Basic COBOL

This chapter introduces the basics of COBOL terminology, coding rules and syntax, and then demonstrates how to view and run a basic COBOL program in Visual Studio Code. In later chapters, we will come back to some of these concepts in greater detail.

8.2.1 The Five COBOL Areas

COBOL source code is column dependent, meaning column rules are strictly enforces. Each line of COBOL source code is written in COBOL reference format. These reference formats have five areas, as shown in the Figure (1) below:

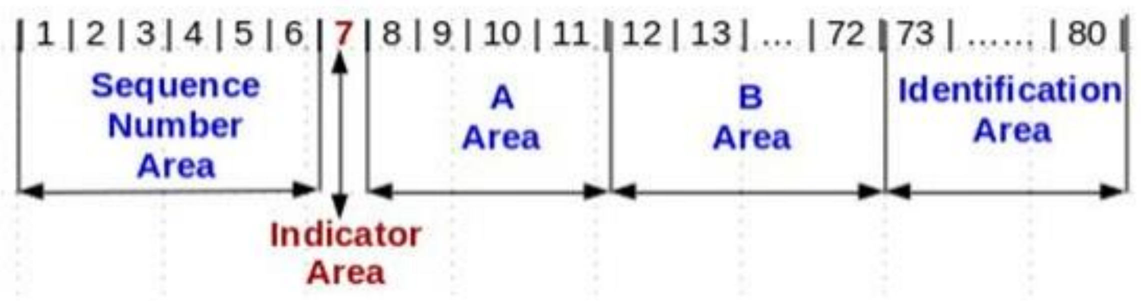


Figure (1) COBOL reference format

8.2.1.1 Sequence Number Area (columns 1-6)

- Blank or reserved for line sequence numbers

8.2.1.2 Indicator Area (Column 7)

- A multi-purpose area
    - Comment line (generally an asterisk symbol)
    - Continuation line (generally a hyphen symbol)
    - Debugging line (D or d)
    - Source listing formatting (a slash symbol)

8.2.1.3 Area A (columns 8-11)

- Certain items must begin in Area A:
    - Level indicators
    - Declarative
    - Division, Section, Paragraph headers
    - Paragraph names
- Column 8 is referred to as the A Margin

8.2.1.4 Area B (Columns 12-72)

- Certain items must begin in Area B:
    - Entries, sentences, statements, clauses
    - Continuation lines
- Column 12 is referred to as the B Margin

8.2.1.5 Identification Area (columns 73-80)

- Ignored by the compiler
- Can be blank or optionally used by the programmer for any purpose

Don't worry too much at this stage about understanding the different areas in COBOL - it will become more apparent as we go through the chapters.

COBOL Structure

COBOL is a hierarchy structure consisting of divisions, sections paragraphs, sentences and statements.

8.3.2 The Four COBOL Divisions

8.3.2.1 Identification Division

The IDENTIFCATION DIVISION identifies the program with a name. There is also other information that can be included in this division, such as the Author name, date last modified etc, but this is optional.

8.3.2.2 Environment Division

The ENVIRONMENT DIVISION describes the aspects of your program that depend on the computing environment, such as the computer configuration, or the computer inputs and outputs.

8.3.2.3 Data Division

The DATA DIVISION is where characteristics of data are defined in one of the following sections:

- FILE SECTION – defines data used in input-output operations.
- LINKAGE SECTION – describes data from another program.
- WORKING-STORAGE SECTION – storage allocated and remaining for the life of program.
- LOCAL-STORAGE SECTION – storage allocated each time a program is called and de-allocated when the program ends
- PROCEDURE DIVISON – contains instructions relate to the manipulation of data. Interfaces with other programs are also specified here.

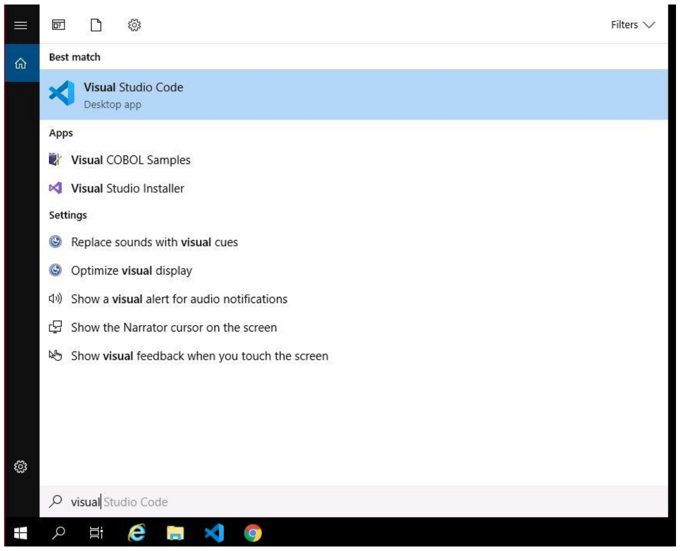
8.4 Procedure Division

The PROCEDURE DIVISON is where the work gets done in the program. The PROCEDURE DIVISON is divided into different elements: sections, paragraphs, sentences, phrases.

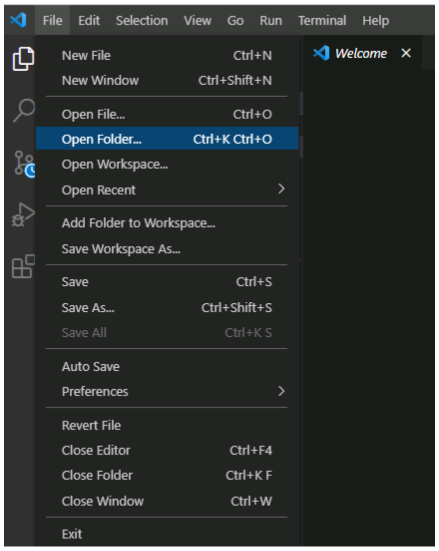
8.5 Lab

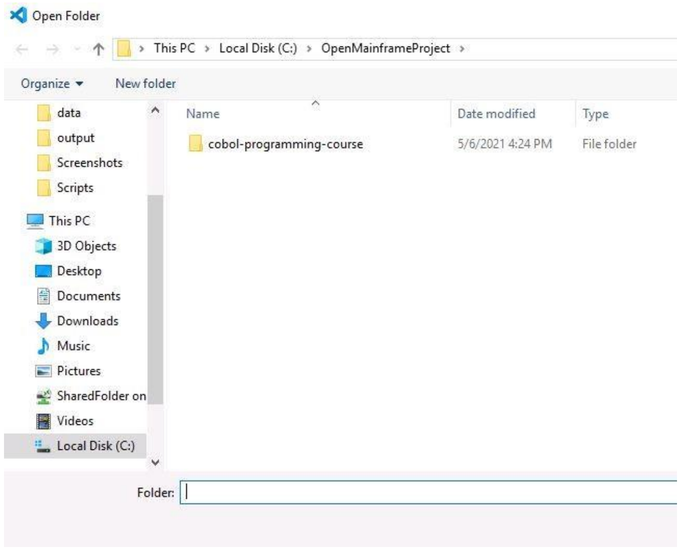
In this exercise you will learn how to compile and run a basic COBOL program in VS Code with the Micro Focus COBOL extension.

1. This lab assumes you have followed the VM installation covered in chapters 6 and 7, or you are using one of the provisioned AWS instances.
2. In your start menu search bar, search for Visual Studio Code and open it.


Filters
Best match
Visual Studio Code
Desktop app
Apps
Visual COBOL Samples
Visual Studio Installer
Settings
Replace sounds with visual cues
Optimize visual display
Show a visual alert for audio notifications
Show the Narrator cursor on the screen
Show visual feedback when you touch the screen
visual Studio Code

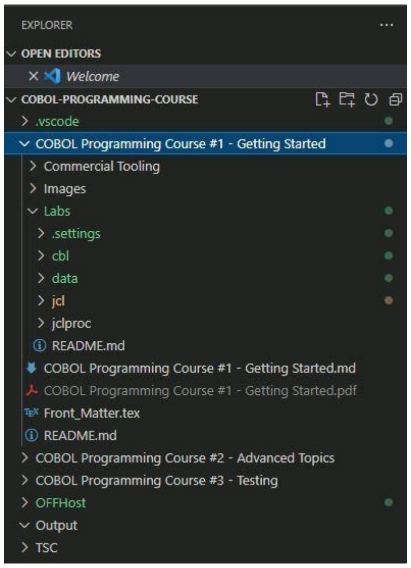
3. In VS Code, open the folder where you have saved this guide's Lab source code. This will open the folder in the VS code browse tab on the left-hand side. If you are using one of the provisioned AWS instances, this is usually found in the C: drive, select the folder called cobol-programming-course.

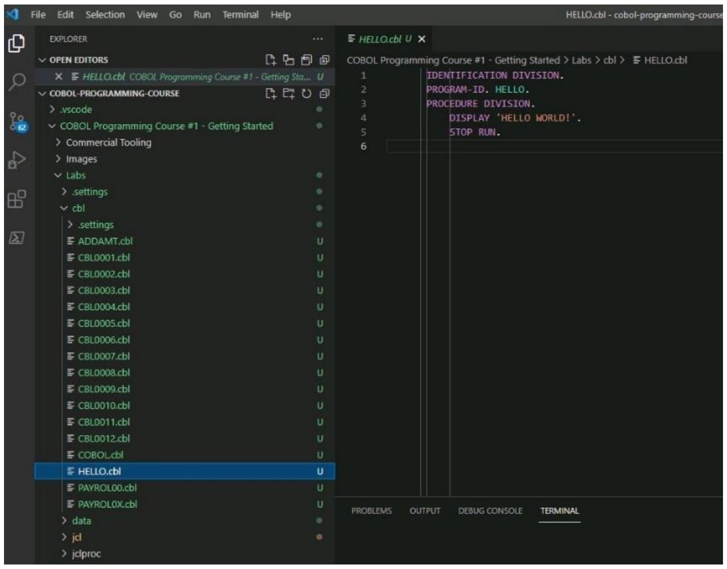




Name	Date modified	Type
cobol-programming-course	5/6/2021 4:24 PM	File folder

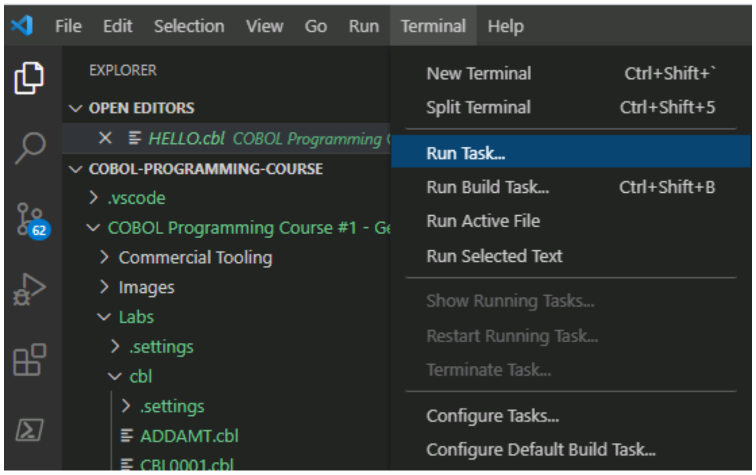
4. Notice you have several folders. One of these contains your COBOL and JCL files, which you will need to complete each exercise. Because of how the environment is provisioned, you will need to open each file you wish to compile or run in the editor, by double clicking on it. So open the file HELLO.cbl.


EXPLORER
OPEN EDITORS
Welcome
COBOL-PROGRAMMING-COURSE
.vscode
COBOL Programming Course #1 - Getting Started
Commercial Tooling
Images
Labs
.settings
cbl
data
jcl
jclproc
README.md
COBOL Programming Course #1 - Getting Started.md
COBOL Programming Course #1 - Getting Started.pdf
Front_Matter.tex
README.md
COBOL Programming Course #2 - Advanced Topics
COBOL Programming Course #3 - Testing
OFFHost
Output
TSC

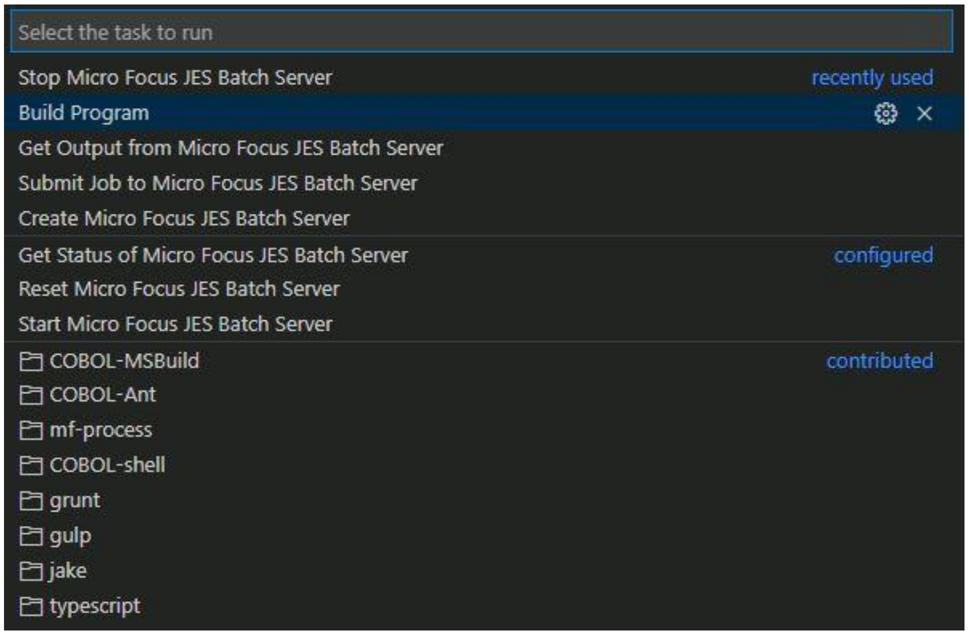


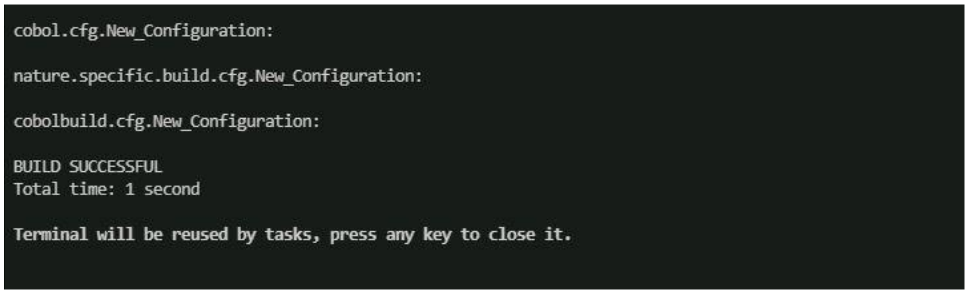
```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. HELLO.
       PROCEDURE DIVISION.
           DISPLAY 'HELLO WORLD!'.
           STOP RUN.
```

5. Now in the top menu, click on Terminal, then Run Task. The Terminal in VS Code is similar to a Command Prompt: it's a space where you type in commands that your operating system understands and executes and can be found at the bottom of the editor. In this case, the commands have been automated, so you only need to click on a few buttons to run the commands.



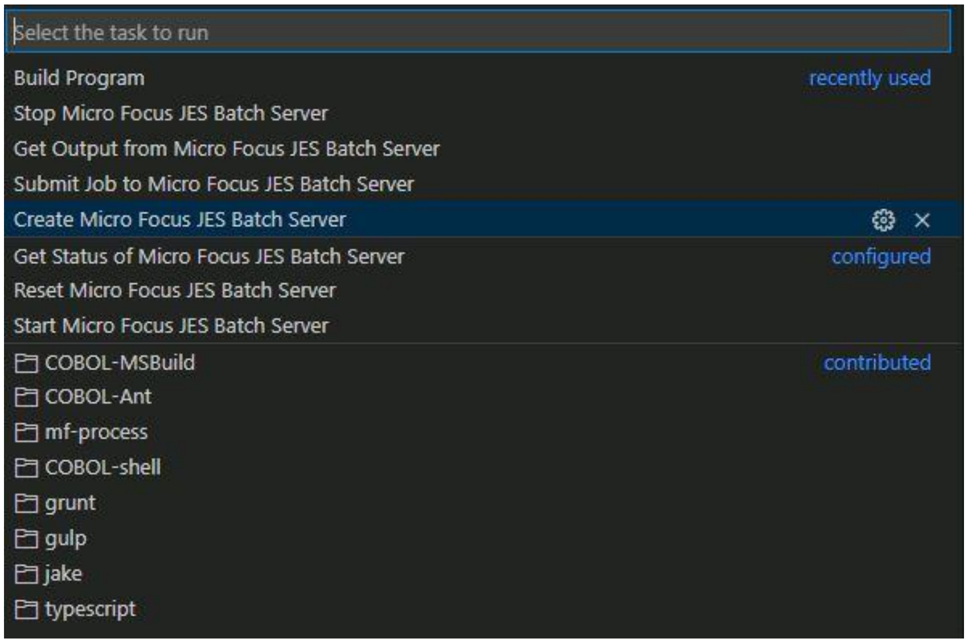
6. In the new top center menu, click on Build Program. Building or compiling a program will create the necessary binary files for the machine to run COBOL code. Notice the Terminal window at the bottom is now populated with everything that has been going on in the background, make sure at the very bottom you can read "BUILD SUCCESSFUL".


Select the task to run
Stop Micro Focus JES Batch Server
recently used
Build Program
Get Output from Micro Focus JES Batch Server
Submit Job to Micro Focus JES Batch Server
Create Micro Focus JES Batch Server
Get Status of Micro Focus JES Batch Server
configured
Reset Micro Focus JES Batch Server
Start Micro Focus JES Batch Server
COBOL-MSBuild
contributed
COBOL-Ant
mf-process
COBOL-shell
grunt
gulp
jake
typescript

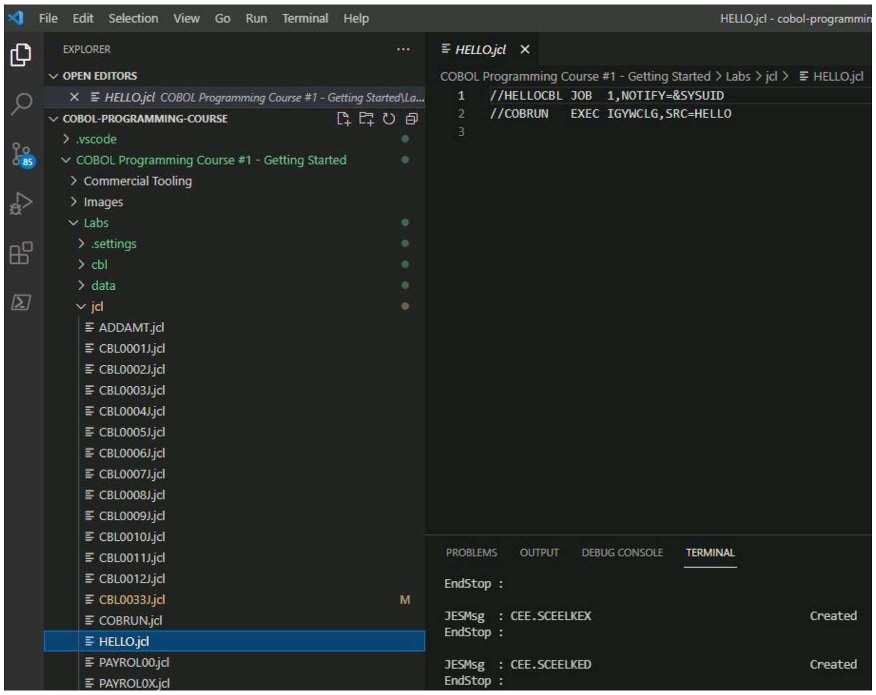


```
cobol.cfg.New_Configuration:

nature.specific.build.cfg.New_Configuration:

cobolbuild.cfg.New_Configuration:

BUILD SUCCESSFUL
Total time: 1 second

Terminal will be reused by tasks, press any key to close it.
```

7. Now that your program is ready to run, you'll need to provision the Enterprise Server on which it will run. So, click on Terminal, Run task, and then Create Micro Focus JES Batch Server: this is the region that emulates the functionality of JES on the mainframe, which means it can run batch jobs though JCL scripts. Batch jobs usually automate a series of operations utilizing OBOL programs without the need for human interaction. When you create a region, this will also be automatically started upon creation, so you don't need to do anything else for now.

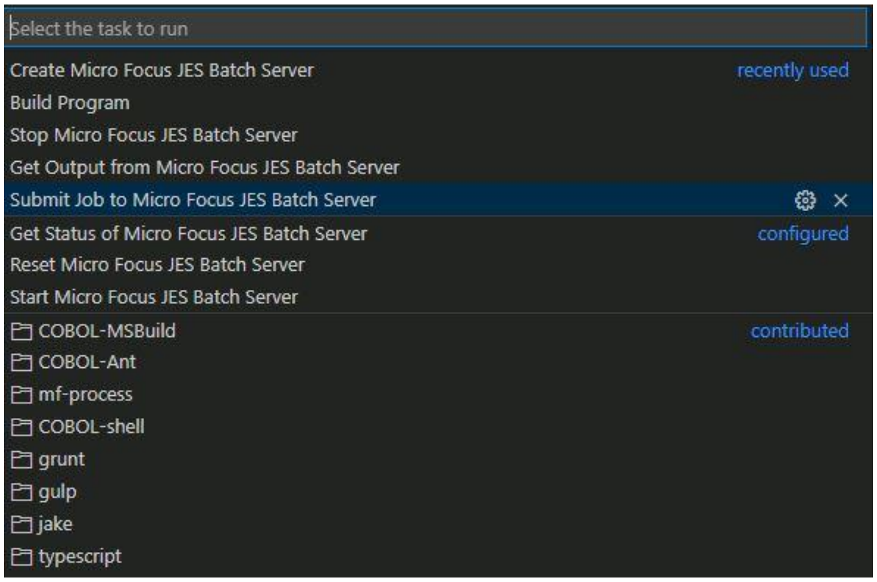

Select the task to run
Build Program
recently used
Stop Micro Focus JES Batch Server
Get Output from Micro Focus JES Batch Server
Submit Job to Micro Focus JES Batch Server
Create Micro Focus JES Batch Server
Get Status of Micro Focus JES Batch Server
configured
Reset Micro Focus JES Batch Server
Start Micro Focus JES Batch Server
COBOL-MSBuild
contributed
COBOL-Ant
mf-process
COBOL-shell
grunt
gulp
jake
typescript

8. In the JCL folder, open the file HELLO.jcl. To submit the job (or run it) click on Terminal, Run task, and then Submit Job to Micro Focus JES Batch Server. This will run the JCL script, which itself contains the necessary information to execute the program HELLO.cbl.

