COBOL

**Cobol Sites**

**Arrays**

<http://stackoverflow.com/questions/16442609/how-to-access-single-character-from-a-string-in-cobol>

**Functions**

<http://www.mainframegurukul.com/tutorials/programming/cobol/cobol-inspect.html>

**File Processing**

<http://cobol.404i.com/fil-a.php>

<http://www.mainframegurukul.com/tutorials/programming/cobol/cobol-file-processing.html>

<http://mainframewizard.com/content/length-each-record-variable-file>

<http://www.mainframegurukul.com/tutorials/programming/cobol/cobol-file-handling.html>

**File Errors**

<http://www.cse.ohio-state.edu/~sgomori/314/filerr.html>

-----------------------------------------------------------------------------------------------------------------------------

**IMP Points**

* **ABBREVATIONS**

**EBCDIC**- Extended Binary Coded Decimal Interchange Code

primarily used on mainframes.

**ASCII  -** American Standard Code for Information Interchange

widely used on PCs.

**COBOL** - COmmon Business Oriented Language

* Array indices are zero based in C-Lang and 1 based in COBOL.
* All cobol pgms must start with 8th column

Cobol verbs should start in AREA-B col 12.

7thcolum is comments one (\*)

* In Cobol **characters are inserted from left to right**

Eg: 01 ws-name PIC X(4) VALUES “SANDEEP”

Then ws-name=SAND

* In Cobol **numbers are inserted from right to left**

Eg: 01 ws-no PIC 9(4) VALUES 123456

Ws-no = 3456

* Data written in another format associate format (**Reference modification)**

**DATA(1:8)**

* **INSPECT verb allows to count and/or replace a character or a group**

of characters. INSPECT has options TALLYING, REPLACING & CONVERTING.

**EG:**

source-string=“AABAbbACABA”

1. **INSPECT** source-string **TALLYING** tally-counter **FOR CHARACTERS**

**BEFORE INITIAL** ‘C’

2. **INSPECT** source-string **TALLYING** tally-counter **FORALL** ‘A’

3. **INSPECT** source-string **TALLYING** tally-counter **FOR ALL** ‘A’

**AFTER INITIAL** ‘B’

4. **INSPECT** source-string **TALLYING** tally-counter **FORLEADING** ‘A’

**5. Trim spaces form a string**

**INSPECT FUNCTION REVERSE (**Source-string**) TALLYING** space-count **FORLEADING** SPACES.

COMPUTE length-of-string = 6 - space-count.

Source-string = " SSET"

**Move Source-string(space-count+1 : length-of-string ) TO ws-target-string.**

* **INSPECT… with REPLACING** option This form replaces a character or group of characters

1. **INSPECT** source-string **REPLACING CHARACTERS BY** ‘#’ **BEFORE INITIAL** ‘C’.
2. **INSPECT** source-string **REPLACING ALL** ‘A’ **BY** ‘#’
3. **INSPECT** source-string **TALLYING** tally-counter **FOR ALL** ‘A’ **AFTER INITIAL** ‘B’**REPLACING ALL** ‘A’ **BY** ‘#’ **AFTER INITIAL** ‘B’

* **Inspect with Converting**

1. **INSPECT** source-string **CONVERTING** ‘BXCY’ **TO** ‘1234’

(Equivalent INSPECT with REPLACING option)

**INSPECT** source-string **REPLACINGALL** ‘B’ **BY** ‘1’

**ALL** ‘X’ **BY** ‘2’

**ALL** ‘C’ **BY** ‘3’

**ALL** ‘Y’ **BY** ‘4’

* A sequential file can be opened in one of the following four modes.

INPUT, OUTPUT, EXTEND and I-O.

* File Control of ENVIRONMENT DIVISION. parameter for cobol

**SELECT WS-INFILE ASSIGN TO INPIWS01**

**FILE STATUS IS WS-INFILE-SW.**

Where status variable in working storage section is

01 **WS-INFILE-SW** PIC X(02) VALUE SPACES.

88 WS-INFILE-SUCESS VALUE '00'.

88 WS-INFILE-EOF VALUE '10'.

* **Variable record** length records have 4 byte field at the start of the record to say how long the record actually is.

01 FILE-SIZE PIC 9(4) COMP-5.

* File definition for variable file

**FD INPUTFL**

**BLOCK CONTAINS 0 RECORDS**

**RECORDING MODE IS V**

**RECORD IS VARYING IN SIZE FROM 1 TO 80 CHARACTERS**

**DEPENDING ON FILE-SIZE.**

**01 INPUTFL-REC PIC X(80).**

* **CURRENT-DATE** is COBOL intrinsic function to get current date, time and difference

between current location time and Greenwich Mean Time.

MOVE **FUNCTION CURRENT-DATE** TO WS-CURRENT-DATE-DATA

This function returns a 20-character alphanumeric field in the below format

01 WS-CURRENT-DATE-DATA.

05 WS-CURRENT-DATE.

10 WS-CURRENT-YEAR PIC 9(04).

10 WS-CURRENT-MONTH PIC 9(02).

10 WS-CURRENT-DAY PIC 9(02).

05 WS-CURRENT-TIME.

10 WS-CURRENT-HOURS PIC 9(02).

10 WS-CURRENT-MINUTE PIC 9(02).

10 WS-CURRENT-SECOND PIC 9(02).

10 WS-CURRENT-MILLISECONDS PIC 9(02).

05 WS-DIFF-FROM-GMT PIC S9(04).

Eg; WS-CURRENT-DATE-DATA contains : **2010111917542857+0800**

* **Types of dates**

**Gregorian date -** YYYYMMDD.(**FUNCTION DATE-OF-INTEGER (integer-date)**

**IntergerDate**(it is a 7 digit date obtained by using function **Integer-OF-Date**)

**Julain Date-** YYYYDDD (**FUNCTION DAY-OF-INTEGER (integer-date)**

)

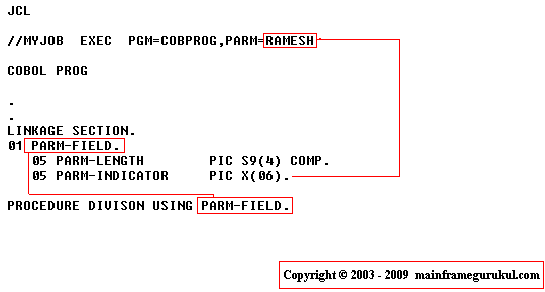
Note: 1600 < YYYY < 9999 and 0 < DDD < 367

* PARM parameter is used to pass information to program

Syntax -> PARM=value

Value is a string can be 1 to 100 characters long.

PASSING PARAMETER USING PARM PARAMETER



PARM-INDICATOR will contain "RAMESH"

PARM-LENGTH contains length of string.

* The PICTURE Clause of a COMPUTATIONAL data item should not contain any character other than 9 or S.
* COMP-1 takes 4 bytes of storage

COMP-2 takes 8 bytes of storage. COMP-2 is more precision than COMP-1.

* COMP-1, COMP-2 don’t have PIC clause because PIC creates relation between data name and data type. Whereas here data is numeric and length is predefined as one-word floating for

COMP-1 and double word floating for COMP-2. So no relation is required using PIC clause.

* In COMP-3one digit takes half a byte. The sign is stored separately as the rightmost half a byte character.

EG:

**S9(1) COMP-3 -> 1BYTE**

S9(2) COMP-3 -> 2BYTE

S9(3) COMP-3 -> 2BYTES

* You need to use HEX ON commmand to see COMP-3 values in PS

**COMP-3 Example :**

975  
86C

The number is +98765

* Beware: Moving character data into a COMP-3 field may give you the dreaded SOC7 data exception abend when the receiving item is referenced. This is because not all bit patterns are valid COMP-3 numbers.
* If last character is C or F, that denotes a positive sign.

If last character is D , that denotes a negative sign.

**01 WS-NUMBER PIC S9(03) COMP-3.**

**01 WS-NUMBER1 PIC S9(03) SIGN LEADING SEPARATE.**

**MOVE WS-NUMBWER TO WS-NUMBER1**

**DISPLAY WS-NUMBER1**

* Moving LOW-VALUES to COMP-3 is not possible. This will give compiler error.
* A file can be opened in the **INPUT** mode only if it already exists.

Such a file becomes an input file from which records can be read

sequentially.

* This **CLOSE** statement means **termination of link between Physical file and logical file.**
* File handling for seq files – FAQ interview question

Reading sequential file

OPEN EMPFILE

**READ EMPFILE**

**AT END imperative statement.**

**END-READ**

or

READ EMPFILE INTO WS-RECORD

AT END imperative statement.

END-READ

WRITE CAPGEMINI-REC FROM WS-RECORD

END-WRITE

or

**WRITE CAPGEMINI-REC**

**END-WRITE**

* Here we can notice huge difference b/w COBOL and Easytrieve.

**In COBOL:**

If we are reading file we refer file name in READ statement,

where as while writing we refer record name with WRITE command.

**In Easytrive:**

Both read and write have reference to dd name of the file

EG:

GET INFILE

PUT OUTFILE FROM INFILE

**Note:** here in easytrive writing data we must make sure to use working storage variable data gets populated into OUTFILE-REC.

* REWRITE command is used to update a record in a file. If file is

opened in **I-O mode then only we can use REWRITE command** on that

file. **This REWRITE is not desirable in Sequential file**. We can update a sequential file but it should contain same length. no changes in length of corresponding rec will not allow  
Before using REWRITE command, corresponding record should be read.

OPEN I-O FILE1.

READ FILE1

AT END MOVE 'Y' TO WS-EOF-SW.

IF WS-FS = 00

MOVE WS-STD-NAME TO STD-NAME

ELSE

DISPLAY ' RECORD NOT FOUND IN FILE...'

END-IF.

CLOSE FILE1.

STOP RUN.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

----------------------------------END--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Questions and their Explanation**

**Q. All about Arrays.**

**Ans**

Arrays in COBOL are known as tables. An array is a linear data structure and is a collection of individual data items of same type. Data items of a table are internally sorted.

## Table Declaration

Table is declared in Data Division. **Occurs** clause is used to define a table. Occurs clause indicates the repetition of data name definition. It can be used only with level numbers starting from 02 to 49. **Do not use occurs clause with Redefines.** Description of one-dimensional and two-dimensional table is as follows −

### **One-Dimensional Table**

In a one-dimensional table, **occurs** clause is used only once in declaration. WSTABLE is the group item that contains table. WS-B names the table elements that occur 10 times.

**Syntax**

Following is the syntax for defining a one-dimensional table −

01 WS-TABLE.

05 WS-A PIC A(10) OCCURS 10 TIMES.

**Example**

[Live Demo](http://tpcg.io/ZGeTQX)

IDENTIFICATION DIVISION.

PROGRAM-ID. HELLO.

DATA DIVISION.

WORKING-STORAGE SECTION.

01 WS-TABLE.

05 WS-A PIC A(10) VALUE 'TUTORIALS' OCCURS 5 TIMES.

PROCEDURE DIVISION.

DISPLAY "ONE-D TABLE : "WS-TABLE.

STOP RUN.

**JCL** to execute the above COBOL program −

//SAMPLE JOB(TESTJCL,XXXXXX),CLASS = A,MSGCLASS = C

//STEP1 EXEC PGM = HELLO

When you compile and execute the above program, it produces the following result −

ONE-D TABLE : TUTORIALS TUTORIALS TUTORIALS TUTORIALS TUTORIALS

### **Two-Dimensional Table**

A two-dimensional table is created with both data elements being variable length. For reference, go through the syntax and then try to analyze the table. The first array (WS-A) can occur from 1 to 10 times and the inner array (WS-C) can occur from 1 to 5 times. For each entry of WS-A, there will be corresponding 5 entries of WS-C.

**Syntax**

Following is the syntax for defining a two-dimensional table −

01 WS-TABLE.

05 WS-A OCCURS 10 TIMES.

10 WS-B PIC A(10).

10 WS-C OCCURS 5 TIMES.

15 WS-D PIC X(6).

**Example**

[Live Demo](http://tpcg.io/tZ8erh)

IDENTIFICATION DIVISION.

PROGRAM-ID. HELLO.

DATA DIVISION.

WORKING-STORAGE SECTION.

01 WS-TABLE.

05 WS-A OCCURS 2 TIMES.

10 WS-B PIC A(10) VALUE ' TUTORIALS'.

10 WS-C OCCURS 2 TIMES.

15 WS-D PIC X(6) VALUE ' POINT'.

PROCEDURE DIVISION.

DISPLAY "TWO-D TABLE : "WS-TABLE.

STOP RUN.

**JCL** to execute the above COBOL program −

//SAMPLE JOB(TESTJCL,XXXXXX),CLASS = A,MSGCLASS = C

//STEP1 EXEC PGM = HELLO

When you compile and execute the above program, it produces the following result −

TWO-D TABLE : TUTORIALS POINT POINT TUTORIALS POINT POINT

## Subscript

Table individual elements can be accessed by using subscript. Subscript values can range from 1 to the number of times the table occurs. A subscript can be any positive number. It does not require any declaration in data division. It is automatically created with occurs clause. **A subscript is occurrence of the array element.**

**Example**

[Live Demo](http://tpcg.io/Qt2kdw)

IDENTIFICATION DIVISION.

PROGRAM-ID. HELLO.

DATA DIVISION.

WORKING-STORAGE SECTION.

01 WS-TABLE.

05 WS-A OCCURS 3 TIMES.

10 WS-B PIC A(2).

10 WS-C OCCURS 2 TIMES.

15 WS-D PIC X(3).

PROCEDURE DIVISION.

MOVE '12ABCDEF34GHIJKL56MNOPQR' TO WS-TABLE.

DISPLAY 'WS-TABLE : ' WS-TABLE.

DISPLAY 'WS-A(1) : ' WS-A(1).

DISPLAY 'WS-C(1,1) : ' WS-C(1,1).

DISPLAY 'WS-C(1,2) : ' WS-C(1,2).

DISPLAY 'WS-A(2) : ' WS-A(2).

DISPLAY 'WS-C(2,1) : ' WS-C(2,1).

DISPLAY 'WS-C(2,2) : ' WS-C(2,2).

DISPLAY 'WS-A(3) : ' WS-A(3).

DISPLAY 'WS-C(3,1) : ' WS-C(3,1).

DISPLAY 'WS-C(3,2) : ' WS-C(3,2).

STOP RUN.

**JCL** to execute the above COBOL program −

//SAMPLE JOB(TESTJCL,XXXXXX),CLASS = A,MSGCLASS = C

//STEP1 EXEC PGM = HELLO

When you compile and execute the above program, it produces the following result −

WS-TABLE : 12ABCDEF34GHIJKL56MNOPQR

WS-A(1) : 12ABCDEF

WS-C(1,1) : ABC

WS-C(1,2) : DEF

WS-A(2) : 34GHIJKL

WS-C(2,1) : GHI

WS-C(2,2) : JKL

WS-A(3) : 56MNOPQR

WS-C(3,1) : MNO

WS-C(3,2) : PQR

## Index

**Table elements can also be accessed using index. An index is a displacement** of element from the start of the table. An index is declared with Occurs clause using INDEXED BY clause. The value of index can be changed using SET statement and PERFORM Varying option.

**Syntax**

Following is the syntax for defining Index in a table −

01 WS-TABLE.

05 WS-A PIC A(10) OCCURS 10 TIMES INDEXED BY I.

**Example**

[Live Demo](http://tpcg.io/3nFqgL)

IDENTIFICATION DIVISION.

PROGRAM-ID. HELLO.

DATA DIVISION.

WORKING-STORAGE SECTION.

01 WS-TABLE.

05 WS-A OCCURS 3 TIMES INDEXED BY I.

10 WS-B PIC A(2).

10 WS-C OCCURS 2 TIMES INDEXED BY J.

15 WS-D PIC X(3).

PROCEDURE DIVISION.

MOVE '12ABCDEF34GHIJKL56MNOPQR' TO WS-TABLE.

PERFORM A-PARA VARYING I FROM 1 BY 1 UNTIL I >3

STOP RUN.

A-PARA.

PERFORM C-PARA VARYING J FROM 1 BY 1 UNTIL J>2.

C-PARA.

DISPLAY WS-C(I,J).

