Beginning COBOL for Programmers

PREV
CHAPTER 4: Procedure Division Basics

NEXT
CHAPTER 6: Control Structures: Iteration

CHAPTER 5

Control Structures: Selection

The last chapter noted that programs that consist only of a sequence of statements are not very useful. To be useful, a program must use selection constructs to execute some statements rather than others and must use iteration constructs to execute certain statements repeatedly.

In this chapter, you examine the selection constructs available to COBOL. In addition to discussing the `IF` and `EVALUATE` statements, this chapter also discusses the condition types recognized by the selection constructs, the creation and use of condition names, the use of the `SET` verb to manipulate condition names, and the proper naming of condition names.

A number of short example programs are introduced in this chapter. Please keep in mind that these are only used to demonstrate particular language elements. They are not intended as realistic examples.

Selection

In most procedural languages, `if` and `case/switch` are the only selection constructs supported. COBOL supports advanced versions of both of these constructs, but it also supports a greater variety of condition types including relation conditions, class conditions, sign conditions, complex conditions, and condition names.

IF Statement

When a program runs, the program statements are executed one after another, in sequence, unless a statement is encountered that alters the order of execution. An `IF` statement is one of the statement types that can alter the order of execution in a program. It allows you to specify that a block of code is to be executed only if the condition attached to the `IF` statement is satisfied. The basic metalanguage for the `IF` statement is given in Figure 5-1.

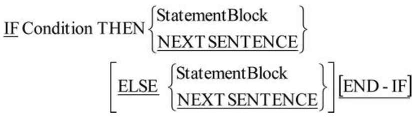


Figure 5-1. Metalanguage for the IF statement/verb

The `StatementBlock` following the `THEN` executes, if the condition is true. The `StatementBlock` following the `ELSE` (if used) executes, if the condition is false. The `StatementBlock`(s) can include any valid COBOL statement including further `IF` constructs. This allows for nested `IF` statements.

One difference from many other programming languages is that when a condition is evaluated, it evaluates to either true or false. It does not evaluate to 1 or 0.

The explicit scope delimiter `END-IF` was introduced in ANS 85 COBOL. In the previous versions of COBOL, scope was delimited by means of the period. Although the scope of the `IF` statement may still be delimited by a period, the END-IF delimiter should always be used because it makes explicit the scope of the IF statement.

There are two problems with using a period as a scope delimiter:

- Periods are hard to see, and this makes it more difficult to understand the code.
- A period delimits all open scopes, and this is a source of many programming errors.

You explore this topic more fully later in the chapter.

Condition Types

The IF statement is not as simple as the metalanguage in Figure 5-1 seems to suggest. The condition that follows the IF is drawn from one of the condition types shown in Table 5-1. If a condition is not a complex condition, then it is regarded as a simple condition. A simple condition may be negated using the NOT keyword. Bracketing a complex condition causes it to be treated as a simple condition.

Table 5-1. *Condition Types*

Condition Type
Relation
Class
Sign
Complex
Condition names

Relation Conditions

Relation conditions are used to test whether a value is less than, equal to, or greater than another value. These conditions will be familiar to programmers of other languages. The use of words as shown in the relation condition metalanguage in Figure 5-2 may come as bit of a shock, but for most conditions the more familiar symbols (= < > >= <=) may be used. There is one exception to this: unlike in many other languages, in COBOL there is no symbol for NOT. You must use the word NOT if you want to express this condition.

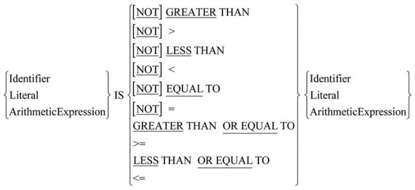


Figure 5-2. *Metalanguage for relation conditions*

Note that the compared values must be type compatible. For instance, it is not valid to compare a string value to a numeric value. Some examples of relation conditions are shown in Example 5-1. Most of these examples are straight forward, but the final example includes an arithmetic expression. In this case, the arithmetic expression is evaluated and then the result is compared with the value in Num1.

Example 5-1. Some Sample Relation Conditions

```
IF Num1 < 10 THEN
     DISPLAY "Num1  < 10"
END-IF

IF Num1 LESS THAN 10
   DISPLAY "Num1 < 10"
END-IF
```

```
IF Num1 GREATER THAN OR EQUAL TO Num2
    MOVE Num1 TO Num2
END-IF

IF Num1 <  (Num2 + ( Num3 / 2))
   MOVE ZEROS TO Num1
END-IF
```

Class Conditions

A class condition does not refer to a class in the OO sense. Instead, it refers to the broad category or class (such as numeric, alphabetic, or alphabetic lower or upper) into which a data item may fall (see the metalanguage for class conditions in Figure 5-3). A class condition is used to discover whether the value of data item is a member of one these classes. For instance, a `NUMERIC` class condition might be used on an alphanumeric (`PIC X`) or a numeric (`PIC 9`) data item to see if it contained numeric data. Or an `ALPHABETIC-UPPER` class condition might be used to discover if a data item contained only capital letters (see Example 5-2).

$$\text{Identifier IS } [\underline{\text{NOT}}] \left\{ \begin{array}{l} \underline{\text{NUMERIC}} \\ \underline{\text{ALPHABETIC}} \\ \underline{\text{ALPHABETIC- LOWER}} \\ \underline{\text{ALPHABETIC- UPPER}} \\ \text{UserDefinedClassName} \end{array} \right\}$$

Figure 5-3. *Metalanguage for class conditions*

Example 5-2. Class Condition That Checks Whether the `StateName` Contains All Capitals

```
IF StateName IS ALPHABETIC-UPPER
   DISPLAY "All the letters in StateName are upper case"
END-IF
```

Notes on Class Conditions

The target of a class test must be a data item with an explicit or implicit usage of `DISPLAY`. In the case of numeric tests, data items with a usage of `PACKED-DECIMAL` may also be tested.

The numeric test may not be used with data items described as alphabetic (`PIC A`) or with group items when any of the elementary items specifies a sign. An alphabetic test may not be used with any data items described as numeric (`PIC 9`).

The `UserDefinedClassName` is a name that you can assign to a set of characters. You must use the `CLASS` clause of the `SPECIAL-NAMES` paragraph, of the `CONFIGURATION SECTION`, in the `ENVIRONMENT DIVISION`, to assign a class name to a set of characters. A data item conforms to the `UserDefinedClassName` if its contents consist entirely of the characters listed in the definition of the `UserDefinedClassName` (see Listing 5-1 in the next section).

User-Defined Class Names

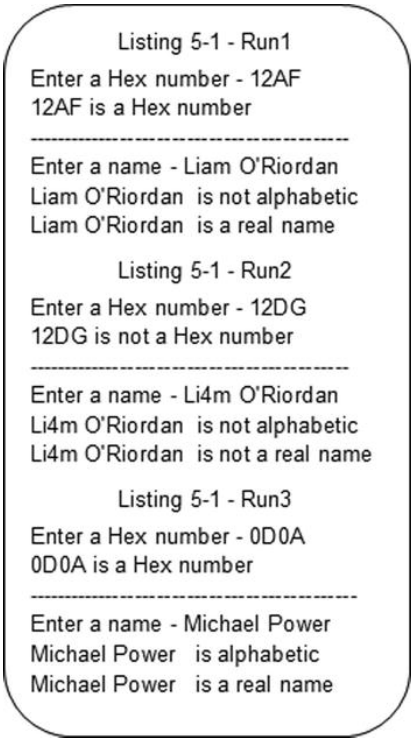
Whereas `ALPHABETIC` and `NUMERIC` are predefined class names that identify a subset of the character set, the `UserDefinedClassName` in the metalanguage (see Figure 5-3) is a name that you can assign to a defined subset of characters. To define the subset, you must create a `CLASS` entry in the `SPECIAL-NAMES` paragraph, of the `CONFIGURATION SECTION`, in the `ENVIRONMENT DIVISION. The CLASS clause` assigns a class name to a defined subset of characters. In a class condition, a data item conforms to the `UserDefinedClassName` if its contents consist entirely of the characters listed in the definition of the `UserDefinedClassName`.

Listing 5-1 is an example program that shows how to define and use a user-defined class name. In this listing, two class names are defined: `HexNumber` and `RealName`. `HexNumber` is used to test that `NumIn` contains only hex digits (0–9 and A–F). `RealName` is used to test that `NameIn` contains only valid characters. `RealName` was created because you can't just use the IS `ALPHABETIC` class condition to test a name—sometimes names, especially Irish names, contain other characters; such as the apostrophe ('). `RealName` allows you to test that the name entered contains only characters from the set you have defined.

Listing 5-1. User-Defined Class Names Used with a Class Condition

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Listing5-1.
AUTHOR. Michael Coughlan.
*> Shows how user defined class names are created and used
```

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    CLASS HexNumber IS "0" THRU "9", "A" THRU "F"
    CLASS RealName  IS "A" THRU "Z", "a" THRU "z", "'", SPACE.

DATA DIVISION.
WORKING-STORAGE SECTION.
01 NumIn       PIC X(4).
01 NameIn      PIC X(15).

PROCEDURE DIVISION.
Begin.
   DISPLAY "Enter a Hex number - " WITH NO ADVANCING
   ACCEPT NumIn.
   IF NumIn IS HexNumber THEN
      DISPLAY NumIn " is a Hex number"
    ELSE
      DISPLAY NumIn " is not a Hex number"
   END-IF

   DISPLAY "----------------------------------"
   DISPLAY "Enter a name - " WITH NO ADVANCING
   ACCEPT NameIn
   IF NameIn IS ALPHABETIC
      DISPLAY NameIn " is alphabetic"
    ELSE
      DISPLAY NameIn " is not alphabetic"
   END-IF

   IF NameIn IS RealName THEN
      DISPLAY NameIn " is a real name"
    ELSE
      DISPLAY NameIn " is not a real name"
   END-IF
   STOP RUN.
```



```
          Listing 5-1 - Run1
Enter a Hex number - 12AF
12AF is a Hex number
----------------------------------
Enter a name - Liam O'Riordan
Liam O'Riordan  is not alphabetic
Liam O'Riordan  is a real name
          Listing 5-1 - Run2
Enter a Hex number - 12DG
12DG is not a Hex number
----------------------------------
Enter a name - Li4m O'Riordan
Li4m O'Riordan  is not alphabetic
Li4m O'Riordan  is not a real name
          Listing 5-1 - Run3
Enter a Hex number - 0D0A
0D0A is a Hex number
----------------------------------
Enter a name - Michael Power
Michael Power   is alphabetic
Michael Power   is a real name
```

How the Program Works

The program accepts a hex number from the user, tests that it contains only valid hex digits, and then displays the appropriate message. The program then accepts a name from the user, uses a class condition to test whether the contents are alphabetic, and displays the appropriate message. The program next tests that `NameIn` contains only the allowed characters and displays the appropriate message. To give you a feel for how the program works, I ran it a number of times and captured the output (see the output attached to Listing 5-1).

