**COBOL (COMMON BUSINESS ORIENTED LANGUAGE)**

* Processing:
  + Batch Process (COBOL+JCL)
  + Online Process (COBOL+CICS)
  + Business Logic will be in COBOL lang only (All the calculations are done in COBOL)
* COBOL is not used for creating any software.
* We can code business logic to maintain the application.
* COBOL is a high-level programming language.
* **Compilation of COBOL program:**
  + Cobol program compilation can be done via JCL or Version control tool (Endevor):
    - When the COBOL source program is run through a compiler, it creates a load module. (IGYCRCTL Utility)
    - The compiler first checks the syntax errors in the COBOL program.
    - Then the compiler converts the COBOL source code into machine level lang in the form of 0's and 1's and kept into Object Module.
    - The Link-Edit process happens, which takes multiple object modules (which are required) and combines them into one single load module.
    - The Load Module will contain the final executable codes which is ready for execution.
    - Then at last it'll check where to store the load module, it can be in private library or system library, that is called a Loadlib.
* **How to code COBOL program:**
  + **DIVISION**
    - **SECTION**
      * **PARAGRAPH**
        + **STATEMENT** 🡪 Command /Instruction to system how to process data
* There are 4 types of **DIVISION** in COBOL:
  + **IDENTIFICATION DIVISION.**
    - Used to provide the in-program documentation details.
    - Out of these 6 paras, only PROGRAM-ID is mandatory.
    - **ID DIVISION.**

**PROGRAM-ID. NAME-OF-PROG.**

**AUTHOR. CODER-NAME.**

**DATE-WRITTEN. DATE.**

**DATE-COMPLIED. DATE.**

**INSTALLATION. APPLICATION-NAME.**

**SECURITY. WARNING-MESSAGE.**

* + **ENVIRONMENT DIVISION.**
    - Used to define the required file in the program.
    - COBOL program types, where ENV Division is required (Batch programs)
      * ONLY COBOL 🡪 NO
      * COBOL + FILE 🡪 YES
      * COBOL + DB2 🡪 NO
      * COBOL + DB2 + FILE 🡪 YES
    - COBOL program types, where ENV Division is not required (Online programs)
      * COBOL + CICS
      * COBOL + CICS + FILE 🡪 We will define the file outside the program called **FCT (File Control Table)**
      * COBOL + CICS + DB2
      * COBOL + CICS + FILE + DB2
    - **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.**

**CONFIGURATION SECTION.**

**INPUT-OUTPUT SECTION.**

**FILE-CONTROL.**

**DEFINE ALL THE FILES (INPUT/OUTPUT)**

* + **DATA DIVISION.**
    - Used to define all variable required in the program.
    - **Variable** is a temporary space allocated in the program to store and process the data. Scope/Life of the variable is till the program ends. Once the program is completed the variable memory is freed up.
    - **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. 🡪** File Variables

**WORKING-STORAGE SECTION. 🡪** Developer variable used for building logics + DB2

**LINKAGE SECTION. 🡪** Different style -> Linkage variable

* + **PROCEDURE DIVISION.**
    - Used to code all the business logic
    - We don’t have pre-defined section/paragraphs
    - We as a developer we can code our own section/paragraphs
    - **PROCEDURE DIVISION.**

**000-MAIN-PARA.**

**……**

**STOP RUN.**

**001-PARA1.**

**……**

* **COBOL code sheet:**
  + Program line should be 1-80 column
  + **1 – 6 :** System auto
  + **7 :** Used for code a comment
  + **8 – 11 :** Area-A, HEADER / DIVISION / SECTION / PARAGRAPH / 01 LEVEL
  + **12 – 72:** Area-B, STATEMENTS
  + **73 – 80:** Change Tags

