Beginning COBOL for Programmers

PREV
CHAPTER 5: Control Structures: Selection

NEXT
TER 7: Introduction to Sequential Files

CHAPTER 6

Control Structures: Iteration

The previous chapter dealt with COBOL's selection constructs: `IF` and `EVALUATE`. In this chapter, you examine the last of the classic structured programming constructs: iteration.

In almost every programming job, there is some task that needs to be done over and over again. The job of processing a file of records is an iteration of this task: get and process record. The job of getting the sum of a stream of numbers is an iteration of this task: get and add number. The job of searching through an array for a particular value is an iteration of this task: get next element and check element value. These jobs are accomplished using iteration constructs.

Other languages support a variety of iteration constructs, each designed to achieve different things. In Modula-2 and Pascal, `While..DO` and `Repeat..Until` implement pre-test and post-test iteration. The `for loop` is used for counting iteration. The many C-language derivatives use `while` and `do..while` for pre-test and post-test iteration, and again the `for loop` is used for counting iteration.

COBOL supports all these different kinds of iteration, but it has only one iteration construct: the `PERFORM` verb (see Table 6-1). Pre-test and post-test iteration are supported by `PERFORM WITH TEST BEFORE` and `PERFORM WITH TEST AFTER`. Counting iteration is supported by `PERFORM..VARYING`. COBOL even has variations that are not found in other languages. `PERFORM..VARYING`, for instance, can take more than one counter, and it has both pre-test and post-test variations. Whereas in most languages the loop target is an inline block of code, in COBOL it can be either an inline block or a named out-of-line block of code.

Table 6-1. *Iteration Constructs and Their COBOL Equivalents*

	C, C++, Java	Modula-2, Pascal	COBOL
Pre-test	`while {}`	`While..DO`	`PERFORM WITH TEST BEFORE UNTIL`
Post-test	`do {} while`	`Repeat..Until`	`PERFORM WITH TEST AFTER UNTIL`
Counting	`for`	`For..DO`	`PERFORM..VARYING..UNTIL`

Paragraphs Revisited

In the `PROCEDURE DIVISION`, a paragraph is a block of code to which you have given a name. A paragraph begins with the paragraph name (see Example 6-1) and ends when the next paragraph or section name is encountered or when the end of the program text is reached. The paragraph name must *always* be terminated with a period (full stop).

There may be any number of statements and sentences in a paragraph; but there must be at least one sentence, and the last statement in the

paragraph must be terminated with a period. In fact, as I mentioned in the previous chapter, there is a style of COBOL programming called the *minimum-period* style [1] - [2], which you should adopt. This style suggests that there should be only one period in the paragraph. It is particularly important to adhere to this style when coding inline loops, because a period has the effect of delimiting the scope of an inline `PERFORM`.

Example 6-1. Two Paragraphs: `ProcessRecord` Ends Where `ProcessOutput` Begins

```
ProcessRecord.
     DISPLAY StudentRecord
     READ StudentFile
        AT END MOVE HIGH-VALUES TO StudentRecord
     END-READ.

ProduceOutput.
     DISPLAY "Here is a message".
```

The PERFORM Verb

Unless it is instructed otherwise, a computer running a COBOL program processes the statements in sequence, starting at the first statement of the `PROCEDURE DIVISION` and working its way down through the program until the `STOP RUN`, or the end of the program text, is reached. The `PERFORM` verb is one way of altering the sequential flow of control in a COBOL program. The `PERFORM` verb can be used for two major purposes;

- To transfer control to a designated block of code
- To execute a block of code iteratively

Whereas the other formats of the `PERFORM` verb implement iteration of one sort or another, this first format is used to transfer control to an out-of-line block of code—that is, to execute an open subroutine. You have probably have come across the idea of a subroutine before. A *subroutine* is a block of code that is executed when invoked by name. Methods, procedures, and functions are subroutines. You may not have realized that there are two types of subroutine:

- Open subroutines
- Closed subroutines

If you have learned BASIC, you may be familiar with open subroutines. If you learned C, Modula-2, or Java, you are probably familiar with closed subroutines.

Open Subroutines

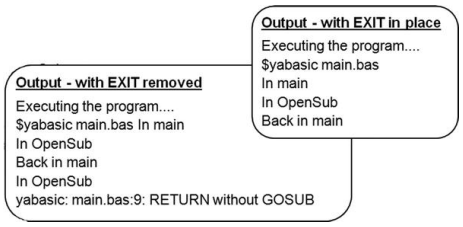
An *open* subroutine is a named block of code that control (by which I mean program statement execution) can fall into, or through. An open subroutine has access to all the data items declared in the main program, and it cannot declare any local data items.

Although an open subroutine is normally executed by invoking it by name, it is also possible, unless you are careful, to fall into it from the main program. In BASIC, the `GOSUB` and `RETURN` commands allow you to implement open subroutines. Example 6-2 is a short BASIC program that illustrates the fall-through problem. Two outputs are provided: one where the `EXIT` statement prevents fall-through and the other where control falls through into `OpenSub` because the `EXIT` statement has been removed.

