**COBOL (COMMON BUSINESS ORIENTED LANGUAGE)**

* It is High level programming language.
* Cobol was developed because in early times as creating program then translating it was very costly.
  + New language should have:
    - Maximal use of English lang
    - Be capable of change
    - Machine independent
    - Easy to use
    - Able to connect o multiple applications

**Coding Rules of COBOL:**

* Col 1-6. This is reserved for sequence numbers, use to label a source statement line.
* Col 7. Indicator area used for comments, continuation, ‘D’
* Col 8-11. Area A. Division, Sections, Paragraph, Level is defined.
* Col 12-72. Area B. Statements, Sentences.
* Col 73-80. Used for program identification, we give identification code.
* COBOL **statements** are combined to form a **sentence** and sentence is terminated by period (.)
* We need at least 2 divisions to execute a Hello World program. Those are Identification Division and Procedure Division.
* Division has to be given at position 8 and statements has to be given at position 12.
* Structure of COBOL:
  + **DIVISION**
  + **SECTION**
  + **PARAGRAPH**
  + **SENTENCE**
  + **STATEMENT**
  + **CLAUSE/VERB**
    - **MOVE, COMPUTE, DISPLAY, PIC,.. all the keywords of cobol.**
  + **CHARACTER/WORD**

**COBOL Structure:**

* **IDENTIFICATION DIVISION (Mandatory)**
  + Supply information about the programs such as program name, name of programmer.

**IDENTIFICATION DIVISION.**

**PROGRAM-ID. HELLOWORLD.**

**AUTHOR. ANUBHAV.**

**DATE-WRITTEN. 13/09/2024.**

**DATE-COMPILED. 13/09/24.**

**SECURITY. THIS IS CONFIDENTIAL.**

* **ENVIRONMENT DIVISION**
  + Describes the environment in which the program will run, such as source and object computer.
  + It is divided in 2 Sections:
    - **CONFIGURATION SECTION.**
      * It gives the details about the machine which is used, for example source computer and the topic computer.
    - **INPUT-OUTPUT SECTION.**
      * Input output section defines the input and output files used in the program and links it to external device where the file is stored.

**ENVIRONMENT DIVISION.**

**INPUT-OUTPUT SECTION.**

**FILE-CONTROL.**

**SELECT** INPUTFL **ASSIGN TO** DDINFL

**ORGANIZATION IS SEQUENTIAL**

**ACCESS MODE IS SEQUENTIAL**

**FILE STATUS IS** INPUTFL-FS**.**

* **DATA DIVISION**
  + It provides the description of data to be processed by the program such as input and output files.
  + It has 3 major sections:
    - **FILE SECTION**
      * Used to provide the File Definition of the files used in the program
      * Under this, it has FD (File Descriptor) for each file to define the layout of the file.
    - **WORKING-STORAGE SECTION**
      * It is used to define all the temporary variables used in the program.
    - **LINKAGE SECTION**
      * Used when a program (main program) calls another program (subprogram)
      * LINKAGE SECTION is defined in the subprogram.
      * We need to define the arguments which are passed by the main program in the LINKAGE SECTION as well.

**DATA DIVISION.**

**FILE SECTION.**

**FD INPUTFL.**

**01 INP-RECORD.**

**05 INVOICE PIC X(10).**

**05 STORE-ID PIC X(05).**

**05 ITEM-ID PIC X(10).**

**WORKING-STORAGE SECTION.**

1. **INPUTFL-FS PIC 9(02).**
2. **WS-ID-DISPLAY PIC X(30).**

* **PROCEDURE DIVISION (Mandatory)**
  + Contains the code of the program.
  + User defined sections, paragraphs, sentences, statements, clause and verbs are defined in this section.

**PROCEDURE DIVISION.**

**MAIN-PARA.**

**MOVE “Environment and data division” TO WS-ID-DISPLAY.**

**DISPLAY WS-ID-DISPLAY.**

**STOP RUN.**

**COBOL BASICS**

**Variables, Literals, Figurative constants:**

* **Variables:**
  + Identifier to hold a value, it identifies a memory.
  + It can be max length of 30 characters.
  + Must only contain digits (0-9), letters (A-Z) and Hyphen (-).
  + Must not be a reserved word.
  + Variable should not contain space.
* **Literals:**
  + Literals are constants, these are directly hard coded to the program.
  + Literals are of 2 types:
    - Numeric literal:
      * Ex: 123, 32
      * Max of 18 characters
      * 0 to 9
      * Sign must be on left side (+ or -)
      * One decimal only
    - Non-numeric literal:
      * Ex: “I am an example”
      * Max of 160 characters.
      * Must start and end with quotes.
* **Figurative Constants:**
  + These are constants which are predefined in COBOL.
  + ZERO | ZEROS | ZEROES
  + SPACE | SPACES
  + HIGH-VALUE | HIGH-VALUES – Highest ordinal position
  + LOW-VALUE | LOW-VALUES – Lowest ordinal position
  + QUOTE | QUOTES
  + ALL
  + NULL | NULLS