**COBOL VARIABLE DECLARATION**

* Get the data as input into the program:
  + Define the variable (Memory in the program)
  + Initialize value (Set as default)
  + Get the data from outside of the program
* From input if we are getting
  + 2 values then we need 2 variables in program.
  + 10 values then we need 10 variables in program.
* **In COBOL variables are of 2 types:**
  + **Group variable** : It can be sub-divided into 1 or more elementary item
  + **Elementary variable** : Single element to store the value (cannot be sub-divided)
* **Level Number:**
  + In COBOL all the variables must be prefixed by a number.
  + We have 01 to 49, 66, 77, 88.
  + **01** : Used to define a group and elementary as well
  + **02 - 49** :
    - These variables should be under 01 level number
    - These numbers can be used to define elementary & also sub-groups
    - When adding elementary variable under a sub-group, always increase the level number.
  + **77**: Used for elementary item only.
  + **66 Level:**
    - Used for **RENAMES** clause. It should not have a picture clause.
    - Ex:
      * **01 EMP-REC.**

**05 EMP-ID PIC 9(05).**

**05 EMP-NAME PIC X(20).**

**05 EMP-DOB PIC X(10).**

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

* + **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(03).**

**88 MONDAY VALUE ‘MON’.**

**88 TUESDAY VALUE ‘TUE’.**

**88 WEDNESDAY VALUE ‘WED’.**

* **PIC clause:**
  + **Used for only elementary item**
  + Used to provide data type & size for the memory allocation
* **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 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.

|  |  |  |
| --- | --- | --- |
| **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** |

**ACCEPT DISPLAY STATEMENT**

* Get the data as input into the COBOL program:
  + Define the variable (Temporary memory allocation in the program to store your data)
  + Initialize the variable
  + Get the data into the variable
* Initialize the COBOL variable:
  + To initialize the variable with some default value, so that the variable should not pick the unknown or garbage value.
  + **Can be done in 2 ways:**
    - **VALUE clause**
      * Used in DATA DIVISION
      * 01 VAR-A PIC 9(03) VALUE ZERO.
      * 01 VAR-A PIC X(03) VALUE SPACE.
      * 01 VAR-A PIC X(04) VALUE ‘ANIL’.
      * Switch variable:
        + 01 SWITCH-VAR-A PIC 9(01) VALUE SPACE.
    - **MOVE / INITIALIZE keyword**
      * Used in PROCEDRUE DIVISION
* **Input-Output Statement:**
  + Ways to get the data as input into COBOL program
    - **VALUE** clause - Within the program constant input
    - **ACCEPT** statement - From JCL
    - **READ** statement - From File
    - **SELECT** Query - From DB2 Table
    - **RECEIVE** command - CICS screen
  + Ways to send the data as output:
    - **DISPLAY** command - To SPOOL
    - **WRITE** statement - To File
    - **INSERT**/UPDATE - To Table
    - **SEND** command - CICS Screen
* **ACCEPT Statement:**
  + Used to get an input data to the variable from JCL
  + We should not pass ‘.’ or say decimal value from JCL to COBOL, it’ll throw a S0C7 abend.
  + Syntax:
    - ACCEPT VAR-A.
    - In JCL:
      * //SYSIN DD \*

INPUT-VAL-1

INPUT-VAL-2

/\*

* **DISPLAY Statement:**
  + Used to pass a message/constant or data of a variable
  + The message/value stored in DD name ‘SYSOUT’
  + Syntax:
    - DISPLAY ‘MESSAGE U WANT’.
    - DISPLAY CUST-NAME.
    - DISPLAY ‘Customer Name is: ‘ CUST-NAME.