Example 6-2. Open Subroutine in Yabasic [3] Showing Output With and Without the `EXIT` Statement

```
REM Demonstrates Open subroutines in Yabasic
REM When the EXIT is removed, control falls
REM through into OpenSub
REM Author. Michael Coughlan
PRINT "In main"
GOSUB OpenSub
PRINT "Back in main"
EXIT

LABEL OpenSub
  PRINT "In OpenSub"
  RETURN
```



In some legacy COBOL programs, falling through the program from paragraph to paragraph is a deliberate strategy. In this scheme, which has been called *gravity-driven programming*, control falls through the program until it encounters an `IF` and `GO TO` combination that drives it to a paragraph in the code above it; after that, control starts to fall through the program again. Example 6-3 provides an outline of how such a program works (`P1`, `P2`, `P3`, and `P4` are paragraph names).

Example 6-3. Model for a Gravity-Driven COBOL Program

```
P1.
  statement
  statement
  statement

P2.
  statement
  statement

P3.
  statement
  IF cond GO TO P2
  statement
  statement
  IF cond GO TO  P3

P4.
  statement
  IF cond GO TO P2
  statement
  statement
  STOP RUN
```

Closed Subroutines

A *closed* subroutine is a named block of code that can *only* be executed by invoking it by name. Control cannot "fall into" a closed subroutine. A closed subroutine can usually declare its own local data, and that data cannot be accessed outside the subroutine. Data in the main program can be passed to the subroutine by means of parameters specified when the subroutine is invoked. In C and Modula-2, procedures and functions implement closed subroutines. In Java, methods are used.

COBOL Subroutines

COBOL supports both open and closed subroutines. Open subroutines are implemented using the first format of the `PERFORM` verb. Closed subroutines are implemented using the `CALL` verb and contained or external subprograms. You learn about contained and external subprograms later in the book.

ISO 2002 ISO 2002 COBOL provides additional support for closed subroutines in the form of methods. Methods in COBOL bear a very strong syntactic resemblance to contained subprograms.

Why Use Open Subroutines?

The open subroutines represented by paragraphs (and sections) are used to make programs more readable and maintainable. Although `PERFORM`ed paragraphs are not as robust as the user-defined procedures or functions found in other languages, they are still useful. They allow you to partition code into a hierarchy of named tasks and subtasks without the formality or overhead involved in coding a procedure or function. COBOL programmers who require the protection of that kind of formal partitioning can use contained or external subprograms.

Partitioning a task into subtasks makes each subtask more manageable; and using meaningful names for the subtasks effectively allows you to document in code what the program is doing. For instance, a block of code

that prints report headings can be removed to a paragraph called `Print-ReportHeadings`. The details of *how* the task is being accomplished can be replaced with a name that indicates *what* is being done.

Consider the partitioning and documentation benefits provided by the program skeleton in Example 6-4. The skeleton contains no real code (only `PERFORM`s and paragraph names), but the hierarchy of named tasks and subtasks allows you to understand that the program reads through a file containing sales records for various shops and for each shop prints a line on the report that summarizes the sales for that shop.

Example 6-4. Program Skeleton

```
PrintSummarySalesReport.
   PERFORM PrintReportHeadings
   PERFORM PrintSummaryBody UNTIL EndOfFile
   PERFORM PrintFinalTotals
   STOP RUN.

PrintSummaryBody.
   PERFORM SummarizeShopSales
           UNTIL ShopId <> PreviousShopId
                 OR EndOfFile
   PERFORM PrintShopSummary

SummarizeShopSales.
   Statements

PrintReportHeadings.
   Statements

PrintShopSummary.
   Statements

PrintFinalTotals.
   Statements
```

Obviously, it is possible to take partitioning to an extreme. You should try to achieve a balance between making the program too fragmented and too monolithic. As a rule of thumb, there should be a good reason for creating a paragraph that contains five statements or fewer.

PERFORM NamedBlock

This first format of the `PERFORM` (see Figure 6-1) is not an iteration construct. It simply instructs the computer to transfer control to an out-of-line block of code (that is, an open subroutine). The block of code may be a paragraph or a section. When the end of the block is reached, control reverts to the statement (not the sentence) immediately following the `PERFORM`.

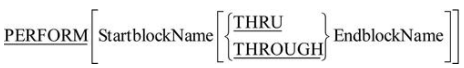


Figure 6-1. *Metalanguage for PERFORM format 1*

In Figure 6-1, `StartblockName` and `EndblockName` are the names of paragraphs or sections. `PERFORM..THRU` instructs the computer to treat the paragraphs or sections from `StartblockName TO EndblockName` as a single block of code.

`PERFORM`s can be nested. A `PERFORM` may execute a paragraph that contains another `PERFORM`, but neither direct nor indirect recursion is allowed. Unfortunately, this restriction is not enforced by the compiler, so a syntax error does not result; but your program will not work correctly if you use recursive `PERFORM`s.

The order of execution of the paragraphs is independent of their physical placement. It does not matter where you put the paragraphs—the `PERFORM` will find and execute them.

How PERFORM Works

Listing 6-1 shows a short COBOL program that demonstrates how `PERFORM` works. The program executes as follows:

1. Control starts in paragraph `LevelOne`, and the message "Starting to run program" is displayed.
2. When `PERFORM LevelTwo` is executed, control is passed to `LevelTwo` and the statements in that paragraph start to execute.

