JCL NOTES

JCL - Procedures or PROC

Some programs and tasks require a larger amount of JCL than a user can easily enter. JCL for these functions can be kept in procedure libraries.

A procedure library member contains only part of the JCL for a given task-usually the fixed, unchanging part of JCL. The user of the procedure supplies the variable part of the JCL for a specific job. In other words, a JCL procedure is like a macro.

For jobs that you run frequently or types of jobs that use the same job control, prepare sets of job control statements, called procedures.

In other words, Set of Job control statements that are frequently used are defined separately as a procedure and it can be invoked as many times as we need from the job. The use of procedures helps in minimizing duplication of code and probability of error.

Statements which are not Allowed in a Procedure

You can place most statements in a procedure, but there are a few exceptions. Some of these exceptions are:

1. The JOB statement and JES2/JES3 Control statements.
2. The JOBCAT and JOBLIB statement.
3. An instream procedure (an internal PROC/PEND pair)
4. SYSIN DD *, DATA statements

There are two types of procedures available in JCL,

1. In-stream procedures.
2. Cataloged procedures.

In-Stream Procedure:

When the procedure is coded within the same JCL member, it is called an Instream Procedure.

An in-stream procedure must begin with a PROC statement, end with a PEND statement, and include only the following other JCL statements: CNTL, comment, DD, ENDCNTL, EXEC, IF/THEN/ELSE/ENDIF, INCLUDE, OUTPUT JCL, and SET.

You must observe the following restrictions regarding in-stream procedures:

- Do not place any JCL statements (other than the ones listed here) or any JES2 or JES3 control statements in the procedure.
- Do not define one in-stream procedure within another, that is, do not use nested in-stream procedures. Refer to Nested procedures for information on methods for nesting procedures.
- Do not use an in-stream procedure if the procedure will be run as a started job under the MASTER subsystem, that is, includes a JOB statement and is started via a START command such as S membername,SUB=MSTR.

Cataloged Procedures:

When the procedure is separated out from the JCL and coded in a different data store, it is called a Cataloged Procedure. A PROC statement is not mandatory to be coded in a cataloged procedure.

The library containing cataloged procedures is a partitioned data set (PDS) or a partitioned data set extended (PDSE).

When a cataloged procedure is called, the calling step receives a copy of the procedure. therefore, a cataloged procedure can be used simultaneously by more than one job.

If you are modifying a cataloged procedure, do not run any jobs that use the procedure during modification.

JCL - In-Stream Procedure

Let's see an example to understand In-Stream procedure,

Example 1:

```
//TESTJCL1 JOB 100,CLASS=C,MSGCLASS=Y,NOTIFY=&SYSUID
//*
```

```
//INSMPROC   PROC                   //*START OF PROCEDURE
//PROC1            EXEC PGM=SORT
//SORTIN     DD DSN=HLQ.FILE.INPUT,DISP=SHR
//SORTOUT    DD DSN=HLQ.FILE.OUTPUT,DISP=SHR
//SYSOUT     DD SYSOUT=*
//SYSIN            DD DSN=HLQ.APP.CARDLIB(SORTCARD),DISP=SHR
//           PEND                   //*END OF PROCEDURE
//*
//STEP1      EXEC INSMPROC
//*
//STEP2      EXEC INSMPROC
```

Explanation:

In the above example, the procedure INSMPROC is called in STEP1 and STEP2.

As discussed in previous chapter, An in-stream procedure INSMPROC coded begin with a PROC statement, end with a PEND statement.

Also, If the called procedure is an in-stream procedure, the system retrieves it from the job input stream. You must place the in-stream procedure before the EXEC statement that calls it.

JCL - Cataloged Procedure

Let see an example to understand cataloged procedure,

Example:

```
//TESTCTLG JOB 100,CLASS=C,MSGCLASS=Y,NOTIFY=&SYSUID
//*
//MYLIBS1  JCLLIB  ORDER=MYCOBOL.BASE.PROCLIB
//*
//STEP1 EXEC CTLGPROC
//*
```

In this example job:

Your JCL begins with a JOB statement that names your job TESTCTLG.

The first step in TESTCTLG, named STEP1, Identifies CTLGPROC as the JCL procedure to be run.

Here, the procedure CTLGPROC is present in MYCOBOL.BASE.PROCLIB library. See below,

```
//CTLGPROC PROC
//*
//PROC1             EXEC PGM=SORT
//SORTIN     DD DSN=HLQ.FILE.INPUT,DISP=SHR
//SORTOUT    DD DSN=HLQ.FILE.OUTPUT,DISP=SHR
//SYSOUT     DD SYSOUT=*
//SYSIN             DD DSN=HLQ.APP.CARDLIB(SORTCARD),DISP=SHR
//*
```

Have you noted JCLLIB statement in example JCL?

//MYLIBS1 JCLLIB ORDER=MYCOBOL.BASE.PROCLIB

The JCLLIB statement allows you to code and use procedures and INCLUDE groups in a private library without the need to use system procedure libraries.

You can code only one JCLLIB statement per job.

Let us assume that the system default procedure library includes SYS1.PROCLIB only. If you do not specify the JCLLIB statement, then the system searches only SYS1.PROCLIB. (Using a procedure describes how the system determines the default procedure library.)

So, We need JCLLIB statement to override the system proc library.

For example,

```
//MYJOB1   JOB     ...
//MYLIBS1  JCLLIB  ORDER=MYCOBOL.BASE.PROCLIB
//S1       EXEC    PROC=MYPROC1
```

Now the system searches the libraries for procedure MYPROC1 in the following order:

1. MYCOBOL.BASE.PROCLIB
2. SYS1.PROCLIB

Hope you understand why we added a JCLLIB statement in the example JCL. Proc CTLGPROC is present in MYCOBOL.BASE.PROCLIB library.

JCL - Nested Procedure

Cataloged and in-stream procedures can invoke other procedures (up to 15 levels of nesting). In a procedure, an EXEC statement can invoke another procedure, which can contain an EXEC statement to invoke another procedure, and so on.

We cannot code an instream procedure within a cataloged procedure.

The following example shows how procedures can be nested:

```
Procedure C:
    //C       PROC
    //CS1     EXEC  PGM=GHI
               .
    //        PEND

 Procedure B:
    //B       PROC
    //BS1     EXEC  PROC=C
               .
    //BS2     EXEC  PGM=DEF
               .
    //        PEND

 Procedure A:
    //A       PROC
    //AS1     EXEC  PROC=B
               .
    //AS2     EXEC  PGM=ABC
               .
    //        PEND

 Job Stream:
    //JOB1    JOB
    //STEP1   EXEC  PROC=A
               .
    //STEP2   EXEC  PGM=JKL
               .
               .
```

The following statements are equivalent to the nested procedures shown above.

```
//JOB1    JOB
   //CS1     EXEC  PGM=GHI
              .
   //BS2     EXEC  PGM=DEF
```

```
              .
   //AS2      EXEC   PGM=ABC
              .
   //STEP2    EXEC   PGM=JKL
              .
              .
```

We don't use nested procedure often in real time projects. We mostly use cateloged procedure.

JCL - Procedure Modification

Procedure should be generic so that it can easily be used by the multiple JOBS by simple overrides. During the invoking of procedures in the JOB, one can do the following.

1. **Override:** Change the dataset names or parameters that are already coded in the procedure.
2. **Addition:** Add new datasets or parameters in the already existing steps of the procedure.
3. **Nullify:** Omit the datasets or parameters that are already coded in procedure.

If a keyword parameter is to override the parameter or be added to every EXEC statement in the procedure, code the parameter in the usual way.

