**JOB Statement**

JOB Statement is the first control statement in a JCL. This gives the identity of the job to the Operating System (OS), in the spool and in the scheduler. The parameters in the JOB statement help the Operating Systems in allocating the right scheduler, required CPU time and issuing notifications to the user.

Syntax

Following is the basic syntax of a JCL JOB statement:

//Job-name JOB Positional-param, Keyword-param

Description

Let us see the description of the terms used in above JOB statement syntax.

Job-name

This gives an id to the job while submitting it to the OS. It is can be length of 1 to 8 with alphanumeric characters and starts just after //.

JOB

This is the keyword to identify it as a JOB statement.

Positional-param

There are positional parameters, which can be of two types:

|  |  |
| --- | --- |
| **Positional Parameter** | **Description** |
| **Account information** | This refers to the person or group to which the CPU time is owed. It is set as per the rules of the company owning the mainframes. If it is specified as (\*), then it takes the id of the user, who has currently logged into the Mainframe Terminal. |
| **Programmer name** | This identifies the person or group, who is in charge of the JCL. This is not a mandatory parameter and can be replaced by a comma. |

Keyword-param

Following are the various keyword parameters, which can be used in JOB statement. You can use one or more parameters based on requirements and they are separated by comma:

|  |  |
| --- | --- |
| **Keyword Parameter** | **Description** |
| **CLASS** | Based on the time duration and the number of resources required by the job, companies assign different job classes. These can be visualized as individual schedulers used by the OS to receive the jobs. Placing the jobs in the right scheduler will aid in easy execution of the jobs. Some companies have different classes for jobs in test and production environment.  Valid values for CLASS parameter are A to Z characters and 0 to 9 numeric (of length 1). Following is the syntax:  **CLASS=0 to 9 | A to Z** |
| **PRTY** | To specify the priority of the job within a job class. If this parameter is not specified, then the job is added to the end of the queue in the specified CLASS. Following is the syntax:  **PRTY=N**  Where N is a number in between 0 to 15 and higher the number, higher is the priority. |
| **NOTIFY** | The system sends the success or failure message (Maximum Condition Code) to the user specified in this parameter. Following is the syntax:  **NOTIFY="userid | &SYSUID"**  Here system sends the message to the user "userid" but if we use NOTIFY = &SYSUID, then the message is sent to the user submitting the JCL. |
| **MSGCLASS** | To specify the output destination for the system and Job messages when the job is complete. Following is the syntax:  **MSGCLASS=CLASS**  Valid values of CLASS can be from "A" to "Z" and "0" to "9". MSGCLASS = Y can be set as a class to send the job log to the JMR (JOBLOG Management and Retrieval: a repository within mainframes to store the job statistics). |
| **MSGLEVEL** | Specifies the type of messages to be written to the output destination specified in the MSGCLASS. Following is the syntax:  **MSGLEVEL=(*ST, MSG*)**  *ST* = Type of statements written to output log   * When *ST* = 0, Job statements only. * When *ST* = 1, JCL along with symbolic parameters expanded. * When *ST* = 2, Input JCL only.   *MSG* = Type of messages written to output log.   * When *MSG* = 0, Allocation and Termination messages written upon abnormal job completion. * When *MSG* = 1, Allocation and Termination messages written irrespective of the nature of job completion. |
| **TYPRUN** | Specifies a special processing for the job. Following is the syntax:  **TYPRUN = SCAN | HOLD**  Where SCAN and HOLD has the following description   * TYPRUN = SCAN checks the syntax errors of the JCL without executing it. * TYPRUN = HOLD puts the job on HOLD in the job queue.To release the job, "A" can be typed against the job in the SPOOL, which will bring the job to execution. |
| **TIME** | Specifies the time span to be used by the processor to execute the job. Following is the syntax:  **TIME=(mm, ss) or TIME=ss**  Where mm = minutes and ss = seconds  This parameter can be useful while testing a newly coded program. In order to ensure that the program does not run for long because of looping errors, a time parameter can be coded so that the program abends when the specified CPU time is reached. |
| **REGION** | Specifies the address space required to run a job step within the job. Following is the syntax:  **REGION=nK | nM**  Here, *region* can be specified as nK or nM where n is a number, K is kilobyte and M is Megabyte.  When REGION = 0K or 0M, largest address space is provided for execution.In critical applications, coding of 0K or 0M is prohibited to avoid wasting the address space. |

Example

//URMISAMP JOB (\*),"tutpoint",CLASS=6,PRTY=10,NOTIFY=&SYSUID,

// MSGCLASS=X,MSGLEVEL=(1,1),TYPRUN=SCAN,

// TIME=(3,0),REGION=10K

Here, JOB statement is getting extended beyond the 70th position in a line,so we continue in the next line which should start with "//" followed by one or more spaces.