**COMPUTE PRACTICALS**

* Get the data as input into the COBOL program:
* **PROGRAM: ADD 2 NUMBERS**

**ID DIVISION.**

**PROGRAM-ID. ADDPGM.**

**AUTHOR. ANUBHAV.**

**DATE-WRITTEN. TODAY.**

**ENVIRONMENT DIVISION.**

**CONFIGURATION SECTION.**

**INPUT-OUTPUT SECTION.**

**FILE-CONTROL.**

**DATA DIVISION.**

**FILE SECTION.**

**WORKING-STORAGE SECTION.**

**01 NUM1 PIC 9(02) VALUE ZERO.**

**01 NUM2 PIC 9(02) VALUE ZERO.**

**01 TOTAL PIC 9(02) VALUE ZERO.**

**01 WS-ABENDPGM PIC X(08) VALUE ‘ABENDPGM’.**

**PROCEDURE DIVISION.**

**000-MAIN-PARA.**

**DISPLAY ‘ADDPGM STARTED’.**

**ACCEPT NUM1.**

**ACCEPT NUM2.**

**COMPUTE TOTAL = NUM1+NUM2**

**ON SIZE ERROR**

**DISPLAY ‘ERROR IN COMPUTE’**

**DISPLAY ‘INPUT ARE’ NUM1 ‘,’ NUM2**

**/\* MOVE ‘3200’ TO RETURN-CODE. /\* To change the return code of the PGM\*/**

**CALL WS-ABENDPGM. /\* This will call Abend program \*/**

**DISPLAY ‘TOTAL IS: ‘TOTAL.**

**STOP RUN.**

* **How to find error in while compiling in SPOOL.**
  + Goto SPOOL
  + Open SYSPRINT
    - Compiler Error:
      * **I:** Information (ignore it)
      * **W:** Warning (ignore it)
      * **E:** Error
      * **S:** Severe Error
* **JCL to run the PGM:**
  + **//JOBCARD**

**//STEP1 EXEC PGM=ADDPGM**

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

**//SYSIN DD \***

**30**

**40**

**/\***

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

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

* + **When we code a program:**
    - If Data is valid
      * Process it
    - If Data is Invalid
      * Skip the data 🡪 put it into error file and process next day
      * Abend the program 🡪 display the information properly where program abended

**MOVE STATEMENT**

* Used to pass the data to the variable.
* Ex:
  + MOVE 10 TO VAR-1.
  + MOVE VAR1 TO VAR-2.
* Why do we use MOVE statement?
  + To initialize the variables in procedure division.
  + Moving the data between Input and Output variables.
* **NUMERIC DATA 🡪 Data move into a variable by RIGHT JUSTIFICATION.**
  + MOVE 100 TO VAR-A. PIC 9(03) -> 100
  + MOVE 100 TO VAR-A. PIC 9(04) -> 0100
  + MOVE 100 TO VAR-A. PIC 9(02) -> 00
* **ALPHA NUMERIC DATA** 🡪 **Data move into a variable by LEFT JUSTIFICATION.**
  + MOVE ‘ANIL’ TO VAR-A. PIC 9(04) -> ANIL
  + MOVE ‘ANIL’ TO VAR-A. PIC 9(05) -> ANILb /\*b is blank space\*/
  + MOVE ‘ANIL’ TO VAR-A. PIC 9(03) -> ANI
* **DECIMAL DATA 🡪 Data move into a variable by POINT JUSTIFICATION.**
  + MOVE 123.456 TO VAR-A PIC 9(02)V9(02). -> 23.45
* **Types of MOVE Statement:**
  + **Simple MOVE**
    - MOVE A TO B.
    - MOVE 10 TO A.
    - MOVE SPACE TO A,B,C.
    - MOVE 10 TO A,B,C.
    - Variable should be unique (Source variable name and target variable name should be diff)
    - VARA – 9(03)

VARB – X(03)

Both are valid cases

MOVE VARA TO VARB.

MOVE VARB TO VARA.

* + - Check the data always it is good or bad.
      * VARA -> 123 -> MOVE VARA TO VARB -> VALID
      * VARA -> 1.2 -> MOVE VARA TO VARB -> VALID
      * VARB -> 123 -> MOVE VARB TO VARA -> VALID
      * VARB -> 1.2 -> MOVE VARB TO VARA -> IN-VALID -> ABEND WITH S0C7 (BAD DATA TO NUMERIC)
  + **Group MOVE**
    - 01 CLAIM-DETAILS-IN.

