



OPEN **MAINFRAME** PROJECT

# COBOL

*Programming Course*

## COBOL Programming Course 3

### Advanced Topics

Version 2.3.0

## Copyright

COBOL Programming Course is licensed under Creative Commons Attribution 4.0 International. To view a copy of this license, visit <https://creativecommons.org/licenses/by/4.0>.

Copyright Contributors to the Open Mainframe Project's COBOL Programming Course

# Contents

<b>1</b>	<b>Resources</b>	<b>5</b>
1.1	Professional manuals	5
1.2	Learn more about recent COBOL advancements	5
<b>2</b>	<b>Numerical Data Representation</b>	<b>6</b>
2.1	Numbering systems	6
2.1.1	Binary System	6
2.1.2	Hexadecimal System	7
2.1.3	EBCDIC Encoding	9
2.1.4	COBOL Picture Clause	10
2.2	Numeric Representations in COBOL	11
2.2.1	Zoned Decimal Format	11
2.2.2	Packed Decimal Format	12
2.2.3	Binary Format	14
2.2.4	COMP-1: Single Precision Floating Point	16
2.2.5	COMP-2: Double Precision Floating Point	16
<b>3</b>	<b>Dynamic-Length Item</b>	<b>17</b>
3.1	Dynamic-Length Elementary Items	17
3.2	Dynamic-Length Group Items	17
3.3	DYNAMIC LENGTH Clause	17
<b>4</b>	<b>UTF-8 Data Type</b>	<b>19</b>
4.1	UTF-8 Data Items	19
4.1.1	Fixed Character-Length UTF-8 Data Items	19
4.1.2	Fixed Byte-Length UTF-8 Data Items	19
4.1.3	Dynamic-Length UTF-8 Data Items	19
4.2	UTF-8 Literals	19
4.2.1	Basic UTF-8 Literals	19
4.2.2	Hexadecimal UTF-8 Literals	20
4.3	UTF-8 Move Rules and Conversion	20
<b>5</b>	<b>COBOL Application Programming Interface (API)</b>	<b>21</b>
5.1	Enterprise COBOL APIs	21
5.1.1	z/OS Middleware	21
5.1.2	COBOL API Communication with Middleware	21
5.1.3	COBOL EXEC SQL	22
5.1.4	COBOL Data Items	22
5.2	SQL Capability within Enterprise COBOL	23
5.2.1	Enterprise COBOL Application Programming and SQL Guide	23
5.2.2	Db2 Data Base Administration (DBA) vs Application Programming	24
5.3	Lab	24
5.3.1	Using VSCode and Zowe Explorer	24
<b>6</b>	<b>COBOL Program Compilation</b>	<b>25</b>
6.1	Compilation via JCL	25
6.1.1	Catalogued JCL Procedure	25
6.1.2	Writing JCL to compile programs	26
6.2	Specifying compiler options	27
6.2.1	Specifying options in the PROCESS statement	28
6.2.2	Specifying options in JCL	28
6.3	Batch compilation	28
6.3.1	Compiler options in a batch compilation	29

<b>7</b>	<b>Multithreading and COBOL</b>	<b>30</b>
7.1	Multithreading	30
7.2	THREAD to support multithreading	30
7.3	Transferring control to multithreaded programs	30
7.4	Ending multithreaded environment	31
7.5	Processing files with multithreading	31
7.5.1	File-definition storage	31
7.5.2	Serializing file access with multithreading	31
7.6	Limitation of COBOL with multithreading	32
<b>8</b>	<b>Program tuning and simplification</b>	<b>33</b>
8.1	Optimal programming style	33
8.1.1	Using structured programming	33
8.1.2	Factoring expressions	34
8.1.3	Using symbolic constants	34
8.2	Choosing efficient data types	34
8.2.1	Efficient computational data types	34
8.2.2	Consistent data types	35
8.2.3	Efficient arithmetic expressions	35
8.2.4	Efficient exponentiations	35
8.3	Handling tables efficiently	35
8.4	Choosing compiler features to enhance performance	36
<b>9</b>	<b>COBOL Challenges</b>	<b>37</b>
9.1	COBOL Challenge - Debugging	38
9.2	COBOL Challenge - The COVID-19 Reports	40
9.2.1	Instructions	40
9.2.2	Advanced Tasks	42
9.2.3	Solution	42
9.3	COBOL Challenge - The Unemployment Claims	43
9.3.1	Our Data	43
9.3.2	Use Case	43
9.3.3	Instructions	43
9.3.4	Craving more programming challenges?	47
9.4	Hacker News Rankings for Mainframe/COBOL Posts	48
9.4.1	A Little Background	48
9.4.2	Our Goal	48
9.4.3	The Plan	48

# 1 Resources

This section provides useful resources in the form of manuals and videos to assist in learning more about the basics of COBOL.

## 1.1 Professional manuals

As Enterprise COBOL experience advances, the need for the professional documentation is greater. An internet search for Enterprise COBOL manuals includes: “Enterprise COBOL for z/OS documentation library - IBM”, link provided below. The site content has tabs for each COBOL release level. As of April 2020, the current release of Enterprise COBOL is V6.3. Highlight V6.3 tab, then select product documentation.

<https://www.ibm.com/support/pages/enterprise-cobol-zos-documentation-library>

Three ‘Enterprise COBOL for z/OS’ manuals are referenced throughout the chapters as sources of additional information, for reference and to advance the level of knowledge. They are:

1. Language Reference - Describes the COBOL language such as program structure, reserved words, etc.

<http://publibfp.boulder.ibm.com/epubs/pdf/igy6lr30.pdf>

2. Programming Guide - Describes advanced topics such as COBOL compiler options, program performance optimization, handling errors, etc.

<http://publibfp.boulder.ibm.com/epubs/pdf/igy6pg30.pdf>

3. Messages and Codes - To better understand certain COBOL compiler messages and return codes to diagnose problems.

<http://publibfp.boulder.ibm.com/epubs/pdf/c2746481.pdf>

## 1.2 Learn more about recent COBOL advancements

- What’s New in Enterprise COBOL for z/OS V6.1:

<https://www.ibm.com/support/pages/cobol-v61-was-announced-whats-new>

- What’s New in Enterprise COBOL for z/OS V6.2:

<https://www.ibm.com/support/pages/cobol-v62-was-announced-whats-new>

- What’s New in Enterprise COBOL for z/OS V6.3:

<https://www.ibm.com/support/pages/cobol-v63-was-announced-whats-new>

## 2 Numerical Data Representation

In the first COBOL Programming Course, various types of data representation were discussed. This chapter seeks to expand upon the binary and hexadecimal numbering systems as well as the various numeric representations in COBOL.

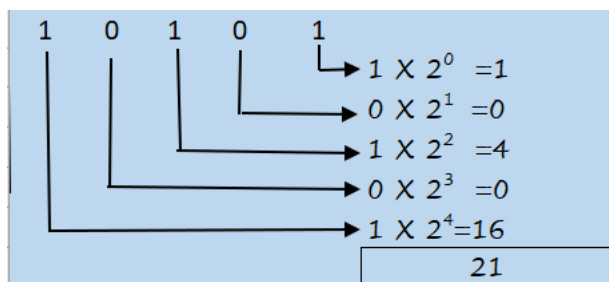
- **Numbering Systems**
  - **Binary System**
  - **Hexadecimal System**
  - **EBCDIC Encoding**
  - **COBOL Picture Clause**
- **Numeric Representations in COBOL**
  - **Zoned Decimal Format**
  - **Packed Decimal Format**
  - **Binary Format**
  - **Single Precision Floating Point**
  - **Double Precision Floating Point**

### 2.1 Numbering systems

A numbering system provides a means to represent numbers. We are most familiar with using the base-10 number system known as decimal. Data such as numerical values and text are internally represented by zeros and ones in most computers, including mainframe computers used by enterprises. This base-2 number system known as binary. Although data is encoded in binary on computers, it is much easier to work with base-16 known as hexadecimal. Each sequence of four binary digits is represented by a hexadecimal value.

#### 2.1.1 Binary System

Just as in our decimal system, a binary integer is a sequence of binary digits 0 and 1 arranged in such an order that the position of each bit implies its value in the integer. The binary representation of the number 21 is:



On the IBM Mainframe system, the two's complement form is used for the representation of binary integers. In this form, the leftmost bit is used to represent the sign of the number: 0 for positive and 1 for negative. For a positive number, the two's complement form is simply the binary form of the number with leading zero(s). For a negative number, the two's complement is obtained by writing out the positive value of the number in binary, then complementing each bit and finally adding 1 to the result. Assuming one byte of storage and b0b1b2b3b4b5b6b7 are the bits, let us look at some examples.

Decimal	Binary	Toggled Bits	Adding 1
+1	0000 0001	1111 1110	1111 1111 (represents -1)
+10	0000 1010	1111 0101	1111 0110 (represents -10)
+28	0001 1100	1110 0011	1110 0100 (represents -28)

As we can see, the sign bit b0 is 0 for a positive integer and 1 for a negative integer. This bit will participate in all arithmetic operations as though it represented the value  $(-b0 * 2^{k-1})$  for a k bit number. In the above

binary representation of -28:

$$-1*2^7 + 1*2^6 + 1*2^5 + 0*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 0*2^0 = -28$$

The number of bits will clearly dictate the range of values that can be stored. With k bits, the maximum positive value that can be stored correctly is  $2^{k-1} - 1$  and the minimum (negative) value will be  $-2^{k-1}$ . The number Zero is always represented with sign bit zero. With k=4:

Decimal	Binary
+1	0001
+2	0010
+3	0011
+4	0100
+5	0101
+6	0110
+7	0111
0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

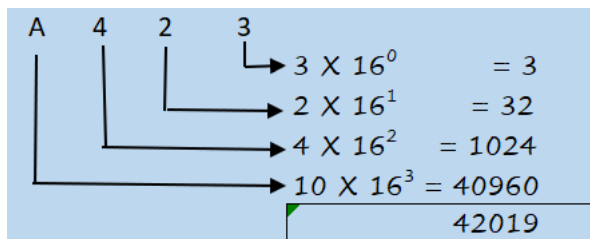
For K=32 bits, the range is  $-2^{31}$  to  $+2^{31} - 1$

### 2.1.2 Hexadecimal System

There are sixteen digits, represented by 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E and F. The first 10 symbols have their usual meaning; the remaining six, A through F, represent the values 10 through 15 when used as hexadecimal digits.

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	10000	10

A423 is the hexadecimal equivalent of the decimal value 42019:



Although data is encoded in binary on computers, it is rather cumbersome to work with binary. The hexadecimal numerals provide a human friendly representation of the binary coded values. An understanding of this system is invaluable to the COBOL programmer as he designs, develops and tests code. Often, hex dumps of the data in memory are used to debug a program and understand what is going on. The conversion between binary and hexadecimal system is easy as  $24 = 16$ . Each hexadecimal digit represents 4 binary digits, also known as a nibble, which is half a byte.

- To convert from hexadecimal to binary, replace each hexadecimal digit with its equivalent 4-bit binary representation
- To convert from binary to hexadecimal, replace every four consecutive binary digits by their equivalent hexadecimal digits, starting from the rightmost digit and adding zeros, on the left if necessary



Binary	<u>0001</u>	<u>0010</u>		
Hex	<b>1</b>	<b>2</b>		
<hr/>				
Binary	<u>0100</u>	<u>0000</u>		
Hex	<b>4</b>	<b>0</b>		
<hr/>				
Binary	<u>1110</u>	<u>0010</u>	<u>0100</u>	<u>1010</u>
Hex	<b>E</b>	<b>2</b>	<b>4</b>	<b>A</b>
<hr/>				
Binary	<u>1111</u>	<u>1101</u>	<u>1011</u>	
Hex	<b>F</b>	<b>D</b>	<b>B</b>	
<hr/>				

The usual convention is to use X' ' to denote a hexadecimal value, B' ' to denote a binary value.

### 2.1.3 EBCDIC Encoding

C' ' is used to represent a character value. It is helpful to familiarize yourself with the 8-bit EBCDIC encoding scheme that is used on the zOS and most IBM mainframes.

		EBCDIC character codes															
		1st hex digit								2nd hex digit							
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	DLE	DS		SP	&	-										0
1	SOH	DC1	SOS				/			a	j			A	J		1
2	STX	DC2	FS	SYN						b	k	s		B	K	S	2
3	ETX	TM								c	l	t		C	L	T	3
4	PF	RES	BYP	PN						d	m	u		D	M	U	4
5	HT	NL	LF	RS						e	n	v		E	N	V	5
6	LC	BS	ETB	UC						f	o	w		F	O	W	6
7	DEL	IL	ESC	EOT						g	p	x		G	P	X	7
8		CAN								h	q	y		H	Q	Y	8
9		EM								i	r	z	'	I	R	Z	9
A	SMM	CC	SM		C CENT	!		:									
B	VT	CU1	CU2	CU3		\$	,	#									
C	FF	IFS		DC4	<	*	%	@									
D	CR	IGS	ENQ	NAK	(	)	_	'									
E	SO	IRS	ACK		+	:	>	=									
F	SI	IUS	BEL	SUB		--	?	"									

## 8-bit EBCDIC Encoding

For numerical representations, the last column is of particular interest here; the character representations of numerical digits 0-9 in the EBCDIC encoding. 'C'5' is encoded, for example, as X'F5' and 'C'9' as encoded as X'F9'.