Miscellaneous Parameters

There are few other parameters, which can be used with JOB Statement but they are not frequently used:

|  |  |
| --- | --- |
| **ADDRSPC** | Type of storage used: Virtual or Real |
| **BYTES** | Size of data to be written to output log and the action to be taken when the size is exceeded. |
| **LINES** | Maximum number of lines to be printed to output log. |
| **PAGES** | Maximum number of pages to be printed to output log. |
| **USER** | User id used to submit the job |
| **PASSWORD** | Password of the user-id specified in the USER parameter. |
| **COND and RESTART** | These are used in conditional job step processing and are explained in detail while discussing conditional Processing. |

**EXEC statement**

Each JCL can be made of many job steps. Each job step can execute a program directly or can call a procedure, which in turn executes one or more programs (job steps). The statement, which holds the job step program/procedure information is the **EXEC statement.**

The purpose of the EXEC statement is to provide required information for the program/procedure executed in the job step. Parameters coded in this statement can pass data to the program in execution, can override certain parameters of JOB statement and can pass parameters to the procedure if the EXEC statement calls a procedure instead of directly executing a program.

Syntax

Following is the basic syntax of a JCL EXEC statement:

//Step-name EXEC Positional-param, Keyword-param

Description

Let us see the description of the terms used in above EXEC statement syntax.

STEP-NAME

This identifies the job step within the JCL. It can be of length 1 to 8 with alphanumeric characters.

EXEC

This is the keyword to identify it as an EXEC statement.

POSITIONAL-PARAM

These are positional parameters, which can be of two types:

|  |  |
| --- | --- |
| **Positional Parameter** | **Description** |
| **PGM** | This refers to the program name to be executed in the job step. |
| **PROC** | This refers to the procedure name to be executed in the job step. We will discuss it a separate chapter. |

KEYWORD-PARAM

Following are the various keyword parameters for EXEC statement. You can use one or more parameters based on requirements and they are separated by comma:

|  |  |
| --- | --- |
| **Keyword Parameter** | **Description** |
| **PARM** | Used to provide parametrized data to the program that is being executed in the job step. This is a program dependant field and do not have definite rules, except that the PARM value has to be included within quotation in the event of having special characters.  For example given below, the value "CUST1000" is passed as an alphanumeric value to the program. If the program is in COBOL, the value passed through a PARM parameter in a JCL is received in the LINKAGE SECTION of the program. |
| **ADDRSPC** | This is used to specify whether the job step require virtual or real storage for execution. Virtual storage is pageable whereas real storage is not and is placed in the main memory for execution. Job steps, which require faster execution can be placed in real storage. Following is the syntax:  **ADDRSPC=VIRT | REAL**  When an ADDRSPC is not coded, VIRT is the default one. |
| **ACCT** | This specifies the accounting information of the job step. Following is the syntax:  **ACCT=(userid)**  This is similar to the positional parameter **accounting information** in the JOB statement. If it is coded both in JOB and EXEC statement, then the accounting information in JOB statement applies to all job steps where an ACCT parameter is not coded. The ACCT parameter in an EXEC statement will override the one present in the JOB statement for that job step only. |

Common Keyword Parameters of EXEC and JOB Statement

|  |  |
| --- | --- |
| **Keyword Parameter** | **Description** |
| **ADDRSPC** | ADDRSPC coded in JOB statement overrides the ADDRSPC coded in EXEC statement of any job step. |
| **TIME** | If TIME is coded in an EXEC statement, then it applies to that job step only. If it is specified in both JOB and EXEC statement, then both will be in effect and can cause time-out error due to either of it. It is not recommended to use TIME parameter in both the JOB and EXEC statement together. |
| **REGION** | If REGION is coded in an EXEC statement, then it applies to that job step only.  REGION coded in JOB statement overrides the REGION coded in EXEC statement of any job step. |
| **COND** | Used to control the job step execution based on the return-code of the previous step.  If a COND parameter is coded in an EXEC statement of a job step, then the COND parameter of the JOB statement (if present) is ignored. The various tests that can be performed using a COND parameter is explained in conditional Processing. |

