAY "TO END PROGRAM, ENTER 0.".
     DISPLAY "TO CALCULATE SALES TAX, ENTER THE SALES AMOUNT.".
     ACCEPT SALES-AMOUNT.
     IF SALES-AMOUNT = ZERO
         MOVE "Y" TO END-OF-SESSION-SWITCH
     ELSE
         COMPUTE SALES-TAX ROUNDED =
             SALES-AMOUNT * .0785
         DISPLAY "SALES TAX = " SALES-TAX.
```

Margin use

- Code all procedure names starting in the A margin, and code all statements in the B margin.

Period use

- Although it isn't required with a COBOL-85 compiler, we recommend that you end each statement with a period. In chapter 5, you'll learn about the alternatives to this practice.

The rules for forming a procedure name

- Use letters, the digits 0 through 9, and hyphens only.
- Don't start or end the name with a hyphen.
- Use a maximum of 30 characters.

Description

- The Procedure Division of a program should be divided into paragraphs like the one above, where each paragraph represents one *procedure* of the program. The name of each paragraph can then be referred to as either a *paragraph name* or a *procedure name*.
- The name of the first procedure should represent the function of the entire program. The names of the procedures it calls should represent the functions performed by those procedures.

Figure 1-11 How to code the Procedure Division

How to code Accept statements

Figure 1-12 shows how the Accept statement works. It gets a value that the user has entered on the keyboard and stores the value in the data item named in the statement. If you look at the syntax for this statement, you can see that you just code the word ACCEPT followed by the data name for the item. Because the Accept statement doesn't display anything on the screen, it is usually issued after a Display statement to tell the user what to enter.

As the table in this figure shows, the data that's stored depends on the Picture of the data item that the entry is stored in. If, for example, the user enters a negative number but the Picture doesn't include an S, the sign isn't stored. If the user enters a numeric value that is larger than the data item can hold, the value is truncated on the left. And if the user enters an alphanumeric value that is larger than the data item can hold, the value is truncated on the right.

Although Accept statements work well when you're using a PC, they present a few problems when you're using a mainframe. As a result, you rarely use Accept statements to get user entries on a mainframe. In chapter 18, though, you can learn how to run programs that use Accept statements on a mainframe. And in chapter 19, you can learn the right way to code interactive programs on a mainframe.

The syntax of the Accept statement

```
ACCEPT data-name
```

An example of an Accept statement

```
ACCEPT SALES-AMOUNT.
```

The operation of some typical Accept statements

Picture	User entry	Value stored	Notes
S999	10	10	
S999	787	787	
S999	-10	-10	
S999	5231	231	Truncated on the left
999	-100	100	Sign dropped
9(3)V99	458.12	458.12	
9(3)V99	45812	812.00	Truncated on the left
9(3)V99	4735.26	735.26	Truncated on the left
X	Y	Y	
X	Yes	Y	Truncated on the right

Description

- When the Accept statement is run, the computer waits for the user to type an entry on the keyboard and press the Enter key.
- When the user presses the Enter key, the entry is stored in the variable identified on the Accept statement, and the cursor moves to the next line on the screen.
- The user entry should be consistent with the Picture of the variable. If it isn't, it will be truncated or adjusted as shown in the table.

Mainframe note

- On an IBM mainframe, the Accept statement gets its data from the SYSIN device. As a result, this device must be set to the terminal keyboard if you want this program to work interactively. Also, you have to enter the data more precisely when you use a mainframe than you do when you use a PC. In chapter 18, you'll learn how to run this type of program on a mainframe.

Figure 1-12 How to code Accept statements

How to code Display statements

Figure 1-13 shows how the Display statement works. If you look at the syntax for this statement, you can see that it consists of the word DISPLAY followed by a series of data names or literals. Here, the braces {} mean that you have a choice between the items separated by a vertical bar (|), and the ellipsis (…) means that you can code as many data names or literals as you need.

When the Display statement is executed, the values represented by the data names and literals are displayed on the screen in the sequence that they're coded. In the examples, you can see that the first four statements display one literal value each. The fifth statement displays the value in a data item. The sixth and seventh statements display an alphanumeric literal followed by the value in a data item followed by another alphanumeric literal.

The syntax of the Display statement