### 2.1.4 COBOL Picture Clause

As a quick reminder, COBOL leverages numeric data with a PIC clause that can contain a 9, V and/or S. These symbols keep the number purely mathematical that can participate in arithmetic.

- 9 is used to indicate numeric data consisting of the digits from 0 to 9
- V indicates where the assumed decimal place is located
- S will remember the sign which is necessary if the data is negative

COBOL PIC	& what it means
PIC 9(6)V99	6 whole numbers and 2 decimal places
PIC V999	3 decimal places
PIC 9(4)V9(4)	4 whole numbers and 4 decimal places
PIC 9(5)	5 whole numbers
PIC S9(5)	5 whole numbers, the sign is remembered
PIC S9(4)V9	4 whole numbers and 1 decimal place, the sign is remembered

Since the number of decimal places is determined and fixed in place by the V, this representation is called fixed point . Let us illustrate with an example to show how the V determines the value stored.

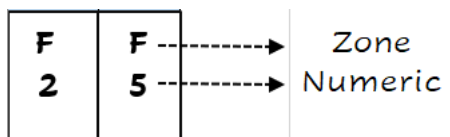
PIC Clause of result	Value of 393/100 stored in result	After Edit
PIC 99V999	03.930	3.93
PIC 99V99	03.93	3.93
PIC 99V9	03.9	3.9
PIC 99	03.	3
PIC V9	.9	0.9
PIC V99	.93	0.93

## 2.2 Numeric Representations in COBOL

In this section, we will investigate the numeric representations in COBOL: - Zoned Decimal (Fixed Point) - Packed Decimal (Fixed Point) - Binary (Fixed Point) - Single Precision Floating Point - Double Precision Floating Point

### 2.2.1 Zoned Decimal Format

In this format, each byte of storage contains one digit. The high order 4-bits (or nibble) are called the Zone bits. The low order 4-bits are called the Decimal or Numeric bits and will contain the binary value for the digit. Considering a simple case, the number 25 is represented as X'F2F5'.



The zone portion is the 'upper half byte' and numeric portion is the 'lower half byte'. This format is the default numeric encoding in COBOL. The coding syntax of USAGE IS DISPLAY can also be used. Let us look at a few valid zoned decimal declarations.

```

01 NUMERIC-ZD-DEC-1 PIC 99.

01 NUMERIC-ZD-DEC-2 PIC 9(7) USAGE IS DISPLAY.

02 NUMERIC-ZD-DEC-3 PIC S9(5).

01 NUMERIC-ZD-DEC-4 PIC S999V99 USAGE IS DISPLAY.

```

As discussed earlier, the first two declarations above are unsigned, indicated by the absence of a S. Such numbers are ‘implied positive’. The next two declarations are explicitly signed by the symbol S and are capable of representing positive and negative numbers. The sign is represented by the rightmost zone bits (in the above example the F above the 5) and is determined as follows:

- F indicates the number is unsigned
- C indicates the number is positive
- D indicates the number is negative

Number	Pic Clause	Hex	Binary	Printed as	Comments
123	PIC 999	X'F1F2F3'	B'1111 0001 1111 0010 1111 0011'	123	
456	PIC S999	X'F4F5C6'	B'1111 0100 1111 0101 1100 0110	45F	Reason : X'C6' is EBCDIC encoding of C'F'
-789	PIC S999	X'F7F8D9'	B'1111 0111 1111 1000 1101 1001	78R	Reason : X'D9' is EBCDIC encoding of C'R'

It is clear some adjusting (or editing) needs to be done before printing a signed number. The V in the declarations above has no storage allocated for it and will also need to be edited for printing purposes. To illustrate the way it works, consider an input file with a number 12345 and the input PIC clause is 999V99. This means that there is a decimal point assumed between 3 and 4. When the number is later aligned with an edited field, say ‘999.99’, the result is printed as 123.45.

When we code arithmetic statements involving zoned decimal fields, under the covers, COBOL converts the data to packed decimal and/or binary representations in order to do the math and the result is converted back to zoned decimal, all seamlessly. This extra step and hence a loss in efficiency is the price to pay for the easy readability that this format provides.

Advantages of Zoned Decimal	Disadvantages of Zoned Decimal
The unsigned, positive numbers are easy to display or print	Not an efficient use of memory or disk space, a 5 digit number will take up 5 bytes
The signed numbers require minimum effort to edit for printing or displaying	Not efficient or suited to perform arithmetic, usually converted to binary/packed decimal to do arithmetic and result converted back to zoned decimal
Easy for programmers to read when testing or debugging programs	Some editing required, esp for signed fields, before printing or displaying

### 2.2.2 Packed Decimal Format

In the zoned decimal format, the rightmost zone bits determine the sign; the other F’s are redundant. When a number is ‘packed’, those extra zone bits are removed, and only the rightmost zone bits are retained. Hence, the move from an unpacked field changes every byte in the field (except the last) from X’Fn’ to X’n’. The nibbles in the last byte get flipped (X’C2’ becomes X’2C’).

<b>Zoned Decimal</b>	X'F1'	X'F2'	X'C3'					<b>3 bytes of storage</b>
123	↓	↘	↘					
<b>Packed Decimal</b>	X'12'	X'3C'						<b>2 bytes of storage</b>
<b>Zoned Decimal</b>	X'F4'	X'F5'	X'D6'					<b>3 bytes of storage</b>
-456	↓	↘	↘					
<b>Packed Decimal</b>	X'45'	X'6D'						<b>2 bytes of storage</b>
<b>Zoned Decimal</b>	X'F6'	X'F7'	X'F8'	X'F9'	X'F9'	X'C0'		<b>6 bytes of storage</b>
678990	↓	↘	↘	↘	↘	↘		
<b>Packed Decimal</b>	X'06'	X'78'	X'99'	X'0C'				<b>4 bytes of storage</b>

As we can observe, when the number is packed into a field that is larger than necessary to hold that number, it is padded with zeroes on the left.

- Number 1 will be stored as X'1F' in 1 byte
- Number +12 will be stored as X'012C' in 2 bytes
- Number -123 will be stored as X'123D' in 2 bytes
- Number 1234 (unsigned) will be stored as X'01234F' in 3 bytes
- Number +12345 will be stored as X'12345C' in 3 bytes

The COBOL syntax for this format is USAGE IS COMP-3 or just COMP-3.

#### 01 NUMERIC-PACKED-DECIMAL PIC S9(5)V99 COMP-3.

As the packed decimal representation stores two digits in one byte, it is a variable length format. Also, as we can see, the digits are stored in decimal notation, and each digit is binary coded. So, COMP-3 exactly represents values with decimal places. A COMP-3 value can have up to 31 decimal digits. This format is somewhat unique and native to mainframe computers such as the IBM z architecture. The zOS has specialized hardware for packed decimal arithmetic and so the system can perform mathematical calculations without having to convert the format. This is, by far, the most utilized numerical value representation in COBOL programs. Storing information in this format may save a significant amount of storage space.

<b>A Zoned decimal field of length ...</b>	<b>Requires a Packed Decimal field of length</b>
1	1
2	2
3	2
4	3
5	3
m (where m is odd)	(m+1)/2
n (where n is even)	(n/2) + 1

It is usually the best choice for arithmetic involving decimal points/fractions. After numerical processing, a packed decimal field is (moved) unpacked into a zoned decimal format which can then be edited for printing purposes.

<b>Advantages of Packed Decimal</b>	<b>Disadvantages of Packed Decimal</b>
It requires less storage compared to zoned decimal	It usually requires more storage than binary
Packed decimal arithmetic has great precision, greatly suited for processing fractions	Not suited for printing or displaying purposes, needs to be converted to zoned decimal first

### 2.2.3 Binary Format

On the IBM Mainframe systems, the other main arithmetic type besides the packed decimal is the binary format which is built for efficiency in integer arithmetic operations. This encoding finds many uses in Legacy applications. Many datasets are created with binary fields. Variable length records and table processing in COBOL use this representation. The binary format is largely implementation dependent and has many variations. On the zOS and IBM Mainframes, the twos complement encoding is used.

The COBOL clauses for this format are COMP, COMP-4, COMPUTATIONAL or BINARY which can be used interchangeably. COMP-5 clause also falls in this category. Let us look at some valid declarations.

```
01 NUMERIC-BINARY-1 PIC 99 COMP.

01 NUMERIC-BINARY-2 PIC 9(7) COMP-4.

02 NUMERIC-BINARY-3 PIC S9(5)V99 COMP-5.
```

The PIC Clause determines the storage space:

- PIC 9(1) through PIC 9(4) will reserve 2 bytes (Binary halfword)
- PIC 9(5) through PIC 9(9) will reserve 4 bytes (Binary fullword)
- PIC 9(10) through PIC 9(18) will reserve 8 bytes (Binary doubleword)

Next, let's look at what numbers can be stored. For the COMP and COMP-4 fields, although the data is stored as binary numbers, the range is limited by the full value of the PIC Clause used in the field definition. The binary format, COMP-5 (also known as 'Native Binary') in which the PIC clause still defines the size of the field but the range of values that can be represented is much higher as every possible bit-value combination is valid.

	PIC	Storage length	Range	All those bits (two's complement)
COMP/COMP-4/BINARY	S9(1)	2 bytes (16 bits)	-9 to +9	
COMP-5			-32,768 to +32,767	
COMP/COMP-4/BINARY	9(1)		0 to 9	
COMP-5			0 to 65,535	
COMP/COMP-4/BINARY	S9(2)		-99 to +99	
COMP-5			-32,768 to +32,767	
COMP/COMP-4/BINARY	S9(4)		-9,999 to +9,999	
COMP-5			-32,768 to +32,767	<div>-2<sup>15</sup> : 1000 0000 0000 0000 -2<sup>15</sup> + 1 : 1000 0000 0000 0001 ..... -1 : 1111 1111 1111 1111 0 : 0000 0000 0000 0000 +1 : 0000 0000 0000 0001 ..... 2<sup>15</sup> - 2 : 0111 1111 1111 1110 2<sup>15</sup> - 1 : 0111 1111 1111 1111</div>

	PIC	Storage length	Range	All those bits (two's complement)
COMP/COMP-4/BINARY	S9(5)	4 bytes (32 bits)	-99,999 to +99,999	
COMP-5			-2,147,483,648 to +2,147,483,647	-2 <sup>31</sup> : 1000 0000 0000 0000 0000 0000 0000 0000 ..... 2 <sup>31</sup> : 0111 1111 1111 1111 1111 1111 1111 1111
COMP/COMP-4/BINARY	S9(9)		-999,999,999 to +999,999,999	
COMP-5			-2,147,483,648 to +2,147,483,647	

	PIC	Storage length	Range	All those bits (two's complement)
COMP/COMP-4/BINARY	S9(10)	8 bytes (64 bits)	-9,999,999,999 to +9,999,999,999	
COMP-5			-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	
COMP/COMP-4/BINARY	S9(18)		-999,999,999,999,999,999 to +999,999,999,999,999,999	
COMP-5			-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	

Those numbers are more than sufficient for most business applications! To give a quick comparison, a two byte packed decimal field can range in value from -999 to +999 only. When faced with larger than capacity values, COMP truncates to the decimal value of the PIC clause and COMP-5 truncates to the size of the field.

Although very much suited for integer processing, the binary format is not a good choice for non-integer arithmetic. Many banking and insurance applications rely on accuracy for their business processing logic and packed decimal format is preferred in such cases. Let's see why. In decimal systems, fractions are represented in terms of negative powers of 10:

$$\frac{3}{4} = 0.75 = 7 \times 10^{-1} + 5 \times 10^{-2}$$

In binary system fractions are represented in terms of negative powers of 2:

$$\frac{3}{4} = 0.75 = 1 \times 2^{-1} + 1 \times 2^{-2}$$

There is a possible loss of accuracy when converting a decimal fraction to a binary fraction as there is not a one-to-one correspondence between the set of numbers expressible in a finite number of binary digits and the set of numbers expressible in a finite number of decimal digits. Let's take the example of the fraction 1/10. In the decimal system:

$$\frac{1}{10} = 0.1 = 1 \times 10^{-1}$$

However, in the binary system, this is a never ending sequence of bits...!!

$$\frac{1}{10} = (.00011001100110011 \dots)_2 = 1 \times 2^{-4} + 1 \times 2^{-5} + 1 \times 2^{-8} + 1 \times 2^{-9} + \dots$$

So, we get different values when, for example, we multiply 1/10 by 10 in the decimal and in the binary systems. In the decimal system,  $10 \times 0.1 = 1.0$ . In the binary system, we get

$$= (1010) \times (0.00011001100110011 \dots) = 0.111111111 \dots$$

i.e. not quite 1.0 !! In scenarios involving a large number of calculations, this type of discrepancy may lead to cumulative rounding errors that may not be acceptable in many business applications. The use of packed decimal works very well in such cases.