03 CLAIM-NUMBER.

03 CLAIM-AMT.

01 CLAIM-DETAILS-OUT.

03 CLAIM-NUMBER.

03 CLAIM-AMOUNT.

MOVE CLAIM-DETAILS-IN TO CLAIM-DETAILS-OUT.

* + - To MOVE the data b/w 2 groups they must be identical:
      * Same number of variable
      * Same sequence
      * Same type & size
  + **Corresponding MOVE**
    - 01 DATE-YMD.

03 YYYY.

03 MM.

03 DD.

01 DATE-DMY.

03 DD.

03 MM.

03 YYYY.

MOVE DATE-YMD CORR TO DATE-DMY.

* + - Corresponding MOVE is not preferred in the project
  + **MOVE with OF clause**
    - 01 CLAIM-DETAILS-IN.

03 CLAIM-NUMBER.

03 CLAIM-AMT.

01 CLAIM-DETAILS-OUT.

03 CLAIM-NUMBER.

03 CLAIM-AMOUNT.

MOVE CLAIM-NUMBER OF CLAIM-DETAILS-IN

TO

CLAIM-NUMBER OF CLAIM-DETAILS-OUT.

* + - ACCEPT CLAIM-NUMBER OF CLAIM-DETAILS-IN.
    - DISPLAY CLAIM-NUMBER OF CLAIM-DETAILS-OUT.

**CONDITIONAL STATEMENT**

* Used for filtering the data.
* For performance the higher occurrence of the value must be in the 1st condition.
* **There are 2 types of conditional statements:**
  + IF-ELSE Condition
  + EVALUATE Statement
* **IF-ELSE**
  + It is used to check the value and control the execution flow.
  + Ex:
    - IF cond1

S1,S2

END-IF.

* + - IF cond1

S1,S2

ELSE

S3,S4

END-IF.

* + - IF cond1

S1,S2

IF cond2

S3,S4

ELSE

S5,S6

END-IF

ELSE

IF cond3

S7,S8

ELSE

S9,S10

END-IF

END-IF.

