COBOL - QUICK GUIDE

COBOL - OVERVIEW

Introduction to COBOL

COBOL is a high-level language. One must understand the way COBOL works. Computers only understand machine code, a binary stream of 0s and 1s. COBOL code must be converted into machine code using a **compiler**. Run the program source through a compiler. The compiler first checks for any syntax errors and then converts it into machine language. The compiler creates a output file which is known as **load module**. This output file contains executable code in the form of 0s and 1s.

Evolution of COBOL

During 1950s, when the businesses were growing in the western part of the world, there was a need to automate various processes for ease of operation and this gave birth to a high-level programming language meant for business data processing.

- In 1959, COBOL was developed by CODASYL (Conference on Data Systems Language).
- The next version, COBOL-61, was released in 1961 with some revisions.
- In 1968, COBOL was approved by ANSI as a standard language for commercial use (COBOL-68).
- It was again revised in 1974 and 1985 to develop subsequent versions named COBOL-74 and COBOL-85 respectively.
- In 2002, Object-Oriented COBOL was released, which could use encapsulated objects as a normal part of COBOL programming.

Importance of COBOL

- COBOL was the first widely used high-level programming language. It is an English-like language which is user friendly. All the instructions can be coded in simple English words.
- COBOL is also used as a self-documenting language.
- COBOL can handle huge data processing.
- COBOL is compatible with its previous versions.
- COBOL has effective error messages and so, resolution of bugs is easier.

Features of COBOL

Standard Language

COBOL is a standard language that can be compiled and executed on machines such as IBM AS/400, personal computers, etc.

Business Oriented

COBOL was designed for business-oriented applications related to financial domain, defense domain, etc. It can handle huge volumes of data because of its advanced file handling capabilities.

Robust Language

COBOL is a robust language as its numerous debugging and testing tools are available for almost all computer platforms.

Structured Language

Logical control structures are available in COBOL which makes it easier to read and modify. COBOL has different divisions, so it is easy to debug.

COBOL - ENVIRONMENT SETUP

Installing COBOL on Windows/Linux

There are many Free Mainframe Emulators available for Windows which can be used to write and learn simple COBOL programs.

One such emulator is Hercules, which can be easily installed on Windows by following a few simple steps as given below:

- Download and install the Hercules emulator, which is available from the Hercules' home site: www.hercules-390.eu
- Once you have installed the package on Windows machine, it will create a folder like **C:/hercules/mvs/cobol**.
- Run the Command Prompt (CMD) and reach the directory C:/hercules/mvs/cobol on CMD.
- The complete guide on various commands to write and execute a JCL and COBOL programs can be found at:

  www.jaymoseley.com/hercules/installmvs/instmvs2.htm

Hercules is an open-source software implementation of the mainframe System/370 and ESA/390 architectures, in addition to the latest 64-bit z/Architecture. Hercules runs under Linux, Windows, Solaris, FreeBSD, and Mac OS X.

A user can connect to a mainframe server in a number of ways such a thin client, dummy terminal, Virtual Client System (VCS), or Virtual Desktop System (VDS). Every valid user is given a login id to enter into the Z/OS interface (TSO/E or ISPF).

Compiling COBOL Programs

In order to execute a COBOL program in batch mode using JCL, the program needs to be compiled, and a load module is created with all the sub-programs. The JCL uses the load module and not the actual program at the time of execution. The load libraries are concatenated and given to the JCL at the time of execution using **JCLLIB** or **STEPLIB**.

There are many mainframe compiler utilities available to compile a COBOL program. Some corporate companies use Change Management tools like **Endevor**, which compiles and stores every version of the program. This is useful in tracking the changes made to the program.

```
//COMPILE   JOB ,CLASS=6,MSGCLASS=X,NOTIFY=&SYSUID
//*
//STEP1     EXEC IGYCRCTL,PARM=RMODE,DYNAM,SSRANGE
//SYSIN     DD DSN=MYDATA.URMI.SOURCES(MYCOBB),DISP=SHR
//SYSLIB    DD DSN=MYDATA.URMI.COPYBOOK(MYCOPY),DISP=SHR
//SYSLMOD   DD DSN=MYDATA.URMI.LOAD(MYCOBB),DISP=SHR
//SYSPRINT  DD SYSOUT=*
//*
```

IGYCRCTL is an IBM COBOL compiler utility. The compiler options are passed using the PARM parameter. In the above example, RMODE instructs the compiler to use relative addressing mode in the program. The COBOL program is passed using the SYSIN parameter. Copybook is the library used by the program in SYSLIB.

Executing COBOL Programs

Give below is a JCL example where the program MYPROG is executed using the input file MYDATA.URMI.INPUT and produces two output files written to the spool.

```
//COBBSTEP  JOB CLASS=6,NOTIFY=&SYSUID
//
//STEP10    EXEC PGM=MYPROG,PARM=ACCT5000
//STEPLIB   DD DSN=MYDATA.URMI.LOADLIB,DISP=SHR
//INPUT1    DD DSN=MYDATA.URMI.INPUT,DISP=SHR
//OUT1      DD SYSOUT=*
//OUT2      DD SYSOUT=*
//SYSIN     DD *
//CUST1     1000
//CUST2     1001
/*
```

The load module of MYPROG is located in MYDATA.URMI.LOADLIB. This is important to note that the above JCL can be used for a non-DB2 COBOL module only.

Executing COBOL-DB2 programs

For running a COBOL-DB2 program, a specialized IBM utility is used in the JCL and the program; DB2 region and required parameters are passed as input to the utility.

The steps followed in running a COBOL-DB2 program are as follows:

- When a COBOL-DB2 program is compiled, a DBRM (Database Request Module) is created along with the load module. The DBRM contains the SQL statements of the COBOL programs with its syntax checked to be correct.
- The DBRM is bound to the DB2 region (environment) in which the COBOL will run. This can be done using the IKJEFT01 utility in a JCL.
- After the bind step, the COBOL-DB2 program is run using IKJEFT01 (again) with the load library and the DBRM library as the input to the JCL.

```
//STEP001  EXEC PGM=IKJEFT01
//*
//STEPLIB  DD DSN=MYDATA.URMI.DBRMLIB,DISP=SHR
//*
//input files
//output files
//SYSPRINT DD SYSOUT=*
//SYSABOUT DD SYSOUT=*
//SYSDBOUT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//DISPLAY  DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
    DSN SYSTEM(SSID)
    RUN PROGRAM(MYCOBB) PLAN(PLANNAME) PARM(parameters to cobol program) -
    LIB('MYDATA.URMI.LOADLIB')
    END
/*
```

In the above example, MYCOBB is the COBOL-DB2 program run using IKJEFT01. Please note that the program name, DB2 Sub-System Id (SSID), and DB2 Plan name are passed within the SYSTSIN DD statement. The DBRM library is specified in the STEPLIB.

Try it Option Online

You really do not need to set up your own environment to start learning COBOL programming language. Reason is very simple, we have already set up COBOL Programming environment online, so that you can compile and execute all the available examples online at the same time when you are doing your theory work. This gives you confidence in what you are reading and to check the result with different options. Feel free to modify any example and execute it online.

Try the following example using our **Try it** option available alongside the code in our website.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
PROCEDURE DIVISION.
DISPLAY 'Hello World'.
STOP RUN.
```

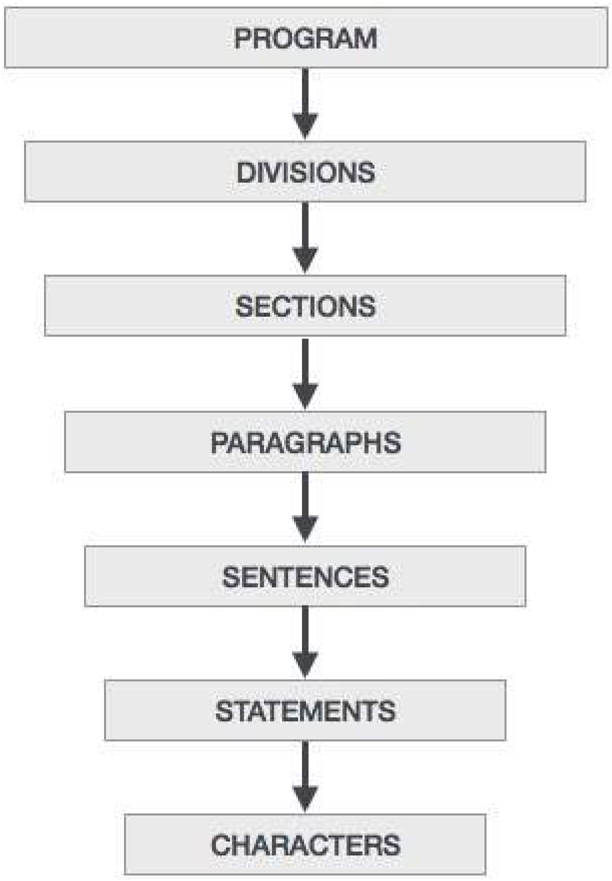
When you compile and execute the above program, it produces the following result:

```
Hello World
```

For some of the examples given in this tutorial, you will find a **Try it** option in our website code selections at the to right corner that will take you to the online compiler. So just make use of it and enjoy your learning. Try it option would work only with the code compatible with OpenCOBOL. The programs that require JCL (Input file, Output file or Parameters) for execution would not run on Tryit option.

COBOL - PROGRAM STRUCTURE

A COBOL program structure consists of divisions as shown in the following image:



A brief introduction of these divisions is given below:

- **Sections** are the logical subdivision of program logic. A section is a collection of paragraphs.
- **Paragraphs** are the subdivision of a section or division. It is either user-defined or a predefined name followed by a period, and consists of zero or more sentences/entries.
- **Sentences** are the combination of one or more statements. Sentences appear only in the Procedure division. A sentence must end with a period.
- **Statements** are meaningful COBOL statement that performs some processing.
- **Characters** are the lowest in the hierarchy and cannot be divisible.

You can co-relate the above-mentioned terms with the COBOL program in the following example:

```
PROCEDURE DIVISION.
A0000-FIRST-PARA SECTION.
FIRST-PARAGRAPH.
ACCEPT WS-ID               - Statement-1  -----|
MOVE '10' TO WS-ID        - Statement-2       |-- Sentence - 1
DISPLAY WS-ID             - Statement-3  -----|
.
```

Divisions

COBOL program consists of four divisions.

Identification Division

It is the first and only mandatory division of every COBOL program. The programmer and the compiler use this division to identify the program. In this Division, PROGRAM-ID is the only mandatory paragraph. PROGRAM-ID specifies the program name that can consist 1 to 30 characters.

Try the following example using the **Try it** option online.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
PROCEDURE DIVISION.
DISPLAY 'Welcome to Tutorialspoint'.
STOP RUN.
```

Given below is the 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:

```
Welcome to Tutorialspoint
```

Environment Division

Environment division is used to specify input and output files to the program. It consists of two sections:

- **Configuration section** provides information about the system on which the program is written and executed. It consists of two paragraphs:

  Source computer : System used to compile the program.

  Object computer : System used to execute the program.

- **Input-Output section** provides information about the files to be used in the program. It consists of two paragraphs:

  File control : Provides information of external data sets used in the program.

  I-O control : Provides information of files used in the program.

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. XXX-ZOS.
OBJECT-COMPUTER. XXX-ZOS.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT FILEN ASSIGN TO DDNAME
ORGANIZATION IS SEQUENTIAL.
```

Data Division

Data division is used to define the variables used in the program. It consists of four sections:

- **File section** is used to define the record structure of the file.
- **Working-Storage section** is used to declare temporary variables and file structures which are used in the program.
- **Local-Storage section** is similar to Working-Storage section. The only difference is that the variables will be allocated and initialized every time program a starts execution.
- **Linkage section** is used to describe the data names that are received from an external program.

COBOL Program

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT FILEN ASSIGN TO INPUT.
       ORGANIZATION IS SEQUENTIAL.
       ACCESS IS SEQUENTIAL.
DATA DIVISION.
FILE SECTION.
FD FILEN
01 NAME PIC A(25).
WORKING-STORAGE SECTION.
01 WS-STUDENT PIC A(30).
01 WS-ID PIC 9(5).
LOCAL-STORAGE SECTION.
01 LS-CLASS PIC 9(3).
LINKAGE SECTION.
01 LS-ID PIC 9(5).
PROCEDURE DIVISION.
DISPLAY 'Executing COBOL program using JCL'.
STOP RUN.
```

The JCL to execute the above COBOL program is as follows:

```
//SAMPLE JOB(TESTJCL,XXXXXX),CLASS=A,MSGCLASS=C
//STEP1 EXEC PGM=HELLO
//INPUT DD DSN=ABC.EFG.XYZ,DISP=SHR
```

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

```
Executing COBOL program using JCL
```

Procedure Division

Procedure division is used to include the logic of the program. It consists of executable statements using variables defined in the data division. In this division, paragraph and section names are user-defined.