**JCL** to execute the above COBOL program −

//SAMPLE JOB(TESTJCL,XXXXXX),CLASS = A,MSGCLASS = C

//STEP1 EXEC PGM = HELLO

When you compile and execute the above program, it produces the following result −

ABC

DEF

GHI

JKL

MNO

PQR

## Set Statement

Set statement is used to change the index value. Set verb is used to initialize, increment, or decrement the index value. It is used with Search and Search All to locate elements in table.

**Syntax**

Following is the syntax for using a Set statement −

SET I J TO positive-number

SET I TO J

SET I TO 5

SET I J UP BY 1

SET J DOWN BY 5

**Example**

[Live Demo](http://tpcg.io/SsSvSR)

IDENTIFICATION DIVISION.

PROGRAM-ID. HELLO.

DATA DIVISION.

WORKING-STORAGE SECTION.

01 WS-TABLE.

05 WS-A OCCURS 3 TIMES INDEXED BY I.

10 WS-B PIC A(2).

10 WS-C OCCURS 2 TIMES INDEXED BY J.

15 WS-D PIC X(3).

PROCEDURE DIVISION.

MOVE '12ABCDEF34GHIJKL56MNOPQR' TO WS-TABLE.

SET I J TO 1.

DISPLAY WS-C(I,J).

SET I J UP BY 1.

DISPLAY WS-C(I,J).

STOP RUN.

**JCL** to execute the above COBOL program.

//SAMPLE JOB(TESTJCL,XXXXXX),CLASS = A,MSGCLASS = C

//STEP1 EXEC PGM = HELLO

When you compile and execute the above program, it produces the following result −

ABC

JKL

## Search

**Search is a linear search method**, which is used to find elements inside the table. It can be performed on sorted as well as unsorted table. It is used only for tables declared by Index phrase. **It starts with the initial value of index. If the searched element is not found, then the index is automatically incremented by 1 and it continues till the end of table.**

**Example**

[Live Demo](http://tpcg.io/1nLjJH)

IDENTIFICATION DIVISION.

PROGRAM-ID. HELLO.

DATA DIVISION.

WORKING-STORAGE SECTION.

01 WS-TABLE.

05 WS-A PIC X(1) OCCURS 18 TIMES INDEXED BY I.

01 WS-SRCH PIC A(1) VALUE 'M'.

PROCEDURE DIVISION.

MOVE 'ABCDEFGHIJKLMNOPQR' TO WS-TABLE.

**SET I TO 1.**

**SEARCH WS-A**

**AT END DISPLAY 'M NOT FOUND IN TABLE'**

**WHEN WS-A(I) = WS-SRCH**

**DISPLAY 'LETTER M FOUND IN TABLE'**

**END-SEARCH.**

STOP RUN.

**JCL** to execute the above COBOL program.

//SAMPLE JOB(TESTJCL,XXXXXX),CLASS = A,MSGCLASS = C

//STEP1 EXEC PGM = HELLO

When you compile and execute the above program, it produces the following result −

LETTER M FOUND IN TABLE

## Search All

**Search All is a binary search method, which is used to find elements inside the table. Table must be in sorted order for Search All option. The index does not require initialization. In binary search, the table is divided into two halves and it determines in which half the searched element is present. This process repeats till the element is found or the end is reached**.

**Example**

[Live Demo](http://tpcg.io/6SY7Io)

IDENTIFICATION DIVISION.

PROGRAM-ID. HELLO.

DATA DIVISION.

WORKING-STORAGE SECTION.

01 WS-TABLE.

05 WS-RECORD OCCURS 10 TIMES ASCENDING KEY IS WS-NUM INDEXED BY I.

10 WS-NUM PIC 9(2).

10 WS-NAME PIC A(3).

PROCEDURE DIVISION.

MOVE '12ABC56DEF34GHI78JKL93MNO11PQR' TO WS-TABLE.

**SEARCH ALL WS-RECORD**

**AT END DISPLAY 'RECORD NOT FOUND'**

**WHEN WS-NUM(I) = 93**

**DISPLAY 'RECORD FOUND '**

**DISPLAY WS-NUM(I)**

**DISPLAY WS-NAME(I)**

END-SEARCH.

**JCL** to execute the above COBOL program −

//SAMPLE JOB(TESTJCL,XXXXXX),CLASS = A,MSGCLASS = C

//STEP1 EXEC PGM = HELLO

When you compile and execute the above program, it produces the following result −

RECORD FOUND

93

MNO

**Q.What is COBOL program?**

**ANS.**COBOL is a high-level programming language first developed by the CODASYL Committee (Conference on Data Systems Languages) in 1960. Since then, responsibility for developing new COBOL standards has been assumed by the American National Standards Institute (ANSI).

Three ANSI standards for COBOL have been produced: in 1968, 1974 and 1985. A new COBOL standard introducing object-oriented programming to COBOL, is due within the next few years.

The word COBOL is an acronym that stands for COmmon Business Oriented Language. As the the expanded acronym indicates, COBOL is designed for developing business, typically file-oriented, applications. It is not designed for writing systems programs. For instance, you would not develop an operating system or a compiler using COBOL.

COBOL is one of the oldest programming languages still in active use. Its name is an acronym for COmmon Business-Oriented Language, defining its **primary domain in business, finance, and administrative systems for companies and governments.**

**The COBOL 2002 standard includes support for object-oriented programming and other modern language features**.

**Q. Accessing bytes of data in cobol (Reference Modification)**

**ANS.** if a string is "NALINI" i have to access a character say 'L' in the string and store it in a character

Eg.

01 MY-VARIABLE PIC X(20).

MOVE 'NALINI' TO MY-VARIABLE

DISPLAY **MY-VARIABLE(3:1)**

Here 3 is the offset and 1 is length of characters to be extracted

**Q. Usage of INSPECT and TALLYING**

**ANS.**

INSPECT verb allows to count and/or replace a character or a group

of characters. INSPECT has options TALLYING, REPLACING & CONVERTING.

INSPECT Examples:

Source-string = “AABAbbACABA”

1. **INSPECT** source-string **TALLYING** tally-counter **FOR CHARACTERS**

**BEFORE INITIAL** ‘C’

In this example tally counter will have count of all characters

before first occurrence of ‘C’ in source-string.

Tally-counter will be ‘7’.

2. **INSPECT** source-string **TALLYING** tally-counter **FOR ALL** ‘A’

In this example tally counter will have count of all occurrences

of ‘A’ in source-string.

Tally-counter will be ‘6’.

3. **INSPECT** source-string **TALLYING** tally-counter **FOR ALL** ‘A’

**AFTER INITIAL** ‘B’

In this example tally counter will have no of occurrences of ‘A’

after first occurrence of ‘B’.

Tally-counter will be ‘4’.

4. **INSPECT** source-string **TALLYING** tally-counter **FOR LEADING** ‘A’

In this example tally counter will have no of leading A’s.

Tally-counter will be ‘2’.

5. Source-string = " SSET"

If we need to get string without spaces, trim the string. we can use

following logic.

**INSPECTFUNCTION REVERSE (**Source-string**) TALLYING** space-count **FORLEADING** SPACES.

COMPUTE length-of-string = 6 - space-count.

Move Source-string(space-count+1 : length-of-string ) TO ws-target-string.

Above INSPECT command get the no of leading spaces

from the string. after executing the INSPECT command

space-count variable contains 2.

In compute statement, space-count subtracted from

length of Source-string. value 4 will be stored

in length-of-string.

In move statement, using reference modification, moved

actual string to ws-target-string. removed spaces.

**INSPECT… with REPLACING** option

This form replaces a character or group of characters

Example:

Source-string = “AABAbbACABA”

1. **INSPECT** source-string **REPLACING CHARACTERS BY** ‘#’ **BEFORE INITIAL** ‘C’.

In above example all characters before first occurrence of ‘C’

in source-string are replaced by ‘#’.

Input : “AABAbbA<b>C</b>ABA”

Output: -“#######CABA”.

2.**INSPECT** source-string **REPLACING ALL** ‘A’ **BY** ‘#’

In above example all occurrences of ‘A’ in source-string are

replaced by ‘#’.

Input :“AABAbbACABA”

Output: - “##B#bb#C#B#”.

TIPS :

With CHARACTERS phrase only one-character size of replacing

string should be specified. And also delimiter (‘C’) must be

of one character.

**Replacing string and replaced string must be of same size if**

**we use phrases ALL, LEADING and FIRST.**

->INSPECT… with TALLYING and REPLACING options

This form counts/tally a character or group of characters and

replaces a character or group of characters.

Example:

Source-string = “AABAbbACAB”

1. **INSPECT** source-string **TALLYING** tally-counter **FORALL**‘A’ **AFTERINITIAL** ‘B’

**REPLACINGALL** ‘A’ **BY** ‘#’ **AFTERINITIAL** ‘B’

In above example all occurrences of ‘A’ after first occurrence

of ‘B’ in source-string are counted and replaced by ‘#’.

Tally-counter will be ‘3’

Source-string will be “AAB#bb#C#B”.

**INSPECT… with CONVERTING option**

This form converts characters from one sequence string to characters

from equal size other sequence string on one to one mapping basis.

These two sequence strings must be of same size.

Example:

Source-string = “AABAbbACAB”

1. **INSPECT** source-string **CONVERTING**‘BXCY’ **TO** ‘1234’

In above example converts the characters B, X, C, Y to 1, 2, 3, and 4

respectively.

Input : “AA<b>B</b>AbbA<b>C</b>A<b>B</b>”

Output: “AA1AbbA3A1”

Equivalent INSPECT with REPLACING option

**INSPECT** source-string **REPLACINGALL** ‘B’ **BY** ‘1’

**ALL** ‘X’ **BY** ‘2’

**ALL** ‘C’ **BY** ‘3’

**ALL** ‘Y’ **BY** ‘4’

**Q.Difference between CONVERTING and REPLACING options in INSPECT**

**ANS**

Source-string = “AABAbbACAB”

INSPECT source-string CONVERTING ‘AB’ TO ‘12’

Output: - “1121bb1C12”

INSPECT source-string REPLACING ‘AB’ TO ‘12’

Output:- “A12AbbAC12”

In example 1, all occurrences of characters A and B are replaced by characters

1 and 2 respectively.

In example2, all occurrences of string ‘AB’ is replaced by string ‘12’.

**Q. Using Parm parameter to receive data into cobol?**

**ANS.**

**Keyword parameters for EXEC**

1. PARM
2. ACCT
3. ADDRSPC
4. DPRTY
5. PERFORM
6. RD

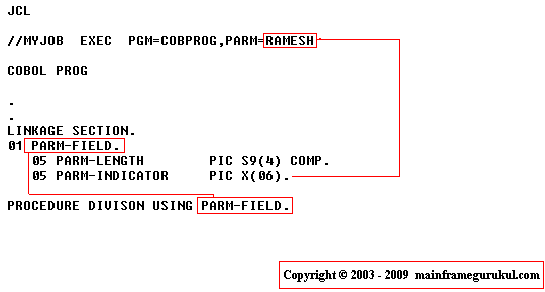
**PARM**

PARAM parameter is used to pass information to program

Syntax -> PARM=value

Value is a string can be 1 to 100 characters long.

PASSING PARAMETER USING PARM PARAMETER



PARM-INDICATOR will contain "RAMESH"

PARM-LENGTH contains length of string.

Remaining parameters , We wont use much

**ACCT** - accounting information for that step

**ADDRSPC** - used to indicate to the system that the job step is use either

virtual or real storage

**DPRTY** - used to assign priority to the job step

**PERFORM** - specifies the rate at which system resources used by job step

**RD** - restart definition is used to specify automatic restart of a job if it abends

**Q. Reading a PS file and saving its data in working storage variable**

**ANS**.

All file processing operations are held in procedure division.

File Operations:

1. OPEN
2. READ
3. WRITE
4. REWRITE
5. CLOSE

**OPEN EMPFILE**

**OPEN {INPUT, OUTPUT, EXTEND, I-O} file-name-1 [, file-name-2] . . .**

The OPEN statement initiates the processing of files. **The success-**

**full execution of an OPEN statement determines the availability of**

**the file for processing.** The successful execution of the OPEN

statement makes the associated record area available to the program;

it does not obtain or release the first data record. If the FILE

STATUS clause is specified in the FILE-CONTROL entry, the associated

operation status is updated when the OPEN statement is executed.

A sequential file can be opened in one of the following four modes.

INPUT, OUTPUT, EXTEND and I-O.

A file can be opened in the **INPUT** mode only if it already exists.

Such a file becomes an input file from which records can be read

sequentially.

When a file is to be created for the first time, it must be opened

in the **OUTPUT** mode. File can be written in this mode.

The **EXTEND** mode also opens a file for writing, but the file pointer

is positioned after the end of the last record. Thus any records

written will get appended to the file.

A file is opened in the **I-O** mode when it needs to be updated. This

mode provides both reading and rewriting of records.

**CLOSE EMPFILE**

This statement terminates processing of file. In COBOL-85 this

CLOSE Statement is optional and STOP RUN automatically closes the

File if it is not explicitly closed. **This CLOSE statement means**

**termination of link between Physical file and logical file.**

**READ EMPFILE**

If a file is opened in INPUT or I-O mode, then a READ statement

make available Next logical record for processing. The primary

function of the READ statement is to fetch records from a file

and place the file pointer at an appropriate position after READ;

it performs certain checks to ensure proper execution of the

program.

**READ EMPFILE**

**AT END imperative statement.**

**END-READ**

If the file reached end of the file and if program tries to read

a record then AT END condition satisfy and imperative statement

will performed.

If **INTO** condition specified like **READ EMPFILE INTO WS-RECORD**.

Here WS-RECORD is data division data name. Program read the file

and places record into WS-RECORD.

**WRITE statement**

**WRITE CAPGEMINI-REC FROM WS-RECORD**

**END-WRITE.**

File opened in OUTPUT or EXTEND mode then we can use WRITE statement

to write records into the file. If from is used, then the data

in WS-RECORD is moved to record and writing take place. If FROM

is omitted then Data moved to record name will be written into

file. If we are reading file we refer file name in READ statement,

where as while writing we refer record name with WRITE command.

**REWRITE statement**

REWRITE command is used to update a record in a file. If file is

opened in I-O mode then only we can use REWRITE command on that

file. **This REWRITE is not desirable in Sequential file**. We can update a sequential file but it should contain same length. no changes in length of corresponding rec will not allow  
Before using REWRITE command, corresponding record should be read.

OPEN I-O FILE1.

READ FILE1

AT END MOVE 'Y' TO WS-EOF-SW.

IF WS-FS = 00

MOVE WS-STD-NAME TO STD-NAME

ELSE

DISPLAY ' RECORD NOT FOUND IN FILE...'

END-IF.

CLOSE FILE1.

STOP RUN.

**A Sample COBOL Program that READ a Sequential files and Write**

**into another file.**

1. Write data into PS file (OUTPUT mode)

**IDENTIFICATION DIVISION.**

PROGRAM-ID.TESTCOBL.

AUTHOR.TESTTEST.

DATE-WRITTEN. 19-NOV-2010.

DATE-COMPILED.

**ENVIRONMENT DIVISION.**

**INPUT-OUTPUT SECTION.**

**FILE-CONTROL.**

SELECT WS-INFILE ASSIGN TO INPIWS01

FILE STATUS IS WS-INFILE-SW.

**DATA DIVISION.**

**FILE SECTION.**

FD WS-INFILE.

01 INP-EMP-REC.

05 INP-EMPID PIC X(08).

05 INP-EMPID PIC X(08).

05 INP-EMP-NAME PIC X(15).

05 INP-EMP-LOB PIC X(10).

05 FILLER PIC X(47).

**WORKING-STORAGE SECTION.**

01 WS-INFILE-SW PIC X(02) VALUE SPACES.

88 WS-INFILE-SUCESS VALUE '00'.

88 WS-INFILE-EOF VALUE '10'.

01 WS-INP-EMP-REC.

05 WS-INP-EMPID PIC X(08).

05 WS-INP-EMP-NAME PIC X(15).

05 WS-INP-EMP-LOB PIC X(10).

05 FILLER PIC X(47).

01 **WS-EOF-SW** PIC X(01) VALUE 'N'.

88 WS-EOF-NO VALUE 'N'.

88 WS-EOF-YES VALUE 'Y'.

**PROCEDURE DIVISION.**

A1000-MAIN-PARA.

