UNIT 1

INTRODUCTION TO JCL

Introduction to JCL

- Objectives
- What is JCL?
- How MVS handles Jobs
- Initiators
- Structure of JCL
- Parameters

Schematic representation of a Job-flow

Objectives

- Understand the need for JCL.
- Understand the concept of a Job.
- Visualize the processing flow of a Job.
- Understand the structure of coding JCL.
- Differentiate between the different types of parameters used in JCL.

What is JCL?

JCL stands for **Job Control Language.**

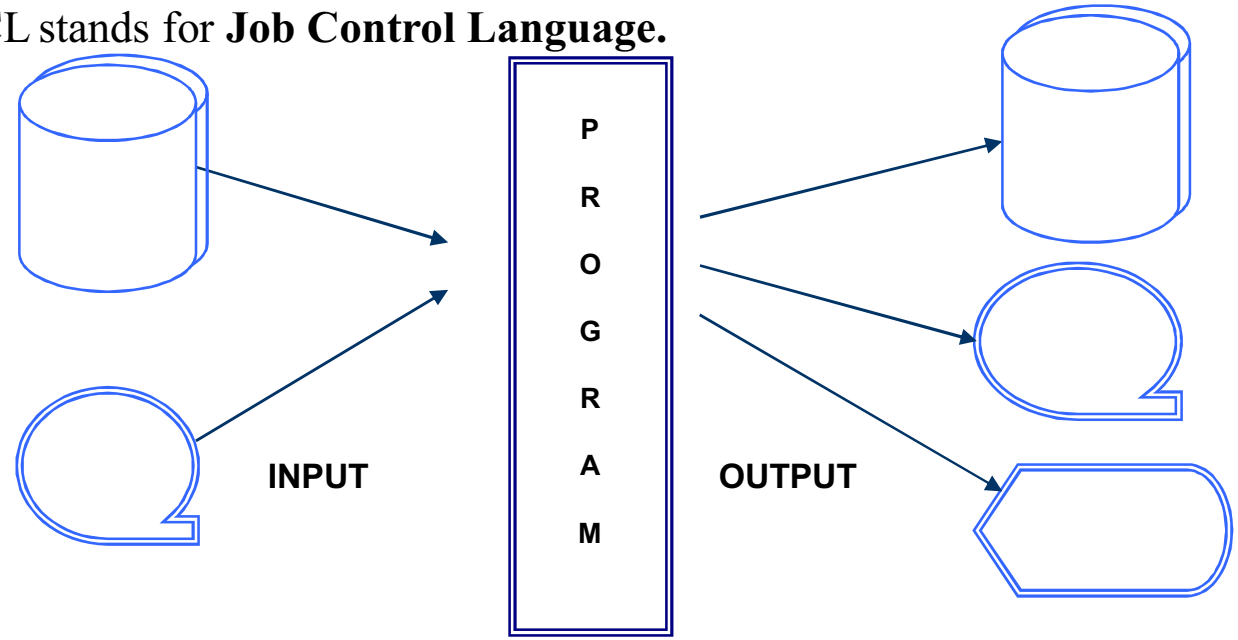


Figure: 1-1

JCL is a language that coordinates the activities of batch programs on the operating system. Using JCL you can tell the system:

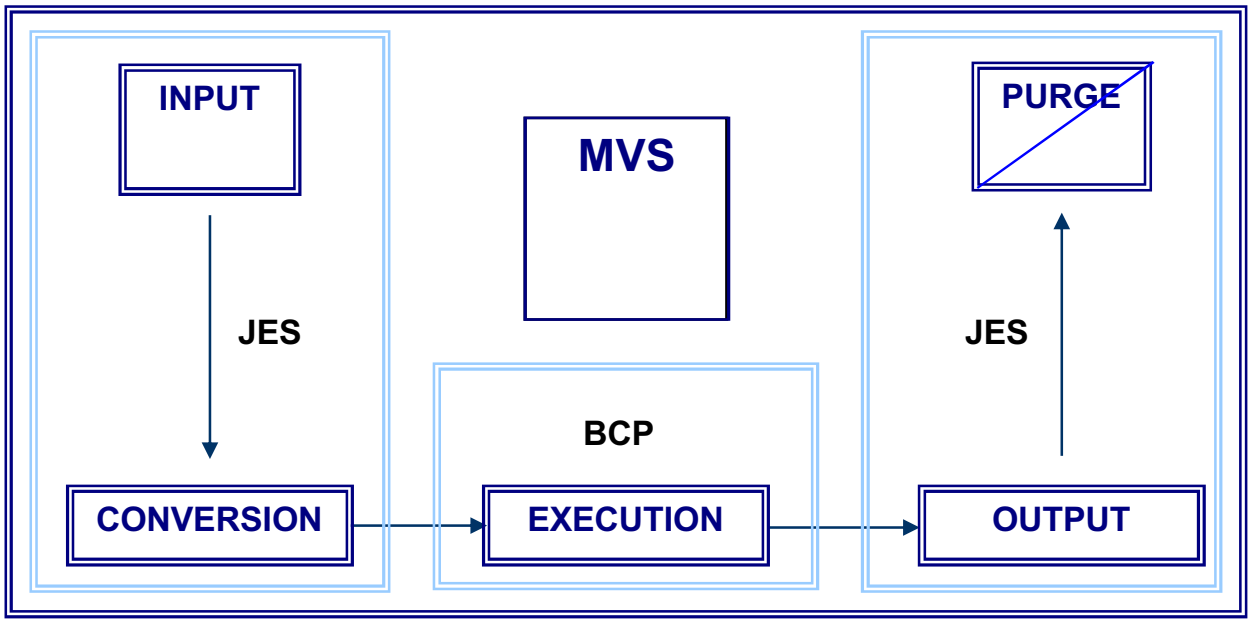
What program(s) you want to execute.

What input and output datasets these program(s) will be using.

What should be done with the datasets once the program(s) end(s).

A Batch Job is a set of JCL statements that are used to input this information to the system.

How MVS handles Jobs



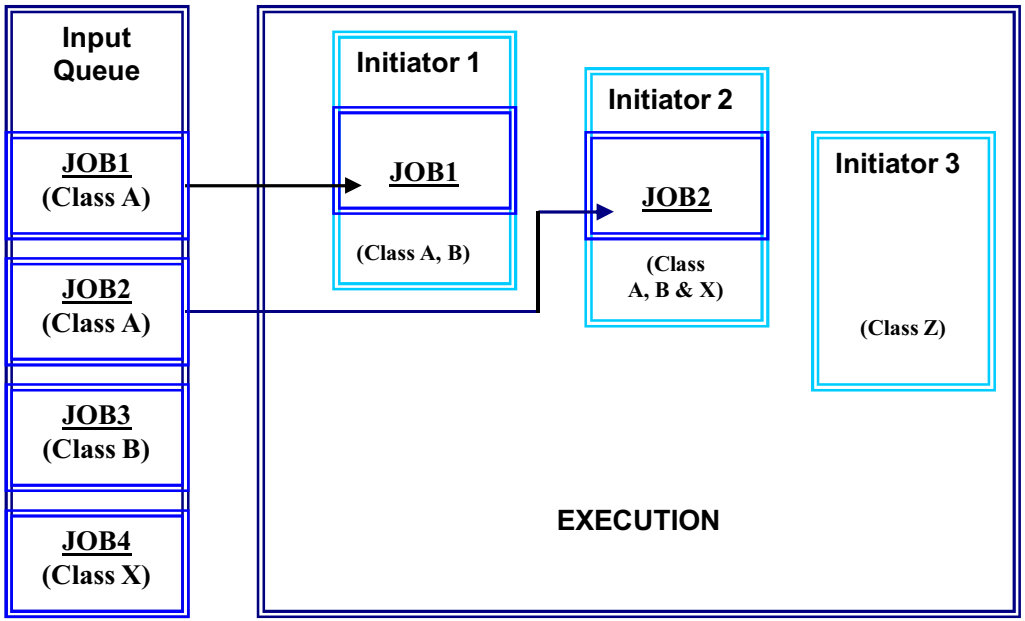
Job Entry Subsystem (JES) handles the Jobs ‘before’ and ‘after’ the execution.