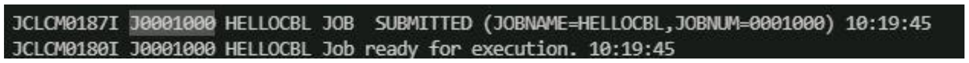


```
//HELLOCBL JOB  1,NOTIFY=&SYSUID
//COBRUN   EXEC IGYWCLG,SRC=HELLO
```

```
EndStop :

JESMsg  : CEE.SCEELKEX                              Created
EndStop :

JESMsg  : CEE.SCEELKED                              Created
EndStop :
```

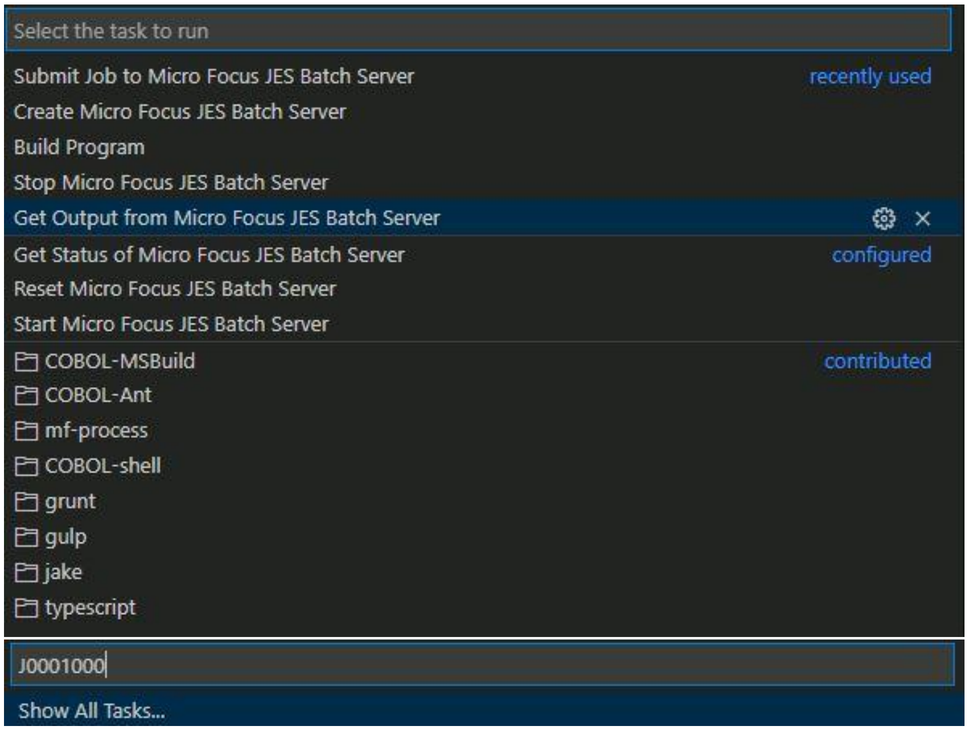

Select the task to run
Create Micro Focus JES Batch Server
recently used
Build Program
Stop Micro Focus JES Batch Server
Get Output from Micro Focus JES Batch Server
Submit Job to Micro Focus JES Batch Server
Get Status of Micro Focus JES Batch Server
configured
Reset Micro Focus JES Batch Server
Start Micro Focus JES Batch Server
COBOL-MSBuild
contributed
COBOL-Ant
mf-process
COBOL-shell
grunt
gulp
jake
typescript

9. To ensure the program run correctly, you will need to check the job output. To do this, you will need the job ID, which appears in the Terminal window after the job was successful, and for the first program you run is usually J0001000. Copy this from the Terminal window.

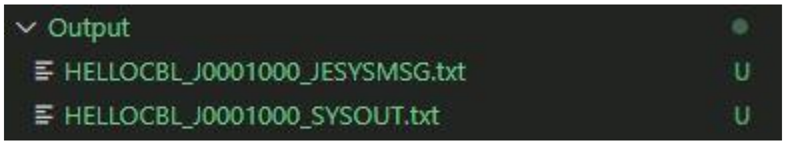


```
JCLCM0187I J0001000 HELLOCBL JOB  SUBMITTED (JOBNAME=HELLOCBL,JOBNUM=0001000) 10:19:45
JCLCM0180I J0001000 HELLOCBL Job ready for execution. 10:19:45
```

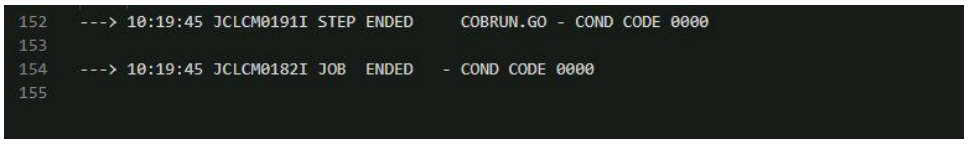
10. To retrieve the output, click on Terminal, Run Task, then Get Output from Micro Focus JES Batch Server, then copy the job ID in the bar that appears with the tag 'ID of Job to Retrieve Output for'.


Select the task to run
Submit Job to Micro Focus JES Batch Server
recently used
Create Micro Focus JES Batch Server
Build Program
Stop Micro Focus JES Batch Server
Get Output from Micro Focus JES Batch Server
Get Status of Micro Focus JES Batch Server
configured
Reset Micro Focus JES Batch Server
Start Micro Focus JES Batch Server
COBOL-MSBuild
contributed
COBOL-Ant
mf-process
COBOL-shell
grunt
gulp
jake
typescript
J0001000
Show All Tasks...

11. You can now find the job output in the Output folder within the main repository. The file name is usually HELLO_ J0001000_JESYSMSG.txt, double click on it to open it in the editor.

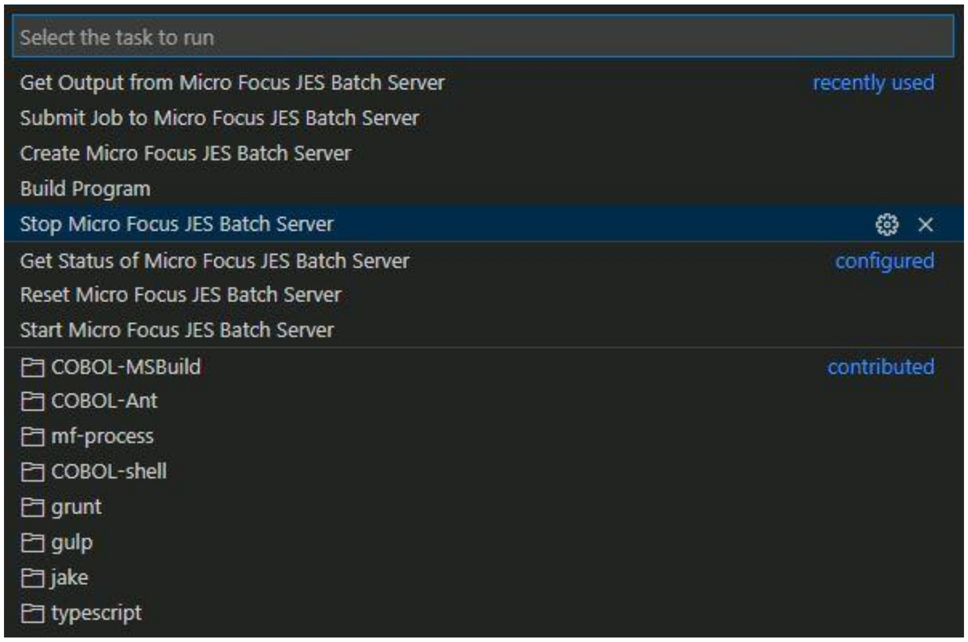

Output
HELLOCBL_J0001000_JESYSMSG.txt U
HELLOCBL_J0001000_SYSOUT.txt U

12. Scroll to the bottom of the file and notice the final conditional code of 0000. A conditional code is a number that the Micro Focus Enterprise Server returns to communicate what happened at each step of the job, and 0000 means the step was successful. In this case, the very last step of the job was successful, which means that the whole job ran successfully.



```
---> 10:19:45 JCLCM0191I STEP ENDED     COBRUN.GO - COND CODE 0000

---> 10:19:45 JCLCM0182I JOB  ENDED   - COND CODE 0000

```

13. Before turning off your machine, remember to stop the Micro Focus JES Batch Server: click on Terminal, Run Task, and then Stop the Micro Focus JES Batch Server. This is required each time you step away from a Lab, otherwise the region status will become ‘unresponsive’, and you will need to reset it.


Select the task to run
Get Output from Micro Focus JES Batch Server recently used
Submit Job to Micro Focus JES Batch Server
Create Micro Focus JES Batch Server
Build Program
Stop Micro Focus JES Batch Server
Get Status of Micro Focus JES Batch Server configured
Reset Micro Focus JES Batch Server
Start Micro Focus JES Batch Server
COBOL-MSBuild contributed
COBOL-Ant
mf-process
COBOL-shell
grunt
gulp
jake
typescript

Chapter 9 – Data Division

In this chapter you will be introduced to the basics of COBOL variables, also known as data-items, and exposed to a few more advanced options. At the end of the chapter is a lab available to compile and execute the COBOL source code provided.

9.1 Coding COBOL Variables/Data-Items

The COBOL reserved word, PICTURE (PIC), determines the length and data type of variable name. Coding a variable/data item is carried out in the DATA DIVISION and consists of:

- Level Numbers – the hierarchy of fields in a record
- Variable name/Data Item name – assigns a name to each field referenced in the program. Must be unique within the program
- Picture Clause – for data type checking

Let's look at these in more detail.

9.2 Variables/Data Items

A COBOL variable, or data-item, is a name used and chosen by the COBOL programmer. This is a piece of code that is designed to hold a data value which can vary, hence the generic term 'variable'.

9.2.1 Variable/Data Item name restrictions and data types

A COBOL variable name is also referred to as a 'data name' and is subject to certain restrictions:

- Must only contain letters (A-Z), digits (0-9), underscores (_) and hyphens (-)
- Maximum length of 30 characters
- Must not be a COBOL reserved word
- Must not contain a space ( ) as a part of the name
- A hyphen cannot appear as the first or last character
- An underscore cannot appear as the first character

A COBOL variable has certain attributes, such as length and data type. During program execution, the COBOL source code tells the compiler these attributes, so it knows how much memory it should be allocated and what data type it should expect.

The most common COBOL data types are:

- Numeric (0-9)
- Alphabetic (A-Z), (a-z) or a space ( )
- Alphanumeric Numeric and Alphabetic Combination

9.2.2 The PICTURE Clause

Data types described by PIC are commonly referred to as a picture clause or pic clause. Some simple pic clauses are:

- PIC 9 – single numeric value where the length is one
- PIC 9(4) – four numeric values where the length is four
- PIC X – single alphanumeric value where the length is one
- PIC X(4) = four alphanumeric values where the length is four

There are several other PIC clause symbols and data types – a full list is available Micro Focus COBOL Picture Clause Reference Manual

9.2.3 Picture Clause Character-string representation

Some PIC clause symbols can only appear once in a PIC clause character-string, while others can appear more than once. For example:

- PIC clause to hold value 1234.56 is coded as follows, where the V represents the decimal position:

PIC 9(4)V99

- PIC clause for a value such as $1234.56 is coded as follows:

  PIC $9,999V99

9.3 Literals

A COBOL literal is a constant data value, meaning the value will not change like a variable can. Let's look at the following COBOL statement:

DISPLAY "HELLO WORLD"

In this example, DISPLAY is a COBOL reserved word, followed by a literal, HELLO WORLD. To denote a literal, we put HELLO WORLD in quotation marks ("").

9.3.1 Figurative Constants

Figurative constants are reserved words that name and refer to specific constant values. Examples of figurative constants are:

- ZERO, ZEROS, ZEROES
- SPACE, SPACES
- HIGH-VALUE, HIGH-VALUES
- LOW-VALUE, LOW-VALUES
- QUOTE, QUOTES
- NULL, NULLS

9.3.2 Data Relationships

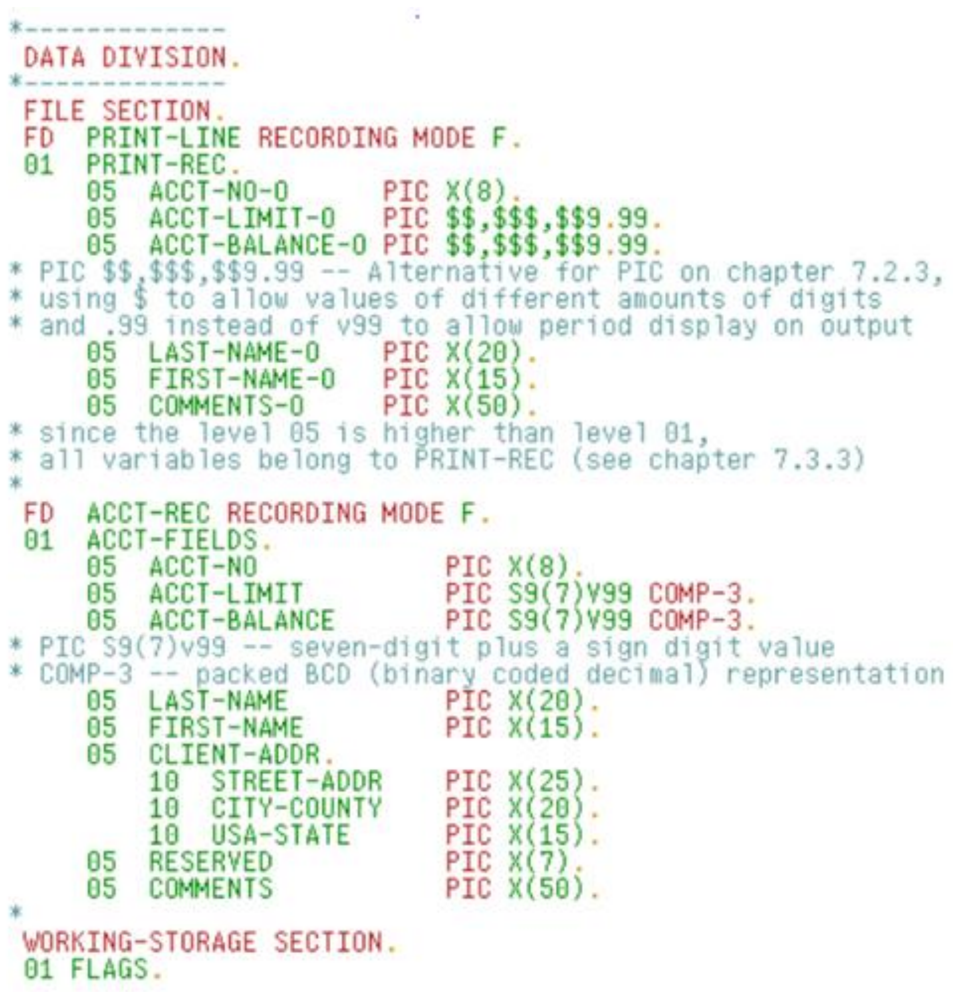
The relationships among all data to be used in a program is defined in the DATA DIVISION through a system of level indicators and level-numbers

9.3.3 Levels of data

After a record is defined, it can be subdivided to provide more detailed data references. A level number is a one-digit or two-digit integer between 01 and 49. There are also three predefined special level numbers:

- 66 – assigns an alternate name to a field or group
- 77 – used for an independent data item
- 88 – a conditional variable, always subordinate to another data item

Observe the following example:

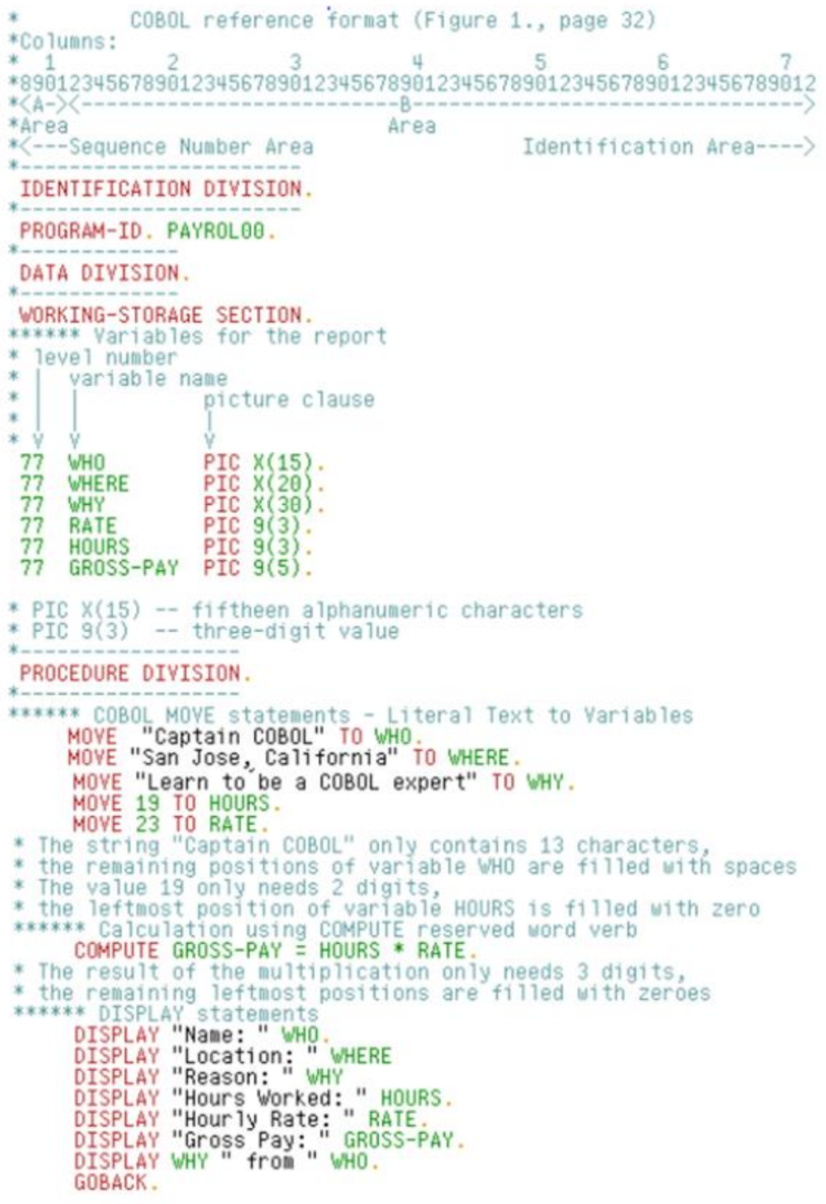


```
*-------------
 DATA DIVISION.
*-------------
 FILE SECTION.
 FD  PRINT-LINE RECORDING MODE F.
 01  PRINT-REC.
     05  ACCT-NO-O      PIC X(8).
     05  ACCT-LIMIT-O   PIC $$,$$$,$$9.99.
     05  ACCT-BALANCE-O PIC $$,$$$,$$9.99.
* PIC $$,$$$,$$9.99 -- Alternative for PIC on chapter 7.2.3,
* using $ to allow values of different amounts of digits
* and .99 instead of v99 to allow period display on output
     05  LAST-NAME-O    PIC X(20).
     05  FIRST-NAME-O   PIC X(15).
     05  COMMENTS-O     PIC X(50).
* since the level 05 is higher than level 01,
* all variables belong to PRINT-REC (see chapter 7.3.3)
*
 FD  ACCT-REC RECORDING MODE F.
 01  ACCT-FIELDS.
     05  ACCT-NO            PIC X(8).
     05  ACCT-LIMIT         PIC S9(7)V99 COMP-3.
     05  ACCT-BALANCE       PIC S9(7)V99 COMP-3.
* PIC S9(7)v99 -- seven-digit plus a sign digit value
* COMP-3 -- packed BCD (binary coded decimal) representation
     05  LAST-NAME          PIC X(20).
     05  FIRST-NAME         PIC X(15).
     05  CLIENT-ADDR.
         10  STREET-ADDR    PIC X(25).
         10  CITY-COUNTY    PIC X(20).
         10  USA-STATE      PIC X(15).
     05  RESERVED           PIC X(7).
     05  COMMENTS           PIC X(50).
*
 WORKING-STORAGE SECTION.
 01 FLAGS.
```

As you can see “01 ACCT-FIELDS” references the following “05”-level variables. Note how then “05 CLIENT-ADDR” is then further subdivided into several “10”-level variables. Any COBOL code which references the name “CLIENT-ADDR” includes the associated “10”-level names.

9.4 MOVE and COMPUTE

MOVE and COMPUTE reserved word statements alter the value of variable names. In the below example (Figure 2) you will see that literals are being sent to the 77-level variable name in the PROCEDURE DIVISION. The COMPUTE statement stores the value of HOURS * RATE in GROSS-PAY. For a COMPUTE statement, you must assign a numeric value data type to the variable by using PIC 9.