PERFORM A2000-OPEN-PARA THRU A200-EXIT.

PERFORM A3000-INPUT-PARA THRU A300-EXIT.

PERFORM A4000-INSERT-PARA THRU A400-EXIT.

PERFORM A5000-CLOSE-PARA THRU A500-EXIT.

STOP RUN.

A100-EXIT.

EXIT.

A2000-OPEN-PARA.

INITIALIZE WS-INFILE-SW WS-INP-EMP-REC WS-EOF-SW.

**OPEN OUTPUT WS-INFILE**

IF WS-INFILE-SUCESS

DISPLAY "FILE OPEN SUCCESSFUL"

ELSE

DISPLAY "FILE OPENING ERROR"

GO TO A100-EXIT

END-IF.

A200-EXIT.

EXIT.

A3000-INPUT-PARA.

ACCEPT WS-INP-EMPID.

ACCEPT WS-INP-EMP-NAME.

ACCEPT WS-INP-EMP-LOB.

DISPLAY WS-INP-EMPID WS-INP-EMP-NAME WS-INP-EMP-LOB

.

A300-EXIT.

EXIT.

A4000-INSERT-PARA.

**WRITE INP-EMP-REC FROM WS-INP-EMP-REC.**

A400-EXIT.

EXIT.

A5000-CLOSE-PARA.

CLOSE WS-INFILE.

A500-EXIT.

EXIT.

**2. Read data into PS file (OUTPUT mode)**

IDENTIFICATION DIVISION.

PROGRAM-ID. READ-EXAMPLE.

AUTHOR Sandeep.

ENVIRONMENT DIVISION.

INPUT-OUTPUT SECTION.

**FILE-CONTROL.**

ASSIGN IN-FILE TO INPIWS01

ASSIGN PRINT-FILE TO PRINTFIL

DATA DIVISION.

FILE SECTION.

**FD IN-FILE.**

**01 CUSTOMER-DETAILS.**

**03 CUS-NAME PIC X(20).**

**03 CUS-NUM PIC 9(6).**

**FD PRINT-FILE.**

**01 PRINT-REC PIC X(60).**

WORKING-STORAGE SECTION.

**01 EOF-FLAG PIC X.**

**88 END-OF-IN-FILE VALUE 'Y'.**

01 P-CUS-DETAILS

03 PIC X(5) VALUE SPACES.

03 P-NAME PIC X(25).

03 P-NUM PIC Z(5)9.

:

**PROCEDURE DIVISION.**

MAIN-PARAGRAPH.

**OPEN INPUT IN-FILE**

\*"Prime" read

**READ IN-FILE**

**AT END MOVE 'Y' TO EOF-FLAG**

**NOT AT END PERFORM PRINT-DETAILS**

**END-READ**

\*Main reading loop

PERFORM UNTIL END-OF-IN-FILE

READ IN-FILE

AT END MOVE 'Y' TO EOF-FLAG

NOT AT END PERFORM PRINT-DETAILS

END-READ

END-PERFORM

STOP RUN.

**PRINT-DETAILS.**

MOVE CUS-NAME TO P-NAME

MOVE CUS-NUM TO P-NUM

**WRITE PRINT-REC FROM P-CUS-DETAILS AFTER 1 LINE.**

**Q. Explain COMP COMP-1 COMP-2 COMP-3?**

**ANS.**

COMP: Normally, a computer can store data in more than one internal form.

In COBOL, a programmer is allowed to specify the internal form of the data item so as to facilitate its use in the most efficient manner. There are only **two general forms** of internal representation namely **COMPUTATIONAL and DISPLAY**.

Only numeric data items can be specified as USAGE IS COMPUTATIONAL and the name itself suggests a data item specified as USAGE IS COMPUTATIONAL can take part in

arithmetic operations more efficiently and any data item can be specified as USAGE IS DISPLAY. If we omit Usage clause compiler assume DISPLAY as default.

In COMP usage Data is stored as Pure Binary format internally. Depending on the size of the data item, it can be stored either in a half-word (2 bytes with range -32,768 to +32767) or

Full-word (4 bytes with range -2,147,483,648 to 2,147,483,647). The PICTURE Clause of a COMPUTATIONAL data item should not contain any character other than 9 or S.

Example: 01 WS--DS-HDR-LEN PIC S9(02) COMP.

Which occupies 2 bytes and data stored as Pure binary format internally.

**Data name length Length in COBOL**

9(01) to 9(04) 2 bytes

9(05) to 9(09) 4 bytes

S9(10) to S9(18) 8 bytes

**COMP-1**: In this case the data item will be represented in one word in the floating point form the number is actually represented in hexadecimal format and is suitable for arithmetic operations.

The PICTURE Clause cannot be specified for COMP-1 items.

COMP-1 takes 4 bytes of storage.

**COMP-2:** This is same as COMP-1, except that the data is represented internally in two words. The advantage is that this increases the precision of the data, which means that more significant

digits are available. Similar to COMP-1, The PICTURE Clause cannot be specified for COMP-2 items also. COMP-2 takes 8 bytes of storage. COMP-2 is more precision than COMP-1.

COMP-1, COMP-2 doesn’t have PIC clause because PIC creates relation between data name and data type. Whereas here data is numeric and length is predefined as one-word floating for

COMP-1 and double word floating for COMP-2. So no relation is required using PIC clause.

**COMP-3**: In this case the data is represented in the decimal form, but one digit takes half a byte. The sign is stored separately as the rightmost half a byte character.

Find Comp-3 storage in bytes: Length of variable/2, if this is not pure number then it takes immediate next number of bytes.

**For example, S9(06): which takes 4 bytes.**

for sign (S) and variable length is 6 which takes 3 bytes. total is 3 so its immediate number is 4. Additional added byte is called slack byte.

COMP-3 Picture clause Number of bytes occupied by the field

S9(1) COMP-3 1

S9(2) COMP-3 2

S9(3) COMP-3 2

S9(4) COMP-3 3

S9(5) COMP-3 3

S9(6) COMP-3 4

S9(7) COMP-3 4

S9(8) COMP-3 5

S9(9) COMP-3 5

S9(10) COMP-3 6

S9(11) COMP-3 6

S9(12) COMP-3 7

S9(13) COMP-3 7

S9(14) COMP-3 8

S9(15) COMP-3 8

S9(16) COMP-3 9

S9(17) COMP-3 9

S9(18) COMP-3 10

1. The USAGE Clause can be specified on data items defined with any level number.

2. The USAGE of a group item is valid for all its sub items.

You need to use HEX ON command to see COMP-3 values in PS

When you use this command it will show two lines per each record.

**You need to read the comp-3 value from left, top to bottom (in two**

**lines) and then move to second character read top to bottom.**

COMP-3 Example :

975  
86C

The number is +98765

last characters represent sign.

If last character is C or F, that denotes a positive sign.

If last character is D, that denotes a negative sign.

Question: Conversion of comp-3 to character

Displaying COMP-3 characters: Before displaying COMP-3 Move the data

to usage display variable then display. If it holds Sign, then Use SIGN

LEADING SEPARATE which uses one extra byte for sign and sign will be

displayed separately.

**01 WS-NUMBER PIC S9(03) COMP-3.**

**01 WS-NUMBER1 PIC S9(03) SIGN LEADING SEPARATE.**

**MOVE WS-NUMBWER TO WS-NUMBER1DISPLAY WS-NUMBER1**

displays data properly.

Moving LOW-VALUES to COMP-3 is not possible. This will give compiler error.

**Q.** **Variable Blocked Files.**

**ANS.**

In non mainframe architectures, All files are essentially variable record length files, in that each record may not have the same record length as the previous record and the end of the record is determined by special characters such as carriage return and line feed. In mainframe environment record are not delimited in this way. A file's structure will be pre-determined when the file is defined as either fixed record length or variable record length. Variable record length records have 4 byte field at the start of the record to say how long the record actually is.**The reason we cater for variable record length file is because most files contain different record types and each record type will have a different record structure. Each record structure will have its own length. Large amounts of space are saved in this way.** In the "old days" when disk space was very expensive, many shops would save further space on a file by creating some form of compression technique to compress the data down. One such example i came across had a field at the beginning of the record, and this field was examined bit for bit to determine whether particular field's where on the record or not. A program was written so that when the file was read it would read the record and expand the record according to the compression algorithm, and when a record was written it would do the reverse.

**IDENTIFICATION DIVISION.**

PROGRAM-ID. TEST1.

ENVIRONMENT DIVISION.

INPUT-OUTPUT SECTION.

**FILE-CONTROL.**

SELECT INPUTFL ASSIGN TO UT-S-INPUTFL.

DATA DIVISION.

FILE SECTION.

FD INPUTFL

BLOCK CONTAINS 0 RECORDS

RECORDING MODE IS V

RECORD IS VARYING IN SIZE FROM 1 TO 80 CHARACTERS

DEPENDING ON FILE-SIZE.

01 INPUTFL-REC PIC X(80).

WORKING-STORAGE SECTION.

01 FILE-SIZE PIC 9(4) COMP-5.

01 SWITCHES.

05 EOF-INPUT-FILE PIC X VALUE 'N'.

88 EOF-INPUT VALUE 'Y'.

**PROCEDURE DIVISION.**

**OPEN INPUT INPUTFL.**

MOVE 'N' TO EOF-INPUT-FILE.

PERFORM UNTIL EOF-INPUT

**READ INPUTFL**

**AT END SET EOF-INPUT TO TRUE**

**END-READ**

IF NOT EOF-INPUT

**DISPLAY FILE-SIZE**

END-IF

END-PERFORM

STOP RUN.

Contents of Input file INPUTFL

------------------------------

THE QUICK BROWN

FOX JUMPS

OVER THE LAZY

DOG

Contents of sysout display

--------------------------

00015

00009

00013

00003

**Q. Date Handling in Cobol like adding one day to current date.**

**ANS.**

**CURRENT-DATE** is COBOL intrinsic function to get current date, time and difference

between current location time and Greenwich Mean Time.

**MOVE FUNCTION CURRENT-DATE TO** WS-CURRENT-DATE-DATA

This function returns a 20-character alphanumeric field in the below format

01 WS-CURRENT-DATE-DATA.

05 WS-CURRENT-DATE.

10 WS-CURRENT-YEAR PIC 9(04).

10 WS-CURRENT-MONTH PIC 9(02).

10 WS-CURRENT-DAY PIC 9(02).

05 WS-CURRENT-TIME.

10 WS-CURRENT-HOURS PIC 9(02).

10 WS-CURRENT-MINUTE PIC 9(02).

10 WS-CURRENT-SECOND PIC 9(02).

10 WS-CURRENT-MILLISECONDS PIC 9(02).

05 WS-DIFF-FROM-GMT PIC S9(04).

**WS-CURRENT-DATE-DATA contains : 2010111917542857+0800**

a) Converting from Gregorian dates to integer date.

COMPUTE integer-date = **FUNCTION INTEGER-OF-DATE** (Gregorian -date).

**Gregorian-date must be in form YYYYMMDD**.

The function result is a 7-digit integer

Note: 1600 < YYYY < 9999; 0 < MM < 13; 0 < DD < 32

(provided that day is valid for the specified month and year combination).

b) Convert from Integer to Gregorian formats.

COMPUTE Gregorian-date = FUNCTION DATE-OF-INTEGER (integer-date)

The function result Gregorian-date is an eight-digit integer in the form

YYYYMMDD.

c) Convert from Julian to Integer formats

COMPUTE integer-date = FUNCTION INTEGER-OF-DAY (Julian-date)

Julian-date must be in the form YYYYDDD,

The function result is a 7-digit integer.

Note: 1600 < YYYY < 9999 and 0 < DDD < 367

(provided the day is valid for the specified year)

**d) Convert from Integer to Julian formats**

**COMPUTE Julian-date = FUNCTION DAY-OF-INTEGER (integer-date)**

The Julian-date is a seven-digit integer in the form YYYYDDD.

Integer-date represents a number of days after December 31, 1600, in

the Gregorian calendar.

All these functions deal with converting between Gregorian dates or

Julian dates and integer format date. This integer format date is

number of days from fixed date

Example : Converting Gregorian format date 20101202 to Julian format date.

This converts 20101202(Gregorian date) to 0149720(integer format date).

COMPUTE Julian-date = FUNCTION DAY-OF-INTEGER (integer-date).

This converts 0149720(integer format date) to 2010336(Julian format date YYYDD)

using above date functions, one can get add or subtract days from current date.

It is easy with these functions.

Example-2: Calculate date after 10 days from Today.

MOVE FUNCTION CURRENT-DATE TO WS-CURRENT-DATE-DATA.

This move statement moves current timestamp (20-bytes) to WS-CURRENT-DATE-DATA.

Since we need only current date, we will take 8 bytes from WS-CURRENT-DATE-DATA.

COMPUTE ws-integer = FUNCTION INEGER-OF-DATE (WS-CURRENT-DATE-DATA(1:8))

This converts 20101202 to integer form 0149720 ( i.e. no of days from fixed date)

ADD 10 TO ws-integer

This calculates the integer form date of future date.

it will add 10 days to ws-integer.

COMPUTE ws-future-date = FUNCTION DATE-OF-INTEGER (ws-integer).

Convert integer form date (i.e. 149730) to date format i.e. 20101222

Q. Cobol Verbs.

ANS.

a) Start -

b) Seek

c) Rerun

d) Delete

Data manipulation/arithmetic verbs

INITIALIZE- INITIALIZE verb initializes values in a data item to default value.

Numeric data items initialized with ZEROS. Alphabetic character codes

initilized with spaces. It wont initialize FILLER.

Example - Initializing elementary data item.

INITIALIZE WS-A.

WS-A WS-AWS-A

PICTURE BEFORE AFTER

9(4) 6847 0000

X(3) 12E <- SPACES

XX99 WE48 <- SPACES

A(4) WELS <- SPACES

Example - Initializing group data item.

01 CUSTOMER-RECORD.

05 Customer-Name.

10 FirstName PIC X(06).

10 MiddleName PIC X(1).

10 LastName PIC X(05).

05 Customer-DOB.

10 Month PIC 99.

10 FILLER PIC X VALUE '/'.

10 Day PIC 99.

10 FILLER PIC X VALUE '/'.

10 Year PIC 99.

After executing following statement, it will initialize all values

to its default values.

INITIALIZE CUSTOMER-RECORD.

FirstName, MiddleName&LastName are initialized with SPACES. Month

, Day & Year are initialized with zeros. Since FILLER wont get initialized

by INITIALIZE verb, we have used VALUE clause which will initialize FILLER

When program execution started.

TIP : Using VALUE clause, we can initialize data items. But it works only once

when program execution started, it will initialize the data item with

whatever value specified with VALUE clause.

Example - 01 WS-Month PIC 99 VALUE 11.

When program execution started WS-Month data item contains value 11.

MOVE

MOVE literal-1/sending-data-item-1 TO receiving-data-item-1.

MOVE is a Cobol verb, when there is a need for transfering value from

onedataitem to another data item, using MOVE verb we can achieve that

task.

In above syntax, value from sending-field-1 will be moved to

receiving-field-1. Sending-filed-1 can a data item OR a literal (string).

Example for move a literal to a numeric data item.

01 WS-A PIC 999 VALUE 000.

....

MOVE 345 TO WS-A.

Before execution of above statement WS-A contains 000 as a value. After

execution of above statement WS-A contains 345.

Example - Move a alphanumeric literal to alpha numeric data item.

01 WS-NAME PIC X(4) VALUE 'JOHN'.

...

MOVE 'JACOB' TO WS-NAME.

Before execution WS-NAME contains value JOHN

Afer execution of MOVE statement, WS-NAME contains JACO.

If you observe only 4 letters from the string JACOB got transfered to WS-NAME

because WS-NAME defined with length of 4 bytes.In alphabetic OR alphanumeric

moves, Receiving fields receive the letters from left to right as shown below.

That is the reason WS-NAME contains JACO

If the receiving field is a numeric data item, data transfer happens from

right to left.

01 WS-A PIC 999 VALUE 890.

01 WS-B PIC 99 VALUE 22.

...

MOVE WS-A TO WS-B.

If numeric data item has any decimal part in it.numeric value after the decimal

will be moved to receiving field from left right. i.e, 1st digist after decimal

movement happens first, then 2nd byte, etc...