Example

Following is a simple example of JCL script along with JOB and EXEC statements:

//TTYYSAMP JOB 'TUTO',CLASS=6,MSGCLASS=X,REGION=8K,

// NOTIFY=&SYSUID

//\*

//STEP010 EXEC PGM=MYCOBOL,PARAM=CUST1000,

// ACCT=(XXXX),REGION=8K,ADDRSPC=REAL,TIME=1440

**DD statement**

Datasets are mainframe files with records organised in a specific format. Datasets are stored on the Direct Access Storage Device (DASD) or Tapes of the mainframe and are basic data storage areas. If these data are required to be used/created in a batch program, then the file (i.e., dataset) physical name along with the file format and organisation are coded in a JCL.

The definition of each dataset used in the JCL is given using the **DD statement**. The input and output resources required by a job step needs to be described within a DD statement with information such as the dataset organisation, storage requirements and record length.

Syntax

Following is the basic syntax of a JCL DD statement:

//DD-name DD Parameters

Description

Let us see the description of the terms used in above DD statement syntax.

DD-NAME

A DD-NAME identifies the dataset or input/output resource. If this is an input/output file used by a COBOL/Assembler program, then the file is referenced by this name within the program.

DD

This is the keyword to identify it as an DD statement.

PARAMETERS

Following are the various parameters for DD statement. You can use one or more parameters based on requirements and they are separated by comma:

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| **DSN** | The DSN parameter refers to the physical dataset name of a newly created or existing dataset. The DSN value can be made up of sub-names each of 1 to 8 characters length, separated by periods and of total length of 44 characters (alphanumeric). Following is the syntax:  **DSN=Physical Dataset Name**  **Temporary datasets** need storage only for the job duration and are deleted at job completion. Such datasets are represented as **DSN=&name** or simply without a DSN specified.  If a temporary dataset created by a job step is to be used in the next job step, then it is referenced as **DSN=\*.stepname.ddname**. This is called **Backward Referencing.** |
| **DISP** | The DISP parameter is used to describe the status of the dataset, disposition at the end of the job step on normal and abnormal completion. DISP is not required in a DD statement only when the dataset gets created and deleted in the same job step (like the temporary datasets). Following is the syntax:  **DISP=(status, normal-disposition, abnormal-disposition)**  Following are valid values for **status**:   * **NEW** : The dataset is newly created by the job step. OUTPUT1 in the example above. * **OLD** : The dataset is already created and will be overwritten in the job step. The job step gains exclusive access on the dataset and no other job can access this dataset until the completion of the job step. * **SHR** : The dataset is already created and will be read in the job step. The dataset can be read by multiple jobs at the same time. Example: INPUT1 and INPUT2. * **MOD** : The dataset is already created. This disposition will be used when there is a need to append new records to the existing dataset (existing records will not be overwritten).   A **normal-disposition** parameter can take one of the following values   * CATLG, UNCATLG, DELETE, PASS and KEEP   A **abnormal-disposition** parameter can take one of the following values   * CATLG, UNCATLG, DELETE and KEEP   Here is the description of CATLG, UNCATLG, DELETE, PASS and KEEP parameters:   * **CATLG** : The dataset is retained with a entry in the system catalog. * **UNCATLG** : The dataset is retained but system catalog entry is removed. * **KEEP** : The dataset is retained without changing any of the catalog entries. KEEP is the only valid disposition for VSAM files. This is to be used only for permanent datasets. * **DELETE** : Dataset is deleted from user and system catalog. * **PASS** : This is valid only for normal disposition. This is used when the dataset is to be passed and processed by the next job step in a JCL   When any of the sub-parameters of DISP are not specified, the default values are as follows:   * **status** : NEW is the default value. * **normal-disposition** : If status is NEW, default normal-disposition is DELETE, else it is KEEP. * **abnormal-disposition** : Same as normal disposition. |
| **DCB** | The Data Control Block (DCB) parameter details the physical characteristics of a dataset. This parameter is required for datasets that are newly created in the job step.  LRECL is the length of each record held within the dataset.  RECFM is the record format of the dataset. RECFM can hold values FB, V or VB. FB is a fixed block organisation where one or more logical records are grouped within a single block. V is variable organisation where one variable length logical record is placed within one physical block. VB is Variable Block organisation where one or more variable length logical records are placed within one physical block.  BLKSIZE is the size of the physical block. The larger the block, greater is the number of records for a FB or VB file.  DSORG is the type of dataset organisation. DSORG can hold values PS (Physical Sequential), PO (Partitioned Organisation) and DA (Direct Organisation).  When there is a need to replicate the DCB values of one dataset to another within the same jobstep or JCL, then it is specified as DCB=\*.stepname.ddname where stepname is the name of the job step and ddname is the dataset from which the DCB is copied.  Check below example where RECFM=FB,LRECL=80 forms the DCB of dataset OUTPUT1. |
| **SPACE** | The SPACE parameter specifies the space required for the dataset in the DASD (Direct Access Storage Disk). Following is the syntax:  **SPACE=(spcunits, (pri, sec, dir), RLSE)**  Here is the description of all the used parameters:   * **spcunits** : This can be one of the CYL(Cylinder), TRK(Tracks) or BLKSIZE(Block Size). * **pri** : This is the primary space required for the dataset. * **sec** : This is the additional space required, when the primary space is not being sufficient. * **ir** : This is the directory blocks required, if the dataset is a PDS (Partitioned Dataset) with members within it. * **RLSE** : This is used to release the unused space at job completion. |
| **UNIT** | The UNIT and VOL parameters are listed in the system catalog for catalogued datasets and hence can be accessed with just the physical DSN name. But for uncataloged datasets, the DD statement should include these parameters. For new datasets to be created, the UNIT/VOL parameters can be specified or the Z/OS allocates the suitable device and volume.  The UNIT parameter specifies the type of device on which the dataset is stored. The device type can be identified using Hardware Address or Device type group. Following is the syntax:  **UNIT=DASD | SYSDA**  Where DASD stands for Direct Access Storage Device and SYSDA stands for System Direct Access and refers to the next available disk storage device. |
| **VOL** | The VOL parameter specifies the volume number on the device identified by the UNIT parameter. Following is the syntax:  **VOL=SER=(v1,v2)**  Where v1, v2 are volume serial numbers. You can use the following syntax as well:  **VOL=REF=\*.DDNAME**  Where REF is the backward reference to the volume serial number of a dataset in any of the preceding job steps in the JCL. |
| **SYSOUT** | The DD statement parameters discussed so far corresponds to data being stored in a dataset. The SYSOUT parameter directs the data to output device based on the class specified. Following is the syntax  **SYSOUT=class**  Where if class is A then it directs output to printer, and if class is **\*** then it directs output to same destination as that of the MSGCLASS parameter in the JOB statement. |