There must be at least one statement in the procedure division. The last statement to end the execution in this division is either **STOP RUN** which is used in the calling programs or **EXIT PROGRAM** which is used in the called programs.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-NAME PIC A(30).
```

```
01 WS-ID PIC 9(5) VALUE '12345'.
PROCEDURE DIVISION.
A000-FIRST-PARA.
DISPLAY 'Hello World'.
MOVE 'TutorialsPoint' TO WS-NAME.
DISPLAY "My name is : "WS-NAME.
DISPLAY "My ID is : "WS-ID.
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:

```
Hello World
My name is : TutorialsPoint
My ID is : 12345
```

COBOL - BASIC SYNTAX

Character Set

'Characters' are lowest in the hierarchy and they cannot be divided further. The COBOL Character Set includes 78 characters which are shown below:

Character	Description
A-Z	Alphabets(Upper Case)
a-z	Alphabets (Lower Case)
0-9	Numeric
	Space
+	Plus Sign
-	Minus Sign or Hyphen
*	Asterisk
/	Forward Slash
$	Currency Sign
,	Comma
;	Semicolon
.	Decimal Point or Period
"	Quotation Marks
(	Left Parenthesis
)	Right Parenthesis
>	Greater than
<	Less than
:	Colon
'	Apostrophe

=	Equal Sign

Coding Sheet

The source program of COBOL must be written in a format acceptable to the compilers. COBOL programs are written on COBOL coding sheets. There are 80 characters position on each line of a coding sheet.

Character positions are grouped into the following five fields:

Positions	Field	Description
1-6	Column Numbers	Reserved for line numbers.
7	Indicator	It can have Asterisk (*) indicating comments, Hyphen (-) indicating continuation and Slash (/) indicating form feed.
8-11	Area A	All COBOL divisions, sections, paragraphs and some special entries must begin in Area A.
12-72	Area B	All COBOL statements must begin in area B.
73-80	Identification Area	It can be used as needed by the programmer.

Example

The following example shows a COBOL coding sheet:

```
000100 IDENTIFICATION DIVISION.                                         000100
000200 PROGRAM-ID. HELLO.                                                000101
000250* THIS IS A COMMENT LINE                                           000102
000300 PROCEDURE DIVISION.                                                000103
000350 A000-FIRST-PARA.                                                   000104
000400     DISPLAY “Coding Sheet”.                                       000105
000500 STOP RUN.                                                           000106
```

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:

```
Coding Sheet
```

Character Strings

Character strings are formed by combining individual characters. A character string can be a

- Comment,
- Literal, or
- COBOL word.

All character strings must be ended with **separators**. A separator is used to separate character strings.

Frequently used separators : Space, Comma, Period, Apostrophe, Left/Right Parenthesis, and Quotation mark.

Comment

A comment is a character string that does not affect the execution of a program. It can be any combination of characters.

There are two types of comments:

Comment Line

Comment line can be written in any column. The compiler does not check a comment line for syntax and treats it for documentation.

Comment Entry

Comment entries are those that are included in the optional paragraphs of an Identification Division. They are written in Area B and programmers use it for reference.

The text highlighted in **Bold** are the commented entries in the following example:

```
000100 IDENTIFICATION DIVISION.                                   000100
000150 PROGRAM-ID. HELLO.                                          000101
000200 AUTHOR. TUTORIALSPOINT.                                      000102
000250* THIS IS A COMMENT LINE                                     000103
000300 PROCEDURE DIVISION.                                         000104
000350 A000-FIRST-PARA.                                            000105
000360/ First Para Begins - Documentation Purpose                 000106
000400     DISPLAY “Comment line”.                                 000107
000500 STOP RUN.                                                    000108
```

JCL to execute 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:

```
Comment Line
```

Literal

Literal is a constant that is directly hard coded in a program. In the following example, "Hello World" is a literal.

```
PROCEDURE DIVISION.
DISPLAY 'Hello World'.
```

There are two types of literals as discussed below:

Alphanumeric Literal

Alphanumeric Literals are enclosed in quotes or apostrophe. Length can be up to 160 characters. An apostrophe or a quote can be a part of a literal only if it is paired. Starting and ending of the literal should be same, either apostrophe or quote.

Example

The following example shows valid and invalid Alphanumeric Literals:

```
Valid:
‘This is valid’
"This is valid"
‘This isn’’t invalid’

Invalid:
```

```
'This is invalid"
'This isn't valid'
```

Numeric Literal

A Numeric Literal is a combination of digits from 0 to 9, +, -, or decimal point. Length can be up to 18 characters. Sign cannot be the rightmost character. Decimal point should not appear at the end.

Example

The following example shows valid and invalid Numeric Literals:

```
Valid:
100
+10.9
-1.9

Invalid:
1,00
10.
10.9-
```

COBOL Word

COBOL Word is a character string that can be a reserved word or a user-defined word. Length can be up to 30 characters.

User-Defined

User-defined words are used for naming files, data, records, paragraph names and sections. Alphabets, digits, and hyphens are allowed while forming user-defined words. You cannot use COBOL reserved words.

Reserved Words

Reserved words are predefined words in COBOL. Different types of reserved words that we use frequently are as follows:

- **Keywords** like ADD, ACCEPT, MOVE, etc.
- **Special characters** words like +, -, *, <, <=, etc
- **Figurative constants** are constant values like ZERO, SPACES, etc. All the constant values of figurative constants are mentioned in the following table:

Figurative Constants

Figurative Constants	Description
HIGH-VALUES	One or more characters which will be at the highest position in descending order.
LOW-VALUES	One or more characters have zeros in binary representation.
ZERO/ZEROES	One or more zero depending on the size of the variable.
SPACES	One or more spaces.
QUOTES	Single or double quotes.
ALL literal	Fills the data-item with Literal.

COBOL - DATA TYPES

Data Division is used to define the variables used in a program. To describe data in COBOL, one must understand the following terms:

- Data Name
- Level Number
- Picture Clause
- Value Clause

```
01            TOTAL-STUDENTS            PIC9(5)            VALUE '125'.
|                    |                     |                    |
|                    |                     |                    |
|                    |                     |                    |
Level Number      Data Name           Picture Clause       Value Clause
```

Data Name

Data names must be defined in the Data Division before using them in the Procedure Division. They must have a user-defined name; reserved words cannot be used. Data names gives reference to the memory locations where actual data is stored. They can be elementary or group type.

Example

The following example shows valid and invalid data names:

```
Valid:
WS-NAME
TOTAL-STUDENTS
A100
100B

Invalid:
MOVE            (Reserved Words)
COMPUTE         (Reserved Words)
100             (No Alphabet)
100+B           (+ is not allowed)
```

Level Number

Level number is used to specify the level of data in a record. They are used to differentiate between elementary items and group items. Elementary items can be grouped together to create group items.

Level Number	Description
01	Record description entry
02 to 49	Group and Elementary items
66	Rename Clause items
77	Items which cannot be sub-divided
88	Condition name entry

- **Elementary items** cannot be divided further. Level number, Data name, Picture clause and Value clause (optional) are used to describe an elementary item.

- **Group items** consist of one or more elementary items. Level number, Data name, and Value clause (optional) are used to describe a group item. Group level number is always 01.

Example

The following example shows Group and Elementary items:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-NAME    PIC X(25).                               ---> ELEMENTARY ITEM
01 WS-CLASS   PIC 9(2)  VALUE  '10'.                   ---> ELEMENTARY ITEM

01 WS-ADDRESS.                                          ---> GROUP ITEM
   05 WS-HOUSE-NUMBER    PIC 9(3).                     ---> ELEMENTARY ITEM
   05 WS-STREET          PIC X(15).                    ---> ELEMENTARY ITEM
   05 WS-CITY            PIC X(15).                    ---> ELEMENTARY ITEM
   05 WS-COUNTRY         PIC X(15)  VALUE 'INDIA'.     ---> ELEMENTARY ITEM
```

Picture Clause

Picture clause is used to define the following items:

- **Data type** can be numeric, alphabetic, or alphanumeric. Numeric type consists of only digits 0 to 9. Alphabetic type consists of letters A to Z and spaces. Alphanumeric type consists of digits, letters, and special characters.
- **Sign** can be used with numeric data. It can be either + or .
- **Decimal point position** can be used with numeric data. Assumed position is the position of decimal point and not included in the data.
- **Length** defines the number of bytes used by the data item.

Symbols used in a Picture clause:

Symbol	Description
9	Numeric
A	Alphabetic
X	Alphanumeric
V	Implicit Decimal
S	Sign
P	Assumed Decimal

Example

The following example shows the use of PIC clause:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-NUM1 PIC S9(3)V9(2).
01 WS-NUM2 PIC PPP999.
01 WS-NUM3 PIC S9(3)V9(2) VALUE -123.45.
01 WS-NAME PIC A(6) VALUE 'ABCDEF'.
01 WS-ID PIC X(5) VALUE 'A121$'.
PROCEDURE DIVISION.
DISPLAY "WS-NUM1 : "WS-NUM1.
DISPLAY "WS-NUM2 : "WS-NUM2.
DISPLAY "WS-NUM3 : "WS-NUM3.
DISPLAY "WS-NAME : "WS-NAME.
DISPLAY "WS-ID : "WS-ID.
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-NUM1 : +000.00
WS-NUM2 : .000000
WS-NUM3 : -123.45
WS-NAME : ABCDEF
WS-ID : A121$
```

Value Clause

Value clause is an optional clause which is used to initialize the data items. The values can be numeric literal, alphanumeric literal, or figurative constant. It can be used with both group and elementary items.

Example

The following example shows the use of VALUE clause:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-NUM1 PIC 99V9 VALUE IS 3.5.
01 WS-NAME PIC A(6) VALUE 'ABCD'.
01 WS-ID PIC 99 VALUE ZERO.
PROCEDURE DIVISION.
DISPLAY "WS-NUM1 : "WS-NUM1.
DISPLAY "WS-NAME : "WS-NAME.
DISPLAY "WS-ID   : "WS-ID.
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-NUM1 : 03.5
WS-NAME : ABCD
WS-ID   : 00
```

COBOL - BASIC VERBS

COBOL verbs are used in the procedure division for data processing. A statement always start with a COBOL verb. There are several COBOL verbs with different types of actions.

Input / Output Verbs

Input/Output verbs are used to get data from the user and display the output of COBOL programs. The following two verbs are used for this process:

Accept Verb

Accept verb is used to get data such as date, time, and day from the operating system or directly the from user. If a program is accepting data from the user, then it needs to be passed through JCL. While getting data from the operating system FROM option is included as shown in the following below example:

```
ACCEPT WS-STUDENT-NAME.
ACCEPT WS-DATE FROM SYSTEM-DATE.
```

Display Verb

Display verb is used to display the output of a COBOL program.

```
DISPLAY WS-STUDENT-NAME.
DISPLAY "System date is : " WS-DATE.
```

COBOL PROGRAM

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-STUDENT-NAME PIC X(25).
01 WS-DATE PIC X(10).
PROCEDURE DIVISION.
ACCEPT WS-STUDENT_NAME.
ACCEPT WS-DATE FROM DATE.
DISPLAY "Name :  " WS-STUDENT_NAME.
DISPLAY "Date : " WS-DATE.
STOP RUN.
```

JCL to execute the above COBOL program:

```
//SAMPLE JOB(TESTJCL,XXXXXX),CLASS=A,MSGCLASS=C
//STEP1 EXEC PGM=HELLO
//INPUT DD DSN=PROGRAM.DIRECTORY,DISP=SHR
//SYSIN DD *
TutorialsPoint
/*
```

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

```
Name : TutorialsPoint
Date : 2014-08-30
```

Initialize Verb

Initialize verb is used to initialize a group item or an elementary item. Data names with RENAME clause cannot be initialized. Numeric data items are replaced by ZEROES. Alphanumeric or alphabetic data items are replaced by SPACES. If we include REPLACING term, then data items can be initialized to the given replacing value as shown in the following example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-NAME PIC A(30) VALUE 'ABCDEF'.
01 WS-ID PIC 9(5).
01 WS-ADDRESS.
05 WS-HOUSE-NUMBER PIC 9(3).
05 WS-COUNTRY PIC X(15).
05 WS-PINCODE PIC 9(6) VALUE 123456.
PROCEDURE DIVISION.
A000-FIRST-PARA.
INITIALIZE WS-NAME, WS-ADDRESS.
INITIALIZE WS-ID REPLACING NUMERIC DATA BY 12345.
DISPLAY "My name is   : "WS-NAME.
DISPLAY "My ID is     : "WS-ID.
DISPLAY "Address      : "WS-ADDRESS.
DISPLAY "House Number : "WS-HOUSE-NUMBER.
DISPLAY "Country      : "WS-COUNTRY.
DISPLAY "Pincode      : "WS-PINCODE.
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:

```
My name is   :
My ID is     : 12345
Address      : 000               000000
House Number : 000
Country      :
Pincode      : 000000
```

Move Verb

Move verb is used to copy data from source data to destination data. It can be used on both elementary and group data items. For group data items, MOVE CORRESPONDING/CORR is used. In try it option, MOVE CORR is not working; but on a mainframe server it will work.