For example, the ACCT parameter applies to all steps:

```
//STEP1  EXEC  PROC=RPT,ACCT=5670
```

If the keyword parameter is to nullify the parameter on every EXEC statement in the procedure, code it without a value following the equal sign.

For example, the ACCT parameter is nullified in all steps:

```
//STEP2  EXEC  PROC=RPT,ACCT=
```

If the keyword parameter is to override the parameter or be added to only one EXEC statement in the procedure, code .procstepname immediately following the keyword. The procstepname is the name field of the procedure EXEC statement containing the keyword parameter to be overridden.

For example, the ACCT parameter applies to only step PSTEPWED:

```
//STEP1  EXEC  PROC=RPT,ACCT.PSTEPWED=5670
```

If the keyword parameter is to nullify the parameter on only one EXEC statement in the procedure, code it with the procstepname.

For example:

```
//STEP2  EXEC  PROC=RPT,ACCT.PSTEPTUE=
```

Note: The override, nullification, or addition applies only to the current execution of the job step; the procedure itself is not changed.

Example: PROC COBCLG has a statement

```
//COB EXEC PGM=IGYCRCTL,REGION=400K
```

Let's see JCL below to Override, Add and nullify the parameter in PROC,

```
//*Overrides the value of 400K
//EXEC COBCLG,REGION.COB=1M

//*Adds 10 seconds time limit for COB step.
//EXEC COBCLG,TIME.COB=(0,10)

//*Nullifies 400K region request. Default region will be
allocated now.
//EXEC COBCLG,REGION.COB=
```

JCL - Procedure Modification Examples

Example 1: Overriding a DSN

PROC has an input data set and this needs to be modified while running the job. So below override can be used.

```
//PROC1 PROC
//*****************************************************************
//* OVERRIDE EXAMPLES
//*****************************************************************
//PRC001 EXEC PGM=IEBGENER
//SYSPRINT DD SYSOUT=A
//SYSIN DD DUMMY
//SYSUT1 DD DSN=PRDG.CST.IEBGNR.IN,DISP=SHR
//SYSUT2 DD DSN=PRDG.CST.IEBGNR.O1T,DISP=(NEW,CATLG,DELETE),
// UNIT=&UNIT,SPACE=(CYL,(1,1),RLSE),
// DCB=(RECFM=FB,LRECL=100,BLKSIZE=1000)
//*
// PEND
```

JOB with Override DSN

Here the JOB gets executed with the new data set given in the JOB instead of the dataset given in the PROC

```
//JOBTST1 JOB MSGLEVEL=(1,1),MSGCLASS=X,NOTIFY=JCLSCAN
//*
//***********************************************************
//* OVERRIDING DATASET
//***********************************************************
//STEP1 EXEC PROC1,UNIT=SYSDB
//PRC001.SYSUT1 DD DSN=PRDG.CST.IEBGNR.IN1,DISP=SHR
```

If the PROC has only one step then Procstep name is not required. It can be coded as below

```
//STEP1 EXEC PROC1,UNIT=SYSDB
//SYSUT1 DD DSN=PKMXG.TCPR.IEBGNR.IN1,DISP=SHR
```

If there are multiple steps and in each step the DD name is same then ignoring proc step name overrides only the first step values. For example:

PROC: Proc name - PROC1

```
//PRC001 EXEC PGM=PGM1
//INFIL DD DSN=TEST.IN.F001,DISP=SHR
.
.
//PRC002 EXEC PGM=PGM2
//INFIL DD DSN=TEST.IN.F002,DISP=SHR
.
.
```

JOB:

In this case only the INFIL of PRC001 gets overridden.

```
//STEP1 EXEC PROC1,UNIT=SYSDB
//INFIL DD DSN=TST.IN.DB.FILE,DISP=SHR
```

Example 2: Overriding DSN sub parameters

UNIT, Volume, Space etc parameters specified against the DSN can also be modified from the job.

PROC:

```
//PROCT PROC
//*********************************************************************
//* OVERRIDING THE DATASET SUB PARAMETERS
//***************************************************************
//PRC001 EXEC PGM=TRKTST
//SYSPRINT DD SYSOUT=A
//SYSIN DD DUMMY
//INFIL1 DD DSN=TTRKY.OVRD.IN.F01,DISP=SHR
//OTFIL1 DD DSN=TTRKY.OVRD.OT.F01,DISP=(NEW,CATLG,DELETE),
// UNIT=SYSDA,SPACE=(CYL,(1,1),RLSE),
// DCB=(RECFM=FB,LRECL=100,BLKSIZE=1000)
//*
// PEND
```

Sub parameters given in the override step would be effective. If you Omit the DSN sub parameters(DISP, UNIT, DCB etc), then these will not be nullified instead it takes this data from the PROC if they exist.

```
//STP01 EXEC PROCT
//PRC001.OTFIL1 DD DSN=TTRKY.OVRD.OT.F01,DISP=(MOD,CATLG,DELETE),
// UNIT=SYSDB,SPACE=(CYL,(2,5),RLSE),
// DCB=(RECFM=FB,LRECL=90,BLKSIZE=9000)
```

If you need to override only one parameter in the DD statement, then you don't need to add all the parameters.

Example 3: Nullifying the Parameters

IN case the parameters specified in the PROC needs to be nullified then code it with out any value. It nullifies the specified sub parameters in all of the PROC steps.

```
//STP01 EXEC PROCT,UNIT=,
```

If the parameters should be nullified in any particular step then step name should be mentioned.

```
//STP01 EXEC PROCT,UNIT.PRC001=
```

Multiple parameters can be nullified.

```
//STP01 EXEC PROCT,UNIT=,ACCT=
```

Example 4: Adding DSN SUb parameters

```
//STP01 EXEC PROC1,ACCT.PRC001=2516
```

Adding and Nullifying parameters can be given together.

```
//STP01 EXEC PROC1,ACCT.PRC001=2516,UNIT=
```

Example 5: Supplying In-Stream data

```
//STP01 EXEC PROC1,UNIT=SYSDB
//PRC001.SYSN DD *
.
.
/*
```

Example 6: Overriding the Concatenated data sets:

1. To override the first dsn in the concatenated list, then coding only the first one is enough.

2. To override other than the first one, then all the DSN needs to be mentioned.

3. Order of the overriding DD statement must be same as the proc dd order.

Below PROC has 4 concatenated data sets as input.

```
//PRO11 PROC
//*********************************************************************
//* OVERRIDING THE CONCATENATED DATA SETS
//****************************************************************
//*
//PRC002 EXEC PGM=IEBGENER
//SYSPRINT DD SYSOUT=A
//SYSIN DD DUMMY
//SYSUT1 DD DSN=PRT.CRX.IEBGNR.IN,DISP=SHR
//       DD DSN=PRT.CRX.IEBGNR.IN1,DISP=SHR
//       DD DSN=PRT.CRX.IEBGNR.IN2,DISP=SHR
//       DD DSN=PRT.CRX.IEBGNR.IN3,DISP=SHR
//SYSUT2 DD DSN=PRT.CRX.IEBGNR.O2T,DISP=(NEW,CATLG,DELETE),
// UNIT=&UNIT,SPACE=(CYL,(1,1),RLSE),
// DCB=(RECFM=FB,LRECL=090,BLKSIZE=9000)
//*
// PEND
```

To override the First parameter –> Only first DSN is enough

```
//STEP1 EXEC PROC2,UNIT=SYSDA
//SYSUT1 DD DSN=PRT.CRX.IEBGNR.IN4,DISP=SHR
```

To override the 2nd DSN –> All other DD statements are must but the DSN can be left blank so that the Original DSN mentioned in the PROC would be taken.

```
//STEP1 EXEC PROC2,UNIT=SYSDA
//SYSUT1 DD
//       DD DSN=PRT.CRX.IEBGNR.IN4,DISP=SHR
//       DD
//       DD
//
```

Note: PGM parameter can not be modified but this can be achieved by symbolic parameters.