Advantages of Binary	Disadvantages of Binary
It has a smaller footprint compared to packed decimal	It is not well suited for decimals or fractions, causing loss of precision
It is usually the most efficient format for integer arithmetic	It requires conversion to packed decimal, then zoned decimal for printing purposes

#### 2.2.4 COMP-1: Single Precision Floating Point

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.

#### 2.2.5 COMP-2: Double Precision Floating Point

COMP-2 extends the range of value that can be represented compared to COMP-1. COMP-2 can represent values up 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.

**Note :** This COBOL program can display the hexadecimal contents (and hence the exact internal representation) of a field. You can declare binary, packed decimal or zoned variable (or anything else, for that matter), do arithmetic with them and use the program to see how they are internally stored.



## 3 Dynamic-Length Item

So far during this course, we have only explored data items that have a fixed length. In other words, you need to define the length you need for each data type. But in this section, we will explore a fairly new concept in Enterprise COBOL - dynamic-length items.

Enterprise COBOL v6.3 supports dynamic-length items, which are items whose logical length might change at runtime.

### 3.1 Dynamic-Length Elementary Items

Let us consider a dynamic-length elementary item. To recall, an elementary item is an item that cannot be further subdivided. These items have a PIC clause since storage is reserved for it. If we used dynamic-length elementary item to send data, it will be treated as a fixed-length item with a length equals to the current length of the dynamic-length item. On the other hand, if we used dynamic-length elementary item to receive data and it's not reference-modified, the content will simply be moved to the receiving's content buffer.

If the content received is longer than the current length, a new larger buffer will be allocated for it. Additionally, if the length of the sender is zero, the receiver's length will be set to zero as well.

Now, if the dynamic-length elementary item is used to receive data and we reference-modified it, the item will be treated as a fixed-length item with a length equals to the current length of the dynamic-length item. In such cases, the compiler will not allocate or reallocated the buffer.

Note that not all statement supports dynamic-length elementary items. Common statement like REDEFINE or RENAME will not work. Additionally, we cannot take their address using the ADDRESS-OF special register. The full list of the statements supported is available on the [Language Reference](#).

When we compare a dynamic-length item with a fixed-length item, the comparison will follow the normal comparison rules (the shorter item will be extended to the right with enough spaces to make both items equal in length and then each character will be compared). Meanwhile, if you compare two dynamic-length elementary items, the lengths will be compared first and if they matched, the characters will then be examined.

We can also set the length of dynamic-length elementary item using the SET LENGTH OF syntax and pass dynamic-length elementary items as fixed-length items to a subroutine using the AS FIXED LENGTH phrase.

Note that doing the intrinsic function MIN and MAX are not supported for dynamic-length items.

### 3.2 Dynamic-Length Group Items

A dynamic-length group item is a group item that contains at least one subordinate dynamic-length elementary item and whose logical length might change at runtime.

Any other group item is considered to be a fixed-length group item. These fixed-length group items can contain variable-length tables through the OCCURS DEPENDING ON clause.

Additionally, we cannot compare or move dynamic-length group items to any other group items. On the other hand, fixed-length group items are always compatible and comparable with other fixed-length group items.

### 3.3 DYNAMIC LENGTH Clause

To define a dynamic length item, we can include the DYNAMIC LENGTH clause on the data description entry. Here are a couple of examples of how to indicate the clause:

```
01 MY-DYN PIC X DYNAMIC.  
01 NAME PIC X DYNAMIC LENGTH.  
01 DYN-PRICE PIC X DYNAMIC LIMIT 500.
```

Let us observe a few things from the examples above. Firstly, we note that the `LENGTH` keyword is optional. Next, we also have a `LIMIT` phrase that specifies the maximum length of the data item. If a sender's length is longer than the receiver's `LIMIT` value, the data will be truncated on the right. This `LIMIT` value defaults to 999999999 if not specified. Lastly, note that we use `PIC X`. To use dynamic-length clause, you can only use `PIC X` or `PIC U` (which is for UTF-8 data item).

## 4 UTF-8 Data Type

With Enterprise COBOL v6.3, we also have a new USAGE, which is UTF-8. This is indicated by the picture symbol 'U'. Unlike NATIONAL or DBCS characters, the byte length of each UTF-8 character varies between 1 and 4 bytes. Enterprise COBOL treats a single UTF-8 character as equivalent to a single Unicode code point.

### 4.1 UTF-8 Data Items

There are three ways that Enterprise COBOL uses to define UTF-8 data items.

#### 4.1.1 Fixed Character-Length UTF-8 Data Items

This type of UTF-8 data item is defined when the PICTURE clause contains one or more 'U' characters, or a single 'U' followed by a repetition factor. Take for example the piece of code below:

```
01 NEW-UTF-CHAR PIC U(10).
```

In this case, we define a fixed character-length UTF-8 data item that holds 10 UTF-8 characters that occupy between 10 (n) and 40 (4 \* n) bytes. Since UTF-8 character's byte length varies, 4 \* n bytes are always reserved for UTF-8 item. If there are unused bytes, those will be padded with the UTF-8 blank statement (x'20'). When truncation is performed, it is done on a character boundary.

#### 4.1.2 Fixed Byte-Length UTF-8 Data Items

Like fixed character-length, we define this by the inclusion of the 'U' character in the PICTURE clause. But now, we will add a phrase called BYTE-LENGTH. Observe the code below:

```
01 NEW-UTF-BYTE PIC U BYTE-LENGTH 10.
```

In this case, we define a fixed byte-length UTF-8 data item that holds 10 bytes of UTF-8 data, this translates to up to 10 characters. When these are used to receive characters with byte length smaller than indicated, the unused bytes are padded by the UTF-8 blank statement (x'20').

#### 4.1.3 Dynamic-Length UTF-8 Data Items

Lastly, we have the dynamic-length UTF-8 data items. This is defined when we have a PICTURE clause with the 'U' character and the DYNAMIC LENGTH clause. Observe the code below:

```
01 NEW-UTF-DYN PIC U DYNAMIC LIMIT 10.
```

With dynamic-length UTF-8 data item, there is no restriction on the number of bytes besides the one indicated on the LIMIT phrase of the DYNAMIC LENGTH clause. Unlike the other two definitions, no padding is involved with the dynamic-length UTF-8 data item. Truncation will only occur on the character boundaries if it exceeds the specified limit.

Note that UTF-8 edited, numeric-edited, decimal and external float are not supported.

## 4.2 UTF-8 Literals

There are two types of UTF-8 literals which are supported on Enterprise COBOL.

### 4.2.1 Basic UTF-8 Literals

```
U'character-data'
```

When we define basic UTF-8 literals, the character-data is converted from EBCDIC to UTF-8. If we have double-byte EBCDIC characters, those must be delimited by shift-out and shift-in characters. The amount of Unicode code points which we can represent here varies depending on the size of the UTF-8 characters, but a maximum of 160 bytes (after conversion) is allowed before truncation.

#### 4.2.2 Hexadecimal UTF-8 Literals

```
UX 'hexadecimal-digits '
```

In this case, the hexadecimal-digits are converted to bytes sequences which are used verbatim as the UTF-8 literal values. There is a minimum of 2 hexadecimal digits and a maximum of 320.

#### 4.3 UTF-8 Move Rules and Conversion

Generally speaking, a UTF-8 data item can be moved only to those of category National or UTF-8. While they can receive items from Alphabetic, Alphanumeric, National or UTF-8. If there are any padding or truncation, those are always done at the UTF-8 character.

Additionally, we can use the intrinsic function DISPLAY-OF to convert national to UTF-8 and UTF-8 to alphanumeric or the intrinsic function NATIONAL-OF to convert UTF-8 to national.

**Note :** For more information, please refer to the [Programming Guide](#).

## 5 COBOL Application Programming Interface (API)

API is the acronym for Application Programming Interface. An API allows two applications to communicate. We use API's everyday from our phones, personal computers, using a credit card to make a payment at a point of sale, etc.

Today's digital infrastructure is instrumented and interconnected. It is the API's that enable the "instrumented" network to be "interconnected". As a result, API has become a highly used acronym in the technology arena. The phrase "API Economy" became strategic term since Forbes declared 2017 "The Year of the API Economy".

Business application solutions were architected decades ago using programming language API's. Long before API became a strategic technology category, mainframe application developers understood the acronym as a way to enable a programming language to communicate with other software. The value of being a programmer in any specific programming language increased by understanding and using API's.

- **Enterprise COBOL APIs**
  - **z/OS Middleware**
  - **COBOL API Communication with Middleware**
  - **COBOL EXEC SQL**
  - **COBOL Data Items**
- **SQL Capability within Enterprise COBOL**
  - **Enterprise COBOL Application Programming and SQL Guide**
  - **Db2 Data Base Administration (DBA) vs Application Programming**
- **Lab**
  - **Using VSCode and Zowe Explorer**

### 5.1 Enterprise COBOL APIs

IBM mainframe flagship operating system, z/OS, includes software that has enabled large scale business applications for decades. The software is frequently referred to as 'middleware'. Examples of z/OS 'middleware' is Db2, a relational database, CICS, transactional processor, IMS, both transactional and hierarchical database, and MQSeries, a mechanism to store and forward data between systems asynchronously.

#### 5.1.1 z/OS Middleware

A fundamental capability of z/OS middleware software is communication enablement of programming languages. The z/OS middleware software includes documentation and examples of how any specific programming language can communicate with the specific z/OS middleware software. A programming language, such as Enterprise COBOL, would use documented interfaces and techniques to initiate services and pass data between the COBOL application program and the middleware.

#### 5.1.2 COBOL API Communication with Middleware

Each middleware has unique reserved words available to Enterprise COBOL.

Enterprise COBOL unique API reserved words are in Example 1.

```
EXEC SQL
EXEC CICS
CALL 'MQ...'
CALL 'CBLTDLI'
```

*Example 1. COBOL API Reserved Words*

Each of the above COBOL API's enable the program to communicate with Db2, CICS, MQSeries, and IMS respectively. When the COBOL source program is compiled, the API reserved words expand the number of lines in the COBOL source code. The expanded lines of code does not need to be fully understood by the

COBOL programmer. The COBOL programmer uses an API to accomplish a task within the logic and the middleware expanded code follows through with accomplishing the task.

### 5.1.3 COBOL EXEC SQL

SQL, Structured Query Language, is the documented standard for communicating with all relational databases. Enterprise COBOL is capable of including Db2 for z/OS SQL. A few simple COBOL EXEC SQL reserved words are shown in Example 2.

```

WORKING-STORAGE SECTION.
*****
* SQL INCLUDE FOR SQLCA
*****
EXEC SQL INCLUDE SQLCA END-EXEC.
*****
* SQL DECLARATION FOR VIEW ACCOUNTS
*****
EXEC SQL DECLARE my-acct-tbl TABLE
              (ACCTNO    CHAR(8) NOT NULL,
               LIMIT     DECIMAL(9,2) ,
               BALANCE   DECIMAL(9,2) ,
               SURNAME   CHAR(20) NOT NULL,
               FIRSTN    CHAR(15) NOT NULL,
               ADDRESS1   CHAR(25) NOT NULL,
               ADDRESS2   CHAR(20) NOT NULL,
               ADDRESS3   CHAR(15) NOT NULL,
               RESERVED   CHAR(7) NOT NULL,
               COMMENTS   CHAR(50) NOT NULL)
              END-EXEC.
*****
* SQL CURSORS
*****
EXEC SQL DECLARE CUR1 CURSOR FOR
              SELECT * FROM my-acct-tbl
              END-EXEC.

PROCEDURE DIVISION.
*-----
LIST-ALL.
EXEC SQL OPEN CUR1 END-EXEC.
EXEC SQL FETCH CUR1 INTO :CUSTOMER-RECORD END-EXEC.
PERFORM PRINT-AND-GET1
              UNTIL SQLCODE IS NOT EQUAL TO ZERO.
EXEC SQL CLOSE CUR1 END-EXEC.

```

*Example 2. COBOL SQL Statements*

### 5.1.4 COBOL Data Items

While the EXEC SQL is expanded into additional lines of code at compile time, COBOL needs data items to manage the data passed between the COBOL program and Db2 table.

The fields in the Db2 table record were defined using CREATE TABLE SQL. The EXEC SQL DECLARE in Table xx describes the Db2 table format within the COBOL program. The COBOL programmer with knowledge of the Db2 table format can code the table format or let Db2 for z/OS generate the code using a DCLGEN utility.