Example

Following is an example, which makes use of DD statements along with various parameters explained above:

//TTYYSAMP JOB 'TUTO',CLASS=6,MSGCLASS=X,REGION=8K,

// NOTIFY=&SYSUID

//\*

//STEP010 EXEC PGM=ICETOOL,ADDRSPC=REAL

//\*

//INPUT1 DD DSN=TUTO.SORT.INPUT1,DISP=SHR

//INPUT2 DD DSN=TUTO.SORT.INPUT2,DISP=SHR,UNIT=SYSDA,

// VOL=SER=(1243,1244)

//OUTPUT1 DD DSN=MYFILES.SAMPLE.OUTPUT1,DISP=(,CATLG,DELETE),

// RECFM=FB,LRECL=80,SPACE=(CYL,(10,20))

//OUTPUT2 DD SYSOUT=\*

**Base Library**

**Base Library** is the Partitioned Dataset (PDS), which holds the load modules of the program to be executed in the JCL or the catalogued procedure, which is called in the program. Base libraries can be specified for the whole JCL in a **JOBLIB** library or for a particular job step in a **STEPLIB** statement.

JOBLIB Statement

A **JOBLIB** statement is used in order to identify the location of the program to be executed in a JCL. The JOBLIB statement is specified after the JOB statement and before the EXEC statement. This can be used only for the in stream procedures and programs.

Syntax

Following is the basic syntax of a JCL JOBLIB statement:

//JOBLIB DD DSN=dsnname,DISP=SHR

The JOBLIB statement is applicable to all the EXEC statements within the JCL. The program specified in the EXEC statement will be searched in the JOBLIB library followed by the system library.

For example, if the EXEC statement is executing a COBOL program, the load module of the COBOL program should be placed within the JOBLIB library.

STEPLIB Statement

A **STEPLIB** statement is used in order to identify the location of the program to be executed within a Job Step. The STEPLIB statement is specified after the EXEC statement and before the DD statement of the job step.