JCL - Symbolic Parameters

Using Symbolic parameters in JCL/PROC is a good practice to modify any PROC parameters from the JOB.

This method gives the flexibility of using different parameters in Different jobs which are using a single PROC.

It is not only used to modify the proc parameters, it can also be used in the stand alone JOB.

While coding symbolic parameters, do not use KEYWORDS, PARAMETERS or SUB-PARAMETERS as symbolic names. Example: Do not use TIME=&TIME but yes you can use TIME=&TM and it is assumed as a right way of coding symbolics.

User-defined symbolic parameters are called JCL Symbols. There are certain symbols called system symbols, which are used for logon job executions. The only system symbol used in batch jobs by normal users is &SYSUID and this is used in the NOTIFY parameter in the JOB statement.

Let's see few examples below,

Example 1: Modifying the PROC parameters using Symbolic Parameters.

SYMB is the proc where its UNIT parameter is not given and instead used a Symbolic parameter.

PROC:

```
//SYMB PROC
//*************************************************************
//* Symbolic Parameters
//*************************************************************
//SYM001 EXEC PGM=PRG01
//SYMF01 DD DSN=SYM.TST.INF01,DISP=SHR
//SYMF02 DD DSN=SYM.TST.OUTF01,DISP=(NEW,CATLG,DELETE),
//    UNIT=&UNIT,SPACE=(CYL,(1,1),RLSE),
//    DCB=(RECFM=FB,LRECL=100,BLKSIZE=1000)
//*
// PEND
```

JOB to Override the UNIT value,

```
//PKMXTST1 JOB MSGLEVEL=(1,1),MSGCLASS=X,NOTIFY=USER1
// JCLLIB ORDER=(SYM.TR.SOURCE)
//*********************************************************
//* JOB REPLACE UNIT PARAMETER
//*********************************************************
//STEP1 EXEC PROC1,UNIT=SYSDA
```

Example 2: Modifying the file name HLQ using Symbolic parameters.

Below proc is written to execute a program TRCX99 where it has one input file and an output file. Input & Output files are not fixed here & it changes based on the Environment from where it triggered.

For example if it gets triggered from Credit card dept. then the File names should start with CR99 and if it is from the debit card dept. then the file names should start with DB99.

So in this case instead of writing two procs one for Credit & another ine for Debit, symbolic parameters can be used to give the file names dynamically.

PROC:

```
//SYMB02 PROC
//*******************************************************************
//* OVERRIDING THE DATASET SUB PARAMETERS
//*****************************************************************
//*
//PRC001 EXEC PGM=TRCX99
//SYSPRINT DD SYSOUT=A
//TRCIN DD DSN=&HLQ..TR.X99.IN,DISP=SHR
//TRCOUT DD DSN=&HLQ..TR.X99.REP,DISP=(NEW,CATLG,DELETE),
// UNIT=&UNIT,SPACE=(CYL,(4,5),RLSE),
// DCB=(RECFM=FB,LRECL=210,BLKSIZE=2100)
//*
// PEND
```

JOB to pass the values:

```
//CRJOB JOB MSGLEVEL=(1,1),MSGCLASS=X,NOTIFY=CRUSER
//STEP1 EXEC SYMB02,UNIT=SYSDA,HLQ=CR99
//
```

Notes: In this way not only the HLQ (high Level Qualifier) but any other part of the file name can be modified.

Observe that Symbolic parameter in the file name should contain a (.) dot

Example 3: Modifying Program name using Symbolic Parameters.

PROC:

```
/SYMB03 PROC
//****************************************************************
//* Symbolic parameters
//****************************************************************
//SYM001 EXEC PGM=&PROG
// . . .
// PEND//
```

JOB to supply the value:

```
//CRJOB JOB MSGLEVEL=(1,1),MSGCLASS=X,NOTIFY=CRUSER
//STEP1 EXEC SYMB03,PROG=TRCX99
//
```

Example 4: Passing the PARM data using Symbolic Parameters.

PROC:

```
//PROC9 PROC
//****************************************
//* Supplying parm
//****************************************
//*
//PRC009 EXEC PGM=&PROG,PARM='&INPARM'
```

JOB to pass the parm data

```
//PRMTST1 JOB MSGLEVEL=(1,1),MSGCLASS=X
// JCLLIB ORDER=(PRTP.TR.SOURCE)
//******************************************************************
//* JOB REPLACE UNIT PARAMETER
//*****************************************************************
//STEP1 EXEC PROC9,PROG=PRM700,INPARM=CR
//
```

In this case parm EXEC statement replaced as,

```
PGM=PRM700,PARM='CR'
```

JCL - Conditional Processing

Before we start, We should know- What is COND in JCL?

1. COND parameter is JCL can be used to skip a step based on the return code from the previous step or steps of the JCL.
2. This COND parameter can also be used to skip a step based on the return code from a previous step or steps in PROC.
3. You can use the COND parameter to run a job even though, any of the previous steps have Abended (Abend means Abnormal Termination).
4. With the COND parameter, you can also run a step only and only if any of the previous steps have Abended.

The 'COND' parameter defines the conditional processing in JCL and this is an important parameter in JCL.

You can set the COND parameter at –

1. **JOB level** –JOB Statement(JOB card).
2. **STEP level** – EXEC statement.

In most cases, It is a good practice to code the COND parameter at STEP Level but based on your requirement, you can code this at JOB level or STEP level or at both the places.

Syntax:

```
COND[.procstepname]=(rc,logical-operator)
or
COND[.procstepname]=(rc,logical-
operator[,stepname][.procstepname])
or
COND=EVEN
```

```
or
COND=ONLY
```

Description of parameters in syntax:

- **rc** : This is the return code
- **logical-operator** : This can be GT (Greater Than), GE (Greater than or Equal to), EQ (Equal to), LT (Lesser Than), LE (Lesser than or Equal to) or NE (Not Equal to).
- **stepname** : This is the job step whose return code is used in the test.

Last two conditions (a) COND=EVEN and (b) COND=ONLY, have been explained below in this chapter.

As discussed earlier, COND can be coded either inside JOB statement or EXEC statement, and in both the cases, it behaves differently as explained below:

Let us take some examples on each type-

Types	**Condition Example**	**Explanation**
Type 1	COND=(0,EQ)	Is return code 0 equals the return code from the previous step? If this condition is true then bypass this step.
Type 2	COND=(4,EQ,STEP10)	Is return code 4 equals the return code from the previous step? if this condition is true then bypass this step.
Type 3	COND=EVEN	Run this steps even though, any of the previous step has Abended.
Type 4	COND=ONLY	Run this step only if any of the previous steps has abended.

Example 1: COND Parameter at JOB level.

If you code a COND parameter at JOB level, then this condition will test against the return code of all the steps of the JCL.

```
//MATEKSD JOB MSGLEVEL=(1,1),NOTIFY=&SYSUID,
//*            COND=(0,NE)
//* EXAMPLE TO SHOW COND AT JOB LEVEL IN JCL
//*
//STEP01   EXEC PGM=CONDPGM1
//STEP02   EXEC PGM=CONDPGM2,COND=(0,EQ)
```

```
//STEP03   EXEC PGM=CONDPGM3,COND=(4,EQ)
```

Explanation:

COND parameter at the JOB level is – COND=(0,NE)

COND=(0 NE) will test – Is return code of 0 not equal to the return code of any of the steps in JCL?