The Base Control Program (BCP) controls when a job gets the CPU for processing (this is called dispatching).

The various stages of a Job while processing are:

- **Input:** The Job is put into JES Input Queue waiting to be dispatched.
- **Conversion**: The Job is checked for Syntax and Validity.
- **Execution:** BCP handles the dispatching of the Job based on priority.
- **Output:** Job is sent to JES Output Queue from where it will be sent for printing.
- **Purge:** Job is in the JES Output Queue from where it can be purged from the system.

Initiators



- An Initiator is a special Address Space to which Batch Jobs are mapped during execution.
- A system has a predetermined number of Initiators.
- A Job must be mapped to an Initiator in order to be executed.

- Each Initiator is associated with one or more Classes and can take Jobs with those Classes only (Each Job has one Class).

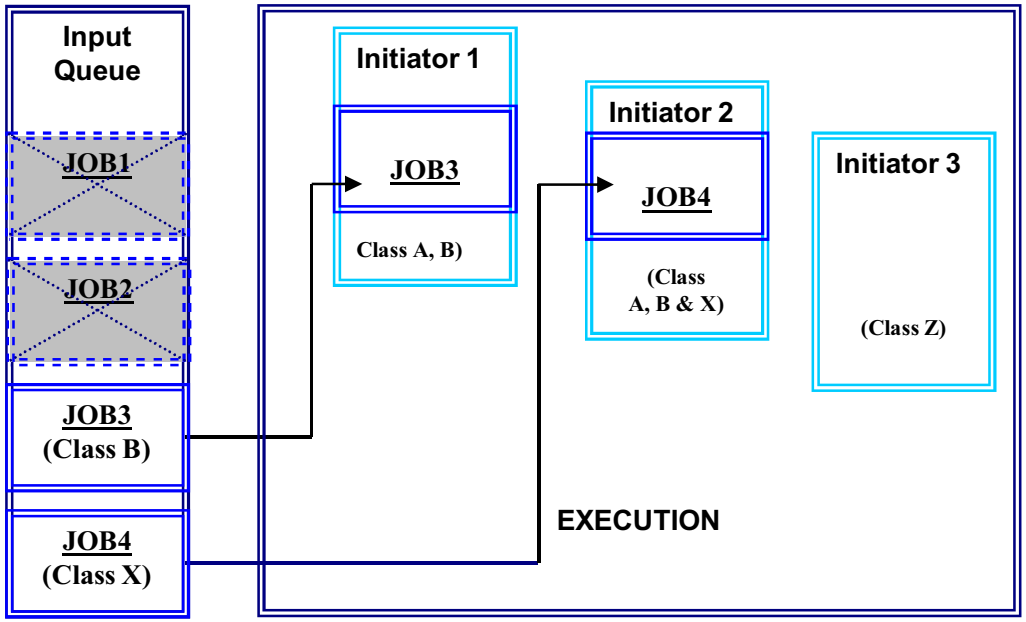
- An Initiator holds a Job until the execution of the entire Job is over. Only then can it take on any other waiting Job.

  In Figure 1-3 for example,

- Jobs Job1 and Job2 get mapped to Initiator1 and Initiator2 respectively since their Classes match with those of the Initiators,

- Job Job3 will have to wait for an Initiator, as, even though Initiator3 is free, it cannot take a Job of Class 'B'. Same case with Job4.