* + **PROGRAM: ACCEPT CUSTOMER DATA AND ADD TITLE TO IT BASED ON GENDER**

**ID DIVISION.**

**PROGRAM-ID. CUSTPGM.**

**ENVIRONMENT DIVISION.**

**CONFIGURATION SECTION.**

**INPUT-OUTPUT SECTION.**

**FILE-CONTROL.**

**DATA DIVISION.**

**FILE SECTION.**

**WORKING-STORAGE SECTION.**

**01 CUST-DETAILS-IN.**

**03 CUST-ID PIC X(05).**

**03 CUST-NAME PIC X(10).**

**03 CUST-GENDER PIC X(10).**

**01 CUST-DETAILS-OUT.**

**03 CUST-ID PIC X(05) VALUE SPACE.**

**03 CUST-TITLE PIC X(03) VALUE SPACE**

**03 CUST-NAME PIC X(10) VALUE SPACE.**

**03 CUST-GENDER PIC X(10) VALUE SPACE.**

**01 WS-ABENDPGM PIC X(08) VALUE ‘ABENDPGM’.**

**LINKAGE SECTION.**

**PROCEDURE DIVISION.**

**000-MAIN-PARA.**

**DISPLAY ‘PROGRAM STARTED’.**

**MOVE SPACE TO CUST-ID OF CUST-DETAILS-IN**

**CUST-NAME OF CUST-DETAILS-IN**

**CUST-GENDER OF CUST-DETAILS-IN.**

**ACCEPT CUST-ID OF CUST-DETAILS-IN.**

**ACCEPT CUST-NAME OF CUST-DETAILS-IN.**

**ACCEPT CUST-GENDER OF CUST-DETAILS-IN.**

**IF CUST-GENDER OF CUST-DETAILS-IN = ‘MALE’**

**MOVE ‘MR’ TO CUST-TITLE OF CUST-DETAILS-OUT**

**ELSE**

**IF CUST-GENDER OF CUST-DETAILS-IN = ‘FEMALE’**

**MOVE ‘MRS’ TO CUST-TITLE OF CUST-DETAILS-OUT**

**ELSE**

**DISPLAY ‘INVALID GENDER INPUT’**

**CALL WS-ABENDPGM**

**END-IF**

**END-IF.**

**MOVE CUST-ID OF CUST-DETAILS-IN**

**TO CUST-ID OF CUST-DETAILS-OUT.**

**MOVE CUST-NAME OF CUST-DETAILS-IN**

**TO CUST-NAME OF CUST-DETAILS-OUT.**

**MOVE CUST-GENDER OF CUST-DETAILS-IN**

**TO CUST-GENDER OF CUST-DETAILS-OUT.**

**DISPLAY ‘CUSTOMER DETAILS WITH TITLE’ CUST-DETAILS-OUT.**

**STOP RUN.**

* **EVALUATE**
  + It is used to check the value and control the execution flow.
  + Ex:
    - **EVALUATE TRUE**

**WHEN GENDER=MALE**

**…**

**WHEN GENDER=FEMALE**

**…..**

**WHEN OTHER**

**….**

**END-EVALUATE.**

* + - **EVALUATE GENDER**

**WHEN MALE**

**…**

**WHEN FEMALE**

**…..**

**WHEN OTHER**

**….**

**END-EVALUATE.**

**PERFORM STATEMENT**

* **CONTINUE & NEXT SENTENCE:**
  + Do nothing statements
  + **CONTINUE** **:** The control will jump to next sentence – After nearest END-IF
  + **NEXT SENTENCE** **:** The control will jump to next sentence – After nearest ‘.’ (period)
* **Control sequence flow:**
  + **STOP RUN :** Used to pass the control out of the COBOL main pgm to operating system
  + **EXIT PROGRAM :** Used to pass the control out of the COBOL sub pgm, and send control to main pgm
  + **GOBACK :** Used to pass the control out of the COBOL pgm, can be used in any program as it sends control back to from where it came
* **Control the program flow within the program:**
  + **PERFORM**
    - Used to pass the control to paragraph for execution
    - To process a logic on a record
    - Used to create loops as well
    - **There are 2 types of PERFORM:**
      * **Inline PERFORM**
        + The statement and logic are coded in b/w PERFORM and END-PERFORM
        + The logic cannot be executed from other parts of the program
        + **PERFORM**

**…**

**…**

**END-PERFORM.**

* + - * **Outline PERFORM**
        + The statement of logic is coded in a paragraph, control is passed to it.
        + The logic can be executed from other parts of the program
        + **PERFORM X-PARA.**

**X-PARA.**

**…..**

**…..**

**…..**

* + - **PERFORM-LOOP**
      * **Execute only 1 time**
        + PERFORM

…

…

END-PERFORM.

* + - * + PERFORM X-PARA.
      * **Execute N number of times**
        + PERFORM N TIMES

…..

…..

END-PERFORM

* + - * + PERFORM X-PARA N TIMES.
      * **Condition (Varying)**
        + Used when we don’t know the loop count
        + COND -> TRUE (Loop EXIT) | FALSE (Loop continue)
        + PERFORM UNTIL CONDITION-IS-TRUE\*\*
        + PERFORM X-PARA UNTIL CONDITION-IS-TRUE\*\*

X-PARA.