3. When `PERFORM LevelThree` is executed, control passes to `LevelThree`. When `PERFORM LevelFour` is executed, the message "Now in LevelFour" is displayed.
4. When the end of `LevelFour` is reached, *control returns to the statement following the* `PERFORM` *that invoked it*, and the message "Back in LevelThree" is displayed.
5. When `LevelThree` ends, control returns to the statement following the `PERFORM`, and the message "Back in LevelTwo" is displayed. Finally, when `LevelTwo` ends, control returns to paragraph `LevelOne`, and the "Back in LevelOne" message is displayed.
6. When `STOP RUN` is reached, the program stops.

Notice that the order of paragraph execution is independent of physical placement. For instance, although the paragraph `LevelTwo` comes after `LevelThree` and `LevelFour` in the program text, it is executed before them.

As I mentioned earlier, although `PERFORM`s can be nested, neither direct nor indirect recursion is allowed. So it would not be valid for paragraph `LevelThree` to contain the statement `PERFORM LevelThree`. This would be direct recursion. Neither would it be valid for LevelTwo to contain the statement `PERFORM LevelOne`. This would be indirect recursion because `LevelOne` contains the instruction `PERFORM LevelTwo`.

A frequent mistake made by beginning COBOL programmers is to forget to include `STOP RUN` at the end of the first paragraph. Example 6-5 shows the output that would be produced by Listing 6-1 if you forgot to include `STOP RUN`. From the output produced, try to follow the order of execution of the paragraphs.

Example 6-5. Output when `STOP RUN` is missing

```
>  Starting to run program
>  >  Now in LevelTwo
>  >  >  Now in LevelThree
>  >  >  >  Now in LevelFour
>  >  >  Back in LevelThree
>  >  Back in LevelTwo
>  Back in LevelOne
>  >  >  >  Now in LevelFour
>  >  >  Now in LevelThree
>  >  >  >  Now in LevelFour
>  >  >  Back in LevelThree
>  >  Now in LevelTwo
>  >  >  Now in LevelThree
>  >  >  >  Now in LevelFour
>  >  >  Back in LevelThree
>  >  Back in LevelTwo
```

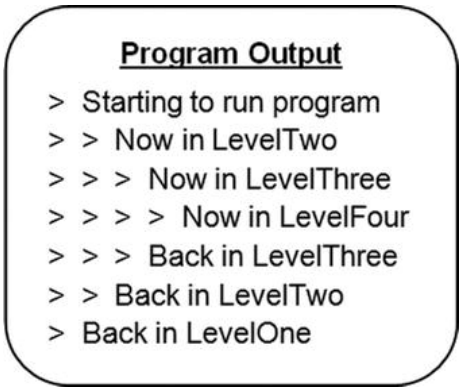
Listing 6-1. Demonstrates How `PERFORM` Works

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Listing6-1.
AUTHOR. Michael Coughlan.
PROCEDURE DIVISION.
LevelOne.
   DISPLAY "> Starting to run program"
   PERFORM LevelTwo
   DISPLAY "> Back in LevelOne"
   STOP RUN.

LevelFour.
   DISPLAY "> > > > Now in LevelFour".

LevelThree.
   DISPLAY "> > > Now in LevelThree"
   PERFORM LevelFour
   DISPLAY "> > > Back in LevelThree".

LevelTwo.
   DISPLAY "> > Now in LevelTwo"
   PERFORM LevelThree
   DISPLAY "> > Back in LevelTwo".
```



PERFORM..THRU Dangers

One variation that exists in all the `PERFORM` formats is `PERFORM..THRU`. When you use `PERFORM..THRU`, all the code from `StartblockName` to `EndblockName` is treated as a single block of code. Because `PERFORM..THRU` is generally regarded as a dangerous construct, it should only be used to `PERFORM` a paragraph and its immediately succeeding paragraph exit.

The problem with using `PERFORM..THRU` to execute a number of paragraphs as one unit is that, in the maintenance phase of the program's life, another programmer may need to create a new paragraph and may physically place it in the middle of the `PERFORM..THRU` block. Suddenly the program stops working correctly. Why? Because now `PERFORM..THRU` is executing an additional, unintentional, paragraph.

Using PERFORM..THRU Correctly

The warning against using `PERFORM..THRU` is not absolute, because when used correctly, `PERFORM..THRU` can be very useful. In COBOL there is no way to break out a paragraph that is the target of a `PERFORM`. All the statements have to be executed until the end of the paragraph is reached. But sometimes, such as when you encounter an error condition, you do not want to execute the remaining statements in the paragraph. This is a circumstance when `PERFORM..THRU` can be handy.

Consider the program outline in Example 6-6. In this example, control will not return to `Begin` until `SumEarnings` has ended, but you do not want to execute the remaining statements if an error is detected. The solution adopted is to hide the remaining statements behind an `IF NoErrorFound` statement. This might be an adequate solution if there were only one type of error; but if there is more than one type, then nested `IF` statements must be used. This quickly becomes unsightly and cumbersome.

Example 6-6. Using `IF`s to Skip Statements When an Error Is Detected

```
PROCEDURE DIVISION.
Begin.
   PERFORM SumEarnings
   STOP RUN.

SumEarnings.
   Statements
   Statements
   IF NoErrorFound
      Statements
      Statements
      IF NoErrorFound
         Statements
         Statements
         Statements
      END-IF
   END-IF.
```

In Example 6-7, `PERFORM..THRU` is used to deal with the problem in a more elegant manner. The dangers of `PERFORM..THRU` are ameliorated by having only two paragraphs in the target block and by using a name for the second paragraph that clearly indicates that it is bound to the first.

Example 6-7. Using `PERFORM..THRU` and `GO TO` to Skip Statements

```
PROCEDURE DIVISION
Begin.
   PERFORM SumEarnings THRU SumEarningsExit
   STOP RUN.