For moving data from a string, MOVE(x:l) is used where x is the starting position and l is the length. Data will be truncated if destination data item PIC clause is less than the source data item PIC clause. If the destination data item PIC clause is more than the source data item PIC clause, then ZEROS or SPACES will be added in the extra bytes. The following example makes it clear:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-NUM1 PIC 9(9).
01 WS-NUM2 PIC 9(9).
01 WS-NUM3 PIC 9(5).
01 WS-NUM4 PIC 9(6).
01 WS-ADDRESS.
05 WS-HOUSE-NUMBER PIC 9(3).
05 WS-COUNTRY PIC X(5).
05 WS-PINCODE PIC 9(6).
01 WS-ADDRESS1.
05 WS-HOUSE-NUMBER1 PIC 9(3).
05 WS-COUNTRY1 PIC X(5).
05 WS-PINCODE1 PIC 9(6).
PROCEDURE DIVISION.
A000-FIRST-PARA.
MOVE 123456789 TO WS-NUM1.
MOVE WS-NUM1 TO WS-NUM2 WS-NUM3.
MOVE WS-NUM1(3:6) TO WS-NUM4.
MOVE 123 TO WS-HOUSE-NUMBER.
MOVE 'INDIA' TO WS-COUNTRY.
MOVE 112233 TO WS-PINCODE.
MOVE WS-ADDRESS TO WS-ADDRESS1.
DISPLAY "WS-NUM1     : " WS-NUM1
DISPLAY "WS-NUM2     : " WS-NUM2
DISPLAY "WS-NUM3     : " WS-NUM3
DISPLAY "WS-NUM4     : " WS-NUM4
DISPLAY "WS-ADDRESS  : " WS-ADDRESS
DISPLAY "WS-ADDRESS1 : " WS-ADDRESS1
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-NUM1     : 123456789
WS-NUM2     : 123456789
WS-NUM3     : 56789
WS-NUM4     : 345678
WS-ADDRESS  : 123INDIA112233
WS-ADDRESS1 : 123INDIA112233
```

Legal Moves

The following table gives information about the legal moves:

	Alphabetic	Alphanumeric	Numeric
Alphabetic	Possible	Possible	Not Possible
Alphanumeric	Possible	Possible	Possible
Numeric	Not Possible	Possible	Possible

Add Verb

Add verb is used to add two or more numbers and store the result in the destination operand.

Syntax

give below is the syntax to Add two or more numbers:

```
ADD A B TO C D

ADD A B C TO D GIVING E

ADD CORR WS-GROUP1 TO WS-GROUP2
```

In syntax-1, A, B, C are added and the result is stored in C (C=A+B+C). A, B, D are added and the result is stored in D (D=A+B+D).

In syntax-2, A, B, C, D are added and the result is stored in E (E=A+B+C+D).

In syntax-3, sub-group items with in WS-GROUP1 and WS GROUP2 are the added and result is stored in WS-GROUP2.

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-NUM1 PIC 9(9) VALUE 10 .
01 WS-NUM2 PIC 9(9) VALUE 10.
01 WS-NUM3 PIC 9(9) VALUE 10.
01 WS-NUM4 PIC 9(9) VALUE 10.
01 WS-NUMA PIC 9(9) VALUE 10.
01 WS-NUMB PIC 9(9) VALUE 10.
01 WS-NUMC PIC 9(9) VALUE 10.
01 WS-NUMD PIC 9(9) VALUE 10.
01 WS-NUME PIC 9(9) VALUE 10.
```

```
PROCEDURE DIVISION.
ADD WS-NUM1 WS-NUM2 TO WS-NUM3 WS-NUM4.
ADD WS-NUMA WS-NUMB WS-NUMC TO WS-NUMD GIVING WS-NUME.
DISPLAY "WS-NUM1     : " WS-NUM1
DISPLAY "WS-NUM2     : " WS-NUM2
DISPLAY "WS-NUM3     : " WS-NUM3
DISPLAY "WS-NUM4     : " WS-NUM4
DISPLAY "WS-NUMA     : " WS-NUMA
DISPLAY "WS-NUMB     : " WS-NUMB
DISPLAY "WS-NUMC     : " WS-NUMC
DISPLAY "WS-NUMD     : " WS-NUMD
DISPLAY "WS-NUME     : " WS-NUME
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-NUM1     : 000000010
WS-NUM2     : 000000010
WS-NUM3     : 000000030
WS-NUM4     : 000000030
WS-NUMA     : 000000010
WS-NUMB     : 000000010
WS-NUMC     : 000000010
WS-NUMD     : 000000010
WS-NUME     : 000000040
```

Subtract Verb

Subtract verb is used for subtraction operations.

Syntax

given below is the syntax for Subtract operations:

```
SUBTRACT A B FROM C D

SUBTRACT A B C FROM D GIVING E

SUBTRACT CORR WS-GROUP1 TO WS-GROUP2
```

In syntax-1, A and B are added and subtracted from C. The Result is stored in C (C=C-(A+B)). A and B are added and subtracted from D. The result is stored in D (D=D-(A+B)).

In syntax-2, A, B, C are added and subtracted from D. Result is stored in E (E=D-(A+B+C))

In syntax-3, sub-group items within WS-GROUP1 and WS-GROUP2 are subtracted and the result is stored in WS-GROUP2.

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-NUM1 PIC 9(9) VALUE 10 .
01 WS-NUM2 PIC 9(9) VALUE 10.
01 WS-NUM3 PIC 9(9) VALUE 100.
01 WS-NUM4 PIC 9(9) VALUE 100.
01 WS-NUMA PIC 9(9) VALUE 10.
01 WS-NUMB PIC 9(9) VALUE 10.
01 WS-NUMC PIC 9(9) VALUE 10.
```

```
01 WS-NUMD PIC 9(9) VALUE 100.
01 WS-NUME PIC 9(9) VALUE 10.
PROCEDURE DIVISION.
SUBTRACT WS-NUM1 WS-NUM2 FROM WS-NUM3 WS-NUM4.
SUBTRACT WS-NUMA WS-NUMB WS-NUMC FROM WS-NUMD GIVING WS-NUME.
DISPLAY "WS-NUM1     : " WS-NUM1
DISPLAY "WS-NUM2     : " WS-NUM2
DISPLAY "WS-NUM3     : " WS-NUM3
DISPLAY "WS-NUM4     : " WS-NUM4
DISPLAY "WS-NUMA     : " WS-NUMA
DISPLAY "WS-NUMB     : " WS-NUMB
DISPLAY "WS-NUMC     : " WS-NUMC
DISPLAY "WS-NUMD     : " WS-NUMD
DISPLAY "WS-NUME     : " WS-NUME
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-NUM1     : 000000010
WS-NUM2     : 000000010
WS-NUM3     : 000000080
WS-NUM4     : 000000080
WS-NUMA     : 000000010
WS-NUMB     : 000000010
WS-NUMC     : 000000010
WS-NUMD     : 000000100
WS-NUME     : 000000070
```

Multiply Verb

Multiply verb is used for multiplication operations.

Syntax

Given below is the syntax to multiply two or more numbers:

```
MULTIPLY A BY B C

MULTIPLY A BY B GIVING E
```

In syntax-1, A and B are multipled and the result is stored in B (B=A*B). A and C are multipled and the result is stored in C (C=A*C).

In syntax-2, A and B are multipled and the result is stored in E (E=A*B).

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-NUM1 PIC 9(9) VALUE 10 .
01 WS-NUM2 PIC 9(9) VALUE 10.
01 WS-NUM3 PIC 9(9) VALUE 10.
01 WS-NUMA PIC 9(9) VALUE 10.
01 WS-NUMB PIC 9(9) VALUE 10.
01 WS-NUMC PIC 9(9) VALUE 10.
PROCEDURE DIVISION.
MULTIPLY WS-NUM1 BY WS-NUM2 WS-NUM3.
MULTIPLY WS-NUMA BY WS-NUMB GIVING WS-NUMC.
DISPLAY "WS-NUM1     : " WS-NUM1
DISPLAY "WS-NUM2      : " WS-NUM2
DISPLAY "WS-NUM3      : " WS-NUM3
DISPLAY "WS-NUMA      : " WS-NUMA
DISPLAY "WS-NUMB      : " WS-NUMB
DISPLAY "WS-NUMC      : " WS-NUMC
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-NUM1      : 000000010
WS-NUM2      : 000000100
WS-NUM3      : 000000100
WS-NUMA      : 000000010
WS-NUMB      : 000000010
WS-NUMC      : 000000100
```

Divide Verb

Divide verb is used for division operations.

Syntax

Given below following is the syntax for division operations:

```
DIVIDE A INTO B

DIVIDE A BY B GIVING C REMAINDER R
```

In syntax-1, B is divided by A and the result is stored in B (B=B/A).

In syntax-2, A is divided by B and the result is stored in C (C=A/B) and remainder is stored in R.

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-NUM1 PIC 9(9) VALUE 5.
01 WS-NUM2 PIC 9(9) VALUE 250.
01 WS-NUMA PIC 9(9) VALUE 100.
01 WS-NUMB PIC 9(9) VALUE 15.
01 WS-NUMC PIC 9(9).
01 WS-REM PIC 9(9).
PROCEDURE DIVISION.
DIVIDE WS-NUM1 INTO WS-NUM2.
DIVIDE WS-NUMA BY WS-NUMB GIVING WS-NUMC REMAINDER WS-REM.
DISPLAY "WS-NUM1      : " WS-NUM1
DISPLAY "WS-NUM2      : " WS-NUM2
DISPLAY "WS-NUMA      : " WS-NUMA
DISPLAY "WS-NUMB      : " WS-NUMB
DISPLAY "WS-NUMC      : " WS-NUMC
DISPLAY "WS-REM       : " WS-REM
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-NUM1     : 000000005
WS-NUM2     : 000000050
WS-NUMA     : 000000100
WS-NUMB     : 000000015
WS-NUMC     : 000000006
WS-REM      : 000000010
```

Compute Statement

Compute statement is used to write arithmetic expressions in COBOL. This is a replacement for Add, Subtract, Multiply, and Divide.

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-NUM1 PIC 9(9) VALUE 10 .
01 WS-NUM2 PIC 9(9) VALUE 10.
01 WS-NUM3 PIC 9(9) VALUE 10.
01 WS-NUMA PIC 9(9) VALUE 50.
01 WS-NUMB PIC 9(9) VALUE 10.
01 WS-NUMC PIC 9(9).
PROCEDURE DIVISION.
COMPUTE WS-NUMC= (WS-NUM1 * WS-NUM2) - (WS-NUMA / WS-NUMB) + WS-NUM3.
DISPLAY "WS-NUM1     : " WS-NUM1
DISPLAY "WS-NUM2     : " WS-NUM2
DISPLAY "WS-NUM3     : " WS-NUM3
DISPLAY "WS-NUMA     : " WS-NUMA
DISPLAY "WS-NUMB     : " WS-NUMB
DISPLAY "WS-NUMC     : " WS-NUMC
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-NUM1     : 000000010
WS-NUM2     : 000000010
WS-NUM3     : 000000010
WS-NUMA     : 000000050
WS-NUMB     : 000000010
WS-NUMC     : 000000105
```

COBOL - DATA LAYOUT

COBOL layout is the description of use of each field and the values present in it. Following are the data description entries used in COBOL:

- Redefines Clause
- Renames Clause
- Usage Clause
- Copybooks

Redefines Clause

Redeifnes clause is used to define a storage with different data description. If one or more data items are not used simultaneously, then the same storage can be utilized for another data item. So the same storage can be referred with different data items.

Syntax

Following is the syntax for Redefines clause:

```
01 WS-OLD PIC X(10).
01 WS-NEW1 REDEFINES WS-OLD PIC 9(8).
01 WS-NEW2 REDEFINES WS-OLD PIC A(10).
```

Following are the details of the used parameters:

- WS-OLD is Redefined Item
- WS-NEW1 and WS-NEW2 are Redefining Item

Level numbers of redefined item and redefining item must be the same and it cannot be 66 or 88 level number. Do not use VALUE clause with a redefining item. In File Section, do not use a redefines clause with 01 level number. Redefines definition must be the next data description you want to redefine. A redefining item will always have the same value as a redefined item.

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-DESCRIPTION.
05 WS-DATE1 VALUE '20140831'.
10 WS-YEAR PIC X(4).
10 WS-MONTH PIC X(2).
10 WS-DATE PIC X(2).
05 WS-DATE2 REDEFINES WS-DATE1 PIC 9(8).
PROCEDURE DIVISION.
DISPLAY "WS-DATE1 : "WS-DATE1.
DISPLAY "WS-DATE2 : "WS-DATE2.
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-DATE1 : 20140831
WS-DATE2 : 20140831
```

Renames Clause

Renames clause is used to give different names to existing data items. It is used to re-group the data names and give a new name to them. The new data names can rename across groups or elementary items. Level number 66 is reserved for renames.

Syntax

Following is the syntax for Renames clause:

```
01 WS-OLD.
10 WS-A PIC 9(12).
10 WS-B PIC X(20).
10 WS-C PIC A(25).
10 WS-D PIC X(12).
```

```
66 WS-NEW RENAMES WS-A THRU WS-C.
```

Renaming is possible at same level only. In the above example ,WS-A, WS-B, and WS-C are at the same level. Renames definition must be the next data description you want to rename. Do not use Renames with 01,77, or 66 level number. The data names used for renames must come in sequence. Data items with occur clause cannot be renamed.

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-DESCRIPTION.
05 WS-NUM.
10 WS-NUM1 PIC 9(2) VALUE 20.
10 WS-NUM2 PIC 9(2) VALUE 56.
05 WS-CHAR.
10 WS-CHAR1 PIC X(2) VALUE 'AA'.
10 WS-CHAR2 PIC X(2) VALUE 'BB'.
66 WS-RENAME RENAMES WS-NUM2 THRU WS-CHAR2.
PROCEDURE DIVISION.
DISPLAY "WS-RENAME : " WS-RENAME.
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-RENAME : 56AABB
```

Usage Clause

Usage clause specifies the operating system in which the format data is stored. It can not be used with level numbers 66 or 88. If usage clause is specified on a group, then all the elementary items will have the same usage clause.The different options available with Usage clause are as follows:

Display

Data item is stored in ASCII format and each character will take 1 byte. It is default usage.