Observe “:CUSTOMER-RECORD” in the EXEC SQL FETCH statement. A colon (:) precedes COBOL program defined variables that are used in SQL statements so that Db2 can distinguish a variable name from a column name. Example 3. shows the COBOL program data items describing the COBOL program variable names.

```
*****
* STRUCTURE FOR CUSTOMER RECORD *
*****
01 CUSTOMER-RECORD.
   02 ACCT-NO          PIC X(8) .
   02 ACCT-LIMIT       PIC S9(7)V99 COMP-3 .
   02 ACCT-BALANCE     PIC S9(7)V99 COMP-3 .
   02 ACCT-LASTN      PIC X(20) .
   02 ACCT-FIRSTN     PIC X(15) .
   02 ACCT-ADDR1      PIC X(25) .
   02 ACCT-ADDR2      PIC X(20) .
   02 ACCT-ADDR3      PIC X(15) .
   02 ACCT-RSRVD      PIC X(7) .
   02 ACCT-COMMENT    PIC X(50) .
```

*Example 3. COBOL Data Item for storing variables where Db2 is the data source*

## 5.2 SQL Capability within Enterprise COBOL

Learning SQL is a separate technical skill. The objective of this brief chapter is familiarization with Enterprise COBOL use of SQL API. A COBOL program is capable of any SQL communication with Db2 for z/OS assuming necessary authority is granted. SQL has four categories as outlined in Example 4. Learning SQL is necessary for a COBOL programmer to become proficient with using the Db2 API for a variety of possible applications where COBOL provides the what, how, and when logic of executing specific SQL.

```
DDL - Data Definition Language
CREATE
ALTER
DROP

DML - Data Manipulation Language
SELECT
INSERT
UPDATE
DELETE

DCL - Data Control Language
GRANT
REVOKE

TCL - Transaction Control Language
COMMIT
ROLLBACK
```

*Example 4. SQL Categories*

### 5.2.1 Enterprise COBOL Application Programming and SQL Guide

Db2 for z/OS V12 is the most current release of Db2 at the moment. The Db2 V12 for z/OS Application Programming and SQL Guide is available using internet search SC27-8845, the Db2 for z/OS professional

manual number. Db2 V12 for z/OS SQL Reference is also necessary to advance programming API capability (SC27-8859).

### 5.2.2 Db2 Data Base Administration (DBA) vs Application Programming

In large enterprise, the roles and responsibilities are divided for a number of reasons. The responsibility of the DBA would include the DDL and DCL outlined in Example 4. The DBA is responsible for managing the entire relational data base environment to insure availability, security, performance, etc. The system programmers and DBAs frequently setup the application development procedures for COBOL programmer development, testing, and maintenance of the COBOL business applications. A COBOL application programmer is typically provided documented procedures to follow to apply their COBOL programming and SQL API expertise.

Enterprise COBOL is a learning journey. Each Enterprise COBOL API is a separate learning journey. As is the case with most professional endeavors, learning, repetition, and applying what is learned is re-iterative process leading to advanced skill levels.

## 5.3 Lab

The lab contains data used in previous labs from “COBOL Programming Course #1 - Getting Started” where the data source was sequential data set, then a VSAM data set. The lab provides JCL to create a personal Db2 table in a DBA-created database name using a DBA-created storage group. The DBA-created storage group directs the create tablespace and table to specific disk storage volumes.

The lab contains Enterprise COBOL source code with Db2 APIs along with the JCL to compile and execute the COBOL programs.

### 5.3.1 Using VSCode and Zowe Explorer

Zowe Explorer is currently without the ability to execute Db2 SQL interactively. It is inevitable Zowe Explorer will eventually have the capability of connecting to relational databases and executing SQL.

Therefore, JCL members were created to create and load user tables following examples provided.

1. Submit `zos.public.db2.jcl(db2setup)` The result is new JCL and CBL members copied into personal JCL and CBL libraries
2. SUBMIT JCL(CRETBL) The result is a personal Db2 tablespace, table, indexspace, and index
3. SUBMIT JCL(LOADTBL) The result is data loaded into the personal Db2 tablespace, table, indexspace, and index
4. SUBMIT JCL(SELTBL) Performs an SQL select to verify that your personal Db2 table is properly loaded with data.
5. Edit each COBOL source code member in your CBL partition data set changing all occurrences of Z# to your personal ID. Example - If your ID was Z80001, then change all occurrences of Z# to Z80001.
6. SUBMIT JCL(CBLDB21C) The result is compile of CBL program CBLDB21 and a Db2 Plan needed for program execution
7. SUBMIT JCL(CBLDB21R) The result is execution of COBOL program CBLDB21 to read the Db2 table and write each record from the Db2 table .
8. Two additional COBOL programs with Db2 API exist, CBLDB22 and CBLDB23 using the same Db2 table as the data source.



## 6 COBOL Program Compilation

In the previous course, we briefly mentioned how a COBOL program is compiled. In this chapter, we will deep dive into the Enterprise COBOL compiler and how you have interacted with it.

To restate what we have learned, the compiler will translate the COBOL program we wrote into language that the system can process. It will also list errors in our source statements and provide information on how to debug them. After compilation, we can review the results and correct any detected errors.

As part of the compilation, we need to define the necessary data sets and specify any compiler options necessary for our program.

- **Compilation via JCL**
  - **Catalogued JCL Procedure**
  - **Writing JCL to compile programs**
- **Specifying compiler options**
  - **Specifying options in the PROCESS statement**
  - **Specifying options in JCL**
- **Batch compilation**
  - **Compiler options in a batch compilation**

### 6.1 Compilation via JCL

The primary method of COBOL program compilation we have done in this course is through JCL or Job Control Language. We have primarily used a set of catalogued procedures provided by IBM which reduces the amount of JCL that we need to write.

In the JCL, we need to include the job description, statement to invoke the compiler and definitions of the needed data sets.

#### 6.1.1 Catalogued JCL Procedure

A catalogued procedure is a set of job control statements in a partitioned data set called the procedure library, or proclib for short. Take for example the following JCL which calls the IBM-supplied catalogued procedure IGYWC for compiling an Enterprise COBOL program:

```
//JOB1          JOB1
//STEP1         EXEC PROC=IGYWC
//COBOL.SYSIN DD *
000100 IDENTIFICATION DIVISION
        * (the source code)
...
/*
```

In the example above, the COBOL program we are trying to compile is sourced directly from within the JCL file. If we store our source code in a data set, we can replace the SYSIN DD statement with the appropriate parameters.

We can also override any compiler options which are not explicitly set by using an EXEC statement that includes the required options. Take for example:

```
//STEP1 EXEC IGYWC ,
//          PARM.COBOL='LIST,MAP,RENT'
```

The content of the PARM statement defines the Enterprise COBOL compiler options we are setting for the program that we wrote. We will discuss more details regarding compiler options in a later section.

### Compile procedure (IGYWC)

The first of the supplied catalogued procedures is the single-step IGYWC. It is a procedure for compiling a program, and it will produce an object module. The step which compiled the program is called COBOL. We are required to supply the SYSIN DD statement for the step to indicate the location of the source program:

```
//COBOL.SYSIN DD *      (or appropriate parameters)
```

If we use copybooks in the program we are compiling, we must also supply a SYSLIB DD statement to indicate the location of our copybooks:

```
//COBOL.SYSLIB DD DISP=SHR,DSN=Z99998.COBLIB
```

### Compile and link-edit procedure (IGYWCL)

The second procedure is the two-step IGYWCL. Like the previous IGYWC, it will produce an object module. But it will also supply that module into the binder (or linkage-editor). The additional step which executes the binder is called LKED. This binder will prepare a load module which will be brought into storage for execution.

Just like IGYWC, you will need to supply a SYSIN DD statement and also a SYSLIB DD statement at the COBOL step if you use copybooks.

### Compile, link-edit and run procedure (IGYWCLG)

The third and last of the IBM-supplied catalogued procedures is IGYWCLG. In addition to compiling and passing the object module to the binder, it will also run the program. The last step which runs the compiled and link-edited program is called GO.

Just like the other two procedures, you will need to supply a SYSIN DD statement and also a SYSLIB DD statement at the COBOL step if you use copybooks. Additionally, if your COBOL program refers to any data set during execution, you will need to specify them in the GO step. A valid DDName of up to 8 characters, as specified in the FILE CONTROL paragraph will be needed:

```
//GO.DDName DD DSN=data-set-name
```

## 6.1.2 Writing JCL to compile programs

Chances are you will not need to manually write any JCL to compile a program. However, if the catalogued procedure does not provide you with the flexibility you need, you can write your job control statements. Let us take a look and study the following example:

```
//jobname JOB acctno,name,MSGCLASS=1          (1)
//stepname EXEC PGM=IGYCRCTL,PARM=(options)    (2)
//STEPLIB DD DSN=IGY.V6R3M0.SIGYCOMP,DISP=SHR (3)
//          DD DSN=SYS1.SCEERUN,DISP=SHR
//          DD DSN=SYS1.SCEERUN2,DISP=SHR
//SYSUT1 DD UNIT=SYSALLDA,SPACE=(subparms)    (4)
//SYSUT2 DD UNIT=SYSALLDA,SPACE=(subparms)
//SYSUT3 DD UNIT=SYSALLDA,SPACE=(subparms)
//SYSUT4 DD UNIT=SYSALLDA,SPACE=(subparms)
//SYSUT5 DD UNIT=SYSALLDA,SPACE=(subparms)
//SYSUT6 DD UNIT=SYSALLDA,SPACE=(subparms)
//SYSUT7 DD UNIT=SYSALLDA,SPACE=(subparms)
//SYSUT8 DD UNIT=SYSALLDA,SPACE=(subparms)
//SYSUT9 DD UNIT=SYSALLDA,SPACE=(subparms)
```

```
//SYSUT10 DD UNIT=SYSALLDA,SPACE=(subparms)
//SYSUT11 DD UNIT=SYSALLDA,SPACE=(subparms)
//SYSUT12 DD UNIT=SYSALLDA,SPACE=(subparms)
//SYSUT13 DD UNIT=SYSALLDA,SPACE=(subparms)
//SYSUT14 DD UNIT=SYSALLDA,SPACE=(subparms)
//SYSUT15 DD UNIT=SYSALLDA,SPACE=(subparms)
//SYSMDECK DD UNIT=SYSALLDA,SPACE=(subparms)
//SYSPRINT DD SYSOUT=A (5)
//SYSLIN DD DSN=MYPROG,UNIT=SYSALLDA, (6)
// DISP=(MOD,PASS),SPACE=(subparms)
//SYSIN DD DSN=dsname,UNIT=device, (7)
// VOLUME=(subparms),DISP=SHR
```

(1): The JOB statement indicates the beginning of a job.

(2): The EXEC statement specifies that the Enterprise COBOL compiler (IGYCRCTL) is to be invoked.

(3): The DD statement here defines where the Enterprise COBOL compiler resides. You will need to check with your system programmer regarding where the compiler is installed on your system. Alongside them, the Language Environment SCEERUN and SCEERUN2 data sets must be included in the concatenation unless they are available in the LNKLIST.

(4): The SYSUT DD statements define the utility data sets that the compiler will use to process the source program. All SYSUT files must be on direct-access storage devices.

(5): The SYSPRINT DD statement defines the data set that receives output from compiler options.

(6): The SYSLIN DD statement defines the data set that receives output from the OBJECT compiler option.

(7): The SYSIN DD statement defines the data set to be used as input to the job step, or in other words, the source code.

For more information on the input and output data set that the Enterprise COBOL compiler can use, please refer to the [IBM Documentation](#).

## 6.2 Specifying compiler options

The compiler is installed with default compiler options. However, when installing the compiler, the system programmer can fix certain compiler settings. You cannot override any compiler options that are fixed.

For the options that are not fixed, there are several ways in which you can override the default settings: - Code them on the PROCESS or CBL statement in the COBOL source code - Include them in the JCL when you start the compiler - Include them in an SYSOPTF data set, which is one of the input data sets the compiler can use

The compiler will then recognize the options in the following order of precedence from highest to lowest: 1. Fixed installation defaults 2. Values of the BUFSIZE, OUTDD, SQL and SQLIMS compiler options for the first program in a batch 3. Options specified on PROCESS (or CBL) statements, preceding the IDENTIFICATION DIVISION 4. Options specified on the compiler invocation 5. Installation defaults that are not fixed

The precedence options in a SYSOPTF data set will depend on where the OPTFILE compiler option is specified. For example, if OPTFILE is specified in a PROCESS statement, the SYSOPTF options will supersede the options specified in the compiler invocation.

Note that this order of precedence also determines which options are in effect when there are conflicting or mutually exclusive options.

For a full list of compiler options, please refer to the [IBM Documentation](#).

### 6.2.1 Specifying options in the PROCESS statement

Within a COBOL program, you can modify most compiler options in the PROCESS statements. We will need to code the statements before the IDENTIFICATION DIVISION header and before any comment lines or compiler-directing statements.

We can also use CBL as a synonym of PROCESS. One or more blanks will be needed to separate a PROCESS or CBL statement from the first option in the list of options. The options themselves must be separated by a comma or a blank, while no spaces should be inserted between individual options and their suboptions.

Furthermore, we can code more than one PROCESS or CBL statement. If we do so, they must follow one another. Note that your organization can inhibit the use of PROCESS statements by fixing up certain compiler settings. If the PROCESS or CBL statement contains an option that is not allowed, the COBOL compiler will generate an error diagnostic.

