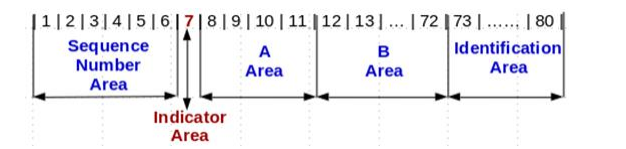
# Basic COBOL

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.

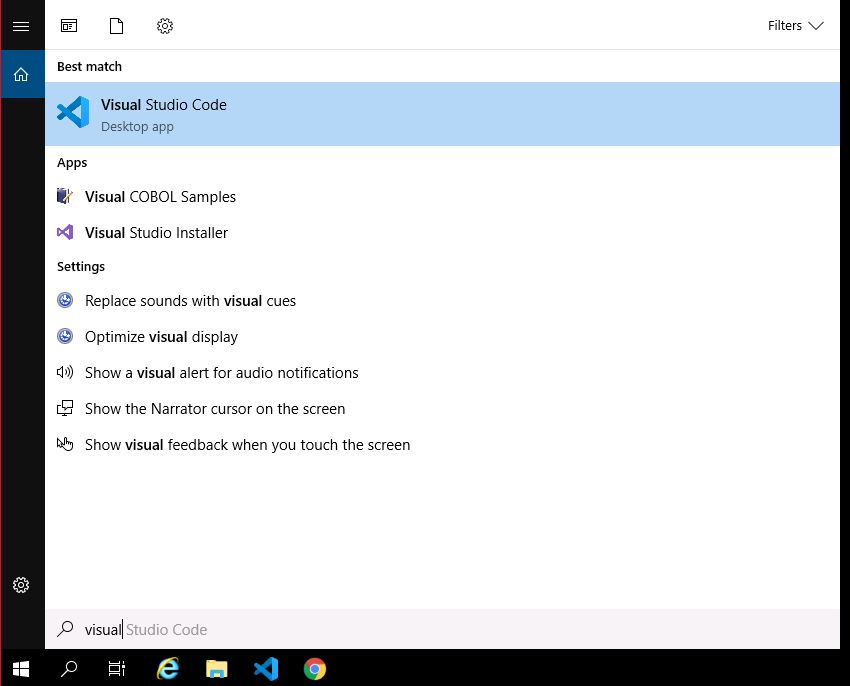
## 8.4 Procedure Division

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

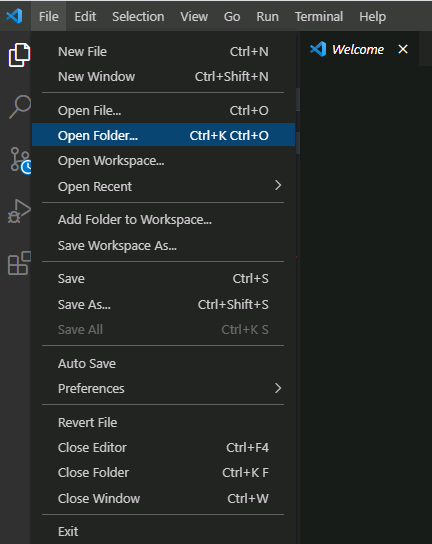
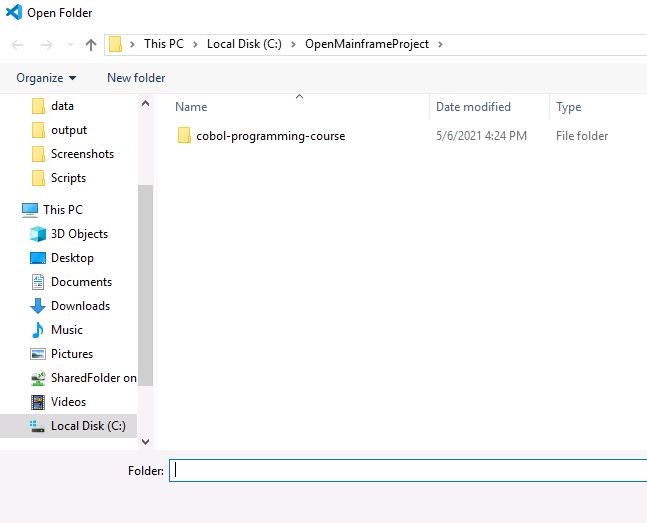
## 8.5 Lab

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

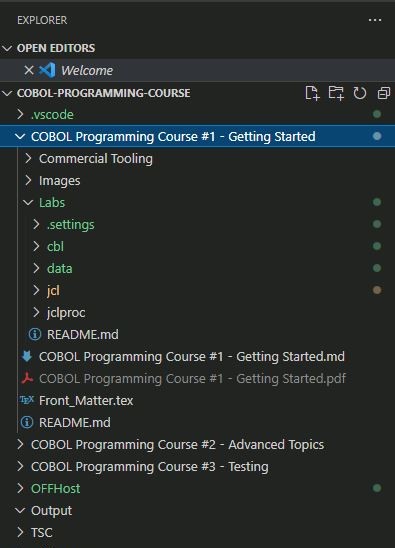
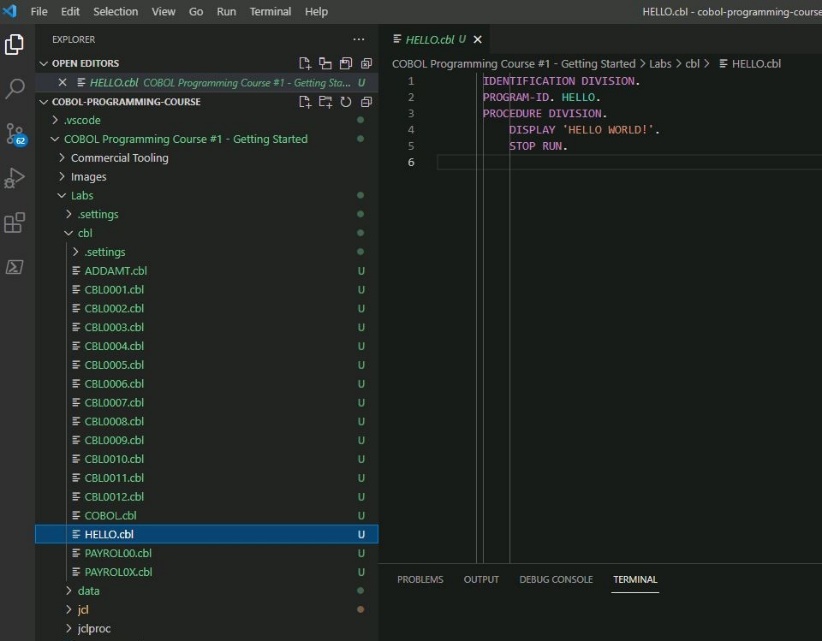
1. This lab assumes you have followed the VM installation covered in chapters 6 and 7, or you are using one of the provisioned AWS instances.
2. In your start menu search bar, search for Visual Studio Code and open it.