Sign Conditions

The sign condition (see the metalanguage in Figure 5-4) is used to discover whether the value of an arithmetic expression is less than, greater than, or equal to zero. Sign conditions are a shorter way of writing certain relation conditions.

$$\text{ArithmeticExpression IS}\,[\underline{\text{NOT}}]\begin{Bmatrix}\underline{\text{POSITIVE}}\\ \underline{\text{NEGATIVE}}\\ \underline{\text{ZERO}}\end{Bmatrix}$$

***Figure 5-4**. Metalanguage for sign conditions*

In Example 5-3, a sign condition is used to discover whether the result of evaluating an arithmetic expression is a negative value. This example also shows the equivalent relation condition.

Example 5-3. Sign Condition Used to Discover Whether a Result Is Negative

```
      IF (Num2 * 10 / 50) - 10 IS NEGATIVE
         DISPLAY "Calculation result is negative"
      END-IF

   *> the equivalent Relation Condition is

      IF (Num2 * 10 / 50) - 10 LESS THAN ZERO
         DISPLAY "Calculation result is negative"
      END-IF
```

Complex Conditions

Unlike sign conditions and class conditions, complex conditions (sometimes called compound conditions) should be familiar to programmers of most languages. Even here, however, COBOL has a tweak—in the form of implied subjects—that you may find unusual. The metalanguage for complex conditions is given in Figure 5-5.

$$\text{Condition}\left\{\begin{Bmatrix}\underline{\text{AND}}\\ \underline{\text{OR}}\end{Bmatrix}\text{Condition}\right\}\ldots$$

***Figure 5-5**. Metalanguage for complex conditions*

Complex conditions are formed by combining two or more simple conditions using the conjunction operator `OR` or `AND`. Any condition (simple, complex, condition name) may be negated by preceding it with the word `NOT`. When `NOT` is applied to a condition, it toggles the true/false evaluation. For instance, if `Num1 < 10` is true then `NOT Num1 < 10` is false.

Like other conditions in COBOL, a complex condition evaluates to either true or false. A complex condition is an expression. Like arithmetic expressions, a complex condition is evaluated from left to right unless the order of evaluation is changed by precedence rules or by bracketing.

The precedence rules that apply to complex conditions are given in Table 5-2. To assist your understanding, the equivalent arithmetic precedence rules have been given alongside the condition rules.

***Table 5-2**. Precedence Rules*

Precedence	Condition Value	Arithmetic Equivalent
1	NOT	**
2	AND	* or /
3	OR	+ or -

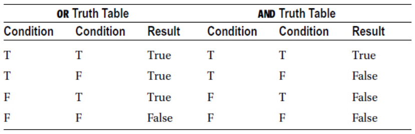
Example 5-4. Complex Condition to Detect Whether the Cursor Is On-screen

```
   *> A complex condition example that detects if the cursor positio
   *> ScrnRow, ScrnCol is on screen (the text screen is 24 lines by
      IF (ScrRow > 0 AND ScrRow < 25) AND (ScrCol > 0 AND ScrCol < 8
         DISPLAY "On Screen"
      END-IF
```

Truth Tables

When a complex condition is being evaluated, it is useful to consider the `OR` and `AND` truth tables, shown in Table 5-3.

Table 5-3. *`OR` and `AND` Truth Tables*



OR Truth Table			**AND** Truth Table		
Condition	Condition	Result	Condition	Condition	Result
T	T	True	T	T	True
T	F	True	T	F	False
F	T	True	F	T	False
F	F	False	F	F	False

The Effect of Bracketing

Bracketing can make the order of evaluation explicit or can change it. Complex conditions are often difficult to understand, so any aid to clarity is welcome. For that reason, when you have to write a complex condition, you should always use bracketing to make explicit what is intended.

Consider the statement

```
IF NOT Num1 < 25 OR Num2 = 80 AND Num3 > 264 THEN
   DISPLAY "Done"
END-IF
```

The rules of precedence govern how this `IF` statement is evaluated. You can leave it like this and hope that future readers will understand it, or you can assist their understanding by using bracketing to make explicit the order of evaluation already governed by those rules.

To apply bracketing, you note that `NOT` takes precedence, so you write `(NOT Num1 < 25)`. `AND` is next according to the precedence rules, so you bracket the `AND`ed conditions to give `(Num2 = 80 AND Num3 > 264)`. Finally, the `OR` is evaluated to give the full condition as

```
IF (NOT Num1 < 25) OR (Num2 = 80 AND Num3 > 264)THEN
   DISPLAY "Done"
END-IF
```

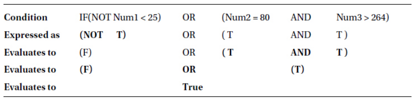
Of course, you can use bracketing to change the order of evaluation. For instance, you can change the previous condition to

```
IF NOT (Num1 < 25 OR Num2 = 80) AND Num3 > 264 THEN
   DISPLAY "Done"
END-IF
```

In the original condition, the order of evaluation was `NOT..AND..OR`, but the new bracketing changes that order to `OR..NOT..AND`. This change has a practical effect on the result of the condition.

Suppose all the simple conditions in the original expression are true. The truth table for that expression yields Table 5-4.

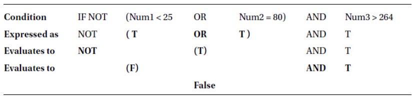
Table 5-4. *IF Statement Evaluation When All the Simple Conditions Are True*



Condition	IF(NOT Num1 < 25)	OR	(Num2 = 80	AND	Num3 > 264)
Expressed as	**(NOT T)**	OR	(T	AND	T)
Evaluates to	(F)	OR	**(T**	**AND**	**T)**
Evaluates to	**(F)**	**OR**		(T)	
Evaluates to		**True**			

The re-bracketed expression yields Table 5-5.

Table 5-5. *The Rebracketed Truth Table*



Condition	IF NOT	(Num1 < 25	OR	Num2 = 80)	AND	Num3 > 264
Expressed as	NOT	**(T**	**OR**	**T)**	AND	T
Evaluates to	**NOT**		(T)		AND	T
Evaluates to		(F)			**AND**	**T**
			False			

Implied Subjects

Although COBOL is often verbose, it does occasionally provide constructs that enable quite succinct statements to be written. The *implied subject* is one of those constructs.

When, in a complex condition, a number of comparisons have to be made against a single data item, it can be tedious to have to repeat the data item for each comparison. For instance, the example code fragment you saw earlier could be rewritten using implied subjects as

```
IF (ScrRow > 0 AND < 25) AND (ScrCol > 0 AND < 81) THEN
   DISPLAY "On Screen"
END-IF
```

In this case, the implied subjects are `ScrRow` and `ScrCol`.

Similarly, using `Grade =` as the implied subject, you can rewrite

```
IF Grade = "A" OR Grade = "B" OR Grade = "C" THEN DISPLAY "Passed
```

as

```
IF Grade = "A" OR "B" OR "C" THEN DISPLAY "Passed"
```

Finally, you can use the implied subject `Num1 >` to rewrite the expression

```
IF Num1 > Num2 AND Num1 > Num3 AND Num1 > Num4 THEN
    DISPLAY "Num1 is the largest"
END-IF
```

as

```
IF Num1 > Num2 AND Num3 AND Num4
    DISPLAY "Num1 is the largest"
END-IF
```

Nested IFs

COBOL allows nested `IF` statements (see Example 5-5). But be aware that although nested `IF` statements may be easy to write, they are somewhat difficult to understand when you return to them after an interval of time. Complex, and nested `IF`, statements are often used as a substitute for clear thinking. When you first attempt to solve a problem, you often don't have a full understanding of it. As a result, your solution may be convoluted and unwieldy. It is often only after you have attempted to solve the problem that you gain sufficient insight to allow you to generate a simpler solution. When you have a better understanding of the problem, you may find that a mere reorganization of your code will greatly reduce both the number and complexity of the `IF` statements required. Simplicity is difficult to achieve but is a highly desirable objective. It is a principle of good program design that your solution should be only as complex as the problem demands.

Example 5-5. Nested `IF..ELSE` Statements

```
*> This example uses nested IF statements including IF..THEN..ELS
*> This is quite a straight forward example of nested IFs but nes
*> can get a lot more convoluted and difficult to understand. It
*> some nested IF statements do not have ELSE branches and others
*> to untangle which ELSE belongs with which IF
   IF InputVal IS NUMERIC
      MOVE InputVal to Num1
      IF Num1 > 5 AND < 25
         IF Num1 < Num2
             MOVE Num2 TO Num1
           ELSE
             MOVE Num1 TO Num2
         END-IF
         DISPLAY "Num1 & Num2 = " Num1 SPACE Num2
       ELSE
         DISPLAY "Num 1 not in range"
      END-IF
    ELSE
      DISPLAY "Input was not numeric"
   END-IF
```

Delimiting Scope: END-IF vs. Period

The scope of an `IF` statement may be delimited by either an `END-IF` or a period (full stop). For a variety of reasons, the explicit `END-IF` delimiter should always be used instead of a period. The period is so problematic that one of the most useful renovations you can perform on legacy COBOL code is to replace the periods with explicit scope delimiters.