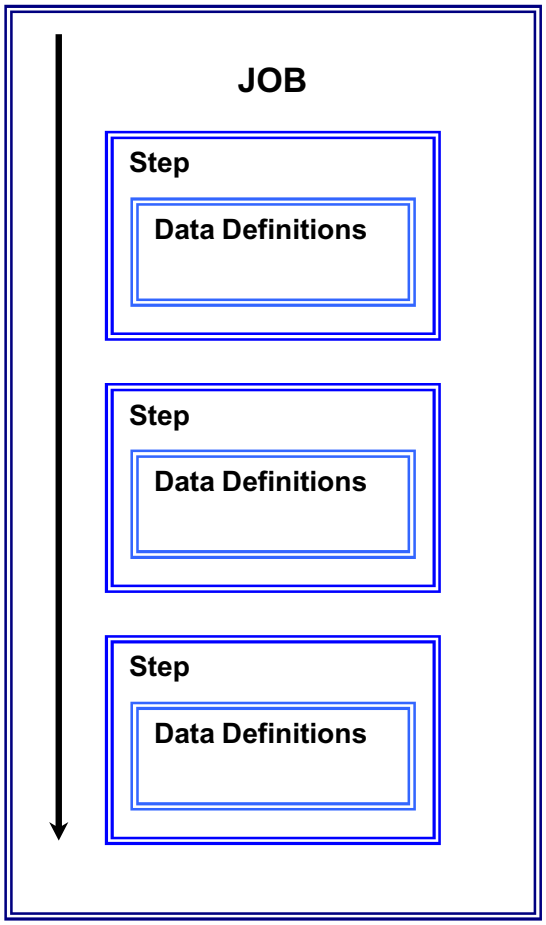
Initiators (Continued)



What happens after Jobs Job1 and Job2 finish execution?

- Initiator1 and Initiator2 are now free,
- Job3 is the next Job in the Input Queue, so, it gets associated with Initiator1,
- Similarly, Job4 gets mapped to Initiator2,
- Initiator3 remains idle waiting for a Job with Class ‘Z’.

Structure of JCL



- A Job contains one or more Steps.
- Each Step signifies the execution of a Program.
- Each Step has its own set of Data Definitions.
  These define the datasets to be used in the step.
- All the Steps in a Job are executed sequentially.
- Max No. of steps can be up to 255.
- Max No. of DD statements can be up to 3273.

Structure of JCL (Continued 1)

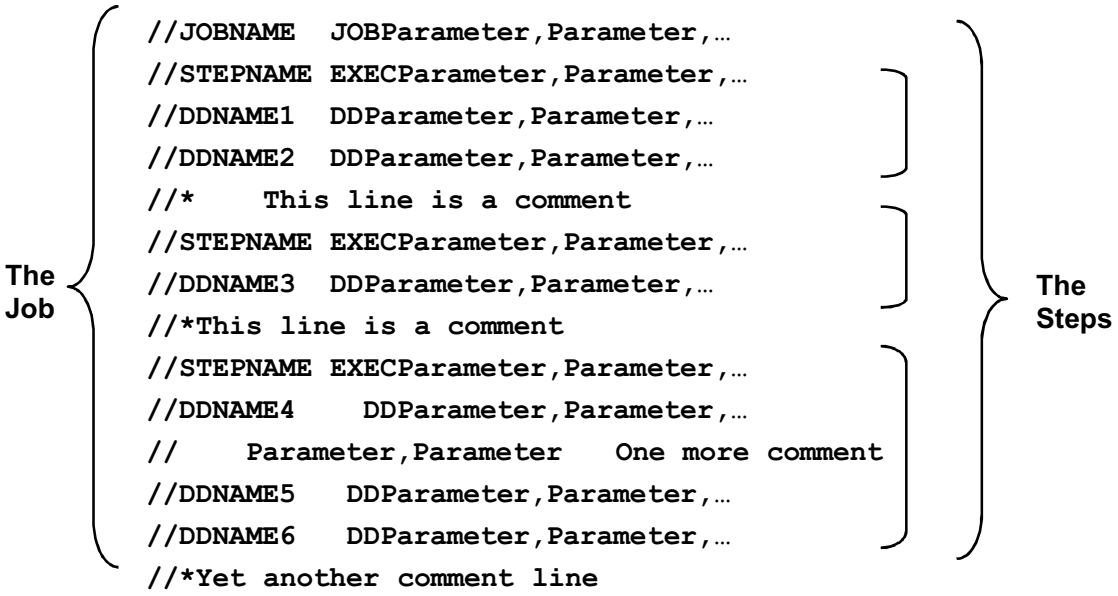


Figure: 1-6

A Job generally looks as displayed above.

There are three primary operations (statement types) in JCL: JOB, EXEC and DD.

A JOB statement signifies the beginning of a Job and also describes the attributes for the entire Job in the form of parameters.

An EXEC statement signifies the beginning of a Step (Execution of a program) and, optionally, the attributes of that Step.