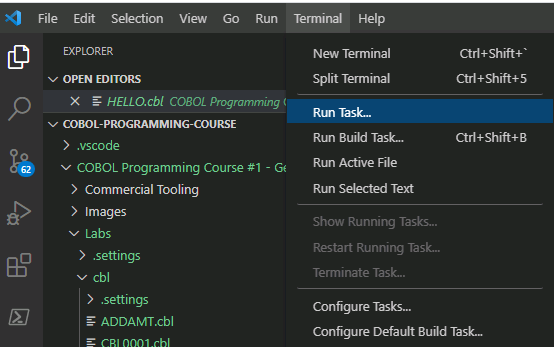
1. In VS Code, open the folder where you have saved this guide’s Lab source code. This will open the folder in the VS code browse tab on the left-hand side. If you are using one of the provisioned AWS instances, this is usually found in the C: drive, select the folder called cobol-programming-course.

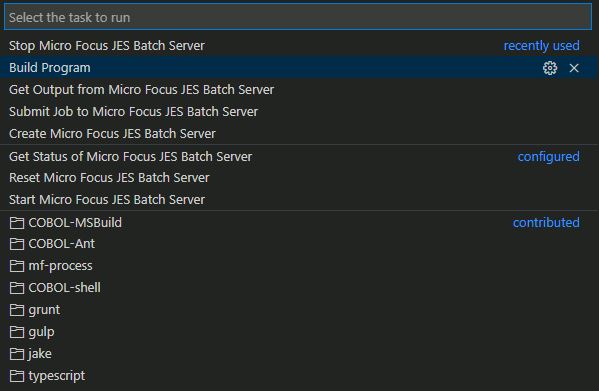
1. Notice you have several folders. One of these contains your COBOL and JCL files, which you will need to complete each exercise. Because of how the environment is provisioned, you will need to open each file you wish to compile or run in the editor, by double clicking on it. So open the file HELLO.cbl.

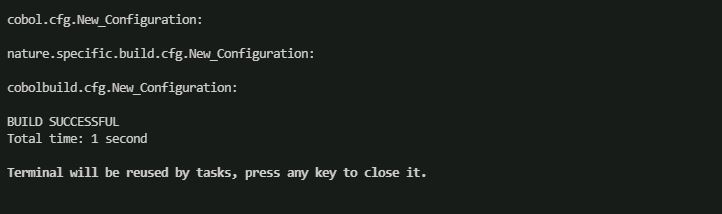
 

1. Now in the top menu, click on Terminal, then Run Task. The Terminal in VS Code is similar to a Command Prompt: it’s a space where you type in commands that your operating system understands and executes and can be found at the bottom of the editor. In this case, the commands have been automated, so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

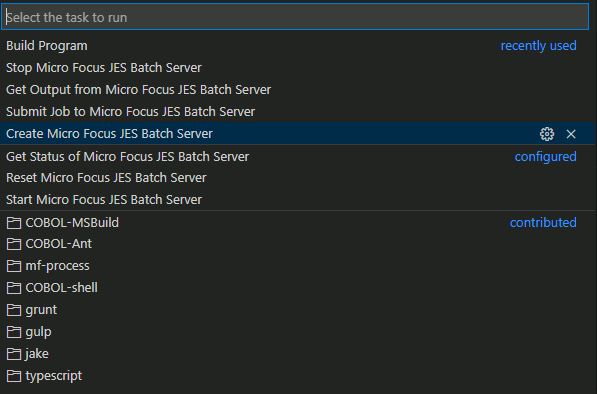


1. In the new top center menu, click on Build Program. Building or compiling a program will create the necessary binary files for the machine to run COBOL code. Notice the Terminal window at the bottom is now populated with everything that has been going on in the background, make sure at the very bottom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