There are two main problems with using a period as a scope delimiter. The first is that periods are hard to see, which makes it more difficult to understand the code. The second problem is that a period delimits *all open scopes*. This is a source of many programming errors.

The code fragments in Example 5-6 illustrate the readability problem. Both `IF` statements are supposed to perform the same task. But the scope of the `IF` statement on the left is delimited by an `END-IF`, whereas the statement on the right is delimited by a period.

Example 5-6. Comparing `END-IF` and Period-Delimited `IF` Statements

```
Statement1            Statement1
Statement2            Statement2
IF Num1 > Num2 THEN   IF Num1 > Num2 THEN
   Statement3            Statement3
   Statement4            Statement4
END-IF                Statement5
Statement5            Statement6.
Statement6.
```

Unfortunately, on the right, the programmer has forgotten to follow `Statement4` with a delimiting period. This means `Statement5` and `Statement6` will be included in the scope of the `IF`. They will be executed only if the condition is true. When periods are used to delimit the scope of an `IF` statement, this is an easy mistake to make; and, once made, it is difficult to spot. A period is small and unobtrusive compared to an `END-IF`.

The problem caused by unexpectedly delimiting scope is illustrated by the following code fragment:

```
IF Num1 < 10
   ADD 10 TO Num1
   MULTIPLY Num1 BY 1000 GIVING NUM2
      ON SIZE ERROR  DISPLAY "Error: Num2 too small".
   DISPLAY "When is this shown?".
```

In this fragment, it looks as if the `DISPLAY` on the final line is executed only when `Num1` is less than 10. However, a period has been used to delimit the scope of the `ON SIZE ERROR` (instead of an `END-MULTIPLY` delimiter), and that period also delimits the scope of the `IF` (all open scopes). This means the `DISPLAY` lies outside the scope of the `IF` and so is always executed.

If you replace the periods with explicit scope delimiters, you can see more clearly what is happening:

```
IF Num1 < 10
   ADD 10 TO Num1
   MULTIPLY Num1 BY 1000 GIVING NUM2
      ON SIZE ERROR  DISPLAY "Error: Num2 too small"
   END-MULTIPLY
END-IF
   DISPLAY "When is this shown?".
```

Even though the indentation used in this version is just as misleading as the period-based version, you are not misled. The explicit scope delimiters used for the `IF` and the `MULTIPLY` make the scope of these statements clear.

The use of delimiting periods in the `PROCEDURE DIVISION` is such a source of programming errors that a minimum period style of programming has been advocated by Howard Tompkins [1] and Robert Baldwin [2]. In the examples in this book, I use a variation of the style suggested by Tompkins. Tompkins was writing before the 1985 standard was produced and so was not able to incorporate `END` delimiters into his scheme. Nowadays, you can adopt a style that uses only a single period per paragraph. Although Tompkins has persuasive arguments for placing that period alone on the line in column 12, for aesthetic reasons I use it to terminate the last statement in the paragraph. Whether you prefer the Tompkins lonely period style or my variation, I strongly suggest that you adopt the minimum period style. That way you will save yourself a world of hurt.

Condition Names

Wherever a condition tests a variable for equality to a value, a set of values, or a range of values, that condition can be replaced by a kind of abstract condition called a *condition name*. Wherever it is legal to have a condition, it is legal have a condition name. Just like a condition, a condition name is either true or false.

Condition names allow you to give a meaningful name to a condition while hiding the implementation details of how the condition is detected. For instance,

```
IF CountryCode = 3 OR 7 OR 10 OR 15 THEN
    MOVE 14 TO CurrencyCode
END-IF
```

may be replaced with

```
IF BritishCountry THEN
   SET CurrencyIsPound TO TRUE
END-IF
```

This example illustrates the readability benefits of using condition names. When you encounter code such as

```
IF CountryCode = 3 OR 7 OR 10 OR 15
```

the meaning of what the `IF` statement is testing is not obvious. You can see that `CountryCode` is being tested for particular values, but why? What is the significance of the values 3,7,10, and 15? What is the significance of moving 14 to the `CurrencyCode`? To discover this information, a maintenance programmer has to read external documentation or in-code comments. Now consider the condition name version of the `IF` statement. It is obvious what you are testing because the test has been given a meaningful name. Similarly, the action taken when `BritishCountry` is true is also obvious. No documentation and no comments are required.

Ease of maintenance is also improved. If the coding system changed and the countries of the British Isles were now represented by the codes 4, 12, 18, and 25, only the definition of the condition name would have to be changed. In the version that did not use the condition name, you would have to change the code values in all the places in the program where the condition was tested.

Defining Condition Names

Condition names are sometimes called *level 88s* because they are created in the `DATA DIVISION` using the special level number 88. The metalanguage for defining condition names is given in Figure 5-6.

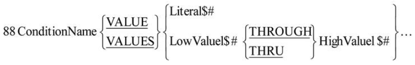


Figure 5-6. *Metalanguage for defining condition names*

Rules

Condition names are always associated with a particular data item and are defined immediately after the definition of that data item. A condition name may be associated with a group data item and elementary data, or even the element of a table. The condition name is automatically set to true or false the moment the value of its associated data item changes.

When the `VALUE` clause is used with condition names, it does not assign a value. Instead, it identifies the value(s) which, if found in the associated data item, make the condition name true.

When identifying the condition values, a single value, a list of values, a range of values, or any combination of these may be specified. To specify a list of values, the entries are listed after the keyword `VALUE`. The list entries may be separated by commas or spaces but must terminate with a period.

Single Condition Name, Single Value

In Example 5-7, the condition name `CityIsLimerick` has been associated with `CityCode` so that if `CityCode` contains the value 2 (listed in the `CityIsLimerick VALUE` clause), the condition name will be automatically set to true.

Example 5-7. Defining and Using a Condition Name

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01  CityCode          PIC 9 VALUE ZERO.
    88 CityIsLimerick VALUE 2.

PROCEDURE DIVISION.
Begin.
    :   :  :  :  :  :  :  :
   DISPLAY "Enter a city code (1-6) - " WITH NO ADVANCING
   ACCEPT CityCode
   IF CityIsLimerick
      DISPLAY "Hey, we're home."
   END-IF
    :   :  :  :  :  :  :  :
```

In the program fragment, `DISPLAY` and `ACCEPT` get a city code from the user. The instant the value in `CityCode` changes, the `CityIsLimerick` condition name will be set to true or false, depending on the value in `CityCode`.

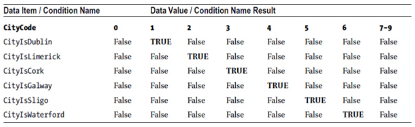
Multiple Condition Names

Several condition names may be associated with a single data item. In Example 5-8, a number of condition names have been associated with `CityCode`. Each condition name is set to true when `CityCode` contains the value listed in the condition name `VALUE` clause. Condition names, like Booleans, can only take the value true or false. If a condition name is not set to true, it is set to false. Table 5-6 shows the Boolean value of each condition name for each value of `CityCode`.

Example 5-8. Associating Many Condition Names with a Data Item

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01  CityCode PIC 9 VALUE ZERO.
    88 CityIsDublin         VALUE 1.
    88 CityIsLimerick       VALUE 2.
    88 CityIsCork           VALUE 3.
    88 CityIsGalway         VALUE 4.
    88 CityIsSligo          VALUE 5.
    88 CityIsWaterford      VALUE 6.
PROCEDURE DIVISION.
Begin.
    :   :  :  :  :  :  :  :
   DISPLAY "Enter a city code (1-6) - " WITH NO ADVANCING
   ACCEPT CityCode
   IF CityIsLimerick
      DISPLAY "Hey, we're home."
   END-IF
   IF CityIsDublin
      DISPLAY "Hey, we're in the capital."
   END-IF
    :   :  :  :  :  :  :  :
```

Table 5-6. *Results for Each Value of* `CityCode`



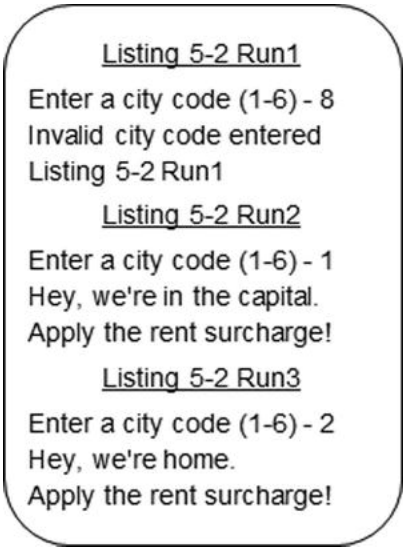
Data Item / Condition Name	Data Value / Condition Name Result							
CityCode	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7–9**
CityIsDublin	False	**TRUE**	False	False	False	False	False	False
CityIsLimerick	False	False	**TRUE**	False	False	False	False	False
CityIsCork	False	False	False	**TRUE**	False	False	False	False
CityIsGalway	False	False	False	False	**TRUE**	False	False	False
CityIsSligo	False	False	False	False	False	**TRUE**	False	False
CityIsWaterford	False	False	False	False	False	False	**TRUE**	False

Overlapping and Multiple-Value Condition Names

When multiple condition names are associated with a single data item, more than one condition name can be true at the same time. In Listing 5-2, `UniversityCity` is true if `CityCode` contains any value between 1 and 4. These values overlap the values of the first four condition names, so if `UniversityCity` is true, then one of those four must also be true.