```
DISPLAY {data-name-1 | literal-1} ...
```

Examples of Display statements

```
DISPLAY " ".
DISPLAY 15000.
DISPLAY "-----------------------------------".
DISPLAY "End of session.".
DISPLAY SALES-AMOUNT.
DISPLAY "THE SALES AMOUNT IS " SALES-AMOUNT ".".
DISPLAY "THE SALES TAX IS " SALES-TAX ".".
```

The data displayed by the statements above

```
(one space or a blank line)
15000
-----------------------------------
End of session.
100.00
THE SALES AMOUNT IS 100.00.
THE SALES TAX IS  7.85.
```

Note: The last three display lines assume that SALES-AMOUNT has a Pic clause of ZZZ.99 and a value of 100.00 and that SALES-TAX has a Pic clause of ZZ.99 and a value of 7.85.

Description

- The Display statement displays one or more literal or variable values on the screen of a monitor or terminal. After it displays these values, the cursor moves to the next line on the screen.
- After the word DISPLAY, you can code one or more literals or variable names. For instance, the first statement above displays an alphanumeric literal value of one space (it looks like a blank line on the screen); the second statement displays a numeric literal value of 15000.
- If you code more than one literal or variable name after the word DISPLAY, you must separate them by one or more spaces. For instance, the last two statements above display an alphanumeric literal, a variable value, and another alphanumeric literal (the period).

Mainframe note

- On an IBM mainframe, the Display statement sends its data to the SYSOUT device. As a result, this device must be set to the terminal screen if you want this program to work interactively. You'll learn more about this in chapter 18.

Figure 1-13 How to code Display statements

How to code Move statements

Figure 1-14 shows you how to code Move statements. If you look at the syntax for this statement, you can see that you code a data name or literal after the word MOVE and a second data name after the word TO. Then, when the statement is executed, the value in the first data name or literal (the *sending field*) is stored in the data item represented by the second data name (the *receiving field*). Note that the original data remains in the sending field after the move operation.

If you look at the table of legal and illegal moves in this figure, you can see that the sending field and the receiving field need to be compatible. So you normally send alphanumeric data to an alphanumeric receiving field and numeric data to either a numeric or a numeric edited receiving field. Other types of moves may work if the items consist of unsigned integers, but you shouldn't need to use them.

When you move a numeric field to a numeric edited field, the data is *edited* before it is stored in the receiving field. This means that it is converted to the more readable form represented by the picture of the receiving field. If you look at the examples of this type of move, you can see that lead zeros are suppressed, commas and decimal points are inserted into numbers, and a minus sign is printed after a number to show that its value is negative. Although this is enough information to get you by for the next few chapters, you'll learn more about moving data to numeric edited fields in chapter 6.

The syntax of the Move statement