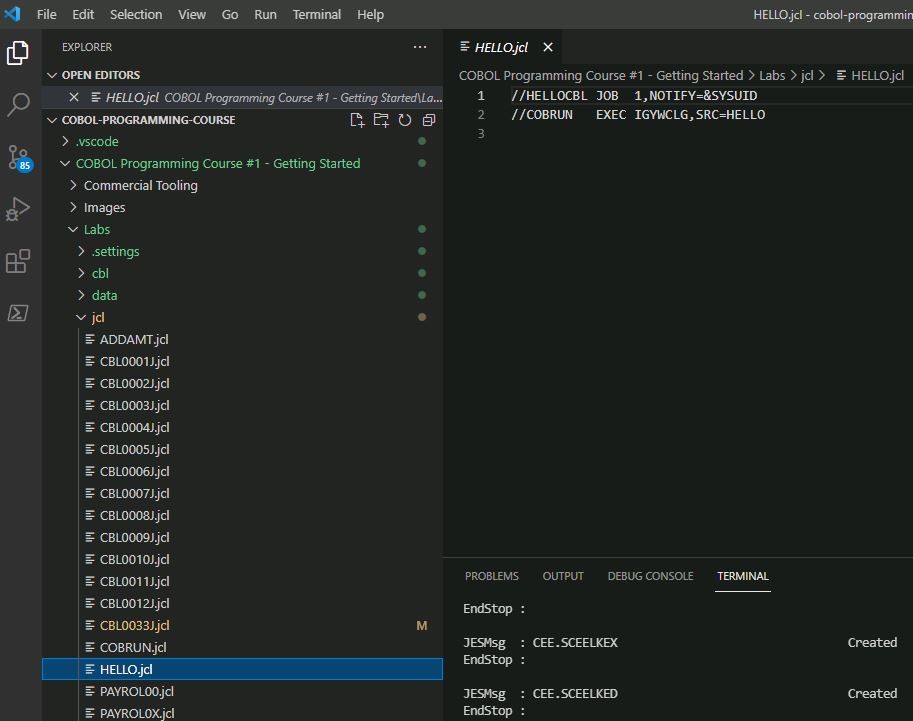


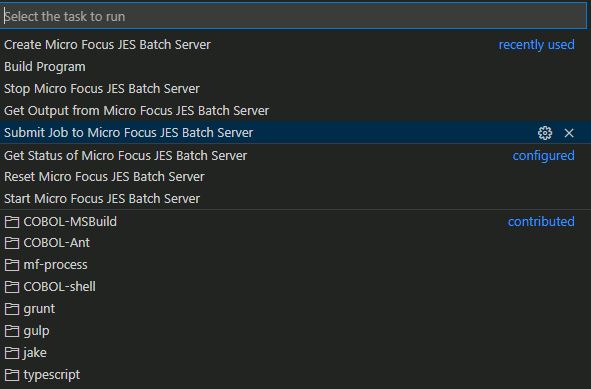


1. Now that your program is ready to run, you’ll need to provision the Enterprise Server on which it will run. So, click on Terminal, Run task, and then Create Micro Focus JES Batch Server: this is the region that emulates the functionality of JES on the mainframe, which means it can run batch jobs though 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.



1. 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.

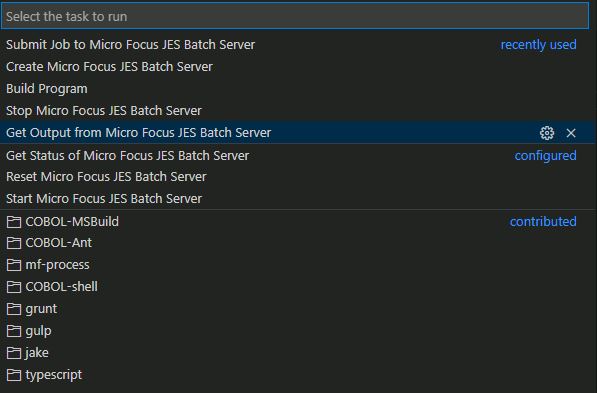




1. To ensure the program run correctly, you will need to check the job output. To do this, you will need the job ID, which appears in the Terminal window after the job was successful, and for the first program you run is usually J0001000. Copy this from the Terminal window.

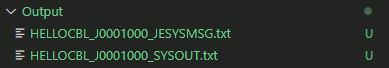


1. To retrieve the output, click on Terminal, Run Task, then Get Output from Micro Focus JES Batch Server, then copy the job ID in the bar that appears with the tag ‘ID of Job to Retrieve Output for’.

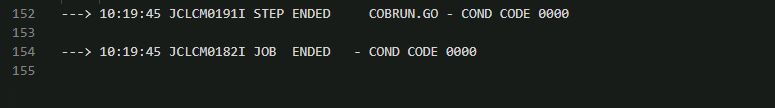




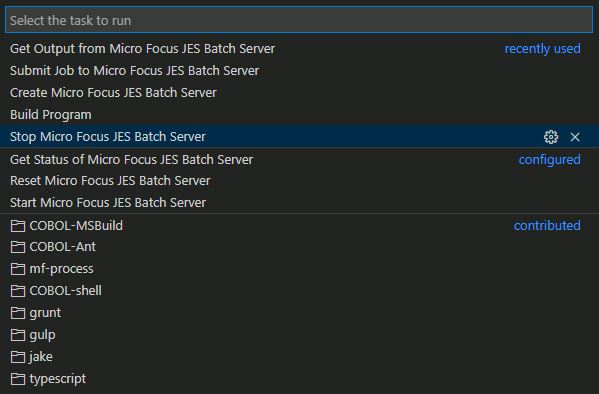
1. You can now find the job output in the Output folder within the main repository. The file name is usually HELLO\_ J0001000\_JESYSMSG.txt, double click on it to open it in the editor.



1. 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.



1. 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.



# 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](https://supportline.microfocus.com/Documentation/AcucorpProducts/docs/v6_online_doc/gtman3/gt3541.htm)

### 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

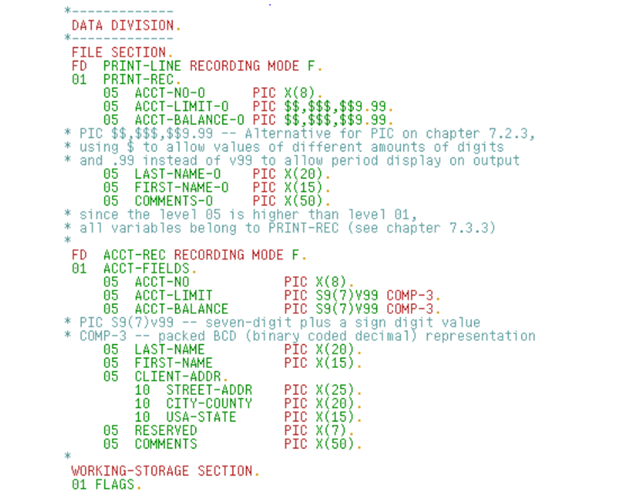
The relationships among all data to be used in a program is defined in the DATA DIVISION through a system of level indicators and level-numbers