Take a look at the following example:

```
PROCESS LIST,MAP.
*-----
IDENTIFICATION DIVISION.
*-----
PROGRAM-ID.      CBL0001
AUTHOR.          Otto B. Fun.
*-----
...
```

### 6.2.2 Specifying options in JCL

We can also specify compiler options using JCL. Take a look at the following example for the catalogued procedures:

```
//STEPS EXEC IGYWC ,
//          PARM.COBOL='LIST,MAP,RENT'
```

Alternatively, if you are making your job control statement:

```
//STEPS EXEC PGM=IGYCRCTL ,
//          PARM='LIST,OBJECT,NOCOMPILE(S)'
```

## 6.3 Batch compilation

We can also compile a sequence of separate COBOL programs through a single invocation of the compiler. We can link the object program produced into one single program object or separate them through the use of the NAME compiler option.

When we compile several programs as part of a single job, we need to determine how many program objects we want and also ensuring each program have the appropriate compiler options and termination sequence.

To create separate program objects, we need to precede each set of objects with the NAME compiler option. When the compiler encounters the option, the first program and all subsequent programs until the next time the NAME compiler option is encountered are link-edited to a single program object.

Additionally, to terminate each program in the sequence, we need to use the END PROGRAM marker. If we omit them, the next program in the sequence will be nested in the preceding program, which may cause a compilation error when a PROCESS statement is encountered.

Take a look at the following example:

```

//jobname JOB acctno,name,MSGLEVEL=1
//stepname EXEC IGYWCL
//COBOL.SYSIN DD *
010100 IDENTIFICATION DIVISION.
010200 PROGRAM-ID PROG1.
. . .
019000 END PROGRAM PROG1.
020100 IDENTIFICATION DIVISION.
020200 PROGRAM-ID PROG2.
. . .
029000 END PROGRAM PROG2.
CBL NAME
030100 IDENTIFICATION DIVISION.
030200 PROGRAM-ID PROG3.
. . .
039000 END PROGRAM PROG3.
/*
//LKED.SYSLMOD DD DSN=&&GOSET
/*
//P2 EXEC PGM=PROG2
//STEPLIB DD DSN=&&GOSET,DISP=(SHR,PASS)
. . .
/*
//P3 EXEC PGM=PROG3
//STEPLIB DD DSN=&&GOSET,DISP=(SHR,PASS)
. . .
/*
//

```

In the JCL, PROG1 and PROG2 are link-edited together to form one program object with the name PROG2. Despite the name, the entry point of this program object will default to the first program in the program object, PROG1. On the other hand, PROG3 is link-edited by itself into a program object with the name PROG3.

### 6.3.1 Compiler options in a batch compilation

As with the compilation of a single program, the order of precedence for each program in the batch sequence is the same.

However, note that if the current program being compiled does not contain a CBL or PROCESS statements, the compiler will use the settings of options in effect for the previous program. On the other hand, if a CBL or PROCESS statement is included, it will be resolved together with the options in effect for the previous program.

Additionally, if any program needs the BUFSIZE, DEFINE, OUTDD, SQL, or SQLIMS option, that option must be in effect for the first program in the sequence.

## 7 Multithreading and COBOL

We can run COBOL programs in multiple threads. To do so, we compile using the `THREAD` compiler option.

Note that COBOL does not directly support the management of the program threads. But we can run the programs that we compile in a multithreaded application server. So, other programs can call the COBOL program we wrote in a way that enables it to run in multiple threads.

### Choosing LOCAL-STORAGE or WORKING-STORAGE

- Data items in the LOCAL-STORAGE SECTION are allocated for each instance of a program invocation. So in this case, each copy of the program will have its copy of the LOCAL-STORAGE data.
- Data items in the WORKING-STORAGE SECTION are only allocated once for each program, so they will be available in their last-used state to all programs invocation.

So, if we want to isolate data to an individual invocation, we need to define the data in the LOCAL-STORAGE SECTION. If we decided to define them in the WORKING-STORAGE SECTION, we need to make sure that the data will not be accessed simultaneously from multiple threads, or if we do, write the appropriate serialization code for it.

### 7.1 Multithreading

Let us first understand how multithreading works.

The operating system and multithreaded applications handle execution flow within a *process*, which is the course of events when the program runs. Programs within a process can share resources, and the processes themselves can be manipulated.

Within a process, an application can initiate one or more *threads*, basically a stream of computer instruction that controls it. A multithreaded process begins with one thread and can create more to perform tasks. These threads can run concurrently.

In a multithreaded environment, a COBOL *run unit* is the portion of the process that includes threads that have actively executing COBOL programs. The run unit will continue until no COBOL program is active in any of the threads. Within the run unit, COBOL programs can call non-COBOL programs and vice versa.

Within a thread, control is transferred between separate COBOL and non-COBOL programs. Each separately called program is a *program invocation instance*. Program invocation instances of a particular program can exist in multiple threads within a given process.

### 7.2 THREAD to support multithreading

As mentioned previously, we will need to use the `THREAD` compiler option for multithreading support. Note that using `THREAD` might adversely affect performance due to the serialization logic that is generated.

To run multiple COBOL programs in more than one thread, all of them must be compiled using the `THREAD` and `RENT` compiler option, and link them with the `RENT` option of the binder.

We will also need to use the `THREAD` option to compile object-oriented clients and classes.

### 7.3 Transferring control to multithreaded programs

When we write COBOL programs for a multithreaded environment, we will need to choose appropriate program linkage statements.

Just like single-threaded environments, a called program is in its initial state when it is first called within a run unit and when it is first called after a `CANCEL` to the called program. We need to ensure that the program we want to `CANCEL` is not active on any thread, or a Language Environment severe error will be produced.

## 7.4 Ending multithreaded environment

We can end a multithread program by using GOBACK, EXIT PROGRAM or STOP RUN.

GO BACK will return control to the caller of the program. If the caller is the first program in a thread, the thread will be terminated. If the thread is the initial one in a process, the process will be terminated.

EXIT PROGRAM runs the same way as GO BACK, except from the main program where it has no effect.

STOP RUN will terminate the entire Language Environment process and return control to the caller of the main program (which might be the operating system). All threads in the process will also be terminated.

## 7.5 Processing files with multithreading

In threaded applications, we can code COBOL statements for input and output in QSAM, VSAM, and line-sequential files.

Each file definition has an implicit serialization lock, which is used with automatic serialization logic during the I/O operations associated with the following statements: OPEN, CLOSE, READ, WRITE, REWRITE, START, DELETE.

However, automatic serialization is not applied to statements specified with the following conditional phrases: AT END, NOT AT END, INVALID KEY, NOT INVALID KEY, AT END-OF-PAGE, NOT AT END-OF-PAGE.

### 7.5.1 File-definition storage

Upon program invocation, the storage associated with file definition (such as FD records) is allocated and available in its last-used state. Therefore, all threads of execution will share this storage. You can depend on automatic serialization for this storage during the execution of the statements mentioned previously, but not between uses of the statements.

### 7.5.2 Serializing file access with multithreading

To take advantage of automatic serialization, we can use one of the recommended following file organization and usage patterns when we access files in threaded programs.

Recommended file organizations: - Sequential organization - Line-sequential organization - Relative organization with sequential access - Indexed organization with sequential access

The recommended pattern for input:

```
OPEN INPUT fn
...
READ fn INTO local-storage-item
...
* Process the record from the local-storage item.
...
CLOSE fn
```

The recommended pattern for output:

```
OPEN OUTPUT fn
...
* Construct output record in local-storage item.
...
WRITE rec from local-storage-item
...
CLOSE fn
```

With other usage patterns, you must ensure that two instances of the program are never simultaneously active on different threads or that serialization logic is coded explicitly by using calls to POSIX services.

To avoid serialization problems, we can define the data items that are associated with the file in the LOCAL-STORAGE SECTION.

## 7.6 Limitation of COBOL with multithreading

In a multithreaded environment, there are some limitations on COBOL programs. In general, we must synchronize access to resources that are visible to the application within a run unit.

- CICS: We cannot run a multithreaded application in CICS. However, programs compiled with the THREAD option can run in CICS as part of an application that does not have multiple threads.
- Recursive: Since we code the programs in a multithreaded application as recursive, we must adhere to all the restrictions and programming constraints that apply to recursive programs.
- Reentrancy: We must compile our multithreading programs with the RENT compiler option and link them with the RENT option of the binder.
- AMODE: We must run multithreaded applications with AMODE 31. However, programs compiled with the THREAD option can run with AMODE 24 as part of an application that does not have multiple threads.
- Older COBOL programs: To run your COBOL programs on multiple threads of a multithreaded application, we must compile them with Enterprise COBOL using the THREAD option.

To see more details on the limitation of COBOL with multithreading, check out the [Programming Guide](#).



## 8 Program tuning and simplification

In the previous chapters, we have seen how you could code COBOL applications. But now, let us explore how to improve them.

When a program is comprehensible, we can assess its performance. However, if the opposite is true, it can make your application difficult to understand and maintain, thus hindering optimization.

To improve performance, we should take note of the following things:

- The underlying algorithm of your business logic
- Data structure

Having a robust algorithm with the appropriate data structure is essential to improve performance.

We can also write programs that result in more efficient use of the available services. We can also use coding techniques to improve our productivity.

If you are interested in learning more about performance tuning with COBOL, check out the [Enterprise COBOL for z/OS Performance Tuning Guide](#).

- **Optimal programming style**
  - Using structured programming
  - Factoring expressions
  - Using symbolic constants
- **Choosing efficient data types**
  - Efficient computational data types
  - Consistent data types
  - Efficient arithmetic expressions
  - Efficient exponentiations
- **Handling tables efficiently**
- **Choosing compiler features to enhance performance**

### 8.1 Optimal programming style

Enterprise COBOL came with an in-build optimizer, and the coding style we use can affect how it handles our code. We can improve optimization through the use of structured programming techniques, factoring expressions, using symbolic constants or grouping constant and duplicate computations.

#### 8.1.1 Using structured programming

Using structured programming statements, such as EVALUATE and inline PERFORM, can make our program more comprehensible and generates a more linear control flow, which enables the optimizer to produce a more efficient code.

We can also use top-down programming constructs. In simpler term, we would work with a very general overview of what our program should do, before using that to build upon the required operations. In COBOL, out-of-line PERFORM statements are a natural way of doing top-down programming. It can be as efficient as an inline PERFORM because the compiler can simplify or remove the linkage code.

Before we continue, let us talk a bit about in-line and out-of-line PERFORM statements. Chances are you have seen them without realizing what they are. Take a look at the example below:

```

PERFORM 010-INITIALIZE
PERFORM UNTIL END-OF-FILE
    READ FILE-DATA INTO WS-DATA
    AT END
        SET END-OF-FILE TO TRUE
    NOT AT END
        PERFORM 020-UPDATE-TRANSACTION
    END-READ
END-PERFORM

```

In the example above, we have two out-of-line PERFORM and one inline PERFORM. An inline PERFORM is executed in the normal flow of a program, while an out-of-line PERFORM will branch to the named paragraph.

It is also suggested to avoid the use of the following constructs:

- ALTER statements
- Explicit GO TO statements
- PERFORM procedures that involve irregular control flow

### 8.1.2 Factoring expressions

We can also factor expressions in our programs to eliminate unnecessary computation. Take a look at the examples below. The first block of code is more efficient than the second block of code.

```

MOVE ZERO TO TOTAL
PERFORM VARYING I FROM 1 BY 1 UNTIL I = 10
    COMPUTE TOTAL = TOTAL + ITEM(I)
END-PERFORM
COMPUTE TOTAL = TOTAL * DISCOUNT

```

```

MOVE ZERO TO TOTAL
PERFORM VARYING I FROM 1 BY 1 UNTIL I = 10
    COMPUTE TOTAL = TOTAL + ITEM(I) * DISCOUNT
END-PERFORM

```

### 8.1.3 Using symbolic constants

If we have a data item that is constant throughout the program, we can initialize it with a VALUE clause and not change it anywhere in the program.

However, if we pass a data item to a subprogram BY REFERENCE, the optimizer will treat it as an external data item and assumes that it is changed at every subprogram call.

## 8.2 Choosing efficient data types

The use of consistent data types can reduce the need for conversions. We can also carefully determine when to use fixed-point and floating-point data types to improve performance.

### 8.2.1 Efficient computational data types

When we use a data item mainly for arithmetic or subscripting purposes, we can code USAGE BINARY on the data description. The operations to manipulate binary data are faster than those for decimal data.

However, when we are dealing with an intermediate result with a large precision, the compiler will use decimal arithmetic. Normally, for fixed-point arithmetic statements, the compiler will use binary arithmetics for

precision of eight or fewer digits. Anything above 18 digits will always be computed using decimal arithmetics, and those in-between can use either form.

Therefore, to produce the most efficient code for a BINARY data item, we need to ensure that it has a sign (indicated with an S in the PICTURE clause) and eight or fewer digits.