```
MOVE {data-name-1 | literal} TO data-name-2
```

Examples of Move statements

```
MOVE "Y" TO END-OF-SESSION-SWITCH.
MOVE 1 TO PAGE-NUMBER.
MOVE NUMBER-ENTERED TO EDITED-NUMBER-ENTERED.
```

Legal and illegal moves

Type of move	Legal?
Alphanumeric to alphanumeric	Yes
Numeric to numeric	Yes
Numeric to numeric edited	Yes
Alphanumeric to numeric	Only if the sending field is an unsigned integer
Alphanumeric to numeric edited	Only if the sending field is an unsigned integer
Numeric to alphanumeric	Only if the sending field is an unsigned integer

Examples of numeric to numeric edited moves

Picture of sending field	Data in sending field	Sign of sending field	Picture of receiving field	Edited result
S9(6)	000123	+	ZZZ,ZZ9-	123
S9(6)	012345	-	ZZZ,ZZ9-	12,345-
S9(6)	000000	(no sign)	ZZZ,ZZ9-	0
S9(4)V99	012345	+	ZZZZ.99	123.45
S9(4)V99	000000	(no sign)	ZZZZ.99	.00

Examples of truncation

Picture of sending field	Data in sending field	Picture of receiving field	Result
X(3)	Yes	X	Y
S9(6)	012345	S9(3)	345

Description

- The Move statement moves data from a literal or a sending field to a receiving field. However, the original data is retained in the sending field.
- If the sending field is a numeric item and the receiving field is numeric edited, the Move statement converts the data from one form to the other.
- If the receiving field is larger than the sending field, the receiving field is filled out with trailing blanks in an alphanumeric move or leading zeros in a numeric move.
- If the receiving field is smaller then the sending field, the data that's moved may be truncated. In general, you should avoid this type of move because you may not get the result that you expect.

Figure 1-14 How to code Move statements

How to code Compute statements

Figure 1-15 shows you how to use the Compute statement for performing calculations. If you're comfortable with mathematical notation, you shouldn't have much trouble with it. To the right of the equals sign, you code an *arithmetic expression* using the *arithmetic operators*. To the left of the equals sign, you code the name of the numeric or numeric edited variable that the result should be stored in.

In the examples, you can see that a variable in the expression also can be the receiving field to the left of the equals sign. In this case, that variable must be defined as numeric. After the expression is calculated using the starting value of the variable, the result is stored in the variable. In the first example in this figure, if YEAR-COUNTER has a value of 5 when the statement starts, it has a value of 6 when the statement is finished.

If the receiving variable isn't used in the expression, though, it can be defined as a numeric edited item. Then, the result of the calculation is edited when it's stored in the receiving item.

The brackets [ ] in the syntax for this statement indicate that the Rounded and On Size Error clauses are optional. If the result of a computation has more decimal places than the definition of the receiving field provides for, you can code the Rounded clause to round the result. If you don't code this clause, the extra decimal places will be truncated.

Similarly, if the result of a computation may be larger than the definition of the receiving field provides for, you can code the On Size Error clause. Then, when the result is too large, this clause is activated. Within this clause, you can code one or more statements that deal with this error. For instance, the second last example in this figure displays an error message when a size error occurs.

What happens if you don't code the On Size Error clause and the result is larger than the receiving field? That depends on the platform and compiler that you're using. On a PC, the result is usually truncated, which means the result is incorrect. On a mainframe, this leads to an error that causes the program to be cancelled, which certainly isn't what you want.

For efficiency, you shouldn't code the Rounded clause when there isn't any need for it. You should also avoid the use of the On Size Error clause. Occasionally, you need it, but most of the time you can define the receiving field so it's large enough for any possible result.

The syntax of the Compute statement

```
COMPUTE data-name [ROUNDED] = arithmetic-expression
     [ON SIZE ERROR statement-group]
```

The arithmetic operators

- \+ Addition
- \- Subtraction
- * Multiplication
- / Division
- ** Exponentiation

Examples of Compute statements

```
COMPUTE YEAR-COUNTER = YEAR-COUNTER + 1.

COMPUTE SALES-TAX ROUNDED =
    SALES-AMOUNT * .0785.

COMPUTE SALES-CHANGE = THIS-YEAR-SALES – LAST-YEAR-SALES.

COMPUTE CHANGE-PERCENT ROUNDED =
        SALES-CHANGE / LAST-YEAR-SALES * 100
    ON SIZE ERROR
        DISPLAY "SIZE ERROR ON CHANGE PERCENT".