All DD statements following an EXEC are the data definitions for that step. They identify the input and output datasets in the step.

These are only a few of the statements available in JCL. There are more, which will be discussed later in this course.

Each statement accepts a list of parameters. Some parameters are optional, others are mandatory.

Structure of JCL (Continued 2)

A JCL statement consists of one or more 80-byte records.

A Continued JCL statement can begin in column 4-16.

Each JCL statement is divided into five fields:

1. IDENTIFIER Field (‘//’, ‘/*’, or ‘//*’)

Indicates a JCL statement and its type. They are:

- a. // in columns 1 and 2, define all JCL statements except the delimiters and comments.
- b. /* in columns 1 and 2 as a delimiter. This signifies the end of in-stream control statements. Installation-designated characters may also be used.
- c .//* in columns 1,2, and 3 depicts a comment statement

2. NAME Field

Identifies a statement so that it can be referred to later. The name field identifies the name of the job, the name of the step, etc,:

- Must begin in column 3.
- 1-8 characters in length (alphanumeric or national (#, @, $))
- First character must be alphabetic or national.
- Must be followed by at least one blank.

Example:

```
//ABC     JOB   'MY ACCOUNT','PROGRAMMER NAME',CLASS=A
```

The example above shows the beginning of job 'ABC', as defined in the name field of the Job statement.

3. OPERATION Field

Specifies the type of statement or command (e.g. JOB, EXEC, DD, etc.):

i. Follows the NAME Field

ii. Must be preceded and followed by at least one blank.

Examples:

```
//ABC     JOB   'MY ACCOUNT','PROGRAMMER NAME',CLASS=A
//STEP01 EXEC   PGM=USERPGM
```

The first example contains the 'JOB' operation, which identifies the JCL statement as a JOB card.

The second example contains the 'EXEC' operation, which identifies the statement as a step (called STEP01).

4. PARAMETER Field

Contains parameters separated by commas:

i. Follows the OPERATION field.

ii. Must be preceded and followed be at least one blank

iii. Consists of two types:

iv. Positional

v. Keyword

Example:

```
//ABC     JOB   'MY ACCOUNT','PROGRAMMER NAME',CLASS=A
```

The above JOB statement contains three parameters. Notice that they are each separated by a comma, with no spaces in between.

5. COMMENT Field:

Comments can contain any information and are used to document elements of the JCL. The Comment Field:

i. Follows the PARAMETER field

ii. Must be preceded by at least one blank. Comment statement is recommended.

Example:

```
//ABC    JOB  'MY ACCOUNT','PROGRAMMER NAME',CLASS=A
//STEP01 EXEC  PGM=APGEN  GENERATES THE MONTHLY A/P REPORT
```

Anything to the right of a space that follows the last parameter on the line is a comment. Notice that there are no parameters following PGM=APGEN, so everything else on the line is treated as a comment.

```
//ABC    JOB  'MY ACCOUNT','PROGRAMMER NAME',CLASS=A
//STEP01 EXEC  PGM=APGEN
//*  THIS STEP GENERATES THE MONTHLY A/P REPORT
```

The Comment statement is also used to document the purpose of JCL statements, or of the job itself.