ADD

Syntax 1 - ADD { literal-1 / data-item-1 } .. TO target-data-item-1 ..

Syntax 2 - ADD { literal-1 / data-item-1 } .. GIVING target-data-item-1 ..

ADD verb adds one or more numeric values and store the value into a target data item.

Let us see some examples.

ADD 10 TO WS-A. Before execution WS-A value is 5,

afterexcution of above statement WS-A value will be 15.

ADD WS-A WS-B GIVING WS-C.

Before execution - WS-A value 100

WS-B value 150 WS-C value 786

After execution of above ADD statement, value in data items are shown below.

WS-A value 100 WS-B value 150 WS-C value 250.

ADD WS-A WS-B GIVING WS-C WS-D.

nowws-c=250 and ws-c=250

SUBTRACT

Syntax 1 -

SUBTRACT { literal-1 / data-item-1 } .. FROM data-item-2 ..

Syntax 2 -

SUBTRACT { literal-1 / data-item-1 } .. FROM { literal-2/data-item-2 }

GIVING data-item-3 ..

Rules - 1. All the values before the word FROM, summed up and subtracted from

each data-item/literl after word FROM

2. If GIVING is used, the result will be stored in the data-items

provided after the word GIVING.

3. If GIVING is not used, data-items provided after FROM will be used

to store the results.

Let us see some examples.

Let us assume WS-A contains value 200

WS-B contains value 100

SUBTRACT 10 FROM WS-A.

now WS-A = 190

SUBTRACT 10 FROM WS-A GIVING WS-B.

now WS-B = 190

SUBTRACT 10 11 12 FROM WS-A GIVING WS-B.

now WS-B = 167

SUBTRACT 10 20 30 FROM WS-A WS-B.

now WS-A = 140

WS-B=40

MULTIPLY

MULTIPLY statement multiplies numeric items.

Format 1.

MULTIPLY < literal-1 / data-item-1 > BY data-item-2 ...

Format 2.

MULTIPLY < literal-1 / data-item-1 > BY < literal-2 / data-item-2 >

GIVING data-item-3 ...

Examples -

(a) MULTIPLY WS-A BY WS-B.

In above example, values in WS-A & WS-B will be multiplied and the result will be stored in WS-B.

(b) MULTIPLY WS-A BY WS-B GIVING WS-C.

In the example (b), values in WS-A & WS-B will be multiplied and the result will be stored in WS-C.

(C) MULTIPLY WS-A BY WS-B GIVING WS-C WS-D.

In the example (c), values in WS-A & WS-B will be multiplied and the result will be stored in WS-C , WS-D.

DIVIDE

DIVIDE statement divide one number by another and store the result.

Format 1.

DIVIDE < literal-1 / data-item-1 > INTO data-item-1...

Format 2.

DIVIDE < literal-1 / data-item-1 > INTO < literal-2 / data-item-2 >

GIVING data-item-3... REMAINDER data-item-4...

Examples -

(a) DIVIDE WS-A INTO WS-B.

Above example is equals to WS-B = WS-B / WS-A. WS-B value is divided by WS-A and the result will be stored in WS-B.

(b) DIVIDE WS-A INTO WS-B GIVING WS-C.

Above example is equals to WS-C = WS-B / WS-A. WS-B value is divided by WS-A and the result will be stored in WS-C.

No change to WS-B, WS-A values.

(c) DIVIDE WS-A INTO WS-B GIVING WS-C REMAINDER WS-D.

This statement very similar to example (b), except using REMAINDER word.after dividing WS-B by WS-A remainder value

will be stored in WS-D.

WS-A value is 100

WS-B value is 350

100 ) 350 ( 3 <--- This value will be moved to WS-C.

300

----

50 <--- Remainder, this value will be moved to WS-D.

----

<u>Using BY instead of INTO....</u>

we can use INTO instead of BY in the above format2. But, make sure that values/data-items used for calcualtion

needs to be exchanged to get the same result.

example (b) can be rewrite using word BY , to get same result in WS-C, need to swap WS-B, WS-A places.

i.e, state should look like DIVIDE WS-B BY WS-A GIVING WS-C. equals WS-C = WS-B / WS-A.

COMPUTE

All the arthemetic, that we can done using verb ADD, SUBTRACT, MULTIPLY,

DIVIDE verbs can be done using COMPUTE statement. In COMPUTE statement,

we need to use following operators to do arthemetic.

Operators +,-,\*,/,\*\*

\*\* = exponentiation

Format.

COMPUTE < data-item-1> [ROUNDED] ... = arithmetic-expression

[ ON SIZE ERROR < imperative statement-1 ] ..

[ NOT ON SIZE ERROR < imperative-statement-2> ] ..

Example 1 - COMPUTE WS-A = WS-B + WS-C - WS-D.

Values in WS-B , WS-C will be added , WS-D value subtracted

from that value and store the final result in WS-A.

data-items values values

beforeexectuion after execution

WS-A 300 850 <--- changed

WS-B 800 800

WS-C 100 100

WS-D 050 050

It is very important to know the order of evaluation, i.e.., In which order

operations are performed in COMPUT statement. Below table , explains us,

the order of evalution.

Priority 1 \*\*

Priority 2 \* or / ( If both appeared , first appeared operator

executed first, from left to right )

Priority 3 + or - ( If both appeared , first appeared operator

executed first, from left to right )

If parentheses appear in the COMPUTE statement, it will override above sequence

priority. Operations within parentheses are performed first.

Statement Order of evaluation

WS-A + 10 \*\* 2 Exponentiation ( 10 \*\* 2) executed first and the

result 100 will be added to WS-A

(WS-A + 10 ) \*\* 2 Addition executed first and exponentiation follows.

ROUNDED This option is available with all arithmetic VERBs and it is optional.

COMPUTE WS-A = 23.456 + 20.034

When we sum values 23.456 and 20.034 we get the result 40.490. But in this case

WS-A picture clause is 99v9 means it can store only 1 decimal position, after

executing above statement WS-A contains the value of 40.4 value 90 will get trun-

cated. More desirable value is 40.5 in this case. rounded to nearest value.

This can be acheived thru the use of ROUNDED option.

COMPUTE WS-A ROUNDED = 23.456 + 20.034

after execution of above statement WS-A contains value of 40.5

ON SIZE ERROR Let take some example to understand this option.

01 WS-A PIC 9(3) VALUE 400.

01 WS-B PIC 9(3) VALUE 800.

01 WS-C PIC 9(3) VALUE 300.

COMPUTE WS-A = WS-B + WS-C.

Above statement, add the value in WS-B and WS-C and store that value in WS-A.

After execution of above statement WS-A contains the value of 100, instead of

1100. Because WS-A can hold only upto 3 bytes.

Please note that, program wont get abended because this overflow/truncation

condition. But value moved to WS-A is not correct one. To aviod these kind

of size errors, best procedure is make sure receiving field has defined with

large enough size to accommodate the result. But sometimes, programmer may

not know the input max number/forget to define the receiving fields with

enough sizes. It is a good practice to use ON SIZE ERROR to catch such

errors.

ON SIZE ERROR option can be used with following arithmetic statements.

- ADD ... ON SIZE ERROR ... <statements>.

- SUBTRACT ... ON SIZE ERROR ... <statements>.

- MULTIPLY ... ON SIZE ERROR ... <statements>.

- DIVIDE ... ON SIZE ERROR ... <statements>.

- COMPUTE ... ON SIZE ERROR ... <statements>.

COMPUTE WS-A = WS-B + WS-C

ON SIZE ERROR MOVE ZEROES TO WS-A.

In above example, If WS-A cannot accomodate the result then ZEROES will be

moved to WS-A.

A size error can occurs in the following ways.

- When receiving field is not large enough to accommodate the result.

- When division by zero occurs

If ON SIZE ERROR option becomes true, statements after this option will get executed.

These statement needs to be ended either by period OR scope terminator i.e, in case of

ADD, scope terminator is END-ADD.

TIP : Always make sure receiving field has large enough to accommodate the result.

I-O verbs

ACCEPT

DISPLAY

String handling verbs

STRING

UNSTRING

INSPECT

File handling verbs

OPEN

CLOSE

READ

WRITE

REWRITE

START

DELETE

Program branching Verbs

CALL

EXIT

EXIT PROGRAM

GO TO

PERFORM

STOP

STOP RUN

**Q. Cobol Arrays/Tables**

**ANS.**

**Q. Cobol Sort**

**ANS.**

**External Sort:**External sort refers to direct sort i.esort utility in JCL

**Internal Sort:**Internal refers to indirect use of sort utility thru programming language like Cobol.Cobol as such doesn’t have any internal sort algorithm.

**Sequencing Records with More Than One SORT Key:**

The SORT verb may be used to sequence records with more than one key ﬁeld.

The ﬁrst KEY ﬁeld indicated is the major ﬁeld to be sorted, the next KEY ﬁelds represent intermediate sort ﬁelds, followed by minor sort ﬁelds.

The following is a SORT statement that sorts records into ascending alphabetic NAME sequence within LEVEL-NO within OFFICE-NO:

SORT SORT-FILE  
*ON ASCENDING KEY OFFICE-NO  
ON ASCENDING KEY LEVEL-NO  
ON ASCENDING KEY NAME*  
USING PAYROLL-FILE-IN  
GIVING SORTED-PAYROLL-FILE-OUT

**Different sequences:** Because all key fields are independent, some key fields can be sorted in ASCENDING sequence and others in DESCENDING sequence.

**Combining the ON keyword:**Note too that the words ON and KEY were not underlined in the instruction format, which means that they are optional words. If all key fields are to be sorted in ascending sequence, as in the preceding, we can condense the coding by using the phrase ON ASCENDING KEY only once. Note that this technique we can combine only when all the sort keys follow same sequence.

SORT SORT-FILE

ON ASCENDING KEY MAJOR-KEY  
                                  INTERMEDIATE-KEY  
                                  MINOR-KEY

**WITH DUPLICATES IN ORDER:** With the most current version of COBOL, you can request the computer to put records with same value for the sort field into the sort file in the same order that they appeared in the original input file. We add the**WITH DUPLICATES IN ORDER** clause to accomplish this.

**Coding Simple Sort Procedure using the USING and GIVING clause:**

3 files are used in a sort:

1. Input ﬁle: File of unsorted input records.

2. Work or sort ﬁle: File used to store records temporarily during the sorting process.

3. Output ﬁle: File of sorted output records.

All these ﬁles would be deﬁned in the ENVIRONMENT DIVISION using standard ASSIGN clauses.

SORT data is usually assigned to a special work device indicated by SYSWORK.

SELECT SORT-FILE ASSIGN TO SYSWORK.

Your system may use SYSWORK (or some other special name) in the ASSIGN clause for the work or sort ﬁle. The SORT-FILE is actually assigned to a temporary work area that is used during processing but not saved.

FDs are used in the DATA DIVISION to deﬁne and describe the input and output ﬁles in a batch program in the usual way.   
The sort or work ﬁle is described with an SD entry (which is an abbreviation for sort ﬁle description).   
SD and FD  entries are very similar.

**Also note that the ﬁeld(s) speciﬁed as the KEY ﬁeld(s) for sorting purposes must be deﬁned as part of the sort record format.**

**SORT  SORT-FILE**

**ON ASCENDING KEY S-DEPT-NO  ->Deﬁned within the SD ﬁle  
USING UNSORTED-MASTER-FILE  
GIVING SORTED-MASTER-FILE**

**STOP RUN**

**Note: Here in cobol internal sort the sort key must be a field in sort-file i.e SD section.**

**SORT  SORT-FILE  
ON ASCENDING KEY TERR  
USING IN-FILE  
GIVING SORTED-MSTR**

This statement performs the following operations:

1.Opens IN-FILEand SORTED-MSTR.  
*2.Moves IN-FILE records to the SORT-FILE.*  
3.Sorts SORT-FILE into ascending sequence by TERR, which is a ﬁeld deﬁned as part of the SD SORT-FILE record.  
*4.Moves the sorted SORT-FILE to the output ﬁle called SORTED-MSTR.*  
5.Closes IN-FILE and SORTED-MSTR after all records have been processed.

Note that the records from the input file are first moved to the sort dataset.

The SORT statement can, however, be used in conjunction with procedures that process records before they are sorted and/or process records after they are sorted.

**INPUT PROCEDURE:**

We can perform certain processing of input records before they are sorted using INPUT PROCEDURE in place of USING clause.

Expanded format:

**SORT SORT-FILE ON ASCENDING KEY SR-EMP-NAME**

**INPUT PROCEDURE IS 2000-PROCESS-INPUT**

**GIVING OUTPUT-FILE**

The INPUT PROCEDURE processes data from the incoming file prior to sorting.

We may use INPUT procedure to perform:

1)    Data validations

2)    Eliminate records

3)    Eliminate unwanted fields

4)    Count number of records.

Earlier when using USING and GIVING clause( simple SORT), the opening and closing of the files was taken care by SOrT itself.   
But when we use the INPUT procedure we should remember that responsibility of opening and closing the input file would be with the input procedure.

Also note that we do not WRITE records to be sorted; instead, we RELEASE them for sorting purposes. We must release records to the sort ﬁle in an INPUT PROCEDURE. With a USING option, this is done for us automatically.

Note that the RELEASE verb is followed by a record-name, just like the WRITE statement.

That is, the RELEASE verb functions just like a WRITE but is used to output sort records. In summary, an INPUT PROCEDURE opens the input ﬁle, processes input records, and releases them to the sort ﬁle so they can then be sorted. After all input records are processed, the input ﬁle must be closed because when INPUT PROCEDURE is used, the input ﬁle is not automatically closed as it is when the USING clause is coded. The format for the RELEASE is:

The RELEASE is the verb used to write records to a sort ﬁle. **Hence we need to move the record first to sort-record and then release it.**

Example:

**MOVE IN-REC TO SORT-REC**

**RELEASE SORT-REC.**

Or

**RELEASE SORT-REC FROM IN-REC.       Functions like a WRITE ... FROM**

**Input Procedure summary:**

1) The INPUT PROCEDURE of the SORT should refer to a paragraph-name but it could refer to a section-name.

2) In the paragraph specified in the INPUT PROCEDURE:

a.       OPEN the input file.

b.       PERFORM a paragraph that will read and process input records until there is no more data.

c.       After all records have been processed, close the input file.

d.       After the last sentence in the INPUT PROCEDURE paragraph is executed, control will then return to the SORT, at which time the records in the sort file will be sorted.

3) At the paragraph that processes input records prior to sorting:

a.       Perform any operations on input that are required.

b.       MOVE input data to the sort record.

c.       RELEASE each sort record, which makes it available for sorting.

d.       Continue to read input until there is no more data.

Note, too, that we never OPEN or CLOSE the sort file-name specified in the SD. It is always opened and closed automatically, as are files specified with USING or GIVING. Only the input file processed in an INPUT PROCEDURE needs to be opened and closed by the program.

**SORT layout different from Input Layout:**

There could be a chance that not all the fields in the input file are required in the sorted output file.  
In such cases the sort file layout would be different that the input file layout.

It would be possible, although inefficient, to (1) first sort the input and produce a sorted master, and (2) then code a separate module to read from the sorted master, moving the data in a rearranged format to a new sorted master.

Instead we can use a input procedure and move only the required fields from the input to sort record and then release it.

**OUTPUT Procedure:**

OUTPUT PROCEDURE is very similar to the INPUT PROCEDURE except that an INPUT PROCEDURE processes presorted records and an OUTPUT PROCEDURE processes records in the sort file after they have been sorted.

The full format for the SORT, including both INPUT and OUTPUT PROCEDURE options, is as follows:

**Output procedure summary:**

1) The OUTPUT PROCEDURE of the SORT should refer to a paragraph-name, but it could refer to a section-name.

2) In the paragraph specified in the OUTPUT PROCEDURE:

          a. OPEN the output file.

          b. PERFORM a paragraph that will RETURN (which is like a READ) and process records from the sort file until there is no more data. The records are in sequence in the sort file.

          c. After all records have been processed, CLOSE the output file.

          d. When the OUTPUT PROCEDURE paragraph has been fully executed, control will then return to the SORT.

3) At the paragraph that processes the sort records after they have been sorted but before they are created as output:

          a. Perform any operations on the work or sort records.

          b. MOVE the work or sort record to the output area.

          c. WRITE each sort record to the output file. (A WRITE ... FROM can be used in place of a MOVE and WRITE.)