SumEarnings.
   Statements
   Statements
   IF ErrorFound
      GO TO SumEarningsExit
   END-IF
   Statements
   Statements

   IF ErrorFound
      GO TO SumEarningsExit
   END-IF
   Statements
   Statements
   Statements

SumEarningsExit.
   EXIT.
```

When the statement `PERFORM SumEarnings THRU SumEarningsExit` is executed, both paragraphs are performed as if they are one paragraph. The `GO TO` jumps to the exit paragraph, which, because the paragraphs are treated as one, is the end of the block of code. This technique allows you to skip over the code that should not be executed when an error is detected.

The `EXIT` statement in `SumEarningsExit` is a dummy statement. It has absolutely no effect on the flow of control. It is in the paragraph merely to conform to the rule that every paragraph must have one sentence. It has the status of a comment.

The `PERFORM..THRU` and `GO TO` constructs used in this example are dangerous. `GO TO` in particular is responsible for the "spaghetti code" that plagues many COBOL legacy systems. For this reason, you should use `PERFORM..THRU` and `GO TO` only as demonstrated in Example 6-7.

PERFORM..TIMES

`PERFORM..TIMES` (see Figure 6-2) is the second format of the `PERFORM` verb.

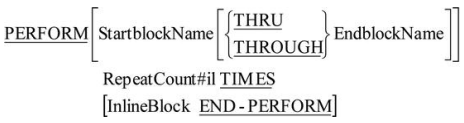


Figure 6-2. *Metalanguage for PERFORM format 2*

This format has no real equivalent in most programming languages, perhaps because of its limited usefulness. It simply allows a block of code to be executed `RepeatCount#il` times before returning control to the statement following `PERFORM`.

Like the other formats of `PERFORM`, this format allows two types of execution:

- Out-of-line execution of a block of code
- Inline execution of a block of code

Example 6-8 gives some example `PERFORM..TIMES` statements. These examples specify the `RepeatCount` using both literals and identifiers and show the inline and out-of-line variants of `PERFORM`.

Example 6-8. Using `PERFORM..TIMES`

```
PERFORM PrintBlankLine 10 Times

MOVE 10 TO RepetitionCount
PERFORM DisplayName RepetitionCount TIMES

PERFORM 15 TIMES
  DISPLAY "Am I repeating myself?"
END-PERFORM
```

Inline Execution

Inline execution will be familiar to programmers who have used the iteration constructs (`while`, `do/repeat`, `for`) of most other programming languages. An inline `PERFORM` iteratively executes a block of code contained within the same paragraph as the `PERFORM`. That is, the loop body is inline with the rest of the paragraph code. The block of code to be executed starts at the keyword `PERFORM` and ends at the keyword `END-PERFORM` (see Listing 6-2).

Listing 6-2. Demonstrates `PERFORM..TIMES` and Inline vs. Out-of-Line Execution

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Listing6-2.
AUTHOR. Michael Coughlan.
*> in-line and out-of-line PERFORM..TIMES

DATA DIVISION.
WORKING-STORAGE SECTION.
01 NumOfTimes PIC 9 VALUE 5.

PROCEDURE DIVISION.
Begin.
   DISPLAY "About to start in-line Perform"
   PERFORM 4 TIMES
      DISPLAY "> > > > In-line Perform"
   END-PERFORM
   DISPLAY "End of in-line Perform"

   DISPLAY "About to start out-of-line Perform"
   PERFORM OutOfLineCode NumOfTimes TIMES
   DISPLAY "End of out-of-line Perform"
   STOP RUN.

OutOfLineCode.
   DISPLAY "> > > > > Out-of-line Perform".
```

Listing 6-2 Output

```
About to start in-line Perform
> > > > In-line Perform
> > > > In-line Perform
> > > > In-line Perform
> > > > In-line Perform
End of in-line Perform
About to start out-of-line Perform
> > > > > Out-of-line Perform
> > > > > Out-of-line Perform
> > > > > Out-of-line Perform
> > > > > Out-of-line Perform
> > > > > Out-of-line Perform
End of out-of-line Perform
```

ANS 85 In-line `PERFORM`s were only introduced as part of the ANS 85 COBOL specification. In older legacy systems, the loop body is always out of line.

Out-of-Line Execution

In an out-of-line `PERFORM`, the loop body is a separate paragraph or section. This is the equivalent, in other languages, of having a procedure, function, or method invocation inside the loop body of a `while` or `for` construct.

When a loop is required, but only a few statements are involved, you should use an inline `PERFORM`. When a loop is required, and the loop body executes some specific task or function, out-of-line code should be used. The paragraph name chosen for the out-of-line code should identify the task or function of the code.

PERFORM..UNTIL

`PERFORM..UNTIL` (see Figure 6-3) is the third format of the `PERFORM` verb. This format implements both pre-test and post-test iteration in COBOL. It is the equivalent of Java's `while` and `do..while` or Pascal's `While` and `Repeat..Until` looping constructs.