If this is true, then terminate this JCL. So, if any of the steps (i.e. STEP01, STEP02 or STEP03) has a return code of Non-Zero, then this condition will be true and the Job will be terminated.

Example 2: COND Parameter at STEP level.

When COND is coded in EXEC statement of a job step and found to be true, only that job step is bypassed, and execution is continued from next job step.

```
//CNDSAMP JOB CLASS=6,NOTIFY=&SYSUID
//*
//STP01 EXEC PGM=SORT
//* Assuming STP01 ends with RC0.

//STP02 EXEC PGM=MYCOBB,COND=(0,EQ,STP01)
//* In STP02, condition evaluates to TRUE and step bypassed.

//STP03 EXEC PGM=IEBGENER,COND=((10,LT,STP01),(10,GT,STP02))
//* In STP03, first condition fails and hence STP03 executes.
//* Since STP02 is bypassed, the condition (10,GT,STP02) in
//* STP03 is not tested.
```

Example 3: JCL COND=EVEN

If you want to execute a particular step even if any of the previous steps have Abended, then you should use – COND=EVEN in that step.

```
//MATEKSD JOB MSGLEVEL=(1,1),NOTIFY=&SYSUID
//*
//* EXAMPLE TO SHOW COND=EVEN IN JCL
//*
//STEP01   EXEC PGM=CONDPGM1
//STEP02   EXEC PGM=CONDPGM2
```

```
//STEP03 EXEC PGM=CONDPGM4,COND=EVEN
```

Explanation:

COND=EVEN will test – STEP03 will run even if STEP01 or STEP02 has Abended. So, irrespective of whether STEP01 or STEP02 has Abended or not, STEP03 will run.

Example 4: JCL COND=ONLY

If you want to execute a particular step only and only if any of the previous steps have Abended, then you should use – COND=ONLY in that step.

Example:

```
//MATEKSD JOB MSGLEVEL=(1,1),NOTIFY=&SYSUID
//*
//* EXAMPLE TO SHOW COND=ONLY IN JCL
//*
//STEP001  EXEC PGM=ONLY3PGM
//STEP002  EXEC PGM=ONLY2PGM
//STEP003  EXEC PGM=ONLY1PGM,COND=ONLY
```

Explanation:

COND=ONLY will test – Here, STEP03 will run only if STEP01 or STEP02 has Abended. So, if STEP01 or STEP02 did not Abend then STEP03 will not run.

Tips:

- There can be multiple COND Parameter in JCL but a maximum of 8 COND parameter is allowed.
- COND parameter can be used at JOB level or STEP level or at both the places.

JCL - IF/THEN/ELSE/ENDIF Statements

Another approach to control the job processing is by using IF-THEN-ELSE constructs. This gives more flexibility and user-friendly way of conditional processing.

In JCL, IF condition is better than the COND parameter because of 2 reasons –

1. IF condition in JCL is easy to code as compared to the COND parameter.
2. JCL programmers with less experience find COND parameter little confusing and boring and it is easier to read and understand IF-ELSE-END-IF condition in JCL.

Syntax:

```
//[name] IF  [(]relational-expression[)] THEN   [comments]
    .
    .    action when relational-expression is true
    .
//[name] ELSE   [comments]
    .
    .    action when relational-expression is false
    .
//[name] ENDIF   [comments]
```

Following is the description of the used terms in the above IF-THEN-ELSE Construct:

name : This is optional and a name can have 1 to 8 alphanumeric characters starting with alphabet, #,$ or @.

Relational-expression : A relational-expression will have a format: KEYWORD OPERATOR VALUE, where KEYWORDS can be RC (Return Code), ABENDCC (System or user completion code), ABEND, RUN (step started execution). An OPERATOR can be logical operator (AND (&), OR (|)) or relational operator (<, <=, >, >=, <>).

comments: The comments field follows THEN, ELSE, and ENDIF after at least one intervening blank.

The below keywords can include a stepname and procstepname to refine the test to a specific job step.

A Keyword List

Keyword	Purpose
ABEND	Tests for an abnormal end of a program
ABEND	Tests that an abnormal end of a program did not occur
ABENDCC	Examines an ABEND condition code

RC	Examines a return code
RUN	Tests if a job step executed
RUN	Tests if a job step did not execute

Syntax:

```
stepname.procstepname.keyword
```

Example 1: Let us see simple IF condition example,

```
//JOBEXP JOB
//STEP01  EXEC MYPROC01
//COND01  IF RC = 0 THEN
//STEP02  EXEC MYPROC02
//CONDE   ELSE
//STEP03  EXEC MYPROC03
//        ENDIF
```

Explanation:

In this example, After STEP01 is completed we check the Return code of STEP01 step. if return code is zero then STEP02 step will execute else step STEP03 will be executed.

Tips:

```
IF  RC = 0   THEN

IF  STEP1.RC < 12  THEN
```

If you not given the stepname, then the highest return code from all job steps is taken for checking.

ABENDCC - Using ABENDCC we can check System/User completion codes.

```
IF ABENDCC = S0C7 THEN
```

Suppose you want to check error code of particular step, give stepnname.ABENDCC, If you not given the stepname, most recent ABEND code that occured is taken for checking.

ABEND- checks for an abnormal end of a program

```
IF  ABEND  THEN

IF  STEP4.PROCAS01.ABEND = TRUE  THEN
```

If you not given any stepname, all steps prior to this condition will be checked.

You can nest another IF construct after the THEN clause or the ELSE clause. You can nest IF/THEN/ELSE/ENDIF statement constructs up to 15 levels of nesting.

Either the THEN clause or ELSE clause must contain at least one EXEC statement.

Let's see few more examples below.

Example 2:

```
//MATEKSD JOB MSGLEVEL=(1,1),NOTIFY=&SYSUID
//*
//* EXAMPLE TO SHOW IF CONDITION IN JCL
//*
//STEP01   EXEC PGM=IFCOND1
//*
//IFSTMT1 IF STEP01.RC = 0  THEN  ---> This will check if the
Return Code(RC)
//STEP02   EXEC PGM=IFCOND2              from STEP01 is 0, if yes
then execute
//STEP03   EXEC PGM=IFCOND3              STEP02 and STEP03.
//        ENDIF
//STEP04   EXEC PGM=IFCOND4
//STEP05   EXEC PGM=IFCOND5
//*
//IFSTMT2 IF STEP05.RC = 04 THEN   ---> This will check if the
Return Code(RC)
//STEP06   EXEC PGM=IFCOND6               from STEP05 is 04, if yes
then execute
//        ELSE                            STEP06, else execute
STEP07.
//STEP07   EXEC PGM=IFCOND7
//        ENDIF
//STEP08   EXEC PGM=IFCOND8
//*
```

Example 3:

```
//JOBIBM JOB CLASS=C,NOTIFY=&SYSUID
//*
//PRC1   PROC
//PST1    EXEC PGM=IDCAMS
//PST2    EXEC PGM=SORT
//       PEND
//STEP01  EXEC PGM=SORT
//IF1    IF STEP01.RC = 0 THEN
//STEP02     EXEC PGM=MYCOBB,PARM=321
//       ENDIF
//IF2    IF STEP01.RUN THEN
//STEP03a    EXEC PGM=IDCAMS
//STEP03b    EXEC PGM=SORT
//       ENDIF
//IF3    IF STEP03b.!ABEND THEN
//STEP04     EXEC PGM=MYCOBB,PARM=654
//       ELSE
//       ENDIF
//IF4    IF (STEP01.RC = 0 & STEP02.RC <= 4) THEN
//STEP05     EXEC PROC=PRC1
//       ENDIF
//IF5    IF STEP05.PRC1.PST1.ABEND THEN
//STEP06     EXEC PGM=IDCAMS
//       ELSE
//STEP07     EXEC PGM=SORT
//       ENDIF
```

Explanation:

- The return code of STEP01 is tested in IF1. If it is 0, then STEP02 is executed. Else, the processing goes to the next IF statement (IF2).
- In IF2, If STEP01 has started execution, then STEP03a and STEP03b are executed.
- In IF3, If STEP03b does not ABEND, then STEP04 is executed. In ELSE, there are no statements. It is called a NULL ELSE statement.
- In IF4, if STEP01.RC = 0 and STEP02.RC <=4 are TRUE, then STEP05 is executed.
- In IF5, if the proc-step PST1 in PROC PRC1 in jobstep STEP05 ABEND, then STEP06 is executed. Else STEP07 is executed.
- If IF4 evaluates to false, then STEP05 is not executed. In that case, IF5 are not tested and the steps STEP06, STEP07 are not executed.