Recall that the SD file as well as files specified with USING or GIVING are opened and closed automatically. The programmer opens and closes the input file in an INPUT PROCEDURE and the output file in an OUTPUT PROCEDURE.

The word GIVING can be followed by more than one file-name, which means that we can create multiple copies of the sorted file.

 An OUTPUT PROCEDURE processes all sorted records in the sort file and handles the transfer of these records to the output file.

 In an INPUT PROCEDURE we RELEASE records to a sort file rather than writing them.

 In an OUTPUT PROCEDURE we RETURN records from the sort file rather than reading them. Syntax for RETURN is almost same as READ.

 RETURN Basically means read from the sort file and release basically means write to the sort file.

Format:

---

**Q. Cobol Merge Statemant?**

**ANS.**

**The Merge Statement:**

          COBOL has a MERGE statement that will combine two or more files into a single file.

          Its format is similar to that of the SORT:

Rules for ASCENDING/DESCENDING KEY, USING, GIVING, and OUTPUT PROCEDURE are the same as for the SORT.

With the USING clause, we indicate the files to be merged.   
**At least two file-names must be included for a merge, but more than two are permitted.**

Unlike the SORT, however, an INPUT PROCEDURE may not be specified with a MERGE statement.   
That is, using the MERGE statement, you may process records only after they have been merged, not before. (input has to come from files)  
The OUTPUT PROCEDURE has the same format as with the SORT.

The MERGE statement automatically handles the opening, closing, and input/output (READ/WRITE functions) associated with the files.

***The files to be merged must each be in sequence by the key field.***

 If ASCENDING KEY is specified, then the merged output file will have records in increasing order by key field, and if DESCENDING KEY is specified, the merged output file will have key fields from high to low.

**Q.Difference b/w EBCDIC & ASCII?**

**Ans.**

The sequencing of characters from lowest to highest, which is referred to as the collating sequence, is somewhat different in EBCDIC than ASCII.

Basic numeric sorting and basic alphabetic sorting are performed the same way in EBCDIC and ASCII.

 These codes are, however, not the same when alphanumeric ﬁelds containing both letters and digits or special characters are sorted.

* Letters are considered “less than” numbers in EBCDIC, and letters are considered “greater than” numbers in ASCII.
* Lowercase letters are considered “less than” uppercase letters in EBCDIC and “greater than” uppercase letters in ASCII.

**Q.Cobol Edit Characters/**

**ANS.**<http://www.csis.ul.ie/cobol/course/EditedPics.htm#part1>

There are four types of Insertion Editing:-

* Simple Insertion
* Special Insertion
* Fixed Insertion
* Floating Insertion

Insertion Editing is so called because the edit symbol is inserted into the data item at the same position it occupies in the picture clause.

Simple Insertion editing consists of specifying the relevant insertion character(s) in thePICTURE string. When a value is moved into the edited item, the insertion characters are inserted into the item at the position specified in the PICTURE.

The comma, the B, the 0, and the slash (/) are the Simple Insertion editing symbols.

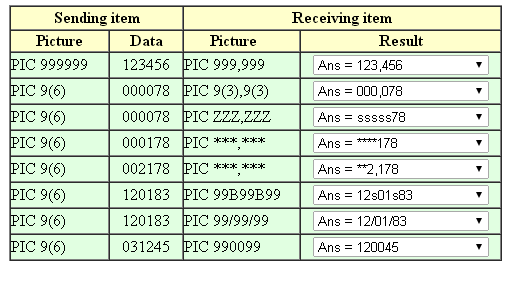
All Simple Insertion symbols count toward the number of characters printed or displayed. For instance, an item described as PIC 99/99/9999 will occupy 10 character positions when printed.

**The comma (,) symbol**  
The comma symbol (,) instructs the computer to insert a comma at the character position where the symbol occurs. The comma counts towards the size of the printed item. The comma symbol cannot be the first symbol in the PICTURE string.

If all characters to the left of the comma are zeros and zero-suppression is called for, the comma is replaced by the replacement symbol (asterisk or space).

**The space or blank (B) symbol**  
A space is inserted where the blank symbol (B) occurs.

**The slash and zero symbols (/ and 0)**  
A slash is inserted where the slash symbol (/) occurs, and a 0 is inserted where the zero symbol (0) occurs.

**EG:** 

**Special Insertion**

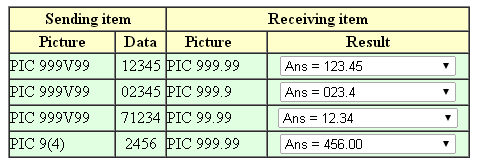
The decimal point is the only Special Insertion symbol. A decimal point is inserted in the character position where the symbol occurs.

**Notes**  
When a numeric data-item is moved into an edited data-item containing the decimal point symbol, alignment occurs along the position of the decimal point symbol, with zero-filling and truncation as necessary.

There may be only one decimal point in each edited picture clause.

The decimal point symbol cannot be mixed with either the V (assumed decimal point) or the P (scaling position) symbol.

EG:

****

**Fixed Insertion**

Fixed Insertion editing inserts the symbol at the beginning or end of the edited item.

The Fixed Insertion editing symbols are:

* the plus (+) and minus (-) signs,
* the letters CR and DB representing credit and debit,
* and the currency symbol usually the $ sign.

All symbols count toward the size of the printed item.

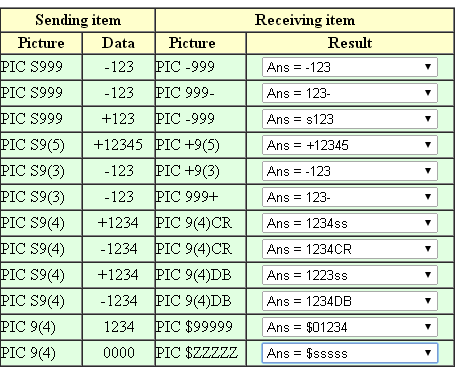
**Plus and minus symbols**  
These must appear in the leftmost or rightmost character positions and they count towards the size of the data item. They must be the first or last character in the PICTUREstring.

**Minus**   
If the sending item is negative, a minus sign is printed. If the sending item is positive, a space is printed instead. Use this to highlight negative values only.

**Plus**If the sending item is negative, a minus in printed and if the sending item is positive, a plus is inserted. Use this to when you always want the sign printed.

**CR and DB**  
CR and DB count towards the data item size and occupy two character positions. They may only appear in the rightmost position. Both are only printed if the sending item is negative. Otherwise two spaces are printed.

**The currency symbol (usually $).**  
The currency symbol must be the leftmost character and it counts towards the size of the item. It may be preceded by a plus or a minus sign.

****

**Floating Insertion**

The problem with using the fixed insertion symbols is that they can be somewhat unsightly. Values like $0045,345.56 or -0012 are more acceptablely presented as $45,345.56 and -12.

What makes these formats more presentable is that the leading zeros have been suppressed and the editing symbol has been "floated" up against the first non-zero digit. In COBOL this is achieved using Floating Insertion.

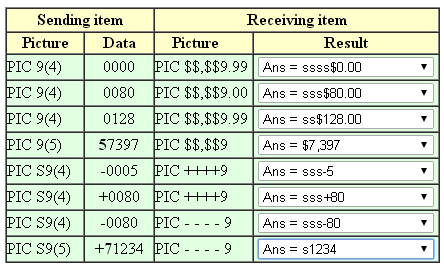
Floating Insertion suppresses leading zeros, and "floats" the insertion symbol up against the first non-zero digit.

The Floating Insertion symbols are;

* The plus and minus signs
* and the currency symbol.

Every floating symbol counts toward the size of the printed item.

Except for the left-most one, which is always printed, each Floating Insertion symbol is a placeholder that may be replaced by a digit. Accordingly, there will always be at least one symbol printed, even though this may be at the cost of truncating the number (see the fourth row in the example below.)

****

**Suppression and Replacement Editing**

Suppression and replacement editing is used to remove leading zeroes from the value to be edited. There are two varieties of suppression and replacement editing-

* Suppression of leading zeros and replacement with spaces
* Suppression of leading zeros and replacement with asterisks

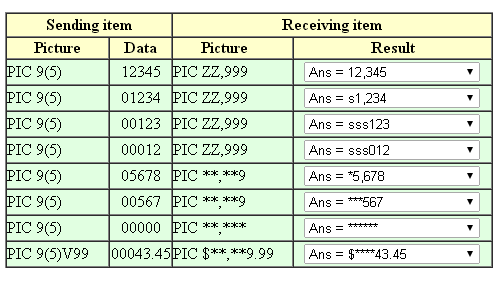
**Notes**The characters **Z** and **\*** are the suppression symbols.

Using Z in an editing picture, instructs the computer to suppress a leading zero in that character position and replace it with a space.

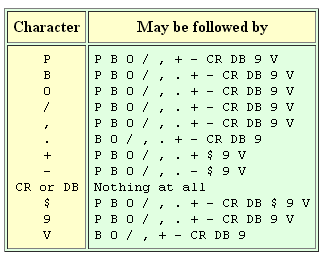
Using an \* in an editing picture, instructs the computer to suppress a leading zero in that character position and replace it with an \*.

If all the character positions in a data item are Z editing symbols and the sending item is 0 then only spaces will be printed.

If a Z or \* is used, the picture clause symbol 9, cannot appear to the left of it.

****

Some combinations of picture symbols are not permitted. The table below shows the combination of symbols that is allowed.



**Q.Cobolintrodcuction.**

**ANS.**

**Cobol** drew inspiration from **FLOW-MATIC**, its fore-runner. **Grace Hopper** the inventor of **FLOW-MATIC**, is the **mother** of **Cobol**.

**COBOL** (**CO**mmon**B**usiness **O**riented **L**anguage) is a high-level programming language suited to develop **business software** such as **payroll systems, accounting packages, ordering systems, billing software, flight-reservation systems** etc. Applications developed in Cobol are in wide-spread use across the globe.   
  
Major **banks** like Citibank, American Express, Bank of New York have their IT systems written in Cobol. In our own country, India, private banks like ICICI and HDFC allow you to bank *any-time*, *any-where*. All branches access software applications on a central server for their **core banking operations** such as, deposits/withdrawals, interest calculation, pass-book maintenance etc. CSC’s **Hogan** is a **core-banking solution(CBS)** developed using Cobol, employed by a myriad of banks around the world. Banking conglomerates like Mastercard, VISA use **Visionplus** software package for credit/debit/prepaid card processing. Visionplus is written in Cobol.

**Insurance companies** like AXA, Metropolitan Life, Statefarm insurance sell **life-insurance, annuities** etc. In life-insurance, when a proposed insured applies for a cover, data is entered into the computer system. **Under-writers** decide whether to accept the risk. **Actuarial tables** are used to calculate premium. The policy is then issued. When deaths occur, a claim is filed and the maturity value is paid. These companies have their own **home-grown software** written in Cobol for **policy processing**. Some companies prefer to buy an off-the-shelf product. Commercial products for Life and annuity policy administration such as **Ingenium** written in Cobol are very popular.   
  
The function of **billing** is common to several sectors such as **Energy and Utility Services, Telecommunication, Retail** etc. Southern California Edison(SCE), an **energy company** has software written in Cobol for invoicing. **Telecommunication** major British Telecom(BT) also uses applications built using Cobol for billing.   
  
**Manufacturing organizations** such as Ford, General Motors use software developed in Cobol for materials management, ordering etc.

|  |
| --- |
| **Q. What does COBOL stand for?** |
| COBOL stands for Common Business Oriented Language. COBOL Programs are used to process. Business-Data. COBOL Programs are easy to code and analyse, because it is an English-like language. Writing a COBOL program is just like writing a passage in English. |

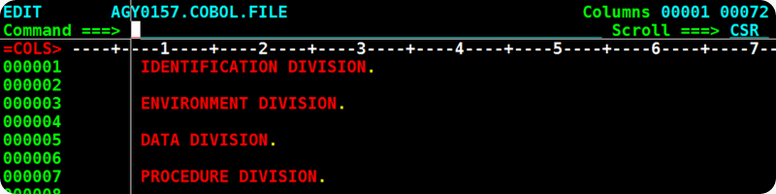
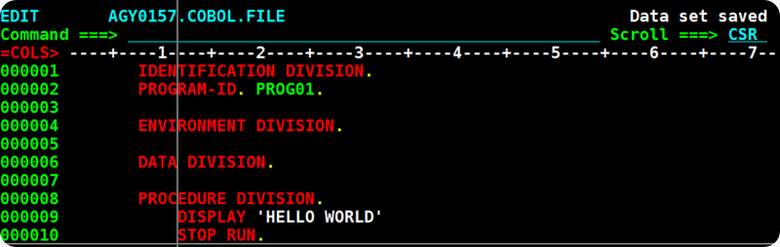
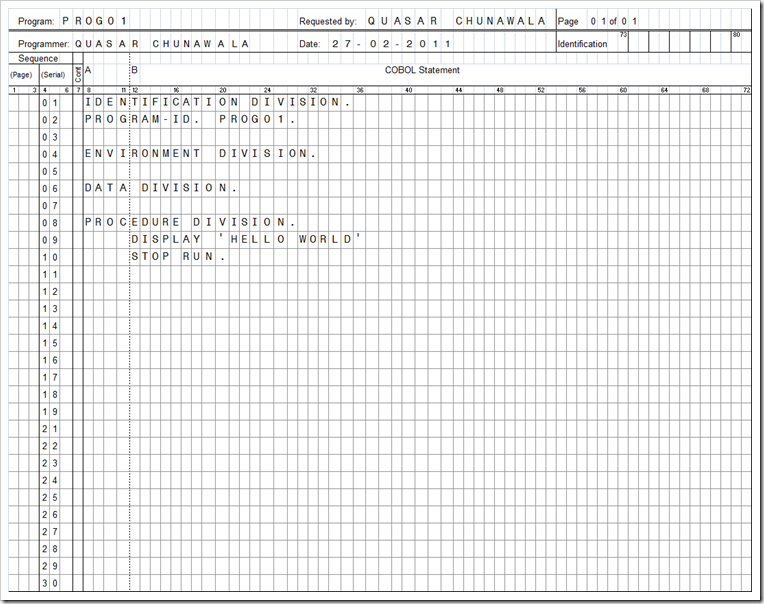
**Q. Are COBOL Programs English-like?**

Just as text in English is organised into sections and para’s, COBOL Programs can also be broken down into divisions, divisions further into sections, and sections into paragraphs. Each paragraph contain sentences, or COBOL Statements. Like the English language, sentences(statements) in COBOL should end with a period(full-stop).   
  
When you start new paragraphs in COBOL, you need to indent them, just as in English you follow indentation rules. COBOL then is whole-lot English like. Even instructions in COBOL, are words borrowed from English, ADD, SUBTRACT, MULTIPLY, DIVIDE etc.

**Q. What’s the basic skeleton of a COBOL Program**

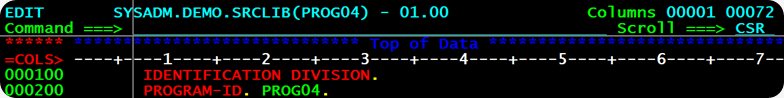
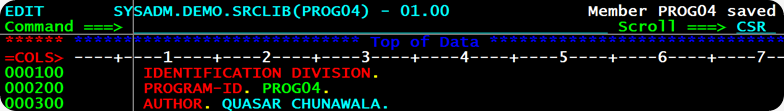
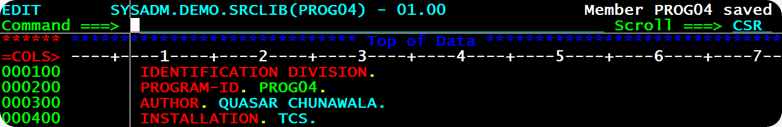
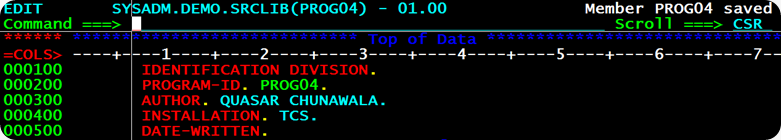
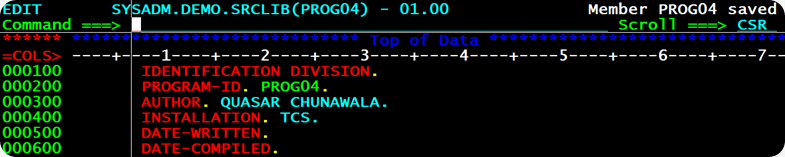
The basic skeleton of a COBOL Program is as follows :  
    
     [](http://lh6.ggpht.com/_sQvdFWqMlMg/TCbzULofFxI/AAAAAAAAC3w/Z51ryuA1fvs/s1600-h/Image198%5b1%5d%5b15%5d.png)  
  
A COBOL Program can be divided or broken down into 4 parts – IDENTIFICATION DIVISION, ENVIRONMENT DIVISION, DATA DIVISION and PROCEDURE DIVISION. The PROCEDURE DIVISION contains the instructions to be executed(performed), one after the other, step-by-step till you reach the end of the Program and stop running it(STOP RUN).   
  