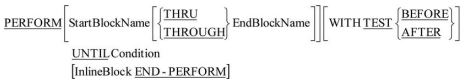


Figure 6-3. *Metalanguage for PERFORM format 3*

Pre-test and post-test iteration structures seem to be strangely implemented in many languages. Some languages confuse *when* the test is done with *how* the terminating condition is tested (Pascal's `While` and `Repeat` structures, for example). In many languages, the test for how the loop terminates emphasizes what makes the loop keep going, rather than what makes it stop. Although this may make formal reasoning about the loop easier, it does not come across as an entirely natural way of framing the question. In your day-to-day life, you do not say, "Heat the water while the water is not boiled" or "Pour water into the cup while the cup is not full."

Pre-test and post-test looping constructs are one area where COBOL seems to have things right. Whether the loop is pre-test or post-test, it is separated from *how* the terminating condition is tested; and the test for termination emphasizes what makes the loop stop, rather than what makes it keep going. In COBOL you might write

```
PERFORM ProcessSalesFile WITH TEST BEFORE
        UNTIL EndOfSalesFile
```

or

```
PERFORM GetNextCharacter WITH TEST AFTER
         UNTIL Letter = "s"
```

Notes on PERFORM..UNTIL

If you use the `WITH TEST BEFORE` phrase, `PERFORM` behaves like a `while` loop and the condition is tested before the loop body is entered. If you use the `WITH TEST AFTER` phrase, `PERFORM` behaves like a `do..while` loop and the condition is tested after the loop body is entered. The `WITH TEST BEFORE` phrase is the default and so is rarely explicitly stated.

How PERFORM..UNTIL Works

Although flowcharts are generally derided as a program-design tool, they are very useful for showing flow of control. The flowcharts in Figure 6-4 and Figure 6-5 show how the `WITH TEST BEFORE` and `WITH TEST AFTER` variations of `PERFORM..UNTIL` work.

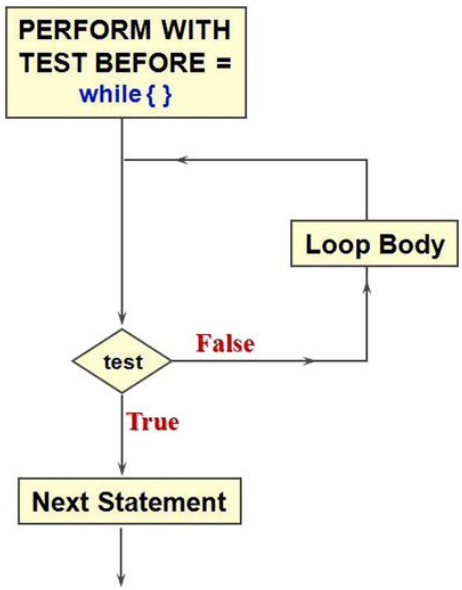


Figure 6-4. *Pre-test loop*

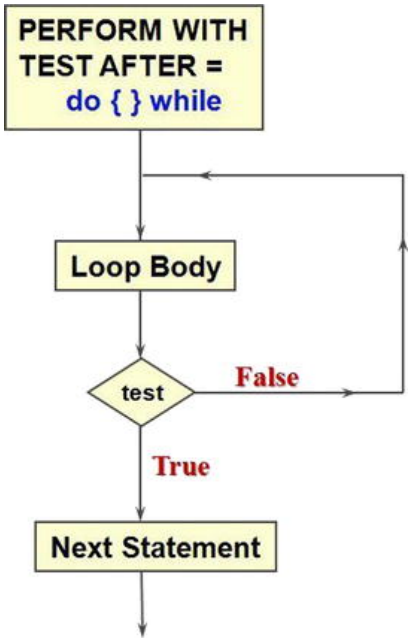


Figure 6-5. *Post-test loop*

Note that the terminating condition is checked only at the beginning of each iteration (`PERFORM WITH TEST BEFORE`) or at the end of each iteration (`PERFORM WITH TEST AFTER`). If the terminating condition is reached in the middle of the iteration, the rest of the loop body is still executed. The terminating condition cannot be checked until all the statements in the loop body have been executed. COBOL has no equivalent of the `break` command that allows control to break out of a loop without satisfying the terminating condition.

PERFORM..VARYING

`PERFORM..VARYING` (see Figure 6-6) is the final format of the `PERFORM` verb.

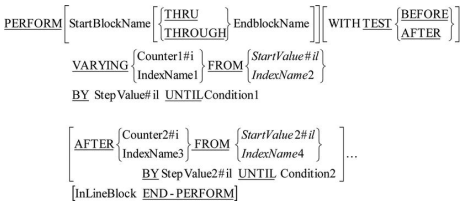


Figure 6-6. *Metalanguage for PERFORM format 4*

`PERFORM..VARYING` is used to implement counting iteration. It is similar to the `for` construct in languages like Pascal, C, and Java. However, there are some differences:

- Most languages permit only one counting variable per loop instruction. COBOL allows up to three. Why only three? Before ANS 85 COBOL, tables were allowed only a maximum of three dimensions, and `PERFORM..VARYING` was used to process them.
- Both pre-test and post-test variations of counting iteration are supported.
- The terminating condition does not have to involve the counting variable. For instance:

```
PERFORM CountRecordsInFile
        VARYING RecordCount FROM 1 BY 1 UNTIL EndOfFile
```

Notes on PERFORM..VARYING

The inline version of `PERFORM..VARYING` cannot take the `AFTER` phrase. This means only one counter may be used with an inline `PERFORM`.