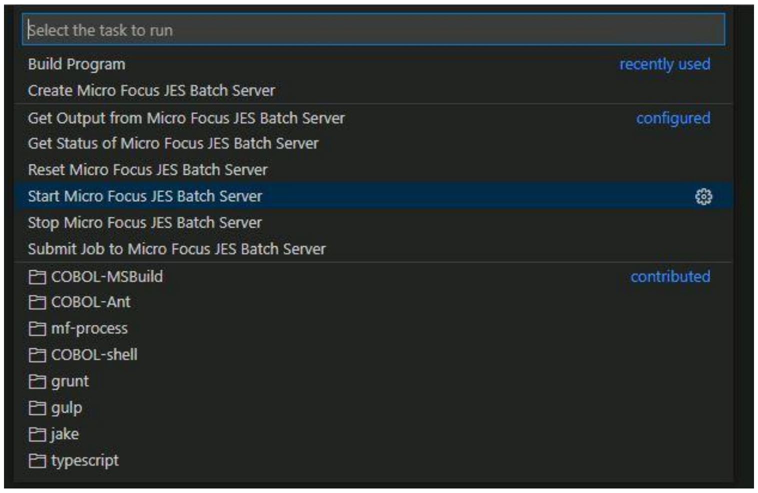
```
*         COBOL reference format (Figure 1., page 32)
*Columns:
*  1         2         3         4         5         6         7
*89012345678901234567890123456789012345678901234567890123456789012
*<A-><-----------------------B------------------------------>
*Area                       Area
*<---Sequence Number Area              Identification Area---->
*-----------------------
 IDENTIFICATION DIVISION.
*-----------------------
 PROGRAM-ID. PAYROL00.
*-------------
 DATA DIVISION.
*-------------
 WORKING-STORAGE SECTION.
****** Variables for the report
* level number
* |  variable name
* |  |          picture clause
* |  |          |
* v  v          v
 77  WHO        PIC X(15).
 77  WHERE      PIC X(20).
 77  WHY        PIC X(30).
 77  RATE       PIC 9(3).
 77  HOURS      PIC 9(3).
 77  GROSS-PAY  PIC 9(5).

* PIC X(15) -- fiftheen alphanumeric characters
* PIC 9(3)  -- three-digit value
*-------------------
 PROCEDURE DIVISION.
*-------------------
****** COBOL MOVE statements - Literal Text to Variables
     MOVE  "Captain COBOL" TO WHO.
     MOVE "San Jose, California" TO WHERE.
     MOVE "Learn to be a COBOL expert" TO WHY.
     MOVE 19 TO HOURS.
     MOVE 23 TO RATE.
* The string "Captain COBOL" only contains 13 characters,
* the remaining positions of variable WHO are filled with spaces
* The value 19 only needs 2 digits,
* the leftmost position of variable HOURS is filled with zero
****** Calculation using COMPUTE reserved word verb
     COMPUTE GROSS-PAY = HOURS * RATE.
* The result of the multiplication only needs 3 digits,
* the remaining leftmost positions are filled with zeroes
****** DISPLAY statements
     DISPLAY "Name: " WHO.
     DISPLAY "Location: " WHERE
     DISPLAY "Reason: " WHY
     DISPLAY "Hours Worked: " HOURS.
     DISPLAY "Hourly Rate: " RATE.
     DISPLAY "Gross Pay: " GROSS-PAY.
     DISPLAY WHY " from " WHO.
     GOBACK.
```

Figure 2 (MOVE and COMPUTE statements)

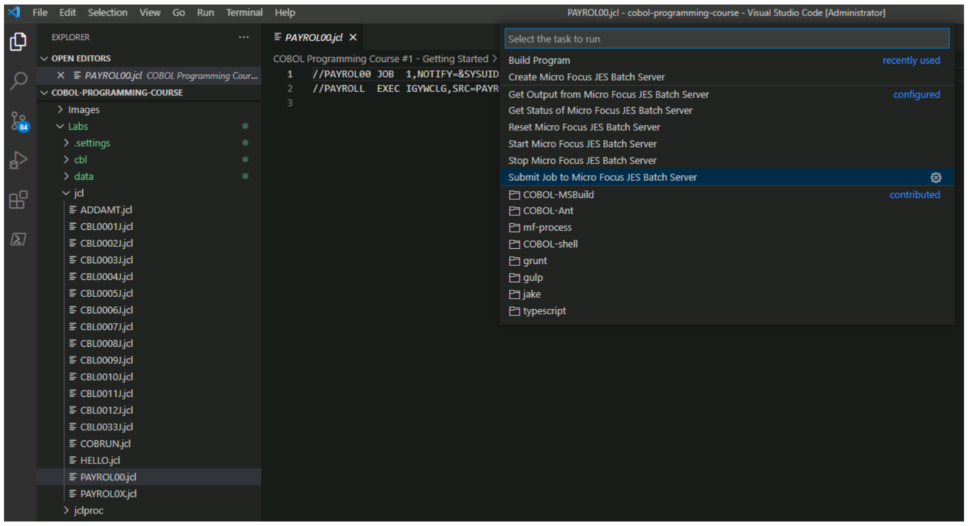
9.5 Lab

In this lab you will compile and debug two more COBOL programs and learn how to deal with a basic compilation error, which you will fix with the theory explained in this chapter.

1. First of all, start the Enterprise Server region. From the Terminal drop down menu, choose Run Task and then Start the Micro Focus JES Batch Server.

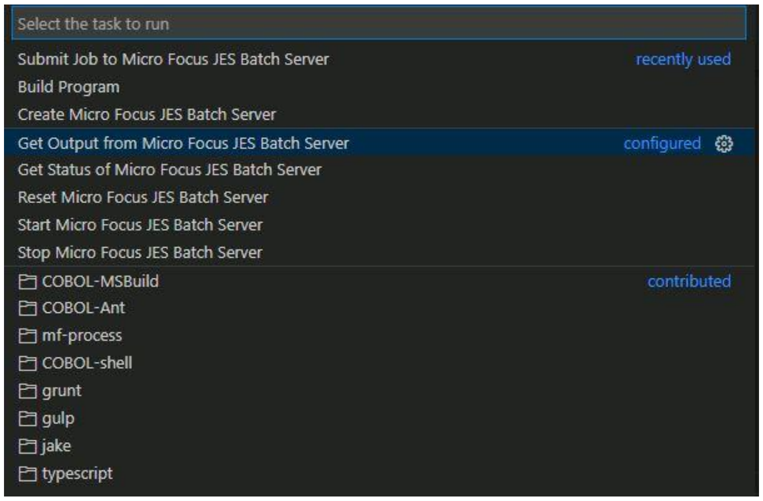

Select the task to run
Build Program
recently used
Create Micro Focus JES Batch Server
Get Output from Micro Focus JES Batch Server
configured
Get Status of Micro Focus JES Batch Server
Reset Micro Focus JES Batch Server
Start Micro Focus JES Batch Server
Stop Micro Focus JES Batch Server
Submit Job to Micro Focus JES Batch Server
COBOL-MSBuild
contributed
COBOL-Ant
mf-process
COBOL-shell
grunt
gulp
jake
typescript

2. In the navigation menu, select and open the member PAYROL00.cbl and take a moment to look at the code. Then compile the program choosing Build Program from the central drop-down menu. Remember that this task compiles the COBOL program that is open in the editor.
3. When the compile is successful, submit job PAYROL00.jcl. To do this, you will need to open the JCL member in the editor and then select Submit Job to Micro Focus JES Batch Server. Wait a moment for the run to be successful.

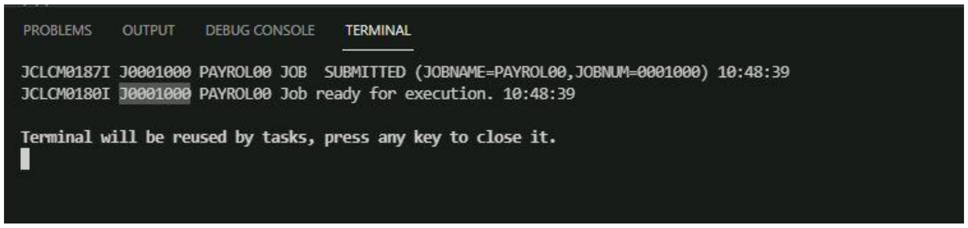


```
//PAYROL00 JOB  1,NOTIFY=&SYSUID
//PAYROLL   EXEC IGYWCLG,SRC=PAYR
```

4. If the run fails, there might be an issue with the Enterprise Server region because it wasn't stopper properly at the end of the previous session. To fix this, select the Reset Micro Focus JES Batch Server menu item, then start the region again.

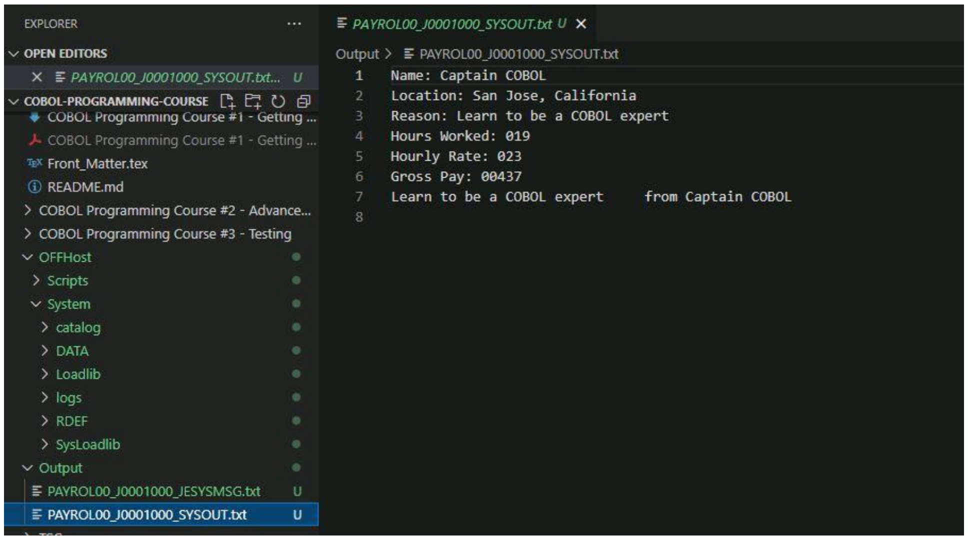

Select the task to run
Submit Job to Micro Focus JES Batch Server
recently used
Build Program
Create Micro Focus JES Batch Server
Get Output from Micro Focus JES Batch Server
configured
Get Status of Micro Focus JES Batch Server
Reset Micro Focus JES Batch Server
Start Micro Focus JES Batch Server
Stop Micro Focus JES Batch Server
COBOL-MSBuild
contributed
COBOL-Ant
mf-process
COBOL-shell
grunt
gulp
jake
typescript

5. If the run is successful, look at the terminal window at the bottom of the editor. Here you will see the Job ID as mentioned before, which you will need to copy in order to download the program output from the Enterprise Server region.



```
JCLCM0187I J0001000 PAYROL00 JOB  SUBMITTED (JOBNAME=PAYROL00,JOBNUM=0001000) 10:48:39
JCLCM0180I J0001000 PAYROL00 Job ready for execution. 10:48:39

Terminal will be reused by tasks, press any key to close it.
```

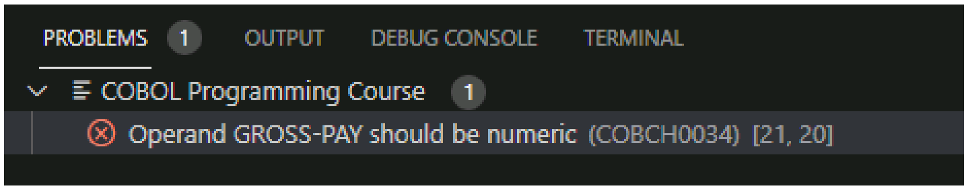
6. Download the output as explained in the previous Lab.
7. Now view the program output. In the Output folder, double click on the file whose name ends in SYSOUT. Note that this tag might change depending on the way the output is referred to in the COBOL code.


EXPLORER
OPEN EDITORS
PAYROL00_J0001000_SYSOUT.txt...
COBOL-PROGRAMMING-COURSE
COBOL Programming Course #1 - Getting ...
COBOL Programming Course #1 - Getting ...
Front_Matter.tex
README.md
COBOL Programming Course #2 - Advance...
COBOL Programming Course #3 - Testing
OFFHost
Scripts
System
catalog
DATA
Loadlib
logs
RDEF
SysLoadlib
Output
PAYROL00_J0001000_JESYSMSG.txt
PAYROL00_J0001000_SYSOUT.txt
PAYROL00_J0001000_SYSOUT.txt
Output > PAYROL00_J0001000_SYSOUT.txt

```
Name: Captain COBOL
Location: San Jose, California
Reason: Learn to be a COBOL expert
Hours Worked: 019
Hourly Rate: 023
Gross Pay: 00437
Learn to be a COBOL expert     from Captain COBOL
```

8. If you compare the source code and the program output, you will see that the program displayed the information in the program output file.

9. Now open the COBOL member PAYROL0X.cbl, look at the source code and then compile the program.
10. The build will be unsuccessful, and you will get an error message in the bottom menu, in the Problems tab. This tab will tell you at what lines the error occurs, and if you click on the message, it will navigate to the correct line that failed compilation in the code.

    ```
    Operand GROSS-PAY should be numeric (COBCH0034) [21, 20]
    ```

    Refer to the theory in this chapter to fix the error, then save and recompile the program.
11. Submit job PAYROL0X.
12. Get the new job output as outlined above and check that it's the same output as the previous job (PAYROL00).
13. Remember to run the task to Stop the Micro Focus Enterprise Server region before you turn off your machine.

Chapter 10 – File Handling

The previous chapter and lab focused on variables and moving literals into variables, and then displaying it using the COBOL DISPLAY statement. This chapter will show you how a COBOL program reads data from an input external data source and writes data to an output external data source. In the lab you will use a simple COBOL program to read each record from a file and write each record to a different file.

10.1 COBOL code used for sequential file handling

There are several file organizations available in COBOL. For now, we will deal with sequential files. A sequential file is a type of file which contains records organized by the order in which they were entered. COBOL code used for sequential fie handling involves:

- ENVIRONMENT DIVISION
    - SELECT clauses
    - ASSIGN clauses
- DATA DIVISION
    - FD statements
- PROCEDURE DIVISION
    - OPEN statements
    - CLOSE statements
    - READ INTO statement
    - WRITE FROM statement

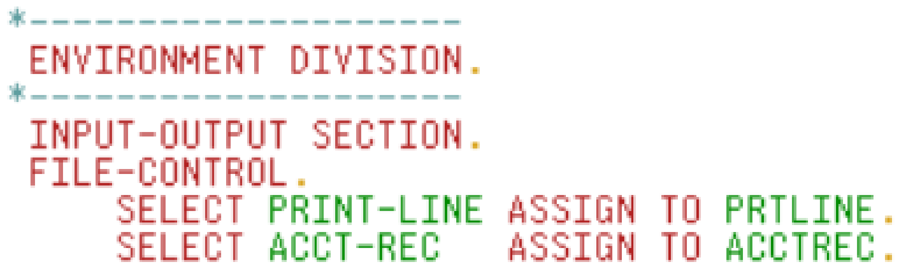
10.1.1 COBOL inputs and outputs

The ENVIRONMEMT DIVISION and DATA DIVISION describe the inputs and outputs used in the PROCEDURE DIVISION. Chapters (8) and (9) introduced variable descriptions in the DATA DIVISION. In the ENVIRONMENT DIVISION and more specifically, the FILE-CONTROL paragraph in the INPUT-OUTPUT SECTION introduces accessing external data sources and moving the data from them.

10.1.2 FILE-CONTROL paragraph

In the ENVIRONMENT DIVISION in *Figure 1* below, you will notice the INPUT-OUTPUT section and the FILE-CONTROL paragraph. In this FILE-CONTROL paragraph, there are two clauses:

- SELECT – This clause creates an internal file name
- ASSIGN – This clause creates a name for an external data source

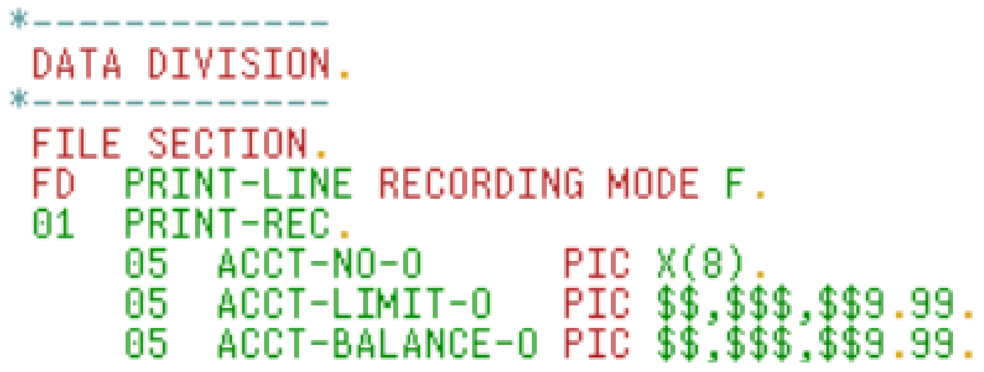


```
*--------------------
 ENVIRONMENT DIVISION.
*--------------------
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT PRINT-LINE ASSIGN TO PRTLINE.
     SELECT ACCT-REC   ASSIGN TO ACCTREC.
```

Figure 1. FILE-CONTROL

When a COBOL program is compiled, it receives more information about the internal file and external data source in the FILE SECTION, located in the DATA DIVISION. The COBOL reserved word 'FD' is used to give the COBOL compiler more information about internal file names in the FILE-SECTION.

Have a look at the record layout below *(Figure 2)*, which consists of level numbers, variable names, data types and lengths.



```
*-------------
 DATA DIVISION.
*-------------
 FILE SECTION.
 FD  PRINT-LINE RECORDING MODE F.
 01  PRINT-REC.
     05  ACCT-NO-O      PIC X(8).
     05  ACCT-LIMIT-O   PIC $$,$$$,$$9.99.
     05  ACCT-BALANCE-O PIC $$,$$$,$$9.99.
```

Figure 2. FILE-SECTION

10.1.4 Data sets, records and fields

A sequential data file has many records. A record is a single line in the data file and has a defined length. Each record can be subdivided into fields where each field has a defined length. Therefore, the sum of all field lengths would equal the sum of the record.

10.1.6 ASSIGN clause

While the SELECT clause name is an internal file name, the ASSIGN clause name describes a data source external to the program. In our labs, the programs rely on Job Control Language (JCL) operations to tell the system what program to load and execute, followed by input and output names needed by the program. The JCL input and output names are called DDNAMEs, with DD being an abbreviation for Data Definition.

COBOL code "SELECT ACCT-REC ASSIGN TO ACCTREC" requires a JCL DDNAME ACCTREC with a DD redirecting ACCTREC to a Micro Focus Enterprise Developer controlled dataset name.

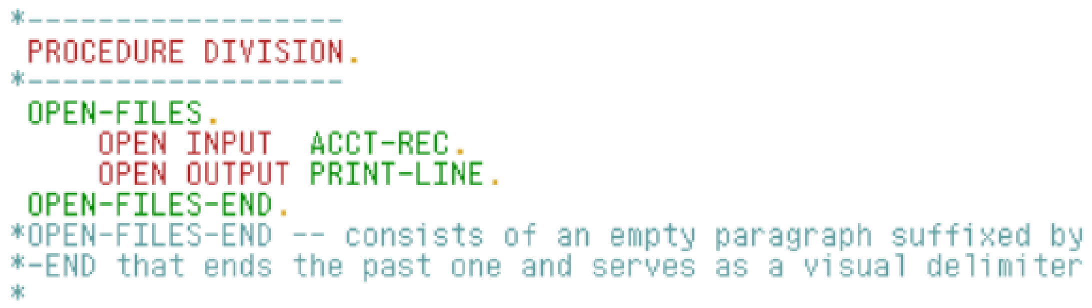
Redirecting ACCT-REC to the external data source ACCTREC using the ASSIGN clause as shown in *Figure 1* allows us more flexibility. This flexibility allows the same COBOL program to access a different environment variable modification:

```
SET ACCTREC="C:\COBOLTraining\COBOLCourse\data\ACCTREC.dat"
```

Don't worry too much about this at this stage as it is not COBOL programming, but rather a separate technical skill – just try to have a high-level understanding of what we covered and why it is relevant to your project.

10.2.1 Opening and Closing Input and Output

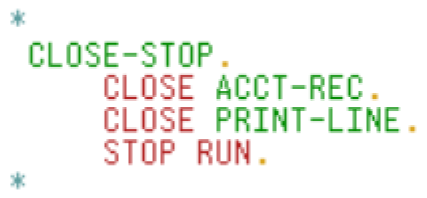
COBOL inputs and outputs must be opened to connect the selected internal name to the assigned external name. The example in *Figure 4* opens the file name ACCT-REC as program input and file name PRINT-LINE as program output.



```
*------------------
 PROCEDURE DIVISION.
*------------------
 OPEN-FILES.
     OPEN INPUT  ACCT-REC.
     OPEN OUTPUT PRINT-LINE.
 OPEN-FILES-END.
*OPEN-FILES-END -- consists of an empty paragraph suffixed by
*-END that ends the past one and serves as a visual delimiter
*
```

Figure 4. OPEN-FILES

COBOL inputs and outputs should always be closed – whether that is when the program has been completed or when the program is done reading from or writing to the internal file name. *Figure 5* closes the internal file names ACCT-REC and PRINT-LINE, and then stops processing with STOP RUN.



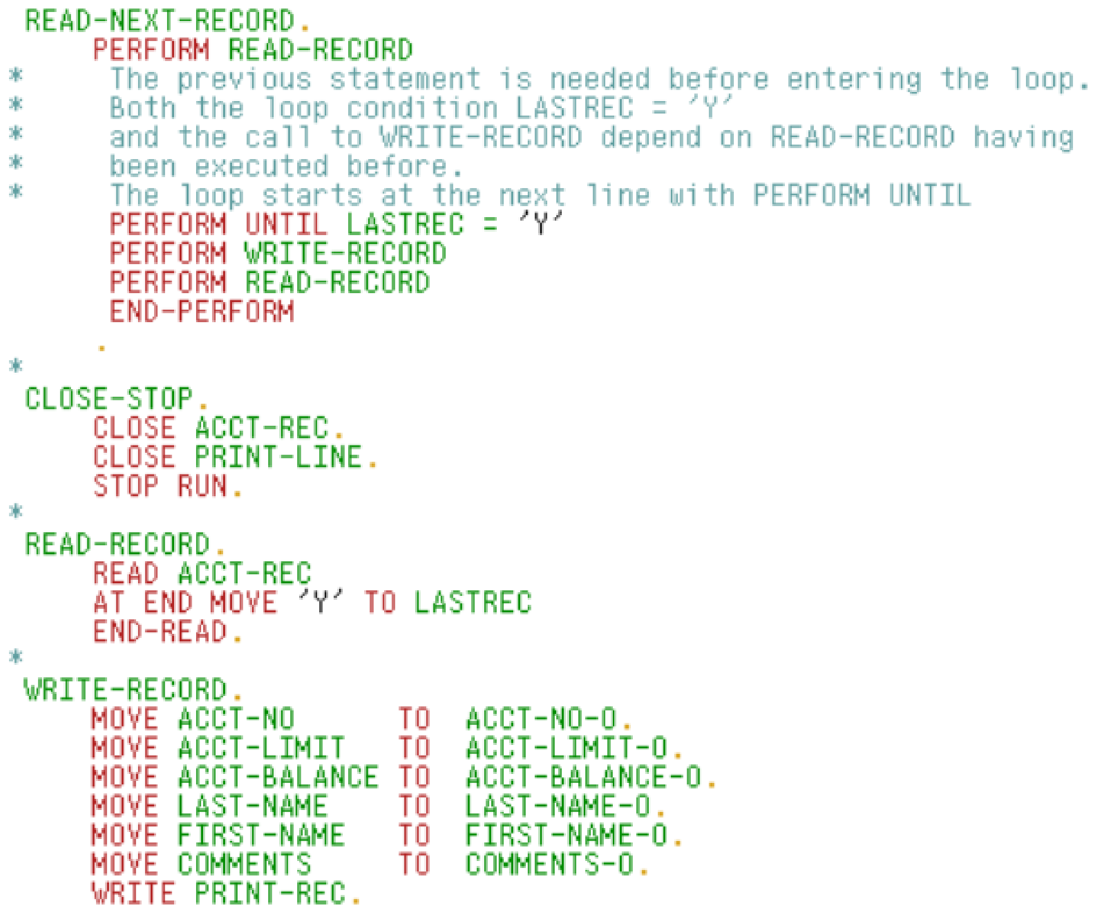
```
*
 CLOSE-STOP.
     CLOSE ACCT-REC.
     CLOSE PRINT-LINE.
     STOP RUN.
*
```

Figure 5 – CLOSE-STOP

10.3 COBOL programming techniques to read and write records sequentially

When reading records, a COBOL program will first check for no records to be read, or check for no more records to be read. If a record exists, the fields in the read record populate variable names defined by the FD clause in the DATA DIVISION.