Syntax

Following is the basic syntax of a JCL STEPLIB statement:

//STEPLIB DD DSN=dsnname,DISP=SHR

The program specified in the EXEC statement will be searched in the STEPLIB library followed by the system library. STEPLIB coded in a job step overrides the JOBLIB statement.

Example

The following example shows the usage of JOBLIB and STEPLIB statements:

//MYJCL JOB ,,CLASS=6,NOTIFY=&SYSUID

//\*

//**JOBLIB** DD DSN=MYPROC.BASE.LIB1,DISP=SHR

//\*

//STEP1 EXEC PGM=MYPROG1

//INPUT1 DD DSN=MYFILE.SAMPLE.INPUT1,DISP=SHR

//OUTPUT1 DD DSN=MYFILES.SAMPLE.OUTPUT1,DISP=(,CATLG,DELETE),

// RECFM=FB,LRECL=80

//\*

//STEP2 EXEC PGM=MYPROG2

//**STEPLIB** DD DSN=MYPROC.BASE.LIB2,DISP=SHR

//INPUT2 DD DSN=MYFILE.SAMPLE.INPUT2,DISP=SHR

//OUTPUT2 DD DSN=MYFILES.SAMPLE.OUTPUT2,DISP=(,CATLG,DELETE),

// RECFM=FB,LRECL=80

Here, the load module of the program MYPROG1 (in STEP1) is searched in the MYPROC.SAMPLE.LIB1. If not found, it is searched in the system library. In STEP2, STEPLIB overrides JOBLIB and load module of the program MYPROG2 is searched in MYPROC.SAMPLE.LIB2 and then in the system library.

INCLUDE Statement

A set of JCL statements coded within a member of a PDS can be included to a JCL using an **INCLUDE** statement. When the JES interprets the JCL, the set of JCL statements within the INCLUDE member replaces the INCLUDE statement.

Syntax

Following is the basic syntax of a JCL INCLUDE statement:

//name **INCLUDE** MEMBER=member-name

The main purpose of INCLUDE statement is reusability. For example, common files to be used across many JCLs can be coded as DD statements within INCLUDE member and used in a JCL.

Dummy DD statements, data card specifications, PROCs, JOB, PROC statements cannot be coded within an INCLUDE member. An INLCUDE statement can be coded within an INCLUDE member and further nesting can be done up to 15 levels.

JCLLIB Statement

A **JCLLIB** statement is used to identify the private libraries used in the job. It can be used both with instream and cataloged procedures.

Syntax

Following is the basic syntax of a JCL JCLLIB statement:

//name **JCLLIB** ORDER=(library1, library2....)

The libraries specified in the JCLLIB statement will be searched in the given order to locate the programs, procedures and INCLUDE member used in the job. There can be only one JCLLIB statement in a JCL; specified after a JOB statement and before EXEC and INCLUDE statement but it cannot be coded within an INCLUDE member.

Example

In the following example, the program MYPROG3 and INCLUDE member MYINCL is searched in the order of MYPROC.BASE.LIB1, MYPROC.BASE.LIB2, system library.

//MYJCL JOB ,,CLASS=6,NOTIFY=&SYSUID

//\*

//MYLIB **JCLLIB** ORDER=(MYPROC.BASE.LIB1,MYPROC.BASE.LIB2)

//\*

//STEP1 EXEC PGM=**MYPROG3**

//INC **INCLUDE** MEMBER=MYINCL

//OUTPUT1 DD DSN=MYFILES.SAMPLE.OUTPUT1,DISP=(,CATLG,DELETE),

// RECFM=FB,LRECL=80

**JCL Procedures**

The **JCL Procedures** are set of statements inside a JCL grouped together to perform a particular function. Usually, the fixed part of the JCL is coded in a procedure. The varying part of the Job is coded within the JCL.

You can use a procedure to achieve parallel execution of a program using multiple input files. A JCL can be created for each input file, and a single procedure can be called simultaneously by passing the input file name as a symbolic parameter.

Syntax

Following is the basic syntax of a JCL procedure definition:

//\*

//*Step-name* **EXEC** *procedure name*

The contents of the procedure are held within the JCL for an instream procedure. The contents are held within a different member of the base library for a cataloged procedure. This chapter is going to explain two types of procedures available in JCL and then finally we will see how we can nest various procedures.

Instream Procedure

When the procedure is coded within the same JCL member, it is called an Instream Procedure. It should start with a PROC statement and end with a PEND statement.