When you use more than one counter, the counter after the `VARYING` phrase is the most significant, that after the first `AFTER` phrase is the next most significant, and the last counter is the least significant. Just like the values in an odometer, the least-significant counter must go through all its values and reach its terminating condition before the next-most-significant counter can be incremented.

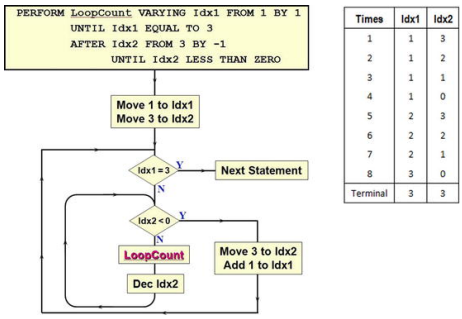
The item after the word `FROM` is the starting value of the counter (initialization). An index item is a special data item. Index items are examined when tables are discussed.

The item after the word `BY` is the step value of the counter (increment). It can be negative or positive. If you use a negative step value, the counter should be signed (`PIC S99`, for instance). When the iteration ends, the counters retain their terminating values.

The `WITH TEST BEFORE` phrase is the default and so is rarely specified.

How PERFORM..VARYING Works

Figure 6-7 shows the flowchart for `PERFORM..VARYING..AFTER`. Because there is no `WITH TEST` phrase, `WITH TEST BEFORE` is assumed. The table shows the number of times the loop body is processed and the value of each counter as displayed in the loop body. The terminating values of the counters are also given.



Times	Idx1	Idx2
1	1	3
2	1	2
3	1	1
4	1	0
5	2	3
6	2	2
7	2	1
8	3	0
Terminal	3	3

Figure 6-7. *PERFORM..VARYING..AFTER*

Note how the counter `Idx2` must go through all its values and reach its terminating value before the `Idx1` counter is incremented. An easy way to understand this is to think of it as an odometer. In an odometer, the units counter must go through all its values 0–9 before the tens counter is incremented.

Many of the example programs in this book provide a gentle preview of language elements to come. Listing 6-3 previews edited pictures. Examine the description of `PrnRepCount` provided by its picture, and review the output produced. Can you figure out how the edited picture works? Why do you think it was necessary to move `RepCount` to `PrnRepCount`? Why not just use the edited picture with `RepCount`?

Listing 6-3. Using `PERFORM..VARYING` for Counting

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Listing6-3.
AUTHOR. Michael Coughlan.

DATA DIVISION.
WORKING-STORAGE SECTION.
01 RepCount          PIC 9(4).
01 PrnRepCount       PIC Z,ZZ9.
01 NumberOfTimes     PIC 9(4) VALUE 1000.

PROCEDURE DIVISION.
Begin.
   PERFORM VARYING RepCount FROM 0 BY 50
           UNTIL RepCount = NumberOfTimes
      MOVE RepCount TO PrnRepCount
      DISPLAY "counting " PrnRepCount
   END-PERFORM
   MOVE RepCount TO PrnRepCount
   DISPLAY "If I have told you once, "
   DISPLAY "I've told you " PrnRepCount " times."
   STOP RUN.
```

Output from Listing 6-3

```
counting    0
counting   50
counting  100
counting  150
counting  200
counting  250
counting  300
counting  350
counting  400
counting  450
counting  500
counting  550
counting  600
counting  650
counting  700
counting  750
counting  800
counting  850
counting  900
counting  950
If I have told you once,
I have told you 1,000 times.
```

Answer `RepCount` can't be an edited picture because an edited picture contains non-numeric characters (spaces, in this case), and you can't do computations with non-numeric characters. You have to do the computations with the numeric `RepCount` and then move it to the edited field `PrnRepCount` when you want it printed.

The explanation of the operation of `PERFORM..VARYING..AFTER` compares the construct to an odometer. The program in Listing 6-4 reinforces this idea by using `PERFORM..VARYING` to emulate an odometer. The program uses both out-of-line and inline versions of `PERFORM..VARYING`. Notice that when the inline variation is used, you cannot have an `AFTER` phrase but must instead use nested `PERFORM`s just as in Java or Pascal. Because the output is voluminous, only the final part is shown here.

Listing 6-4. Odometer Simulation

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Listing6-4.
AUTHOR. Michael Coughlan.

DATA DIVISION.
WORKING-STORAGE SECTION.
01 Counters.
   02 HundredsCount   PIC 99 VALUE ZEROS.
   02 TensCount       PIC 99 VALUE ZEROS.
   02 UnitsCount      PIC 99 VALUE ZEROS.

01 Odometer.
   02 PrnHundreds     PIC 9.
   02 FILLER          PIC X VALUE "-".
   02 PrnTens         PIC 9.
   02 FILLER          PIC X VALUE "-".
   02 PrnUnits        PIC 9.

PROCEDURE DIVISION.
Begin.
    DISPLAY "Using an out-of-line Perform".
    PERFORM CountMileage
            VARYING HundredsCount FROM 0 BY 1 UNTIL HundredsCount
            AFTER TensCount FROM 0 BY 1 UNTIL TensCount > 9
            AFTER UnitsCount FROM 0 BY 1 UNTIL UnitsCount > 9

    DISPLAY "Now using in-line Perform"
    PERFORM VARYING HundredsCount FROM 0 BY 1 UNTIL HundredsCount
        PERFORM VARYING TensCount FROM 0 BY 1 UNTIL TensCount > 9
            PERFORM VARYING UnitsCount FROM 0 BY 1 UNTIL UnitsCou
                MOVE HundredsCount TO PrnHundreds
                MOVE TensCount TO PrnTens
                MOVE UnitsCount TO PrnUnits
                DISPLAY "In - " Odometer
            END-PERFORM
        END-PERFORM
    END-PERFORM
    DISPLAY "End of odometer simulation."
    STOP RUN.

CountMileage.
   MOVE HundredsCount TO PrnHundreds
   MOVE TensCount     TO PrnTens
   MOVE UnitsCount    TO PrnUnits
   DISPLAY "Out - " Odometer.
```