Under a division, you may have several sections as well, and sections in turn can have several paragraphs. Divisions, sections and paragraphs help make the COBOL Program more structured, and the flow of the COBOL Program is easy to grasp.

**Q. What are the Alignment(Indentation rules) in COBOL?**

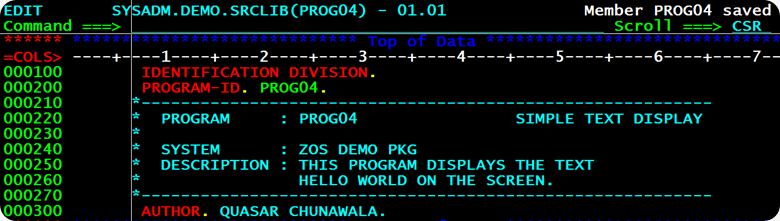
The lines in a COBOL Program are indented or aligned, as its done in English. When you write COBOL Program on the Mainframe terminal, you have 80 columns on a line to code on. The area spanning from columns 8-11 is called Area A. The area spanning from columns 12-72 is called Area B.   
  
In COBOL, division names, sections and paragraphs begin in Area A. See below, how I’ve coded IDENTIFICATION DIVISION, ENVIRONMENT DIVISION,..etc. starting on Column 8 -   
  
[](http://lh4.ggpht.com/_sQvdFWqMlMg/TCbzimaPIOI/AAAAAAAAC34/BgzI0jqtOM0/s1600-h/Image199%5b1%5d%5b5%5d.png)  
  
You code COBOL Instructions, COBOL Statements in Area B(that’s anywhere between Column 12 – 72). So, all your COBOL Instructions in the PROCEDURE DIVISION like ADD, SUBTRACT, MOVE should begin on or after Column 12. Look below, how I’ve coded COBOL Instructions in Area B.   
  
[](http://lh5.ggpht.com/_sQvdFWqMlMg/TCbzzMZidRI/AAAAAAAAC4A/bsu6pDnaBsk/s1600-h/Image200%5b1%5d%5b5%5d.png)    
  
An interesting bit of trivia, back in the early days, COBOL Programs were coded on a COBOLCoding-Sheet, before punching the Instructions onto a Punched-Card. Here's a picture of, how a Coding-Sheet. looked like.   
  
[](http://lh5.ggpht.com/_sQvdFWqMlMg/TWpYRZ3F-MI/AAAAAAAAEKY/MC06KVKhMNM/s1600-h/image%5b5%5d.png)

**Q. What is the IDENTIFICATION DIVISION used for?**

The IDENTIFICATION DIVISION is used to identify the COBOL Program to the system. Under the IDENTIFICATION DIVISION, you can code several para-entries like PROGRAM-ID, AUTHOR etc. which supply extra-descriptive information about the COBOL Program.

The PROGRAM-ID para assigns a unique-name to the COBOL-Program. You simply write the name of your COBOL Program after the PROGRAM-ID Para. Here’s how you code it -   
  
[](http://lh5.ggpht.com/_sQvdFWqMlMg/TJwMzu821fI/AAAAAAAADKM/b-GJVmHQJC4/s1600-h/Image262%5b5%5d.png)  
  
The AUTHOR Paragraph tells, who’s written this COBOL Program. If you’ve coded the COBOL Program, you can put your name as the AUTHOR of the program.   
  
[](http://lh5.ggpht.com/_sQvdFWqMlMg/TJwM58wJt6I/AAAAAAAADKU/GWHvHmNB1U8/s1600-h/Image263%5b5%5d.png)  
  
The INSTALLATION would be the name of the company, where the COBOL Software Program was designed and written. For example, if I work for TCS, I would code :  
  
[](http://lh5.ggpht.com/_sQvdFWqMlMg/TJwM-lOazYI/AAAAAAAADKc/JE7UpijUlQg/s1600-h/Image264%5b4%5d.png)  
  
The DATE-WRITTEN is the date when the COBOL Source-Program was coded. When you choose not to write a date, the COBOL Compiler software would insert a date for free.   
  
[](http://lh6.ggpht.com/_sQvdFWqMlMg/TJwNDf48DwI/AAAAAAAADKk/v0fbj3Id5k8/s1600-h/Image265%5b6%5d.png)  
  
The DATE-COMPILED is the date when the COBOL-Program was compiled. Again, if you don’t supply a Compile date, the Compiler software inserts one for you.   
  
[](http://lh6.ggpht.com/_sQvdFWqMlMg/TJwNIGupcWI/AAAAAAAADKs/KaN3wqSxqvc/s1600-h/Image266%5b5%5d.png)  
  
Under the IDENTIFICATION DIVISION, you must compulsorily code the PROGRAM-ID Para, you can skip off coding the AUTHOR, INSTALLATION, DATE-WRITTEN and DATE-COMPILE Entries. PROGRAM-ID Para is must.

**Q. How do you document a COBOL Program?**

A well-documented COBOL Program is easy to analyse. You can quickly understand the functionality of the COBOL Program, if it contains comments. In COBOL, you code comments by putting an asterisk \* in Column 7.   
  
[](http://lh5.ggpht.com/_sQvdFWqMlMg/TJ1hlexSJHI/AAAAAAAADK0/RlWOcFFwecc/s1600-h/Image2675.png)  
  
You must always supply document what the program-name is, which system, a brief description of the processing it does.

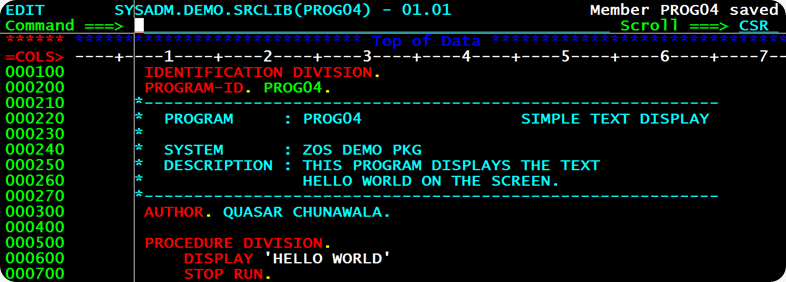
**Q. What is PROCEDURE DIVISION?**

The PROCEDURE DIVISION is the starting-point of the COBOL-Program. This is where the Mainframe begins to run(execute the COBOL instructions). Hence, all your instructions/programming logic must be written inside this PROCEDURE DIVISION.

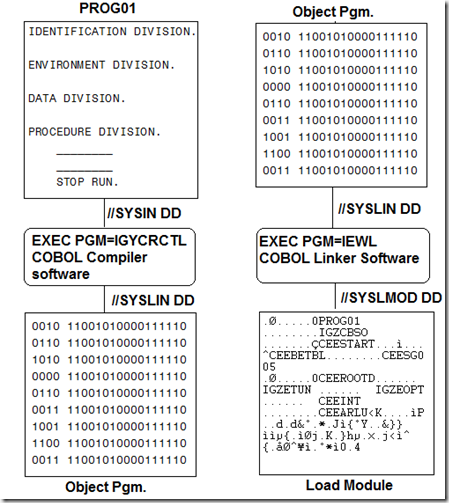
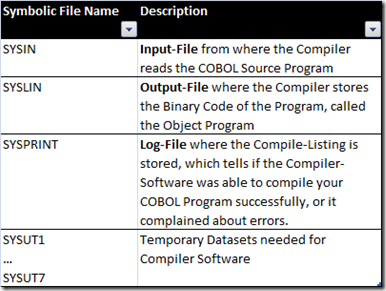
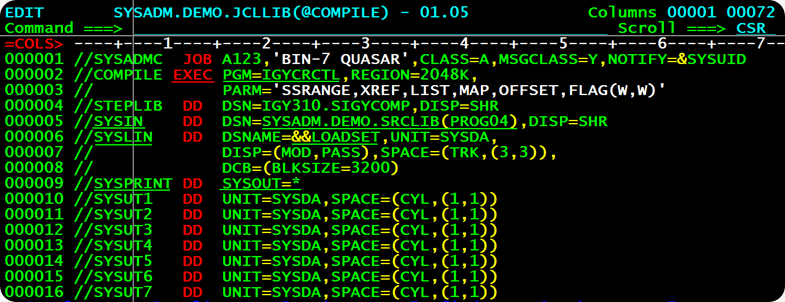
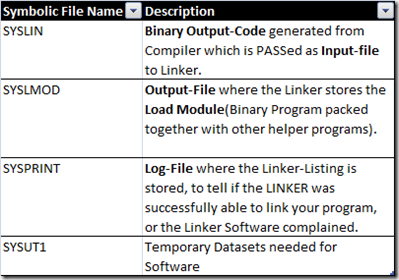
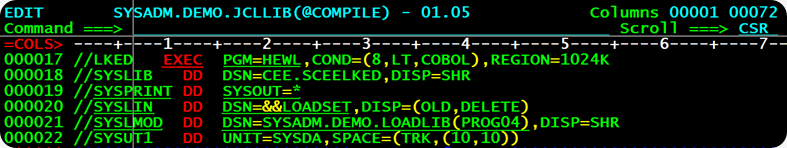
Under the PROCEDURE DIVISION, you can write Instructions one-after-the other. The Mainframe executes, runs the instruction in the PROCEDURE DIVISION, one-by-one, step-by-step sequentially, till you say, STOP RUNning the Program. To stop running the COBOL Program, the last-instruction you code must be STOP RUN.

PROCEDURE DIVISION.  
   Instruction-1   
   Instruction-2   
   Instruction-3   
   Instruction-4   
   ...   
   Instruction-N   
   STOP RUN.

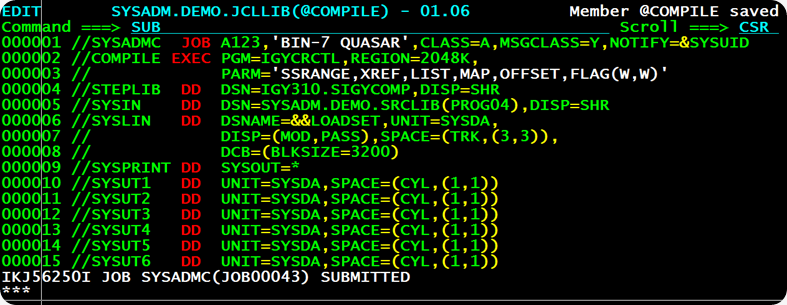
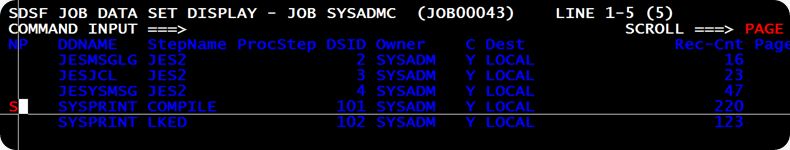
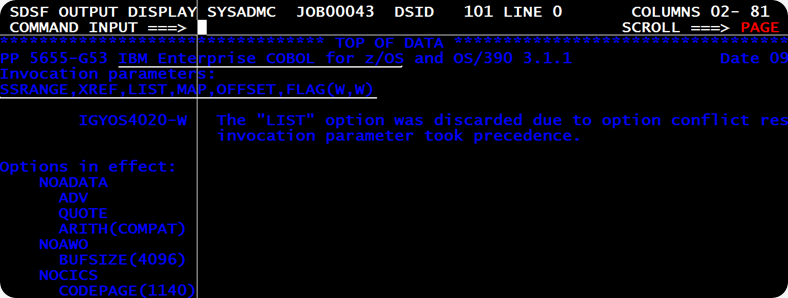
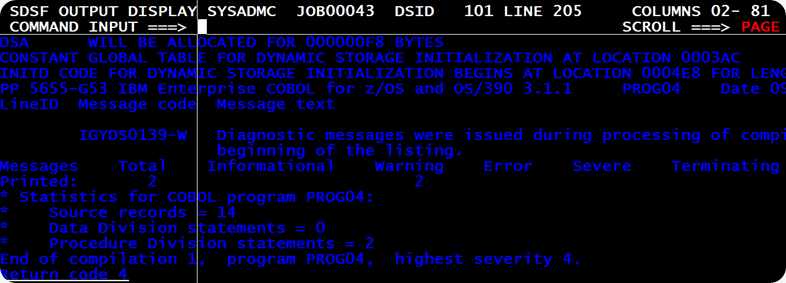
**Q. How do you display text on the Screen in COBOL?**

To Display data on the Screen, in COBOL you code the DISPLAY Instruction. You code DISPLAY followed by the text that want to print on the Screen, enclosed in Single Quotes. Say for example, if I want to display Hello World on the Screen, I would write -   
  
[](http://lh3.ggpht.com/_sQvdFWqMlMg/TJ1hn1aCSQI/AAAAAAAADK8/njrwPFFrdvs/s1600-h/Image2686.png)

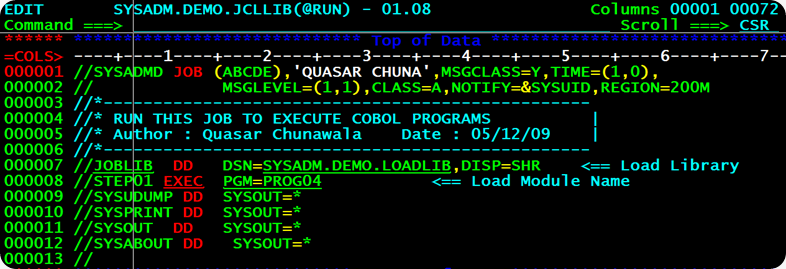
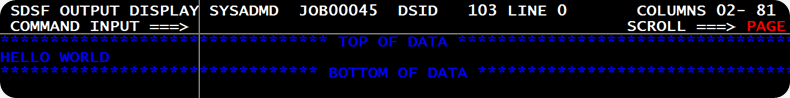
**Q. How do I setup  Compile-Link Job?**

To run and process data using a Program, you need to first Compile-Link a COBOL Program. The Free COBOL-Compiler Software on Mainframes is IGYCRCTL. The Compiler-Software works like a Translator, translates the Program from   
COBOL to Binary Language.   
                [](http://lh4.ggpht.com/_sQvdFWqMlMg/TJ1hpzMn-1I/AAAAAAAADLE/ORLiUpw8QOU/s1600-h/Image2696.png)  
  
Like any other Program on Mainframes, the free Compiler IGYCRCTL also expects you to supply some Input-Output Files.    
                     [](http://lh3.ggpht.com/_sQvdFWqMlMg/TJ1hsX0YmYI/AAAAAAAADLM/9M8F6mlDdB4/s1600-h/Image270%5b7%5d.png)  
To run the Compiler-Software IGYCRCTL, I set-up a job like this – I’ve set //SYSIN DD Input-File to point to my COBOL Source Program which resides at SYSADM.DEMO.SRCLIB(PROG04). //SYSLIN DD Output-File, where the compiler stores Binary Object code points to a Temporary-FileLOADSET. The //SYSPRINT DD File is set to point to the SpoolSYSOUT=\*. This contains the Compile-Listing. I look at the Compile-Listing, and know if my program successfully compiled, or the compiler complained about my Program(its faulty, has syntax errors). //SYSUT1 to //SYSUT7are temporary-work-Files.   
  