Syntax

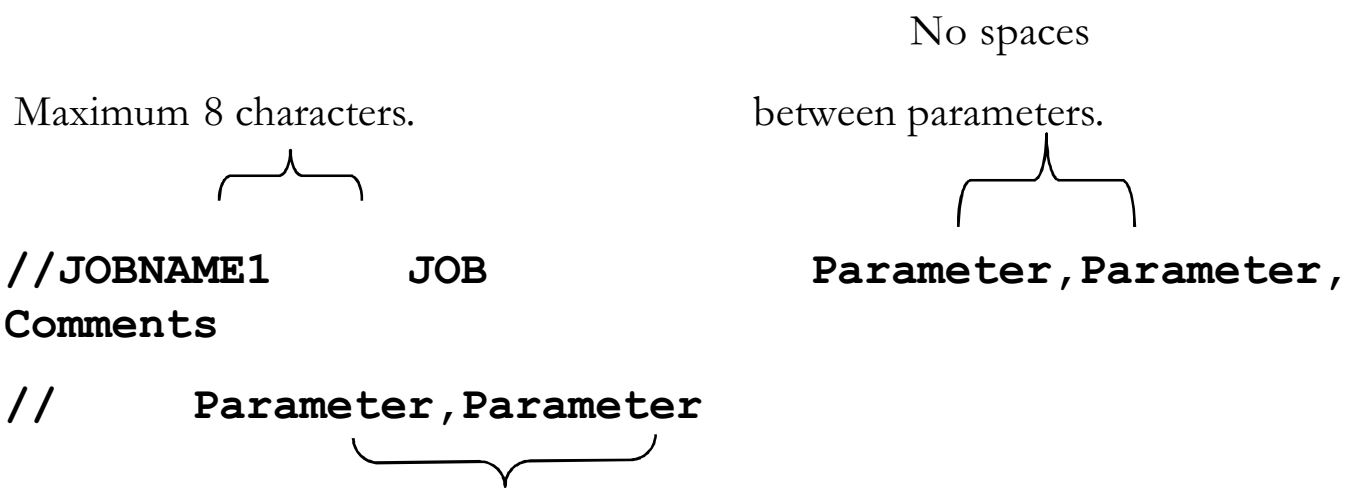


Figure: 1-7

Figure 1-7 shows an example of how a JOB statement might be coded.

Important points to note:

All JCL statements start with two forward-slashes (//) starting from column 1.

Code the name for the statement (optional). Do not leave any blanks between the slashes and the name assigned to the statement.

Leave at least one blank and code the Operation (In this case, 'JOB').

Leave no blanks between two parameters.

The parameters for a single statement can span across multiple lines.

Parameters continued from a previous line must always begin between columns 4 and 16.

Separate the individual parameters by a comma (,).

If two forward-slashes followed by an asterisk (//*) are found starting from column 1,

the entire line is treated as a comment.

Caution: If a blank is left between two parameters on the same line, all the parameters

on that line following the blank are treated as comment entries.

Parameters

There are two types of parameters: Positional parameters and Keyword parameters.

1) Positional parameters

If coded, must appear in a particular order.

2) Keyword parameters

Can appear in any order.

Example:

```
//JOBNAME  JOB A,B,X=1,Y=2,Z=3
```

- he above example, assume A and B to be positional parameters and, X,Y and Z to be Keyword parameters.

- Hence, the statement may also be coded as,

```
//JOBNAME   JOB A,B,Z=3,Y=2,X=1
```

But NOT as,

```
//JOBNAME   JOB B,A,X=1,Y=2,Z=3
```

If you do not want to code any value for the first positional parameter (A) then, you have to notify the absence by coding a positional comma before (B), i.e.

```
//JOBNAME   JOB ,B,Y=2,X=1,Z=3
```

Observe that keyword parameters are usually identifiable by an equal-to (=) operator. In other words, they accept values in the form of one or more constants or parameters.

Subsets of these two types of parameters are listed in the next page.

Parameters (Continued)

Positional Keyword parameters

Even though PGM is a keyword parameter (accepts program name), if it is coded, it must appear immediately after EXEC.

Example:

```
//STEPNAME EXEC PGM=Program1
```

Keyword parameters that accept positional sub-parameters

There are sub-parameters that may also be positional.

Example:

```
//DDNAME DD DISP=(A,B,C)
```

CANNOT be coded as

```
//DDNAME DD DISP=(B,C,A)
```

Keyword parameters that accept keyword parameters

There are also sub-parameters that are not positional.