Listing 6-4 Partial Output

```
In - 9-8-0
In - 9-8-1
In - 9-8-2
In - 9-8-3
In - 9-8-4
In - 9-8-5
In - 9-8-6
In - 9-8-7
In - 9-8-8
In - 9-8-9
In - 9-9-0
In - 9-9-1
In - 9-9-2
In - 9-9-3
In - 9-9-4
In - 9-9-5
In - 9-9-6
In - 9-9-7
In - 9-9-8
In - 9-9-9
End of odometer simulation.
```

You might be wondering why the word `FILLER` is used in the description of `Odometer`. In COBOL, instead of having to make up dummy names, you can use `FILLER` when you need to reserve an area of storage but are never going to refer to it by name. For instance, in the data item `Odometer`, you want to separate the digits with hyphens, so you declare a character of storage for each hyphen and assign it the value `-`. But you will never refer to this part of `Odometer` by name. The hyphens only have significance as part of the group item.

Summary

This chapter examined the iteration constructs supported by COBOL. You learned the differences between COBOL's version of pre-test and post-test iteration and those of other languages. I contrasted counting iteration in its `PERFORM..VARYING..AFTER` implementation, which has both pre-test and post-test variations, with the offerings of other languages. You also explored the ability to create open subroutines in COBOL, and I provided a rationale for using them.

LANGUAGE KNOWLEDGE EXERCISE

Unleash your 2B pencil. It is exercise time again.

In the columns provided, write out what you would expect to be displayed on the computer screen if you ran the program shown in Listing 6-5. Use the Continue Run column to show what happens after the statement `DISPLAY "STOP RUN should be here".` has been executed.

Listing 6-5. Program to Test Your Knowledge of the `PERFORM` Verb

```
DATA DIVISION.                                          S
IDENTIFICATION DIVISION.
PROGRAM-ID. Listing6-5.
AUTHOR. Michael Coughlan.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 LoopCount PIC 9 VALUE 1.
01 LoopCount2 PIC 9 VALUE 1.
```

```
PROCEDURE DIVISION.                                    S
P1.
DISPLAY "S-P1"
PERFORM P2
PERFORM P3
MOVE 7 TO LoopCount
PERFORM VARYING LoopCount
FROM 1 BY 1 UNTIL LoopCount = 2
DISPLAY "InLine - " LoopCount
END-PERFORM
DISPLAY "E-P1".
DISPLAY "STOP RUN should be here".
```

```
P2.
DISPLAY "S-P2"
PERFORM P5 WITH TEST BEFORE VARYING LoopCount
FROM 1 BY 1 UNTIL LoopCount > 2
DISPLAY "E-P2".
```

```
P3.
DISPLAY "S-P3"
PERFORM P5
PERFORM P6 3 TIMES
DISPLAY "E-P3".
```

```
P4.
DISPLAY "P4-" LoopCount2
ADD 1 TO LoopCount2.
```

```
P5.
DISPLAY "S-P5"
DISPLAY LoopCount "-P5-" LoopCount2
PERFORM P4 WITH TEST AFTER UNTIL LoopCount2 > 2
DISPLAY "E-P5".
```

```
P6.
DISPLAY "P6".
```

PROGRAMMING EXERCISE 1

In this programming exercise, you amend the program you wrote for the programming exercise in Chapter 5 (or amend the answer provided in Listing 5-11). That programming exercise required you to create a calculator program, but the program halted after only one calculation.

Amend the program so it runs until the user enters the letter *s* instead of an operator (+ - / *). The result of running the program is shown in the sample output in Example 6-9.

Example 6-9. Sample Run (User Input Shown in Bold)

```
Enter an arithmetic operator (+ - * /) (s to end) :
Enter a single digit number -4
Enter a single digit number -5
Result is = 20.00
Enter an arithmetic operator (+ - * /) (s to end) :
Enter a single digit number -3
Enter a single digit number -3
Result is = 06.00
Enter an arithmetic operator (+ - * /) (s to end) :
Enter a single digit number -5
Enter a single digit number -3
Result is = -02.00
Enter an arithmetic operator (+ - * /) (s to end) :
Enter a single digit number -5
Enter a single digit number -3
```

```
Result is = 00.60
Enter an arithmetic operator (+ - * /) (s to end) ::
End of calculations
```

PROGRAMMING EXERCISE 2

Write a program that gets the user's name and a countdown value from the keyboard and then displays a countdown before displaying the name that was entered. Use `PERFORM..VARYING` to create the countdown.