//SAMPINST JOB 1,CLASS=6,MSGCLASS=Y,NOTIFY=&SYSUID

//\*

//INSTPROC **PROC** //\*START OF PROCEDURE

//PROC1 EXEC PGM=SORT

//SORTIN DD DSN=&DSNAME,DISP=SHR

//SORTOUT DD SYSOUT=\*MYINCL

//SYSOUT DD SYSOUT=\*

//SYSIN DD DSN=&DATAC LRECL=80

// **PEND** //\*END OF PROCEDURE

//\*

//STEP1 EXEC INSTPROC,DSNME=MYDATA.URMI.INPUT1,

// DATAC=MYDATA.BASE.LIB1(DATA1)

//\*

//STEP2 EXEC INSTPROC,DSNME=MYDATA.URMI.INPUT2

// DATAC=MYDATA.BASE.LIB1(DATA1)

//\*

In the above example, the procedure INSTPROC is called in STEP1 and STEP2 using different input files. The parameters DSNAME and DATAC can be coded with different values while calling the procedure and these are called as **symbolic parameters**. The varying input to the JCL such as file names, datacards, PARM values, etc., are passed as symbolic parameters to procedures.

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.

Cataloged Procedure

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. Following is an example of JCL where it's calling CATLPROC procedure:

//SAMPINST JOB 1,CLASS=6,MSGCLASS=Y,NOTIFY=&SYSUID

//\*

//STEP EXEC **CATLPROC**,PROG=CATPRC1,DSNME=MYDATA.URMI.INPUT

// DATAC=MYDATA.BASE.LIB1(DATA1)

Here, the procedure CATLPROC is cataloged in MYCOBOL.BASE.LIB1. PROG,DATAC and DSNAME are passed as symbolic parameters to the procedure CATLPROC.

//CATLPROC **PROC** PROG=,BASELB=MYCOBOL.BASE.LIB1

//\*

//PROC1 EXEC PGM=&PROG

//STEPLIB DD DSN=&BASELB,DISP=SHR

//IN1 DD DSN=&DSNAME,DISP=SHR

//OUT1 DD SYSOUT=\*

//SYSOUT DD SYSOUT=\*

//SYSIN DD DSN=&DATAC

//\*

Within the procedure, the symbolic parameters PROG and BASELB are coded. Please note that the PROG parameter within the procedure is overridden by the value in the JCL and hence PGM takes the value CATPRC1 during execution.

Nested Procedures

Calling a procedure from within a procedure is called a **nested procedure**. Procedures can be nested up to 15 levels. The nesting can be completely in-stream or cataloged. We cannot code an instream procedure within a cataloged procedure.

//SAMPINST JOB 1,CLASS=6,MSGCLASS=Y,NOTIFY=&SYSUID

//\*

//SETNM SET DSNM1=INPUT1,DSNM2=OUTPUT1

//INSTPRC1 **PROC** //\* START OF PROCEDURE 1

//STEP1 EXEC PGM=SORT,DISP=SHR

//SORTIN DD DSN=&DSNM1,DISP=SHR

//SORTOUT DD DSN=&DSNM2,DISP=(,PASS)

//SYSOUT DD SYSOUT=\*

//SYSIN DD DSN=&DATAC

//\*

//STEP2 EXEC **PROC=INSTPRC2**,DSNM2=MYDATA.URMI.OUTPUT2

// **PEND** //\* END OF PROCEDURE 1

//\*

//INSTPRC2 **PROC** //\* START OF PROCEDURE 2

//STEP1 EXEC PGM=SORT

//SORTIN DD DSN=\*.INSTPRC1.STEP1.SORTOUT

//SORTOUT DD DSN=&DSNM2,DISP=OLD

//SYSOUT DD SYSOUT=\*

//SYSIN DD DSN=&DATAC

// **PEND** //\* END OF PROCEDURE 2

//\*

//JSTEP1 EXEC INSTPRC1,DSNM1=MYDATA.URMI.INPUT1,

// DATAC=MYDATA.BASE.LIB1(DATA1)

//\*

In the above example, the JCL calls the procedure INSTPRC1 in JSTEP1 and procedure INSTPRC2 is being called within the procedure INSTPRC1. Here, the output of INSTPRC1 (SORTOUT) is passed as input (SORTIN) to INSTPRC2.

A **SET statement** is used to define commonly used symbolics across job steps or procedures. It initializes the previous values in the symbolic names. It has to be defined before the first use of the symbolic names in the JCL.