The following example to calculates the number of bytes required:

```
01 WS-NUM PIC S9(5)V9(3) USAGE IS DISPLAY.
It requires 8 bytes as sign and decimal doesn't require any byte.

01 WS-NUM PIC 9(5) USAGE IS DISPLAY.
It requires 5 bytes as sign.
```

COMPUTATIONAL / COMP

Data item is stored in binary format. Here data items must be integer.

The following example calculates the number of bytes required:

```
01 WS-NUM PIC S9(n) USAGE IS COMP.

If 'n' = 1 to 4, it takes 2 bytes.
If 'n' = 5 to 9, it takes 4 bytes.
If 'n' = 10 to 18, it takes 8 bytes.
```

COMP-1

Data item is similar to Real or Float and is represented as a single precision floating point number. Internally data is stored in hexadecimal format. COMP-1 does not accept PIC clause. Here 1 word is equal to 4 bytes.

COMP-2

Data item is similar to Long or Double and is represented as double precision floating point number. Internally data is stored in hexadecimal format. COMP-2 does not specify PIC clause. Here 2 word is equal to 8 bytes.

COMP-3

Data item is stores in pack decimal format. Each digit occupies half a byte (1 nibble) and the sign is stored at the right most nibble.

The following example calculates the number of bytes required:

```
01 WS-NUM PIC 9(n) USAGE IS COMP.
Number of bytes = n/2 (If n is even)
Number of bytes = n/2 + 1(If n is odd, consider only integer part)

01 WS-NUM PIC 9(4) USAGE IS COMP-3 VALUE 21.
It requires 2 bytes of storage as each digit occupies half a byte.

01 WS-NUM PIC 9(5) USAGE IS COMP-3 VALUE 21.
It requires 3 bytes of storage as each digit occupies half a byte.
```

Copybooks

A COBOL copybook is a selection of code that defines data structures. If a particular data structure is used in many programs then instead of writing the same data structure again, we can use copybooks. We use the COPY statement to include a copybook in a program. COPY statement is used in the Working-Storage Section.

the following example to include copybook inside COBOL program:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
COPY ABC.
```

Here ABC is the copybook name. The following data items in ABC copybook can be used inside a program.

```
01 WS-DESCRIPTION.
 05 WS-NUM.
  10 WS-NUM1 PIC 9(2) VALUE 20.
  10 WS-NUM2 PIC 9(2) VALUE 56.
 05 WS-CHAR.
  10 WS-CHAR1 PIC X(2) VALUE 'AA'.
  10 WS-CHAR2 PIC X(2) VALUE 'BB'.
```

COBOL - CONDITION STATEMENTS

Conditional statements are used to change the execution flow depending on certain conditions specified by the programmer. Conditional statements will always evaluate to true or false. Conditions are used in IF, Evaluate and Perform statements. The different types of conditions are as follows:

- IF Condition Statement
- Relation Condition

- Sign Condition
- Class Condition
- Condition-Name Condition
- Negated Condition
- Combined Condition

IF Condition Statement

IF statement checks for conditions. If a condition is true the IF block is executed; and if the condition is false, the ELSE block is executed.

END-IF is used to end the IF block. To end the IF block, a period can be used instead of END-IF. But it is always preferable to use END-IF for multiple IF blocks.

Nested-IF : IF blocks appearing inside another IF block. There is no limit to the depth of nested IF statements.

Syntax

Following is the syntax of IF condition statements:

```
IF [condition] THEN
[COBOL statements]
ELSE
[COBOL statements]
END-IF.
```

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-NUM1 PIC 9(9).
01 WS-NUM2 PIC 9(9).
01 WS-NUM3 PIC 9(5).
01 WS-NUM4 PIC 9(6).
PROCEDURE DIVISION.
A000-FIRST-PARA.
MOVE 25 TO WS-NUM1 WS-NUM3.
MOVE 15 TO WS-NUM2 WS-NUM4.
IF WS-NUM1 > WS-NUM2 THEN
DISPLAY 'IN LOOP 1 - IF BLOCK'
   IF WS-NUM3 = WS-NUM4 THEN
   DISPLAY 'IN LOOP 2 - IF BLOCK'
   ELSE
   DISPLAY 'IN LOOP 2 - ELSE BLOCK'
   END-IF
ELSE
DISPLAY 'IN LOOP 1 - ELSE BLOCK'
END-IF.
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:

```
IN LOOP 1 - IF BLOCK
IN LOOP 2 - ELSE BLOCK
```

Relation Condition

Relation condition compares two operands, either of which can be an identifier, literal, or arithmetic expression. Algebraic comparison of numeric fields is done regardless of size and usage clause.

For non-numeric operands

If two non-numeric operands of equal size are compared then the characters are compared from left with the corresponding positions till the end is reached. The operand containing greater number of characters is declared greater.

If two non-numeric operands of unequal size are compared, then the shorter data item is appended with spaces at the end till the size of the operands becomes equal and then compared according to the rules mentioned in the previous point.

Syntax

Given below following is the syntax of Relation condition statements:

```
[Data Name/Arithmetic Operation]

          [IS] [NOT]

[Equal to (=),Greater than (>), Less than (<),
 Greater than or Equal (>=), Less than or equal (<=) ]

[Data Name/Arithmetic Operation]
```

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-NUM1 PIC 9(9).
01 WS-NUM2 PIC 9(9).
PROCEDURE DIVISION.
A000-FIRST-PARA.
MOVE 25 TO WS-NUM1.
MOVE 15 TO WS-NUM2.
IF WS-NUM1 IS GREATER THAN OR EQUAL TO WS-NUM2 THEN
DISPLAY 'WS-NUM1 IS GREATER THAN WS-NUM2'
ELSE
DISPLAY 'WS-NUM1 IS LESS THAN WS-NUM2'
END-IF.
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-NUM1 IS GREATER THAN WS-NUM2
```

Sign Condition

Sign condition is used to check the sign of a numeric operand. It determines whether a given numeric value is greater than, less than, or equal to ZERO.

Syntax

Following is the syntax of Sign condition statements:

```
[Data Name/Arithmetic Operation]

          [IS] [NOT]

[Positive, Negative or Zero]

[Data Name/Arithmetic Operation]
```

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-NUM1 PIC S9(9) VALUE -1234.
01 WS-NUM2 PIC S9(9) VALUE 123456.
PROCEDURE DIVISION.
A000-FIRST-PARA.
IF WS-NUM1 IS POSITIVE THEN
DISPLAY 'WS-NUM1 IS POSITIVE'.
IF WS-NUM1 IS NEGATIVE THEN
DISPLAY 'WS-NUM1 IS NEGATIVE'.
IF WS-NUM1 IS ZERO THEN
DISPLAY 'WS-NUM1 IS ZERO'.
IF WS-NUM2 IS POSITIVE THEN
DISPLAY 'WS-NUM2 IS POSITIVE'.
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-NUM1 IS NEGATIVE
WS-NUM2 IS POSITIVE
```

Class Condition

Class condition is used to check if an operand contains only alphabets or numeric data. Spaces are considered in ALPHABETIC, ALPHABETIC-LOWER, and ALPHABETIC-UPPER.

Syntax

Following is the syntax of Class condition statements:

```
[Data Name/Arithmetic Operation>]

          [IS] [NOT]

[NUMERIC, ALPHABETIC, ALPHABETIC-LOWER, ALPHABETIC-UPPER]

[Data Name/Arithmetic Operation]
```

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-NUM1 PIC X(9) VALUE 'ABCD '.
```

```
01 WS-NUM2 PIC 9(9) VALUE 123456789.
PROCEDURE DIVISION.
A000-FIRST-PARA.
IF WS-NUM1 IS ALPHABETIC THEN
DISPLAY 'WS-NUM1 IS ALPHABETIC'.
IF WS-NUM1 IS NUMERIC THEN
DISPLAY 'WS-NUM1 IS NUMERIC'.
IF WS-NUM2 IS NUMERIC THEN
DISPLAY 'WS-NUM1 IS NUMERIC'.
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-NUM1 IS ALPHABETIC
WS-NUM1 IS NUMERIC
```

Condition-name Condition

A condition name is a user-defined name. It contains a set of values specified by the user. It behaves like Boolean variables. They are defined with level number 88. It will not have a PIC clause.

Syntax

Following is the syntax of user-defined condition statements:

```
88 [Condition-Name] VALUE [IS, ARE] [LITERAL] [THRU LITERAL].
```

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-NUM PIC 9(3).
88 PASS VALUES ARE 041 THRU 100.
88 FAIL VALUES ARE 000 THRU 40.
PROCEDURE DIVISION.
A000-FIRST-PARA.
MOVE 65 TO WS-NUM.
IF PASS
DISPLAY 'Passed with ' WS-NUM ' marks'.
IF FAIL
DISPLAY 'FAILED with ' WS-NUM 'marks'.
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:

```
Passed with 065 marks
```

Negated Condition

Negated condition is given by using the NOT keyword. If a condition is true and we have given NOT in front of it, then its final value will be false.

Syntax

Following is the syntax for Negated condition statements:

```
IF NOT [CONDITION]
COBOL Statements
END-IF.
```

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-NUM1 PIC 9(2) VALUE 20.
01 WS-NUM2 PIC 9(9) VALUE 25.
PROCEDURE DIVISION.
A000-FIRST-PARA.
IF NOT WS-NUM1 IS LESS THAN WS-NUM2 THEN
DISPLAY 'IF-BLOCK'
ELSE
DISPLAY 'ELSE-BLOCK'
END-IF.
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:

```
ELSE-BLOCK
```

Combined Condition

A combined condition contains two or more conditions connected using logical operators AND or OR.

Syntax

Following is the syntax of combined condition statements:

```
IF [CONDITION] AND [CONDITION]
COBOL Statements
END-IF.
```

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-NUM1 PIC 9(2) VALUE 20.
01 WS-NUM2 PIC 9(2) VALUE 25.
01 WS-NUM3 PIC 9(2) VALUE 20.
PROCEDURE DIVISION.
A000-FIRST-PARA.
IF WS-NUM1 IS LESS THAN WS-NUM2 AND WS-NUM1=WS-NUM3 THEN
DISPLAY 'Both condition OK'
ELSE
DISPLAY 'Error'
```

```
END-IF.
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:

```
Both condition OK
```

Evaluate Verb

Evaluate verb is a replacement of series of IF-ELSE statement. It can be used to evaluate more than one condition. It is similar to SWITCH statement in C programs.

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-A PIC 9 VALUE 0.
PROCEDURE DIVISION.
MOVE 3 TO WS-A.
EVALUATE TRUE
WHEN WS-A > 2
DISPLAY 'WS-A GREATER THAN 2'
WHEN WS-A < 0
DISPLAY 'WS-A LESS THAN 0'
WHEN OTHER
DISPLAY 'INVALID VALUE OF WS-A'
END-EVALUATE.
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-A GREATER THAN 2
```

COBOL - LOOP STATEMENTS

There are some tasks that need to be done over and over again like reading each record of a file till its end. The loop statements used in COBOL are:

- Perform Thru
- Perform Until
- Perform Times
- Perform Varying

Perform Thru

Perform Thru is used to execute a series of paragraph by giving the first and last paragraph names in the sequence. After executing the last paragraph control is returned back.

In-line Perform

Statements inside the PERFORM will be executed till END-PERFORM is reached.

Syntax

Following is the syntax of In-line perform:

```
PERFORM
DISPLAY 'HELLO WORLD'
END-PERFORM.
```

Out-of-line Perform

Here, a statement is executed in one paragraph and then the control is transferred to other paragraph or section.

Syntax

Following is the syntax of Out-of-line perform:

```
PERFORM PARAGRAPH1 THRU PARAGRAPH2
```

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
PROCEDURE DIVISION.
A-PARA.
PERFORM DISPLAY 'IN A-PARA'
END-PERFORM.
PERFORM C-PARA THRU E-PARA.
B-PARA.
DISPLAY 'IN B-PARA'.
STOP RUN.
C-PARA.
DISPLAY 'IN C-PARA'.
D-PARA.
DISPLAY 'IN D-PARA'.
E-PARA.
DISPLAY 'IN E-PARA'.
```

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:

```
IN A-PARA
IN C-PARA
IN D-PARA
IN E-PARA
IN B-PARA
```

Perform Until

In perform until', a paragraph is executed until the given condition becomes true. With test before is the default condition and it indicates that the condition is checked before the execution of statements in a paragraph.