**DATA DIVISION.**

**WORKING-STORAGE SECTION.**

**01 WS-VARIABLE PIC X(30).**

**01 WS-LITERAL-1 PIC 9(03) VALUE 123.**

**01 WS-LITERAL-2 PIC A(15) VALUE “I AM A LITERAL”.**

**01 WS-LITERAL-3 PIC X(03) VALUE ‘X23’.**

**PROCEDURE DIVISION.**

**MAIN-PARA.**

**MOVE SPACES TO WS-VARIABLE.**

**MOVE ‘THIS DATA TO WS-VAR’ TO WS-VARIABLE.**

**DISPLAY WS-LITERAL-1.**

**DISPLAY WS-LITERAL-3.**

**STOP RUN.**

**Data Types of COBOL:**

* Data types are denoted by PICTURE (PIC) clause.
* PIC clause provides the data types along with length.
  + **Numeric (9):** 0 to 9, max length is 18. Ex: PIC 9(2).
  + **Alphabet (A):** A to Z, a to z, max length is 255. Ex: PIC A(10).
  + **Alphanumeric (X):** combination of numeric and alphabets. Ex: PIC X(10).
  + **Signed (S):** It links a sign to a number. If this is present the number is signed number, if not then number is unsigned number.
  + **+ sign:** Used to print (+) as sign
  + **– sign:** Used to print (-) as sign
  + **V: Implied decimal**, does not hold any memory space. This is not used for display but rather for computation. Ex: PIC 9(4)V999, if the value is defined as 123456, then it is like 0123.456 but it will not be displayed like 123.456. If this is used in computation, the value will be 123.456 which will take part in computation.
  + **.: Actual decimal point**. Used for display not for calculation or any arithmetic operations. Ex: PIC 9(4).99, if the value is 123456, then it will display 1234.56 but it cannot be used for computation.
  + **Z**: Used to suppress only the leading zeroes with blanks. Does not do anything with non-zeros. Ex: PIC ZZ9(2).99, if value is 0012.34 then it will be bb12.34
  + **Comma**: To insert a comma into the data item at a particular position.
  + **$**: Dollar symbol to insert a dollar sign at the first position, this is normally used for currency.

**WORKING-STORAGE SECTION. Displayed in output**

**01 WS-NUMERIC PIC 9(3) VALUE 123. 123**

**01 WS-ALPHABET PIC A(10) VALUE ‘ANUBHAV’. ANUBHAV**

**01 WS-ALPHANUMERIC PIC X(5) VALUE ‘XX123’. XX123**

**01 WS-SIGNED-P PIC S9(3) VALUE +123. 12C**

**01 WS-SIGNED-N PIC S9(3) VALUE -123. 12L**

**01 WS-POSSIGN PIC +9(3) VALUE 123. +123**

**01 WS-NEGSIGN PIC -9(3) VALUE 123. -123**

**01 WS-IMPLIED-DEC PIC 9(3)V99 VALUE 123.45. 12345**

**01 WS-ACTUAL-DEC PIC 9(3).99 VALUE 12345. 123.45**

**01 WS-ZERO-SUP PIC ZZ99 VALUE 0012. 12**

|  |  |  |
| --- | --- | --- |
| **Actual No** | **Signed Positive** | **Signed Negative** |
| **0** | **{** | **}** |
| **1** | **A** | **J** |
| **2** | **B** | **K** |
| **3** | **C** | **L** |
| **4** | **D** | **M** |
| **5** | **E** | **N** |
| **6** | **F** | **O** |
| **7** | **G** | **P** |
| **8** | **H** | **Q** |
| **9** | **I** | **R** |

**Level Numbers:**

* **01 Level:**
  + Topmost level.
  + This can be individual data item or group data item.
  + There cannot be another 01 level within a 01 level.
  + Ex:
    - 01 EMP-ID PIC X(20). Individual Item
    - 01 EMP-RECORD. Group Item

05 EMP-ID PIC 9(5).

05 EMP-NAME.

10 FIRST-NAME PIC A(10).

10 LAST-NAME PIC A(10).

* + Level number with same numeric value are considered to be at the same level.
  + As we move down, the level number must increase.
  + Level 01-49 is used for general purpose.
* **66 Level:**
  + Used for **RENAMES** clause. It should not have a picture clause.
  + Ex:
    - 01 EMP-REC.

05 EMP-ID PIC 9(5).

05 EMP-NAME PIC X(20).

05 EMP-DOB PIC X(10).

**66 EMP-DETAIL RENAMES EMP-ID THRU EMP-DOB.**

* **77 Level:**
  + Normally avoid using it.
  + It is used for individual data item or elementary data item.
  + It cannot be further subdivided.
* **88 Level:**
  + It is used for conditional processing.
  + This must be coded under a group item; it works on the principle of true or false.
  + Ex:

01 CHECK-DAY.

05 DAY PIC X(3).

88 MONDAY VALUE ‘MON’.

88 TUESDAY VALUE ‘TUE’.

88 WEDNESDAY VALUE ‘WED’.

**WORKING-STORAGE SECTION.**

**01 STUDENTS.**

**05 STUDENT-ID PIC 9(3) VALUE 12345.**

**05 FILLER PIC X(1).**

**05 STUDENT-NAME.**

**10 FNAME PIC A(10).**

**10 LNAME PIC A(10).**

**05 STUDENT-DOB PIC X(10) VALUE 21/08/1980.**

**05 STUDENT-DEPT PIC X(3).**

**88 COMPUTER VALUE ‘CMP’.**

**88 ELECTRONICS VALUE ‘EEE’.**

**88 MECHANICAL VALUE ‘MEC’.**

**88 CIVIL VALUE ‘CIV’.**

**66 STUDENT-DETAILS RENAMES STUDENT-ID THRU STUDENT-NAME.**

**77 WS-STUDENT-FLAG PIC X(1) VALUE ‘A’.**

**PROCEDURE DIVISION.**

**MAIN-PARA.**

**MOVE ‘MICHAEL’ TO FNAME.**

**MOVE ‘EVE’ TO LNAME.**

**DISPLAY STUDENTS.**

**DISPLAY STUDENT-DETAILS.**

**SET ELECTRONICS TO TRUE.**

**DISPLAY STUDENT-DEPT.**

**STOP RUN.**

**Output:**

**12345 MICHAEL EVE 21/08/1980**

**12345 MICHAEL EVE**

**EEE**

**DISPLAY and MOVE clause:**

* **Display:**
  + To display the output.
* **Move:**
  + To assign a value to a variable.
  + Various types of move:
    - Simple move
    - Substring move

**MOVE “HELLO WORLD” TO WS-HELLO.**

**MOVE WS-HELLO(7:5) TO WS-REF-MOD.**

**DISPLAY WS-REF-MOD. /\* WORLD**

* + - Corresponding move
      * When GP-1 has same variable as GP-2, then it’ll move the value of variables to the corresponding variables irrespective of the position of variables.