Observe *Figure 6*:



```
       READ-NEXT-RECORD.
           PERFORM READ-RECORD
      *     The previous statement is needed before entering the loop.
      *     Both the loop condition LASTREC = 'Y'
      *     and the call to WRITE-RECORD depend on READ-RECORD having
      *     been executed before.
      *     The loop starts at the next line with PERFORM UNTIL
            PERFORM UNTIL LASTREC = 'Y'
            PERFORM WRITE-RECORD
            PERFORM READ-RECORD
            END-PERFORM
           .
      *
       CLOSE-STOP.
           CLOSE ACCT-REC.
           CLOSE PRINT-LINE.
           STOP RUN.
      *
       READ-RECORD.
           READ ACCT-REC
           AT END MOVE 'Y' TO LASTREC
           END-READ.
      *
       WRITE-RECORD.
           MOVE ACCT-NO      TO  ACCT-NO-O.
           MOVE ACCT-LIMIT   TO  ACCT-LIMIT-O.
           MOVE ACCT-BALANCE TO  ACCT-BALANCE-O.
           MOVE LAST-NAME    TO  LAST-NAME-O.
           MOVE FIRST-NAME   TO  FIRST-NAME-O.
           MOVE COMMENTS     TO  COMMENTS-O.
           WRITE PRINT-REC.
```

Figure 6 – Reading and Writing Records

Let’s have a look at this example a little closer. The READ-NEXT-RECORD paragraph is a COBOL programming technique used to read all records from a sequential file UNTIL the last record is read. Then, the READ-RECORD paragraph executes the COBOL READ statement, which populates the external sequential file with the variables associated with the ACCT-REC internal file name.

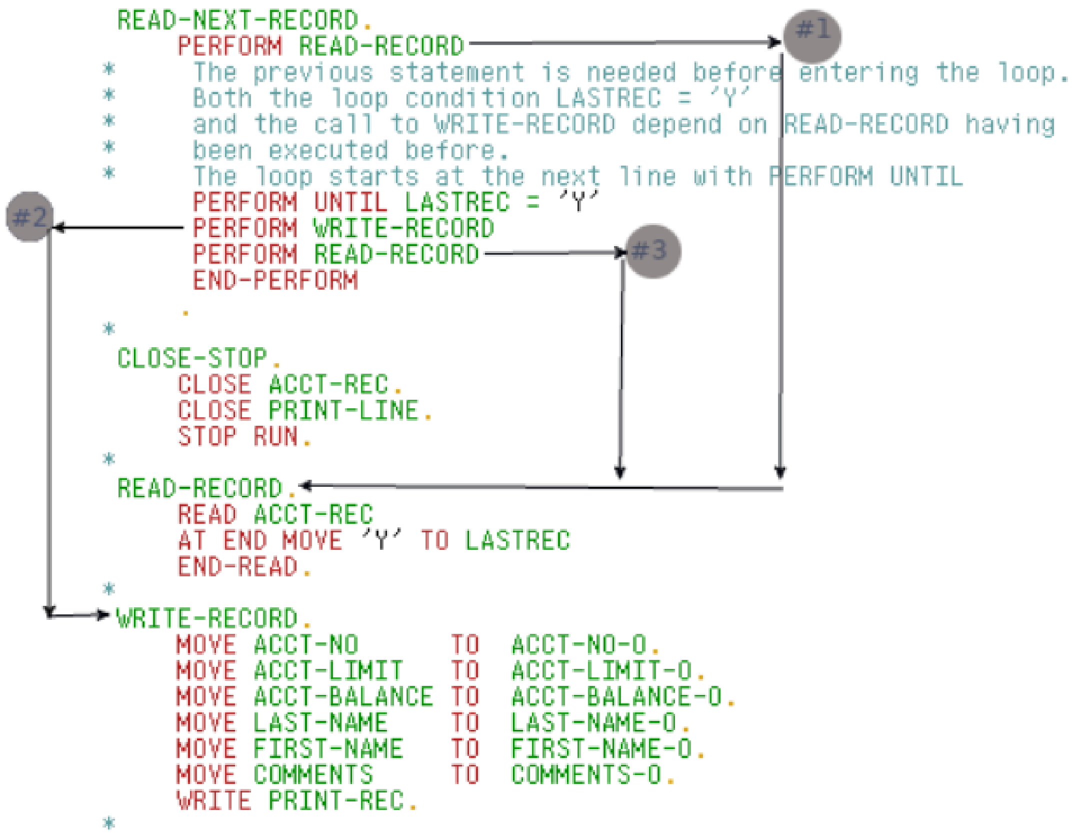
The WRITE-RECORD paragraph contains several MOVE statements. These statements move each input file variable name to an output file variable name. In the final sentence, the program writes the collection of output file variable names, PRINT-REC.

When the final record is read, then ‘Y’ is moved into the LASTREC variable. This then hands control to the CLOSE-STOP paragraph.

PRINT-REC is assigned to the PRTREC external data source. The associated environment variable path redirects the output path to the specified file name, in this case PRTREC.

When the last record is encountered, CLOSE-STOP is executed, stopping the program.

The diagram in *Figure 7* shows the iterative process in the COBOL program.



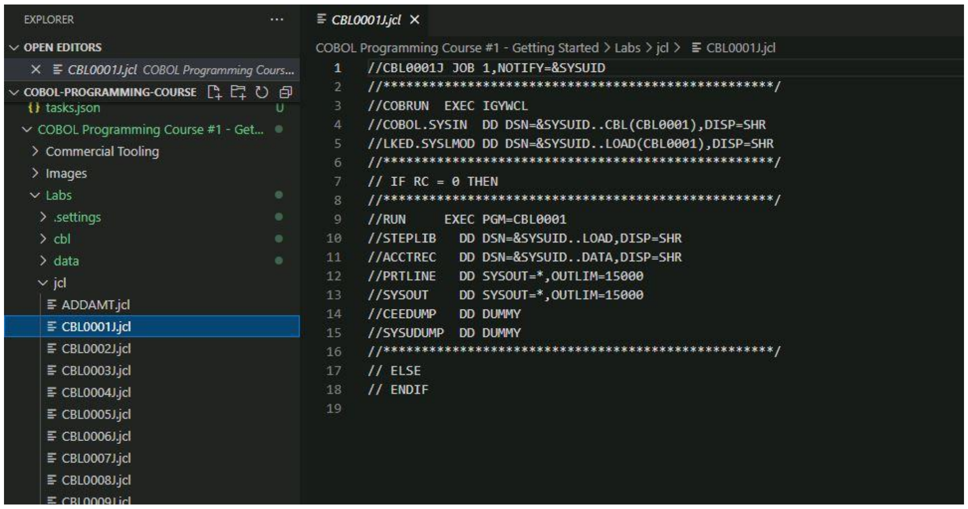
```
 READ-NEXT-RECORD.
     PERFORM READ-RECORD
*      The previous statement is needed before entering the loop.
*      Both the loop condition LASTREC = 'Y'
*      and the call to WRITE-RECORD depend on READ-RECORD having
*      been executed before.
*      The loop starts at the next line with PERFORM UNTIL
      PERFORM UNTIL LASTREC = 'Y'
      PERFORM WRITE-RECORD
      PERFORM READ-RECORD
      END-PERFORM
     .
*
 CLOSE-STOP.
     CLOSE ACCT-REC.
     CLOSE PRINT-LINE.
     STOP RUN.
*
 READ-RECORD.
     READ ACCT-REC
     AT END MOVE 'Y' TO LASTREC
     END-READ.
*
 WRITE-RECORD.
     MOVE ACCT-NO      TO  ACCT-NO-O.
     MOVE ACCT-LIMIT   TO  ACCT-LIMIT-O.
     MOVE ACCT-BALANCE TO  ACCT-BALANCE-O.
     MOVE LAST-NAME    TO  LAST-NAME-O.
     MOVE FIRST-NAME   TO  FIRST-NAME-O.
     MOVE COMMENTS     TO  COMMENTS-O.
     WRITE PRINT-REC.
*
```

Figure 7 – Iterative Processing

10.4 Lab

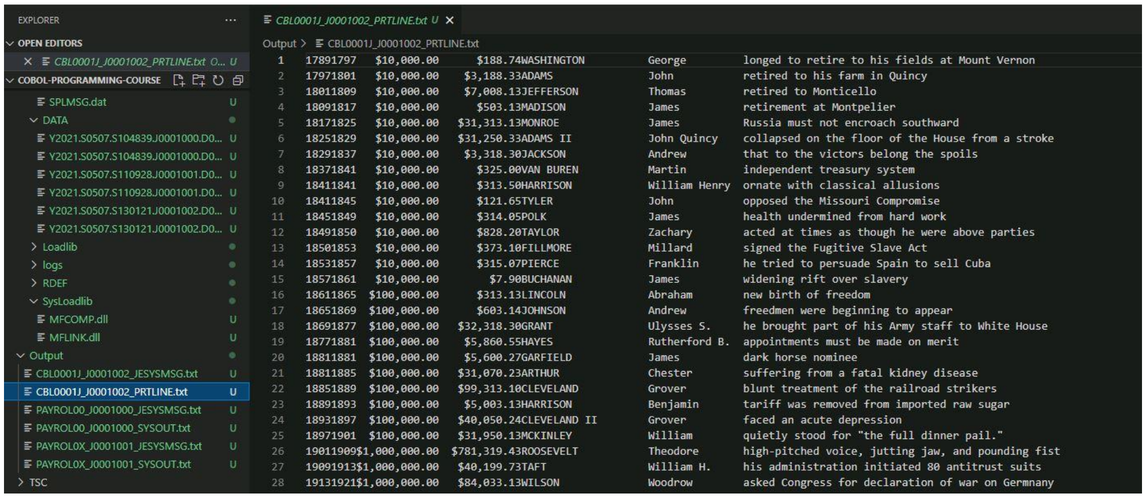
In this Lab, you will explore file handling and explore the end-of-file coding technique. If a step has an asterisk (*) at the end of it, there will be a hint associates with it at the bottom of this Lab.

1. If not already, Open VS Code and start the Enterprise Server region. If there are any issues with the region, try resetting it and then starting it again.
2. Open and view the two COBOL members CBL0001 and CBL0002.
3. Open and view the JCL members CBL0001J, CBL0002J and CBL0033J.
4. Compile CBL0001 and submit job CBL0001J.


CBL0001J.jcl
COBOL Programming Course #1 - Getting Started > Labs > jcl > CBL0001J.jcl

```
//CBL0001J JOB 1,NOTIFY=&SYSUID
//***************************************************/
//COBRUN  EXEC IGYWCL
//COBOL.SYSIN  DD DSN=&SYSUID..CBL(CBL0001),DISP=SHR
//LKED.SYSLMOD DD DSN=&SYSUID..LOAD(CBL0001),DISP=SHR
//***************************************************/
// IF RC = 0 THEN
//***************************************************/
//RUN     EXEC PGM=CBL0001
//STEPLIB   DD DSN=&SYSUID..LOAD,DISP=SHR
//ACCTREC   DD DSN=&SYSUID..DATA,DISP=SHR
//PRTLINE   DD SYSOUT=*,OUTLIM=15000
//SYSOUT    DD SYSOUT=*,OUTLIM=15000
//CEEDUMP   DD DUMMY
//SYSUDUMP  DD DUMMY
//***************************************************/
// ELSE
// ENDIF
```

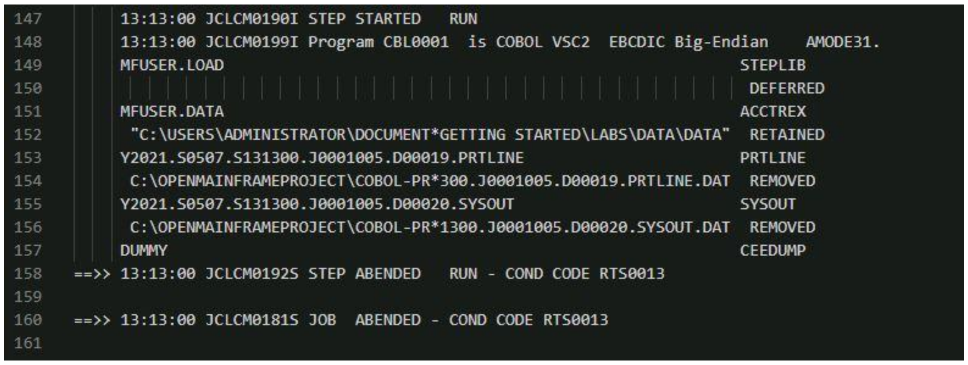
5. View the job output to ensure the job ran successfully.
6. Now look at the program output.


CBL0001J_J0001002_PRTLINE.txt
Output > CBL0001J_J0001002_PRTLINE.txt

```
17891797   $10,000.00      $188.74WASHINGTON         George         longed to retire to his fields at Mount Vernon
17971801   $10,000.00    $3,188.33ADAMS              John           retired to his farm in Quincy
18011809   $10,000.00    $7,008.13JEFFERSON          Thomas         retired to Monticello
18091817   $10,000.00      $503.13MADISON            James          retirement at Montpelier
18171825   $10,000.00   $31,313.13MONROE             James          Russia must not encroach southward
18251829   $10,000.00   $31,250.33ADAMS II           John Quincy    collapsed on the floor of the House from a stroke
18291837   $10,000.00    $3,318.30JACKSON            Andrew         that to the victors belong the spoils
18371841   $10,000.00      $325.00VAN BUREN          Martin         independent treasury system
18411841   $10,000.00      $313.50HARRISON           William Henry  ornate with classical allusions
18411845   $10,000.00      $121.65TYLER              John           opposed the Missouri Compromise
18451849   $10,000.00      $314.05POLK               James          health undermined from hard work
18491850   $10,000.00      $828.20TAYLOR             Zachary        acted at times as though he were above parties
18501853   $10,000.00      $373.10FILLMORE           Millard        signed the Fugitive Slave Act
18531857   $10,000.00      $315.07PIERCE             Franklin       he tried to persuade Spain to sell Cuba
18571861   $10,000.00        $7.90BUCHANAN           James          widening rift over slavery
18611865  $100,000.00      $313.13LINCOLN            Abraham        new birth of freedom
18651869  $100,000.00      $603.14JOHNSON            Andrew         freedmen were beginning to appear
18691877  $100,000.00   $32,318.30GRANT              Ulysses S.     he brought part of his Army staff to White House
18771881  $100,000.00    $5,860.55HAYES              Rutherford B.  appointments must be made on merit
18811881  $100,000.00    $5,600.27GARFIELD           James          dark horse nominee
18811885  $100,000.00   $31,070.23ARTHUR             Chester        suffering from a fatal kidney disease
18851889  $100,000.00   $99,313.10CLEVELAND          Grover         blunt treatment of the railroad strikers
18891893  $100,000.00    $5,003.13HARRISON           Benjamin       tariff was removed from imported raw sugar
18931897  $100,000.00   $40,050.24CLEVELAND II       Grover         faced an acute depression
18971901  $100,000.00   $31,950.13MCKINLEY           William        quietly stood for "the full dinner pail."
19011909$1,000,000.00  $781,319.43ROOSEVELT          Theodore       high-pitched voice, jutting jaw, and pounding fist
19091913$1,000,000.00   $40,199.73TAFT               William H.     his administration initiated 80 antitrust suits
19131921$1,000,000.00   $84,033.13WILSON             Woodrow        asked Congress for declaration of war on Germnany
```

If you compare it with the source code, you will see that this program processes data from a given data file (found in the data folder, found in the Labs repository) and prints it out on a new data file, which is the PRTLINE program output.

7. Compile CBL0002.
8. Notice the unsuccessful build. Fix the code and recompile. *
9. Submit job CBL0002J and view the job output and the program output, which should look the same as the one from CBL0001.
10. Submit job CBL0033J and view the job output.
11. Scroll to the bottom and notice the ABEND message.



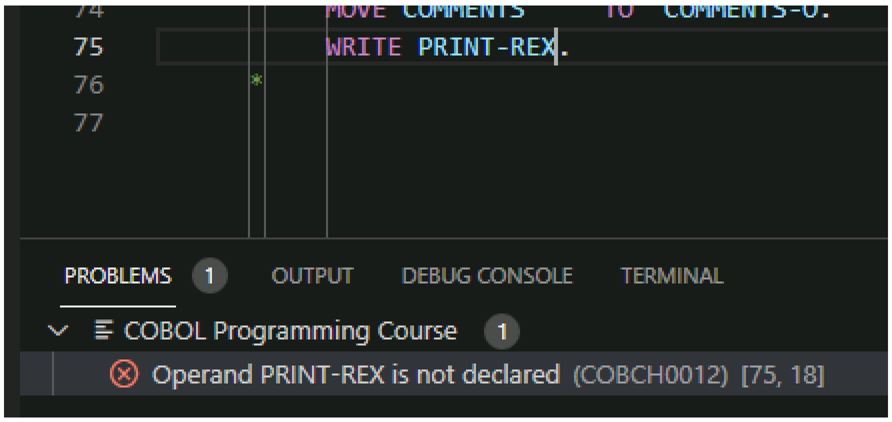
```
     13:13:00 JCLCM0190I STEP STARTED   RUN
     13:13:00 JCLCM0199I Program CBL0001  is COBOL VSC2  EBCDIC Big-Endian    AMODE31.
     MFUSER.LOAD                                                        STEPLIB
                                                                         DEFERRED
     MFUSER.DATA                                                        ACCTREX
      "C:\USERS\ADMINISTRATOR\DOCUMENT*GETTING STARTED\LABS\DATA\DATA"  RETAINED
     Y2021.S0507.S131300.J0001005.D00019.PRTLINE                        PRTLINE
      C:\OPENMAINFRAMEPROJECT\COBOL-PR*300.J0001005.D00019.PRTLINE.DAT  REMOVED
     Y2021.S0507.S131300.J0001005.D00020.SYSOUT                         SYSOUT
      C:\OPENMAINFRAMEPROJECT\COBOL-PR*1300.J0001005.D00020.SYSOUT.DAT  REMOVED
     DUMMY                                                              CEEDUMP
==>> 13:13:00 JCLCM0192S STEP ABENDED   RUN - COND CODE RTS0013

==>> 13:13:00 JCLCM0181S JOB  ABENDED - COND CODE RTS0013
```

ABEND stands for 'abnormal end' and signals that the job failed due to an error in the JCL.

12. Open JCL member CBL0033J, find the error and fix it. *
13. Save and the resubmit the job.
14. View the job and program output, to ensure that it looks the same as the ones from CBL0001 and CBL0002.

Hints:

1. Compare the source code of CBL0001 and CBL0002 to fix the error located at line 75.



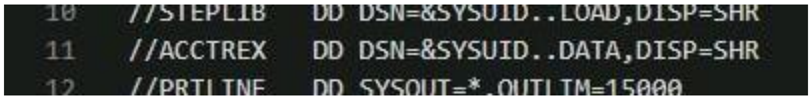
```
74               MOVE COMMENTS      TO  COMMENTS-O.
75               WRITE PRINT-REX.
76          *
77
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL

COBOL Programming Course 1

Operand PRINT-REX is not declared (COBCH0012) [75, 18]

2. At line 11, you will notice the incorrect spelling of ACCTREC, which you can confirm looking at the COBOL source code in the working storage section.



```
10   //STEPLIB    DD DSN=&SYSUID..LOAD,DISP=SHR
11   //ACCTREX    DD DSN=&SYSUID..DATA,DISP=SHR
12   //PRTLINE    DD SYSOUT=*,OUTLIM=15000
```

Chapter 11 - Program Structure

In this chapter we discuss the concept of structured programming, how it relates to COBOL, and highlight the key techniques within the COBOL language that allow you to write well-structured programs.

It is important to learn how to structure your code well because it makes the code much easier to read and follow, especially if someone else was to try to read and decipher your code.

11.1 Styles of Programming

Before we discuss in more detail how to structure a program written in COBOL, it is important to understand the type of language COBOL is and how it may differ from other programming languages.

11.1.1 What is Structured Programming?

COBOL, as well as other languages such as Python and C, use techniques called structured programming. Structured programming is the name given to a set of programming styles which could include loops, functions, methods and more, and allow a programmer to organize their code in a meaningful way.

Unstructured programming, also known as spaghetti code, allows the flow of the execution to branch wildly around the source code. Although COBOL does contain these structures, such as GOTO or JUMP, they are generally considered bad practice and should be used sparingly.

11.1.2 What is Object Oriented Programming?

Object-Oriented programming, or OO programming, differs from structured programming, although it borrows a lot of the same concepts. In OO programming, code is spit up into multiple classes, each made up of variables and a sequence of methods. It is possible to use Object Oriented COBOL, however we won't be covering it in this guide. If you would like to learn about Object Oriented COBOL, read the Micro Focus Introduction to Object-Oriented Programming for COBOL Developers.

11.1.3 COBOL programming style

COBOL doesn't directly have some of the components of a structured programming language such as Java. COBOL doesn't contain 'for' or 'while' loops, nor does it contain defined functions or methods. Because COBOL is designed to be a language which is easy to read, these concepts are embodied through the PERFORM keyword and the concept of paragraphs. This allows the programmer to still create these structures, but in a way that is easy to read and follow.

11.2 Structure of the Procedure Division

As you already know, a COBOL program is split into several divisions. This chapter tells you how to structure the procedure division specifically, in order to make sure it is easy to read, understandable and maintainable in the future. Typically, execution in a COBOL program begins at the first statement within the procedure division and progresses sequentially through each line until it reaches the end of the source code.

1.2.2 Inline and out of line perform statements

The PERFORM keyword is a very flexible element of the COBOL language. At the most basic level, a PERFORM allows control to be transferred to another section of code, such as a paragraph. Once this section has executed, control returns to the following line of code. Take the following example:

```
    OPEN OUTPUT PRINT-LINE.