Syntax

Following is the syntax of perform until:

```
PERFORM A-PARA UNTIL COUNT=5

PERFORM A-PARA WITH TEST BEFORE UNTIL COUNT=5

PERFORM A-PARA WITH TEST AFTER UNTIL COUNT=5
```

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-CNT PIC 9(1) VALUE 0.
PROCEDURE DIVISION.
A-PARA.
PERFORM B-PARA WITH TEST AFTER UNTIL WS-CNT>3.
STOP RUN.
B-PARA.
DISPLAY 'WS-CNT : 'WS-CNT.
ADD 1 TO WS-CNT.
```

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-CNT : 0
WS-CNT : 1
WS-CNT : 2
WS-CNT : 3
```

Perform Times

In 'perform times', a paragraph will be executed the number of times specified.

Syntax

Following is the syntax of perform times:

```
PERFORM A-PARA 5 TIMES.
```

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
PROCEDURE DIVISION.
A-PARA.
PERFORM B-PARA 3 TIMES.
STOP RUN.
B-PARA.
DISPLAY 'IN B-PARA'.
```

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:

```
IN B-PARA
IN B-PARA
IN B-PARA
```

Perform Varying

In perform varying, a paragraph will be executed till the condition in Until phrase becomes true.

Syntax

Following is the syntax of perform varying:

```
PERFORM A-PARA VARYING A FROM 1 BY 1 UNTIL A=5.
```

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-A PIC 9 VALUE 0.
PROCEDURE DIVISION.
A-PARA.
PERFORM B-PARA VARYING WS-A FROM 1 BY 1 UNTIL WS-A=5
STOP RUN.
B-PARA.
DISPLAY 'IN B-PARA ' WS-A.
```

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:

```
IN B-PARA 1
IN B-PARA 2
IN B-PARA 3
IN B-PARA 4
```

GO TO Statement

GO TO statement is used to change the flow of execution in a program. In GO TO statements transfer goes only in the forward direction. It is used to exit a paragraph. The different types of GO TO statements used are as follows:

Unconditional GO TO

```
GO TO para-name.
```

Conditional GO TO

```
GO TO para-1 para-2 para-3 DEPENDING ON x.
```

If 'x' is equal to 1, then the control will be transferred to first paragraph and if 'x' is equal to 2, then the control will be transferred to the second paragraph, and so on.

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-A PIC 9 VALUE 2.
PROCEDURE DIVISION.
A-PARA.
```

```
DISPLAY 'IN A-PARA'
GO TO B-PARA.
B-PARA.
DISPLAY 'IN B-PARA '.
GO TO C-PARA D-PARA DEPENDING ON WS-A.
C-PARA.
DISPLAY 'IN C-PARA '.
D-PARA.
DISPLAY 'IN D-PARA '.
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:

```
IN A-PARA
IN B-PARA
IN D-PARA
```

COBOL - STRING HANDLING

String handling statements in COBOL are used to do multiple functional operations on strings. Following are the string handling statements:

- Inspect
- String
- Unstring

Inspect

Inspect verb is used to count or replace the characters in a string. String operations can be performed on alphanumeric, numeric, or alphabetic values. Inspect operations are performed from left to right. The options used for the string operations are as follows:

Tallying

Tallying option is used to count the string characters.

Syntax

Following is the syntax of Tallying option:

```
INSPECT input-string
TALLYING output-count FOR ALL CHARACTERS
```

The parameters used are:

- input-string : The string whose characters are to be counted.
- output-count : Data item to hold the count of characters.

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-CNT1 PIC 9(2) VALUE 0.
01 WS-CNT2 PIC 9(2) VALUE 0.
01 WS-STRING PIC X(15) VALUE 'ABCDACDADEAAAFF'.
```

```
PROCEDURE DIVISION.
INSPECT WS-STRING TALLYING WS-CNT1 FOR ALL CHARACTERS.
DISPLAY "WS-CNT1 : "WS-CNT1.
INSPECT WS-STRING TALLYING WS-CNT2 FOR ALL 'A'.
DISPLAY "WS-CNT2 : "WS-CNT2
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-CNT1 : 15
WS-CNT2 : 06
```

Replacing

Replacing option is used to replace the string characters.

Syntax

Following is the syntax of Replacing option:

```
INSPECT input-string REPLACING ALL char1 BY char2.
```

The parameter used is:

- input-string : The string whose characters are to be replaced from char1 to char2.

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-STRING PIC X(15) VALUE 'ABCDACDADEAAAFF'.
PROCEDURE DIVISION.
DISPLAY "OLD STRING : "WS-STRING.
INSPECT WS-STRING REPLACING ALL 'A' BY 'X'.
DISPLAY "NEW STRING : "WS-STRING.
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:

```
OLD STRING : ABCDACDADEAAAFF
NEW STRING : XBCDXCDXDEXXXFF
```

String

String verb is used to concatenate the strings. Using STRING statement, two or more strings of characters can be combined to form a longer string. 'Delimited By' clause is compulsory.

Syntax

Following is the syntax of String verb:

```
STRING ws-string1 DELIMITED BY SPACE
ws-string2 DELIMITED BY SIZE
INTO ws-destination-string
WITH POINTER ws-count
ON OVERFLOW DISPLAY message1
NOT ON OVERFLOW DISPLAY message2
END-STRING.
```

Following are the details of the used parameters:

- ws-string1 and ws-string2 : Input strings to be concatenated
- ws-string : Output string
- ws-count : Used to count the length of new concatenated string
- Delimited specifies the end of string
- Pointer and Overflow are optional

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-STRING PIC A(30).
01 WS-STR1 PIC A(15) VALUE 'Tutorialspoint'.
01 WS-STR2 PIC A(7) VALUE 'Welcome'.
01 WS-STR3 PIC A(7) VALUE 'To AND'.
01 WS-COUNT PIC 99 VALUE 1.
PROCEDURE DIVISION.
STRING WS-STR2 DELIMITED BY SIZE
WS-STR3 DELIMITED BY SPACE
WS-STR1 DELIMITED BY SIZE
INTO WS-STRING
WITH POINTER WS-COUNT
ON OVERFLOW DISPLAY 'OVERFLOW!'
END-STRING.
DISPLAY 'WS-STRING : 'WS-STRING.
DISPLAY 'WS-COUNT : 'WS-COUNT.
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-STRING : WelcomeToTutorialspoint
WS-COUNT : 25
```

Unstring

Unstring verb is used to split one string into multiple sub-strings. Delimited By clause is compulsory.

Syntax

Following is the syntax of Unstring verb:

```
UNSTRING ws-string DELIMITED BY SPACE
INTO ws-str1, ws-str2
WITH POINTER ws-count
ON OVERFLOW DISPLAY message
```

```
NOT ON OVERFLOW DISPLAY message
END-UNSTRING.
```

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-STRING PIC A(30) VALUE 'WELCOME TO TUTORIALSPOINT'.
01 WS-STR1 PIC A(7).
01 WS-STR2 PIC A(2).
01 WS-STR3 PIC A(15).
01 WS-COUNT PIC 99 VALUE 1.
PROCEDURE DIVISION.
UNSTRING WS-STRING DELIMITED BY SPACE
INTO WS-STR1, WS-STR2, WS-STR3
END-UNSTRING.
DISPLAY 'WS-STR1 : 'WS-STR1.
DISPLAY 'WS-STR2 : 'WS-STR2.
DISPLAY 'WS-STR3 : 'WS-STR3.
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-STR1 : WELCOME
WS-STR2 : TO
WS-STR3 : TUTORIALSPOINT
```

COBOL - TABLE PROCESSING

Arrays in COBOL are known as tables. An array is a linear data structure and is 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. WS-TABLE 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

```
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

```
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 valus can range from 1 to the number of times the table occurs. Subscript can be any positive number. It does not require any declaration in data division. It is automatically created with occurs clause.

Example

```
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

```
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

```
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

```
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 determines in which half the searched element is present. This process repeats till the element is found or the end is reached.

Example

```
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
```

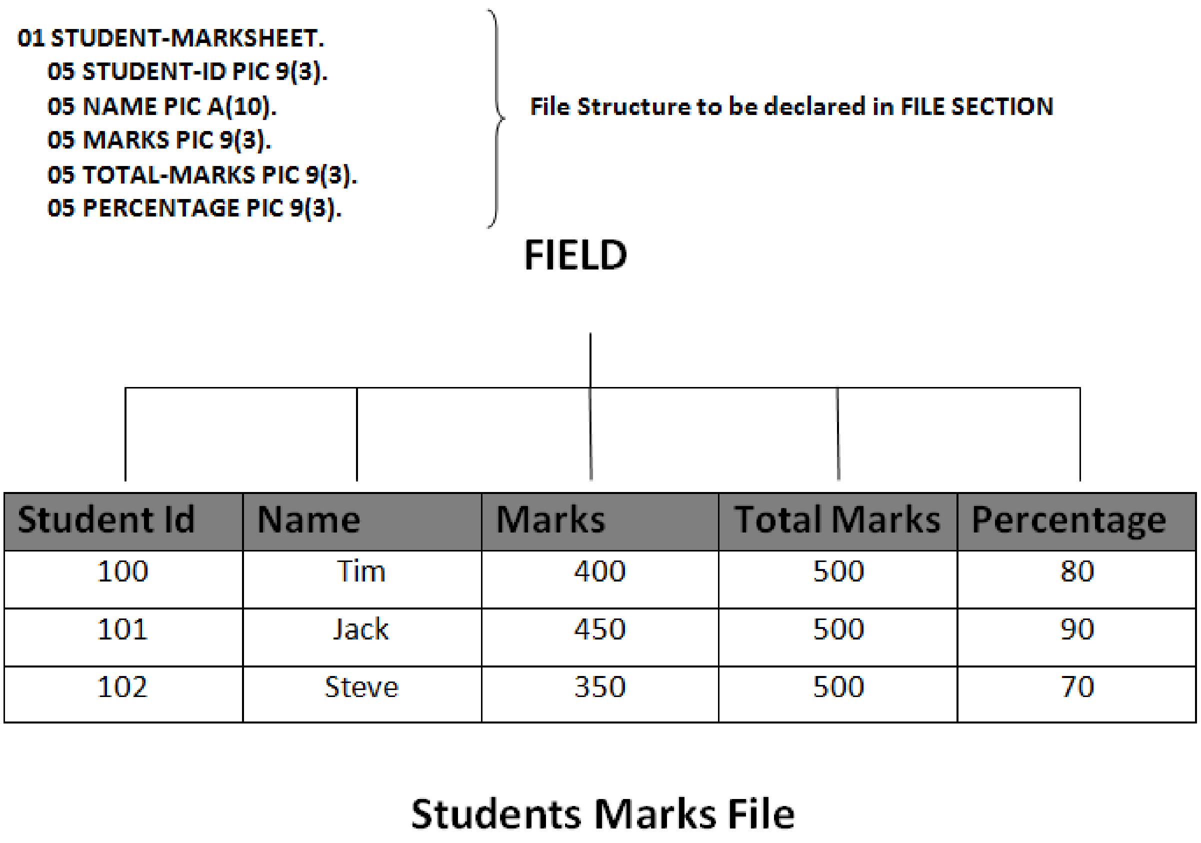
COBOL - FILE HANDLING

The concept of files in COBOL is different from that in C/C++. While learning the basics of 'File' in COBOL, the concepts of both languages should not be co-related. Simple text files cannot be used in COBOL, instead **PS (Physical Sequential)** and **VSAM** files are used. PS files will be discussed in this module.

To understand file handling in COBOL, one must know the basic terms. These terms only serve to understand the fundamentals of file handling. Further in-depth terminology would be discussed in the chapter 'File Handling Verbs'. Following are the basic terms:

- Field
- Record
- Physical Record
- Logical Record
- File

The following example helps in understanding these terms:



```
01 STUDENT-MARKSHEET.
   05 STUDENT-ID PIC 9(3).
   05 NAME PIC A(10).
   05 MARKS PIC 9(3).
   05 TOTAL-MARKS PIC 9(3).
   05 PERCENTAGE PIC 9(3).
```

Student Id	Name	Marks	Total Marks	Percentage
100	Tim	400	500	80
101	Jack	450	500	90
102	Steve	350	500	70

Field

Field is used to indicate the data stored about an element. It represents a single element as shown in the above example such as student id, name, marks, total marks, and percentage. The number of characters in any field is known as field size, for example student name can have 10 characters.

Fields can have the following attributes:

- **Primary keys** are those fields that are unique to each record and are used to identify a particular record. For example, in students marks file, each student will be having a unique student id which forms the primary key.
- **Secondary keys** are unique or non-unique fields that are used to search for related data. For example, in students marks file, full name of student can be used as secondary key when student id is not known.
- **Descriptors** fields are used to describe an entity. For example, in students marks file, marks and percentage fields that add meaning to the record are known descriptors.

Record

Record is a collection of fields that is used to describe an entity. One or more fields together form a record. For example, in students marks file, student id, name, marks, total marks and percentage form one record. The cumulative size of all the fields in a record is known the as record size. The records present in a file may be of fixed length or variable length.

Physical Record

Physical record is the information that exists on the external device. It is also known as a block.

Logical Record

Logical record is the information used by the program. In COBOL programs, only one record can be handled at any point of time and it is called as logical record.

File

File is a collection of related records. For example, the students marks file consists of records of all the students.

COBOL - FILE ORGANIZATION

File organization indicates how the records are organized in a file. There are different types of organizations for files so as to increase their efficiency of accessing the records. Following are the types of file organization schemes:

- Sequential file organization
- Indexed sequential file organization
- Relative file organization

The syntaxes, in this module, mentioned along with their respective terms only refer to their usage in the program. The complete programs using these syntaxes would be discussed in the chapter 'File handling Verbs'.

Sequential File Organization

A sequential file consists of records that are stored and accessed in sequential order. Following are the key attributes of sequential file organization:

- Records can be read in sequential order. For reading the 10th record, all the previous 9 records should be read.
- Records are written in sequential order. A new records cannot be inserted in between. A new record is always inserted at the end of the file.
- After placing a record into a sequential file, it is not possible to delete, shorten, or lengthen a record.
- Order of the records, once inserted, can never be changed.

- Updation of record is possible. A record can be overwritten, if the new record length is same as the` old record length.

- Sequential output files are good option for printing.

Syntax

Following is the syntax of sequential file organization:

```
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT file-name ASSIGN TO dd-name-jcl
  ORGANIZATION IS SEQUENTIAL
```

Indexed Sequential File Organization

An indexed sequential file consists of records that can be accessed sequentially. Direct access is also possible. It consists of two parts:

- **Data File** contains records in sequential scheme.

- **Index File** contains the primary key and its address in the data file.

Following are the key attributes of sequential file organization:

- Records can be read in sequential order just like in sequential file organization.

- Records can be accessed randomly if the primary key is known. Index file is used to get the address of a record and then the record is fetched from the data file.

- Sorted index is maintained in this file system which relates the key value to the position of the record in the file.

- Alternate index can also be created to fetch the records.

Syntax

Following is the syntax of indexed sequential file organization:

```
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT file-name ASSIGN TO dd-name-jcl
  ORGANIZATION IS INDEXED
  RECORD KEY IS primary-key
  ALTERNATE RECORD KEY IS rec-key
```

Relative File Organization

A relative file consists of records ordered by their **relative address**. Following are the key attributes of relative file organization:

- Records can be read in sequential order just like in sequential and indexed file organization.

- Records can be accessed using relative key. Relative key represents the record’s location relative to the address of the start of the file.

- Records can be inserted using relative key. Relative address is calculated using relative key.

- Relative file provides the fastest access to the records.

- The main disadvantage of this file system is that if some intermediate records are missing, they will also occupy space.

Syntax

Following is the syntax of relative file organization:

```
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT file-name ASSIGN TO dd-name-jcl
  ORGANIZATION IS INDEXED
  RELATIVE KEY IS rec-key
```

COBOL - FILE ACCESS MODE

Till now, file organization schemes have been discussed. For each file organization scheme, different access modes can be used. Following are the types of file access modes:

- Sequential Access
- Random Access
- Dynamic Access

The syntaxes, in this module, mentioned along with their respective terms only refer to their usage in the program. The complete programs using these syntaxes would be discussed in the next chapter.

Sequential Access

When the access mode is sequential, the method of record retrieval changes as per the selected file organization.

- **For sequential files**, records are accessed in the same order in which they were inserted.
- **For indexed files**, the parameter used to fetch the records are the record key values.
- **For relative files**, relative record keys are used to retrieve the records.

Syntax

Following is the syntax of sequential access mode:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT file-name ASSIGN TO dd-name
  ORGANIZATION IS SEQUENTIAL
  ACCESS MODE IS SEQUENTIAL


ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
  FILE-CONTROL.
  SELECT file-name ASSIGN TO dd-name
  ORGANIZATION IS INDEXED
  ACCESS MODE IS SEQUENTIAL
  RECORD KEY IS rec-key1
  ALTERNATE RECORD KEY IS rec-key2


ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
  FILE-CONTROL.
  SELECT file-name ASSIGN TO dd-name
  ORGANIZATION IS RELATIVE
  ACCESS MODE IS SEQUENTIAL
  RELATIVE KEY IS rec-key1
```

Random Access

When the access mode is RANDOM, the method of record retrieval changes as per the selected file organization.

- For indexed files, records are accessed according to the value placed in a key field which can be primary or alternate key. There can be one or more alternate indexes.

- For relative files , records are retrieved through relative record keys.

Syntax

Following is the syntax of random access mode:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
  FILE-CONTROL.
  SELECT file-name ASSIGN TO dd-name
  ORGANIZATION IS INDEXED
  ACCESS MODE IS RANDOM
  RECORD KEY IS rec-key1
  ALTERNATE RECORD KEY IS rec-key2


ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
  FILE-CONTROL.
  SELECT file-name ASSIGN TO dd-name
  ORGANIZATION IS RELATIVE
  ACCESS MODE IS RANDOM
  RELATIVE KEY IS rec-key1
```

Dynamic Access

Dynamic access supports both sequential and random access in the same program. With dynamic access, one file definition is used to perform both sequential and random processing like accessing some records in sequential order and others records by their keys.

With relative and indexed files, the dynamic access mode allows you to switch back and forth between sequential access mode and random access mode while reading a file by using the NEXT phrase on the READ statement. NEXT and READ functionalities will be discussed in the next chapter.

Syntax

Following is the syntax of dynamic access mode:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
  FILE-CONTROL.
  SELECT file-name ASSIGN TO dd-name
  ORGANIZATION IS SEQUENTIAL
  ACCESS MODE IS DYNAMIC
  RECORD KEY IS rec-key1
  ALTERNATE RECORD KEY IS rec-key2


ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
  FILE-CONTROL.
  SELECT file-name ASSIGN TO dd-name
  ORGANIZATION IS RELATIVE
  ACCESS MODE IS DYNAMIC
  RELATIVE KEY IS rec-key1
```

COBOL - FILE HANDLING VERBS

File handling verbs are used to perform various operations on files. Following are the file handling verbs:

- Open

- Read
- Write
- Rewrite
- Delete
- Start
- Close

Open Verb

Open is the first file operation that must be performed. If Open is successful, then only further operations are possible on a file. Only after opening a file, the variables in the file structure are available for processing. **FILE STATUS** variable is updated after each file operation.

Syntax

```
OPEN "mode" file-name.
```

Here, file-name is string literal, which you will use to name your file. A file can be opened in the followin modes:

Mode	Description
Input	Input mode is used for existing files. In this mode, we can only read the file, no other operations are allowed on the file.
Output	Output mode is used to insert records in files. If a **sequential file** is used and the file is holding some records, then the existing records will be deleted first and then new records will be inserted in the file. It will not happen so in case of **indexed file** or **relative file.**
Extend	Extend mode is used to append records in a **sequential file**. In this mode, records are inserted at the end. If file access mode is **Random** or **Dynamic**, then extend mode cannot be used.
I-O	Input-Output mode is used to read and rewrite the records of a file.

Read Verb

Read verb is used to read the file records. The function of read is to fetch records from a file. At each read verb, only one record can be read into the file structure. To perform a read operation, open the file in INPUT or I-O mode. At each read statement, the file pointer is incremented and hence the successive records are read.

Syntax

Following is the syntax to read the records when the file access mode is sequential:

```
READ file-name NEXT RECORD INTO ws-file-structure
AT END DISPLAY 'End of File'
NOT AT END DISPLAY 'Record Details:' ws-file-structure
END-READ.
```

Following are the parameters used:

- NEXT RECORD is optional and is specified when an indexed sequential file is being read sequentially.
- INTO clause is optional. ws-file-structure is defined in the Working-Storage Section to get the

values from the READ statement.

- AT END condition becomes True when the end of file is reached.

ExampleThe following example reads an existing file using line sequential organization. This program can be compiled and executed using **Try it** option where it will display all the records present in the file.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT STUDENT ASSIGN TO 'input.txt'
      ORGANIZATION IS LINE SEQUENTIAL.
DATA DIVISION.
FILE SECTION.
FD STUDENT.
01 STUDENT-FILE.
 05 STUDENT-ID PIC 9(5).
 05 NAME PIC A(25).
WORKING-STORAGE SECTION.
01 WS-STUDENT.
 05 WS-STUDENT-ID PIC 9(5).
 05 WS-NAME PIC A(25).
01 WS-EOF PIC A(1).
PROCEDURE DIVISION.
   OPEN INPUT STUDENT.
   PERFORM UNTIL WS-EOF='Y'
 READ STUDENT INTO WS-STUDENT
 AT END MOVE 'Y' TO WS-EOF
 NOT AT END DISPLAY WS-STUDENT
 END-READ
   END-PERFORM.
   CLOSE STUDENT.
STOP RUN.
```

Suppose the input file data available in the **input.txt** file contains the following:

```
20003 Mohtashim M.
20004 Nishant Malik
20005 Amitabh Bachhan
```

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

```
20003 Mohtashim M.
20004 Nishant Malik
20005 Amitabh Bachhan
```

Syntax

Following is the syntax to a read record when the file access mode is random:

```
READ file-name RECORD INTO ws-file-structure
KEY IS rec-key
INVALID KEY DISPLAY 'Invalid Key'
NOT INVALID KEY DISPLAY 'Record Details: ' ws-file-structure
END-READ.
```

Example: The following example reads an existing file using indexed organization. This program can be compiled and executed using **JCL** on Mainframes where it will display all the records present in the file. On Mainframes server we do not use text files; instead we use PS files.

Let's assume that the file present on Mainframes have same content as input.txt file in the above example.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT STUDENT ASSIGN TO IN1
       ORGANIZATION IS INDEXED
       ACCESS IS RANDOM
       RECORD KEY IS STUDENT-ID
       FILE STATUS IS FS.
DATA DIVISION.
FILE SECTION.
FD STUDENT.
01 STUDENT-FILE.
 05 STUDENT-ID PIC 9(5).
 05 NAME PIC A(25).
WORKING-STORAGE SECTION.
01 WS-STUDENT.
 05 WS-STUDENT-ID PIC 9(5).
 05 WS-NAME PIC A(25).
PROCEDURE DIVISION.
    OPEN INPUT STUDENT.
 MOVE 20005 TO STUDENT-ID.
 READ STUDENT RECORD INTO WS-STUDENT-FILE
 KEY IS STUDENT-ID
 INVALID KEY DISPLAY 'Invalid Key'
 NOT INVALID KEY DISPLAY WS-STUDENT-FILE
 END-READ.
    CLOSE STUDENT.
STOP RUN.
```

JCL to execute the above COBOL program:

```
//SAMPLE JOB(TESTJCL,XXXXXX),CLASS=A,MSGCLASS=C
//STEP1 EXEC PGM=HELLO
//IN1 DD DSN=STUDENT-FILE-NAME,DISP=SHR
```

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

```
20005 Amitabh Bachhan
```

Write Verb

Write verb is used to insert records in a file. Once the record is written, it is no longer available in the record buffer. Before inserting records into the file, move values into the record buffer and then perform write verb.

Write statement can be used with **FROM** option to directly write records from the working storage variables. From is an optional clause. If the access mode is sequential, then to write a record,s the file must open in Output mode or Extend mode. If the access mode is random or dynamic, then to write a record, the file must open in Output mode or I-O mode.

Syntax

Following is the syntax to read record when the file organization is sequential:

```
WRITE record-buffer [FROM ws-file-structure]
END-WRITE.
```

Following is the syntax to read a record when the file organization is indexed or relative:

```
WRITE record-buffer [FROM ws-file-structure]
INVALID KEY DISPLAY 'Invalid Key'
NOT INVALID KEY DISPLAY 'Record Inserted'
END-WRITE.
```

Example: The following example shows how to insert a new record in a new file when the organization is sequential.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT STUDENT ASSIGN TO OUT1
       ORGANIZATION IS SEQUENTIAL
       ACCESS IS SEQUENTIAL
       FILE STATUS IS FS.
DATA DIVISION.
FILE SECTION.
FD STUDENT
01 STUDENT-FILE.
 05 STUDENT-ID PIC 9(5).
 05 NAME PIC A(25).
 05 CLASS PIC X(3).
WORKING-STORAGE SECTION.
01 WS-STUDENT.
 05 WS-STUDENT-ID PIC 9(5).
 05 WS-NAME PIC A(25).
 05 WS-CLASS PIC X(3).
PROCEDURE DIVISION.
 OPEN EXTEND STUDENT.
 MOVE 1000 TO STUDENT-ID.
 MOVE 'Tim' TO NAME.
 MOVE '10' TO CLASS.
 WRITE STUDENT-FILE
 END-WRITE.
 CLOSE STUDENT.
STOP RUN.
```

JCL to execute the above COBOL program:

```
//SAMPLE JOB(TESTJCL,XXXXXX),CLASS=A,MSGCLASS=C
//STEP1 EXEC PGM=HELLO
//OUT1 DD DSN=OUTPUT-FILE-NAME,DISP=(NEW,CATALOG,DELETE)
```

When you compile and execute the above program, it will add a new record to the output file.