### 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

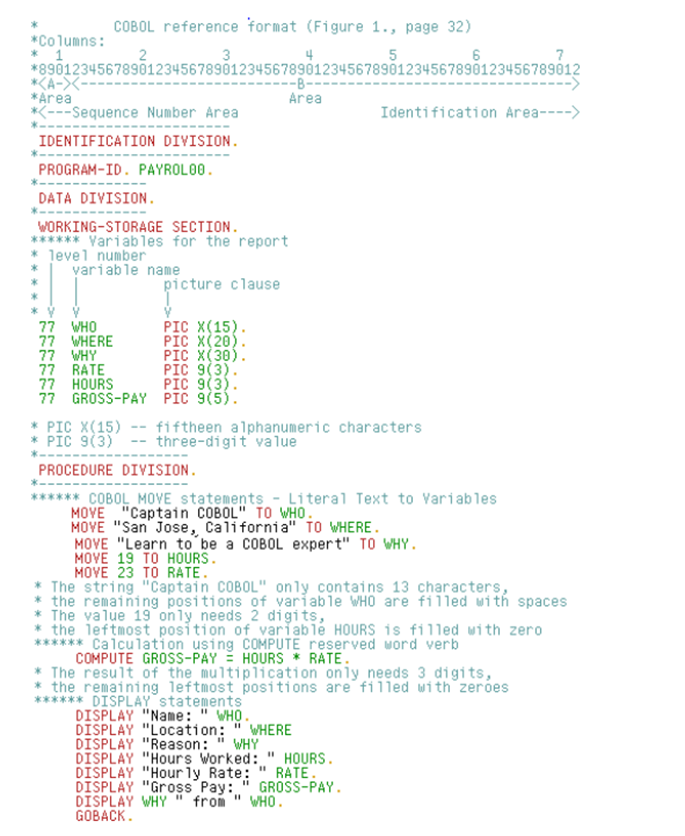
Observe the following example:



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.

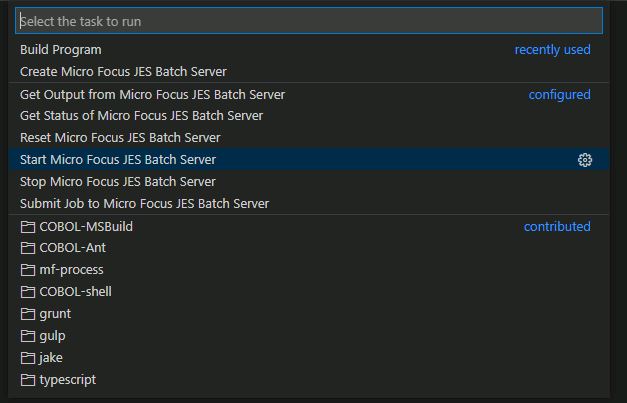


*Figure 2 (MOVE and COMPUTE statements)*

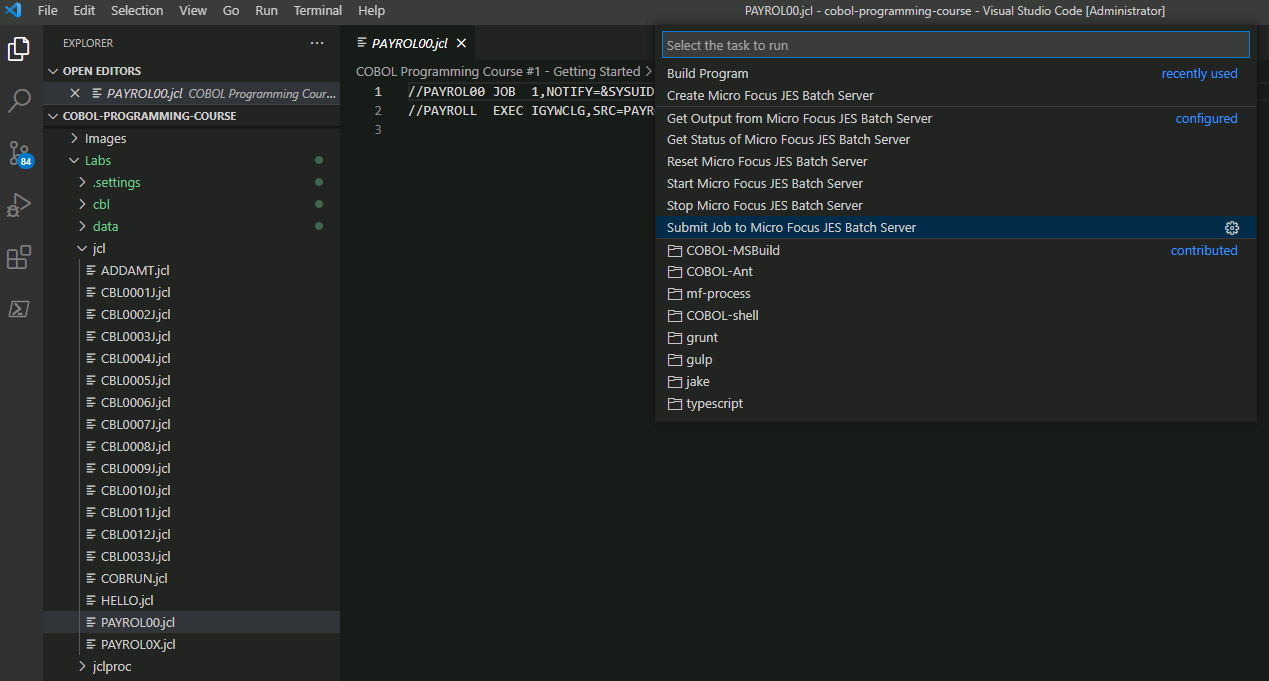
## 9.5 Lab

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.

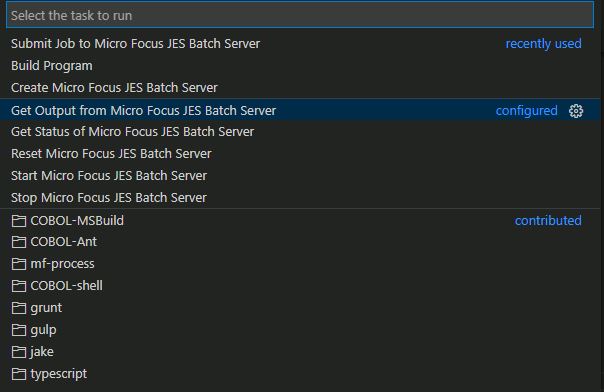
1. First of all, start the Enterprise Server region. From the Terminal drop down menu, choose Run Task and then Start the Micro Focus JES Batch Server.