Let's have a look at the below description to understand a little more about the above program:

* SET parameter initializes DSNM1=INPUT1 and DSNM2=OUTPUT1.
* When INSTPRC1 is called in JSTEP1 of JCL, DSNM1=MYDATA.URMI.INPUT1 and DSNM2=OUTPUT1., i.e., the value initialized in SET statement is reset with the value set in any of the job step/procedures.
* When INSTPRC2 is called in STEP2 of INSTPRC1, DSNM1=MYDATA.URMI.INPUT1 and DSNM2=MYDATA.URMI.OUTPUT2.

JCL - Conditional Processing

The Job Entry System uses two approaches to perform conditional processing in a JCL. When a job completes, a return code is set based on the status of execution. The return code can be a number between 0 (successful execution) to 4095 (non-zero shows error condition). The most common conventional values are:

* 0 = Normal - all OK
* 4 = Warning - minor errors or problems.
* 8 = Error - significant errors or problems.
* 12 = Severe error - major errors or problems, the results should not be trusted.
* 16 = Terminal error - very serious problems, do not use the results.

A job step execution can be controlled based on the return code of the previous step(s) using the **COND** parameter and **IF-THEN-ELSE** construct, which has been explained in this tutorial.

COND parameter

A **COND** parameter can be coded in the JOB or EXEC statement of JCL. It is a test on the return code of the preceding job steps. If the test is evaluated to be true, the current job step execution is bypassed. Bypassing is just omission of the job step and not an abnormal termination. There can be at most eight conditions combined in a single test.

Syntax

Following is the basic syntax of a JCL COND Parameter:

COND=(rc,logical-operator)

or

COND=(rc,logical-operator,stepname)

or

COND=EVEN

or

COND=ONLY

Here is the description of parameters used:

* **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 tutorial.

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

COND inside JOB statement

When COND is coded in JOB statement, the condition is tested for every job step. When the condition is true at any particular job step, it is bypassed along with the job steps following it. Following is an example:

//CNDSAMP JOB CLASS=6,NOTIFY=&SYSUID,COND=(5,LE)

//\*

//STEP10 EXEC PGM=FIRSTP

**//\* STEP10 executes without any test being performed.**

//STEP20 EXEC PGM=SECONDP

**//\* STEP20 is bypassed, if RC of STEP10 is 5 or above.**

**//\* Say STEP10 ends with RC4 and hence test is false.**

**//\* So STEP20 executes and lets say it ends with RC16.**

//STEP30 EXEC PGM=SORT

**//\* STEP30 is bypassed since 5 <= 16.**

COND inside EXEC statement

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.**

COND=EVEN

When COND=EVEN is coded, the current job step is executed, even if any of the previous steps abnormally terminate. If any other RC condition is coded along with COND=EVEN, then the job step executes if none of the RC condition is true.

//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),EVEN)

**//\* In STP03, condition (10,LT,STP01) evaluates to true,**

**//\* hence the step is bypassed.**

COND=ONLY

When COND=ONLY is coded, the current job step is executed, only when any of the previous steps abnormally terminate. If any other RC condition is coded along with COND=ONLY, then the job step executes if none of the RC condition is true and any of the previous job steps fail abnormally.

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

//\*

//STP01 EXEC PGM=SORT

**//\* Assuming STP01 ends with RC0.**

//STP02 EXEC PGM=MYCOBB,COND=(4,EQ,STP01)

**//\* In STP02, condition evaluates to FALSE, step is executed**

**//\* and assume the step abends.**

//STP03 EXEC PGM=IEBGENER,COND=((0,EQ,STP01),ONLY)

**//\* In STP03, though the STP02 abends, the condition**

**//\* (0,EQ,STP01) is met. Hence STP03 is bypassed.**

IF-THEN-ELSE Construct

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.

Syntax

Following is the basic syntax of a JCL IF-THEN-ELSE Construct:

//name IF condition THEN

list of statements //\* action to be taken when condition is true

//name ELSE

list of statements //\* action to be taken when condition is false

//name ENDIF

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 @.
* **Condition** : A condition 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 (<, <=, >, >=, <>).