**MOVE CORRESPONDING GP-1 TO GP-2.**

**ACCEPT clause:**

* This is used to accept the value from the JCL or a system defined value.
* Syntax:
  + ACCEPT IDENTIFIER FROM [Mnemonic-name]
* Example:
  + ACCEPT EMP-DET.
  + ACCEPT CURRENT-DATE FROM DATE.
  + ACCEPT CURRET-TIME FROM TIME.
* Data to the program can be passed using ‘SYSIN’ parameter which is coded in JCL.
* The output of the program can be seen in SYSOUT which has to be coded in JCL.

**SYSIN and SYSOUT:**

* **SYSIN Parameter:**
  + It is a way to pass input data to the program.
  + To accept the data in the program, we need to code ‘ACCEPT’ statement.
  + DD \* is called in-stream data
  + Syntax:
    - //SYSIN DD \*

Values…

/\*

* + **Note: There is another way of passing the data to the program which is ‘PARM’. If SYSIN and PARM both are defined in JCL, PARM will override the values in SYSIN.**
* **SYSOUT Parameter:**
  + It is used to display the output of the program in SPOOL or to an output class.
  + Syntax:
    - //DDNAME DD SYSOUT=\*

**First Program:**

**IDENTIFICATION DIVISION.**

**PROGRAM-ID. COB01.**

**PROCEDURE DIVISION.**

**MAIN-PARA.**

**DISPLAY ‘Hello World’.**

**STOP RUN.**

**Create Compile JCL:**

**//JOBCARD**

**//\***

**//STEP01 EXEC COBOLCL,**

**// COPYLIB=NUHID.XXX.COPYLIB, /\*PDS where copybooks are kept\*/**

**// LOADLIB=NUHID.XXX.LOADLIB,**

**// SRCLIB=NUHID.XXX.XXX, /\*PDS where COBOL source code is kept\*/**

**// MEMBER=COB01 /\*Member were code is kept\*/**

**When we compile our program then the executable code will be kept under load module, which will be kept under load lib.**

**Create RUN JCL:**

**//JOBCARD**

**//\***

**//STEP01 EXEC PGM=COB01**

**//STEPLIB DD DSN=NUHID.XXX.LOADLIB,DISP=SHR**

**//SYSPRINT DD SYSOUT=\***

**//SYSOUT DD SYSOUT=\***

**ARITHMETIC OPERATIONS**

* There are 5 imp arithmetic operations present in COBOL:
  + **ADD**
    - ADD {CORRESPONDING/CORR} {IDENTIFIER1} TO {IDENTIFIER2} GIVING {IDENTIFIER3} {ROUNDED}

{ON SIZE ERROR imperative-statement-1}

{NOT ON SIZE ERROR imperative-statement-2}

END-ADD

* + - ADD A TO B. /\*B=A+B\*/
    - ADD CORR A TO B.
    - ADD A TO B GIVING C.
    - ADD A B GIVING C D. /\*C=A+B, D=A+B\*/
    - ADD A TO B ROUNDED.
    - ADD A TO B ROUNDED /\*A=60, B=50, THEN B=A+B I.E. B=110\*/

ON SIZE ERROR DISPLAY ‘SIZE ERROR’.

* + - A PIC 9(2) VALUE 60.

B PIC 9(2) VALUE 50.

ADD A TO B. **/\*B SHOULD BE 110 BUT IT’LL 10 AS B IS 9(2)\*/**

* + **SUBTRACT**
    - SUBTRACT {CORRESPONDING/CORR} {IDENTIFIER1} FROM {IDENTIFIER2} GIVING {IDENTIFIER3} {ROUNDED}

{ON SIZE ERROR imperative-statement-1}

{NOT ON SIZE ERROR imperative-statement-2}

* + - SUBTARCT A FROM B. /\*B=B-A\*/
    - SUBTARCT A TO B GIVING C. /\*C=B-A\*/
    - SUBTARCT CORR A FROM B.
    - SUBTARCT A B FROM C. /\*C=C-(A+B)\*/
    - SUBTARCT A FROM B ROUNDED.
    - SUBTARCT A FROM B ROUNDED

ON SIZE ERROR DISPLAY ‘SIZE ERROR’.

* + **MULTIPLY**
    - MULTIPLY {IDENTIFIER1} BY {IDENTIFIER2} GIVING {IDENTIFIER3} {ROUNDED}

{ON SIZE ERROR imperative-statement-1}

{NOT ON SIZE ERROR imperative-statement-2}

* + - MULTIPLY A BY B. /\*B=AxB\*/
    - MULTIPLY A BY B GIVING C.
    - MULTIPLY A BY B GIVING C D. /\*C=AxB, D=AxB\*/
    - MULTIPLY A BY B ROUNDED

ON SIZE ERROR DISPLAY ‘SIZE ERROR’.

* + **DIVIDE**
    - DIVIDE {IDENTIFIER1} BY/INTO {IDENTIFIER2} GIVING {IDENTIFIER3} {ROUNDED}

REMAINDER {IDENTIFIRE4}

{ON SIZE ERROR imperative-statement-1}

{NOT ON SIZE ERROR imperative-statement-2}

* + - DIVIDE A INTO B. /\*B=B/A\*/
    - DIVIDE A BY B. WRONG SYNTAX
    - DIVIDE A INTO B GIVING C. /\*C=B/A\*/
    - DIVIDE A BY B GIVING C. /\*C=A/B\*/
    - DIVIDE A INTO B GIVING C D. /\*C=B/A, D=B/A\*/
    - DIVIDE A BY B GIVING C D. /\*C=A/B, D=A/B\*/
    - DIVIDE A BY B GIVING C REMAINDER D. /\*A=106, B=20, I.E. C=A/B=5, D=6\*/
    - DIVIDE A INTO B GIVING C REMAINDER D. /\*A=12, B=105, I.E. C=B/A=8, D=9\*/
    - DIVIDE A BY B GIVING C REMAINDER D

ON SIZE ERROR MOVE 1 TO B.