The program should produce results similar to those shown in Example 6-10. For purposes of illustration, user input is in bold.

Example 6-10. Sample Run

```
Enter your name :-Mike Ryan
Enter the count-down start value :-05
Getting ready to display your name.
05
04
03
02
01
Your name is Mike Ryan
```

LANGUAGE KNOWLEDGE EXERCISE—ANSWER

```
DATA DIVISION.
IDENTIFICATION DIVI-
SION.
PROGRAM-ID. List-
ing6-5.
AUTHOR. Michael
Coughlan.
DATA DIVISION.
WORKING-STORAGE SEC-
TION.
01 LoopCount  PIC 9
VALUE 1.
01 LoopCount2 PIC 9
VALUE 1.

PROCEDURE DIVISION.
P1.
DISPLAY "S-P1"
PERFORM P2
PERFORM P3
MOVE 7 TO LoopCount
PERFORM VARYING
LoopCount
FROM 1 BY 1 UNTIL
LoopCount = 2
DISPLAY "InLine - "
LoopCount
END-PERFORM
DISPLAY "E-P1".
DISPLAY "STOP RUN
should be here".

P2.
DISPLAY "S-P2"
PERFORM P5 WITH TEST
BEFORE VARYING Loop-
Count
FROM 1 BY 1 UNTIL
LoopCount > 2
DISPLAY "E-P2".
```

Start Run	Continue Run
S-P1	
S-P2	
S-P5	S-P2
1-P5-1	S-P5
P4-1	1-P5-5
P4-2	P4-5
E-P5	E-P5
S-P5	S-P5
2-P5-3	2-P5-6
P4-3	P4-6
E-P5	E-P5
E-P2	S-P3
S-P3	S-P5
S-P5	3-P5-7
3-P5-4	P4-7
P4-4	E-P5
E-P5	P6
P6	P6
P6	P6
E-P3	E-P3
InLine - 1	P4-8
E-P1	S-P5
STOP RUN should be here	3-P5-9
	P4-9
	E-P5
	P6

```
P3.
DISPLAY "S-P3"
PERFORM P5
PERFORM P6 3 TIMES
DISPLAY "E-P3".

P4.
DISPLAY "P4-" Loop-
Count2
ADD 1 TO LoopCount2.

P5.
DISPLAY "S-P5"
DISPLAY LoopCount "-
P5-" LoopCount2
PERFORM P4 WITH TEST
AFTER UNTIL Loop-
Count2 > 2
DISPLAY "E-P5".

P6.
DISPLAY "P6".
```

PROGRAMMING EXERCISE 1—ANSWER

Listing 6-6. The Full Calculator Program

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Listing6-6.
AUTHOR. Michael Coughlan.
*> Continually calculates using two numbers and an
*> Ends when "s" is entered instead of an operator.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 Num1       PIC 9      VALUE ZERO.
01 Num2       PIC 9      VALUE ZERO.
01 Result     PIC --9.99 VALUE ZEROS.
01 Operator   PIC X      VALUE SPACE.
   88 ValidOperator      VALUES "*", "+", "-", "/",
   88 EndOfCalculations  VALUE "s".

PROCEDURE DIVISION.
Begin.
   PERFORM GetValidOperator UNTIL ValidOperator
   PERFORM UNTIL EndOfCalculations OR NOT ValidOper
      PERFORM GetTwoNumbers
      EVALUATE Operator
        WHEN "+" ADD      Num2 TO   Num1 GIVING Res
        WHEN "-" SUBTRACT Num2 FROM Num1 GIVING Res
        WHEN "*" MULTIPLY Num1 BY   Num2 GIVING Res
        WHEN "/" DIVIDE   Num1 BY   Num2 GIVING Res
      END-EVALUATE
      DISPLAY "Result is = ", Result
      MOVE SPACE TO Operator
      PERFORM GetValidOperator UNTIL ValidOperator
   END-PERFORM
   DISPLAY "End of calculations"
   STOP RUN.

GetValidOperator.
   DISPLAY "Enter an arithmetic operator (+ - * /)
            WITH NO ADVANCING
   ACCEPT Operator.

GetTwoNumbers.
   DISPLAY "Enter a single digit number - " WITH NO
   ACCEPT Num1

   DISPLAY "Enter a single digit number - " WITH NO
   ACCEPT Num2.
```

PROGRAMMING EXERCISE 2—ANSWER

Listing 6-7. Uses `PERFORM..VARYING` to Display a Countdown from XX to 01

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Listing6-7.
AUTHOR. Michael Coughlan.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 UserName      PIC X(20).
01 StartValue    PIC 99 VALUE ZEROS.
01 Countdown     PIC 99 VALUE ZEROS.

PROCEDURE DIVISION.
DisplayCountdown.
   DISPLAY "Enter your name :- " WITH NO ADVANCING
   ACCEPT UserName

   DISPLAY "Enter the count-down start value :- " W
   ACCEPT StartValue

   PERFORM VARYING Countdown FROM StartValue BY -1 U
       DISPLAY Countdown
   END-PERFORM

   DISPLAY "Your name is " UserName
   STOP RUN.
```

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

3 . Compiled and run at compileoneline.com—Execute BASIC Program Online (Yabasic 2.9.15). www.compileonline.com/execute_basic_online.php (http://www.compileonline.com/execute_basic_online.php)