COMPUTE NUMBER-SQUARED = NUMBER-ENTERED ** 2.
```

Description

- The Compute statement calculates the *arithmetic expression* to the right of the equals sign and stores the result in the variable to the left of the equals sign.
- Within the expression, you use the *arithmetic operators* for addition, subtraction, multiplication (*), division, and exponentiation (**). Exponentiation means “raise to the power of” so A ** 2 is the same as A^2.
- All variables in the arithmetic expression must be numeric items.
- The variable that will contain the result of the arithmetic expression can be a numeric edited item if that variable isn’t used in the arithmetic expression. Otherwise, it must be numeric.
- You can code the Rounded clause whenever the result of a calculation can have more decimal places than are specified in the picture of the result field. If you don’t use the Rounded clause, the extra decimal places are truncated.
- You can code the On Size Error clause when there’s a chance that the result may be larger than the receiving field. If it is, the statements in this clause are executed.

Figure 1-15 How to code Compute statements

How to code arithmetic expressions

Figure 1-16 gives you more information about the coding and evaluation of arithmetic expressions. To start, it gives the *order of precedence* of the arithmetic operations. Unless parentheses are used, this means that the exponentiation operations are done first, from left to right in the expression. Then, the multiplication and division operations are done from left to right. Last, the addition and subtraction operations are done from left to right.

If this isn't the sequence in which you want the operations done, you can use parentheses to change that sequence. Then, the operations in the innermost sets of parentheses are done first, followed by the operations in the next sets of parentheses, and so on until the operations in all the sets of parentheses have been done. Within a set of parentheses, though, the operations are still done in the order of precedence.

If that sounds complicated, the examples in this figure should help you understand how this works. There you can see how the use of parentheses can affect the result. In general, you should use parentheses to clarify the sequence of operations whenever there's any doubt about how the expression will be evaluated.

The order of precedence for arithmetic operations

1. Exponentiation (**)
2. Multiplication and division (* and /)
3. Addition and subtraction (+ and -)

The use of parentheses

- When you use parentheses within an arithmetic expression, the operations in the inner sets of parentheses are done first, followed by the operations in the outer sets of parentheses.

Examples of arithmetic expressions

Expression	A	B	C	D	Result
A + B + C	2	3	2		7
A + B + C	2	-3	2		1
A – B – C	2	3	2		-3
A + B * C	2	3	2		8
(A + B) * C	2	3	2		10
A + B * C ** D	2	3	2	2	14
(A + B) * C ** D	2	3	2	2	20
(A / (B * C)) ** D	12	3	2	2	4
A – B / B	125	100			124
(A – B) / B	125	100			.25
(A – B) / B * 100	125	100			25

Examples of Compute statements

Statement	A (After) S9(3)V9	A (Before) S9(3)V9	B S9(3)	C S9(3)
COMPUTE A = A + B	5.0	2.0	3	
COMPUTE A = A + 1	3.0	2.0	1	
COMPUTE A ROUNDED = B / C	.3	?	1	3
COMPUTE A = B / C * 100	66.6	?	2	3
COMPUTE A ROUNDED = B / C * 100	66.7	?	2	3
COMPUTE A = 200 / B – C	37.0	?	5	3
COMPUTE A = 200 / (B – C)	100.0	?	5	3
COMPUTE A = 10 ** (B – C)	Size Error	?	5	1
COMPUTE A = A + (A * .1)	110.0	100.0		
COMPUTE A = A * 1.1	110.0	100.0		

Description

- Unless parentheses are used, the operations in an expression take place from left to right in the *order of precedence*.
- To clarify or override the sequence of operations, you can use parentheses.

Figure 1-16 How to code arithmetic expressions

How to code Add statements

Figure 1-17 presents the basic syntax for the two formats of the Add statement. You use this statement for simple additions that can be coded as easily with this statement as they can be with the Compute statement.

When you use the first format of this statement, the result is stored in the data item named in the To clause. If, for example, YEAR-COUNTER has a value of 7 when this statement is executed

```
ADD 1 TO YEAR-COUNTER
```

it has a value of 8 after the statement is executed. In this case, the receiving field has to be a numeric item.

When you use the second format, the result is stored in the data item named in the Giving clause. With this format, you can use either a numeric or a numeric edited item as the receiving field. If you use a numeric edited item, the result is edited when it is sent to the receiving field.

The optional clauses of the Add statement are like those of the Compute statement. If the result may have more decimal places than the receiving field provides for, you can code a Rounded clause. If the result may be larger than the receiving field provides for, you can code an On Size Error clause. In most cases, though, you won't need either of these clauses with Add statements.

In addition to the Add statement, COBOL provides Subtract, Multiply, and Divide statements. These statements, which are presented in chapter 7, have formats that are similar to those for the Add statement. In general, though, the Compute statement is easier to use and its code is easier to read, so you may never need those statements.

The syntax of the Add statement

Format 1

```
ADD {data-name-1 | literal} TO data-name-2 [ROUNDED]
    [ON SIZE ERROR statement-group]
```

Format 2

```
ADD {data-name-1 | literal-1} {data-name-2 | literal-2} ...
    GIVING data-name-3 [ROUNDED]
    [ON SIZE ERROR statement-group]
```

Examples of Add statements

Format 1