* + **COMPUTE:**
    - COMPUTE is used to assign the value of the arithmetic operations which happens at the right side of ‘=’ to the literal/variable which is present at the left side of the ‘=’.
    - COMPUTE can combine all the arithmetic operation and assign the result to a variable.
    - COMPUTE {IDENTIFIER1} ROUNDED = {OPERATIONS LIKE +,-,\*,/,\*\*}

{ON SIZE ERROR/NOT ON SIZE ERROR} {IMPERATIVE-STATEMENT}

* + - COMPUTE A = B-C.
    - COMPUTE D ROUNDED = A+B.
    - COMPUTE E ROUNDED = (A\*B)/(D-C)

NOT ON SIZE ERROR

DISPLAY ‘NOT ON SIZE ERROR’.

**CONDITIONAL STATEMENTS**

**Conditional Processing:**

* It is used to process a sentence based on certain condition:
  + **PERFORM:**
    - Iteration and looping can be done using PERFORM.
    - **PERFORM {PARA-1/SECTION}**

**………**

**END-PERFORM.**

**PARA-1.**

**…..**

**…..**

* + - **PERFORM {PARA-1/SECTION-1} THRU {PARA-N/SECTION-N} {NUMBER} TIMES**

**……**

**END-PERFORM.**

* + - **PERFORM {PARA/SECTION} WITH TEST BEFORE/AFTER UNTIL {CONDITION-IS-TRUE}**

**………**

**END-PERFORM.**

PERFORM PARA-1 WITH TEST BEFORE UNTIL AGE>10 /\*condition is checked before\*/

PERFORM PARA-1 WITH TEST AFTER UNTIL AGE>10 /\*condition is checked after\*/

PERFORM PARA-1 UNTIL AGE>10

* + - **PERFORM {PARA-1} THRU {PARA-N} WITH TEST BEFORE/AFTER**

**VARYING {INDEX-VARIABLE} FROM {START-COUNTER} BY {INCREMENT-COUNTER}**

**UNTIL {CONDITION}**

**….**

**END-PERFORM.**

PERFORM PARA-1

VARYING A FROM 1 BY 1 UNTIL A>10

…..

END-PERFORM.

* + **IF-ELSE**
    - **IF condition-is-true**

**{statements} / {CONTINUE/NEXT SENTENCE}**

**ELSE**

**{statements} / {CONTINUE/NEXT SENTENCE}**

**END-IF.**

* + - **IF condition-is-true**

**{statements} / {CONTINUE/NEXT SENTENCE}**

**IF condition-is-true**

**{statements} / {CONTINUE/NEXT SENTENCE}**

**ELSE**

**{statements} / {CONTINUE/NEXT SENTENCE}**

**END-IF**

**ELSE**

**{statements} / {CONTINUE/NEXT SENTENCE}.**

* + - NOTE:
      * We can say, IF A EQUAL B | IF A = B
      * We can say, IF A NOT EQUAL B | IF A NE B | IF A NOT = B
  + **EVALUATE**
    - It is similar to switch statement.
    - If we have to check multiple condition, we use EVALUATE in place of IF-ELSE.
    - Here, {cond-for-evaluate} can be:
      * Constants/literals
      * Expression/identifier
      * Reference
      * TRUE/FALSE
    - **EVALUATE {cond-for-evaluate} ALSO {cond-for-evaluate} ALSO …**

**WHEN {possible-cond1} ALSO {possible-cond2} ALSO…**

**{STATEMENTS}**

**WHEN {possible-cond1} ALSO {possible-cond2} ALSO…**

**{STATEMENTS}**

**…..**

**…..**

**WHEN OTHER**

**{STATEMENTS}**

**END-EVALUATE.**

Example:

EVALUATE TRUE

WHEN A=5

….

WHEN A=10

…

WHEN OTHER

……

END-EVALUATE.

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

EVALUATE A

WHEN 5

…..

WHEN 10

……

WHEN 15

…..

END-EVALUATE.

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

EVALUATE AGE

WHEN 0 THRU 9

….

WHEN 10 THRU 19

…..

WHEN 20 THRU 45

…..

WHEN OTHER

…..

END-EVALUATE

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

EVALUATE AGE

WHEN 1

WHEN 5

WHEN 9

…..

WHEN 10

WHEN 15

…..

WHEN OTHER

…….

END-EVALUATE.

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

EVALUATE TRUE ALSO AGE /\*It means if A=5 is TRUE and AGE is 1 the execute that statement\*/

WHEN A=5 ALSO 1

…..

WHEN A=10 ALSO 10

……

WHEN OTHER

……

END-EVALUATE.

**INTERNAL TABLE (ARRAYS)**

**ARRAY:**

* If there are multiple similar items which needs to be places under a group, we have to define it multiple times.
  + To sort this issue, arrays are introduced in COBOL.
  + Similar items of same behaviour can be grouped together in internal table in COBOL to optimize it usage.
* An array is a collection of similar data elements stored at continuous memory locations.
* Syntax:
  + **01 WEEK-TABLE.**

**05 DAY-NAME PIC X(10) OCCURS 7 TIMES.**

**OR**

**05 DAY-NAME OCCURS 7 TIMES PIC X(10).**

* We can define a table using **OCCURS** clause:
  + OCCURS clause can be used to specify the number of occurrences of an element in table array.
  + OCCURS clause must be defined for group or elementary data items only.
  + We can have multiple OCCURS clause depending on whether we want to create a 1-D or 7-D array. (Max we can have 7-D array).
  + To access the elements of the table, we can use **Index or Subscript.**
  + Table is divided into ROWS and COLUMNS.
* **Index:**
  + It is offset or number of displacement position from the starting of the array.
  + Used to reference an array/table element.
  + **INDEX BY** clause is used to declare INDEX.
  + INDEX must not be defined in WORKING-STORAGE SECTION.
  + Syntax:
    - **01 WEEK-TABLE.**

**05 DAY-NAME PIC X(9) OCCURS 7 TIMES**

**ASCENDING/DESCENDING KEY IS key1**

**INDEXED BY index-name.**

* + - Ex:

DAY-NAME(1) has displacement of 0

DAY-NAME(2) has displacement of 9

DAY-NAME(3) has displacement of 18 and so on..