But for a data item that is larger than eight digits or is used with DISPLAY or NATIONAL data items, we can use PACKED-DECIMAL. The code generated can be as fast as BINARY data items in some cases, especially if the statement is complicated or involves rounding.

To produce the most efficient code for a PACKED-DECIMAL data item, we need to ensure it has a sign (indicated with an S in the PICTURE clause), an odd number of digits, and 15 or fewer digits in the PICTURE clause (since the instructions the compiler use are faster with 15 or fewer digits).

### 8.2.2 Consistent data types

In operations with operands of different types, one of the operands will be converted to the same type as the other. This would require several instructions.

Therefore, to improve performance, we can avoid conversions by using consistent data types and by giving both operands the same usage and appropriate PICTURE specifications.

### 8.2.3 Efficient arithmetic expressions

Computation of arithmetic expressions that are evaluated in floating point is most efficient when little or no conversion is involved. We can use operands that are COMP-1 or COMP-2 to produce the most efficient code.

We can also define integer items as BINARY or PACKED-DECIMAL with nine or fewer digits to enable quick conversion to floating-point data. Note that conversion from COMP-1 or COMP-2 to a fixed-point integer with nine or fewer digits is efficient when the value of the COMP-1 or COMP-2 item is less than 1,000,000,000.

### 8.2.4 Efficient exponentiations

We can use floating point for exponentiations for large exponents to achieve faster and more accurate results. For example, the first statement below is computed more quickly and accurately compared with the second statement:

```
COMPUTE FIXED-POINT1 = FIXED-POINT2 ** 100000.E+00
COMPUTE FIXED-POINT1 = FIXED-POINT2 ** 100000
```

By using floating-point exponent, the compiler will use floating-point arithmetics to compute the exponentiations.

## 8.3 Handling tables efficiently

We can also use several techniques to improve the efficiency of your table-handling applications.

To refer to table elements efficiently, we can:

- **Use indexing rather than subscripting.** Since the value of the index is already has the element size factored into it, preventing any need in calculating during run time.
- **Use relative indexing.** Relative index references can be executed at least as fast as direct index references, and sometimes faster.

Regardless of how you reference your table elements, we can also:

- **Specify the element length to match that of related tables.** This will enable the optimizer to reuse the index or subscript computed for one table.

- **Avoid errors in reference by coding index and subscript checks into your program.**

We can also improve the efficiency of tables by:

- Using binary data items for all subscripts
- Using binary data items for variable-length table items
- Using fixed-length data items whenever possible
- Organize tables according to the type of search method used

## 8.4 Choosing compiler features to enhance performance

Our choice of performance-related compiler options can affect how well our program is optimized. We may have a customized system that requires certain options to be set for optimum performance. We can choose compiler features by following these steps:

1. Review the listed option settings to see your system defaults.
2. Determine which options are fixed, and thus nonoverridable, by checking with your system programmer.
3. For options that are not fixed, we can select performance-related options for compiling our program.  
Note that it is best practice to confer with your system programmer to ensure that the options you choose are appropriate for programs at your site.

Another compiler feature to consider is the `USE FOR DEBUGGING ON ALL PROCEDURES` statement which can greatly affect the compiler optimizer. The use of the `ON ALL PROCEDURES` option will generate extra code at each transfer to a procedure name. Although these are useful for debugging, they will make your program larger and thus inhibit optimization.

For a listing of performance-related compiler options, please check the [IBM Documentation](#).

## 9 COBOL Challenges

As you have now handled some basic exercises, we have prepared a new section containing more advanced exercises that test your ability to resolve bugs and other issues in COBOL programs. Each exercise will have a short description and a goal to be accomplished.

Happy Coding!

- **COBOL Challenge - Debugging**
- **COBOL Challenge - The COVID-19 Reports**
- **COBOL Challenge - The Unemployment Claims**
- **Hacker News Rankings for Mainframe/COBOL Posts**

## 9.1 COBOL Challenge - Debugging

It is 2020 in Washington, D.C. John Doe runs a program which provides financial reports on US Presidents and tallies the number of reports from the state of Virginia. Everything seems OK. (see below)

ZOWE

> DATA SETS

> UNIX SYSTEM SERVICES (USS)

✓ JOBS

- > Favorites
- ✓ ca32
  - ✓ BLD RUN(JOB46247) - CC 0...
    - JES2:JESMSG LG(2)
    - JES2:JESJCL(3)
    - JES2:JESYSMSG(4)
    - COBRUN:SYS PRINT(101)
    - COBRUN:SYS PRINT(102)
    - COBRUN:PRTLINE(104)

BLDRUN.JOB46247.PRTLINE X

33	19131921	WILSON	\$1,000,000.00	\$84,033.13
34	19211923	HARDING	\$1,000,000.00	\$11,829.27
35	19231929	COOLIDGE	\$1,000,000.00	\$10,619.20
36	19291933	HOOVER	\$1,000,000.00	\$31,318.33
37	19331945	ROOSEVELT II	\$5,000,000.00	\$31,310.23
38	19451953	TRUMAN	\$5,000,000.00	\$60,992.53
39	19531961	EISENHOWER	\$5,000,000.00	\$32,502.50
40	19611963	KENNEDY	\$1,700,000.00	\$5,084,035.13
41	19631969	JOHNSON II	\$1,700,000.00	\$833.13
42	19691974	NIXON	\$1,700,000.00	\$600.34
43	19741977	FORD	\$1,700,000.00	\$5,051,318.40
44	19771981	CARTER	\$100,000.00	\$3,118,826.10
45	19811989	REAGAN	\$100,000.00	\$50,278.80
46	19891993	BUSH	\$100,000.00	\$40,793.10
47	19932001	CLINTON	\$100,000.00	\$8,118,313.14
48	20012009	BUSH II	\$100,000.00	\$31,313.20
49	20092017	OBAMA	\$9,950,000.00	\$92,311.00
50	20172020	TRUMP	\$8,100,000.00	\$10.00
51	Virginia Clients = 008			
52				

DEBUG CONSOLE

PROBLEMS

...

Filter. E.g.: text, \*\*/\*.ts, !\*\*/node\_modules/\*\*

No problems have been detected in the workspace so far.

John is satisfied, as he can see that everything is working as it should be. He calls it a day and goes home.

The next day, when he comes back to the office, his colleague Mari tells him “I’ve made some changes to one of your programs so that it also tallies the number of presidents who spent more than their allowed limit. Check it out.”

He runs his usual reports and sees the following:

ZOWE

DATA SETS

UNIX SYSTEM SERVICES (USS)

JOBS

Favorites

ca32

BLDRUN(JOB52697) - CC 0...

JES2JESMSG(2)
JES2JESJCL(3)
JES2JESYSMSG(4)
COBRUN:SYSPRINT(101)
COBRUN:SYSPRINT(102)
COBRUN:PRTLINE(104)

BLDRUN.JOB52697.PRTLINE

33	19131921	WILSON	\$1,000,000.00	\$84,033.13
34	19211923	HARDING	\$1,000,000.00	\$11,829.27
35	19231929	COOLIDGE	\$1,000,000.00	\$10,619.20
36	19291933	HOOVER	\$1,000,000.00	\$31,318.33
37	19331945	ROOSEVELT II	\$5,000,000.00	\$31,310.23
38	19451953	TRUMAN	\$5,000,000.00	\$60,992.53
39	19531961	EISENHOWER	\$5,000,000.00	\$32,502.50
40	19611963	KENNEDY	\$1,700,000.00	\$5,084,035.13
41	19631969	JOHNSON II	\$1,700,000.00	\$833.13
42	19691974	NIXON	\$1,700,000.00	\$600.34
43	19741977	FORD	\$1,700,000.00	\$5,051,318.40
44	19771981	CARTER	\$100,000.00	\$3,118,826.10
45	19811989	REAGAN	\$100,000.00	\$50,278.80
46	19891993	BUSH	\$100,000.00	\$40,793.10
47	19932001	CLINTON	\$100,000.00	\$8,118,313.14
48	20012009	BUSH II	\$100,000.00	\$31,313.20
49	20092017	OBAMA	\$9,950,000.00	\$92,311.00
50	20172020	TRUMP	\$8,100,000.00	\$10.00
51	V	ADAMS II004	John Quincy	
52	ACCOUNTS OVERLIMIT		0046	
53				

DEBUG CONSOLE
PROBLEMS

Filter. E.g.: text, \*\*/\*.ts, !\*\*/node\_modules/\*\*

No problems have been detected in the workspace so far.

Clearly, Mari's changes to the program that generates the reports have broken something.

Can you fix the code to get the correct result? The new source code is named **CBL0106** and the JCL is **CBL0106J**. In case you get stuck, the solution is in the file **CBL0106C**.

You can find them in the github repository for the COBOL course, in the subfolder **/COBOL Programming Course #2 - Advanced Topics/Challenges/Debugging**.

## 9.2 COBOL Challenge - The COVID-19 Reports

Today, you are tasked to create a COVID-19 Summary Report of all the countries around the world, using information from the COVID19API website.

### 9.2.1 Instructions

1. Extract the response from this API: <https://api.covid19api.com/summary>. You will receive a JSON file that is similar to the image below:

```
1  {
2    "Global": {
3      "NewConfirmed": 70871,
4      "TotalConfirmed": 2470922,
5      "NewDeaths": 4940,
6      "TotalDeaths": 169952,
7      "NewRecovered": 21835,
8      "TotalRecovered": 645094
9    },
10   "Countries": [
11     {
12       "Country": "ALA Aland Islands",
13       "CountryCode": "AX",
14       "Slug": "ala-aland-islands",
15       "NewConfirmed": 0,
16       "TotalConfirmed": 0,
17       "NewDeaths": 0,
18       "TotalDeaths": 0,
19       "NewRecovered": 0,
20       "TotalRecovered": 0,
21       "Date": "2020-04-21T16:08:17Z"
22     },
23     {
24       "Country": "Afghanistan",
25       "CountryCode": "AF",
26       "Slug": "afghanistan",
27       "NewConfirmed": 30,
28       "TotalConfirmed": 1026,
29       "NewDeaths": 3,
30       "TotalDeaths": 36,
```

2. Convert that file to CSV format. It should look like this. In my example, I only chose the “Countries” part.



```

build > countries.txt
1 Country,CountryCode,Slug,NewConfirmed,TotalConfirmed,NewDeaths,TotalDeaths,NewRecovered,TotalRecovered
2 "ALA Aland Islands","AX","ala-aland-islands",0,0,0,0,0,"2020-04-22T16:54:51Z"
3 "Afghanistan","AF","afghanistan",66,1092,0,36,15,150,"2020-04-22T16:54:51Z"
4 "Albania","AL","albania",25,609,0,26,18,345,"2020-04-22T16:54:51Z"
5 "Algeria","DZ","algeria",93,2811,8,392,53,1152,"2020-04-22T16:54:51Z"
6 "American Samoa","AS","american-samoa",0,0,0,0,0,"2020-04-22T16:54:51Z"
7 "Andorra","AD","andorra",0,717,0,37,34,282,"2020-04-22T16:54:51Z"
8 "Angola","AO","angola",0,24,0,2,0,6,"2020-04-22T16:54:51Z"
9 "Anguilla","AI","anguilla",0,0,0,0,0,"2020-04-22T16:54:51Z"
10 "Antarctica","AQ","antarctica",0,0,0,0,0,"2020-04-22T16:54:51Z"
11 "Antigua and Barbuda","AG","antigua-and-barbuda",0,23,0,3,4,7,"2020-04-22T16:54:51Z"
12 "Argentina","AR","argentina",90,3031,11,147,103,840,"2020-04-22T16:54:51Z"
13 "Armenia","AM","armenia",62,1401,2,24,29,609,"2020-04-22T16:54:51Z"
14 "Aruba","AW","aruba",0,0,0,0,0,"2020-04-22T16:54:51Z"
15 "Australia","AU","australia",0,6547,0,67,0,4124,"2020-04-22T16:54:51Z"
16 "Austria","AT","austria",78,14873,21,491,340,10971,"2020-04-22T16:54:51Z"
17 "Azerbaijan","AZ","azerbaijan",44,1480,1,20,74,865,"2020-04-22T16:54:51Z"
18 "Bahamas","BS","bahamas",5,65,0,9,1,12,"2020-04-22T16:54:51Z"
19 "Bahrain","BH","bahrain",66,1973,0,7,15,784,"2020-04-22T16:54:51Z"
20 "Bangladesh","BD","bangladesh",434,3382,9,110,2,87,"2020-04-22T16:54:51Z"
21 "Barbados","BB","barbados",0,75,0,5,6,25,"2020-04-22T16:54:51Z"
22 "Belarus","BY","belarus",459,6723,4,55,63,577,"2020-04-22T16:54:51Z"
23 "Belgium","BE","belgium",973,40956,170,5998,107,9002,"2020-04-22T16:54:51Z"
24 "Belize","BZ","belize",0,18,0,2,0,2,"2020-04-22T16:54:51Z"
25 "Benin","BJ","benin",0,54,0,1,0,27,"2020-04-22T16:54:51Z"
26 "Bermuda","BM","bermuda",0,0,0,0,0,"2020-04-22T16:54:51Z"
27 "Bhutan","BT","bhutan",1,6,0,0,0,2,"2020-04-22T16:54:51Z"
28 "Bolivia","BO","bolivia",34,598,1,34,6,37,"2020-04-22T16:54:51Z"
29 "Bosnia and Herzegovina","BA","bosnia-and-herzegovina",33,1342,2,51,56,437,"2020-04-22T16:54:51Z"

```

- Using Zowe, upload the CSV file to the mainframe.