    MOVE 'THE NUMBER IS: ' TO MSG-HEADER OF PRINT-REC.

    PERFORM WRITE-NEW-RECORD.
    PERFORM WRITE-NEW-RECORD.
    PERFORM WRITE-NEW-RECORD.
    PERFORM WRITE-NEW-RECORD.
    PERFORM WRITE-NEW-RECORD.
    PERFORM WRITE-NEW-RECORD.
    PERFORM WRITE-NEW-RECORD.
    PERFORM WRITE-NEW-RECORD.
    PERFORM WRITE-NEW-RECORD.
    PERFORM WRITE-NEW-RECORD.


    CLOSE PRINT-LINE.
    STOP RUN.


WRITE-NEW-RECORD.
    ADD 1 TO COUNTER GIVING COUNTER
    MOVE COUNTER TO MSG-TO-WRITE
    WRITE PRINT-REC.
```

In this example, the three lines of code that constructed a new line of output and printed it has been extracted into a new paragraph called WRITE-NEW-RECORD. This paragraph is then performed ten times by the use of the PERFORM keyword. Each time the PERFORM keyword is used, execution jumps to the paragraph WRITE-NEW-RECORD, executes the three lines contains within that paragraph before returning to the line following the PERFORM statement. The concept of a paragraph will be covered later in this chapter in further depth.

1.2.3 Using Performs to code a loop

The code we have built so far is not optimal – the repetition of the PERFORM statement ten times is inelegant and can be optimized. Observe the following snippet of code:

```
MOVE 'THE NUMBER IS: ' TO MSG-HEADER OF PRINT-REC.
```

```
PERFORM VARYING COUNTER FROM 01 BY 1 UNTIL COUNTER EQUAL 11
MOVE COUNTER TO MSG-TO-WRITE
WRITE PRINT-REC
END-PERFORM.

CLOSE PRINT-LINE.
STOP RUN.
```

In this example, this loop runs from the PERFORM statement keyword to the END-PERFORM keyword. Each time the execution iterates over the loop, the value of COUNTER is incremented and tested by one.

11.2.4 Learning bad behavior using the GO TO keyword

If there is one thing COBOL programmers can agree on, it is that the use of the GO TO statement is generally a bad idea. Using this statement can result in compilation errors and code which is messy and hard to follow.

So why is the GO TO command so frowned upon? To answer this, we must understand the key difference between GO TO and PERFORM. When using the PERFORM keyword, the control gets handed to the relevant paragraph, and then afterwards the execution returned to the line following the PERFORM statement. On the other hand, when the GO TO keyword was used, the paragraph it refers to is executed, the same way it is when using the PERFORM statement, but then after the execution, the COBOL code continues on to the next line following the executed paragraph, which causes a branch of execution that doesn't return to the line of code that issued it.

By giving you some understanding of the GO TO command's behavior, you will be better equipped when looking through existing code and maintaining it.

11.3 Paragraphs as blocks of code

In this section, we will talk more about paragraphs, how they work and what they can be used for. A COBOL paragraph is basically a block of code containing a sequence of actions that could be used multiple times within the same program.

Paragraphs can be ended by END-PROGRAM, END-METHOD, END FACTORY or END-OBJECT. Most of these are used within Object Oriented COBOL, which is not discussed here.

Considering that a program can be made up of multiple paragraphs and that the PERFORM keyword can be used to call the paragraph, it is easy to see that good paragraph design can really help make your COBOL more structured and readable.

11.3.1 Designing the content of a paragraph

There are no restrictions as to what content can go inside a paragraph, however there are two main reasons why you might want to refactor code to be inside a paragraph:

1. To group a sequence of COBOL sentences together that achieve a particular function or task, such as open all the files that an application is using, calculate a particular function or perform some data validation. Grouping such functions into a paragraph allows you to give them a meaningful name to explain the purpose of the lines of code.
2. The sequence of sentences will be used within a loop. Extracting these lines into a paragraph and then using the PERFORM keyword to create a loop can make for very comprehensible code.

Remember that you can also perform other paragraphs within existing paragraphs. This nested calling of paragraphs can, again, help to structure your code.

11.3.2 Order and naming of paragraphs

There is no requirement about the order that paragraphs should appear within a COBOL program. A paragraph can be called from a point either before or after where it is declared. Although there are no restrictions enforced by the language, there are some techniques that you can follow which will make your programs easier to follow and understand. Some of these techniques and best practices are:

- Name each paragraph to correspond best with its function or behaviour. A paragraph named OPEN-INPUT-FILES is a lot easier to understand than one named DO-FILE-STUFF.
- Order the paragraphs in the general sequence they will be executed at runtime.
- Some COBOL programmers prefix the name of paragraphs with a number that increases throughout the source code – see *Example 8*
- Because the paragraphs are numbered and appear in the source code in that order, when a sentence references a paragraph, it is easier to know where in the program that paragraph might appear. When initially structuring a program this way, the numbers would only increment the highest significant figure, allowing for new paragraphs to be inserted in between if needed. Although the ride of modern editors, which allow outlining and instant jumping to a reference or declaration, makes this technique of less necessity, it is still useful to understand.

```
           PERFORM 1000-OPEN-FILES.
           PERFORM 2000-READ-NEXT-RECORD.
           GO TO 3000-CLOSE-STOP.
       1000-OPEN-FILES.
           OPEN INPUT  ACCT-REC.
           OPEN OUTPUT PRINT-LINE.
      *
       2000-READ-NEXT-RECORD.
           PERFORM 4000-READ-RECORD
           PERFORM UNTIL LASTREC = 'Y'
           PERFORM 5000-WRITE-RECORD
```

```
           PERFORM 4000-READ-RECORD
           END-PERFORM.
      *
       3000-CLOSE-STOP.
           CLOSE ACCT-REC.
           CLOSE PRINT-LINE.
           STOP RUN.
      *
       4000-READ-RECORD.
           READ ACCT-REC
           AT END MOVE 'Y' TO LASTREC
           END-READ.
      *
       5000-WRITE-RECORD.
           MOVE ACCT-N   -     TO  ACCT-NO-O.
           MOVE ACCT-LIMIT     TO  ACCT-LIMIT-O.
           MOVE ACCT-BALANCE   TO  ACCT-BALANCE-O.
           MOVE LAST-NAME      TO  LAST-NAME-O.
           MOVE FIRST-NAME     TO  FIRST-NAME-O.
           MOVE COMMENTS       TO  COMMENTS-O.
           WRITE PRINT-REC.
```

Example 8. Numbered paragraphs

- Lastly, it is common to explicitly end a paragraph by coding an empty paragraph following each paragraph (see *Example 9).* This paragraph does not contain any code, has the same name as the paragraph it is closing suffixed with –END.

```
 1000-OPEN-FILES.
     OPEN INPUT  ACCT-REC.
     OPEN OUTPUT PRINT-LINE.
 1000-OPEN-FILES-END.
*
 2000-READ-NEXT-RECORD.
     PERFORM 4000-READ-RECORD
     PERFORM UNTIL LASTREC = 'Y'
     PERFORM 5000-WRITE-RECORD
     PERFORM 4000-READ-RECORD
     END-PERFORM.
 2000-READ-NEXT-RECORD-END.
```

Example 9. Explicitly closed paragraphs

11.4 Program Control with paragraphs

So far in this chapter we have discussed the importance of using paragraphs to structure your code. In doing this, we used the PERFORM keyword several times to execute the paragraphs we had created. In this section, we will discuss in more detail some of the ways in which the PERFORM keyword can be used.

11.4.1 PERFORM TIMES

Perhaps the simplest way of repeating a PERFORM statement is to use the TIMES keyword to perform a paragraph or sections of code a static number of times, as shown in *Example 10*

```
PERFORM 10 TIMES
 MOVE FIELD-A TO FIELD-B
 WRITE RECORD
END-PERFORM.
```

Example 10. TIMES

11.4.2 PERFORM THROUGH

You may require a sequential list of paragraphs to be executed in turn, instead of performing them individually. The THROUGH or THRU keyword can be used to list the start and end paragraphs of the list. Execution will progress through each of the paragraphs as they appear in the source code, from beginning to end, before returning to the line following the initial PERFORM statement, which can be seen in *Example 12*

```
 1000-PARAGRAPH-A.
     PERFORM 2000-PARAGRAPH-B THRU
             3000-PARAGRAPH-C.
*
 2000-PARAGRAPH-B.
     ...
*
 3000-PARAGRAPH-C.
     ...
*
 4000-PARAGRAPH-D.
     ...
```

Example 12. PERFORM THRU

NOTE: The use of the THROUGH/THRU keyword can also be used alongside the TIMES, UNTIL and VARYING keywords, to allow the list of paragraphs to be executed rather than just a single paragraph or blocks of code.

With structured programming, the use of sections rather than PERFORM THROUGH is recommended.

11.4.3 PERFORM UNTIL

Adding the UNTIL keyword to a perform sentence allows you to iterate over a group of sentences until the Boolean (true/false) condition is met.

11.4.4 Perform Varying

With the PERFORM VARYING syntax, the variable counter is tested to see if it equals 11, as long as it doesn't then it is incremented, and the sentences nested within the perform statement are exectuted as shown in *Example 18*.

```
PERFORM VARYING COUNTER FROM 01 BY 1 UNTIL COUNTER EQUAL 11
     AFTER COUNTER-2 FROM 01 BY 1 UNTIL COUNTER-2 EQUAL 5
...
END-PERFORM.
```

Example 18. Perform Varying

11.5 Using Subprograms

So far, we have only examined the internal structure of a single COBOL program. As programs increase in function and number, a programmer may want certain aspects of a program's function to be made available to other programs within the system. By extracting these functions into their own separate programs and allowing them to be called from other programs can reduce the amount of code duplication and therefore decrease the cost of maintenance, as fixes to shared modules only need to be made once.

Note: Although here we will describe the COBOL native way of calling another program, please note that some middleware products will provide APIs that might do this in an enhanced way.

When calling another program, we need to consider three main concerns:

- How we will reference the program we wish to call
- The parameters we want to send the target program
- The parameters we want the target program to return

11.5.1 Specifying the target program

To call a target program we will use the keyword CALL followed by a reference to the target program we wish to call. The two main ways to do this are by a literal value or by referencing a variable, as shown in *Example 19.*

```
CALL 'PROGA1' ...
...
MOVE 'PROGA2' TO PROGRAM-NAME.
CALL PROGRAM-NAME ...
```

Example 19. Basic Call

11.5.2 Specifying Program Variables

Now that we have identified the name of the program we wish to call, we must identify the variables that the calling program might want to send. These are individually specified by the USING keyword.

By default, COBOL will pass data items BY REFERENCE. This means that both the calling and the target programs are able to read and write to the same area of memory that is represented by the variable. That means that id the target program updates the content of the variable, that change will be visible to the calling program once execution has returned.

It is also possible to pass variables BY CONTENT, which allows a copy of the variable to be passed to the target program. Although the target program is able to update the variable, those updates will not be visible to the calling program.

Note: When passing variables either BY REFERENCE or BY CONTENT, you can send data items of any level. This means that you can pass entire data structures, which can be very useful for dealing with common records.

You may also come across the phrase BY CONTENT being used In a CALL sentence. BY CONTENT is very similar to the BY VALUE phrase, however this is primarily used when COBOL is calling a program written in a different language (such as C).

11.5.3 Specifying the return value

Finally, the RETURNING phrase is used to specify the variable that should be used to store the return value. This can be any elementary data-item declared within the Data Division. Some programs may not return anything, or you might have passed values BY REFERENCE to the target program, in which case any updates to these variables will be visible once the target program returns.

11.6 Summary

In summary, this chapter should provide the necessary foundation to understand structured programming and how it relates to COBOL and its importance to understanding and maintaining code. You should be able to identify the key features of structured programming, and understand the general concept of the best practices regarding the structure of the Procedure Division, including the design and content of paragraphs, program control options and ways to call other programs within the same system.

11.7 Lab

In this Lab you will compare the program structure of COBOL member CBL0003 with the previous two programs, CBL0001 and CBL0002.

1. Open and view the COBOL member CBL0003.
2. Open and compare the source code of CBL0001 and CBL0002 with CBL0003. You will notice the program structure varies between the three programs.
3. Compile CBL0003 and submit job CBL0003J.
4. View the program output, it should look the same as the ones from CBL0001 and CBL0002. The different program structure doesn’t affect the output, the three programs produce the same output PRTLINE file.

Chapter 12 – File Output

In order to format output, there must be a structured layout that is easy to read and understand. Designing a structured layout involved column headings and variable alignment using spaces, numeric format, currency format etc. This chapter aims to explain this concept using example COBOL code to design column headings and align data names under such headings. At the end of the chapter there is a lab which practices implementation of the components covered.

A capability of COBOL that is not covered in this chapter is that COBOL is a web enabled computer language. To learn more, you can visit the Micro Focus Web Services Support or Native and Managed COBOL

12.1 Review of COBOL Write Output Services

This section briefly reviews certain aspects of the ENVIRONMENT DIVISION for the purpose of understanding how it ties together with the content of this chapter.

12.1.1 Environment Division

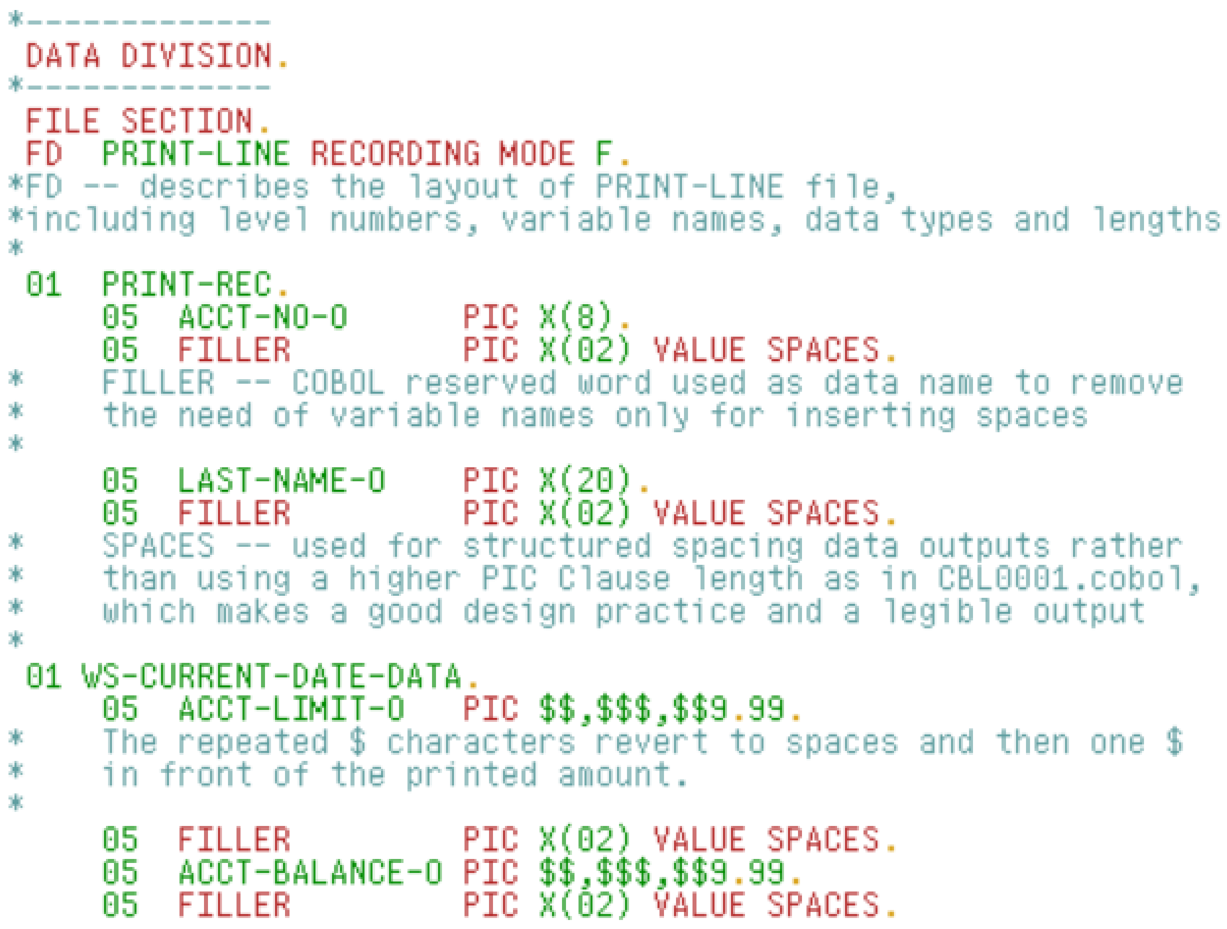
You will remember from earlier chapters that the Environment Division plays an important role when it comes to file handling in a COBOL program. The 'File Handling' section covered the SELECT and ASSIGN programmer chosen names. This section, however, focuses on output.

12.2 File Descriptor

The File Description (FD) entry represents the highest level of organization in the FILE SECTION. The FD entry describes the layout of the file defined by a previous FILE-CONTROL SELECT statement. Therefore, the FD entry connects the SELECT file name with a defined layout of the file name.

12.2.1 Filler

Observe the data name FILLER. While most data fields have unique names, FILLER is a COBOL reserved word data name that is useful for output formatting. This is partly because FILLER allocates memory space without the need for a name. Additionally, FILLER allocated memory has a defined length in the output line and may contain spaces or any literal. Look at *Figure 2* below:



```
*-------------
 DATA DIVISION.
*-------------
 FILE SECTION.
 FD  PRINT-LINE RECORDING MODE F.
*FD -- describes the layout of PRINT-LINE file,
*including level numbers, variable names, data types and lengths
*
 01  PRINT-REC.
     05  ACCT-NO-O      PIC X(8).
     05  FILLER         PIC X(02) VALUE SPACES.
*    FILLER -- COBOL reserved word used as data name to remove
*    the need of variable names only for inserting spaces
*
     05  LAST-NAME-O    PIC X(20).
     05  FILLER         PIC X(02) VALUE SPACES.
*    SPACES -- used for structured spacing data outputs rather
*    than using a higher PIC Clause length as in CBL0001.cobol,
*    which makes a good design practice and a legible output
*
 01 WS-CURRENT-DATE-DATA.
     05  ACCT-LIMIT-O   PIC $$,$$$,$$9.99.
*    The repeated $ characters revert to spaces and then one $
*    in front of the printed amount.
*
     05  FILLER         PIC X(02) VALUE SPACES.
     05  ACCT-BALANCE-O PIC $$,$$$,$$9.99.
     05  FILLER         PIC X(02) VALUE SPACES.
```

Figure 2 shows multiple VALUE SPACES for FILLER. SPACES create white space between data-items in the output which is valuable in keeping the code readable. More specifically in *Figure 2* FILLER PIC X(02) VALUE SPACES represents the output line containing two spaces.

12.3 Report and Column Headers

Writing report of column headers requires a structured output layout designed by the programmer, as shown in *Figure 3*. The designed output structure layout is implemented within the Data Division and includes the headers listed and defined below:

HEADER-1:

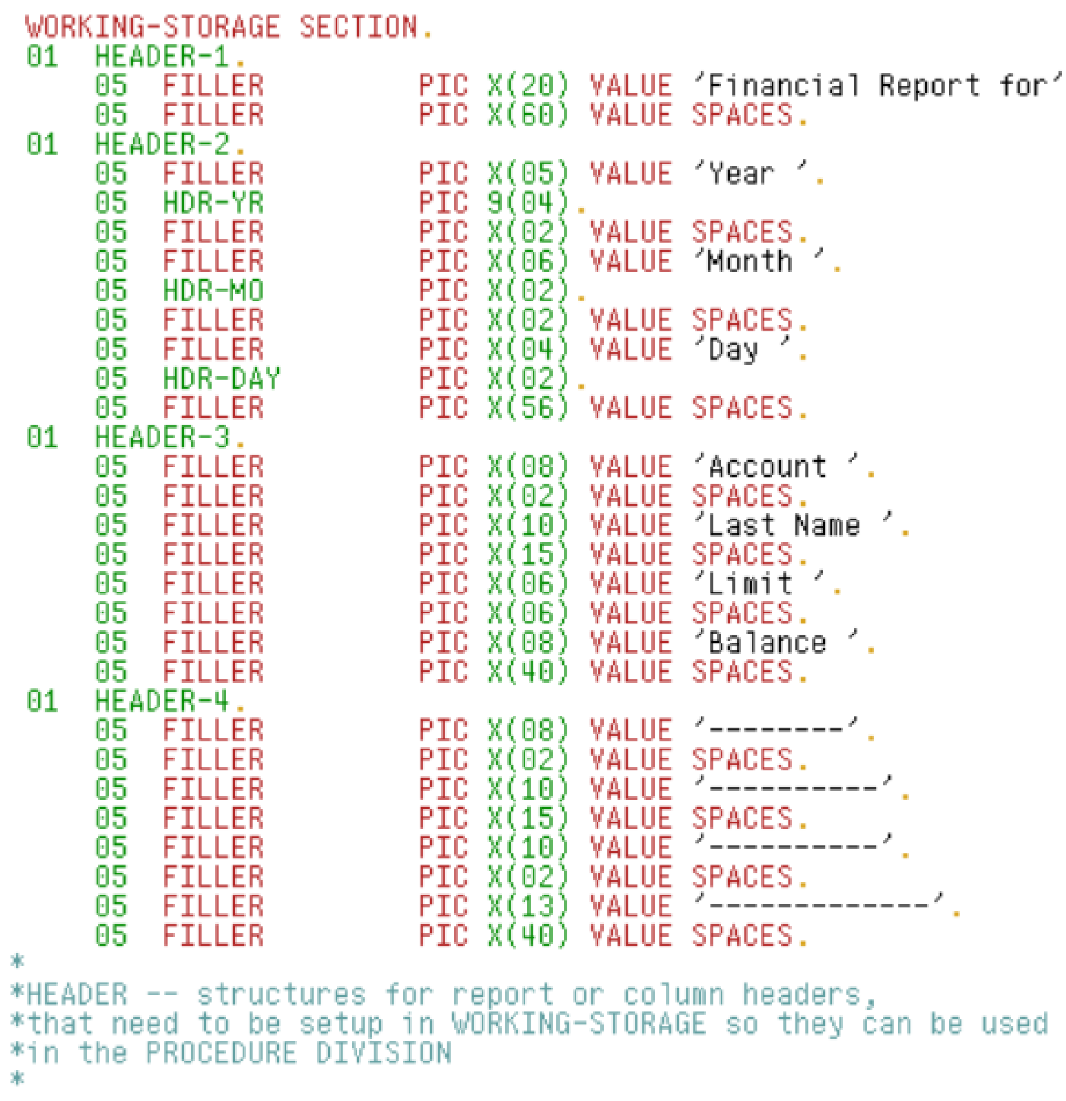
- Writes a literal
- Example: Financial Report for'

HEADER-2:

- Writes literals
- Examples:
    - 'Year' followed by a variable name
    - 'Month' followed by a variable name
    - 'Day' followed by a variable name

HEADER-3:

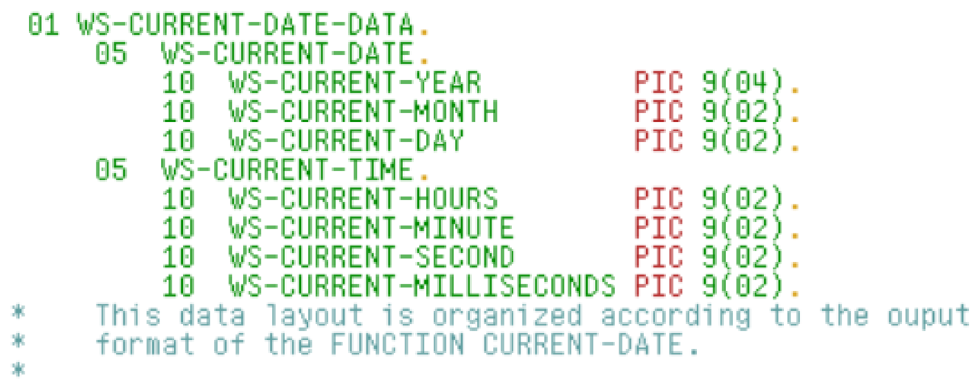
- Writes literals
- Examples:
    - 'Account' followed by FILLER spacing
    - 'Last Name' followed by FILLER spacing
    - 'Limit' followed by FILLER spacing
    - 'Balance' followed by FILLER spacing