If we increment the index by 1, the length gets increase by the length defined in the PIC clause.

* + - How to assign a value to INDEX?
      * SET clause is used to initialize INDEX or to set the value of the INDEX.
      * SET index-1 to index-2
      * SET index to 1
      * SET index-1 UP BY 1
      * SET index-1 DOWN BY 1
      * SET is also used to set the value of switch to ON/OFF. It can also be used to assign TRUE/FALSE to a conditional variable.
* **Subscript:**
  + It is an integer data item which describes the number of occurrences of array element in table.
  + Used to reference an array/table element.
  + Subscript is not the actual memory location rather it refers a position of the data item. It refers the memory location a particular data item resides.
  + Subscript should have an entry in WORKING-STORAGE SECTION with ‘PIC S(4) COMP.’
  + To initialize the subscript, we can use MOVE statement.
  + Subscript starts from number 1 and increment by 1 every time by using ADD statement.
* **1-D Array:**

**IDENTIFICATION DIVISION.**

**PROGRAM-ID. ONEDARRAY.**

**ENVIRONMENT DIVISION.**

**DATA DIVISION.**

**WORKING-STORAGE SECTION.**

**01 WS-TABLE.**

**05 WS-PRD PIC X(10) OCCURS 3 TIMES.**

**PROCEDURE DIVISION.**

**PERFORM PARA-1.**

**STOP-RUN.**

**PARA-1.**

**MOVE ‘SOAP ‘ TO WS-PRD(1).**

**MOVE ‘DETERGENT ‘ TO WS-PRD(2).**

**MOVE ‘SHAMPOO ‘ TO WS-PRD(3).**

**DISPLAY WS-PRD(1).**

**DISPLAY WS-PRD(2).**

**DISPLAY WS-PRD(3).**

* **2-D Array:**

**IDENTIFICATION DIVISION.**

**PROGRAM-ID. TWODARRAY.**

**ENVIRONMENT DIVISION.**

**DATA DIVISION.**

**WORKING-STORAGE SECTION.**

**01 WS-STORE-TABLE. /\*Will have total 15 data items\*/**

**05 WS-STORE-ID OCCURS 3 TIMES.**

**10 WS-PRD PIC X(10) OCCURS 5 TIMES.**

**01 WS-SYS-TIME PIC 9(8).**

**01 X PIC 9(1).**

**01 Y PIC 9(1).**

**PROCEDURE DIVISION.**

**ACCEPT WS-SYS-TIME FROM TIME.**

**PERFORM PARA-1 VARYING X FROM 1 BY 1 UNTIL X>3**

**AFTER J FROM 1 BY 1 UNTIL J>5.**

**STOP-RUN.**

**PARA-1.**

**MOVE WS-SYS-TIME TO WS-PRD(I, J).**

**DISPLAY ‘PRODUCT’ I ‘,’ J ‘:’ WS-PRD(I, J).**

**ADD 1 TO WS-SYS-TIME.**

* **INDEXED BY Clause:**

**IDENTIFICATION DIVISION.**

**PROGRAM-ID. INDEXBY.**

**ENVIRONMENT DIVISION.**

**DATA DIVISION.**

**WORKING-STORAGE SECTION.**

**01 WS-STORE.**

**05 WS-PRD OCCURS 3 TIMES INDEXED BY ID.**

**10 WS-ITEM-CODE PIC 9(5).**

**01 WS-ITEM-VAL PIC 9(5) VALUE 12345.**

**PROCEDURE DIVISION.**

**SET ID TO 1.**

**PERFORM PARA-1 UNTIL IDX > 3.**

**STOP-RUN.**

**PARA-1.**

**MOVE WS-ITEM-VAL TO WS-ITEM-CODE(ID).**

**DISPLAY ‘PRODUCT’ WS-ITEM-CODE(ID).**

**ADD 1 TO WS-ITEM-VAL.**

**SET ID UP BY 1. /\*INDEX WILL BE GETTING INCREMENTED\*/**

* **Subscript:**

**IDENTIFICATION DIVISION.**

**PROGRAM-ID. ONEDARRAY.**

**ENVIRONMENT DIVISION.**

**DATA DIVISION.**

**WORKING-STORAGE SECTION.**

**01 WS-STORE.**

**05 WS-PRD OCCURS 3 TIMES.**

**10 WS-ITEM-CODE PIC 9(5).**

**01 WS-ITEM-VAL PIC 9(5) VALUE 12345.**

**01 WS-SUB PIC 9(1). /\*SUBSCRIPT DECLARED\*/**

**PROCEDURE DIVISION.**

**MOVE 1 TO WS-SUB.**

**PERFORM PARA-1 UNTIL WS-SUB > 3.**

**STOP-RUN.**

**PARA-1.**

**MOVE WS-ITEM-VAL TO WS-ITEM-CODE(WS-SUB).**

**DISPLAY ‘PRODUCT’ WS-ITEM-CODE(WS-SUB).**

**ADD 1 TO WS-ITEM-VAL.**

**ADD 1 TO WS-SUB.**

* **Difference between INDEX & SUBSCRIPT:**

|  |  |
| --- | --- |
| **INDEX** | **SUBSCRIPT** |
| Offset or displacement of elements from the beginning | Number of occurrences of the array element |
| INDEXED BY clause is used here | Integer is used to give subscript |
| Index is faster as it refers to the memory location so has better performance | Subscript is slower as it does not refer to the memory location. It refers to position of item. |
| Index is not defined in working-storage section | It is defined in working-storage section |
| Initialized by SET operation | Initialized by VALUE or MOVE statement |
| Index is increased by SET operation | Subscript is increased by ADD operation |