Listing 5-2. Multiple Condition Names with Overlapping Values

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Listing5-2.
AUTHOR.  Michael Coughlan.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  CityCode PIC 9 VALUE ZERO.
    88 CityIsDublin         VALUE 1.
    88 CityIsLimerick       VALUE 2.
    88 CityIsCork           VALUE 3.
    88 CityIsGalway         VALUE 4.
    88 CityIsSligo          VALUE 5.
    88 CityIsWaterford      VALUE 6.
    88 UniversityCity       VALUE 1 THRU 4.
    88 CityCodeNotValid     VALUE 0, 7, 8, 9.

PROCEDURE DIVISION.
Begin.
   DISPLAY "Enter a city code (1-6) - " WITH NO ADVANCING
   ACCEPT CityCode
   IF CityCodeNotValid
      DISPLAY "Invalid city code entered"
     ELSE
       IF CityIsLimerick
          DISPLAY "Hey, we're home."
       END-IF
       IF CityIsDublin
          DISPLAY "Hey, we're in the capital."
       END-IF
       IF UniversityCity
          DISPLAY "Apply the rent surcharge!"
       END-IF
   END-IF
   STOP RUN.
```



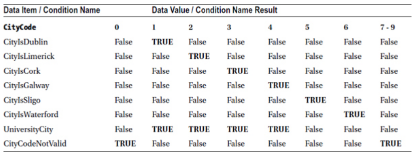
```
Listing 5-2 Run1
Enter a city code (1-6) - 8
Invalid city code entered
Listing 5-2 Run1
Listing 5-2 Run2
Enter a city code (1-6) - 1
Hey, we're in the capital.
Apply the rent surcharge!
Listing 5-2 Run3
Enter a city code (1-6) - 2
Hey, we're home.
Apply the rent surcharge!
```

The list of values that follows a condition name may be a single value, a number of values, or a range of values, or any mixture of these. When a range is specified, the word `THROUGH` or `THRU` is used to separate the minimum and maximum values in the range. In Listing 5-2, `UniversityCity` is true if `CityCode` contains any value between 1 and 4, whereas `CityCodeNotValid` is true if `CityCode` contains a value of 0 or 7 or 8 or 9. In Listing 5-2 I have chosen to list the individual values for `CityCodeNotValid`, but the value list could have been written as:

```
88 CityCodeNotValid     VALUE 0, 7 THRU 9.
```

Table 5-7 shows the Boolean value of the condition names for each value of `CityCode`.

Table 5-7*. Results for Each Value of* `CityCode`



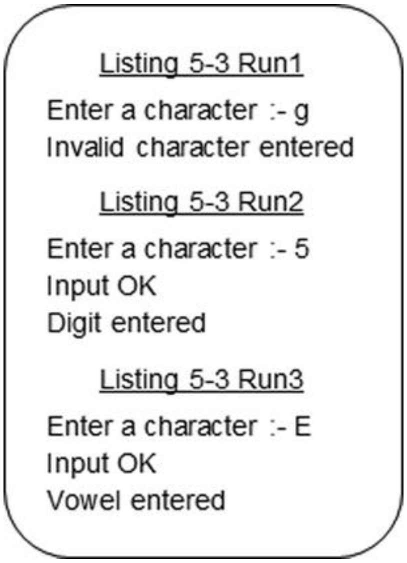
Data Item / Condition Name	Data Value / Condition Name Result							
CityCode	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7 - 9**
CityIsDublin	False	**TRUE**	False	False	False	False	False	False
CityIsLimerick	False	False	**TRUE**	False	False	False	False	False
CityIsCork	False	False	False	**TRUE**	False	False	False	False
CityIsGalway	False	False	False	False	**TRUE**	False	False	False
CityIsSligo	False	False	False	False	False	**TRUE**	False	False
CityIsWaterford	False	False	False	False	False	False	**TRUE**	False
UniversityCity	False	**TRUE**	**TRUE**	**TRUE**	**TRUE**	False	False	False
CityCodeNotValid	**TRUE**	False	False	False	False	False	False	**TRUE**

Values Can Be Alphabetic or Numeric

The list of values specified for a condition name can be numeric or alphabetic, as shown in Listing 5-3.

Listing 5-3*.* Multiple Condition Names with Overlapping Values

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Listing5-3.
AUTHOR.  Michael Coughlan.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 InputChar     PIC X.
   88 Vowel      VALUE  "A","E","I","O","U".
   88 Consonant  VALUE  "B" THRU "D", "F","G","H"
                        "J" THRU "N", "P" THRU "T"
                        "V" THRU "Z".
   88 Digit      VALUE  "0" THRU "9".
   88 ValidChar  VALUE  "A" THRU "Z", "0" THRU "9".

PROCEDURE DIVISION.
Begin.
   DISPLAY "Enter a character :- " WITH NO ADVANCING
   ACCEPT InputChar
   IF ValidChar
      DISPLAY "Input OK"
    ELSE
      DISPLAY "Invalid character entered"
   END-IF
   IF Vowel
      DISPLAY "Vowel entered"
   END-IF
   IF Digit
      DISPLAY "Digit entered"
   END-IF
   STOP RUN.
```



```
Listing 5-3 Run1

Enter a character :- g
Invalid character entered

Listing 5-3 Run2

Enter a character :- 5
Input OK
Digit entered

Listing 5-3 Run3

Enter a character :- E
Input OK
Vowel entered
```

List Values Can Be Whole Words

Although I have used single characters in the examples so far, condition names are not restricted to values with only single characters. Whole words can be used if required, as shown in Listing 5-4.

Listing 5-4. Words as Value Items

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Listing5-4.
AUTHOR.  Michael Coughlan.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  MakeOfCar        PIC X(10).
    88 VolksGroup  VALUE "skoda", "seat",
                         "audi", "volkswagen".
    88 GermanMade  VALUE "volkswagen", "audi",
                         "mercedes", "bmw",
                         "porsche".
PROCEDURE DIVISION.
Begin.
   DISPLAY "Enter the make of car - " WITH NO ADVANCING
   ACCEPT MakeOfCar
   IF VolksGroup AND GermanMade
      DISPLAY "Your car is made in Germany by the Volkswagen Grou
    ELSE
      IF VolksGroup
         DISPLAY "Your car is made by the Volkswagen Group."
      END-IF
      IF GermanMade
         DISPLAY "Your car is made in Germany."
      END-IF
   END-IF
   STOP RUN.
```

Using Condition Names Correctly

A condition name should express the true condition being tested. It should not express the test that sets the condition name to true. For instance, in Listing 5-2, a value of 1 in the data item `CityCode` indicates that the city is Dublin, a value of 2 means the city is Limerick, and so on. These condition names allow you to replace conditions such as

```
IF CityCode = 1
```

and

```
IF CityCode = 2
```

with the more meaningful statements

```
IF CityIsDublin
```

and

```
IF  CityIsLimerick.
```

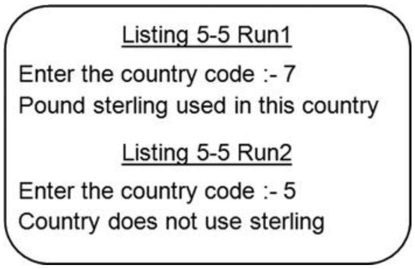
Many COBOL beginners would use condition names such as CityCode-Is1 or CityCodeIs2 to express these conditions. Those condition names are meaningless because they express the value that makes the condition name true instead of expressing the meaning or significance of CityCode containing a particular value. A value of 1 or 2 in CityCode is how you detect that the city is Dublin or Limerick. It is not the value of CityCode that ultimately interests you; it is the meaning or significance of that value.

Example Program

Listing 5-5 is a small but complete program showing how the British-Country and CurrencyIsPound condition names might be defined and used. There is something unusual about this example, however. What do you imagine happens to the associated data item when the CurrencyIs-Pound condition name is set to true?

Listing 5-5. Detecting BritishCountry and Setting and Using CurrencyIsPound

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Listing5-5.
AUTHOR.  Michael Coughlan.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 CountryCode       PIC 999 VALUE ZEROS.
   88 BritishCountry VALUES 3, 7, 10, 15.

01 CurrencyCode        PIC 99 VALUE ZEROS.
   88 CurrencyIsPound  VALUE 14.
   88 CurrencyIsEuro   VALUE 03.
   88 CurrencyIsDollar VALUE 28.

PROCEDURE DIVISION.
Begin.
   DISPLAY "Enter the country code :- " WITH NO ADVANCING
   ACCEPT CountryCode

   IF BritishCountry THEN
      SET CurrencyIsPound TO TRUE
   END-IF
   IF CurrencyIsPound THEN
      DISPLAY "Pound sterling used in this country"
    ELSE
      DISPLAY "Country does not use sterling"
   END-IF
   STOP RUN.
```



Listing 5-5 Run1

```
Enter the country code :- 7
Pound sterling used in this country
```

Listing 5-5 Run2

```
Enter the country code :- 5
Country does not use sterling
```

Setting a Condition Name to True

In Listing 5-5, the SET verb is used to set CurrencyIsPound to true. The way condition names normally work is that a value placed into the associated data item automatically sets the condition names that list that value to true. When a condition name is manually set to true using the SET verb, the value listed for that condition name is forced into the associated data item. In Listing 5-5, when the SET verb is used to set CurrencyIs-Pound to true, the value 14 is forced into CurrencyCode.

When a condition name that lists more than one value is set to true, the first of the values listed is forced into the associated data item. For instance, if BritishCountry were set to true, then the value 3 would be forced into CountryCode.