```
 WORKING-STORAGE SECTION.
 01  HEADER-1.
     05  FILLER          PIC X(20) VALUE 'Financial Report for'.
     05  FILLER          PIC X(60) VALUE SPACES.
 01  HEADER-2.
     05  FILLER          PIC X(05) VALUE 'Year '.
     05  HDR-YR          PIC 9(04).
     05  FILLER          PIC X(02) VALUE SPACES.
     05  FILLER          PIC X(06) VALUE 'Month '.
     05  HDR-MO          PIC X(02).
     05  FILLER          PIC X(02) VALUE SPACES.
     05  FILLER          PIC X(04) VALUE 'Day '.
     05  HDR-DAY         PIC X(02).
     05  FILLER          PIC X(56) VALUE SPACES.
 01  HEADER-3.
     05  FILLER          PIC X(08) VALUE 'Account '.
     05  FILLER          PIC X(02) VALUE SPACES.
     05  FILLER          PIC X(10) VALUE 'Last Name '.
     05  FILLER          PIC X(15) VALUE SPACES.
     05  FILLER          PIC X(06) VALUE 'Limit '.
     05  FILLER          PIC X(06) VALUE SPACES.
     05  FILLER          PIC X(08) VALUE 'Balance '.
     05  FILLER          PIC X(40) VALUE SPACES.
 01  HEADER-4.
     05  FILLER          PIC X(08) VALUE '--------'.
     05  FILLER          PIC X(02) VALUE SPACES.
     05  FILLER          PIC X(10) VALUE '----------'.
     05  FILLER          PIC X(15) VALUE SPACES.
     05  FILLER          PIC X(10) VALUE '----------'.
     05  FILLER          PIC X(02) VALUE SPACES.
     05  FILLER          PIC X(13) VALUE '-------------'.
     05  FILLER          PIC X(40) VALUE SPACES.
*
*HEADER -- structures for report or column headers,
*that need to be setup in WORKING-STORAGE so they can be used
*in the PROCEDURE DIVISION
*
```

Figure 3

HEADER-2 includes the year, month and day of the report together with FILLER area, creating blank spaces between them as you can see in *Figure 3*. *Figure 4* however is an example of the data name layout used to store the values of CURRENT-DATE. The information COBOL provides in CURRENT-DATE is used to populate the output file in HEADER-2.

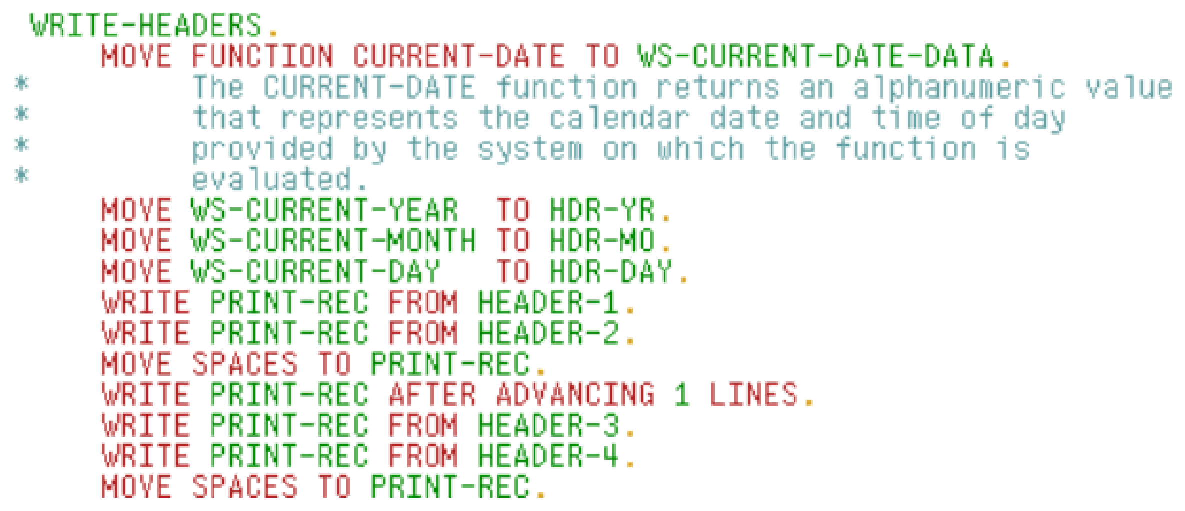


```
  01 WS-CURRENT-DATE-DATA.
     05  WS-CURRENT-DATE.
         10  WS-CURRENT-YEAR         PIC 9(04).
         10  WS-CURRENT-MONTH        PIC 9(02).
         10  WS-CURRENT-DAY          PIC 9(02).
     05  WS-CURRENT-TIME.
         10  WS-CURRENT-HOURS        PIC 9(02).
         10  WS-CURRENT-MINUTE       PIC 9(02).
         10  WS-CURRENT-SECOND       PIC 9(02).
         10  WS-CURRENT-MILLISECONDS PIC 9(02).
 *   This data layout is organized according to the ouput
 *   format of the FUNCTION CURRENT-DATE.
 *
```

Figure 4

12.4 Procedure Division

Figures 1 through 4 are a designed data layout that includes a data line and report headers. Using the storage mapped by the data line and report headers, COBOL processing logic can write the headers followed by each data line. *Figure 5* is an example of an execution logic used to write the header layout structure in a COBOL program.



```
 WRITE-HEADERS.
     MOVE FUNCTION CURRENT-DATE TO WS-CURRENT-DATE-DATA.
*         The CURRENT-DATE function returns an alphanumeric value
*         that represents the calendar date and time of day
*         provided by the system on which the function is
*         evaluated.
     MOVE WS-CURRENT-YEAR  TO HDR-YR.
     MOVE WS-CURRENT-MONTH TO HDR-MO.
     MOVE WS-CURRENT-DAY   TO HDR-DAY.
     WRITE PRINT-REC FROM HEADER-1.
     WRITE PRINT-REC FROM HEADER-2.
     MOVE SPACES TO PRINT-REC.
     WRITE PRINT-REC AFTER ADVANCING 1 LINES.
     WRITE PRINT-REC FROM HEADER-3.
     WRITE PRINT-REC FROM HEADER-4.
     MOVE SPACES TO PRINT-REC.
```

Figure 5

12.4.1 MOVE sentences

The COBOL MOVE statement on line 1 (in the WRITE-HEADERS paragraph) is collecting the current date information from the system and storing that information in a defined data name layout, WS-CURRENT-DATE-DATA. The use of the reserved word FUNCTION means that whatever follows is a COBOL intrinsic function. The sentences on lines 2, 3 and 4 are storing the date information- year, month and day- in HEADER-2 defined data name areas, HDR-YR, HDR-MO and HDR-DAY. The final sentence in the paragraph writes spaces into the PRINT-REC area to clear out the line storage in preparation for writing the data lines.

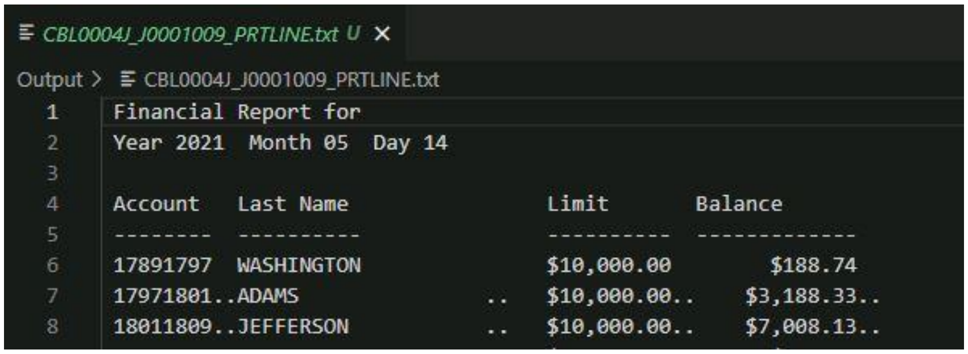
12.4.2 PRINT-REC FROM sentences

PRINT-REC is opened for output resulting in PRINT-REC FROM following through with a write PRINT-REC FROM a different header or defined data layout. The sentences on lines 5 and 6 write the PRINT-REC FROM defined header data names (HEADER-1 and HEADER-2) from *Figure 3*. The PRINT-REC file descriptor data names in *Figure 2* are effectively replaced with the content of the header data names in *Figure 3* written to output. The sentences on lines 7 and 8 result in a blank line written between headers. The sentences on lines 9 and 10 write the PRINT-REC FROM defined HEADER-3 and HEADER-4 data names from *Figure 3*. The PRINT-REC file descriptor data names in *Figure 2* are effectively replaces with the content of the header data names in *Figure 3.*

12.5 Lab

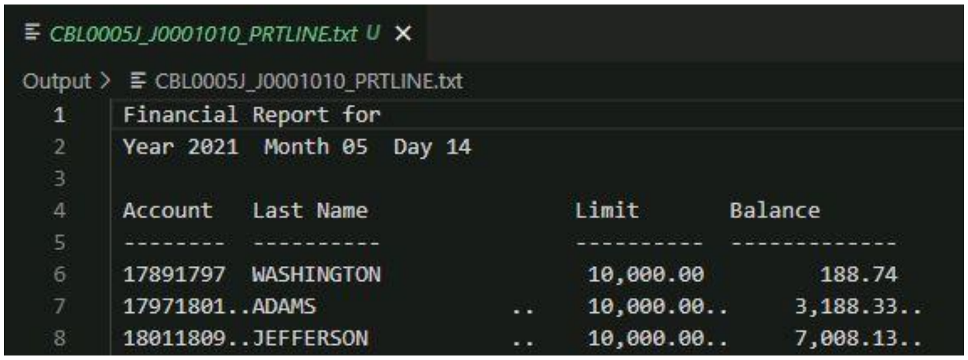
In this lab, you will compare the program output from COBOL members CBL0004 and CBL0005 and explore the different displays of the PIC clauses within the source code.

1. Compile COBOL member CBL0004 and run job CBL0004J.
2. Look at the program output. You will notice the report now has a header, as you can see in the CBL0004 source code.



```
CBL0004J_J0001009_PRTLINE.txt U ×
Output > CBL0004J_J0001009_PRTLINE.txt
Financial Report for
Year 2021  Month 05  Day 14

Account   Last Name                Limit       Balance
--------  ----------               ----------  -------------
17891797  WASHINGTON               $10,000.00        $188.74
17971801..ADAMS                 .. $10,000.00..    $3,188.33..
18011809..JEFFERSON             .. $10,000.00..    $7,008.13..
```

3. Compile COBOL member CBL0005 and run job CBL0005J.
4. Compare the program output from CBL0005 and CBL0004. Notice that the first is missing the currency sign ($) in front of all the numeric fields.

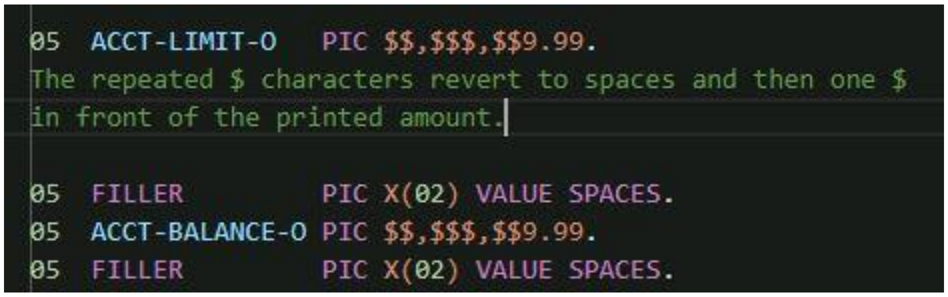


```
CBL0005J_J0001010_PRTLINE.txt U ×
Output > CBL0005J_J0001010_PRTLINE.txt
Financial Report for
Year 2021  Month 05  Day 14

Account   Last Name                Limit       Balance
--------  ----------               ----------  -------------
17891797  WASHINGTON                10,000.00         188.74
17971801..ADAMS                 ..  10,000.00..     3,188.33..
18011809..JEFFERSON             ..  10,000.00..     7,008.13..
```

5. Open CBL0005 and fix the source code to have the output file include the currency sign. *
6. Save, recompile CBL0005 and resubmit job CBL0005J.
7. Download and view the new CBL0005 output. It should now include the currency sign in front of each numeric field.

Hint:

1. Compare line 25 between CBL0004 and CBL0005.



```
05  ACCT-LIMIT-O   PIC $$,$$$,$$9.99.
The repeated $ characters revert to spaces and then one $
in front of the printed amount.

05  FILLER         PIC X(02) VALUE SPACES.
05  ACCT-BALANCE-O PIC $$,$$$,$$9.99.
05  FILLER         PIC X(02) VALUE SPACES.
```

13 Conditional Expressions

This chapter explores how programs make decisions based upon the logic written by the programmer. Specifically, programs make these decisions within the PROCEDURE DIVISION of the source code. We will expand on several topics regarding conditional expressions written in COBOL, with a supplementary lab towards the end.

13.1 Boolean logic, operators, operands and identifiers

Program decisions are made using Boolean logic where a conditional expression is either true or false, yes or no.

A simple example could be a variable named 'LANGUAGE'. Many programming languages exist; therefore, the value of variable LANGUAGE could be Java, COBOL etc. Assume the value of LANGUAGE is COBOL. Boolean logic is:

IF LANGUAGE = COBOL, THEN DISPLAY "COBOL" ELSE DISPLAY "NOT COBOL".

IF triggers the Boolean logic to determine the condition of true/false, yes/no, applied to LANGUAGE = COBOL. This is known as a conditional expression.

The Boolean IF verb operates on two operands or identifiers. In the example above, LANGUAGE and COBOL are both operands.

13.1.1 COBOL conditional expressions and operators

Three of the most common type of COBOL conditional expressions are:

- General relation condition
- Class condition
- Sign condition

A list of COBOL Boolean relational operators for each of the common type of COBOL conditional expressions are represented in figures 1, 2 and 3 below.

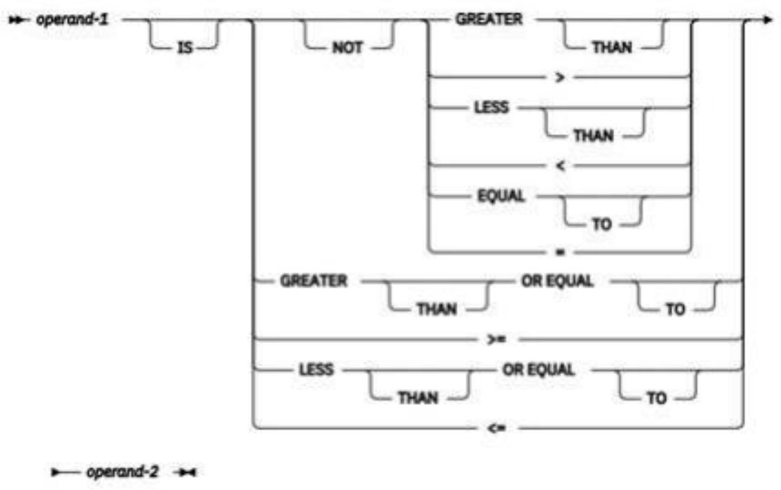


Figure 1. General relation condition operators

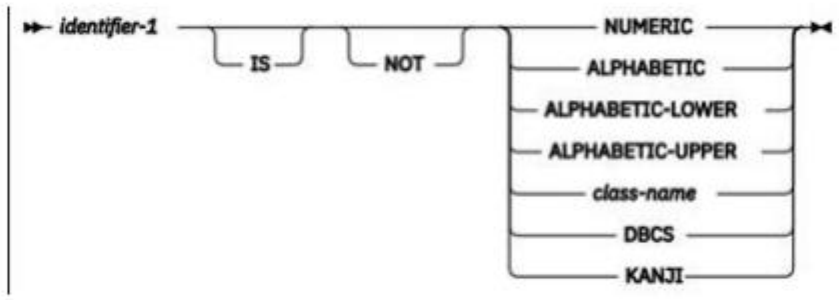


Figure 2. Class condition operators

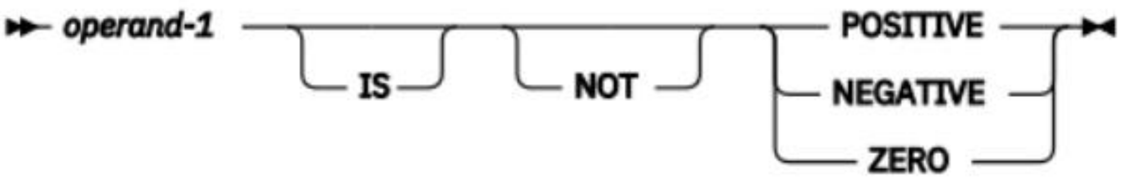


Figure 3. Sign condition operators

13.1.2 Examples of conditional expressions using Boolean operators

A simple conditional expression can be written as:

IF 5 > 1 THEN DISPLAY ‘5 is greater than 1’ ELSE DISPLAY ‘1 is greater than 5’.

Compounded conditional expressions are enclosed in parenthesis and their Boolean operators are:
AND
OR

The code snippet below demonstrates a compounded conditional expression using the AND Boolena operator:

IF (5 > 1 AND 1 > 2)THEN …… ELSE ……

The conditional expression evaluates to false because while 5 > 1 is true, 1 > 2 is false. The AND operation requires both expressions to be true to return true for the entire compounded condition expression. More conditional operators used for relation, class and sign conditions are discussed further on in the chapter.

13.2 Conditional expression reserved words and terminology

So far on this training course we have touched upon the necessity and use of COBOL reserved words. This section expands on the reserved words used when processing conditional statements.

13.2.1 IF, EVALUATE, PERFORM and SEARCH

These are COBOL reserved words available for the processing of conditional expressions, where a condition is a state that can be set or changed.

13.2.2 Conditional states

TRUE and FALSE are among the most common conditional states.

13.2.3 Conditional names

A conditional-name is a variable name defined by the programmer with the TRUE condition state. In COBOL, a conditional name is declared in the WORKING STORAGE SECTION with an 88-level number. The purpose of the 88-level is to improve readability by simplifying IF and PERFORM UNTIL statements.

The 88-level conditional data-name is assigned a value at compile time, and cannot be changed by the program during execution. However, the program can change the data name value in the level number above the 88-level conditional data-name.

Have a look at the following example:

```
WORKING-STORAGE.
01 USA-STATE      PIC X(2) VALUE SPACES.
   88 STATE       VALUE 'TX'.
....
....
PROCEDURE DIVISION.
....
....
MOVE 'AZ' TO USA-STATE.
....
....
IF STATE DISPLAY 'The State is Texas'
   ELSE DISPLAY  'The State is not Texas'
END-IF.
....
....
```

```
MOVE 'TX' TO USA-STATE.
....
....
IF STATE DISPLAY 'The State is Texas'
   ELSE DISPLAY  'The State is not Texas'
END-IF.
```

Example 1

In the first instance, the program will write ‘The State is Not Texas. This is because the program moved ‘AZ’ to the value USA-STATE, which is not equal to the 88-level conditional data-name, TX. The second IF STATE writes ‘The State is Texas’ because the value of USA-STATE is equal to the assigned 88-level value of TX.

Numerous 88-level conditional data-names can follow a 01-level data-name. As a result, an IF reference to 01-level data-name expression can have numerous values that would return true.

Other level number data-names require the condition expression to include a Boolean operator as shown in *Example 2*, where a value can be stored in the 05-level STATE data-name to be compared with some other stored value. Therefore, a little extra coding is needed.

```
WORKING-STORAGE.
01 USA-STATE.
   05 STATE       PIC X(2) VALUE SPACES.
....
....
PROCEDURE DIVISION.
....
....
MOVE 'AZ' TO STATE.
....
....
IF STATE = 'TX' DISPLAY 'The State is Texas'
   ELSE DISPLAY  'The State is not Texas'
END-IF.
....
....
MOVE 'TX' TO STATE.
....
....
IF STATE = 'TX' DISPLAY 'The State is Texas'
   ELSE DISPLAY  'The State is not Texas'
END-IF.
```

Example 2

13.3 Conditional operators

Relational operators compare numeric, character string, or logical data. The result of the comparison, either true (1) or false (0), can be used to make a decision regarding program flow. *Table 1* displays a list of relational operators, how they can be written, and their meaning.

13.4 Conditional expressions

A conditional expression causes the object program to select alternative paths of control, depending on the truth value of a test. Conditional expressions are specified in EVALUATE, IF, PERFORM and SEARCH statements.

13.4.1 IF ELSE (THEN) statements

IF statements are used to Implement or evaluate relational operations. IF ELSE is used to code a choice between two processing actions. The inclusion of the word THEN is optional. When an IF statement is present, the statements following the IF statement are then processes based on the truth of the conditional expression. Statements are processed until and END-IF or an ELSE statement is encountered. The ELSE statement can appear on any line before the END-IF. IF statements, regardless of how many lines, are explicitly terminated by using END-IF. This is demonstrated in *Example 3*.