**Hint:** You can use the command `zowe files ul ftds "file location" "dataset name"` to upload the CSV file to the mainframe.

- Create a new member in your \*.CBL data set to write your COBOL program.

**Hint:** You can create a member using Zowe Explorer or Zowe CLI.

- Write a COBOL program that reads the uploaded CSV file and reformats it to display the contents like this:

```

*****
DATE: 2020-04-22
TIME: T16:54:5
COUNTRY: "Antigua and Barbuda"
COUNTRY CODE: "AG"
SLUG: "antigua-and-barbuda"
NEW CONFIRMED CASES: 00000
TOTAL CONFIRMED CASES: 00023
NEW DEATHS: 00000
TOTAL DEATHS: 00003
NEW RECOVERIES: 00004
TOTAL RECOVERIES: 00007
*****

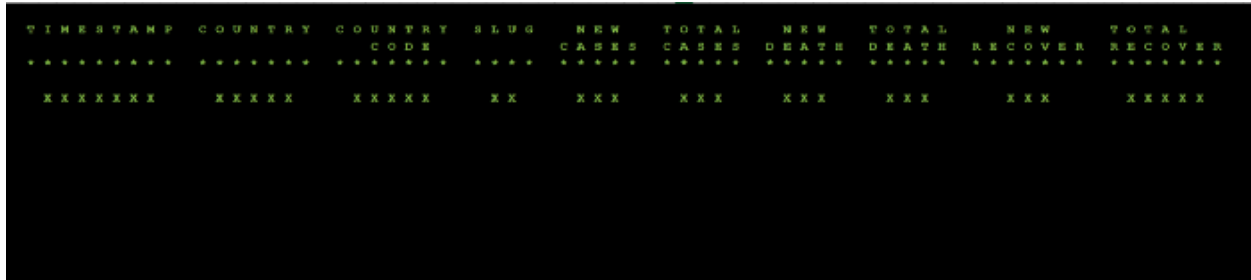
```

- Compile and test your work.

### 9.2.2 Advanced Tasks

If you want a more challenging approach, try the optional tasks below:

- Reformat the data into a Report Form like this:



TIMESTAMP	COUNTRY	COUNTRY CODE	SLUG	NEW CASES	TOTAL CASES	NEW DEATH	TOTAL DEATH	NEW RECOVER	TOTAL RECOVER
*****	*****	*****	****	*****	*****	*****	*****	*****	*****
XXXXXXX	XXXXXX	XXXXXX	XX	XXX	XXX	XXX	XXX	XXX	XXXXXX

- Automate. Using NPM and Zowe CLI, run all these steps and create a “one click” COBOL build similar to this:



### 9.2.3 Solution

To check the solution, refer to the blog post [here](#).

Happy Coding!

*Disclaimer: This challenge is also posted in [Medium.com](#).*

## 9.3 COBOL Challenge - The Unemployment Claims

Now let's try a more advanced challenge! Your task is to create an end-to-end solution. Our end goal is to build an application that will fire Zowe APIs to the mainframe and display the result in the application. This is how the flow would look:



*Of course, you do not have to complete the whole challenge if you do not want to. But it would be great if you do*

### 9.3.1 Our Data

The data that we are going to use will come from <https://www.data.gov/>. According to their website, this is a repository of data that is available for public use. For more information, please visit their website.

To be more specific, we are going to get the monthly unemployment claims of the state of Missouri. I chose this because it is separated according to different categories:

- **By Age:** <https://catalog.data.gov/dataset/missouri-monthly-unemployment-claims-by-age>
- **By Ethnicity:** <https://catalog.data.gov/dataset/missouri-monthly-unemployment-claims-by-ethnicity>
- **By Industry:** <https://catalog.data.gov/dataset/missouri-monthly-unemployment-claims-by-industry>
- **By Race:** <https://catalog.data.gov/dataset/missouri-monthly-unemployment-claims-by-race>
- **By Gender:** <https://catalog.data.gov/dataset/missouri-monthly-unemployment-claims-by-sex>

You can consume the data in different formats such as CSV, RDF, JSON or XML. You can choose whatever format you like.

### 9.3.2 Use Case

You are given a new set of data for The Unemployment Claims. Your tasks are as follows:

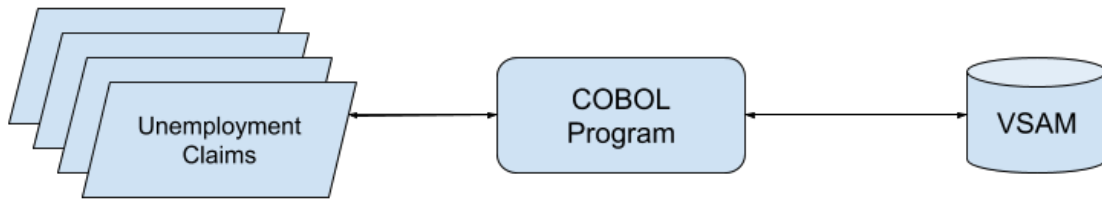
- To create a new database for the new set of data and combine the data based on the Record ID field.
- To provide a way for other COBOL programs and other applications to access this newly created database.
- To create a report specifying all the information available in the newly created database. The report will contain, but not be limited to, the following information: Record ID, Age, Ethnicity, Industry, Race and Gender.

### 9.3.3 Instructions

1. Create a database. This can be done in various ways but the easiest one the I could think of is a VSAM file. First, create a COBOL program that will consume your data. Then, using the RECORD-ID as the key (which is more visible when reading the CSV file), create a KSDS VSAM file and store all the information there.

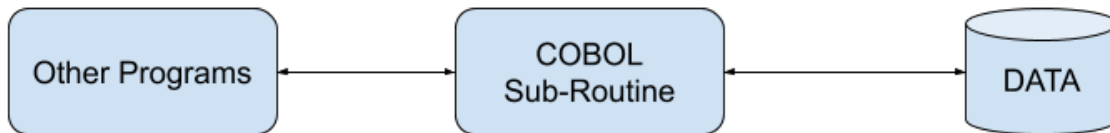
*What this means is that one record will have the RECORD-ID as the key and all the information from The Monthly Unemployment Claims (Fields from Age, Ethnicity, Industry, Race and Gender) will be added or connected to the RECORD-ID.*

The flow would look like this:



2. Create a COBOL sub-routine. This program will allow other programs to read the data from the VSAM file. This sub-routine should be able to perform the following tasks:

- accept requests to get information about a specific record ID.
- *(Optional)* accept requests to get information about all the records inside the database. What does this mean? It means that instead of providing a record ID, I could provide an indicator that I want to create a report of all the records inside the database.

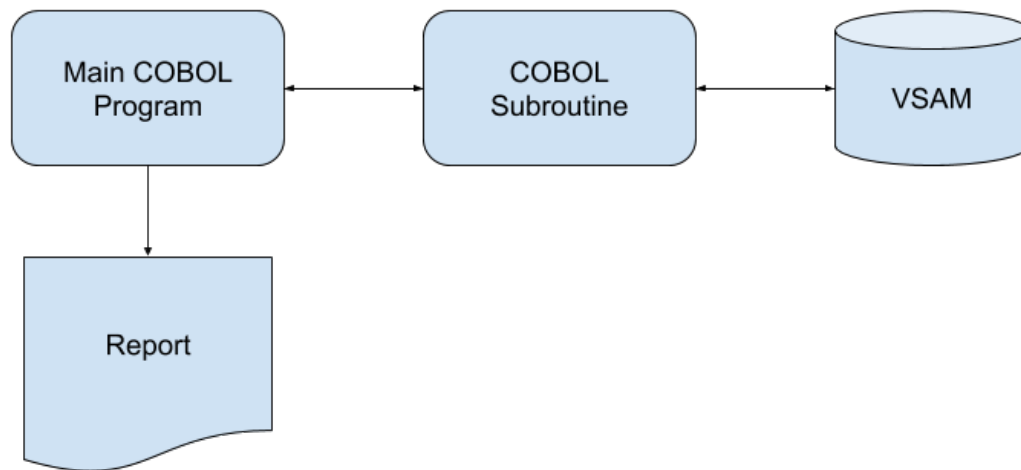


*The purpose of the COBOL sub-routine is to allow other COBOL programs or other application to access the information inside the VSAM file.*

3. Create a Main COBOL Program. This program will create a report based on the records inside the newly created database. The process is as follows:

- The program calls the COBOL sub-routine passing the Record ID or, optionally, an indicator that you want to print all records in the database.
- It receives the response from the sub-routine.
- It processes the response and generates a report. This report can be a formal report or just a display in the SYSOUT. It's up to you.

The flow should look like this:

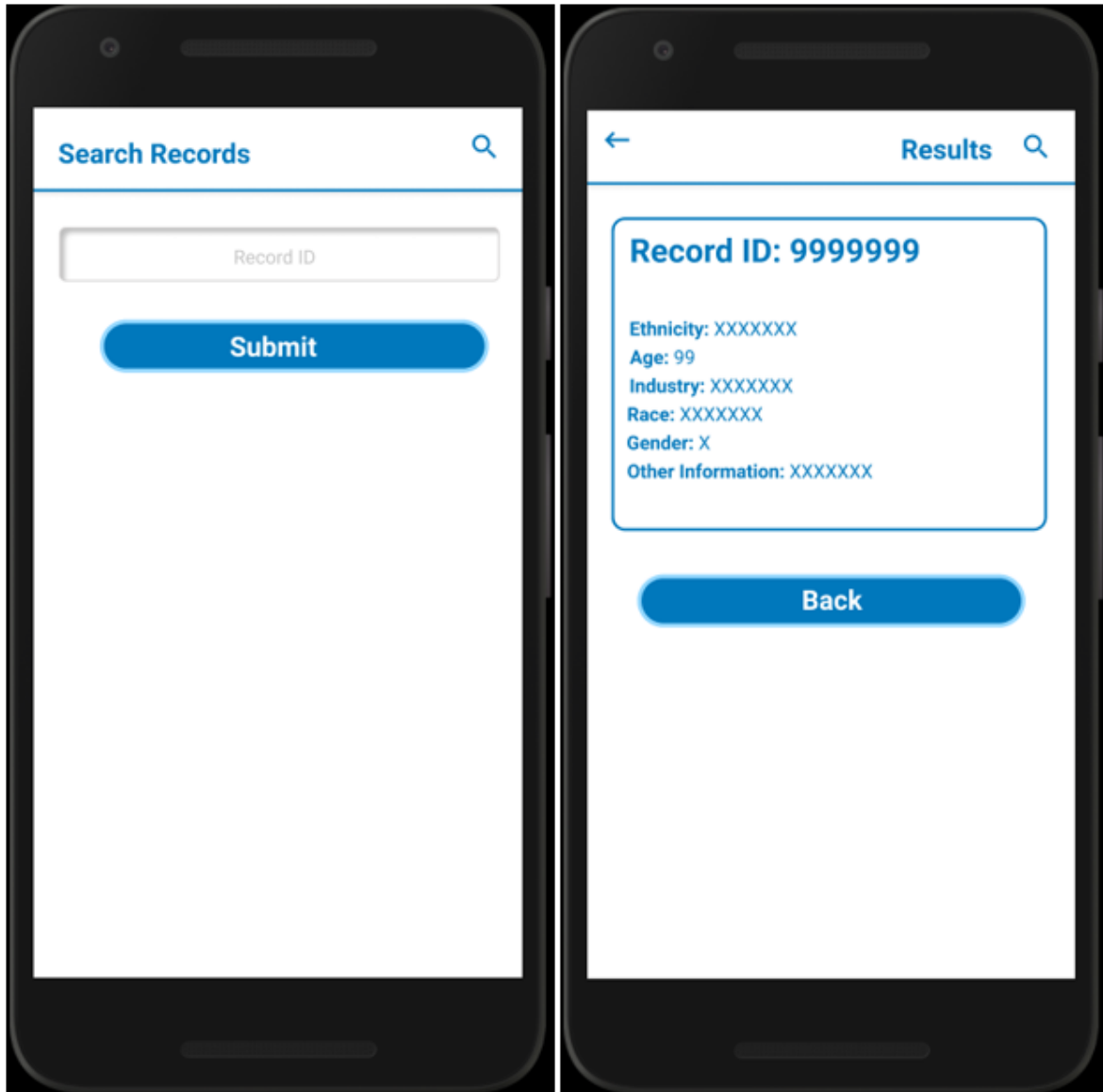


4. Create your JCLs.

*By this point, if you choose to do the exercise using COBOL programs only, you should be able to read the data from your VSAM file, process it and generate a report. The generated report could be an information of a specific record or multiple records.*

5. (Optional) Create an application. It can be any type of application; a Mobile App, a Web App or an Electron App. It is up to you. In this application you should be able to view a record by providing a RECORD-ID. The flow would be similar to Step #3.