…

…

…

* + - **PERFORM-THRU**
      * Used to execute multiple paragraphs together
  + **GOTO**
    - Used to pass the control to paragraphs for execution
    - To skip a logic for a record
    - Cannot be used for loop
  + **EXIT**
    - End the paragraph -EXIT the para statement
* **How to code a project style program:**
  + **Top-Down Approach:** Starts from main para and
  + **Modular coding/paragraphs coding:** To split a big logic into multiple paragraphs
  + Syntax:
  + **PROCEDURE DIVISION.**

**000-MAIN-PARA.**

**DISPLAY ‘PGM STARTED’**

**PERFORM 1000-INITIALIZE-PARA THRU 1000-EXIT.**

**PERFORM 2000-GET-INPUT-PARA THRU 2000-EXIT. /\*GET ONLY 1 RECORD\*/**

**PERFORM 3000-PROCESS-PARA THRU 3000-EXIT**

**UNTIL END-OF-INPUT = ‘YES’.**

**CLOSE THE FILES.**

**DISPLAY ‘TOTAL INPUT REC’ WS-IN-REC-CNT.**

**DISPLAY ‘TOTAL OUTPUT REC’ WS-OUT-REC-CNT.**

**STOP RUN.**

**1000-INITIALIZE-PARA.**

1. **INITIALIZE ALL NUMERIC VARIABLE.**
2. **SET ON/OFF THE SWITCH VARIABLE. /\*T/F, Y/N, 0/1\*/**
3. **GET THE SYSTEM/REQUIRED DATES.**
4. **OPEN THE FILES AS REQUIRED.**
5. **BUILD THE OCCUES TABLES (LOOK-UP)**
6. **GET 1ST REC HERE ONLY**

**1000-EXIT.**

**EXIT.**

**2000-GET-INPUT-PARA.**

1. **ACCEPT OR READ FOR FILE.**
2. **INCREASE THE INPUT VARIABLE COUNTER.**

**2000-EXIT.**

**EXIT.**

**3000-PROCESS-PARA.**

1. **PERFORM 3100-AUDIT-PARA THRU 3100-EXIT**
2. **PERFORM 3200-LOGIC-PARA THRU 3200-EXIT**
3. **PERFORM 3300-CAL-PARA THRU 3300-EXIT**
4. **PERFORM 3400-MOVE-IN-OUT THRU 3400-EXIT**
5. **PERFORM 3500-SEND-OUTPUT THRU 3500-EXIT**
6. **PERFORM 2000-GET-INPUT-PARA THRU 2000-EXIT**

**3000-EXIT.**

**EXIT.**

**3200-LOGIC-PARA.**

**PERFORM 3210-CHECK-FLD1-PARA THRU 3210-EXIT**

**PERFORM 3220-CHECK-FLD2-PARA THRU 3220-EXIT**

* + **Code a program:**
    - Credit card program
    - Accept the multiple record as input
    - Audit the data
      * Check if the variable/input are given or not
      * Remove the duplicates
      * If valid data 🡪 Output
    - Variable
      * Data variable 🡪 Input file
      * Logic variable 🡪 Developer variable
        + Looping variable
        + Validation switch
        + Input count / Output count
        + Date variable
        + Hold data variable
  + **ID DIVISION.**