```
1000 Tim         10
```

Rewrite Verb

Rewrite verb is used to update the records. File should be opened in I-O mode for rewrite operations. It can be used only after a successful Read operation. Rewrite verb overwrites the last record read.

Syntax

Following is the syntax to read record when the file organization is sequential:

```
REWRITE record-buffer [FROM ws-file-structure]
END-REWRITE.
```

Following is the syntax to read a record when the file organization is indexed or relative:

```
REWRITE record-buffer [FROM ws-file-structure]
INVALID KEY DISPLAY 'Invalid Key'
NOT INVALID KEY DISPLAY 'Record Updated'
END-REWRITE.
```

Example: The following example shows how to update an existing record which we have inserted in previous Write step:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT STUDENT ASSIGN TO IN1
       ORGANIZATION IS INDEXED
       ACCESS IS RANDOM
       RECORD KEY IS STUDENT-ID
       FILE STATUS IS FS.
DATA DIVISION.
FILE SECTION.
FD STUDENT
01 STUDENT-FILE.
 05 STUDENT-ID PIC 9(4).
 05 NAME PIC A(12).
 05 CLASS PIC X(3).
WORKING-STORAGE SECTION.
01 WS-STUDENT.
 05 WS-STUDENT-ID PIC 9(5).
 05 WS-NAME PIC A(25).
 05 WS-CLASS PIC X(3).
PROCEDURE DIVISION.
 OPEN I-O STUDENT.
 MOVE '1000' TO STUDENT-ID.
 READ STUDENT
 KEY IS STUDENT-ID
 INVALID KEY DISPLAY ‘KEY IS NOT EXISTING’
 END-READ.
 MOVE 'Tim Dumais' TO NAME.
 REWRITE STUDENT-FILE
 END-REWRITE.
 CLOSE STUDENT.
STOP RUN.
```

JCL to execute the above COBOL program:

```
//SAMPLE JOB(TESTJCL,XXXXXX),CLASS=A,MSGCLASS=C
//STEP1 EXEC PGM=HELLO
//IN1 DD DSN=OUTPUT-FILE-NAME,DISP=SHR
```

When you compile and execute the above program, it will update the record:

```
1000 Tim Dumais  10
```

Delete Verb

Delete verb can be performed only on indexed and relative files. The file must be opened in I-O mode. In sequential file organization, records cannot be deleted. The record last read by the Read statement is deleted in case of sequential access mode. In random access mode, specify the record key and then perform the Delete operation.

Syntax

Following is the syntax to delete a record:

```
DELETE file-name RECORD
INVALID KEY DISPLAY 'Invalid Key'
NOT INVALID KEY DISPLAY 'Record Deleted'
END-DELETE.
```

Example to delete an existing record:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT STUDENT ASSIGN TO OUT1
       ORGANIZATION IS INDEXED
       ACCESS IS RANDOM
       RECORD KEY IS STUDENT-ID
       FILE STATUS IS FS.
DATA DIVISION.
FILE SECTION.
FD STUDENT
01 STUDENT-FILE.
 05 STUDENT-ID PIC 9(4).
 05 NAME PIC A(12).
 05 CLASS PIC X(3).
WORKING-STORAGE SECTION.
01 WS-STUDENT.
 05 WS-STUDENT-ID PIC 9(5).
 05 WS-NAME PIC A(25).
 05 WS-CLASS PIC X(3).
PROCEDURE DIVISION.
 OPEN I-O STUDENT.
 MOVE '1000' TO STUDENT-ID.
 DELETE STUDENT RECORD
 INVALID KEY DISPLAY 'Invalid Key'
 NOT INVALID KEY DISPLAY 'Record Deleted'
 END-DELETE.
 CLOSE STUDENT.
STOP RUN.
```

JCL to execute the above COBOL program:

```
//SAMPLE JOB(TESTJCL,XXXXXX),CLASS=A,MSGCLASS=C
//STEP1 EXEC PGM=HELLO
//OUT1 DD DSN=OUTPUT-FILE-NAME,DISP=SHR
```

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

```
Record Deleted
```

Start Verb

Start verb can be performed only on indexed and relative files. It is used to place the file pointer at a specific record. The access mode must be sequential or dynamic. File must be opened in I-O or Input mode.

Syntax

Following is the syntax to place the pointer at a specific record:

```
START file-name KEY IS [=, >, <, NOT, <= or >=] rec-key
INVALID KEY DISPLAY 'Invalid Key'
NOT INVALID KEY DISPLAY 'File Pointer Updated'
END-START.
```

Close Verb

Close verb is used to close a file. After performing Close operation the variables in the file structure will not be available for processing. The link between program and file is lost.

Syntax

Following is the syntax to close a file:

```
CLOSE file-name.
```

COBOL - SUBROUTINES

Cobol subroutine is a program that can be compiled independently but cannot be executed independently. There are two types of subroutines: **internal subroutines** like **Perform** statements and **external subroutines** like **CALL** verb.

Call Verb

Call verb is used to transfer the control from one program to another program. The program that contains the CALL verb is the **Calling Program** and the program being called is known as the **Called Program**. Calling program execution will halt until the called program finishes the execution. Exit Program statement is used in the Called program to transfer the control back.

Called Program Constraints

Following are the called program requirements:

- **Linkage section** must be defined in the called program. It consists of data elements passed in the program. The data items should not have Value clause. PIC clause must be compatible with the variables passed through the calling program.
- **Procedure division using** has a list of variables passed from the calling program and the order must be same as mentioned in the Call verb.
- **Exit program** statement is used in the Called program to transfer the control back. It must be the last statement in the called program.

The parameters can be passed between programs in two ways:

- By Reference
- By Content

Call By Reference

If the values of variables in the called program are modified, then their new values will reflect in the calling program. If **BY** clause is not specified, then variables are always passed by reference.

Syntax

Following is the syntax of calling subroutine by reference:

```
CALL sub-prog-name USING variable-1, variable-2.
```

Example

Following **example** is the MAIN calling program and UTIL is the called program:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-STUDENT-ID PIC 9(4) VALUE 1000.
01 WS-STUDENT-NAME PIC A(15) VALUE 'Tim'.
PROCEDURE DIVISION.
CALL 'UTIL' USING WS-STUDENT-ID, WS-STUDENT-NAME.
DISPLAY 'Student Id : ' WS-STUDENT-ID
DISPLAY 'Student Name : ' WS-STUDENT-NAME
STOP RUN.
```

Called Program

```
IDENTIFICATION DIVISION.
PROGRAM-ID. UTIL.
DATA DIVISION.
LINKAGE SECTION.
01 LS-STUDENT-ID PIC 9(4).
01 LS-STUDENT-NAME PIC A(15).
PROCEDURE DIVISION USING LS-STUDENT-ID, LS-STUDENT-NAME.
DISPLAY 'In Called Program'.
MOVE 1111 TO LS-STUDENT-ID.
EXIT PROGRAM.
```

JCL to execute the above COBOL program:

```
//SAMPLE JOB(TESTJCL,XXXXXX),CLASS=A,MSGCLASS=C
//STEP1 EXEC PGM=MAIN
```

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

```
In Called Program
Student Id : 1111
Student Name : Tim
```

Call By Content

If the values of variables in the called program are modified, then their new values will not reflect in the calling program.

Syntax

Following is the syntax of calling subroutine by content:

```
CALL sub-prog-name USING
BY CONTENT variable-1, BY CONTENT variable-2.
```

Example

Following **example** is the MAIN calling program and UTIL is the called program:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-STUDENT-ID PIC 9(4) VALUE 1000.
01 WS-STUDENT-NAME PIC A(15) VALUE 'Tim'.
PROCEDURE DIVISION.
CALL 'UTIL' USING BY CONTENT WS-STUDENT-ID, BY CONTENT WS-STUDENT-NAME.
DISPLAY 'Student Id : ' WS-STUDENT-ID
DISPLAY 'Student Name : ' WS-STUDENT-NAME
STOP RUN.
```

Called Program

```
IDENTIFICATION DIVISION.
PROGRAM-ID. UTIL.
DATA DIVISION.
LINKAGE SECTION.
01 LS-STUDENT-ID PIC 9(4).
01 LS-STUDENT-NAME PIC A(15).
PROCEDURE DIVISION USING LS-STUDENT-ID, LS-STUDENT-NAME.
DISPLAY 'In Called Program'.
MOVE 1111 TO LS-STUDENT-ID.
EXIT PROGRAM.
```

JCL to execute the above COBOL program:

```
//SAMPLE JOB(TESTJCL,XXXXXX),CLASS=A,MSGCLASS=C
//STEP1 EXEC PGM=MAIN
```

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

```
In Called Program
Student Id : 1000
Student Name : Tim
```

Types of Call

There are two types of calls`:

- **Static Call** occurs when a program is compiled with the NODYNAM compiler option. A static called program is loaded into storage at compile time.
- **Dynamic Call** occurs when a program is compiled with the DYNAM and NODLL compiler option. A dynamic called program is loaded into storage at runtime.

COBOL - INTERNAL SORT

Sorting of data in a file or merging of two or more files is a common necessity in almost all business-oriented applications. Sorting is used for arranging records either in ascending or descending order, so that sequential processing can be performed. There are two techniques which are used for sorting files in COBOL:

- **External sort** is used to sort files by using the SORT utility in JCL. We have discussed this in the JCL chapter. As of now, we will focus on internal sort.
- **Internal sort** is used to sort files with in a COBOL program. **SORT** verb is used to sort a file.

Sort Verb

Three files are used in the Sort process in COBOL:

- **Input file** is the file which we have to sort either in ascending or descending order.
- **Work file** is used to hold records while the sort process is in progress. Input file records are transferred to the work file for the sorting process. This file should be defined in the File-Section under SD entry.
- **Output file** is the file which we get after sorting process. It is the final output of the Sort verb.

Syntax

Following is the syntax to sort a file:

```
SORT work-file ON ASCENDING KEY rec-key1
               [ON DESCENDING KEY rec-key2]
USING input-file GIVING output-file.
```

SORT performs the following operations:

- Opens work-file in the I-O mode, input-file in the INPUT mode and output-file in the OUTPUT mode.
- Transfers the records present in the input-file to the work-file.
- Sorts the SORT-FILE in ascending/descending sequence by rec-key.
- Transfers the sorted records from the work-file to the output-file.

- Closes the input-file and the output-file and deletes the work-file.

Example

In the following below example INPUT is the input file which needs to be sorted in ascending order:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
 SELECT INPUT ASSIGN TO IN.
 SELECT OUTPUT ASSIGN TO OUT.
 SELECT WORK ASSIGN TO WRK.
DATA DIVISION.
FILE SECTION.
FD INPUT.
 01 INPUT-STUDENT.
  05 STUDENT-ID-I PIC 9(5).
  05 STUDENT-NAME-I PIC A(25).
FD OUTPUT.
 01 OUTPUT-STUDENT.
  05 STUDENT-ID-O PIC 9(5).
  05 STUDENT-NAME-O PIC A(25).
SD WORK.
 01 WORK-STUDENT.
  05 STUDENT-ID-W PIC 9(5).
  05 STUDENT-NAME-W PIC A(25).
PROCEDURE DIVISION.
 SORT WORK ON ASCENDING KEY STUDENT-ID-O
 USING INPUT GIVING OUTPUT.
 DISPLAY 'Sort Successful'.
STOP RUN.
```

JCL to execute the above COBOL program:

```
//SAMPLE JOB(TESTJCL,XXXXXX),CLASS=A,MSGCLASS=C
//STEP1 EXEC PGM=HELLO
//IN DD DSN=INPUT-FILE-NAME,DISP=SHR
//OUT DD DSN=OUTPUT-FILE-NAME,DISP=SHR
//WRK DD DSN=&&TEMP
```

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

```
Sort Successful
```

Merge Verb

Two or more identically sequenced files are combined using Merge statement. Files used in the merge process:

- Input Files : Input-1, Input-2
- Work File
- Output File

Syntax

Following is the syntax to merge two or more files:

```
MERGE work-file ON ASCENDING KEY rec-key1
               [ON DESCENDING KEY rec-key2]
USING input-1, input-2 GIVING output-file.
```

Merge performs the following operations:

- Opens the work-file in the I-O mode, input-files in the INPUT mode and output-file in the OUTPUT mode.
- Transfers the records present in the input-files to the work-file.
- Sorts the SORT-FILE in ascending/descending sequence by rec-key.
- Transfers the sorted records from the work-file to the output-file.
- Closes the input-file and the output-file and deletes the work-file.