ISO 2002 In standard ANS 85 COBOL, the SET verb cannot be used to set a condition name to false. This can be done in ISO 2002 COBOL, but in that case the level 88 entry must be extended to include the phrase

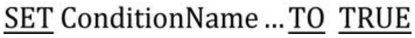
WHEN SET TO FALSE IS LiteralValue$#

To set a condition name to true, you use the SET verb. You might think, therefore, that the SET verb is used only for manipulating condition names. But the SET verb is a strange fish. It is used for a variety of unconnected purposes. For instance, it is used to set a condition name to true. It is used to increment or decrement an index item. It is used to assign the value of an index to an ordinary data item and vice versa. It is used to set On or Off the switches associated with mnemonic names. In ISO 2002 COBOL, it is used to manipulate pointer variables (yes, ISO 2002 COBOL has pointers) and object references. It is often the target of implementer extensions.

Because the `SET` verb has so many different unrelated uses, instead of dealing with it as a single topic I discuss each format as you examine the construct to which it is most closely related.

SET Verb Metalanguage

Figure 5-7 shows the metalanguage for the version of the SET verb that is used to set a condition name to true. When the `SET` verb is used to set a condition name, the first condition value specified after the `VALUE` clause in the definition is moved to the associated data item. So setting the condition name to true changes the value of the associated data item. This can lead to some interesting data-manipulation opportunities.



SET ConditionName ... TO TRUE

Figure 5-7. Metalanguage for the `SET` verb condition name version

In summary, any operation that changes the value of the data item may change the status of the associated condition names, and any operation that changes the status of a condition name will change the value of its associated data item.

SET Verb Examples

In ANS 85 COBOL, you cannot use the `SET` verb to set a condition name to false. But you can work around this restriction. Consider Example 5-9. This is more a pattern for processing sequential files than real COBOL code, but it serves to illustrate the point.

Example 5-9. Setting the `EndOfFile` Condition Name

```
01  EndOfFileFlag     PIC 9 VALUE ZERO.
    88 EndOfFile      VALUE 1.
    88 NotEndOfFile   VALUE 0.

    :   :  :  :  :  :  :  :

READ InFile
   AT END SET EndOfFile TO TRUE
END-READ
PERFORM UNTIL EndOfFile
   Process Record
   READ InFile
      AT END SET EndOfFile TO TRUE
   END-READ
END-PERFORM
Set NotEndOfFile TO TRUE.
```

In this example, the condition name `EndOfFile` has been set up to flag that the end of the file has been reached. You cannot set `EndOfFile` to false, but you can work around this problem by setting another condition name associated with the same data item to true. When `EndOfFile` is set to true, 1 is forced into the data item `EndOfFileFlag`, and this automatically sets `NotEndOfFile` to false. Similarly, when `NotEndOfFile` is set to true, 0 is forced into `EndOfFileFlag`, and this automatically sets `EndOfFile` to false.

Design Pattern: Reading a Sequential File

Because this is your first look at how COBOL processes sequential (as opposed to direct access) files, it might be useful to preview some of the material in Chapter 7 by providing a brief explanation now. The `READ` verb copies a record (a discrete package of data) from the file on backing storage and places it into an area of memory set up to store it. When the `READ` attempts to read a record from the file but discovers that the end of the file has been reached, it activates the `AT END` clause and executes whatever statements follow that clause.

Example 5-9 shows the pattern you generally use to process a stream of items when you can only discover that you have reached the end of the stream by attempting to read the next item. In this pattern, a loop processes the data in the stream. Outside the loop, you have a read to get the first item in the stream or to discover that the stream is empty. Inside the loop, you have statements to process the stream item and get the next item in the stream.

Why do you have this strange arrangement? The chief reason is that this arrangement allows you to place the read at the end of the loop body so that as soon as the end of the file is detected, the loop can be terminated. If you used a structure such as

```
PERFORM UNTIL EndOfFile
   READ InFile
```

```
      AT END SET EndOfFile TO TRUE
   END-READ
   Process Record
END-PERFORM
```

then when the end of file was detected, the program would still attempt to process the nonexistent record. Of course, the last valid record would still be in memory, so that last record would be processed twice. Many COBOL beginners make this programming error.

Many beginners attempt to solve this problem by only processing the record if the end of file has not been detected. They use a structure like this:

```
PERFORM UNTIL EndOfFile
   READ InFile
      AT END SET EndOfFile TO TRUE
   END-READ
   IF NOT EndOfFile
     Process Record
   END-IF
END-PERFORM
```

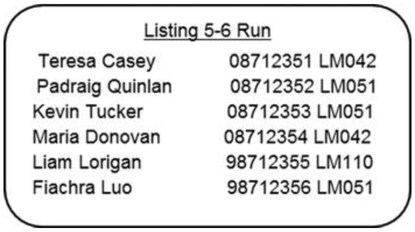
The problem with this arrangement is that the `IF` statement will be executed for every record in the file. Because COBOL often deals with very large data sets, this could amount to the execution of millions, maybe even hundreds of millions, of unnecessary statements. It is more elegant and more efficient to use what is called the *read-ahead* technique. The read-ahead has a read outside the loop to get the first record and a read inside the loop to get the remaining records. This approach has the added advantage of allowing the empty file condition to be detected before the loop is entered.

Group Item Condition Names

In Example 5-9, stand-alone condition names were used to flag the end-of-file condition. Because the `EndOfFile` condition name is closely related to the file, it would be better if the declaration of the condition name were kept with the file declaration. The example program in Listing 5-6 shows how that might be done. It also demonstrates how a condition name can be used with a group (as opposed to elementary) data item.

Listing 5-6. Reading a File and Setting the EndOfStudentFile Condition Name

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Listing5-6.
AUTHOR. Michael Coughlan.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
   SELECT StudentFile ASSIGN TO "Listing5-6-TData.Dat"
    ORGANIZATION IS LINE SEQUENTIAL.

DATA DIVISION.
FILE SECTION.
FD StudentFile.
01 StudentDetails.
     88  EndOfStudentFile  VALUE HIGH-VALUES.
     02  StudentId       PIC X(8).
     02  StudentName     PIC X(25).
     02  CourseCode      PIC X(5).

PROCEDURE DIVISION.
Begin.
   OPEN INPUT StudentFile
   READ StudentFile
      AT END SET EndOfStudentFile TO TRUE
   END-READ
   PERFORM UNTIL EndOfStudentFile
      DISPLAY StudentName SPACE StudentId SPACE CourseCode
      READ StudentFile
         AT END SET EndOfStudentFile TO TRUE
      END-READ
   END-PERFORM
   CLOSE StudentFile
   STOP RUN.
```



Listing 5-6 Run

```
Teresa Casey           08712351 LM042
Padraig Quinlan        08712352 LM051
Kevin Tucker           08712353 LM051
Maria Donovan          08712354 LM042
Liam Lorigan           98712355 LM110
Fiachra Luo            98712356 LM051
```

In Listing 5-6, the condition name `EndOfStudentFile` is associated with the group item (which also happens to be a record) `StudentDetails`. When `EndOfStudentFile` is set to true, the entire `StudentDe-`

`tails` area of storage (38 characters) is flushed with highest possible character value.

This arrangement has two major advantages:

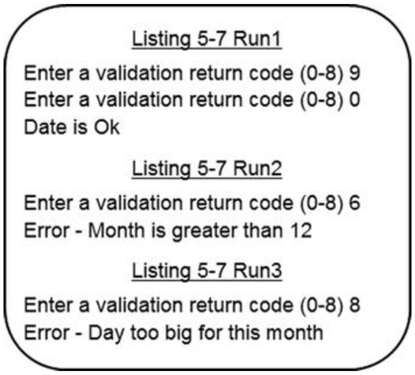
- The `EndOfStudentFile` condition name is kept with its associated file.
- Flushing the record with `HIGH-VALUES` at the end of the file eliminates the need for an explicit condition when doing a key-matching update of a sequential file.

Condition Name Tricks

When you become aware that setting a condition name forces a value into the associated data item, it is tempting to see just how far you can take this idea. Listing 5-7 takes advantage of the way condition names work to automatically move an appropriate error message into a message buffer. The program is just a stub to test this error-messaging idea; it doesn't actually validate the date. Instead, the user manually enters one of the codes that would be returned by the date-validation routine.

Listing 5-7. Using Condition Names to Set Up a Date-Validation Error Message

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Listing5-7.
AUTHOR. Michael Coughlan.

DATA DIVISION.
WORKING-STORAGE SECTION.
01 ValidationReturnCode  PIC 9.
   88 DateIsOK           VALUE 0.
   88 DateIsInvalid      VALUE 1 THRU 8.
   88 ValidCodeSupplied  VALUE 0 THRU 8.

01 DateErrorMessage      PIC X(35) VALUE SPACES.
   88 DateNotNumeric     VALUE "Error - The date must be numeric"
   88 YearIsZero         VALUE "Error - The year cannot be zero".
   88 MonthIsZero        VALUE "Error - The month cannot be zero"
   88 DayIsZero          VALUE "Error - The day cannot be zero".
   88 YearPassed         VALUE "Error - Year has already passed".
   88 MonthTooBig        VALUE "Error - Month is greater than 12"
   88 DayTooBig          VALUE "Error - Day greater than 31".
   88 TooBigForMonth     VALUE "Error - Day too big for this mont

PROCEDURE DIVISION.
Begin.
   PERFORM ValidateDate UNTIL ValidCodeSupplied
   EVALUATE ValidationReturnCode
      WHEN    0   SET DateIsOK       TO TRUE
      WHEN    1   SET DateNotNumeric TO TRUE
      WHEN    2   SET YearIsZero     TO TRUE
      WHEN    3   SET MonthIsZero    TO TRUE
      WHEN    4   SET DayIsZero      TO TRUE
      WHEN    5   SET YearPassed     TO TRUE
      WHEN    6   SET MonthTooBig    TO TRUE
      WHEN    7   SET DayTooBig      TO TRUE
      WHEN    8   SET TooBigForMonth TO TRUE
   END-EVALUATE

   IF DateIsInvalid THEN
      DISPLAY DateErrorMessage
   END-IF
   IF DateIsOK
      DISPLAY "Date is Ok"
   END-IF
   STOP RUN.

ValidateDate.
   DISPLAY "Enter a validation return code (0-8) " WITH NO ADVANC
   ACCEPT ValidationReturnCode.
```



```
Listing 5-7 Run1
Enter a validation return code (0-8) 9
Enter a validation return code (0-8) 0
Date is Ok

Listing 5-7 Run2
Enter a validation return code (0-8) 6
Error - Month is greater than 12

Listing 5-7 Run3
Enter a validation return code (0-8) 8
Error - Day too big for this month
```

EVALUATE

In Listing 5-7, the `EVALUATE` verb is used to `SET` a particular condition name depending on the value in the `ValidationReturnCode` data item. You probably did not have much difficulty working out what the `EVALUATE` statement is doing because it has echoes of how the `switch`/`case` statement works in other languages. Ruby programmers, with their `when`-branched `case` statement, were probably particularly at home. But the resemblance of `EVALUATE` to the `case`/`switch` used in other languages is superficial. `EVALUATE` is far more powerful than these constructs. Even when restricted to one subject, `EVALUATE` is more powerful because it is not limited to ordinal types. When used with multiple subjects, `EVALUATE` is a significantly more powerful construct. One common use for the multiple-subject `EVALUATE` is the implementation of decision-table logic.