This is an example of a possible application design:



**Hint:** How can I accomplish this? By using the Zowe CLI NPM package, you can fire Zowe APIs that submit your JCLs and get the results. From there, you can view the output and display it in your application. This article can provide a good example.

**Hint:** What APIs am I going to use? You will use the Jobs Submit API and View Jobs Spool or View Dataset API. For more information, please visit [this site](#).

6. *(Optional)* Create a CI/CD process that will create a nightly build of your application. [This article](#) can help explain that process.

Sample CI/CD Build using CircleCI:



#### 9.3.4 Craving more programming challenges?

Add more functionality to your COBOL Sub-routine like:

- Insert a new record
- Update an existing record
- Delete an existing record

I hope that by taking this challenge, you will be able to learn something new!

Happy Coding!

*Disclaimer: This challenge is also posted in [Medium.com](https://medium.com).*

## 9.4 Hacker News Rankings for Mainframe/COBOL Posts



We will explore the popular Hacker News website for this challenge. Hacker News is an online community started by Paul Graham for sharing “Anything that good hackers would find interesting. That includes more than hacking and startups”.

### 9.4.1 A Little Background

The site offers a dynamic list of posts/stories, submitted by users, each of which could be expanded into its own unique comment thread. Readers can upvote or downvote links and comments, and the top thirty links are featured on the front page. Today, more than five million people read Hacker News each month, and landing a blog post on the front page is a badge of honor for many technologists.

### 9.4.2 Our Goal

We will be working on a Hacker News 2015-2016 dataset from Kaggle with a full year’s worth of stories: Our goal is to extract only the Mainframe/COBOL related stories and assign ranking scores to them based on (a simplified version) the published Hacker News ranking algorithm. We will create a front page report that reflects this ranking order. The algorithm works in a way that nothing stays on the front page for too long, so a story’s score will eventually drop to zero over time (the gravity effect). Since our posts are spread out over a year and as older posts will always have a lower (or zero) ranking, we will distort the data so all our stories have the same date and consider only the times in the ranking score calculation. This will give all our posts a fair chance of landing the front page. Our front page report is published at 11:59pm. [Here’s some additional information on the ranking.](#)

### 9.4.3 The Plan

- There are different creative ways of accomplishing this but here’s our plan: We will have a COBOL program that reads the input CSV file and retrieves only the *Mainframe/COBOL* stories. It then calculates the ranking score for the stories by factoring in the time they were posted and the number of votes they received. Each of the records is then written to an output dataset along with the ranking score.
- We will then use DFSORT to sort the output dataset on ranking score, highest to lowest and display the posts as a simple report mimicking the front page.

Let’s get started! 1. Take a look and familiarize yourself with the dataset on `z/OS: ZOS.PUBLIC.HACKER.NEWS`. This is a CSV file that serves as input to your COBOL program. The file was created by downloading [this Kaggle dataset](#), removing the lengthy URL column that is of no relevance to us and uploading it to `z/OS`. You can directly reference this DS in your JCL. Please avoid making a copy as it is fairly large with around 300,000 records.



```

id,title,num_points,num_comments,author,created_at
12579008,You have two days to comment if you want stem cells to be classified as your own,1,0,altsta
12579005,SQLAR the SQLite Archiver,1,0,blacksqr,9/26/2016 3:24
12578997,What if we just printed a flatscreen television on the side of our boxes?,1,0,pavel_lishin,
12578989,algorithmic music,1,0,poindontcare,9/26/2016 3:16
12578979,How the Data Vault Enables the Next-Gen Data Warehouse and Data Lake,1,0,markgainor1,9/26/20
12578975,Saving the Hassle of Shopping,1,1,bdoux,9/26/2016 3:13
12578954,Macalifa A new open-source music app for UWP that won't suck,1,0,thecodrr,9/26/2016 3:06
12578942,GitHub theweavrs/Macalifa: A music player written for UWP,1,0,thecodrr,9/26/2016 3:04
12578919,Google Allo first Impression,3,0,jandll,9/26/2016 2:57
12578918,Advanced Multimedia on the Linux Command Line,1,0,mynameislegion,9/26/2016 2:56
12578908,Ask HN: What TLD do you use for local development?,4,7,Sevrene,9/26/2016 2:53
12578893,Muroc Maru,1,0,x43b,9/26/2016 2:46
12578879,Why companies make their products worse,4,0,RachelF,9/26/2016 2:40
12578866,Tuning AWS SQS Queues,3,0,srt32,9/26/2016 2:37
12578857,The Promise of GitHub,2,0,ttam,9/26/2016 2:34
12578834,Joint R&D Has Its Ups and Downs,1,0,Lind5,9/26/2016 2:28
12578831,IBM announces next implementation of Apples Swift developer language,2,0,phodo,9/26/2016 2:
12578822,Amazons Algorithms Dont Find You the Best Deals,1,1,yarapavan,9/26/2016 2:26
12578816,Ruffled Feathers,1,0,Thevet,9/26/2016 2:23
12578806,The Veil of Ignorance Design and Accessibility,3,0,muratmutlu,9/26/2016 2:21
12578796,OMeta#: Who? What? When? Where? Why? (2008),1,0,adamnemecek,9/26/2016 2:18
12578791,Burning Ship fractal,2,0,colinprince,9/26/2016 2:17
12578786,From Hiroko to Susie: The untold stories of Japanese war brides,2,0,kawera,9/26/2016 2:16
12578753,ROBOLUTION:Robocalyptic Themed Machine Learning and Computer Vision Tutorials,2,0,v3ss0n,9/
12578738,Segas Plans for World Domination (1993),2,0,luu,9/26/2016 2:04
12578725,Google Car: Sense and Money Impasse,4,0,kawera,9/26/2016 2:01
12578705,Why an open Web is important when sea levels are rising,1,0,mynameislegion,9/26/2016 1:58
12578700,Forever 23: The Rapid Rise and Sudden Disappearance of Velva Darling,1,0,samclemens,9/26/20
12578694,Emergency dose of epinephrine that does not cost an arm and a leg,2,1,dredmorbis,9/26/2016

```

2. Create your COBOL program in <userid>.CBL using VS Code with the Code4z extension installed and enabled – This program will :

1. Read in each record in the input CSV file
2. Select only the records that have mention of the words **Mainframe** or **COBOL** (ignore case) in the Title field
3. Calculate the ranking score for each record based on the number of votes it received and the time it was posted (Ignore date as we assume all posts were created on the same date)

$$score = \frac{(votes - 1)^{0.8}}{(age_{hours} + 2)^{1.8}}$$

4. Write the record to an output file along with the ranking score

```

01  HACK-IN-FIELDS.
05  HACK-IN-ID          PIC X(8).
05  HACK-IN-TITLE       PIC X(96).
05  HACK-IN-POINTS      PIC 9(4).
05  HACK-IN-COMMENTS    PIC 9(4).
05  HACK-IN-AUTHOR      PIC X(15).
05  HACK-IN-CREATE-DT   PIC X(16).

01  HACK-OUT-FIELDS.
05  HACK-OUT-ID         PIC X(8).
05  HACK-OUT-TITLE      PIC X(96).
05  HACK-OUT-POINTS     PIC 9(4).
05  HACK-OUT-COMMENTS   PIC 9(4).
05  HACK-OUT-AUTHOR     PIC X(15).
05  HACK-OUT-TIME       PIC X(05).
05  HACK-OUT-RANKING-SCORE PIC S9999V999999 COMP-3.

```

3. Copy/Modify/Create a JCL in <userid>.JCL for compiling/linking and running the program against input/output datasets.
4. Submit the job (via Zowe Explorer or Zowe CLI), debug and test to create the output dataset.
5. Next add a new step in the JCL member to run the DFSORT utility on the output dataset from the previous step. The sort should be done on the ranking score field, from highest to lowest. Use DFSORT to also print headers for our front page. As this is a new utility not covered in the course, please check out these links to explore this very powerful and versatile tool:

Getting started with DFSORT

Example with DFSORT

6. Run and debug until the front page looks ready! Which posts ranked among the highest? Here's a look at the generated report:

							Hacker News Front Page 07/01/2020 at 23:59				
							All Mainframe/COBOL stories				
ID	Title						Points	Comments	Author	Time	Score
11376711	An 18 Year Old Buys a Mainframe [video]						653	174	tbatchelli	19:33	6.254802
12096250	"Interviewing my mother a mainframe COBOL programmer"						534	273	Svenskunganka	18:35	4.137754
11717632	A Node.js bridge for COBOL						73	52	reimertz	21:31	2.069649
12039703	LzLabs launches product to move mainframe COBOL code to Linux cloud						30	2	CrankyBear	21:33	1.013454
10880931	IBM ported Go to s390x mainframes						220	120	pythonist	14:34	0.930646
10447581	Qui-binary arithmetic: how a 1960s IBM mainframe does math						76	8	JoachimS	17:36	0.688492
10408692	Fixing the core memory in a vintage IBM 1401 mainframe						89	13	kens	16:30	0.626695
11112124	IBM Launches New Mainframe with Focus on Security and Hybrid Cloud						40	21	protomyth	18:45	0.532160
12317098	COBOL in the cloud						2	0	Rauchg	23:53	0.263029
11763733	Finally: Brainfuck is ready for the mainframe/enterprise						20	11	flok	17:43	0.235401
12333862	Mainframes as a lifestyle choice						113	44	k4rtik	3:09	0.156296
10384287	Open source projects for modern COBOL development						53	27	vezzy-fnord	0:07	0.067588
10964105	C++ getting a 50 year old cobol feature						2	0	petke	21:29	0.066714
10191432	Visual Studio 2015 now supports COBOL desktop and web apps						9	0	lonefounder	13:54	0.059502
11856986	Puppet DevOps comes to the mainframe						3	0	CrankyBear	19:16	0.056486
11912067	The mainframe is dead. Long live the mainframe (rumor Z Systems division sale)						2	0	protomyth	20:52	0.052944
11536087	Mainframes That Handle Our Most Sensitive Data Are Open to Internet Attacks						4	0	rbafffy	17:28	0.050957
11483353	"Want to Open Your Bank to APIs? Not with That Mainframe You Don't"						2	0	jackgavigan	20:36	0.048317
12122912	COBOL and Legacy Code as a Systemic Risk						4	0	mooreds	16:56	0.045680
10187828	12-minute Mandelbrot: fractals on a 50 year old IBM 1401 mainframe						2	0	coloneltcb	19:50	0.038021
10531906	Restoring RAM: Fixing memory inside a 50-year-old IBM mainframe						10	1	ghosh	8:31	0.033683
11448764	Eigen library on mainframe ported to zEC13 SIMD						3	1	edelsohn	16:58	0.033246
10801995	"Yes the World Still Needs COBOL Programmers"						3	3	SunTzu55	16:50	0.032379
10647448	Is SystemVerilog the COBOL of Electronic Design?						16	15	BooneJS	3:15	0.031541
11816903	Unit-testing COBOL programs on IBM mainframes						2	0	rbafffy	18:27	0.026389
11405514	First COBOL Coding Bootcamp Grace Hopper Academy						3	1	dy	15:30	0.025346
10466052	Python vs. R vs. COBOL: Which Is Best for Data Science?						2	0	gexos	17:48	0.022736
11805879	SUSE: OpenStack for the mainframe and beyond						4	0	CrankyBear	11:19	0.019155
10238442	ObjC is our generations COBOL						4	4	tl	10:57	0.018323
11720141	Why are we still using Mainframes?						6	18	parialegend	6:22	0.017078
11108734	IBM Launches New Mainframe with Focus on Security and Hybrid Cloud						4	0	romarv	9:23	0.015328
11372150	What happens when a 18 years old buys a mainframe						8	2	znpy	0:54	0.014361
11922710	Rumors of COBOL's demise have been greatly exaggerated: Meet GnuCOBOL						2	0	alxsanchez	14:11	0.011765
10231540	New Ubuntu distribution for IBM mainframes						3	0	technolo-g	4:17	0.006842
11173371	"OpenMainframe announces new investments tech focus and internship program"						2	0	jrepin	9:46	0.006638
10201440	Pitfalls of Invalid Programs and Data in COBOL						3	0	snake117	0:53	0.005265
10320620	Yo sysprog: Cobol lives on in Visual Studio 2015						1	0	tmaxxcar	19:16	0.000000
11418787	Ask HN: Mimic Mainframe Financial Computations in Real-Time						1	1	furiousandre	1:13	0.000000
10474289	Restoring RAM: Fixing memory inside a 50-year-old IBM mainframe						1	1	TimMeade	21:07	0.000000
10572194	Ask HN: What happened to the old mainframes? Were many preserved?						1	4	hoodoof	1:57	0.000000

Hope you have fun working on this Challenge. Happy COBOL coding!