The IF-THEN-ELSE will not be executed in the case of abnormal termination of the job such as user cancelling the job, job time expiry or a dataset is backward referenced to a step that is bypassed.

JCL - IF/THEN/ELSE/ENDIF Statements

Another approach to control the job processing is by using IF-THEN-ELSE constructs. This gives more flexibility and user-friendly way of conditional processing.

In JCL, IF condition is better than the COND parameter because of 2 reasons –

1. IF condition in JCL is easy to code as compared to the COND parameter.
2. JCL programmers with less experience find COND parameter little confusing and boring and it is easier to read and understand IF-ELSE-END-IF condition in JCL.

Syntax:

```
//[name] IF  [(]relational-expression[)] THEN   [comments]
     .
     .    action when relational-expression is true
     .
//[name] ELSE   [comments]
     .
     .    action when relational-expression is false
     .
//[name] ENDIF   [comments]
```

Following is the description of the used terms in the above IF-THEN-ELSE Construct:

name : This is optional and a name can have 1 to 8 alphanumeric characters starting with alphabet, #,$ or @.

Relational-expression : A relational-expression will have a format: KEYWORD OPERATOR VALUE, where KEYWORDS can be RC (Return Code), ABENDCC (System or user completion code), ABEND, RUN (step started execution). An OPERATOR can be logical operator (AND (&), OR (|)) or relational operator (<, <=, >, >=, <>).

comments: The comments field follows THEN, ELSE, and ENDIF after at least one intervening blank.

The below keywords can include a stepname and procstepname to refine the test to a specific job step.

A Keyword List

Keyword	Purpose

ABEND	Tests for an abnormal end of a program
ABEND	Tests that an abnormal end of a program did not occur
ABENDCC	Examines an ABEND condition code
RC	Examines a return code
RUN	Tests if a job step executed
RUN	Tests if a job step did not execute

Syntax:

```
stepname.procstepname.keyword
```

Example 1: Let us see simple IF condition example,

```
//JOBEXP JOB
//STEP01  EXEC MYPROC01
//COND01  IF RC = 0 THEN
//STEP02  EXEC MYPROC02
//CONDE   ELSE
//STEP03  EXEC MYPROC03
//        ENDIF
```

Explanation:

In this example, After STEP01 is completed we check the Return code of STEP01 step. if return code is zero then STEP02 step will execute else step STEP03 will be executed.

Tips:

```
IF  RC = 0   THEN

IF  STEP1.RC < 12  THEN
```

If you not given the stepname, then the highest return code from all job steps is taken for checking.

ABENDCC - Using ABENDCC we can check System/User completion codes.

```
IF ABENDCC = S0C7 THEN
```

Suppose you want to check error code of particular step, give stepnname.ABENDCC, If you not given the stepname, most recent ABEND code that occured is taken for checking.

ABEND- checks for an abnormal end of a program

```
IF  ABEND  THEN

IF  STEP4.PROCAS01.ABEND = TRUE  THEN
```

If you not given any stepname, all steps prior to this condition will be checked.

You can nest another IF construct after the THEN clause or the ELSE clause. You can nest IF/THEN/ELSE/ENDIF statement constructs up to 15 levels of nesting.

Either the THEN clause or ELSE clause must contain at least one EXEC statement.

Let's see few more examples below.

Example 2:

```
//MATEKSD JOB MSGLEVEL=(1,1),NOTIFY=&SYSUID
//*
//* EXAMPLE TO SHOW IF CONDITION IN JCL
//*
//STEP01   EXEC PGM=IFCOND1
//*
//IFSTMT1 IF STEP01.RC = 0  THEN  ---> This will check if the
Return Code(RC)
//STEP02   EXEC PGM=IFCOND2             from STEP01 is 0, if yes
then execute
//STEP03   EXEC PGM=IFCOND3             STEP02 and STEP03.
//        ENDIF
//STEP04   EXEC PGM=IFCOND4
//STEP05   EXEC PGM=IFCOND5
//*
//IFSTMT2 IF STEP05.RC = 04 THEN   ---> This will check if the
Return Code(RC)
//STEP06   EXEC PGM=IFCOND6              from STEP05 is 04, if yes
then execute
//        ELSE                           STEP06, else execute
STEP07.
//STEP07   EXEC PGM=IFCOND7
//        ENDIF
```

```
//STEP08   EXEC PGM=IFCOND8
//*
```

Example 3:

```
//JOBIBM JOB CLASS=C,NOTIFY=&SYSUID
//*
//PRC1   PROC
//PST1    EXEC PGM=IDCAMS
//PST2    EXEC PGM=SORT
//       PEND
//STEP01  EXEC PGM=SORT
//IF1    IF STEP01.RC = 0 THEN
//STEP02     EXEC PGM=MYCOBB,PARM=321
//       ENDIF
//IF2    IF STEP01.RUN THEN
//STEP03a    EXEC PGM=IDCAMS
//STEP03b    EXEC PGM=SORT
//       ENDIF
//IF3    IF STEP03b.!ABEND THEN
//STEP04     EXEC PGM=MYCOBB,PARM=654
//       ELSE
//       ENDIF
//IF4    IF (STEP01.RC = 0 & STEP02.RC <= 4) THEN
//STEP05     EXEC PROC=PRC1
//       ENDIF
//IF5    IF STEP05.PRC1.PST1.ABEND THEN
//STEP06     EXEC PGM=IDCAMS
//       ELSE
//STEP07     EXEC PGM=SORT
//       ENDIF
```

Explanation:

- The return code of STEP01 is tested in IF1. If it is 0, then STEP02 is executed. Else, the processing goes to the next IF statement (IF2).
- In IF2, If STEP01 has started execution, then STEP03a and STEP03b are executed.
- In IF3, If STEP03b does not ABEND, then STEP04 is executed. In ELSE, there are no statements. It is called a NULL ELSE statement.
- In IF4, if STEP01.RC = 0 and STEP02.RC <=4 are TRUE, then STEP05 is executed.
- In IF5, if the proc-step PST1 in PROC PRC1 in jobstep STEP05 ABEND, then STEP06 is executed. Else STEP07 is executed.
- If IF4 evaluates to false, then STEP05 is not executed. In that case, IF5 are not tested and the steps STEP06, STEP07 are not executed.

The IF-THEN-ELSE will not be executed in the case of abnormal termination of the job such as user cancelling the job, job time expiry or a dataset is backward referenced to a step that is bypassed.

JCL - Utility Programs

Data set utilities:

Data set utilities reorganize, change, or compare data at the data set or record level. These programs are widely used in batch jobs.