Decision Tables

A *decision table* is a way to model complicated logic in a tabular form. Decision tables are often used by systems analysts to express business rules that would be too complicated and/or too confusing to express in a textual form.

For instance, suppose an amusement park charges different admission fees depending on the age and height of visitors, according to the following rules:

- If the person is younger than 4 years old, admission is free.
- If the person is between 4 and 7, admission is $10.
- If between 8 and 12, admission is $15.
- If between 13 and 64, admission is $25.
- If 65 or older, admission is $10.
- In addition, in view of the height restrictions on many rides, persons shorter than 48 inches who are between the ages of 8 and 64 receive a discount. Persons between 8 and 12 are charged a $10 admission fee, whereas those between the ages of 13 and 64 are charged $18.

You can represent this textual specification using the decision table in Table 5-8.

Table 5-8. *Amusement Park Decision Table*

Age	Height in inches	Admission
< 4	NA	$0
4 - 7	NA	$10
8 - 12	Height >= 48	$15
8 - 12	Height < 48	$10
13 - 64	Height >= 48 inches	$25
13 - 64	Height < 48	$18
>= 65	NA	$10

EVALUATE Metalanguage

The `EVALUATE` metalanguage (see Figure 5-8) looks very complex but is actually fairly easy to understand. It is, though, somewhat difficult to explain in words, so I mainly use examples to explain how it works.

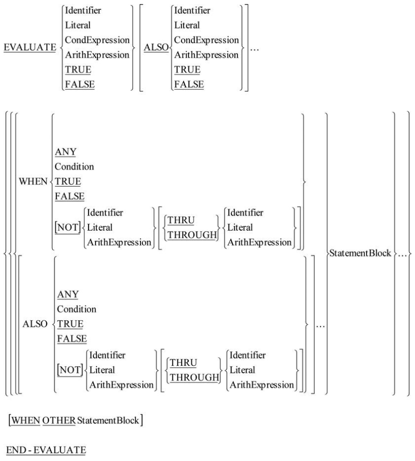


Figure 5-8. Metalanguage for the `EVALUATE` verb

Notes

The following are the `WHEN` branch rules:

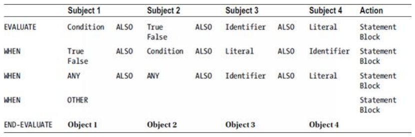
- Only one `WHEN` branch is chosen per execution of `EVALUATE`.
- The order of the `WHEN` branches is important because checking of the branches is done from top to bottom.
- If any `WHEN` branch is chosen, the `EVALUATE` ends. The `break` required in other languages to stop execution of the remaining branches is not required in COBOL.
- If none of the `WHEN` branches can be chosen, the `WHEN OTHER` branch (if it exists) is executed.
- If none of the `WHEN` branches can be chosen, and there is no `WHEN OTHER` phrase, the `EVALUATE` simply terminates.

The items immediately after the word `EVALUATE` and before the first `WHEN` are called *subjects*. The items between the `WHEN` and its statement block are called *objects*.

The number of subjects must equal the number of objects, and the objects must be compatible with the subjects. For instance, if the subject is a condition, then the object must be either `TRUE` or FALSE. If the subject is a data item, then the object must be either a literal value or a data item.

Table 5-9 lists the combinations you may have. If there are four subjects, then each `WHEN` branch must list four objects. If the value of a particular object does not matter, the keyword `ANY` may be used.

Table 5-9. *`EVALUATE` Subject/Object Combinations*



	Subject 1		Subject 2		Subject 3		Subject 4	Action
EVALUATE	Condition	ALSO	True False	ALSO	Identifier	ALSO	Literal	Statement Block
WHEN	True False	ALSO	Condition	ALSO	Literal	ALSO	Identifier	Statement Block
WHEN	ANY	ALSO	ANY	ALSO	Identifier	ALSO	Literal	Statement Block
WHEN	OTHER							Statement Block
END-EVALUATE	**Object 1**		**Object 2**		**Object 3**		**Object 4**	

EVALUATE Examples

This section looks at three examples of the `EVALUATE` verb.

Payment Totals Example

Shoppers choose the method of payment as Visa, MasterCard, American Express, Check, or Cash. A program totals the amount paid by each payment method. After a sale, the sale value is added to the appropriate total. Condition names (`ByVisa`, `ByMasterCard`, `ByAmericanExpress`, `ByCheck`, `ByCash`) have been set up for each of the payment methods.

You could code this as follows:

```
IF ByVisa ADD SaleValue TO VisaTotal
   ELSE
     IF ByMasterCard ADD SaleValue TO MasterCardTotal
        ELSE
          IF ByAmericanExpress ADD SaleValue TO AmericanExpressTo
             ELSE
               IF ByCheck ADD SaleValue TO CheckTotal
                  ELSE
                    IF ByCash ADD SaleValue TO CashTotal
                    END-IF
               END-IF
          END-IF
     END-IF
END-IF
```

You can replace these nested IF statements with the neater and easier-to-understand EVALUATE statement:

```
EVALUATE TRUE
  WHEN ByVisa            ADD SaleValue TO VisaTotal
  WHEN ByMasterCard      ADD SaleValue TO MasterCardTotal
  WHEN ByAmericanExpress ADD SaleValue TO AmericanExpressTotal
  WHEN ByCheck           ADD SaleValue TO CheckTotal
  WHEN ByCash            ADD SaleValue TO CashTotal
END-EVALUATE
```

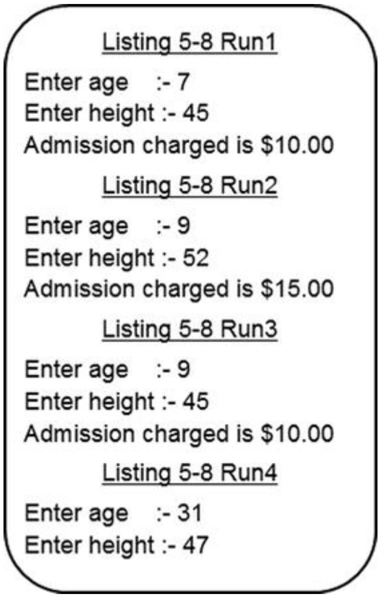
In this example, the objects must all be either conditions or condition names, because the subject is TRUE.

Amusement Park Example

EVALUATE can be used to encode a decision table. Listing 5-8 shows how the Amusement Park decision table from Table 5-8 might be encoded.

Listing 5-8. Amusement Park Admission

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Listing5-8.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 Age            PIC 99 VALUE ZERO.
   88 Infant      VALUE 0 THRU 3.
   88 YoungChild  VALUE 4 THRU 7.
   88 Child       VALUE 8 THRU 12.
   88 Visitor     VALUE 13 THRU 64.
   88 Pensioner   VALUE 65 THRU 99.

01 Height         PIC 999 VALUE ZERO.

01 Admission      PIC $99.99.

PROCEDURE DIVISION.
Begin.
   DISPLAY "Enter age    :- " WITH NO ADVANCING
   ACCEPT Age
   DISPLAY "Enter height :- " WITH NO ADVANCING
   ACCEPT Height

   EVALUATE TRUE        ALSO       TRUE
     WHEN   Infant      ALSO       ANY         MOVE 0  TO Admissi
     WHEN   YoungChild  ALSO       ANY         MOVE 10 TO Admissi
     WHEN   Child       ALSO   Height >= 48    MOVE 15 TO Admissi
     WHEN   Child       ALSO   Height < 48     MOVE 10 TO Admissi
     WHEN   Visitor     ALSO   Height >= 48    MOVE 25 TO Admissi
     WHEN   Visitor     ALSO   Height < 48     MOVE 18 TO Admissi
     WHEN   Pensioner   ALSO       ANY         MOVE 10 TO Admissi
   END-EVALUATE