**SEARCH & SEARCH ALL**

**SEARCH:**

* **SEARCH TABLE-NAME [VARYING {IDENTIFIER/INDEX1}]**

**AT END Statement**

**WHEN ConditionPass**

**{Statement/NEXT SENTENCE/CONTINUE}….**

**END-SEARCH.**

**SEARCH ALL:**

* **SEARCH ALL TABLE-NAME**

**AT END Statement**

**WHEN ConditionPass**

**{Statement/NEXT SENTENCE/CONTINUE}….**

**END-SEARCH.**

|  |  |
| --- | --- |
| **SEARCH** | **SEARCH ALL** |
| Is used to find an element or records in Linear manner (Sequential search) | Is used to find an element or record in Binary manner (Binary Search) |
| We can code SEARCH for table which has records either in sorted manner or unsorted manner. | We can code SEARCH ALL for tables which has records which are sorted in the table. |
| It is used mostly for table which does not have records in sorted order. | It is used mostly for table which has records in sorted order. |
| Here index must be initialised | Here, the index does not require initialisation |
| There is no concept of ASCENDING or DESCENDING KEY for SEARCH. | ASCENDING or DESCENDING KEY must be defined in array while using SEARCH ALL. |

**IDENTIFICATION DIVISION.**

**PROGRAM-ID. SEARCHALL.**

**ENVIRONMENT DIVISION.**

**DATA DIVISION.**

**WORKING-STORAGE SECTION.**

**01 WS-STORE.**

**05 WS-PRD OCCURS 3 TIMES ASCENDING KEY IS WS-ITEM-CODE INDEXED BY ID.**

**10 WS-ITEM-CODE PIC 9(5).**

**01 WS-ITEM-VAL PIC 9(5) VALUE 12345.**

**PROCEDURE DIVISION.**

**SET ID TO 1.**

**PERFORM PARA-1 UNTIL IDX > 3.**

**PERFORM SEARCH-ALL-PARA.**

**STOP-RUN.**

**PARA-1.**

**MOVE WS-ITEM-VAL TO WS-ITEM-CODE(ID).**

**DISPLAY ‘PRODUCT’ WS-ITEM-CODE(ID).**

**ADD 1 TO WS-ITEM-VAL.**

**SET ID UP BY 1.**

**SEARCH-ALL-PARA.**

**SEARCH ALL WS-PRD**

**AT END DISPLAY ‘REC NOT FOUND’**

**WHEN WS-ITEM-CODE(ID) = 12347**

**DISPLAY ‘REC FOUND’**

**DISPLAY WS-ITEM-CODE(ID)**

**END-SEARCH.**

**STRING HANDLING**

**String Handling:**

* A string can have multiple operations involved in it:
  + Combining multiple strings into one.
  + To split a string into various other sub strings.
  + To find out the pattern.
  + To replace a particular character or a group of characters with other characters.
* String handling can be divided into 2 types:
  + **STRING** operations: To combine multiple strings into one string.
  + **UNSTRING** operations: To split a string into smaller sub strings.

**STRING:**

* STRING is used to combine multiple strings into one destination string.
* Syntax:

**STRING identifier1/literal1 DELIMITED BY identifier22/literal22/SIZE**

**Identifier2/literal2 DELIMITED BY identifier33/literal33/SIZE**

**INTO identifier3**

**WITH POINTER pointer-id**

**ON OVERFLOW {imperative statement1}**

**NOT ON OVERFLOW {imperative statement2}**

**END-STRING.**

* **DELIMITED BY:**
  + Optional phrase which is used to determine the limits of the string operation.
  + **DELIMITED BY SIZE :** To limit the string based on the size of source string
  + **DELIMITED BY SPACE :** To limit the string based on the initial space
  + **DELIMITED BY other-delimiter :** To limit the string based on a delimiter which is defined in the clause, it can be /,\*,&,’.
* **WITH POINTER:**
  + Optional phrase used to point the starting position of the destination string item.
  + **WITH POINTER N** [Here N is numeric, length of the string > N>=0]
* **ON OVERFLOW:**
  + When the size of the destination string is less that the size of the string which is formed after the concatenation.
* **NOT ON OVERFLOW:**
  + When the size of the destination string is greater than or equal to the size of the string which is formed after the concatenation. This case is successful concatenation.

**IDENTIFICATION DIVISION.**

**PROGRAM-ID. STRINGOP.**

**DATA DIVISION.**

**WORKING-STORAGE SECTION.**

**01 WS-STR1 PIC X(10) VALUE ‘I AM AN/’.**

**01 WS-STR2 PIC X(12) VALUE ‘EXAMPLE OF ‘.**

**01 WS-STR3 PIC X(26) VALUE ‘STRING OPERATION …..’.**

**01 WS-F-STRING PIC X(30).**

**01 WS-POINTER PIC 9(02).**

**PROCEDURE DIVISION.**

**PERFORM STRING-OP.**

**DISPLAY WS-POINTER.**

**DISPLAY WS-F-STRING.**

**STOP RUN.**

**STRING-OP.**

**MOVE 1 TO WS-POINTER.**

**STRING WS-STR1 DELIMITED BY ‘/’**

**WS-STR2 DELIMITED BY SIZE /\*LENGTH IS 12 SO IT’LL TAKE 12 CHARACTERS\*/**

**WS-STR3 DELIMITED BY SPACE**

**INTO WS-F-STRING**

**WITH POINTER WS-POINTER**

**ON OVERFLOW DISPLAY ‘OVERFLOW’**

**NOT ON OVERFLOW DISPLAY ‘NOT OVERFLOW’**

**END-STRING.**

**Output:**

**I AM AN EXAMPLE OF STRING.**

**27**

**UNSTRING:**

* It is opposite of STRING, it splits the string into smaller sub strings.
* Syntax:

**UNSTRING identifier/literal DELIMITED BY [ALL] identifier22/literal22/SIZE**

**INTO identifier1/literal1 [DELIMITER IN] Hold-identifier1 COUNT IN identifier123**

**Identifier2/literal2 [DELIMITER IN] Hold-identifier2 COUNT IN identifier456**

**Identifier3/literal3 [DELIMITER IN] Hold-identifier3 COUNT IN identifier789**

**…..**

**WITH POINTER identifier-for-pointer**

**TALLYING IN tally-identifier**

**ON OVERFLOW {imperative statement1}**

**NOT ON OVERFLOW {imperative statement2}**

**END-UNSTRING.**

* **DELIMITER IN hold-identifier1**
  + This informs the system about the field where the delimiter has to be received.
* **COUNT IN identifier123:**
  + Count field which contains the final count of each character in that particular receiving field.
* **WITH POINTER identifier-for-pointer:**
  + This phrase is used to save the final count after reading each character of the sending fields. Each character read adds 1 to the count.
* **TALLYING IN tally-identifier:**
  + Contains the final value which is equal to the initial value plus the number of receiving string fields.
* **ON OVERFLOW {imperative statement}:**
  + OVERFLOW condition met:
    - When identifier-for-pointer < 1.
    - The identifier-for-pointer > length of the sending field.
    - All the data receiving fields have been used but the sending field still contains unexamined character positions.
* **NOT ON OVERFLOW {imperative statement}:**
  + This is a case of successful UNSTRING operation.

**IDENTIFICATION DIVISION.**

**PROGRAM-ID. UNSTRING.**

**DATA DIVISION.**

**WORKING-STORAGE SECTION.**

**01 WS-STR1 PIC X(02).**

**01 WS-STR2 PIC X(05).**

**01 WS-STR3 PIC X(25).**

**01 WS-DOT PIC X(1) VALUE ‘.’.**

**01 WS-STRING PIC A(30).**

**PROCEDURE DIVISION.**

**MOVE ‘I.AM UNSTRING-EXAMPLE’ TO WS-STRING.**

**PERFORM UNSTRING-OP.**

**DISPLAY WS-STR1.**

**DISPLAY WS-STR2.**

**DISPLAY WS-STR3.**

**STOP RUN.**

**UNSTRING-OP.**

**UNSTRING WS-STRING DELIMITED BY WS-DOT OR ALL SPACES**

**INTO WS-STR1**

**WS-STR2**

**WS-STR3**

**END-UNSTRING.**

**Output:**

**I**

**AM**

**UNSTRING-EXAMPLE**

**=======================**

**IDENTIFICATION DIVISION.**

**PROGRAM-ID. UNSTRING.**

**DATA DIVISION.**

**WORKING-STORAGE SECTION.**

**01 WS-STR1 PIC X(02).**

**01 WS-STR2 PIC X(05).**

**01 WS-STR3 PIC X(25).**

**01 WS-DOT PIC X(1) VALUE ‘.’.**

**01 WS-STRING PIC A(30).**

**01 WS-FLD PIC S9(3).**

**01 HLD-DLTMR-1 PIC X(1) VALUE SPACES.**

**01 CNT-1 PIC 9(2) VALUE ZEROS.**

**01 CNT-2 PIC 9(2) VALUE ZEROS.**

**01 WS-POINTER PIC 9(2).**

**01 WS-STRING PIC A(30).**

**PROCEDURE DIVISION.**

**MOVE ‘I.AM UNSTRING-EXAMPLE’ TO WS-STRING.**

**MOVE 1 TO WS-POINTER.**

**MOVE 1 TO WS-FLD.**

**PERFORM UNSTRING-OP.**

**DISPLAY WS-STR1.**

**DISPLAY WS-STR2.**

**DISPLAY WS-STR3.**

**DISPLAY HLD-DLTMR-1.**

**DISPLAY CNT-1.**

**DISPLAY CNT-2.**

**DISPLAY WS-FLD.**

**DISPLAY WS-POINTER.**

**STOP RUN.**

**UNSTRING-OP.**

**UNSTRING WS-STRING DELIMITED BY WS-DOT OR ALL SPACES**

**INTO WS-STR1 DELIMITER IN HLD-DLTMR-1 COUNT IN CNT-1**

**WS-STR2 COUNT IN CNT-2**

**WS-STR3**

**WITH POINTER WS-POINTER**

**TALLYING IN WS-FLD**

**ON OVERFLOW DISPLAY ‘OVERFLOW’**

**NOT ON OVERFLOW DISPLAY ‘NOT OVERFLOW’**

**END-UNSTRING.**

**Output:**

**NOT OVERFLOW**

**I**

**AM**

**UNSTRING-EXAMPLE**

**.**

**01 /\*. is having 1 character\*/**

**02 /\*AM is having 2 characters\*/**

**00D /\*D is signed positive with actual value of 4. It is shown as D because we have used a signed declaration\*/**

**31**

**INSPECT:**

* INSPECT is used to analyse a string to find the count of occurrence of a particular alphabet or substring or to replace a particular character’s with other character’s
* To count the appearance of an alphabet or character or a sub-string in a String.
  + **INSPECT input-string TALLYING counter FOR ALL SPACES.**
* To replace an alphabet or character or a sub-string in a string.
  + **INSPECT input-sting REPLACING ALL X BY Y.**

**CALL-INTER PROGRAM COMMUNICATION**

**Different types of control passing:**

* **GO TO**
  + This is mostly used to go to specific paragraph.
* **EXIT PROGRAM**
  + Mostly used in the CALLED program (sub-program), it gives the control back to the CALLING program (main-program).
* **GO BACK**
  + If this is coded in CALLED program then it is similar to EXIT PROGRAM. If it is coded in CALLING program, it is similar to STOP RUN. In CALLING program, it is a better practice to code STOP RUN instead of GO BACK.
* **STOP RUN**
  + To stop the execution of the program. It closes all the files used in the program. Code this in CALLING program but never code in CALLED program.