```
IF FACIAL-EXP = 'HAPPY' THEN
   DISPLAY 'I am glad you are happy'
ELSE DISPLAY 'What can I do to make you happy'
END-IF.
```

Example 3

13.4.2 EVALUATE statements

EVALUATE statements are used to code a choice among three or more possible actions. The explicit terminator for an EVALUATE statement is END-EVALUATE. The EVALUATE statement is an expanded form of the IF statement which allows you to avoid nesting IF statements, which is a common source of logic errors and debugging issues. EVALUATE operates on both text string values and numerical variables. Using the same FACIAL-EXP conditional-name as before, have a look at the COBOL code implementing an EVALUATE statement in *Example 4*.

```
EVALUATE FACIAL-EXP
 WHEN 'HAPPY'
  DISPLAY 'I am glad you are happy'
 WHEN 'SAD'
  DISPLAY 'What can I do to make you happy'
 WHEN 'PERPLEXED'
  DISPLAY 'Can you tell me what you are confused about'
 WHEN 'EMOTIONLESS'
  DISPLAY 'Do you approve or disapprove'
END-EVALUATE
```

Example 4

13.4.3 PERFORM statements

A PERFORM with UNTIL phrase is a conditional expression. In the UNTIL phrase format, the procedures referred to are performed until the condition specified by the UNTIL phrase evaluates to be true. Using the FACIAL-EXP conditional-name, the SAY-SOMETHING-DIFFERENT paragraph is executed continuously until FACIAL-EXP contains 'HAPPY', as shown in *Example 5*.

```
PERFORM SAY-SOMETHING-DIFFERENT BY FACIAL-EXP UNTIL 'HAPPY'
END-PERFORM.
```

13.4.4 SEARCH statements

The SEARCH statement searches a table for an element that satisfies the specified condition. Tables are created with an OCCURS clause applied to WORKING STORAGE data-names. A WHEN clause is used in SEARCH statements to verify if the element searched for satisfies the specified condition. Assuming FACIAL-EXP has many possible values, then SEARCH WHEN is an alternative conditional expression. Observe *Example 6.*

```
SEARCH FACIAL-EXP
WHEN 'HAPPY' STOP RUN
END-SEARCH
```

Example 6

13.5 CONDITIONS

A conditional expression can be specified in either simple conditions or complex conditions. Both simple and complex can be enclosed within any number of paired parentheses, but do not change whether the condition is simple or complex. This section will cover three out of the five simple conditions:

- Relation
- Class

- Sign

13.5.1 Relation Conditions

A relation condition specifies the comparison of two operands. The relational operator that joins the two operands specifies the type of comparison. The relation condition is true if the specified relation exists between the two operands; the relation condition is false if the specified relation does not exist. Here is a list of a few defined comparisons:

- Numeric comparisons – two operands of a class numeric
- Alphanumeric comparisons – two operands of a class alphanumeric
- DBCS (Double Byte Character Set) comparisons – two operands of DBCS
- National comparisons – two operands of a class national

13.5.2 Class conditions

The class condition determines whether the content of a data item is alphabetic, alphabetic-lower, alphabetic-upper, numeric, DBCS, KKANJI or contains only the characters in the set of characters specified by the CLASS cause, as defined in the SPECIAL-NAMES paragraph of the environment division. Provided below is a list of a few valid forms on the class condition for different types of data.

- Numeric
    - IS NUMERIC or IS NOT NUMERIC
- Alphabetic
    - IS ALPHABTEIC or IS NOT ALPHABTEIC
    - IS ALPHABETIC-LOWER / ALPHABETIC-UPPER
    - IS NOT ALPHABETIC-LOWER / ALPHABETIC-UPPER
- DBCs
    - IS DBCS or IS NOT DBCS
    - IS KANJI or IS NOT KANJI

13.5.3 Sign conditions

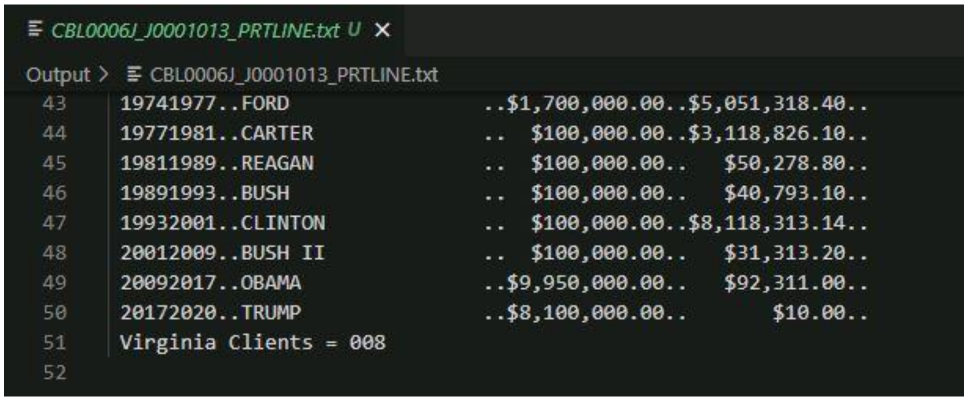
The sign condition determines whether the algebraic value of a numeric operand is greater than, less than or equal to zero. A unsigned operand is either POSITIVE or ZERO. When a numeric conditional variable is defined with a sign, the following are available:

- IS POSITIVE
- IS NEGATIVE
- IS ZERO

13.6 Lab

In this Lab, you will explore conditional clauses within COBOL members CBL0006 and CBL0007.

1. Open and view programs CBL0005 and CBL0006. Notice the new working storage member CLIENTS-PER-STATE, and the new paragraph IS-STATE-VIRGINIA. In this paragraph, the program checks whether the client is from Virginia and then writes the total amount of clients from Virginia in the report.
2. Compile CBL0006 and submit job CBL0006J.
3. View the program output and check the last line of the report as shown below.

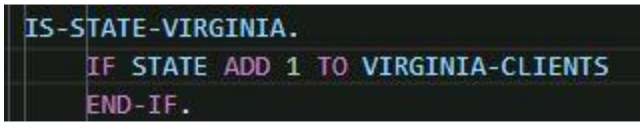


```
CBL0006J_J0001013_PRTLINE.txt U ×
Output > CBL0006J_J0001013_PRTLINE.txt
19741977..FORD                    ..$1,700,000.00..$5,051,318.40..
19771981..CARTER                  ..  $100,000.00..$3,118,826.10..
19811989..REAGAN                  ..  $100,000.00..   $50,278.80..
19891993..BUSH                    ..  $100,000.00..   $40,793.10..
19932001..CLINTON                 ..  $100,000.00..$8,118,313.14..
20012009..BUSH II                 ..  $100,000.00..   $31,313.20..
20092017..OBAMA                   ..$9,950,000.00..   $92,311.00..
20172020..TRUMP                   ..$8,100,000.00..       $10.00..
Virginia Clients = 008
```

4. Now compile CBL0007.
5. Notice the error in the Problems tab and fix the error in the source code of CBL0007. *
6. Save, recompile CBL0007 and submit job CBL0007J.
7. Look at the output and check it looks like the one from CBL0006.

Hint:

1. The error is located at line 146. Compare with the source code of CBL0007.



```
IS-STATE-VIRGINIA.
     IF STATE ADD 1 TO VIRGINIA-CLIENTS
     END-IF.
```

Chapter 14 – Arithmetic Expressions

This chapter introduces the concept of implementing arithmetic expressions in COBOL programs. We will review the basic concept of arithmetic expressions, operators, statements, limitations, statement operands. Following the chapter is a lab where you can practice the implementation of what you have learned.

14.1 What is an arithmetic expression?

Arithmetic expressions are used as operands of certain conditional and arithmetic statements. An arithmetic expression can consist of any of the following items:

1. An identifier described as a numeric elementary item (including numeric functions)
2. A numeric literal
3. The figurative constant ZERO
4. Identifiers and literals, as defined in items 1, 2 and 3, separated by arithmetic operators
5. Two arithmetic expressions, as defined in items 1, 2, 3 or 4, separated by an arithmetic operator
6. An arithmetic expression, as defined in items 1, 2, 3, 4 or 5, enclosed in parentheses
7. Any arithmetic expression can be preceded by a unary operator

Identifiers and literals that appear in arithmetic expressions must represent either numeric elementary items or numeric literals on which arithmetic can be performed. If the value of an expression to be raised to a power is zero, the exponent must have a greater value than zero. Otherwise, the size error condition exists. In any case where no real number exists as the result of an evaluation, the size error condition exists.

14.1.1 Arithmetic operators

Five binary arithmetic operators and two unary arithmetic operators can be used in arithmetic expressions. These operators are represented by specific characters that must be preceded and followed by a space. However, no space is required between a left parentheses and unary operator. These binary and unary arithmetic operators are listed in *Table 1*.

Binary operator	Meaning	Unary operator	Meaning
+	Addition	+	Multiplication by +1
-	Subtraction	-	Multiplication by -1
*	Multiplication		
/	Division		
**	Exponentiation		

14.1.2 Arithmetic statements

Arithmetic statements are used for computations. Individual operations are specified by the ADD, SUBTRACT, MULTIPLY and DIVIDE statements. These individual operations can be combined symbolically in a formula that uses the COMPUTE statement for ease of programming and performance. It is considered best practice to use the COMPUTE statement rather than the separate arithmetic statements written in a series because the COMPUTE statement, arithmetic statements can be combined with fewer restrictions.

14.2 Arithmetic expression precedence rules

Order of operation rules have been hammered into your head throughout the years of learning mathematics, remember the classic PEMDAS (parentheses, exponents, multiply, divide, add, subtract)? Arithmetic expressions in COBOL are not exempt from these rules and often use parentheses to specify the order in which elements are to be evaluated.

14.2.1 Parentheses

Parentheses are used to denote modifications to normal order of operations (precedence rules). An arithmetic expression within the parentheses is evaluated first and result is used in the rest of the expression. When expressions are contained within nested parentheses, evaluation proceeds from the least inclusive to the most inclusive set. That means you work from the inner most expression within parentheses to the outer most. The precedence for how to solve an arithmetic expression in Micro Focus Enterprise Developer with parentheses is:

1. Parentheses (simplify the expression inside them)
2. Unary operator
3. Exponents
4. Multiplication and division (from left to right)
5. Addition and subtraction (from left to right)

Parentheses either eliminate ambiguities in logic where consecutive operations appear at the same hierarchic level or modify the normal hierarchic sequence of execution when necessary. When the order of consecutive operations at the same hierarchic level is not completely specified by parentheses, the order is from left to right.

An arithmetic expression can begin only with a left parenthesis, a unary operator, or an operand (that is, an identifier or a literal). It can end only with a right parenthesis or an operand. An arithmetic expression must contain at least one reference to an identifier or a literal.

There must be a one-to-one correspondence between left and right parentheses in an arithmetic expression, with each left parenthesis placed to the left of its corresponding right parenthesis. If the first operator in an arithmetic expression is a unary operator, it must be immediately preceded by a left parenthesis if that arithmetic expression immediately follows an identifier or another arithmetic expression.

14.3 Arithmetic expression limitations

Exponents in fixed-point exponential expressions cannot contain more than nine digits. The compiler will truncate any exponent with more than nine digits. In the case of truncation, the compiler will issue a diagnostic message if the exponent is a literal or constant; if the exponent is a variable or data-name, a diagnostic message is issued at run-time.

Detailed explanation of fixed-point exponential expressions is an advanced topic and beyond the scope of the chapter. However, reference is made to fixed-point exponential expressions for your awareness as you advance your experience level with COBOL programming and arithmetic applied to internal data representations.

14.4 Arithmetic statement operands

The data descriptions of operands in an arithmetic statement don't need to be the same. Throughout the calculation, the compiler performs any necessary data conversion and decimal point alignment.

14.4.1 Size of operands

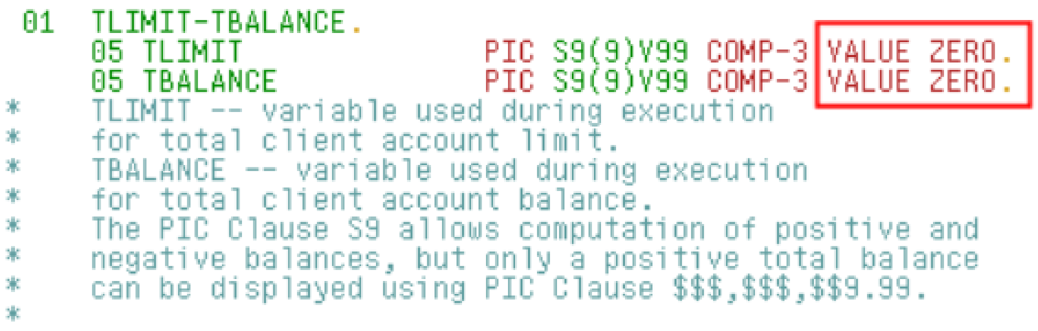
If the ARITH(COMPAT) compiler option is in effect, the maximum size of each operand is 18 decimal digits.
If the ARITH(EXTEND) compiler option is in effect, the maximum size of each operand is 31 decimal digits.

Statement	Determination of the composite of operands
SUBTRACT, ADD	Superimposing all operands in a given statement, except those following the word GIVING.
MULTIPLY	Superimposing all receiving data-items
DIVIDE	Superimposing all receiving data items except the REMAINDER data-item
COMPUTE	Restriction does not apply

In all arithmetic statements, it is important to define data with enough digits and decimal places to ensure the required accuracy in the result.

14.5 Examples of COBOL arithmetic statements

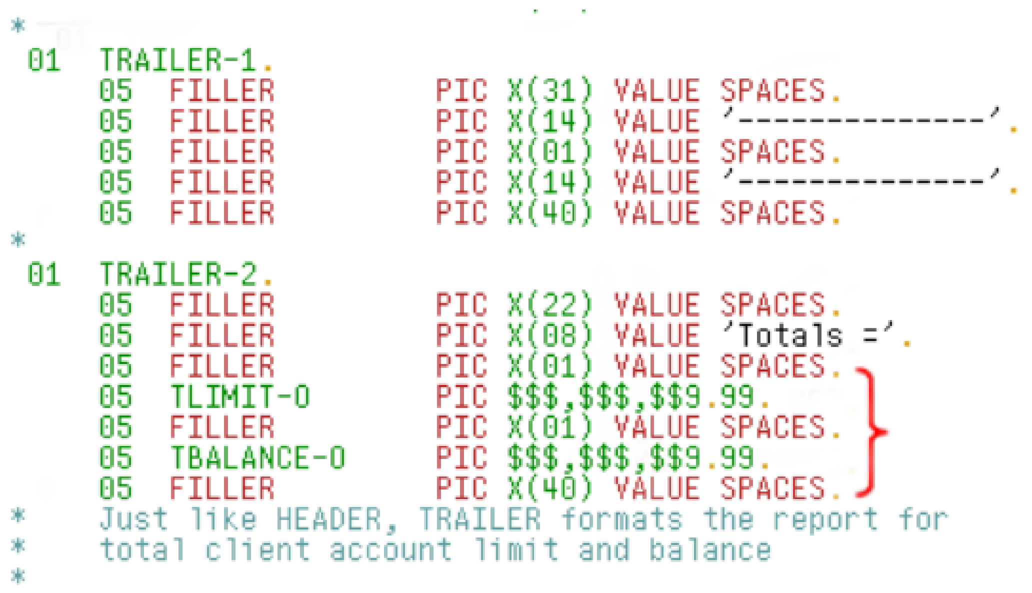
In this section, COBOL source code used in previous labs will be modified to demonstrate arithmetic processing. *Figure 1* shows level number data-items will be used to total client account limit and client account balance. Observe that the initial value is ZERO.



```
 01   TLIMIT-TBALANCE.
      05 TLIMIT                PIC S9(9)V99 COMP-3 VALUE ZERO.
      05 TBALANCE              PIC S9(9)V99 COMP-3 VALUE ZERO.
*     TLIMIT -- variable used during execution
*     for total client account limit.
*     TBALANCE -- variable used during execution
*     for total client account balance.
*     The PIC Clause S9 allows computation of positive and
*     negative balances, but only a positive total balance
*     can be displayed using PIC Clause $$$,$$$,$$9.99.
*
```

Figure 1. Number level data-items

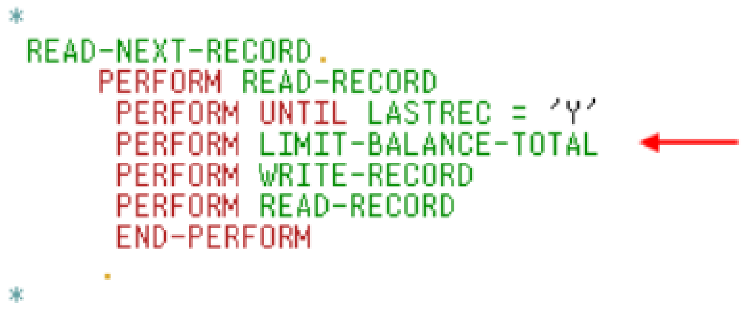
Shown in Figure 2 is another example of number level data-items in the WORKING-STORAGE section. These data-items are report trailer lines that are used to write a formatted total account limit and total account balance for all clients in the report. Observe the TLIMIT and TBALANCE data-items with large currency number picture clauses.



```
 *
  01  TRAILER-1.
      05  FILLER          PIC X(31) VALUE SPACES.
      05  FILLER          PIC X(14) VALUE '--------------'.
      05  FILLER          PIC X(01) VALUE SPACES.
      05  FILLER          PIC X(14) VALUE '--------------'.
      05  FILLER          PIC X(40) VALUE SPACES.
 *
  01  TRAILER-2.
      05  FILLER          PIC X(22) VALUE SPACES.
      05  FILLER          PIC X(08) VALUE 'Totals ='.
      05  FILLER          PIC X(01) VALUE SPACES.
      05  TLIMIT-O        PIC $$$,$$$,$$9.99.
      05  FILLER          PIC X(01) VALUE SPACES.
      05  TBALANCE-O      PIC $$$,$$$,$$9.99.
      05  FILLER          PIC X(40) VALUE SPACES.
 *    Just like HEADER, TRAILER formats the report for
 *    total client account limit and balance
 *
```

Figure 2. Number level data-items

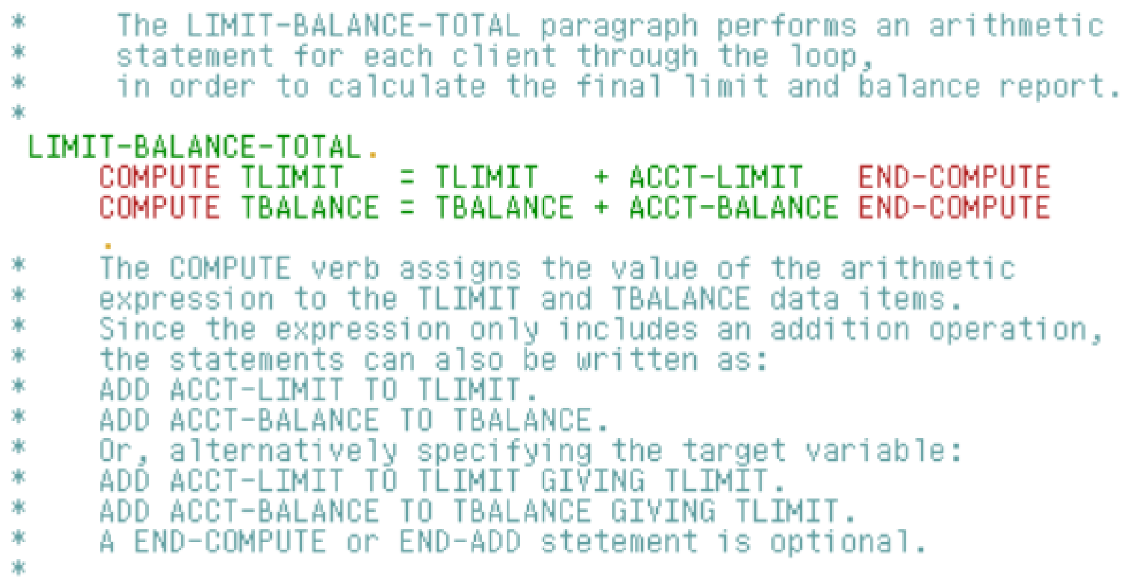
In *Figure 3* the READ-NEXT-RECORD paragraph, located within the PROCEDURE DIVISION, includes a PERFORM LIMIT-BALANCE-TOTAL statement. The result of this statement is to transfer control to the LIMIT-BALANCE-TOTAL paragraph, located within the PROCEDURE DIVISION, to perform the COMPUTE statements.



```
 *
  READ-NEXT-RECORD.
      PERFORM READ-RECORD
       PERFORM UNTIL LASTREC = 'Y'
       PERFORM LIMIT-BALANCE-TOTAL
       PERFORM WRITE-RECORD
       PERFORM READ-RECORD
       END-PERFORM
      .
 *
```

Figure 3 READ-NEXT-RECORD

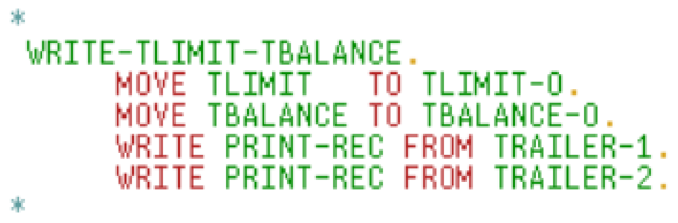
Figure 4 is an example of two COMPUTE statements in the paragraph, LIMIT-BALANCE-TOTAL. Notice that the results of the COMPUTE statements are to add client ACCT-LIMIT to the current TLIMIT and add client ACCT-BALANCE to TBALANCE totals each time the paragraph is executed, which is one time for each client record read in our example.