```
ADD 1 TO YEAR-COUNTER.
ADD CUSTOMER-SALES TO GRAND-TOTAL-SALES.
```

Format 2

```
ADD OLD-BALANCE NEW-CHARGES
    GIVING NEW-BALANCE.
ADD JAN-SALES FEB-SALES MAR-SALES
    GIVING FIRST-QUARTER-SALES.
```

Description

- When you use format 1, the value in data-name-1 or a literal value is added to the value in data-name-2, and the result is stored in data-name-2. As a result, data-name-2 must be defined as a numeric item.
- When you use format 2, two or more values are added together and the result is stored in the data item that's named in the Giving clause. As a result, that item can be defined as either a numeric or numeric edited item.
- You can code the Rounded clause whenever the result can have more decimal places than is specified in the picture of the result field.
- You can code the On Size Error clause when there's a chance that the result may be larger than the receiving field. If it is, the statements in this clause are executed.

Figure 1-17 How to code Add statements

How to code If statements

Figure 1-18 shows you how to code an If statement. After the word *If*, you code a condition followed by one or more statements that are executed if the condition is true. After the word Else, you code one or more statements that are performed if the condition isn't true. You also can code an End-If *delimiter* to mark the end of the statement.

To code a simple condition within an If statement, you use a *relational operator* to set up a comparison between two data names or between a data name and a literal. If you want the negative of a condition, you code the word NOT before the operator. For instance, NOT > (not greater than) is the same as <= (less than or equal to).

As the syntax for the If statement shows, the Else clause and the End-If delimiter are optional. This is illustrated by the first two examples in this figure, which don't include these items. Here, the first statement executes a Move statement when the SALES-AMOUNT is zero. The second statement executes a Compute and a Display statement when the SALES-AMOUNT isn't equal to zero.

The next example is an If statement that includes both an Else clause and the End-If delimiter. This statement gets the same result as the two statements without the Else clauses.

The last example shows that you can code If statements within If statements. The result can be called *nested If statements*, and this nesting can continue many levels deep. Although End-If delimiters are used in this example, this nest of If statements would work the same without them. Sometimes, though, End-If delimiters are required for the logic to work correctly.

Note in all of the examples that periods aren't used to mark the ends of the statements within the If statement. Instead, one period is coded at the end of an entire nest of If statements. If you code a period within an If statement, the compiler assumes that the statement is supposed to end, which is a common coding error. That's why you use an End-If delimiter whenever you need to mark the end of an If statement within an If statement.

The syntax of the If statement

```
IF condition
    statement-group-1
[ELSE
    statement-group-2]
[END-IF]
```

The syntax of a simple condition

```
{data-name-1 | literal} relational-operator {data-name-2 | literal}
```

The relational operators

Operator	Meaning	Typical conditions
>	Greater than	NUMBER-ENTERED > ZERO
<	Less than	999 < NUMBER-ENTERED
=	Equal to	END-OF-SESSION-SWITCH = "Y"
>=	Greater than or equal to	LINE-COUNT >= LINES-ON-PAGE
<=	Less than or equal to	SALES-THIS-YEAR <= SALES-LAST-YEAR
NOT	The negative of the operator that follows	NUMBER-ENTERED NOT = 0

Examples of If statements

If statements without Else and End-If clauses

```
IF SALES-AMOUNT = ZERO
    MOVE "Y" TO END-OF-SESSION-SWITCH.
IF SALES-AMOUNT NOT = ZERO
    COMPUTE SALES-TAX ROUNDED = SALES-AMOUNT * .0785
    DISPLAY "SALES TAX = " SALES-TAX.
```

An If statement with Else and End-If clauses

```
IF SALES-AMOUNT = ZERO
    MOVE "Y" TO END-OF-SESSION-SWITCH
ELSE
    COMPUTE SALES-TAX ROUNDED = SALES-AMOUNT * .0785
    DISPLAY "SALES TAX = " SALES-TAX
END-IF.
```

Nested If statements