**What is CALL and Different types of CALL:**

* **CALL:**
  + A program can call other program to perform a set of tasks. In this case, the program which calls other program is called CALLING program while the program which is called is known as CALLED program.
  + There are 2 types of calls:
    - **STATIC CALL**
      * **CALL ‘SUB-PROGRAM’**
      * **CALL ‘SUB-PROGRAM’ USING parameter-1, parameter-2,…**
      * Both programs calling and called programs are compiled as separate PDS member have their own load modules, but are link edit together at same time. This is why static call is faster than dynamic call.
      * Static call occupies more storage but it takes less time for execution compared to dynamic calls.
      * In static all the called program is embedded into calling program load module.
    - **DYNAMIC CALL**
      * **CALL ‘WS-VARIABLE’** Here WS-VARIABLE contains name of the program
      * **CALL ‘WS-SUB-PGM’**
      * **CALL ‘WS-SUB-PGM’ USING parameter-1, parameter-2,…**
      * Here WS-SUB-PGM is a working-storage variable

**01 WS-SUB-PGM PIC X(10) VALUE ‘SUB-PGM-NAME’.**

**OR**

**ACCEPT WS-SUB-PGM. From JCL value will be passed**

* + - * In this case both the calling and called programs are compiled as separate PDS members and they are link edit separately.
      * Dynamic call occupies less memory as the compiled module is loaded into the memory only when it is called, but it takes more time for execution.
* While calling a subprogram either it can be simple call without any value being passed to the subprogram or by passing arguments.
  + Without passing any value (Simple CALL).
  + **CALL BY REFERENCE**
    - To refer the same memory location (address) for parameters in calling and called program.
    - If we change the value of any parameter in sub-program, the same value will be reflected in main-program.
    - **USING BY REFERENCE identifier1, identifier2,..**
  + **CALL BY CONTENT**
    - To send the copy of the content of the parameters but not the memory location (address).
    - Any changes to the parameters in sub-program will not reflect in the main-program.
    - **USING BY CONTENT identifier1, identifier2,…**
  + **CALL BY VALUE**
    - It is similar to CALL BY CONTENT but only limited values can be send to the sub-program.
    - We can either pass an integer or one byte alphanumeric value. Not preferred choice.
    - **USING BY VALUE identifier1, identifier2,..**

**Calling program:**

**DATA DIVISION.**

**WORKING-STORAGE SECTION.**

**01 WS-SEQ PIC 9(08).**

**PROCEDURE DIVISION.**

**MAIN-PARA.**

**ACCEPT WS-SEQ FROM TIME.**

**DISPLAY ‘IM CALLING PROG: ‘ WS-SEQ.**

**CALL ‘SUB-PROG’ USING WS-SEQ. /\*IT’S CALL BY REFERENCE BY DEFAULT\*/**

**STOP RUN.**

**Called program:**

**DATA DIVISION.**

**LINKAGE SECTION.**

**01 LS-SEQ PIC 9(08).**

**PROCEDURE DIVISION USING LS-SEQ.**

**MAIN-PARA.**

**MOVE LS-SEQ TO WS-PROD(I,J).**

**ADD 1 TO LS-SEQ.**

**EXIT PROGRAM.**

**USAGE CLAUSE**

* USAGE clause is used to denote the internal representation of data.
* There are 2 types of USAGE clause:
  + **Display (DISPLAY)**
  + **Computational (COMP)**

**DISPLAY:**

* Syntax: (By default USAGE is DISPLAY)

USAGE AS DISPLAY

* The records are stored in ASCII format.
* Ex:

**01 WS-VAR PIC S9(2)V9(2) USAGE IS DISPLAY.** /\*This occupies 4 bytes internally in memory\*/

* Here Sign and decimal does not occupy any extra byte

**COMPUTATIONAL:**

* It is divided into 4 types:
  + **COMP or COMPUTATIONAL**
    - COMP Stored as Binary format
    - COMP can be used only for whole numbers (no decimal numbers should be included here)
    - **01 WS-VAR PIC 9(Z) USAGE IS COMP.**
      * **If Z = 1 to 4, it occupies 2 Bytes**
      * **If Z = 5 to 9, it occupies 4 Bytes**
      * **If Z = 10 to 18, it occupies 8 Bytes**
  + **COMP-1**
    - COMP-1 stored as Hexadecimal form.
    - It is single precision floating point
    - It is internal floating point and **always occupies 4 Bytes in memory.**
    - **It must not have a PIC clause.**
    - **01 WS-VAR USAGE IS COMP-1.**
  + **COMP-2**
    - COMP-1 stored as Hexadecimal form.
    - It is double precision floating point
    - It is internal floating point and **always occupies 8 Bytes in memory.**
    - **It must not have a PIC clause.**
    - **01 WS-VAR USAGE IS COMP-2.**
  + **COMP-3**
    - COMP-3 is stored as PACKED DECIMAL FORM.
    - It is double precision floating point
    - Each digit occupies ½ byte. 1 byte is equal to 2 nibbles so ½ byte is 1 nibble
    - It sign is present, it is stored in the right most nibble.
    - **01 WS-VAR PIC 9(Z) USAGE IS COMP-3.**
      * **To calculate the Number of Bytes:**
        + **Z is even then, No. of bytes = (Z/2)+1**
        + **Z is odd then, No. of bytes = (Z/2)+(1/2)**
    - **01 WS-VAR PIC(5) USAGE IS COMP-3. ->** 3 bytes
    - **01 WS-VAR PIC(4) USAGE IS COMP-3.** -> 3 bytes
  + **COMP-4 /\*It is equivalent to Binary i.e. COMP\*/**