Utility programs are pre-written programs, widely used in mainframes by system programmers and application developers to achieve day-to-day requirements, organising and maintaining data.

A few of them are listed below with their functionality:

1. **IEFBR14**- The utility program IEFBR14 performs no action other than return a completion code of 0; however, "running" this utility invokes other system components that perform useful tasks.

2. **IDCAMS**- Although it provides other functions, IDCAMS, which is the program name for access method services, is used primarily to define and manage VSAM data sets and integrated catalog facility catalogs.

3. **IEBCOPY**- IEBCOPY is a data set utility that is used to copy or merge members between one or more partitioned data sets, or partitioned data sets extended (PDSEs), in full or in part.

4. **IEBDG**- The IEBDG utility provides a pattern of test data to be used as a programming debugging aid. This pattern of data can then be analyzed quickly for predictable results.

5. **IEBGENER**- The IEBGENER utility is a copy program that has been part of the operating system since the first release of OS/360. One of its many uses is to copy a sequential data set, a member of a partitioned data set (PDS) or PDSE, or a z/OS UNIX System Services (z/OS UNIX) file such as a HFS file.

6. **IEBUPDTE**- The IEBUPDTE utility creates multiple members in a partitioned data set, or updates records within a member. While it can be used for other types of records, its main use is to create or maintain JCL procedure libraries or assembler macro libraries.

JCL - IEFBR14 Utility

The IEFBR14 program is nothing more than a null program. Its name is derived from an assemble language instruction that is used to exit a procedure or program. and this is exactly what is does. It executed a single statment which specifies the end of the program.

The utility program IEFBR14 performs no action other than return a completion code of 0; however, "running" this utility invokes other system components that perform useful tasks.

1. Allocate/create datasets
2. Delete datasets
3. Uncatlog Datasets
4. Catalog Datasets
5. Setting return code to zero

The IEFBR14 program is not connsidered by IBM to be a utility program. However, it is used like a utility in that not do anything by itself.

Since IEFBR14 is a do nothing program, it can also be used to check the syntax of your JCL without affecting any datasets.

We mostly use IEFBR14 utility to create or delete the PS(partition dataset) file.

Let us see few examples below.

Example 1: Allocate/Create empty PS dataset.

```
//JOBIBMKS  JOB
(123),'IBMMAINFRAMER',CLASS=C,MSGCLASS=S,MSGLEVEL=(1,1),
//       NOTIFY=&SYSUID
//*
//STEP001  EXEC PGM=IEFBR14
//SYSPRINT DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//SYSDUMP  DD SYSOUT=*
//DD01     DD DSN=userid.TEST.PSFILE,
//            DISP=(NEW,CATLG,DELETE),
//            SPACE=(TRK,(12,33),RLSE),UNIT=SYSDA,
//            DCB=(DSORG=PS,RECFM=FB,LRECL=80,BLKSIZE=800)
//*
```

OUTPUT: This job will create the PS(userid.TEST.PSFILE) with defined parameter value.

Explanation:

- **SYSPRINT** - This is optional DD statement. It is used by utility programs for their output.

- **SYSOUT** - This is optional DD statement. This specifies system defined dd name used for file status codes, system abend codes information and output of the display statement.
- **SYSDUMP** - This is optional DD statement. It is used by the system for dumping when an abend occurs that causes a system dump.
- **DD01** - This DD statment specifies PS dataset attributes for creation.

Mostly we don't add optional DD statment in the JCL. See below JCL without optional DD statement.

```
//JOBIBMKS  JOB
(123),'IBMMAINFRAMER',CLASS=C,MSGCLASS=S,MSGLEVEL=(1,1),
//       NOTIFY=&SYSUID
//*
//STEP001  EXEC PGM=IEFBR14
//DD01     DD DSN=userid.TEST.PSFILE,
//            DISP=(NEW,CATLG),
//            SPACE=(TRK,(12,33),RLSE),UNIT=SYSDA,
//            DCB=(DSORG=PS,RECFM=FB,LRECL=80,BLKSIZE=800)
//*
```

Example 2: Delete PS dataset.

```
//JOBIBMKS JOB
(123),'IBMMAINFRAMER',CLASS=C,MSGCLASS=S,MSGLEVEL=(1,1),
//       NOTIFY=&SYSUID
//*
//STEP001  EXEC PGM=IEFBR14
//SYSPRINT DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//SYSDUMP  DD SYSOUT=*
//DD01     DD DSN=userid.TEST.PSFILE,
//            DISP=(OLD,DELETE,DELETE)
//*
```

OUTPUT: This job will delete the PS(userid.TEST.PSFILE) dataset.

Explanation:

- **SYSPRINT** - This is optional DD statement. It is used by utility programs for their output.
- **SYSOUT** - This is optional DD statement. This specifies system defined dd name used for file status codes, system abend codes information and output of the display statement.

- **SYSDUMP** - This is optional DD statement. It is used by the system for dumping when an abend occurs that causes a system dump.
- **DD01 DD** - This is optional DD statement. This specifies PS dataset with DISP for deletion (DISP=(OLD,DELETE,DELETE)).

As discussed above, Mostly we don't add optional DD statment in the JCL. See below JCL without optional DD statement.

```
//JOBIBMKS JOB
(123),'IBMMAINFRAMER',CLASS=C,MSGCLASS=S,MSGLEVEL=(1,1),
//       NOTIFY=&SYSUID
//*
//STEP001  EXEC PGM=IEFBR14
//DD01     DD DSN=userid.TEST.PSFILE,
//            DISP=(OLD,DELETE)
//*
```

Example 3: Uncatalog dataset.

```
//JOBIBMKS JOB
(123),'IBMMAINFRAMER',CLASS=C,MSGCLASS=S,MSGLEVEL=(1,1),
//       NOTIFY=&SYSUID
//*
//STEP001  EXEC PGM=IEFBR14
//DD01     DD DSN=userid.TEST.PSFILE,
//            DISP=(OLD,UNCATLG)
//*
```

In this example, the IEFBR14 program is used to uncatalog the dataset called userid.TEST.PSFILE

Example 4: Catalog dataset.

```
//JOBIBMKS JOB
(123),'IBMMAINFRAMER',CLASS=C,MSGCLASS=S,MSGLEVEL=(1,1),
//       NOTIFY=&SYSUID
//*
```

```
//STEP001  EXEC PGM=IEFBR14
//DD01     DD DSN=userid.TEST.PSFILE,
//            DISP=(OLD,CATLG)
//*
```

In this example, the IEFBR14 program is used to catalog the dataset called userid.TEST.PSFILE

JCL - IEBGENER Utility

The IEBGENER utility is a copy program. One of its many uses is to copy a sequential data set, a member of a partitioned data set (PDS) or PDSE.

IEBGENER also can filter data, change a data set's logical record length (LRECL) and block size (BLKSIZE), and generate records.

This is most commonly used utility programs. It is used to copy one sequential file to another.

Rules for Coding:

- Original dataset name must be specified.
- Destination dateset name must be specified.
- Attributes of both datasets(such as record format and logic record length), must be the same

A following JCL executes the IEBGENER program.

Example 1: COPY PS dataset.

```
//JOBIBMKS JOB
(123),'IBMMAINFRAMER',CLASS=C,MSGCLASS=S,MSGLEVEL=(1,1),
//       NOTIFY=&SYSUID
//STEP001  EXEC PGM=IEBGENER
//SYSPRINT DD SYSOUT=A
//SYSUT1   DD DSN=userid.FILE1.INPUT,DISP=SHR
//SYSUT2   DD DSN=userid.FILE2.OUTPUT,
//        DISP=(NEW,CATLG,DELETE),UNIT=DISK1,
//        SPACE=(TRK,20,10),RLSE),
//SYSIN    DD DUMMY
//
```

The utility IEBGENER is exeuted and the output if directed to the device destinated by A. Traditionally, this the line printer.