```
IF SALES-AMOUNT >= 10000
    IF SALES-AMOUNT < 50000
        COMPUTE SALES-COMMISSION = SALES * COMMISSION-RATE-1
    ELSE
        COMPUTE SALES-COMMISSION = SALES * COMMISSION-RATE-2
    END-IF
END-IF.
```

Description

- The If statement executes one group of statements if the condition it contains is true, another group of statements if the condition is false and an Else clause is coded.
- When coding an If statement within an If statement, you are coding *nested If statements*.
- The End-If *delimiter* can be used to mark the end of an If statement. This can be useful when you are nesting If statements.

Figure 1-18 How to code If statements

How to code Perform statements

The top portion of figure 1-19 shows how to code the Perform statement to execute a procedure. When the Perform statement is executed, the program skips to the first statement in the procedure named in the statement. Then, the computer executes the statements in this procedure. When the last statement in the procedure is executed, control returns to the first statement after the Perform statement.

This is an important COBOL statement because it lets you divide a large program into a number of manageable procedures. You'll see this statement illustrated in the interactive program presented at the end of this chapter. And you'll appreciate the value of this statement when you study the programs in later chapters.

How to code Perform Until statements

The lower portion of figure 1-19 shows how to code the Perform Until statement. With this statement, a procedure is performed until a condition becomes true. To code the condition in the Until clause of this statement, you use the same relational operators and syntax that you use for If statements.

If you look back to the Procedure Division code in figure 1-11, you should now understand what the Perform Until statement does. It performs the procedure named 100-CALCULATE-ONE-SALES-TAX until the value of the data item named END-OF-SESSION-SWITCH becomes equal to Y. This happens after the Accept statement gets a value of zero from the keyboard and the If statement that follows moves a Y to the switch field. When the condition becomes true, the computer continues with the next statement after the Perform Until statement.

How to code the Stop Run statement

The Stop Run statement stops the execution of a program. Each program should include just one Stop Run statement, and it is usually coded as the last statement in the first procedure of a program.

The syntax of the Perform statement

```
PERFORM procedure-name
```

An example of a Perform statement

```
PERFORM 100-GET-USER-ENTRIES.
```

The operation of the Perform statement

- The Perform statement skips to the procedure that's named and executes the statements in that procedure. Then, it returns to the statement after the Perform, and the program continues.

The syntax of the Perform Until statement

```
PERFORM procedure-name
    UNTIL condition
```

An example of a Perform Until statement

```
PERFORM 100-CALCULATE-ONE-SALES-TAX
    UNTIL END-OF-SESSION-SWITCH = "Y".
```

The operation of the Perform Until statement

- The Perform Until statement tests a condition to see whether it is true. If the condition isn't true, the statement performs the procedure that it names. This continues until the condition becomes true. Then, the program continues with the statement after the Perform Until statement.
- The execution of a Perform Until statement is often referred to as a *processing loop*, or simply a *loop*.
- If the condition never becomes true, the program will continue until the operator interrupts it. This is a programming error.
- The condition in a Perform Until statement is formed the same way it is formed in an If statement.

Figure 1-19 How to code Perform and Perform Until statements

Another interactive COBOL program

This chapter ends by presenting another interactive program. That will give you a chance to see how the COBOL you've learned in this chapter can be applied to another application.

An interactive session

Figure 1-20 presents an interactive session that is controlled by another COBOL program. Here, the program prompts the user to enter three values: investment amount, number of years, and interest rate. Then, the program calculates the future value of the investment and displays the result. The program repeats this until the user indicates that the program should end.

In this example, the future value is the value of an investment of $10,000 that is held for 10 years and gets 10.5 percent yearly interest. In other words, the program calculates how much the $10,000 investment will be worth if it draws 10.5% interest for 10 years. In this case, the result is $27,140.81.

To get the result, the program calculates the interest on the investment one year at a time using this formula:

```
Interest = Investment * Interest / 100
```

Here, the interest rate is divided by 100 so an entry of 10.5 gets treated as .105. As a result, the interest for the first year is $10,000 times .105, or $1050. The program then adds the interest for the year to the principal so the investment value at the end of one year is $11,050.

The program continues in this way for nine more years. For each year, the interest is calculated the same way, but the investment amount keeps increasing. That's the way compound interest works. By the end of the ten years, the investment amount has grown from $10,000 to $27,140.81.

Another interactive session on a monitor or terminal