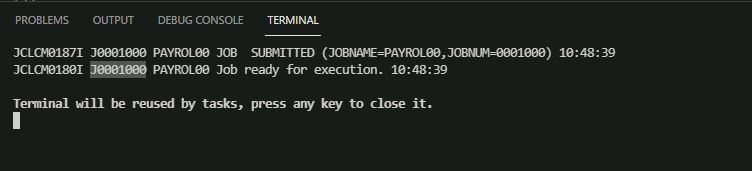
1. In the navigation menu, select and open the member PAYROL00.cbl and take a moment to look at the code. Then compile the program choosing Build Program from the central drop-down menu. Remember that this task compiles the COBOL program that is open in the editor.
2. When the compile is successful, 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.



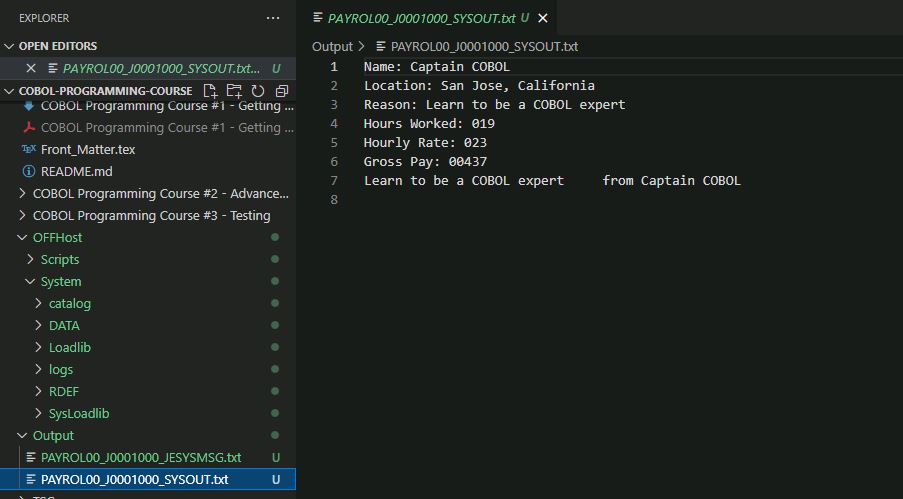
1. If the run fails, there might be an issue with the Enterprise Server region because it wasn’t stopper properly at the end of the previous session. To fix this, select the Reset Micro Focus JES Batch Server menu item, then start the region again.



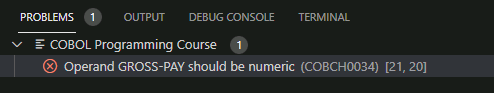
1. If the run is successful, look at the terminal window at the bottom of the editor. Here you will see the Job ID as mentioned before, which you will need to copy in order to download the program output from the Enterprise Server region.



1. Download the output as explained in the previous Lab.
2. 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.



1. If you compare the source code and the program output, you will see that the program displayed the information in the program output file.
2. Now open the COBOL member PAYROL0X.cbl, look at the source code and then compile the program.
3. 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.



Refer to the theory in this chapter to fix the error, then save and recompile the program.

1. Submit job PAYROL0X.
2. Get the new job output as outlined above and check that it’s the same output as the previous job (PAYROL00).
3. 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

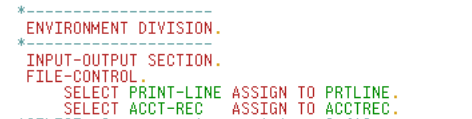
### 10.1.1 COBOL inputs and outputs

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:

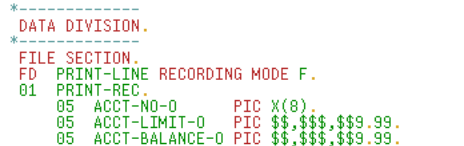
* SELECT – This clause creates an internal file name
* ASSIGN – This clause creates a name for an external data source



*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.



*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.

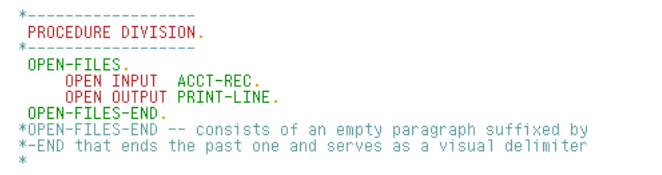
Redirecting ACCT-REC to the external data source ACCTREC using the ASSIGN clause as shown in *Figure 1* allows us more flexibility. This flexibility allows the same COBOL program to access a different environment variable modification:

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.



*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.

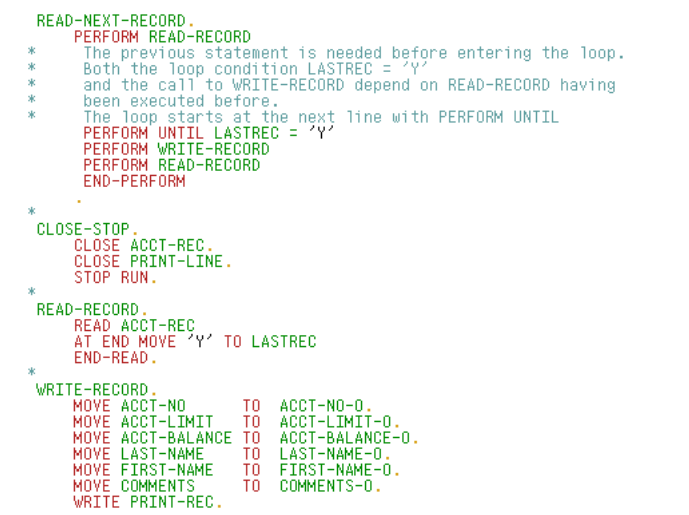


*Figure 5 – CLOSE-STOP*

## 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*:



*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.