   DISPLAY "Admission charged is " Admission
   STOP RUN.
```



```
Listing 5-8 Run1
Enter age    :- 7
Enter height :- 45
Admission charged is $10.00
Listing 5-8 Run2
Enter age    :- 9
Enter height :- 52
Admission charged is $15.00
Listing 5-8 Run3
Enter age    :- 9
Enter height :- 45
Admission charged is $10.00
Listing 5-8 Run4
Enter age    :- 31
Enter height :- 47
```

Acme Book Club Example

The Acme Book Club is the largest online book club in the world. The book club sells books to both members and non-members all over the world. For each order, Acme applies a percentage discount based on the quantity of books in the current order, the value of books purchased in the last three months (last quarter), and whether the customer is a member of the Book Club.

Acme uses the decision table in Table 5-10 to decide what discount to apply. Listing 5-9 is a small test program that uses `EVALUATE` to implement the decision table.

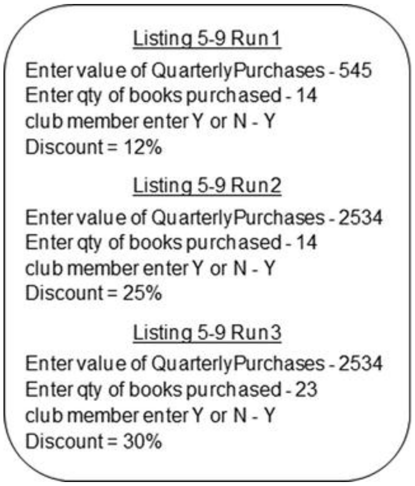
Table 5-10*. Acme Book Club Discount Decision Table*

QtyOfBooks	QuarterlyPurchases (QP)	ClubMember	% Discount
1–5	< 500	ANY	0
1–5	< 2000	Y	7
1–5	< 2000	N	5
1–5	>= 2000	Y	10
1–5	>= 2000	N	8
6–20	< 500	Y	3
6–20	< 500	N	2
6–20	< 2000	Y	12
6–20	< 2000	N	10
6–20	>= 2000	Y	25
6–20	>= 2000	N	15
21–99	< 500	Y	5
21–99	< 500	N	3
21–99	< 2000	Y	16
21–99	< 2000	N	15
21–99	>= 2000	Y	30
21–99	>= 2000	N	20

Listing 5-9*.* Acme Book Club Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Listing5-9.
AUTHOR. Michael Coughlan.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 Member          PIC X VALUE SPACE.

01 QP              PIC 9(5) VALUE ZEROS.
*> QuarterlyPurchases

01 Qty             PIC 99 VALUE ZEROS.

01 Discount        PIC 99 VALUE ZEROS.

PROCEDURE DIVISION.
Begin.
   DISPLAY "Enter value of QuarterlyPurchases - " WITH NO ADVANCI
   ACCEPT QP
   DISPLAY "Enter qty of books purchased - " WITH NO ADVANCING
   ACCEPT Qty
   DISPLAY "club member enter Y or N - " WITH NO ADVANCING
   ACCEPT Member
```

```
EVALUATE Qty      ALSO   TRUE     ALSO Member
  WHEN  1 THRU 5  ALSO QP <  500  ALSO ANY  MOVE 0  TO Discoun
  WHEN  1 THRU 5  ALSO QP <  2000 ALSO "Y"  MOVE 7  TO Discoun
  WHEN  1 THRU 5  ALSO QP <  2000 ALSO "N"  MOVE 5  TO Discoun
  WHEN  1 THRU 5  ALSO QP >= 2000 ALSO "Y"  MOVE 10 TO Discoun
  WHEN  1 THRU 5  ALSO QP >= 2000 ALSO "N"  MOVE 8  TO Discoun

  WHEN  6 THRU 20 ALSO QP <  500  ALSO "Y"  MOVE 3  TO Discoun
  WHEN  6 THRU 20 ALSO QP <  500  ALSO "N"  MOVE 2  TO Discoun
  WHEN  6 THRU 20 ALSO QP <  2000 ALSO "Y"  MOVE 12 TO Discoun
  WHEN  6 THRU 20 ALSO QP <  2000 ALSO "N"  MOVE 10 TO Discoun
  WHEN  6 THRU 20 ALSO QP >= 2000 ALSO "Y"  MOVE 25 TO Discoun
  WHEN  6 THRU 20 ALSO QP >= 2000 ALSO "N"  MOVE 15 TO Discoun

  WHEN 21 THRU 99 ALSO QP <  500  ALSO "Y"  MOVE 5  TO Discoun
  WHEN 21 THRU 99 ALSO QP <  500  ALSO "N"  MOVE 3  TO Discoun
  WHEN 21 THRU 99 ALSO QP <  2000 ALSO "Y"  MOVE 16 TO Discoun
  WHEN 21 THRU 99 ALSO QP <  2000 ALSO "N"  MOVE 15 TO Discoun
  WHEN 21 THRU 99 ALSO QP >= 2000 ALSO "Y"  MOVE 30 TO Discoun
  WHEN 21 THRU 99 ALSO QP >= 2000 ALSO "N"  MOVE 20 TO Discoun
END-EVALUATE
DISPLAY "Discount = " Discount "%"
STOP RUN.
```



Listing 5-9 Run1

```
Enter value of QuarterlyPurchases - 545
Enter qty of books purchased - 14
club member enter Y or N - Y
Discount = 12%
```

Listing 5-9 Run2

```
Enter value of QuarterlyPurchases - 2534
Enter qty of books purchased - 14
club member enter Y or N - Y
Discount = 25%
```

Listing 5-9 Run3

```
Enter value of QuarterlyPurchases - 2534
Enter qty of books purchased - 23
club member enter Y or N - Y
Discount = 30%
```

Summary

The three classic constructs of structured programming are sequence, selection, and iteration. You have already noted that a COBOL program starts execution with the first statement in the PROCEDURE DIVISION and then continues to execute the statements one after another in sequence until the STOP RUN or the end-of-the-program text is encountered, unless some other statement changes the order of execution. In this chapter, you examined the IF and EVALUATE statements. These statements allow a program to selectively execute program statements. In the next chapter, you discover how iteration, the final classic construct, is implemented in COBOL.

References

1 . Tompkins HE. In defense of teaching structured COBOL as computer science (or, notes on being sage struck). ACM SIGPLAN Notices. 1983; 18(4): 86-94.

2 . Baldwin RR. A note on H.E. Tompkins's minimum-period COBOL style. ACM SIGPLAN Notices. 1987; 22(5): 27-31. http://doi.acm.org/10.1145/25267.25273 (http://doi.acm.org/10.1145/25267.25273)

doi: 10.1145/25267.25273

LANGUAGE KNOWLEDGE EXERCISES

Getting out your 2B pencil once more, write answers to the following questions.

1. For each of the following condition names, which do you consider to be inappropriately named? Suggest more suitable names for these only.

```
01 Country-Code        PIC XX.
   88 ~~Code-Is-US~~       VALUE "US".
   United States

01 Operating-System    PIC X(15).
```

Not clear ✓

```
   88 ~~Windows-Or-UNIX~~  VALUE "WINDOWS".
```

windows

```
01 Room-Type              PIC X(20).
   88 Double-Room         VALUE "DOUBLE".
```

```
   88 Single-Room         VALUE "SINGLE".
```

2. Write an IF statement that uses the SET verb to manually set the condition name InvalidCode to true if DeptCode contains anything except 1, 6, or 8. 01 Deptcode Pic 9 values 0, 2 thru 5, 7

✗ If DeptCode SET InvalidCode TO True.
End-If.

If NOT (DeptCode =1 or 6 or 8) THEN
    SET InvalidCode TO TRUE.
END-IF

3. Assume the variable DeptCode in question 2 is described as

```
01    DeptCode             PIC 9.
```

88 InvalidCode value 0, 2 THRU5, 7, 9.

Write a level 88 condition name called InvalidCode that is automatically set to true when the statement ACCEPT DeptCode accepts any value other than 1, 6, or 8.

4. In each of the following five groups of skeleton IF statements, state whether the statements in each group have the same effect (in the sense that they evaluate to true or false). Answer yes or no.

Do these statements have the same effect? Answer

```
IF Num1 = 1 OR Num1 NOT = 1...
IF NOT (Num1 = 1 AND Num1 = 2) ...
```

always true ✓

```
IF TransCode IS NOT = 3 OR Total NOT > 2550  ...
IF NOT (TransCode IS = 3 OR Total > 2550) ...
```

X

```
IF Num1 = 31 OR Num2 = 12 AND Num3 = 23 OR Num4 = 6 ...
IF (Num1 = 31 OR (Num2 = 12 AND Num3 = 23)) OR Num4 = 6 ...
```

```
IF Num1 = 15 OR Num1 = 12 OR Num1 = 7 AND City = "Cork" ...
IF (Num1 = 15 OR Num1 = 12 OR Num1 = 7) AND City = "Cork" .
```

X

```
IF (Num1 = 1 OR Num1 = 2) AND (Num2 = 6 OR Num2 = 8) ...
IF Num1 = 1 OR Num1 = 2 AND Num2 = 6 OR Num2 =  8 ...
```

X

5. Write an EVALUATE statement to implement the decision part of a game of rock, paper, scissors. Most of the program has been written for you. Just complete the EVALUATE. ADD a WHEN OTHER branch to the EVALUATE to detect when a player enters a code other than 1, 2, or 3.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Listing5-10.
AUTHOR. Michael Coughlan.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 PlayerGuess-A    PIC 9  VALUE 1.
   88 Rock-A        VALUE 1.
   88 Paper-A       VALUE 2.
   88 Scissors-A    VALUE 3.

01 PlayerGuess-B    PIC 9  VALUE 2.
   88 Rock-B        VALUE 1.
   88 Paper-B       VALUE 2.
   88 Scissors-B    VALUE 3.