Example

Following is a simple example showing the usage of IF-THEN-ELSE:

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

//\*

//PRC1 PROC

//PST1 EXEC PGM=SORT

//PST2 EXEC PGM=IEBGENER

// PEND

//STP01 EXEC PGM=SORT

//IF1 IF STP01.RC = 0 THEN

//STP02 EXEC PGM=MYCOBB1,PARM=123

// ENDIF

//IF2 IF STP01.RUN THEN

//STP03a EXEC PGM=IEBGENER

//STP03b EXEC PGM=SORT

// ENDIF

//IF3 IF STP03b.!ABEND THEN

//STP04 EXEC PGM=MYCOBB1,PARM=456

// ELSE

// ENDIF

//IF4 IF (STP01.RC = 0 & STP02.RC <= 4) THEN

//STP05 EXEC PROC=PRC1

// ENDIF

//IF5 IF STP05.PRC1.PST1.ABEND THEN

//STP06 EXEC PGM=MYABD

// ELSE

//STP07 EXEC PGM=SORT

// ENDIF

Let's try to look into the above program to understand it in little more detail:

* The return code of STP01 is tested in IF1. If it is 0, then STP02 is executed. Else, the processing goes to the next IF statement (IF2).
* In IF2, If STP01 has started execution, then STP03a and STP03b are executed.
* In IF3, If STP03b does not ABEND, then STP04 is executed. In ELSE, there are no statements. It is called a NULL ELSE statement.
* In IF4, if STP01.RC = 0 and STP02.RC <=4 are TRUE, then STP05 is executed.
* In IF5, if the proc-step PST1 in PROC PRC1 in jobstep STP05 ABEND, then STP06 is executed. Else STP07 is executed.
* If IF4 evaluates to false, then STP05 is not executed. In that case, IF5 are not tested and the steps STP06, STP07 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.

Setting Checkpoints

You can set checkpoint dataset inside your JCL program using **SYSCKEOV,** which is a DD statement.

A **CHKPT** is the parameter coded for multi-volume QSAM datasets in a DD statement. When a CHKPT is coded as CHKPT=EOV, a checkpoint is written to the dataset specified in the SYSCKEOV statement at the end of each volume of the input/output multi-volume dataset.

//CHKSAMP JOB CLASS=6,NOTIFY=&SYSUID

//\*

//STP01 EXEC PGM=MYCOBB

//SYSCKEOV DD DSNAME=SAMPLE.CHK,DISP=MOD

//IN1 DD DSN=SAMPLE.IN,DISP=SHR

//OUT1 DD DSN=SAMPLE.OUT,DISP=(,CATLG,CATLG)

// CHKPT=EOV,LRECL=80,RECFM=FB

In the above example, a checkpoint is written in dataset SAMPLE.CHK at the end of each volume of the output dataset SAMPLE.OUT.

Restart Processing

You can restart processing ether using automated way using the **RD parameter** or manual using the **RESTART parameter**.

**RD parameter** is coded in the JOB or EXEC statement and it helps in automated JOB/STEP restart and can hold one of the four values: R, RNC, NR or NC.

* **RD=R** allows automated restarts and considers the checkpoint coded in the CHKPT parameter of the DD statement.
* **RD=RNC** allows automated restarts, but overrides (ignores) the CHKPT parameter.
* **RD=NR** specifies that the job/step cannot be automatically restarted. But when it is manually restarted using the RESTART parameter, CHKPT parameter (if any) will be considered.
* **RD=NC** disallows automated restart and checkpoint processing.

If there is a requirement to do automated restart for specific abend codes only, then it can be specified in the **SCHEDxx** member of the IBM system parmlib library.

**RESTART parameter** is coded in the JOB or EXEC statement and it helps in manual restart of the JOB/STEP after the job failure. RESTART can be accompanied with a checkid, which is the checkpoint written in the dataset coded in the SYSCKEOV DD statement. When a checkid is coded, the SYSCHK DD statement should be coded to reference the checkpoint dataset after the JOBLIB statement (if any), else after the JOB statement.

//CHKSAMP JOB CLASS=6,NOTIFY=&SYSUID,RESTART=(STP01,chk5)

//\*

//SYSCHK DD DSN=SAMPLE.CHK,DISP=OLD

//STP01 EXEC PGM=MYCOBB

//\*SYSCKEOV DD DSNAME=SAMPLE.CHK,DISP=MOD

//IN1 DD DSN=SAMPLE.IN,DISP=SHR

//OUT1 DD DSN=SAMPLE.OUT,DISP=(,CATLG,CATLG)

// CHKPT=EOV,LRECL=80,RECFM=FB

In the above example, chk5 is the checkid, i.e., STP01 is restarted at checkpoint5. Please note that a SYSCHK statement is added and SYSCKEOV statement is commented out in the previous program explained in Setting Checkpoint section.