IEBGENER requires four DD(data definition) statements with the DD names shown in the above example,

Explanation:

- The SYSIN DD statement is used as a DUMMY dataset since there are no control statements. When IEBGENER encounters this SYSIN statment, it immediately gives an end-of-file indication
- The SYSPRINT statement is for messages from IEBGENER.
- The SYSUT1 statement is for input file. The name of the file to be copied is userid.FILE1.INPUT.
- The SYSUT2 statement is for output file, which will be a copy of the file identified in SYSUT1. This file is called userid.FILE2.OUTPUT.

JCL - IEBCOPY Utility

IEBCOPY is a data set utility that is used to copy or merge members between one or more partitioned data sets or partitioned data sets extended (PDSEs). The copying can be full copy or partial copy.

IEBCOPY used to create a backup of a partitioned data set into a sequential data set and to copy members from the backup into a partitioned data set. IEBCOPY automatically lists the number of unused directory blocks.

More specifically, this utility is commonly used for several purposes:

1. To copy selected (or all) members from one partitioned data set to another.
2. To copy a partitioned data set into a unique sequential format known as an unloaded partitioned data set. As a sequential data set it can be written on tape, sent by FTP, or manipulated as a simple sequential data set.
3. To read an unloaded partitioned data set (which is a sequential file) and recreate the original partitioned data set. Optionally, only selected members might be used.
4. To compress partitioned data sets (in place) to recover lost space.

Example 1:

All members of input data set userid.TEST.DATA.IN are copied to output dataset userid.TEST.DATA.OUT

```
//JOBIBMKS JOB
(123),'IBMMAINFRAMER',CLASS=C,MSGCLASS=S,MSGLEVEL=(1,1),
//       NOTIFY=&SYSUID
//*
//STEP001  EXEC  PGM=IEBCOPY
//SYSPRINT DD  SYSOUT=A
//SYSOUT1  DD  DSNAME=userid.TEST.DATA.IN,
//             DISP=SHR,UNIT=DISK,
//             VOL=SER=1234
//SYSOUT2  DD  DSNAME=userid.TEST.DATA.OUT,
//             DISP=(NEW,KEEP),
//             SPACE=(TRK,(10,22,40),RLSE),
//             UNIT=DISK,VOL=SER=1234,
//SYSIN    DD DUMMY
/*
```

Explanation:

- SYSUT1 DD defines a PDS - userid.TEST.DATA.IN that contains 5 members (XXX, YYY, ZZZZ, AAA and BBB).
- SYSUT2 DD defines a new PDS - userid.TEST.DATA.OUT that is to be kept after the copy operation.
- Input and output data sets are identified as SYSUT1 and SYSUT2 DD statements, the SYSIN data set is not needed. So We mentioned as DUMMY dataset since there are no control statements.
- The SYSUT1 data set will be copied to the SYSUT2 data set.
- After copy operation is completed, PDS - userid.TEST.DATA.OUT will contain the all the members that are in PDS - userid.TEST.DATA.OUT

For example, if you wanted to copy specific members(for example - XXX and YYY) from input PDS - userid.TEST.DATA.IN. Then instead of using the DUMMY parameter on the SYSIN DD statement, you could substitute this JCL.

```
//SYSIN DD *
  COPY OUTDD=SYSUT2,
       INDD=SYSUT1
  SELECT MEMBER=(XXX,YYY)
/*
```

The SELECT statement specifies the member names to be processed, with the OUTDD and INDD parameters specifying the DD names to be used for output and input, respectively. You would have to use this JCL if you used names other than SYSUT1 and SYSUT2 for the input and output DD statements.

Restoring a partitioned data set from an unloaded copy automatically compresses (recovers lost space) the data set.

Example 2: Including members of PDS in a COPY command

```
//JOBIBMKS JOB
(123),'IBMMAINFRAMER',CLASS=C,MSGCLASS=S,MSGLEVEL=(1,1),
//       NOTIFY=&SYSUID
//*
//STEP001  EXEC  PGM=IEBCOPY
//SYSPRINT DD  SYSOUT=A
//SYSOUT1  DD  DSNAME=userid.TEST.DATA.IN,
//             DISP=SHR,VOL=SER=1234
//             UNIT=DISK1
//SYSOUT2  DD  DSNAME=userid.TEST.DATA.OUT,
//             DISP=(NEW,KEEP),SPACE=(TRK,(50,20,50), RLSE)
//             UNIT=DISK1,
//             VOL=SER=ABC,
//SYSIN    DD  *
  COPY  INDD=SYSOUT1,
        OUTDD=SYSOUT2
  SELECT MEMBER=(XXX,YYY)
/*
//
```

Example 3: Excluding members of PDS in a COPY command.

```
//JOBIBMKS JOB
(123),'IBMMAINFRAMER',CLASS=C,MSGCLASS=S,MSGLEVEL=(1,1),
//       NOTIFY=&SYSUID
//*
//STEP001  EXEC PGM=IEBCOPY
//*********************************************************
//SYSPRINT DD SYSOUT=A
//SYSUT1   DD DSN=userid.JCLTEST.JCL,DISP=SHR
//SYSUT2   DD DSN=userid.JCLTEST.JCLS.OTHERS,
//          DISP=(NEW,CATLG,DELETE),
//          SPACE=(CYL,(10,10,60),RLSE),
//           DCB=(LRECL=80,RECFM=FB,BLKSIZE=800)
//SYSIN    DD *
  COPY INDD=SYSUT1,
       OUTDD=SYSUT2
  EXCLUDE MEMBER=(PGM1,PGM2,PGM3,PGM4,
       PGM5,PGM6)
/*
```

Example 4: Renaming a member while copying

```
//JOBIBMKS JOB
(123),'IBMMAINFRAMER',CLASS=C,MSGCLASS=S,MSGLEVEL=(1,1),
//       NOTIFY=&SYSUID
//*
//STEP001  EXEC PGM=IEBCOPY
//*********************************************************
//SYSPRINT DD SYSOUT=A
//SYSUT1   DD DSN=userid.JCLTEST.JCL,DISP=SHR
//SYSUT2   DD DSN=userid.JCLTEST.JCL.OTHER,
//       DISP=(NEW,CATLG,DELETE),
//       SPACE=(CYL,(40,50,60),RLSE),
//       DCB=(LRECL=80,RECFM=FB,BLKSIZE=800)
//SYSIN DD *
  COPY INDD=SYSUT1,
       OUTDD=SYSUT2
  SELECT MEMBER=(PGM1,PGM2,PGM3,PGM4,
       PGM5,(PROGRAM6,PGM6))
/*
```

Example 5: Compressing the Datasets

Datasets can be compressed together in order to move efficiently utilize the device space that they are stored on. The JCL that does this is similar to the above examples.

```
//JOBIBMKS JOB
(123),'IBMMAINFRAMER',CLASS=C,MSGCLASS=S,MSGLEVEL=(1,1),
//       NOTIFY=&SYSUID
//*
//STEP001  EXEC PGM=IEBCOPY
//*********************************************************
//SYSPRINT DD SYSOUT=A
```

```
//SYSUT1   DD DSN=userid.JCLTEST.JCL,DISP=SHR
//SYSIN DD *
  COPY INDD=SYSUT1,
       OUTDD=SYSUT1
/*
```

If you take another look at this JCL, you will realize that the INDD and OUTDD statements reference to the same dataset.