Example

In the following example, INPUT1 and INPUT2 are input the files which are to be merged in ascending order:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
 SELECT INPUT1 ASSIGN TO IN1.
 SELECT INPUT2 ASSIGN TO IN2.
 SELECT OUTPUT ASSIGN TO OUT.
 SELECT WORK ASSIGN TO WRK.
DATA DIVISION.
FILE SECTION.
FD INPUT1.
 01 INPUT1-STUDENT.
  05 STUDENT-ID-I1 PIC 9(5).
  05 STUDENT-NAME-I1 PIC A(25).
FD INPUT2.
 01 INPUT2-STUDENT.
  05 STUDENT-ID-I2 PIC 9(5).
  05 STUDENT-NAME-I2 PIC A(25).
FD OUTPUT.
 01 OUTPUT-STUDENT.
  05 STUDENT-ID-O PIC 9(5).
  05 STUDENT-NAME-O PIC A(25).
SD WORK.
 01 WORK-STUDENT.
  05 STUDENT-ID-W PIC 9(5).
  05 STUDENT-NAME-W PIC A(25).
PROCEDURE DIVISION.
 MERGE WORK ON ASCENDING KEY STUDENT-ID-O
 USING INPUT1, INPUT2 GIVING OUTPUT.
 DISPLAY 'Merge Successful'.
STOP RUN.
```

JCL to execute the above COBOL program:

```
//SAMPLE JOB(TESTJCL,XXXXXX),CLASS=A,MSGCLASS=C
//STEP1 EXEC PGM=HELLO
//IN1 DD DSN=INPUT1-FILE-NAME,DISP=SHR
//IN2 DD DSN=INPUT2-FILE-NAME,DISP=SHR
//OUT DD DSN=OUTPUT-FILE-NAME,DISP=SHR
//WRK DD DSN=&&TEMP
```

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

```
Merge Successful
```

COBOL - DATABASE INTERFACE

As of now, we have learnt the use of files in COBOL. Now, we will discuss how a COBOL program interacts with DB2. It involves the following terms:

- Embedded SQL
- DB2 Application Programming
- Host Variables
- SQLCA
- SQL Queries
- Cursors

Embedded SQL

Embedded SQL statements are used in COBOL programs to perform standard SQL operations. Embedded SQL statements are preprocessed by SQL processor before the application program is compiled. COBOL is known as the **Host Language**. COBOL-DB2 applications are those applications that include both COBOL and DB2.

Embedded SQL statements work like normal SQL statements with some minor changes. For example, that output of a query is directed to a predefined set of variables which are referred as **Host Variables**. An additional INTO clause is placed in the SELECT statement.

DB2 Application Programming

Following are rules to be followed while coding a COBOL-DB2 program:

- All the SQL statements must be delimited between **EXEC SQL** and **END-EXEC**.
- SQL statements must be coded in Area B.
- All the tables that are used in a program must be declared in the Working-Storage Section. This is done by using the **INCLUDE** statement.
- All SQL statements other than INCLUDE and DECLARE TABLE must appear in the Procedure Division.

Host Variables

Host variables are used for receiving data from a table or inserting data in a table. Host variables must be declared for all values that are to be passed between the program and the DB2. They are declared in the Working-Storage Section.

Host variables cannot be group items, but they may be grouped together in host structure. They cannot be **Renamed** or **Redefined**. Using host variables with SQL statements, prefix them with a **colon (:)**.

Syntax

Following is the syntax to declare host variables and include tables in Working-Storage section:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
 EXEC SQL
 INCLUDE table-name
 END-EXEC.
 EXEC SQL BEGIN DECLARE SECTION
 END-EXEC.
  01 STUDENT-REC.
   05 STUDENT-ID PIC 9(4).
   05 STUDENT-NAME PIC X(25).
   05 STUDENT-ADDRESS X(50).
 EXEC SQL END DECLARE SECTION
 END-EXEC.
```

SQLCA

SQLCA is a SQL communication area through which DB2 passes the feedback of SQL execution to the program. It tells the program whether an execution was successful or not. There are a number of predefined variables under SQLCA like **SQLCODE** which contains the error code. The value '000' in SQLCODE states a successful execution.

Syntax

Following is the syntax to declare an SQLCA in the Working-Storage section:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
 EXEC SQL
 INCLUDE SQLCA
 END-EXEC.
```

SQL Queries

Lets assume we have one table named as 'Student' that contains Student-Id, Student-Name, and Student-Address.

The STUDENT table contains the following data:

```
Student Id  Student Name  Student Address
1001     Mohtashim M.  Hyderabad
1002   Nishant Malik  Delhi
1003     Amitabh Bachan  Mumbai
1004   Chulbul Pandey  Lucknow
```

The following **example** shows the usage of **SELECT** query in a COBOL program:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
 EXEC SQL
 INCLUDE SQLCA
 END-EXEC.
 EXEC SQL
 INCLUDE STUDENT
 END-EXEC.
 EXEC SQL BEGIN DECLARE SECTION
 END-EXEC.
  01 WS-STUDENT-REC.
   05 WS-STUDENT-ID PIC 9(4).
   05 WS-STUDENT-NAME PIC X(25).
   05 WS-STUDENT-ADDRESS X(50).
 EXEC SQL END DECLARE SECTION
 END-EXEC.
PROCEDURE DIVISION.
 EXEC SQL
  SELECT STUDENT-ID, STUDENT-NAME, STUDENT-ADDRESS
  INTO :WS-STUDENT-ID, :WS-STUDENT-NAME, WS-STUDENT-ADDRESS FROM STUDENT
  WHERE STUDENT-ID=1004
 END-EXEC.
 IF SQLCODE=0
  DISPLAY WS-STUDENT-RECORD
 ELSE DISPLAY 'Error'
 END-IF.
STOP RUN.
```

JCL to execute the above COBOL program:

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

```
//STEP001  EXEC PGM=IKJEFT01
//STEPLIB  DD DSN=MYDATA.URMI.DBRMLIB,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//SYSTSIN  DD *
    DSN SYSTEM(SSID)
    RUN PROGRAM(HELLO) PLAN(PLANNAME) -
    END
/*
```

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

```
1004 Chulbul Pandey  Lucknow
```

The following **example** shows the usage of **INSERT** query in a COBOL program:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
 EXEC SQL
 INCLUDE SQLCA
 END-EXEC.
 EXEC SQL
 INCLUDE STUDENT
 END-EXEC.
 EXEC SQL BEGIN DECLARE SECTION
 END-EXEC.
  01 WS-STUDENT-REC.
   05 WS-STUDENT-ID PIC 9(4).
   05 WS-STUDENT-NAME PIC X(25).
   05 WS-STUDENT-ADDRESS X(50).
 EXEC SQL END DECLARE SECTION
 END-EXEC.
PROCEDURE DIVISION.
 MOVE 1005 TO WS-STUDENT-ID.
 MOVE 'TutorialsPoint' TO WS-STUDENT-NAME.
 MOVE 'Hyderabad' TO WS-STUDENT-ADDRESS.
 EXEC SQL
  INSERT INTO STUDENT(STUDENT-ID, STUDENT-NAME, STUDENT-ADDRESS)
  VALUES (:WS-STUDENT-ID, :WS-STUDENT-NAME, WS-STUDENT-ADDRESS)
 END-EXEC.
 IF SQLCODE=0
  DISPLAY 'Record Inserted Successfully'
  DISPLAY WS-STUDENT-REC
 ELSE DISPLAY 'Error'
 END-IF.
STOP RUN.
```

JCL to execute the above COBOL program.

```
//SAMPLE JOB(TESTJCL,XXXXXX),CLASS=A,MSGCLASS=C
//STEP001  EXEC PGM=IKJEFT01
//STEPLIB  DD DSN=MYDATA.URMI.DBRMLIB,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//SYSTSIN  DD *
    DSN SYSTEM(SSID)
    RUN PROGRAM(HELLO) PLAN(PLANNAME) -
    END
/*
```

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

```
Record Inserted Successfully
1005 TutorialsPoint  Hyderabad
```

The following **example** shows the usage of **UPDATE** query in a COBOL program:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
 EXEC SQL
 INCLUDE SQLCA
 END-EXEC.
 EXEC SQL
 INCLUDE STUDENT
 END-EXEC.
 EXEC SQL BEGIN DECLARE SECTION
 END-EXEC.
  01 WS-STUDENT-REC.
   05 WS-STUDENT-ID PIC 9(4).
   05 WS-STUDENT-NAME PIC X(25).
   05 WS-STUDENT-ADDRESS X(50).
 EXEC SQL END DECLARE SECTION
 END-EXEC.
PROCEDURE DIVISION.
 MOVE 'Bangalore' TO WS-STUDENT-ADDRESS.
 EXEC SQL
  UPDATE STUDENT SET STUDENT-ADDRESS=:WS-STUDENT-ADDRESS
  WHERE STUDENT-ID=1003
 END-EXEC.
 IF SQLCODE=0
  DISPLAY 'Record Updated Successfully'
 ELSE DISPLAY 'Error'
 END-IF.
STOP RUN.
```

JCL to execute the above COBOL program:

```
//SAMPLE JOB(TESTJCL,XXXXXX),CLASS=A,MSGCLASS=C
//STEP001  EXEC PGM=IKJEFT01
//STEPLIB  DD DSN=MYDATA.URMI.DBRMLIB,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//SYSTSIN  DD *
    DSN SYSTEM(SSID)
    RUN PROGRAM(HELLO) PLAN(PLANNAME) -
    END
/*
```

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

```
Record Updated Successfully
```

The following **example** shows the usage of **DELETE** query in a COBOL program:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
 EXEC SQL
 INCLUDE SQLCA
 END-EXEC.
 EXEC SQL
 INCLUDE STUDENT
 END-EXEC.
 EXEC SQL BEGIN DECLARE SECTION
 END-EXEC.
  01 WS-STUDENT-REC.
   05 WS-STUDENT-ID PIC 9(4).
   05 WS-STUDENT-NAME PIC X(25).
   05 WS-STUDENT-ADDRESS X(50).
 EXEC SQL END DECLARE SECTION
 END-EXEC.
PROCEDURE DIVISION.
 MOVE 1005 TO WS-STUDENT-ID.
 EXEC SQL
  DELETE FROM STUDENT
  WHERE STUDENT-ID=:WS-STUDENT-ID
 END-EXEC.
 IF SQLCODE=0
  DISPLAY 'Record Deleted Successfully'
 ELSE DISPLAY 'Error'
 END-IF.
STOP RUN.
```

JCL to execute the above COBOL program:

```
//SAMPLE JOB(TESTJCL,XXXXXX),CLASS=A,MSGCLASS=C
//STEP001  EXEC PGM=IKJEFT01
//STEPLIB  DD DSN=MYDATA.URMI.DBRMLIB,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//SYSTSIN  DD *
    DSN SYSTEM(SSID)
    RUN PROGRAM(HELLO) PLAN(PLANNAME) -
    END
/*
```

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

```
Record Deleted Successfully
```

Cursors

Cursors are used to handle multiple row selections at a time. They are data structures that hold all the results of a query. They can be defined in the Working-Storage Section or the Procedure Division. Following are the operations associated with Cursor:

- Declare
- Open
- Close
- Fetch

Declare Cursor

Cursor declaration can be done in the Working-Storage Section or the Procedure Division. The first statement is the DECLARE statement which is a non-executable statement.

```
EXEC SQL
 DECLARE STUDCUR CURSOR FOR
 SELECT STUDENT-ID, STUDENT-NAME, STUDENT-ADDRESS FROM STUDENT
 WHERE STUDENT-ID >:WS-STUDENT-ID
END-EXEC.
```

Open

Before using a cursor, Open statement must be performed. The Open statement prepares the

SELECT for execution.

```
EXEC SQL
 OPEN STUDCUR
END-EXEC.
```

Close

Close statement releases all the memory occupied by the cursor. It is mandatory to close a cursor before ending a program.

```
EXEC SQL
 CLOSE STUDCUR
END-EXEC.
```

Fetch

Fetch statement identifies the cursor and puts the value in the INTO clause. A Fetch statement is coded in loop as we get one row at a time.

```
EXEC SQL
 FETCH STUDCUR
 INTO :WS-STUDENT-ID, :WS-STUDENT-NAME, WS-STUDENT-ADDRESS
END-EXEC.
```

The following **example** shows the usage of cursor to fetch all the records from the STUDENT table:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
DATA DIVISION.
WORKING-STORAGE SECTION.
 EXEC SQL
 INCLUDE SQLCA
 END-EXEC.
 EXEC SQL
 INCLUDE STUDENT
 END-EXEC.
 EXEC SQL BEGIN DECLARE SECTION
 END-EXEC.
  01 WS-STUDENT-REC.
   05 WS-STUDENT-ID PIC 9(4).
   05 WS-STUDENT-NAME PIC X(25).
   05 WS-STUDENT-ADDRESS X(50).
 EXEC SQL END DECLARE SECTION
 END-EXEC.
 EXEC SQL
  DECLARE STUDCUR CURSOR FOR
  SELECT STUDENT-ID, STUDENT-NAME, STUDENT-ADDRESS FROM STUDENT
  WHERE STUDENT-ID >:WS-STUDENT-ID
 END-EXEC.
PROCEDURE DIVISION.
 MOVE 1001 TO WS-STUDENT-ID.
 PERFORM UNTIL SQLCODE = 100
 EXEC SQL
  FETCH STUDCUR
  INTO :WS-STUDENT-ID, :WS-STUDENT-NAME, WS-STUDENT-ADDRESS
 END-EXEC
 DISPLAY WS-STUDENT-REC
 END-PERFORM
STOP RUN.
```

JCL to execute the above COBOL program:

```
//SAMPLE JOB(TESTJCL,XXXXXX),CLASS=A,MSGCLASS=C
//STEP001  EXEC PGM=IKJEFT01
```

```
//STEPLIB  DD DSN=MYDATA.URMI.DBRMLIB,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//SYSTSIN  DD *
    DSN SYSTEM(SSID)
    RUN PROGRAM(HELLO) PLAN(PLANNAME) -
    END
/*
```

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

```
1001 Mohtashim M.  Hyderabad
1002 Nishant Malik  Delhi
1003 Amitabh Bachan  Mumbai
1004 Chulbul Pandey  Lucknow
```