[](http://lh3.ggpht.com/_sQvdFWqMlMg/TJ1hu5d-5cI/AAAAAAAADLU/vQJ6lTZdhng/s1600-h/Image271%5b7%5d.png)  
  
The Output Binary-Object-Code generated by the Compiler-Software IGYCRCTL, cannot  run on its own, standalone. To get it running, it needs other Helper Programs. The Linker-Software packs and bundles together my Binary-Program along-with any other Helper-Programs, it may need into one composite Load-Module. My Program is all set to go! The Free COBOL-Linker Software on Mainframes is HEWL. It expects the following files -    
                     [](http://lh6.ggpht.com/_sQvdFWqMlMg/TJ1hxNi4WfI/AAAAAAAADLc/ovEULjU1aPA/s1600-h/Image272%5b4%5d.png)  
To run the Linker Software HEWL, I added a LKED Step to my job. I have set the //SYSLIN DD Input-File to point to BinaryObject-Code stored at the   
Temporary-DatasetLOADSET created in the COMPILE Step. Bundled-up Output Load Module of the Linker HEWL is stored in //SYSLMOD File. I have made the //SYSLMOD DD File point to the dataset SYSADM.DEMO.LOADLIB(PROG04). The Load-Module PROG04 would be stored in my personal Load-Library SYSADM.DEMO.LOADLIB.   
  
[](http://lh6.ggpht.com/_sQvdFWqMlMg/TJ1hzafDQ6I/AAAAAAAADLk/7ZdyWC1vKpA/s1600-h/Image273%5b4%5d.png)

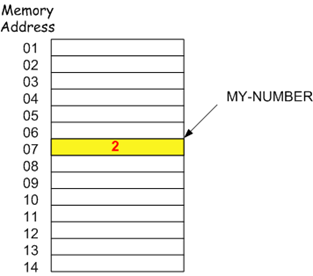
**Q. I’ve hit SUB on the Compile-Link Job. What next?**

Hit SUBmit on the Compile-Link job after you’ve set it up.   
  
[](http://lh4.ggpht.com/_sQvdFWqMlMg/TJ246L3a1CI/AAAAAAAADLs/QiGcmoonvIA/s1600-h/Image274%5b8%5d.png)    
Start SDSF to see the Output of the Compile-Link Job in the Spool, and check Compile SYSPRINT-Listing for any errors.   
  
[](http://lh6.ggpht.com/_sQvdFWqMlMg/TJ248lEutGI/AAAAAAAADL0/MWOEk64W5xs/s1600-h/Image275%5b4%5d.png)  
  
Here’s how, the Compile-Listing looks. It shows that I running Enterprise COBOL Version. The SSRANGE, XREF, LIST, MAP, OFFSET Compiler options have been turned ON.   
  
[](http://lh5.ggpht.com/_sQvdFWqMlMg/TJ24-SD-bvI/AAAAAAAADL8/aIVcY-2SnFk/s1600-h/Image277%5b5%5d.png)  
  
The Compiler Software IGYCRCTL leaves behind  signature, a Two-Digit Return-Code as a signal, to indicate if the COBOL-Program compiled successfully, or it contained errors. Check the Return-Code set by the Compiler, by scrolling to the bottom of the Compile-Listing. The Return-Code set by the Compiler is RC=04. A Zero RC suggests, the COBOL Program is correct, an RC=04 denotes Warnings(W). This is acceptable. When the Compiler sets the RC=08, the COBOL Program contains Errors(E), and an RC=12 says the COBOL Program has   
Severe-Errors(S). When the Compiler complains with Errors(E) and   
Severe Messages(S), you need to go back and correct your COBOL Program.   
  
[](http://lh4.ggpht.com/_sQvdFWqMlMg/TJ25AlhkKRI/AAAAAAAADME/k6fcmrXT-cs/s1600-h/Image276%5b5%5d.png)

**Q. How do I run my Program?**

To run your program , you need to write a Job. I’d run my Program, by setting up a job like this -   
  
[](http://lh6.ggpht.com/_sQvdFWqMlMg/TJ69RLZTIGI/AAAAAAAADMM/9aQ8bokw91Y/s1600-h/Image278%5b6%5d.png)  
  
I need to run the Program PROG04, so I write EXEC PGM=PROG04. The Mainframe Computer should pickup this Load Module from my Personal Load Library SYSADM.DEMO.LOADLIB, so I added a //JOBLIB DD Statement pointing to my   
Load-Library. When I SUBmit this Job, the Output Listing in the Spool, shows up as -   
  
[](http://lh3.ggpht.com/_sQvdFWqMlMg/TJ69TMDcj-I/AAAAAAAADMU/LAm9oO92mN8/s1600-h/Image279%5b6%5d.png)

**Q. What is a Variable? What are Literals?**

A Computer Program takes Data as Input, performs processing on the Input Data, and produces and Output. You would like to store the Input Data and Output Results in Computer Memory, so that you can retrieve it for later purposes.   
  
Let's take a look at how ComputerMemory looks like. Just like on the street people live in houses, ComputerMemory is organised as a series of Cells. These cells do not house people, instead they house Data.   
  
You can visualize a picture of computer memory like the one below:   
  
 [](http://lh4.ggpht.com/_sQvdFWqMlMg/TSlsIUmpXII/AAAAAAAAD8U/7hcKuZ_wzwY/s1600-h/image%5b3%5d.png)  
Suppose you have stored the number 2, at some Memory Location in the Computer Storage. Next time, you want to retrieve the contents of this cell. How to go about it? In ComputerMemory, how do you refer to a particular Cell or Memory Location? In the real world, houses on a street have different names.   
  
In a similar fashion, what you have to do is, you need to assign a name to the Computer Memory Location or House, say MY-NUMBER. And then, you can access the contents of this Memory Location, using the name MY-NUMBER. Next time, you simply have to say, "DISPLAY the contents of MY-NUMBER", and you'll get the Output=2.   
  
I am going to tell you a little more about the data(contents) stored inside a Computer Memory Location. If the data stored in a Computer Memory Location can change, for example if the value in MY-NUMBER can be changed(modified) to 3, such a Memory Location is called Variable. The word Variable is due to the fact, that the contents of such a Cell can vary or change. On the other hand, a Literal means constant data.   
  
**Together, Variables and Literals are Computer Storage Areas, where data is kept, and identified a unique name.**

**Q. What are the rules for naming Variables?**

Every Variable(Computer Storage Area) must have a name. It's a good idea to assign relevant and meaningful names to variables. For example, PRINCIPAL, NUMBER-OF-YEARS, RATE-OF-INTEREST are  self-explanatory names.   
  
Whitespaces are not allowed. If a Variable name has multiple words, separate them by Hyphens, like AREA-OF-CIRCLE

**Q. What is declaration of Variables? What is DATA DIVSION?**

You can't directly store data in a Variable(Mainframe Computer Storage Area). First, you must declare or announce the Variable.   
  
Declaration actually causes the Mainframe Computer, to keep aside Storage Space for your Data. During declaration, you must specify exactly how many Bytes of Space you need, to store your Data – One Byte, Two Bytes, Three Bytes how much? Depending on your space requirements, the Mainframe Computer honours your request, and exclusively reserves(books) Storage Space for you. Now, you can go ahead and store some data in it.   
  
Remember that, the MainframeComputer is like a miser. It does not give away any Bytes for free. Even if you want a single byte of Memory Space, you must first ask the Mainframe Computer for it. Telling the MainframeComputer, how much space you need to store Data, is called Declaration.   
  
DATA DIVISION contains the declaration(list) of all the Variables(Computer Storage Areas), you want to use in the COBOL Program.

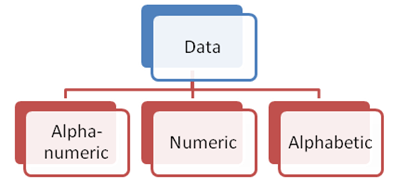
**Q. How to declare Variables in the DATA DIVISION?**

Declaring a Variable in COBOL is very easy. In COBOL, you code the data-name of the Variable, followed by Data-type. Suppose, I want to calculate Simple Interest on Rs. 1000, for 5 years, at 20 percent Interest.  
  
The assumption is, to store one character you need 1 Byte. To store the Input Data-Items on the Mainframe Computer, I shall need three Variables – PRINCIPAL(4 Bytes large), NUMBER-OF-YEARS(1-Byte) and RATE-OF-INTEREST(2-Bytes). How to declare these variables in COBOL? You always code the Data-Name followed by Data-Type.

[?](http://www.mainframes360.com/2009/08/cobol-tutorial-data-division-part-i.html)

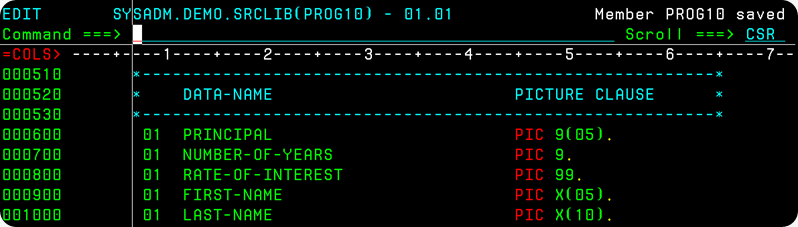
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | ----+----1----+----2----+----3----+----4----+----5----+----6----+----7--        \*---------------------------------------------------------------\*        \*    DATA-NAME                        DATA-TYPE                 \*        \*---------------------------------------------------------------\*         01  PRINCIPAL                        PIC 9999.         01  NUMBER-OF-YEARS                  PIC 99.         01  RATE-OF-INTEREST                 PIC 99. |

**Q. What are data-types? What are the basic data-types in COBOL?**

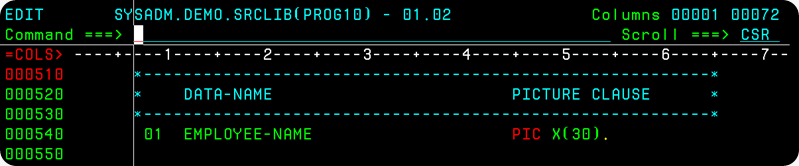
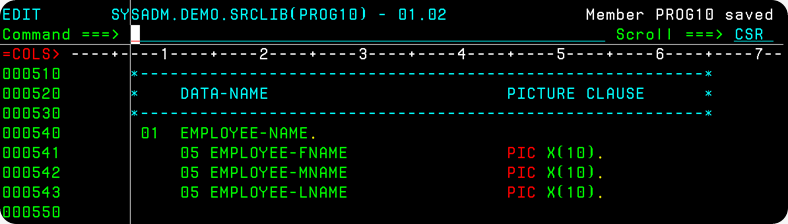
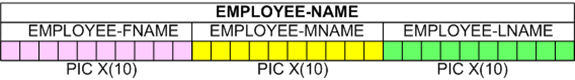
Data-type indicates whether a variable can hold Alphabetic Characters such as 'A','B','C',... etc. or numeric data such as 123, –500, 6159, ... etc. The data that you can store in a COBOL-Variable falls into one of these classes :  
  
[](http://lh3.ggpht.com/_sQvdFWqMlMg/TSmjdsWDhRI/AAAAAAAAD8k/IGBTaoSxHmY/s1600-h/image%5b17%5d.png)  
  
Alpha-numeric data consists of Alphabetic character and numbers. For example,    
'HELLO 123 @$','INDIA IS THE 3RD LARGEST ECONOMY' are alpha-numeric strings. In COBOL, the Symbol X implies Alpha-numeric.   
  
Numeric Data refers to numbers which are used in Arithmetic-computation. 123, 3.14159, –642.70 are examples of Numeric Data. In COBOL, the Symbol 9  implies Numeric.   
  
Alphabetic Data refers to Non-Numeric Data. 'HELLO HOW DO YOU DO', 'NOT BAD!' are examples of pure Alphabetic-Strings. In COBOL, the Symbol A implies Alphabetic.

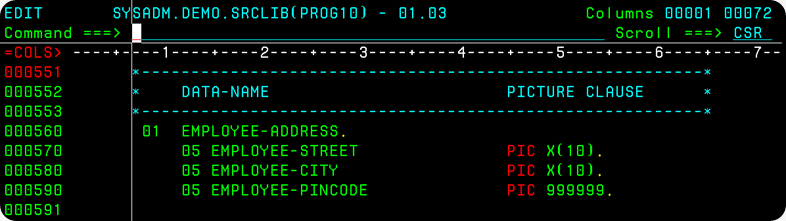
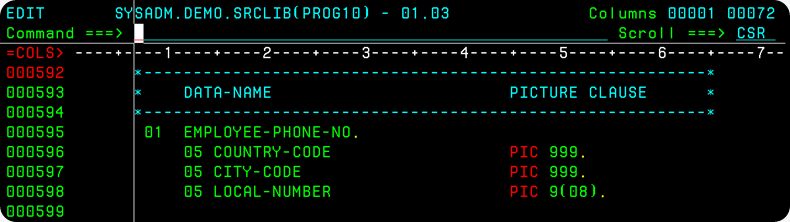
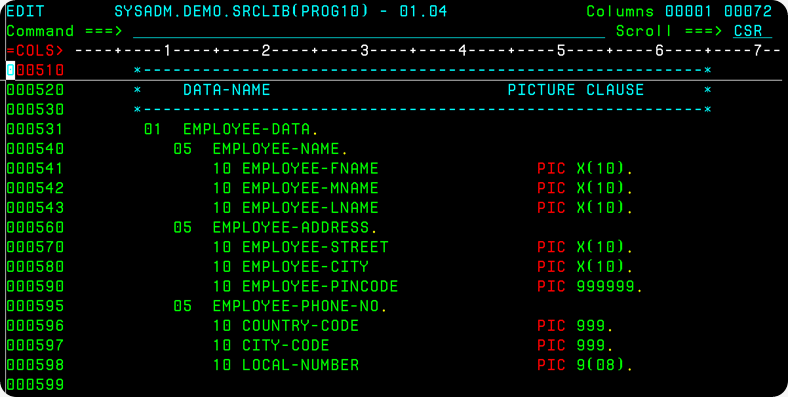
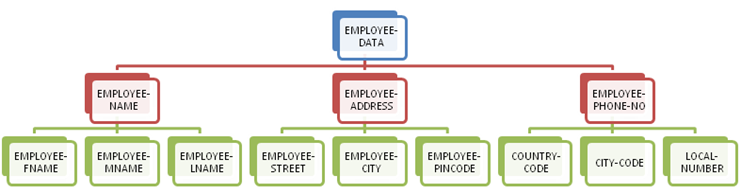
**Q. What is PICTURE Clause?**

PICTURE Clause specifies the Data-Type and Size(in Bytes) of a Variable. PICTURE Clause is coded after the Data-Name. You may code PICTURE or simply PIC.   
  
[](http://lh6.ggpht.com/_sQvdFWqMlMg/TSmjf3mN1NI/AAAAAAAAD8s/tlasPhD8W5s/s1600-h/image%5b27%5d.png)  
  
PRINCIPAL Variable is defined as PIC 99999. 9 means PRINCIPAL Variable can hold Numeric Data. As it is a PIC 99999, five times, PRINCIPAL Variable occupies 5-Bytes of Storage space. Generally, you can store one character in a Byte. So, in 5-Bytes of Storage Space, you can store a number upto Five-Digits large.   
  
NUMBER-OF-YEARS Variable is defined as PIC 9. NUMBER-OF-YEARS occupies 1-Byte of Storage-Space. In 1-Byte Space, you can store a Single-Digit Number.   
  
RATE-OF-INTEREST Variable is defined as PIC 99, which suggests it is 2-Bytes big and numeric type. In 2-Bytes Space, you store a number upto .  
  
FIRST-NAME Variable is specified as PIC XXXXXX. X stands for Alpha-numeric, so FIRST-NAME can hold alpha-numeric Data. Further, as its PIC XXXXXX(Six Times), you can store a alphanumeric Textual-word upto Six Characters large in it. Size or Length of FIRST-NAME is Six.   
  
LAST-NAME Variable is specified as PIC XXXXXXXXXX. This means, you can store an alpha-numeric string  upto ten characters long in the LAST-NAME Variable.   
  
You can code PIC XXXXXXXXXX, in short as PIC X(10). Similarly, you may code

PIC 99999 as PIC 9(05) in short.  
  