```
---------------------------------------
To end the program, enter 0.
To perform another calculation, enter 1.
1
---------------------------------------
Enter investment amount (xxxxx).
10000
Enter number of years (xx).
10
Enter yearly interest rate (xx.x).
10.5
Future value =    27,140.81
---------------------------------------
To end the program, enter 0.
To perform another calculation, enter 1.
0
---------------------------------------
End of session.
```

Description

- The shaded numbers above are the entries made by the computer user. All of the other characters are displayed by the COBOL program.
- This program calculates the future value of an investment amount that accumulates interest at a given rate for a given number of years. This can be referred to as compound interest.

Figure 1-20 An interactive session for calculating the future value of an investment

The COBOL code

Figure 1-21 presents the Working-Storage Section and Procedure Division for the interactive session shown in figure 1-20. Unlike the program in figure 1-4, which is coded in capital letters, this one is coded in lowercase letters. Instead of blank comments, this program uses blank lines (with no asterisk in column 7) to provide spacing within the code. This is the style that is commonly used when developing COBOL programs on a PC.

In the Working-Storage Section, you can see how 01 levels are used to group the data items into fields that receive user entries from Accept statements and fields that are required by calculations. Although the 05 levels could be coded as 77 levels, the grouping makes it easier to find the fields you're looking for.

In procedure 100, you can see a Perform statement that performs procedure 110-get-user-values. This puts six related statements in a separate procedure, which makes the program easier to read and understand. The alternative is to code these statements in procedure 100 in place of the Perform statement (without the periods that end these statements).

Also in procedure 100, you can see a Perform Until statement that performs procedure 120 until a field named year-counter is greater than a field named number-of-years. Note that the field named year-counter is set to 1 before this Perform Until statement is executed and that this field is increased by 1 each time procedure 120 is executed. That means procedure 120 will be performed once for each year that the investment amount is held. Note also that the investment amount that has been entered by the user is moved to a field named future-value before this procedure is performed.

Now, if you study procedure 120, you can see that a single Compute statement calculates a new future value each time this procedure is executed. In effect, this statement says: The new future value equals the old future value plus the interest that is calculated for the year. Because the investment amount is moved to the future-value field before this procedure is performed, the interest for the first repetition is based on the original investment amount. But after that, the interest is compounded.

When the Perform Until statement in procedure 100 ends, a Move statement moves the future value into a numeric edited field and a Display statement displays the edited future value. The program then returns to the Perform Until statement in procedure 000, which performs procedure 100 again.

If you're new to programming, this should give you a pretty good idea of what programming is like. Every detail matters. And even in a simple program like this, the logic can get complicated in a hurry.

Do you understand exactly how this program works? If not, you don't need to worry about that right now. In the exercises for the next chapter, you'll step through this program one statement at a time and see how the data changes after each statement is executed. That should clear up any questions you still have about how this program works.

The future value program

```
working-storage section.

01  user-entries.

    05  number-entered              pic 9         value 1.
    05  investment-amount           pic 99999.
    05  number-of-years             pic 99.
    05  yearly-interest-rate        pic 99v9.

01  work-fields.

    05  future-value                pic 9(7)v99.
    05  year-counter                pic 999.
    05  edited-future-value         pic z,zzz,zzz.99.

procedure division.

000-calculate-future-values.

    perform 100-calculate-future-value
        until number-entered = zero.
    display "End of session.".
    stop run.

100-calculate-future-value.

    display "---------------------------------------".
    display "To end the program, enter 0.".
    display "To perform another calculation, enter 1.".
    accept number-entered.
    display "---------------------------------------".
    if number-entered = 1
        perform 110-get-user-values
        move investment-amount to future-value
        move 1 to year-counter
        perform 120-calculate-next-fv
            until year-counter > number-of-years
        move future-value to edited-future-value
        display "Future value = " edited-future-value.

110-get-user-values.

    display "Enter investment amount (xxxxx).".
    accept  investment-amount.
    display "Enter number of years (xx).".
    accept  number-of-years.
    display "Enter yearly interest rate (xx.x).".
    accept  yearly-interest-rate.

120-calculate-next-fv.