**PROGRAM-ID. CLAIMPGM.**

**ENVIRONMENT DIVISION.**

**INPUT-OUTPUT SECTION.**

**FILE CONTROL.**

**DATA DIVISION.**

**FILE SECTION.**

**WORKING-STORAGE SECTION.**

**01 CARD-DETAILS-IN.**

**03 CARD-NUMBER PIC X(05).**

**03 CUST-NAME PIC X(10).**

**03 CARD-LIMIT PIC S9(05).**

**03 CARD-DUE-AMT PIC S9(05).**

**01 CARD-DETAILS-OUT.**

**03 CARD-NUMBER PIC X(05) VALUE SPACE.**

**03 CUST-NAME PIC X(10) VALUE SPACE.**

**03 CARD-LIMIT PIC S9(05) VALUE ZERO.**

**03 CARD-DUE-AMT PIC S9(05) VALUE ZERO.**

**01 WS-ANY-MORE-INPUT PIC X(01) VALUE SAPCE.**

**01 WS-VALID-INPUT PIC X(01) VALUE SPACE.**

**01 WS-IN-REC-CNT PIC 9(02) VALUE ZERO.**

**01 WS-OUT-REC-CNT PIC 9(02) VALUE ZERO.**

**PROCEDURE DIVISION.**

**000-MAIN-PARA.**

**PERFORM 100-INITIAL-PARA THRU 100-EXIT.**

**PERFORM 200-GET-INPUT-PARA THRU 200-EXIT.**

**PERFORM 300-PROCESS-PARA THRU 300-EXIT**

**UNTIL WS-ANY-MORE-INPUT=’N’.**

**DISPLAY ‘TOTAL INPUT REC COUNT’ WS-IN-REC-CNT.**

**DISPLAY ‘TOTAL OUTPUT REC COUNT’ WS-OUT-REC-CNT.**

**STOP RUN.**

**100-INITIAL-PARA.**

**MOVE ZERO TO WS-IN-REC-CNT WS-OUT-REC-CNT.**

**MOVR ‘Y’ TO WS-ANY-MORE-INPUT.**

**MOVE SPACES TO CARD-NUMBER OF CARD-DETAILS-IN**

**CUST-NAME OF CARD-DETAILS-IN.**

**MOVE ZEOR TO CARD-LIMIT OF CARD-DETAILS-IN**

**CARD-DUE-AMT OF CARD-DETAILS-IN.**

**100-EXIT.**

**EXIT.**

**200-GET-INPUT-PARA.**

**ACCEPT CARD-NUMBER OF CARD-DETAILS-IN.**

**ACCEPT CUST-NAME OF CARD-DETAILS-IN.**

**ACCEPT CARD-LIMIT OF CARD-DETAILS-IN.**

**ACCEPT CARD-DUE-AMT OF CARD-DETAILS-IN.**

**ADD +1 TO WS-IN-REC-CNT.**

**200-EXIT.**

**EXIT.**

**300-PROCESS-PARA.**

**MOVE ‘Y’ TO WS-VALID-INPUT.**

**PERFORM 310-AUDIT-INPUT-PARA THRU 310-EXIT.**

**IF WS-VALID-INPUT=’Y’**

**PERFORM 340-MOVE-DISPLAY-PARA THRU 340-EXIT**

**END-IF.**

**ACCEPT WS-ANY-MORE-INPUT.**

**IF WS-ANY-MORE-INPUT = ‘Y’**

**PERFORM 200-GET-INPUT-PARA THRU 200-EXIT.**

**END-IF.**

**300-EXIT.**

**EXIT.**

**310-AUDIT-INPUT-PARA.**

**IF CARD-NUMBER OF CARD-DETAILS-IN = SPACE**

**MOVE ‘N’ TO WS-INVALID-INPUT**

**GOTO 310-EXIT.**

**END-IF.**

**IF CUST-NAME OF CARD-DETAILS-IN = SPACE**

**MOVE ‘N’ TO WS-INVALID-INPUT**

**GOTO 310-EXIT.**

**END-IF.**

**IF CARD-LIMIT OF CARD-DETAILS-IN < 0**

**MOVE ‘N’ TO WS-INVALID-INPUT**

**GOTO 310-EXIT.**

**END-IF.**

**IF CARD-DUE-AMT OF CARD-DETAILS-IN < 0**

**MOVE ‘N’ TO WS-INVALID-INPUT**

**END-IF.**

**310-EXIT.**

**EXIT.**

**340-MOVE-DISPLAY-PARA.**

**MOVE CARD-NUMBER OF CARD-DETAILS-IN TO**

**CARD-NUMBER OF CARD-DETAILS-OUT.**

**MOVE CUST-NAME OF CARD-DETAILS-IN TO**

**CUST-NAME OF CARD-DETAILS-OUT.**

**MOVE CARD-LIMIT OF CARD-DETAILS-IN TO**

**CARD-LIMIT OF CARD-DETAILS-OUT.**

**MOVE CARD-DUE-AMT OF CARD-DETAILS-IN TO**

**CARD-DUE-AMT OF CARD-DETAILS-OUT.**

**DISPLAY ‘VALID CARD DETAILS ARE’ CARD-DETAILS-OUT.**

**ADD +1 TO WS-OUT-REC-CNT.**

**340-EXIT.**

**EXIT.**

**JCL INPUT:**

**//SYSIN DD\***

**C1111**

**ABCD**

**10000**

**04000**

**Y**

**C2222**

**EFGH**

**20000**

**01000**

**N**

**/\***

**FILE HANDLING**

* Data in mainframe can be stored in file or database.