PROCEDURE DIVISION.
BEGIN.
   DISPLAY "Guess for player A (1=rock, 2=scissors, 3=paper
           WITH NO ADVANCING
   ACCEPT PlayerGuess-A
   DISPLAY "Guess for player B (1=rock, 2=scissors, 3=paper
           WITH NO ADVANCING
   ACCEPT PlayerGuess-B
```

EVALUATE____

```
Evaluate True Also True
when Rock-A Also Rock-B Display "Draw"
when Rock-A Also Paper-B Display "player B Wins"
when Rock-A Also Scissor-B. Display "player A Wins"

when other Display "Evaluate problem"
End-Evaluate.
Stop Run.
```

PROGRAMMING EXERCISE

Listing 4-2 is a program that accepts two numbers from the user, multiplies them together, and then displays the result. Modify the program so that

- It also accepts an operator symbol (`+ - / *`).
- It uses `EVALUATE` to discover which operator has been entered and to apply that operator to the two numbers entered.
- It uses the condition name `ValidOperator` to identify the valid operators and only displays the result if the operator entered is valid.
- The `Result` data item is changed to accommodate the possibility that subtraction may result in a negative value.
- The `Result` data item is changed to accommodate the decimal fractions that may result from division. The result data item should be able to accept values with up to two decimal places (for example, 00.43 or 00.74).

LANGUAGE KNOWLEDGE EXERCISES—ANSWERS

1. For each of the following condition names, which do you consider to be inappropriately named? Suggest more suitable names for these only.

```
01 Country-Code         PIC XX.
   88 UnitedStates      VALUE "US".
* Change
* Example of use  - IF UnitedStates DISPLAY "We are in Amer


01 Operating-System     PIC X(15).
   88 Windows           VALUE " WINDOWS".
* Change
* Example of use  - IF Windows DISPLAY "Windows is best" EN

01 Room-Type            PIC X(20).
   88 Double-Room       VALUE "DOUBLE".
   88 Single-Room       VALUE "SINGLE".
* No change.
* Example of use -IF Double-Room ADD DoubleRoomSurchage TO
```

2. Write an `IF` statement that uses the `SET` verb to manually set the condition name `InvalidCode` to true if `DeptCode` contains anything except 1, 6, or 8.

```
IF NOT (DeptCode = 1 OR DeptCode = 6 OR DeptCode = 8) THEN
    SET InvalidCode TO TRUE
END-IF.
```

Or, using implied subjects:

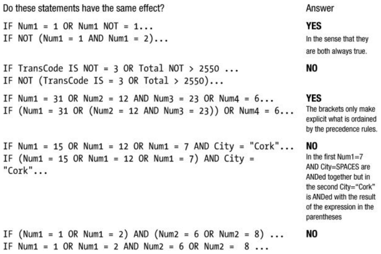
```
IF NOT (DeptCode = 1 OR 6 OR 8) THEN
    SET InvalidCode TO TRUE
END-IF.
```

3. Assume the variable `DeptCode` in question 2 is described as

```
01 DeptCode           PIC 9.
   88 InvalidCode     VALUE 0, 2 THRU 5,7,9.
```

Write a level 88 condition name called `InvalidCode` that is automatically set to true when the statement `ACCEPT DeptCode` accepts any value other than 1, 6, or 8.

4. In each of the following five groups of skeleton `IF` statements, state whether the statements in each group have the same effect (in the sense that they evaluate to true or false). Answer yes or no.



Do these statements have the same effect?	Answer

`IF Num1 = 1 OR Num1 NOT = 1...`
`IF NOT (Num1 = 1 AND Num1 = 2)...`
	YES
	In the sense that they are both always true.

`IF TransCode IS NOT = 3 OR Total NOT > 2550 ...`
`IF NOT (TransCode IS = 3 OR Total > 2550)...`
	NO

`IF Num1 = 31 OR Num2 = 12 AND Num3 = 23 OR Num4 = 6...`
`IF (Num1 = 31 OR (Num2 = 12 AND Num3 = 23)) OR Num4 = 6...`
	YES
	The brackets only make explicit what is ordained by the precedence rules.

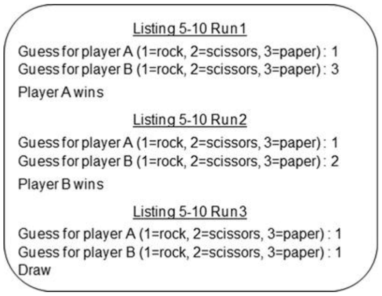
`IF Num1 = 15 OR Num1 = 12 OR Num1 = 7 AND City = "Cork"...`
`IF (Num1 = 15 OR Num1 = 12 OR Num1 = 7) AND City = "Cork"...`
	NO
	In the first Num1=7 AND City=SPACES are ANDed together but in the second City="Cork" is ANDed with the result of the expression in the parentheses

`IF (Num1 = 1 OR Num1 = 2) AND (Num2 = 6 OR Num2 = 8) ...`
`IF Num1 = 1 OR Num1 = 2 AND Num2 = 6 OR Num2 =  8 ...`
	NO

5. Write an `EVALUATE` statement to implement the decision part of a game of rock, paper, scissors. Most of the program has been written for you. Just complete the `EVALUATE`. `ADD` a `WHEN OTHER` branch to the `EVALUATE` to detect when a player enters a code other than 1, 2, or 3.

Listing 5-10. Rock, Paper, Scissors Game

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Listing5-10.
AUTHOR. Michael Coughlan.
DATA DIVISION.
WORKING-STORAGE SECTION.
   01 PlayerGuess-A    PIC 9  VALUE 1.
   88 Rock-A        VALUE 1.
   88 Paper-A       VALUE 2.
   88 Scissors-A    VALUE 3.

01 PlayerGuess-B    PIC 9  VALUE 2.
   88 Rock-B        VALUE 1.
   88 Paper-B       VALUE 2.
   88 Scissors-B    VALUE 3.

PROCEDURE DIVISION.
BEGIN.
   DISPLAY "Guess for player A (1=rock, 2=scissors, 3=paper)
           WITH NO ADVANCING
   ACCEPT PlayerGuess-A
   DISPLAY "Guess for player B (1=rock, 2=scissors, 3=paper)
           WITH NO ADVANCING
   ACCEPT PlayerGuess-B
   EVALUATE  TRUE      ALSO    TRUE
      WHEN Rock-A      ALSO    Rock-B      DISPLAY "Draw"
      WHEN Rock-A      ALSO    Paper-B     DISPLAY "Player B
      WHEN Rock-A      ALSO    Scissors-B  DISPLAY "Player A
      WHEN Paper-A     ALSO    Rock-B      DISPLAY "Player A
      WHEN Paper-A     ALSO    Paper-B     DISPLAY "Draw"
      WHEN Paper-A     ALSO    Scissors-B  DISPLAY "Player B
      WHEN Scissors-A  ALSO    Rock-B      DISPLAY "Player B
      WHEN Scissors-A  ALSO    Paper-B     DISPLAY "Player A
      WHEN Scissors-A  ALSO    Scissors-B  DISPLAY "Draw"
      WHEN OTHER   DISPLAY "Evaluate problem"
   END-EVALUATE
   STOP RUN.
```

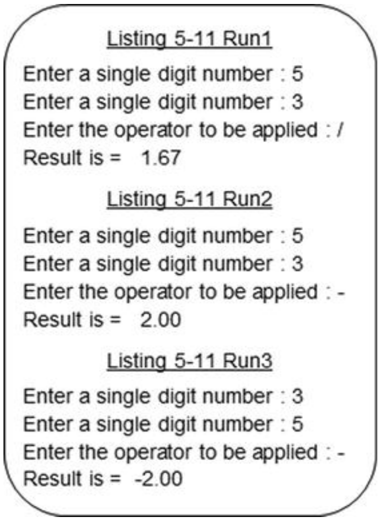


```
Listing 5-10 Run1
Guess for player A (1=rock, 2=scissors, 3=paper) : 1
Guess for player B (1=rock, 2=scissors, 3=paper) : 3
Player A wins
Listing 5-10 Run2
Guess for player A (1=rock, 2=scissors, 3=paper) : 1
Guess for player B (1=rock, 2=scissors, 3=paper) : 2
Player B wins
Listing 5-10 Run3
Guess for player A (1=rock, 2=scissors, 3=paper) : 1
Guess for player B (1=rock, 2=scissors, 3=paper) : 1
Draw
```

PROGRAMMING EXERCISE ANSWER

Listing 5-11. Simple Calculator

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  Listing5-11.
AUTHOR.  Michael Coughlan.
*> Accepts two numbers and an operator from the use
*> Applies the appropriate operation to the two num

DATA DIVISION.
WORKING-STORAGE SECTION.
01  Num1         PIC 9  VALUE 7.
01  Num2         PIC 9  VALUE 3.
01  Result       PIC --9.99 VALUE ZEROS.
01  Operator     PIC X  VALUE "-".
    88 ValidOperator   VALUES "*", "+", "-", "/".


PROCEDURE DIVISION.
CalculateResult.
    DISPLAY "Enter a single digit number : " WITH N
    ACCEPT Num1
    DISPLAY "Enter a single digit number : " WITH N
    ACCEPT Num2
    DISPLAY "Enter the operator to be applied : " W
    ACCEPT Operator
    EVALUATE Operator
      WHEN "+"   ADD Num2 TO Num1 GIVING Result
      WHEN "-"   SUBTRACT Num2 FROM Num1 GIVING Resu
      WHEN "*"   MULTIPLY Num2 BY Num1 GIVING Resul
      WHEN "/"   DIVIDE Num1 BY Num2 GIVING Result Rounded
      WHEN OTHER DISPLAY "Invalid operator entered"
    END-EVALUATE
    IF ValidOperator
       DISPLAY "Result is = ", Result
    END-IF
    STOP RUN.
```



```
Listing 5-11 Run1
Enter a single digit number : 5
Enter a single digit number : 3
Enter the operator to be applied : /
Result is =   1.67
Listing 5-11 Run2
Enter a single digit number : 5
Enter a single digit number : 3
Enter the operator to be applied : -
Result is =   2.00
Listing 5-11 Run3
Enter a single digit number : 3
Enter a single digit number : 5
Enter the operator to be applied : -
Result is =  -2.00
```

Support / Sign Out

PREV
CHAPTER 4: Procedure Division Basics

NEXT
CHAPTER 6: Control Structures: Iteration