    compute future-value rounded =
        future-value +
            (future-value * yearly-interest-rate / 100).
    add 1 to year-counter.
```

Figure 1-21 The COBOL code for the future value program

Perspective

The goal of this chapter has been to get you started with COBOL programming as quickly as possible. If this chapter has succeeded, you now should understand how the code in the two illustrative programs works. You may even be ready to code simple interactive programs of your own.

Before you can actually run a COBOL program, though, you need to enter it into the computer, compile it, and test it. That's why the next chapter shows you how to do these tasks when using Micro Focus Personal COBOL on your own PC. When you complete that chapter, you'll be able to develop programs of your own.

Summary

- COBOL is a standard programming language that can be run on any computer that has a COBOL compiler to create the machine language that will run on the computer. Most COBOL compilers today are based on the 1985 standards.
- An *interactive program* gets its input data from the keyboard of a mainframe *terminal* or PC, and it displays its output data on the screen of a terminal or PC *monitor*. In contrast, a *batch program* processes the data in one or more disk files without interacting with the user.
- A COBOL program consists of four divisions: Identification, Environment, Data, and Procedure.
- When you code a program, some lines must start in the A margin and some in the B margin. If a line has an asterisk in column 7, it is treated as a *comment*, which means that it is ignored by the compiler. To make a program easier to read, you can use *blank comments* or blank lines for vertical spacing and extra spaces for horizontal spacing.
- The Working-Storage Section in the Data Division defines the *data items* that are used by the program. For each item, you code a Picture clause to define the format. You can also code a Value clause to give the item a starting value. A data item can also be referred to as a *field* or *variable.*
- The Procedure Division consists of *procedures*, or *paragraphs*. When a program is run, the computer executes the statements in this Division in sequence starting with the first one in the first procedure, unless a Perform or Perform Until statement changes this sequence by performing one of the other procedures.
- The Accept and Display statements are input and output statements. The Move statement moves data from one field in storage to another, sometimes with *editing*. The Compute and Add statements are arithmetic statements. The If statement is a logical statement. And the Stop Run statement ends the execution of a COBOL program.

Terms

platform
mainframe computer
mid-range computer
terminal
central processing unit
CPU
internal storage
main memory
PC
monitor
processor
RAM
random-access memory
COBOL standards
COBOL-85
COBOL-2000
compile
COBOL compiler
extension
interactive program
batch program
comment
indicator column
blank comment
A margin
B margin
syntax
data name
data item
reserved word
variable
variable name
field
alphanumeric item
numeric item
numeric edited item
byte
literal
alphanumeric literal
numeric literal
figurative constant
group item
elementary item
procedure name
paragraph name
procedure
sending field
receiving field
editing
arithmetic expression
arithmetic operator
order of precedence
delimiter
relational operator
nested If statements
processing loop
loop

Objectives

- Given a COBOL listing for a simple interactive program like the ones in this chapter, explain what each statement in the program does.
- Identify the primary difference between a mainframe computer and a PC.
- Distinguish between these sets of terms: terminal and monitor; CPU and processor; main memory and RAM.
- Distinguish between a set of COBOL standards and a COBOL compiler.
- Identify the location and use of these portions of a line of COBOL code: indicator column, A margin, B margin.
- List three rules for forming any name in a COBOL program.
- Describe the purpose of a Picture clause and a Value clause.
- Distinguish between a group item and an elementary item.
- Describe the operation of any of the statements presented in this chapter:

Accept	Compute	Perform
Display	Add	Perform Until
Move	If	Stop Run

Questions

The questions that follow are designed to test whether you can do the tasks defined by the chapter objectives. Although questions aren't included after the other chapters in this book, objectives are included so you'll know what you should be able to do when you complete a chapter.

1. What is the primary difference between a mainframe computer and a PC?
2. What is the difference between a terminal and a monitor? A CPU and a processor?
3. In a line of COBOL coding, where is the indicator column located and what is it used for?
4. List three rules for forming any COBOL name.
5. What does a Picture clause do? A Value clause?
6. What's the difference between a group and an elementary item?
7. What does each of the following statements do when it is run?

Accept	Move	Perform	Compute
Display	If	Perform Until	Stop Run

Refer to figure 1-4 as you answer the questions that follow. These are designed to make sure that you know how this program works.

8. What are the first six statements that are executed when the program is run? (Just list the verbs of these statements in sequence.)
9. If the user enters a zero when the first Accept statement is run, what statements are run before the program ends? (Just list the verbs.)
10. If the user enters 25 when the first Accept statement is run, what statements are run before the Accept statement is run again? (Just list the verbs.)
11. What happens if the user enters 100000 and the entry is truncated?
12. If the Value clause for the END-OF-SESSION-SWITCH mistakenly sets the value to Y instead of N, what statements will be executed before the program ends? (Just list the verbs.)

Refer to figure 1-21 as you answer the questions that follow.

13. If the investment amount that's entered by the user is $125 and the interest rate is 10 percent, what is the value of the future-value field after procedure 120 has been executed one time?
14. If the picture of the future-value field is changed to S9(7), what does this field contain after procedure 120 has been executed one time?