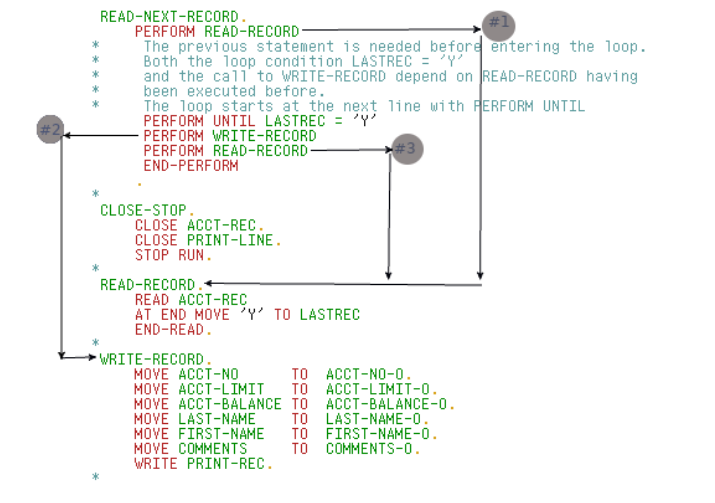
The WRITE-RECORD paragraph contains several MOVE statements. These statements move each input file variable name to an output file variable name. In the final sentence, the program writes the collection of output file variable names, PRINT-REC.

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.



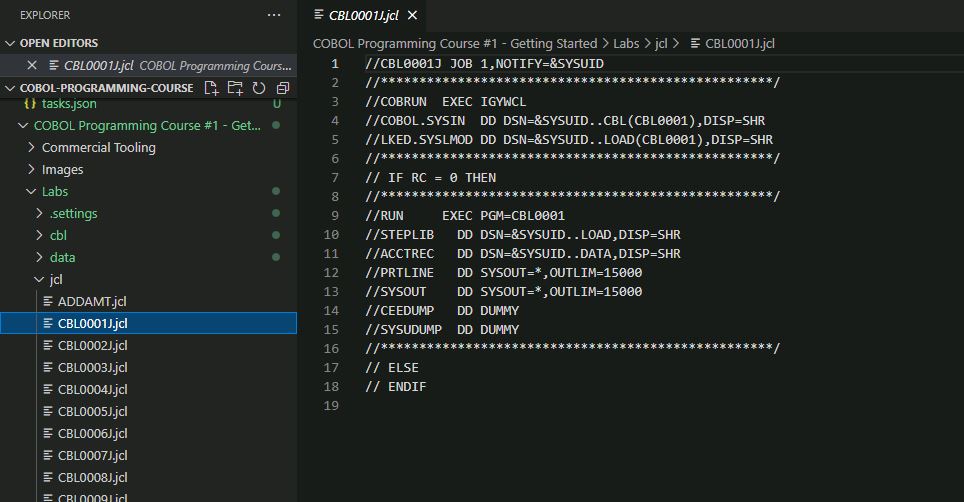
*Figure 7 – Iterative Processing*

### 

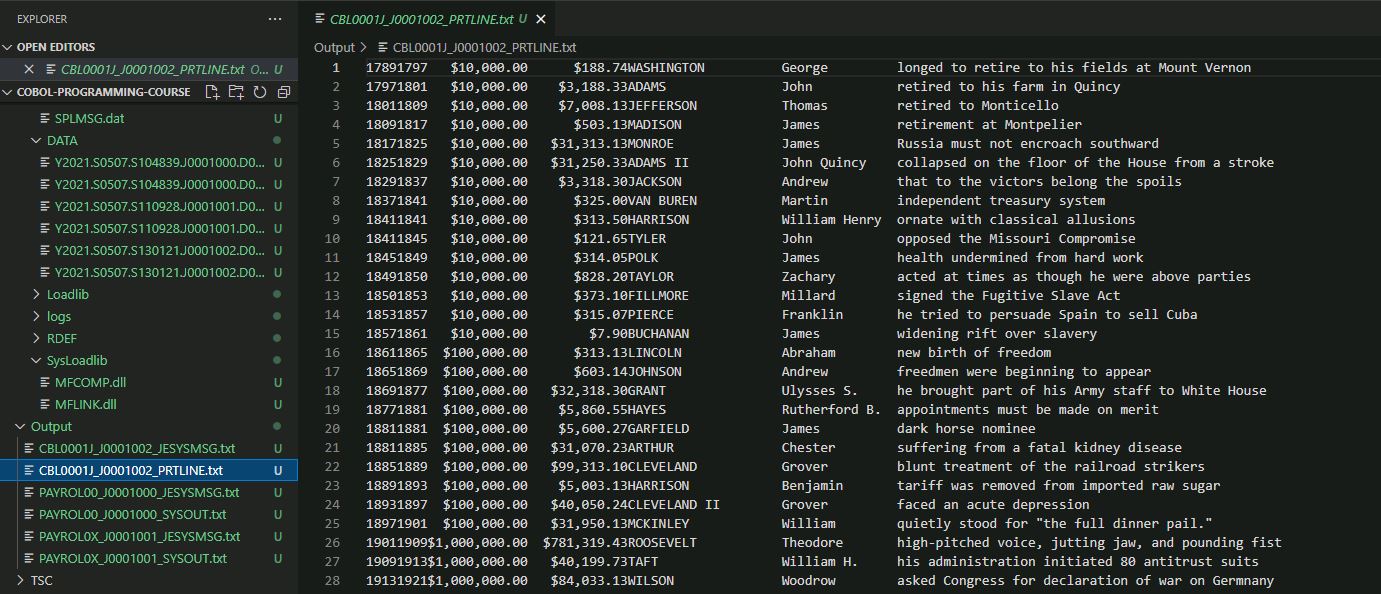
## 10.4 Lab

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.