```
*     The LIMIT-BALANCE-TOTAL paragraph performs an arithmetic
*     statement for each client through the loop,
*     in order to calculate the final limit and balance report.
*
 LIMIT-BALANCE-TOTAL.
     COMPUTE TLIMIT   = TLIMIT   + ACCT-LIMIT   END-COMPUTE
     COMPUTE TBALANCE = TBALANCE + ACCT-BALANCE END-COMPUTE
     .
*    The COMPUTE verb assigns the value of the arithmetic
*    expression to the TLIMIT and TBALANCE data items.
*    Since the expression only includes an addition operation,
*    the statements can also be written as:
*    ADD ACCT-LIMIT TO TLIMIT.
*    ADD ACCT-BALANCE TO TBALANCE.
*    Or, alternatively specifying the target variable:
*    ADD ACCT-LIMIT TO TLIMIT GIVING TLIMIT.
*    ADD ACCT-BALANCE TO TBALANCE GIVING TLIMIT.
*    A END-COMPUTE or END-ADD stetement is optional.
*
```

Figure 4. COMPUTE statements

The WRITE-TLIMIT-TBALANCE paragraph shown in *Figure 5* is positioned within the PROCEDURE DIVISION to be executed immediately after all records are read and before the final paragraph that closes the files and terminates program execution.



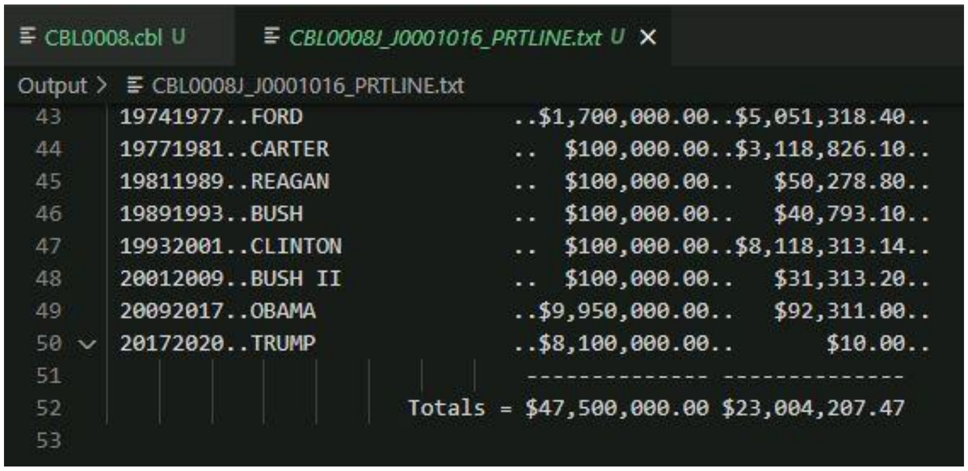
```
*
 WRITE-TLIMIT-TBALANCE.
     MOVE TLIMIT   TO TLIMIT-O.
     MOVE TBALANCE TO TBALANCE-O.
     WRITE PRINT-REC FROM TRAILER-1.
     WRITE PRINT-REC FROM TRAILER-2.
*
```

Figure 5. WRITE-TLIMIT-TBALANCE

14.6 Lab

In this Lab, you will explore arithmetic statements in COBOL and handle their correct use.

1. Open and view COBOL members CBL0008 and CBL0009.
2. Compile CBL0008 and submit job CBL0008J.
3. Look at the relative program output. Notice the new totals at the bottom of the report as shown below.

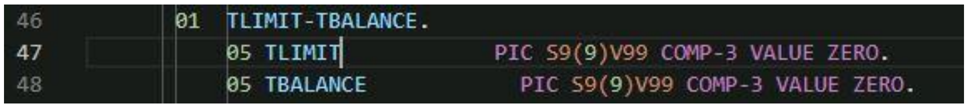

CBL0008.cbl U
CBL0008J_J0001016_PRTLINE.txt U
Output > CBL0008J_J0001016_PRTLINE.txt

```
19741977..FORD                 ..$1,700,000.00..$5,051,318.40..
19771981..CARTER               ..  $100,000.00..$3,118,826.10..
19811989..REAGAN               ..  $100,000.00..   $50,278.80..
19891993..BUSH                 ..  $100,000.00..   $40,793.10..
19932001..CLINTON              ..  $100,000.00..$8,118,313.14..
20012009..BUSH II              ..  $100,000.00..   $31,313.20..
20092017..OBAMA                ..$9,950,000.00..   $92,311.00..
20172020..TRUMP                ..$8,100,000.00..       $10.00..
                                 -------------- --------------
                        Totals = $47,500,000.00 $23,004,207.47
```

4. Compile CBL0009.
5. The build will be unsuccessful. Fix the error, save and recompile. *
6. Submit job CBL0009J and view the relative program output. It should look the same as the one from CBL0008.

Hint:

1. The error is located at line 47. Compare with the same line in CBL0008.



```
46        01  TLIMIT-TBALANCE.
47            05 TLIMIT            PIC S9(9)V99 COMP-3 VALUE ZERO.
48            05 TBALANCE            PIC S9(9)V99 COMP-3 VALUE ZERO.
```

Chapter 15 – Data Types

A COBOL programmer must be aware that the computer stored internal data representation and formatting can differ, where the difference must be defined in the COBOL source code. Understanding the computer's internal data representation requires familiarity with binary, hexadecimal, ASCII and EBCIDIC. Packed-Decimal is needed to explain COBOL Computational and Display data format. This chapter aims to familiarize the reader with these different 'types' of data representation.

- Data representation
    - Numerical value representation
    - Text representation
- COBOL DISPLAY vs COMPUTATIONAL
- Lab

15.1 Data Representation

Data such as numerical values and text are internally represented by zeros and ones in most computers, including mainframe computers used by enterprises. While data representation is a somewhat complex topic in computer science, a programmer does not always need to fully understand how various alternative representations work. It is important, however, to understand the differences and how to specify a specific representation when needed.

15.1.1 Numerical Value Representation

COBOL has five computational (numerical) value representations. The awareness of these representations is important due to two main reasons. The first reason being, when a COBOL program needs to read or write data, it needs to understand how data is represented in the dataset. The second reason is when there are specific requirements regarding the precision and range of values being processed.

15.1.1.1 COMP-1

This is also known as a single-precision floating point number representation. Due to the floating-point nature, a COMP-1 value can be very small and close to zero, or it can be very large (about 10 to the power of 38. However, a COMP-1 value has limited precision. This means that even though a COMP-1 value can be up to 10 to the power of 38, it can only maintain about seven significant decimal digits. Any value that has more than seven significant digits are rounded. This means that a COMP-1 value cannot exactly represent a bank balance like $1,234,567.89 because this value has nine significant digits. Instead, the amount is rounded. The main application of COMP-1 is for scientific numerical value storage as well as computation.

15.1.1.2 COMP-2

This is also known as a double-precision floating point number representation. COMP-2 extends the range of value that can be represented compared to COMP-1. COMP-2 can represent values up to about to about 10 to the power of 307. Like COMP-1, COMP-2 values also have a limited precision. Due to the expanded format, COMP-2 has more significant digits, approximately 15 decimal digits. This means that once a value reaches certain quadrillions (with no decimal places), it can no longer be exactly represented in COMP-2.

COMP-2 supersedes COMP-1 for more precise scientific data storage as well as computation. Note that COMP-1 and COMP-2 have limited applications in financial data representation or computation.

15.1.1.3 COMP-3
This is also known as packed BCD (binary coded decimal) representation. This is the most utilized numerical value representation in COBOL programs. Packed BCD is also somewhat unique and native to mainframe computers.

Unlike COMP-1 or COMP-2, packed BCD has no inherent precision limitation that is independent to the range of values. This is because COMP-3 is a variable width format that depends on the actual value format. COMP-3 exactly represents values with decimal places. A COMP-3 value can have up to 31 decimal digits.

15.1.1.4 COMP-4
COMP-4 is only capable of representing integers. Compared to COMP-1 and COMP-2, COMP-4 can store and compute with integer values exactly (unless a division is involved). Although COMP-3 can also be used to represent integer values, COMP-4 is more compact.

15.1.1.5 COMP-5
COMP-5 is based on COMP-4, but with the flexibility of being able to specify the position of a decimal point. COMP-5 has the space efficiency of COMP-4, and the exactness of COMP-3. Unlike COMP-3, however, a COMP-5 value cannot exceed 18 digital digits.

15.1.2 Text Representation
COBOL programs often need to represent text data such as names and addresses.

15.1.2.1 EBCIDIC
Extended Binary Coded Decimal Interchange Code (EBCDIC) is an eight-binary-digits character encoding standards, where the eight digital positions are divided into two pieces. EBCDIC is used to encode text data so that text can be printed or displayed correctly on devices that also understand EBCDIC.

15.1.2.2 ASCII
American Standard Code for Information Interchange (ASCII) is another binary digital character encoding standard.

15.1.2.3 EBCDIC vs ASCII
Why are these two standards used when they seemingly perform the same function?

EBCDIC is a standard that traces its root to punch cards designed in 1931. ASCII, on the other hand, is a standard that was created, unrelated to IBM punch cards in 1967. A COBOL program natively understands EBCDIC, and can comfortably process data originally captured in punch cards as early as 1931.

ASCII is mostly used by non-IBM computers.

COBOL can encode and process text data in EBCDIC or ASCII. This means a COBOL program can simultaneously process data captured in a census many decades ago while exporting data to a cloud service using ASCII or Unicode. It is important to point out, however, that a programmer must have the awareness and choose the appropriate encoding.

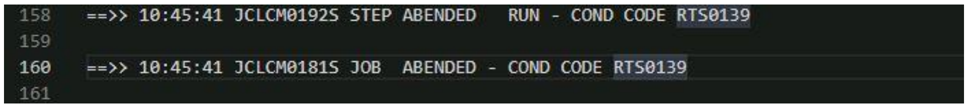
15.2 COBOL DISPLAY vs COMPUTATIONAL

The EBCDIC format representation of alphabetic characters is in a DISPLAY format. Zoned decimal for numbers, without the sign, is in a DISPLAY format. Packed decimal, binary and floating point are **not** in a DISPLAY format. COBOL can describe packed decimal, binary and floating point fields using COMPUTATIONAL, COMP-1, COMP-2, COMP-3, COMP-4 and COMP-5 reserved words.

15.3 Lab

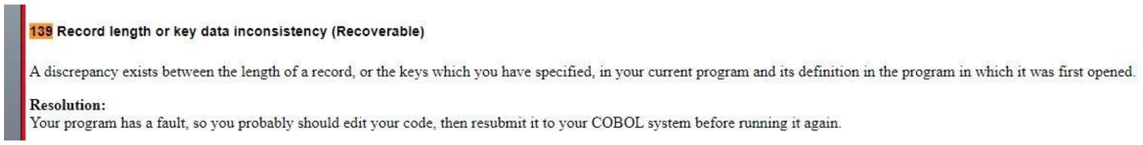
In this Lab, you will work with different types of numeric fields and their relative output record lengths.

1. Compile COBOL member CBL0010 and submit job CBL0010J.
2. The run will be unsuccessful. Download the job output and open it, then scroll to the bottom and look at the conditional code.



```
==>> 10:45:41 JCLCM0192S STEP ABENDED   RUN - COND CODE RTS0139

==>> 10:45:41 JCLCM0181S JOB  ABENDED - COND CODE RTS0139

```

3. The conditional code RTS0139 indicates an inconsistency in the record lengths, as shown in the Micro Focus Enterprise Server documentation https://www.microfocus.com/documentation/object-cobol/oc41books/emrunt.htm. Find the error in the source code of CBL0010 and fix it. *



139 Record length or key data inconsistency (Recoverable)

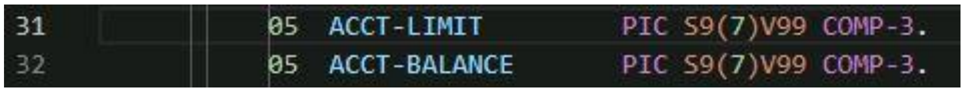
A discrepancy exists between the length of a record, or the keys which you have specified, in your current program and its definition in the program in which it was first opened.

Resolution:
Your program has a fault, so you probably should edit your code, then resubmit it to your COBOL system before running it again.

4. Save, recompile and resubmit job CBL0010J.
5. Check the job output to confirm the run was successful. Now, if you download and view the program output, the correct record length is used.

Hint:

1. The error is located at line 31.



```
31          05  ACCT-LIMIT          PIC S9(7)V99 COMP-3.
32          05  ACCT-BALANCE        PIC S9(7)V99 COMP-3.
```

Chapter 16 – Intrinsic Functions

Today's COBOL is not your parents' COBOL. Today's COBOL includes decades of feature/function rich advancements and performance improvements. Decades of industry specifications are applied to COBOL to address the growing needs of businesses. Micro Focus Enterprise Developer has evolved the DNA of COBOL into a powerful, maintainable, trusted and time-tested computer language with no end in sight.

Among the new COBOL capabilities is the JSON GENERATE and JSON PARSE, providing an easy-to-use coding mechanism to transform DATA DIVISION defined data-items into JSON for a browser, a smart phone or any other IoT device into DATA DIVISION provisioned data-items for processing. Frequently, the critical data accessed by a smart phone, such as a bank balance, is stored and controlled by Enterprise Server where a COBOL program is responsible for retrieving and returning the bank balance to the smart phone. COBOL has become a web enabled computer language.

16.1 - What is an intrinsic function?

Intrinsic functions are effectively re-usable code with simple syntax implementation and are another powerful COBOL capacity. Intrinsic functions enable desired logic processing with a simple line of code. They also provide capabilities for manipulating strings and numbers. Because the value of an intrinsic function is derived automatically at the time of reference, you do not need to define these functions in the DATA DIVISION.

16.1.1 Intrinsic Function Syntax

Written as

FUNCTION function-name (argument)

Where function-name must be one of the intrinsic function names. You can reference a function by specifying its name, along with any required arguments, in a PROCEDURE DIVISION statement. Functions are elementary data items, and return alphanumeric characters, national characters, numeric or integer values.

```
01   Item-1    Pic x(30)   Value "Hello World!".
01   Item-2    Pic x(30).
. . .
     Display Item-1
     Display Function Upper-case(Item-1)
     Display Function Lower-case(Item-1)
     Move Function Upper-case(Item-1) to Item-2
     Display Item-2
```

Example 1. COBOL FUNCTION reserved word usage

The code shown in Example 1 above displays the following messages on the system logical output device:

Hello World! HELLO WORLD! hello world! HELLO WORLD!

16.1.2 Categories of Intrinsic Functions

The intrinsic functions can be grouped into six categories, based on the type of service performed. They are as follows:

1. Mathematical
2. Statistical
3. Date/time
4. Financial
5. Character-handling
6. General

Intrinsic functions operate against alphanumeric, national, numeric and integer data-items.

- **Alphanumeric** functions are of class and category alphanumeric. The value returned has an implicit usage of DISPLAY. The number of character positions in the value returned is determined by the function definition.
- **National** functions are of class and category national. The value returned has an implicit usage of NATIONAL and is represented in national characters (UTF-16). The number of character positions in the value returned is determined by the function definition.
- **Numeric** functions are of class and category numeric. The returned value is always considered to have an operational sign and is a numeric intermediate result.
- **Integer** functions are of class and category numeric. The returned value is always considered to hve an operational sign and is an integer intermediate result. The number of digit positions in the value returned is determined by the function definition.

16.2 Intrinsic Functions in Enterprise Developer 7.0

The current release of Enterprise Developer 7.0 includes several intrinsic functions. Each one of these functions falling into one of the aforementioned six categories. While an entire book could be written on intrinsic functions, a single example for each of the six categories is provided in this section.

16.2.1 Mathematical Example

Example 2 is storing into X the total of A + B + value resulting from C divided by D. FUNCTION SUM enables the arithmetic operation.

```
Compute x = Function Sum(a b (c / d))
```

Example 2. Mathematical intrinsic function

16.2.2 Statistical Example

Example 3 shows three COBOL functions, MEAN, MEDIAN and RANGE where the arithmetic values are stored in Avg-Tax, Median-Tax and Tax-Range using the data names with assigned pic clause values.

```
01   Tax-S           Pic 99v999 value .045.
01   Tax-T           Pic 99v999 value .02.
01   Tax-W           Pic 99v999 value .035.
01   Tax-B           Pic 99v999 value .03.
01   Ave-Tax         Pic 99v999.
01   Median-Tax      Pic 99v999.
01   Tax-Range       Pic 99v999.
. . .
     Compute Ave-Tax      = Function Mean   (Tax-S Tax-T Tax-W Tax-B)
     Compute Median-Tax   = Function Median (Tax-S Tax-T Tax-W Tax-B)
     Compute Tax-Range    = Function Range  (Tax-S Tax-T Tax-W Tax-B)
```

Example 3. Statistical intrinsic function

16.2.3 Date/time example

Example 4 shows usage of three COBOL functions, Current-Date, Integer-of-Date, and Date-of-Integer applied to MOVE, ADD and COMPUTE statements.

```
01   YYYYMMDD        Pic 9(8).
01   Integer-Form    Pic S9(9).
. . .
     Move Function Current-Date(1:8) to YYYYMMDD
     Compute Integer-Form = Function Integer-of-Date(YYYYMMDD)
     Add 90 to Integer-Form
     Compute YYYYMMDD = Function Date-of-Integer(Integer-Form)
     Display 'Due Date: ' YYYYMMDD
```

Example 4. Date/time intrinsic function

16.2.4 Financial Example

Example 5 shows application of COBOL function ANNUITY financial algorithm where values for loan amount, payments, interest and number of periods are input to ANNUITY function.

```
01   Loan                 Pic 9(9)V99.
01   Payment              Pic 9(9)V99.
01   Interest             Pic 9(9)V99.
01   Number-Periods       Pic 99.
. . .
     Compute Loan = 15000
     Compute Interest = .12
     Compute Number-Periods = 36
     Compute Payment = Loan * Function Annuity((Interest / 12)
        Number-Periods)
```

Example 5. Financial intrinsic function

16.2.5 Character-handling example

Example 6 shows uses of COBOL function UPPER-CASE, where a string or alphabetic variable processed by UPPER-CASE will translate any lower-case characters to upper case.

```
MOVE FUNCTION UPPER-CASE("This is shouting!") TO SOME-FIELD
DISPLAY SOME-FIELD
Output - THIS IS SHOUTING!
```

Example 6. Character-handling intrinsic function

16.3 Use of Intrinsic functions with reference modifiers

A reference modification defines a data item by specifying the leftmost character position and an optional length for the data item, where a colon (:) is used to distinguish the leftmost character position from the optional length, as shown in Example 7.

```
05 LNAME     PIC X(20).

LNAME(1:1)
LNAME(4:2)
```

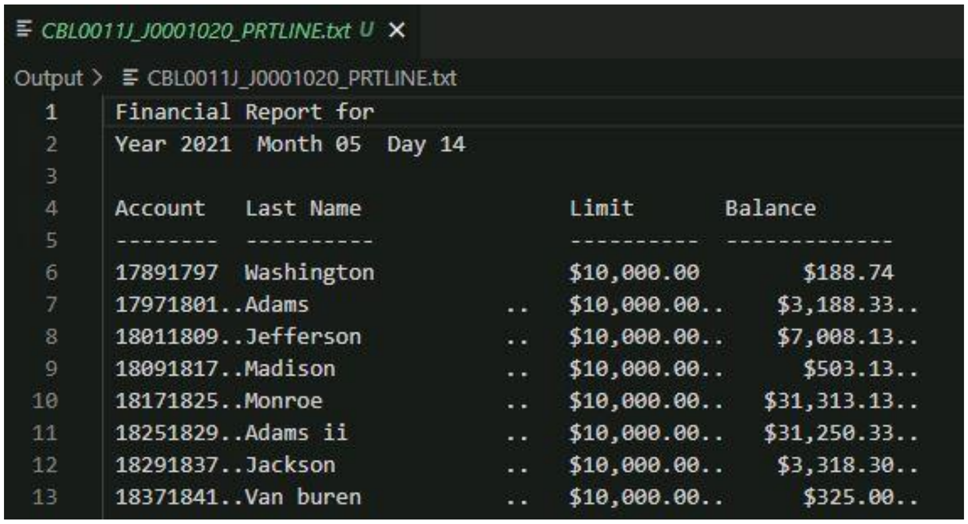
Example 7. Reference modification

Reference modification, LNAME(1:1), would return only the first character of data item LNAME, while reference modification, LNAME(4:2), would return the fourth and fifth characters of LNAME as the result of starting in the fourth character position with a length of two. If LNAME of value SMITH was the data item being referenced in the intrinsic function, the first reference would output, S. Considering those same specs, the second reference would output, TH.

16.4 Lab

In this Lab, you will work with the handling of the upper case and lower-case intrinsic functions.

1. Compile COBOL member CBL0011 and submit job CBL0011J.
2. Download and view the program output. Notice the names are now shown in lower-case letters, with only the initials being upper-case.


CBL0011J_J0001020_PRTLINE.txt U ×
Output > CBL0011J_J0001020_PRTLINE.txt

```
Financial Report for
Year 2021  Month 05  Day 14

Account   Last Name                 Limit       Balance
--------  ----------                ----------  -------------
17891797  Washington                $10,000.00        $188.74
17971801..Adams                  .. $10,000.00..    $3,188.33..
18011809..Jefferson              .. $10,000.00..    $7,008.13..
18091817..Madison                .. $10,000.00..      $503.13..
18171825..Monroe                 .. $10,000.00..   $31,313.13..
18251829..Adams ii               .. $10,000.00..   $31,250.33..
18291837..Jackson                .. $10,000.00..    $3,318.30..
18371841..Van buren              .. $10,000.00..      $325.00..
```

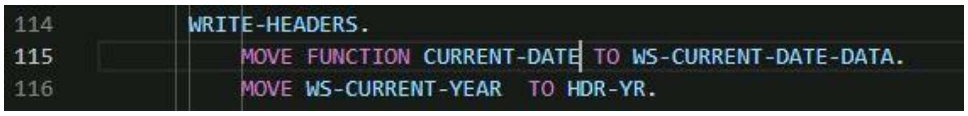
If you look at the source code, in the WRITE-RECORD paragraph, you will find the intrinsic function that makes the letters displayed as lower-case.

3. Compile CBL0012. The compilation will be unsuccessful.

4. Fix the error in the source code, save and recompile. *
5. Submit job CBL0012J, download and view the program output to check it now looks the same as the one from CBL0011.

Hint:

1. The error is located at line 115 in CBL0012, compare it with line 120 in CBL0011 to correct it.



```
114          WRITE-HEADERS.
115              MOVE FUNCTION CURRENT-DATE TO WS-CURRENT-DATE-DATA.
116              MOVE WS-CURRENT-YEAR  TO HDR-YR.
```