Example:

```
//DDNAME  DD DCB=(A=1,B=2)
```

CAN be coded as

```
//DDNAME  DD DCB=(B=2,A=1)
```

Schematic representation of a Job-flow

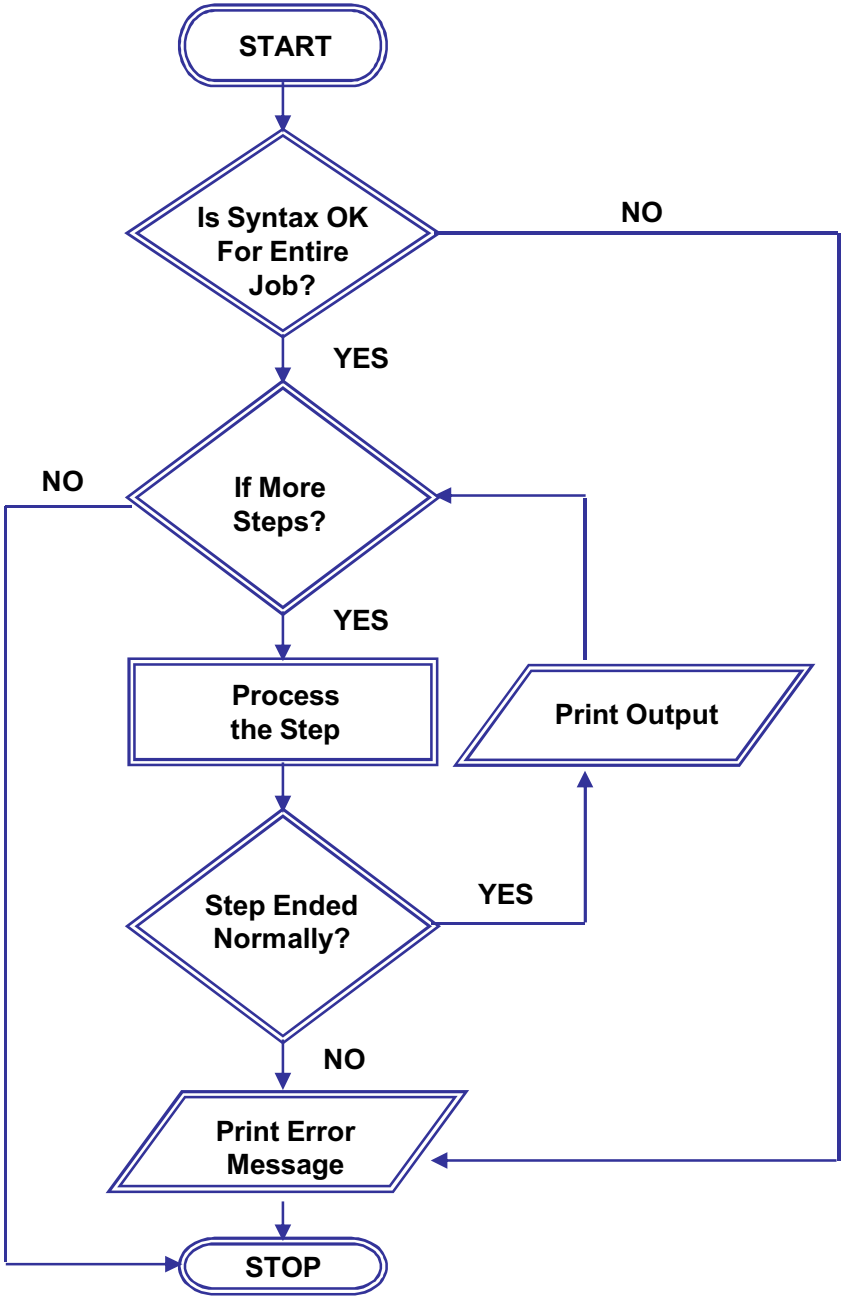


Figure: 1-8

When a job is submitted, JES prepares it for execution, and temporarily stores its information on DASD until OS/390 is ready to accept it.

When an initiator becomes available, JES selects the job for OS/390 execution. As OS/390 executes the JOB there is syntax check.

If a JCL syntax error is detected, the system bypasses the entire job, notify the user and does not attempt job execution.

Otherwise the job is executed step by step, simultaneously printing the output after execution of each step.