If you executing this job will result in all members of the partioned dataset called userid.JCLTEST.JCL being compressed together, resulting in the relese of wasted space between records and the datset members.

JCL - IDCAMS Utility

IDCAMS stands for Integrated Data Cluster Access Method Services. IDCAMS utility is used to create, modify and delete the VSAM datasets.

IDCAMS Utility is very useful utility to manipulate VSAM datasets

A typical example of a simple use of IDCAMS is as follows:

Example 1: Allocating and Loading Data Into VSAM

```
//JOBIBMKS JOB
(123),'IBMMAINFRAMER',CLASS=C,MSGCLASS=S,MSGLEVEL=(1,1),
//       NOTIFY=&SYSUID
//*
//STEP001  EXEC PGM=IDCAMS
//SYSPRINT DD *
//DATAIN   DD DISP=OLD,DSN=userid.TEST.INPUT
//SYSIN    DD *
  DEFINE CLUSTER (
           NAME (userid.TEST.VSAM) -
           VOLUMES(WORK02) -
           CYLINDERS(1 1) -
           RECORDSIZE(72 100) -
           KEYS(9 8) -
           INDEXED)
  REPRO INFILE(DATAIN) -
```

```
        OUTDATASET(userid.DATA.VSAM) -
        ELIMIT(200)
/*
```

Example 2: Deleting a VSAM dataset

```
//JOBIBMKS JOB
(123),'IBMMAINFRAMER',CLASS=C,MSGCLASS=S,MSGLEVEL=(1,1),
//       NOTIFY=&SYSUID
//*
//STEP001  EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
   DELETE useird.DATA.VSAM CLUSTER
/*
```

- More Topics will be added soon...

JCL - Generation Data Group or GDG

- Generation Data Groups (GDGs) are group of datasets related to each other by a common name.
- The common name is referred as GDG base and each dataset associated with the base is called a GDG version.
- A Maximum of 255 generations are allowed for a GDG base.

For example,

MYDATA.TEST.SAMPLE.GDG is the GDG base name.

The datasets are named as MYDATA.TEST.SAMPLE.GDG.G0001V00, MYDATA.TEST.SAMPLE.GDG.G0002V00 and so on.

- G – Stands for Generation number – Value range 0000 – 9999

- V – Stands for Version number – Value range 00 – 99

Use of GDG:

1. We do not need to create a new JCL or change the name of the JCL every time in a weekly run or daily run or monthly run or yearly run of a JCL.
2. You can set the limit of the related files(generations).
3. We can easily keep track of all generation of data sets.
4. We can delete or uncatalog the older generation.
5. Any particular generation can be referred easily.

Current version of the GDG is referred as MYDATA.TEST.SAMPLE.GDG(0),
Previous versions are referred as (-1), (-2) and so on.
Next(new) version to be created in a program is refered as
MYDATA.TEST.SAMPLE.GDG(+1).

How to create a GDG?

STEP1: Create a GDG base (using IDCAMS utility – Access method service utility)

```
//DEFGDG1   JOB  ...
//STEP1    EXEC PGM=IDCAMS
//SYSPRINT DD   SYSOUT=A
//SYSIN    DD   *
     DEFINE GDG -
           (NAME(userid.MYGDG.TEST) -
           EMPTY -
           NOSCRATCH -
           LIMIT(15))
/*
```

List of PARAMETERS used to create GDG:

NAME – Name of the GDG Base.

LIMIT – To limit the maximum number of generations.

EMPTY/NOEMPTY:

NOEMPTY – Uncatalog only the oldest generation in GDG when the limit is reached.
EMPTY – Uncatalog all the generations when a limit is reached.

SCRATCH/NOSCRATCH:

SCRATCH -Physically delete the dataset(generation) which is uncataloged.
NOSCRATCH – Don't Physically delete the dataset(generation) which is uncataloged.

STEP2: Using IEFBR14 utility, we create the new generations.

```
//DEFGDS   JOB  ...
//STEP1    EXEC PGM=IEFBR14
//GDSDD1   DD   DSN=ICFUCAT1.GDG02(+1),DISP=(NEW,CATLG),
//         SPACE=(TRK,(5,2)),STORCLAS=GRPVOL1,DATACLAS=ALLOC01
//SYSPRINT DD   SYSOUT=A
//SYSIN    DD   *
/*
```

If there is no generation created for this gdg base, this job creates new generation - ICFUCAT1.GDG02.G0001V00. otherwise, it will create a new generation based on existing generation number.

After you have created several generations. if you display your GDG would look like this in ISPF 3.4:

```
ICFUCAT1.GDG02
ICFUCAT1.GDG02.G0001V00
ICFUCAT1.GDG02.G0002V00
ICFUCAT1.GDG02.G0003V00
```

Difference between EMPTY and NOEMPTY?

Some of you might be confused between EMPTY and NOEMPTY.

Let us take an example –

Suppose, you have a GDG Base, IBM.SALES.REPORT and you have created 3 generations like –

```
IBM.SALES.REPORT.G0001V00

IBM.SALES.REPORT.G0002V00

IBM.SALES.REPORT.G0003V00
```

Now, when you try to create another generation and if you have coded 'NOEMPTY' – It means that it will remove the the oldest generation from the catalog when the limit is reached which means that only IBM.SALES.REPORT.G0001V00 will be uncataloged. So, now the 3 generations which are present are -

```
IBM.SALES.REPORT.G0002V00

IBM.SALES.REPORT.G0003V00

IBM.SALES.REPORT.G0004V00
```

Now, Suppose, you have a GDG Base, IBM.SALES.STATUS and you have created 3 generations like IBM.SALES.STATUS.G0001V00, IBM.SALES.STATUS.G0002V00, and IBM.SALES.STATUS.G0003V00.

Now, what happens when you try to create another generation and If you have coded 'EMPTY' – It means that it will remove all the older generations from the catalog which means that IBM.SALES.STATUS.G0001V00, IBM.SALES.STATUS.G0002V00, and IBM.SALES.STATUS.G0003V00 will be uncataloged and IBM.SALES.STATUS.G00004V00 will be created.

How to use GDG in a JCL?

In the following example, the latest version of MYDATA.IBM.SAMPLE.GDG is used as input to the program and a new version of MYDATA.IBM.SAMPLE.GDG is created as the output.

```
//CNDSAMP JOB CLASS=6,NOTIFY=&SYSUID
```

```
//*
//STP01   EXEC PGM=MYCOBB
//IN1     DD DSN=MYDATA.IBM.SAMPLE.GDG(0),DISP=SHR
//OUT1    DD DSN=MYDATA.IBM.SAMPLE.GDG(+1),DISP=(,CALTG,DELETE)
//        LRECL=100,RECFM=FB
```

Here, if the GDG had been referred by the actual name like MYDATA.IBM.SAMPLE.GDG.G0001V00, then it leads to changing the JCL every time before execution. Using (0) and (+1) makes it dynamically substitute the GDG version for execution.

How to delete a GDG?

If you need to delete your GDG, delete the individual data sets (G0001V00, G0002V00, etc.) and then run this IDCAMS job to delete the GDG.

```
//STEP010 EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//SYSIN DD *
  DELETE (MYDATA.IBM.SAMPLE.GDG) GDG FORCE
/*
```

How to modify existing GDG parameters?

If you wish to change the parameter of the GDG Base. We use ALTER command,

For example, if you want to change the number of generations run this IDCAMS alter example where the number of generation is increased to 50.

```
//STEP010 EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//SYSIN DD *
  ALTER MYDATA.IBM.SAMPLE.GDG LIMIT(50)
/*
```