[](http://lh5.ggpht.com/_sQvdFWqMlMg/TSmjiiZJc5I/AAAAAAAAD80/kQRvrS46KW4/s1600-h/image%5b33%5d.png)

**Q. What are group and Elementary Data Items?**

COBOL provides the facility to provide a detailed-breakup of a Variable(Computer Storage Area). A field or variable can be broken down further into smaller sub-fields. Consider the name of an Employee stored in the EMPLOYEE-NAME variable. It is declared as follows -   
  
[](http://lh5.ggpht.com/_sQvdFWqMlMg/TS6JAxS7N8I/AAAAAAAAD88/S7SpkiUWHT0/s1600-h/image%5b6%5d.png)    
  
The Employee's name contains his First-Name, Middle-Name and his Last-Name, all put together. Therefore, it is possible to divide the EMPLOYEE-NAME into three parts EMPLOYEE-FNAME, EMPLOYEE-MNAME and EMPLOYEE-LNAME.   
  
[](http://lh4.ggpht.com/_sQvdFWqMlMg/TS6JC2e-nzI/AAAAAAAAD9E/_vJcoNmUEJY/s1600-h/image%5b13%5d.png)    
  
I have broken down the EMPLOYEE-NAME 30-Character Field, into EMPLOYEE-FNAME(10 characters) EMPLOYEE-MNAME(10 Characters) and EMPLOYEE-LNAME(10 Characters) fields. The EMPLOYEE-NAME is called a GroupData-Item. The sub-ordinate data-items under it – EMPLOYEE-FNAME, EMPLOYEE-MNAME and EMPLOYEE-LNAME are called Elementary or Simple Data-Items. A visual representation of the EMPLOYEE-NAME Area and its break-up is shown below.   
  
 [](http://lh6.ggpht.com/_sQvdFWqMlMg/TS-Z6PSas7I/AAAAAAAAD9M/FyoHYibHK8g/s1600-h/image%5b18%5d.png)

In a similar fashion, the Address of an Employee generally consists of a Street, a City and Pin-Code. Therefore, in the COBOL Program you may specify a detailed break-up of Address like this.    
  
[](http://lh5.ggpht.com/_sQvdFWqMlMg/TS-Z7xWGqLI/AAAAAAAAD9U/ePpseui36wA/s1600-h/image%5b24%5d.png)  
  
The Group Data-item Address is composed of Street, a 10-Byte Alphanumeric Field, City being a 10-byte alphanumeric field again and pin-code, a 6-digit numeric field. The Address Field has a sum-total size of 10 Bytes + 10 Bytes + 6 Bytes = 26 Bytes. Pictorially it may be represented as follows.    
  
 [](http://lh6.ggpht.com/_sQvdFWqMlMg/TS-Z9EoGGnI/AAAAAAAAD9c/PzvT0b_2_PM/s1600-h/image%5b29%5d.png)  
Likewise, the phone-number of an Employee, would consist of Country- Code, City-Code and the Actual number. Look at, how I've coded EMPLOYEE-PHONE-NO Group-Item in COBOL.   
  
[](http://lh6.ggpht.com/_sQvdFWqMlMg/TS-cuhLk6EI/AAAAAAAAD9k/0YZLyeOKvUk/s1600-h/image%5b34%5d.png)  
  
The Full-Name of the Employee, the Address of the Employee and his   
Phone-No.together represent an Employee's Data. COBOL allows aggregation, putting together Data-items, under a head, a higher-level data-item. So, I have clubbed EMPLOYEE-NAME, EMPLOYEE-ADDRESS and EMPLOYEE-PHONE-NO under one roof, EMPLOYEE-DATA.    
  
[](http://lh5.ggpht.com/_sQvdFWqMlMg/TTEWtDxbgdI/AAAAAAAAD9s/A9rnqGz3A1w/s1600-h/image%5b5%5d.png)  
  
Take a look at the above picture. I'll just quickly run you through the EMPLOYEE-DATA Item's structure. EMPLOYEE-DATA is used to hold or store the data of an Employee. The Group Item EMPLOYEE-DATA is broken down into EMPLOYEE-NAME, EMPLOYEE-ADDRESS and EMPLOYEE-PHONE-NO.    
 [image](http://lh6.ggpht.com/_sQvdFWqMlMg/TTFI4W_D0_I/AAAAAAAAD-A/pU4VZAsSvTk/s1600-h/image20.png)    
EMPLOYEE-NAME is a group-item in turn, consisting of EMPLOYEE-FNAME, EMPLOYEE-MNAME and EMPLOYEE-LNAME Elementary Items.   
  
EMPLOYEE-ADDRESS is a group-item internally made up of EMPLOYEE-STREET, EMPLOYEE-CITY and EMPLOYEE-PINCODE Elementary Items.   
  
The COUNTRY-CODE, CITY-CODE and LOCAL-NUMBER together constitute the Group-Item EMPLOYEE-PHONE-NO.   
  
[](http://lh3.ggpht.com/_sQvdFWqMlMg/TTFI7OORqRI/AAAAAAAAD-I/jSW3CC6ZVZg/s1600-h/image15.png)  
The EMPLOYEE-DATA Item can be represented with the help of a Inverted Hierarchical Tree-Like picture shown above. This is the structure of EMPLOYEE-DATA. In this manner, COBOL allows you to specify the format or structure of the data, by creating Group and Elementary Data-Items.

Q: What is CALL statement in COBOL?

Call Statement in cobol calls sub program from main program

by Content or by reference but default is by reference

eg.

CALL "sub-prog".

CALL "sub-prg" using var-1, var-2.

CALL program-name using var-1, var-2.

Call Statement in cobol calls sub program from main program

by Content or by reference but default is by reference

syntex is

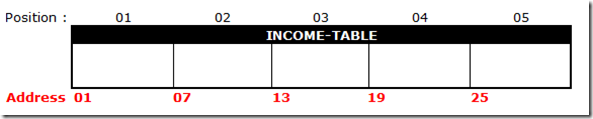
CALL program-name by Content/Reference by Var-1, Var-2.

**Q. What is a table(array) in Cobol?**

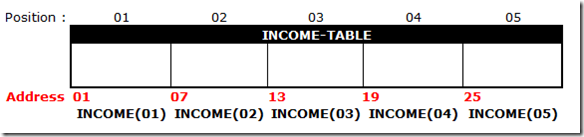
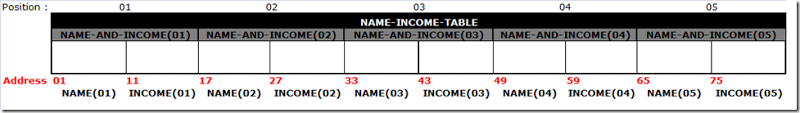
In the Cobol language, you can declare variables to hold simple integer numbers like 1, 2, 3, 2400, -6 and so forth, real numbers like 3.14159, 6.023 x 10-23, text-strings like 'HELLO WORLD'. A variable can hold only oneinstance of a data-item. It can accommodate just one value.   
  
For example, the variable MY-BOOK has a picture string X(30). The variable MY-BOOK can hold just one item – only one book’s name like 'SNOW WHITE AND SEVEN DWARFS'.   
  
[](http://lh6.ggpht.com/-rym4r5Uqcmc/UG-VvzwDjqI/AAAAAAAAE7Q/KFpptphPiD4/s1600-h/image%5b12%5d.png)  
  
In your day-to-day life, you often come across a list of items, say your shopping list, your phone-book(list of contacts), your favourite books(list of books) etc. What if, you would like to store a list of related-items together?   
  
An Array or a Table in Cobol can hold a listofdataitems**.** A table is a list, a collection of similar data-items. Tables store multiple data items of the same type. Take data items, which are logically-related to each other, and put them together, in a table. For example, a table of integer numbers = {1, 2, 5, 8, -6}.

**Q. How do I define and create tables?**

Suppose, you would like to store the incomes of five employees  {Rs. 1000, Rs. 2000, Rs.5000, Rs. 6000, Rs.8000} together in a table.    
  
In Cobol, each table should be assigned a uniquename. This is called the table-name. For example, I might call the table as the ‘Income Table’.   
  
The ‘Income table’ is said to have five members – 1000, 2000, 5000, 6000 and 8000. These are known as the elements of the table. The number of elements in a table is the table size or length.   
  
I start off by creating a simple variable INCOME. The variable INCOME has a picture string PIC 9999V99, six bytes. It can hold just one item – only one salary like 1000.00.   
  
[](http://lh5.ggpht.com/-oKSoLWNYcuI/UG-pKTtsgcI/AAAAAAAAE70/ms1rFW7GI1o/s1600-h/image%5b17%5d.png)  
  
The OCCURS clause in Cobol is used to setup a table. It specifies the number of elements, the size of the table. To create a list of five incomes, I code it as OCCURS 05 TIMES. To create a list of ten incomes, I shall write OCCURS 10 TIMES and so forth. **The OCCURS clause follows the picture string.**  
[](http://lh6.ggpht.com/-Za53bnAnLjg/UG-pM1iHtlI/AAAAAAAAE8A/n4x88_b1U6A/s1600-h/image%5b22%5d.png)  
  
I have created a list of five incomes. But, I must assign a unique name to this list or table. In India, the entire force of elite commando’s that are highly trained, are the National Security Guards(NSG). Similarly in my Cobol program, the entire list of five income items, are to be assigned a single top-level name. **This table-name must be declared at the 01-Level.** I shall call it as the INCOME-TABLE.

[](http://lh6.ggpht.com/-1cQmL3eLw9E/UG-pPIiePSI/AAAAAAAAE8U/IsU0IiHdYpI/s1600-h/image%5b27%5d.png)    
  
On declaring the INCOME-TABLE, space is allocated, set-aside in computer memory as illustrated in the picture below. The striking thing about arrays is, that neighbouring or consecutive storage locations in computer memory are reserved for storing the incomes.    
  
  [](http://lh3.ggpht.com/-nxRoMBjokII/UG-s7gXlHyI/AAAAAAAAE9I/2xQDonCxEHY/s1600-h/image%5b45%5d.png)  
  
Study the above memory map carefully. The first INCOME element of the INCOME-TABLE would be stored at memory address 01. INCOME’s are six bytes in size. So, the second INCOME element would be stored at the address 01 + Six Bytes = 07. The third at 07 + Six bytes = 13, fourth at 13 + Six Bytes = 19 and so forth.Consecutivememorylocations are booked to store the elements of an array.   
  
To re-iterate, an array is simply a set of values stored in consecutive storage locations and assigned one data-name.

**Q. How do you refer to individual elements of a table?**

You refer to the individual elements of a table, by the position of that element. Which INCOME element do you want – first, second, third,...? The first INCOME element is referred to as   
INCOME(01). The second INCOME element is referred to as INCOME(02). In general, the n’th INCOME element is referred to as INCOME(n).   
  
   [](http://lh5.ggpht.com/-Xx6uMMrlLu8/UG-s-kd_czI/AAAAAAAAE9Y/IgIOVW3YVfo/s1600-h/image%5b40%5d.png)  
  
Bear in mind that, INCOME(03) refers to the third INCOME field(starting at address 13) alone. If we write INCOME-TABLE, it refers to the entire table of five INCOME Fields together.   
  
As a further illustration of a 1-Dimensional table, assume that we would like to store the include the names of the employee’s along with their corresponding income figures as shown in the picture below.   
  
[](http://lh5.ggpht.com/-4ZQZB2lNgTk/UG-ztdQg7qI/AAAAAAAAE98/LG2tkdgt7DM/s1600-h/image%5b50%5d.png)  
  
This time, the NAME and the INCOME fields both OCCURS 05 TIMES. I can club the NAME and INCOME Fields under a single group item, NAME-AND-INCOME which will OCCUR 05 TIMES. The table name shall be NAME-INCOME-TABLE.   
  
[](http://lh3.ggpht.com/-SH59UIPi_KM/UG-zwKlDijI/AAAAAAAAE-M/BSie8RDKamQ/s1600-h/image%5b55%5d.png)  
  
If we write NAME(01), we are referring to the first NAME field of ten bytes(at address 01). If we write NAME(02), it indicates the second NAME field of ten bytes(at address 17). If we write INCOME(02), it points to the second INCOME field of six bytes(at address 27). If we write NAME-AND-INCOME(02), it implies both the NAME(02) and INCOME(02) Fields starting at address 17. The NAME-INCOME-TABLE stands for the entire table of ten storage fields.   
The total size of the table = 10 bytes x 5 fields + 06 bytes x 5 fields = 80 Bytes.

**Q.Cobolfile handling?**

**ANS.**

**Q. diffrence between renames and redifnes with examples**

**ANS.**

Redefines are used to reuse the storage space where as

renames are used to regroup the elementary items.

Redefines are coded with 0 to 49 level numbers where

as renames are declared with 66 level no.

pic clause is required for redefines where as pic clauseis not used for renames.

**Ex:**

**For redefines**

**1.**

01 A.

02 PIC X(6) VALUE 'SWATHI'.

02 REDEFINES A PIC X(2).

**2.**

01 DATE2 REDEFINES DATE1.

05 DATE2-DD PIC X(2).

05 DATE2-MM PIC X(2).

05 DATE2-YYYY PIC X(4).

PROCEDURE DIVISION.

10000-MAIN-PARA.

MOVE 08092010 TO DATE1

DISPLAY 'DATE =' DATE1

DISPLAY 'YEAR OF DATE=' DATE2-YYYY

STOP RUN.

**fOR RENAMES**

**1.**

01 A.

02 B PIC X(5).

02 C PIC X(5).

02 D.

03 D1 PIC X(2).

03 D2 PIC X(5).

02 E PIC X(6).

66 GROUP1 RENAMES C THRU E.

2.

WORKING-STORAGE SECTION.

01 GROUP1.

05 REC1.

10 ELEMENT1 PIC X(3) VALUE 'WWW'.

10 ELEMENT2 PIC X(9) VALUE 'MAINFRAME'.

05 REC2.

10 ELEMENT3 PIC X(6) VALUE 'WIZARD'.

10 ELEMENT4 PIC X(1) VALUE '.'.

10 ELEMENT5 PIC X(3) VALUE 'COM'.

**66 GROUP2 RENAMES ELEMENT2 THRU ELEMENT5.**

PROCEDURE DIVISION.

10000-MAIN-PARA.

DISPLAY 'RENAME CLAUSE DEPICTED BY ' GROUP2.

STOP RUN.

RENAMES lets you give a name to series of items without needing to have a group item above them. For example,

**01 full-date.  
  05 dt-month pic 99.  
  05 dt-day  pic 99.  
  05 dt-year pic 9999.  
01 short-date renames dt-month thru dt-day.**

Notice that there is no need to define a group item that includes only dt-month and dt-day, because I can use "thru". For "redefines" you would need a separate group item for those 2 items.  
  
Another advantage of "renames" is that you don't need to specify any pic. It just puts together all the pic's that you are renaming to create a new one for the renamed item. In this example, it automatically comes up with pic 9999 for short-date, without you having to put it in the code.  
  
**On the other hand, the disadvantage of "renames" is that you can't change the format at all. That is what "redefines" is for.**

**Q. Date Modification or reformatting use of corresponding**

ANS.

IDENTIFICATION DIVISION.                              
PROGRAM-ID.    FILSTDY.                                
AUTHOR.        INTEL.                                
DATE-WRITTEN.  NOV 08 2006.                            
DATA DIVISION.                                        
WORKING-STORAGE SECTION.                              
01 WS-DATE.   
   05 YYYY             PIC X(4).   
   05 MM               PIC X(2).   
   05 DD               PIC X(2).   
  
01 WS-MM-DD-YYYY .      
   05 MM               PIC X(2).   
   05 FILLER          PIC X(1) VALUE '/'.   
   05 DD               PIC X(2).    
   05 FILLER          PIC X(1) VALUE '/'.   
   05 YYYY             PIC X(4).    
      
PROCEDURE DIVISION.                                    
MAIN-PARA.                                            
    ACCEPT WS-DATE FROM DATE YYYYMMDD.            
    **MOVE CORRESPONDING WS-DATE TO  WS-MM-DD-YYYY**   
    DISPLAY 'MM/DD/YYYY:' WS-MM-DD-YYYY   
                                                      
      GOBACK.

**Q. What is the difference between Copy and Include?**

ANS. cobol compiler can able to understand COPY statement.   
INCLUDE statement normally uses in DB2 programs to include DCLGEN copy books. At that time, pre-compiler (we use this in db2 programs compilation process ) will process these include statements.

|  |
| --- |
|  |
|  |
|  |