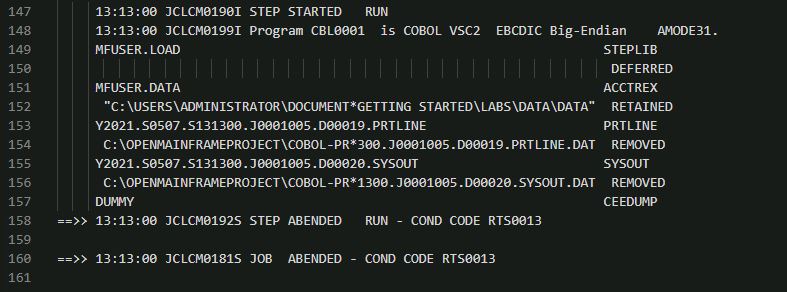


1. View the job output to ensure the job ran successfully.
2. Now look at the program output.



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.

1. Compile CBL0002.
2. Notice the unsuccessful build. Fix the code and recompile. \*
3. Submit job CBL0002J and view the job output and the program output, which should look the same as the one from CBL0001.
4. Submit job CBL0033J and view the job output.
5. Scroll to the bottom and notice the ABEND message.

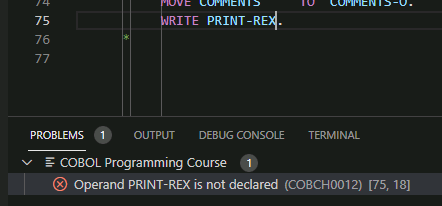


ABEND stands for ‘abnormal end’ and signals that the job failed due to an error in the JCL.

1. Open JCL member CBL0033J, find the error and fix it. \*
2. Save and the resubmit the job.
3. 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.



1. 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.



# 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](https://www.microfocus.com/documentation/visual-cobol/vc60/VS2019/GUID-A80CA9E8-7B68-4637-91E9-42078C8C9275.html).

### 11.1.3 COBOL programming style

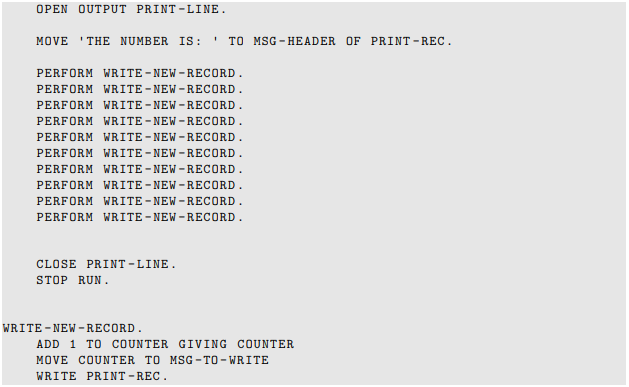
COBOL doesn’t directly have some of the components of a structured programming language such as Java. COBOL doesn’t contain ‘for’ or ‘while’ loops, nor does it contain defined functions or methods. Because COBOL is designed to be a language which is easy to read, these concepts are embodied through the 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:

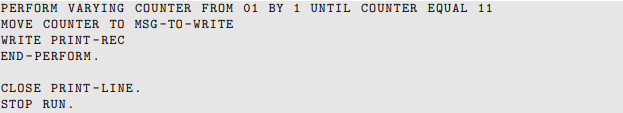


In this example, the three lines of code that constructed a new line of output and printed it has been extracted into a new paragraph called WRITE-NEW-RECORD. This paragraph is then performed ten times by the use of the PERFORM keyword. Each time the PERFORM keyword is used, execution jumps to the paragraph 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:





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:

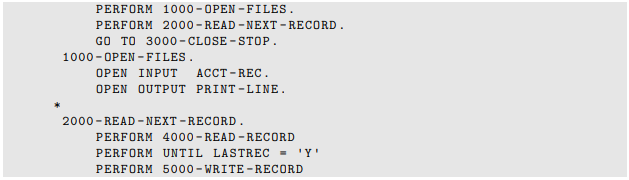
1. To group a sequence of COBOL sentences together that achieve a particular function or task, such as open all the files that an application is using, calculate a particular function or perform some data validation. Grouping such functions into a paragraph allows you to give them a meaningful name to 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.

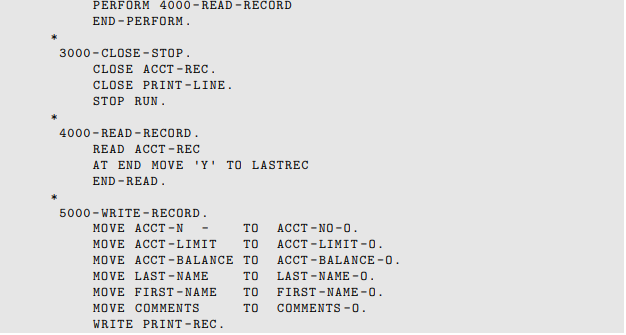
Remember that you can also perform other paragraphs within existing paragraphs. This nested calling of paragraphs can, again, help to structure your code.

### 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:

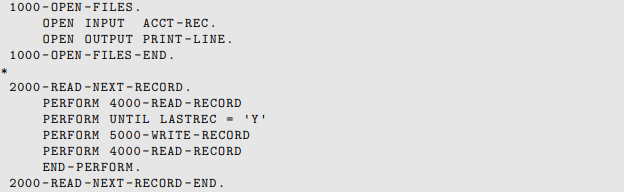
* Name each paragraph to correspond best with its function or behaviour. A paragraph named OPEN-INPUT-FILES is a lot easier to understand than one named DO-FILE-STUFF.
* Order the paragraphs in the general sequence they will be executed at runtime.
* Some COBOL programmers prefix the name of paragraphs 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.





*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.



*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

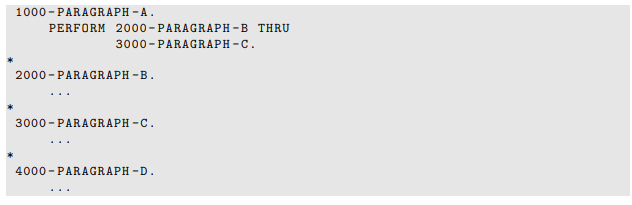
Perhaps the simplest way of repeating a PERFORM statement is to use the TIMES keyword to perform a paragraph or sections of code a static number of times, as shown in *Example 10*



*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*



*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.