* File is a collection of records and Records are collection of fields (File layout is a structure of the file record)
* File can be OPEN, READ, WRITE, REWRITE, DELETE & CLOSE
* How to code File pgm:

**ID DIVISION. /\*No change**

**ENVIRONMENT DIVISION.**

**INPUT-OUTPUT SECTION.**

**FILE-CONTROL.**

**SELECT FILE-NAME ASSIGN TO DD-NAME-1**

**ORGANIZATION IS SEQUENTIAL**

**ACCESS MODE IS SEQUENTIAL**

**FILE STATUS IS WS-STATUS-VARIABLE-1.**

**SELECT FILE-NAME ASSIGN TO DD-NAME-2**

**ORGANIZATION IS SEQUENTIAL**

**ACCESS MODE IS SEQUENTIAL**

**FILE STATUS IS WS-STATUS-VARIABLE-2.**

**DATA DIVISION.**

**FILE SECTION.**

**FD FILE-NICK-NAME-1**

**LABEL RECORD ARE STANDARD.**

**01 ACCOUNT-DETAILS.**

**03 ACCT-NUM PIC X**

**03 CUST-NAME PIC X**

**03 CUST-ADD.**

**05 CITY PIC X**

**05 STATE PIC X**

**WORKING-STORAGE SECTION.**

**01 WS-STATUS-VARIABLE-1 PIC X(02) VALUE SPACE.**

**01 WS-STATUS-VARIABLE-2 PIC X(02) VALUE SPACE.**

**PROCEDURE DIVISION.**

* **ENVIRONMENT DIVISION.**
  + **SELECT** : Used to tell file nick name and JCL DD name.
  + **ORGANIZATION** : Use to tell the type of the file, how data will be arranged
    - QSAM 🡪 PS File 🡪 **SEQUENTIAL** (Allow duplicates, No sorted order)
    - VSAM 🡪 KSDS Cluster 🡪 **INDEXED** (No duplicates, Sorted order)
  + **ACCESS MODE**  : Use to tell how to access the data in the file
    - **SEQUENTIAL** : Read the record 1 by 1 from top to bottom -> PS & KSDS
    - **RANDOM** : Read a particular record based on key value -> KSDS only
    - **DYNAMIC** : Random + Sequential -> KSDS only
  + **FILE STATUS** : This is a RETURN CODE that tells file operation result
    - We need to declare the variable that will hold the RC
    - Check file status code: (Interview questions)
      * **00** : successful
      * **10** : End of file (EOF)
      * **21, 22, 23, 24** : KSDS error
      * **39** : File attribute mismatch when you open file
      * **41, 42, 43** : Logic mistake
      * **35/96** : Invalid/Missed the DDNAME in JCL
      * **97** : KSDS Index issue that needs to be verified
* **DATA DIVISION**
  + **FILE SECTION**
    - In file section VALUE clause is not allowed
* **PROCEDURE DIVISION**
  + **OPEN** the file
    - Access the - READ/WRITE on file
  + **CLOSE** the file
  + We don’t do update on file (Rewrite)
  + We cannot delete a record
  + When we READ a file, we get only 1 record from the file and that is stored in the file layout
* A
* A
* A
* A
* A