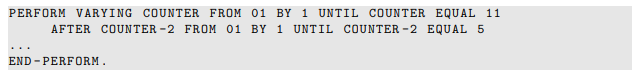
With structured programming, the use of sections rather than PERFORM THROUGH is recommended.

### 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*.



*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.*



*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](https://www.microfocus.com/documentation/visual-cobol/vc60/VS2019/BKDIDIWINTS001.html)

## 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

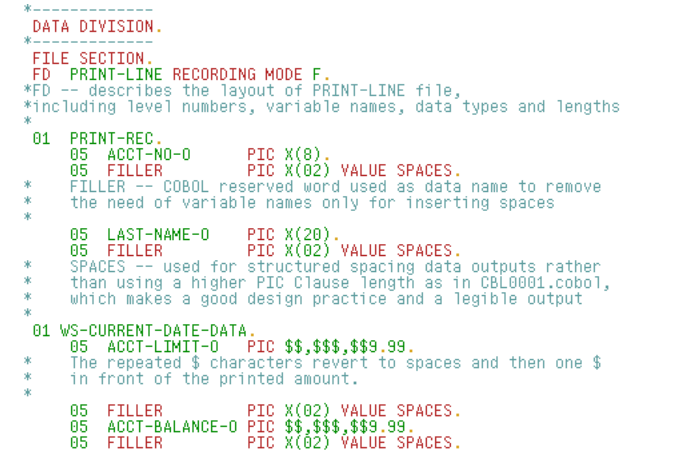
You will remember from earlier chapters that the Environment Division plays an important role when it comes to file handling in a COBOL program. The ‘File Handling’ section covered the SELECT and ASSIGN programmer chosen names. This section, however, focuses on output.

### 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:



*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:**

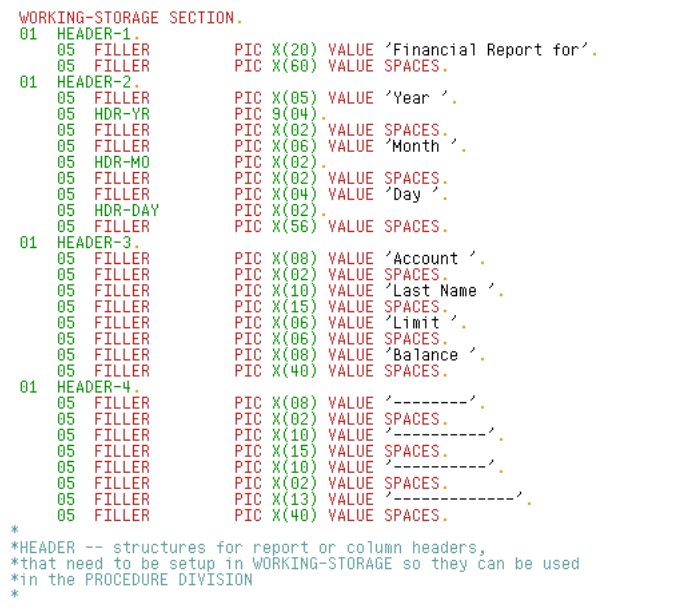
* Writes a literal
* Example: Financial Report for’

**HEADER-2:**

* Writes literals
* Examples:
  + ‘Year’ followed by a variable name
  + ‘Month’ followed by a variable name
  + ‘Day’ followed by a variable name

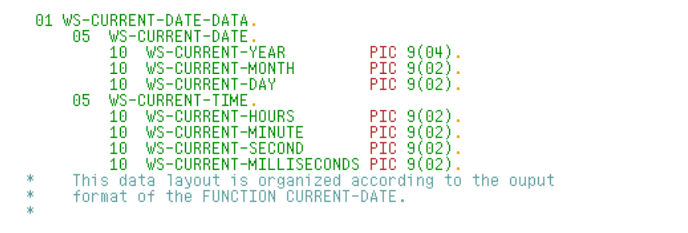
**HEADER-3:**

* Writes literals
* Examples:
  + ‘Account’ followed by FILLER spacing
  + ‘Last Name’ followed by FILLER spacing
  + ‘Limit’ followed by FILLER spacing
  + ‘Balance’ followed by FILLER spacing



*Figure 3*

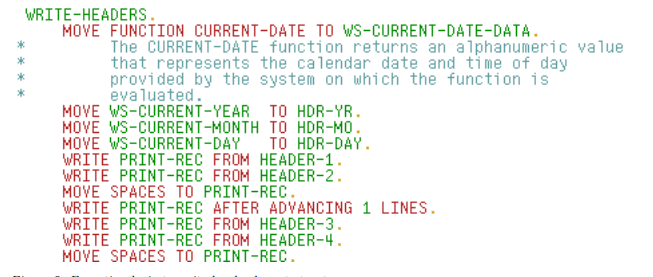
**HEADER-2** includes the year, month and day of the report together with FILLER area, creating blank spaces between them as you can see in *Figure 3*. *Figure 4* however is an example of the data name layout used to store the values of CURRENT-DATE. The information COBOL provides in CURRENT-DATE is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



*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.



*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.

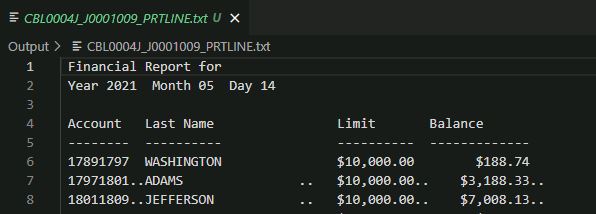
### 12.4.2 PRINT-REC FROM sentences

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.*

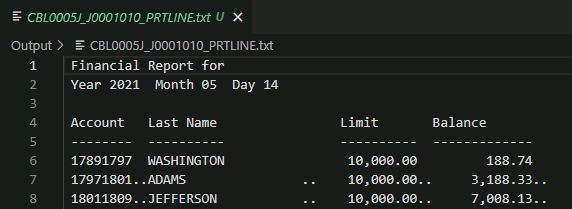
## 12.5 Lab

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.



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



1. Open CBL0005 and fix the source code to have the output file include the currency sign. \*
2. Save, recompile CBL0005 and resubmit job CBL0005J.
3. 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.