Unit 1 Exercises

Complete the following:

1. Which of the following is not a stage during a Job's processing? (*Circle One*)

   A. Input

   B. Conversion

   C. Execution

   D. Substantiation

   E. Output

   F. Purge

2. The MVS subsystem that handles a Job before and after execution is called

   _____

3. A special Address Space to which Batch Jobs are mapped during execution is called a

   _____

4. In order for a Job to use an Initiator, the Job must have the same _____ as the initiator.

5. The _ statement signifies the beginning of a Job.

6. The _ statement signifies the beginning of a step.

7. The _ statement contains the data definitions for that step.

8. Which of the following is not a field in a JCL statement? *(Circle One)*

A. The Identifier field

B. The Summary field

C. The Name field

D. The Operation field

E. The Parameter field

F. The Comment field

UNIT 2

The JOB Statement

The JOB Statement

Objectives

Purpose of the JOB statement

Syntax

Parameters

- Accounting information and Programmer name
- CLASS
- MSGCLASS
- NOTIFY
- MSGLEVEL
- REGION
- TIME
- RESTART
- TYPRUN

Objectives

- Understand what the JOB statement accomplishes
- Learn the most important parameters for the JOB statement
- Code different JOB statements depending on the need of the Job

Purpose of the JOB statement

The JOB statement is needed for the following reasons

- To identify the Job and the person who submitted the Job.
- Specify which Class the Job is to belong to.
- Specify how the output should be handled.
- To make an entity or group accountable for the execution of the Job.

The JOB statement should always be the first statement in a Job.

You can code more than one Job in the same member, though it is not usually recommended.

Syntax

The JOB statement accepts both positional and keyword parameters.

Example:

```
//ABC     JOB   'MY ACCOUNT','PROGRAMMER NAME',CLASS=A
```

- You can use the same Jobname for more than one of your Jobs.

- When a job is submitted to the CPU, it is assigned a unique JES number. Thus, if multiple jobs with the same Jobname are submitted to the CPU, they are tracked individually by their JES number.

Parameters – Accounting Information and Programmer Name

Parameter Type: Positional parameters

Accounting information and Programmer‘s name are both positional parameters.

Accounting information appears first followed by Programmer name.

Accounting Information: Identifies the account code to be used for the job. This is primarily used for billing purposes.

An installation dependent value ranging from 1 to 143 characters.

A comma will replace absence of the accounting Information field.

Programmer Name: Identifies the person or department who is responsible for the job.

A character string ranging from 1 to 20 characters.

A comma will replace absence of the Programmer Name field.

Appears at the bottom of every page of printed output for the Job.

Examples:

//MYJOB JOB MTPLACC25,’JOHN SMITH’,…….

//YOURJOB JOB ,’SUSIE JOHNSON’,….

//HISJOB JOB MTPLAC30,,……..

Parameter – CLASS

Type: Keyword parameter

Purpose: To assign a Class to the Job.

Syntax:

//JOBNAME JOB ACCTINFO,'PROGRAMMER NAME',CLASS=A,..

CLASS=Class

(1 Character. Installation dependent. 'A' thru 'Z' and '0' thru '9')

Example:

```
//ABC     JOB  ACCT123,'PROGRAMMER NAME',CLASS=A,…….
```

Job ABC will execute in class 'A'.

- As we have seen in Unit 1, the Class of a Job determines the input priority (a basis for Initiator selection)
- As most installations do not prefer the users specifying the dispatching priority for their Jobs(a different parameter), the CLASS parameter provides a less explicit way to specify the input priority. For example, the user might select a Class that most Initiators can handle.

Parameter – MSGCLASS

Type: Keyword parameter

Purpose: To specify how the Job's output is to be handled.

Syntax:

```
//JOBNAME    JOB   ACCTINFO,'PROGRAMMER ,
                   NAME',CLASS=A,MSGCLASS=A..
```

MSGCLASS=Message-class

(1 character. Installation Dependent. 'A' thru 'Z' of '0' thru '9')

Example:

```
//ABC      JOB   ACCT123,'PROGRAMMER NAME',MSGCLASS=A
```

- Using the MSGCLASS parameter, you can tell the system how to handle the output of your Job.
- The valid values that you can specify for this parameter are dependent on your installation.

In one installation, MSGCLASS=Z may mean that the output has to be immediately routed to a printer. In another installation, it may mean that the output has to be held in the Output Queue for later retrieval.

Parameter – NOTIFY

Type: Keyword parameter

Purpose: To specify which user is to be notified after the Job finishes execution.

Syntax:

```
//JOBNAME    JOB  ACCT123,'PROGRAMMER NAME',NOTIFY=IBMUSER
```

NOTIFY=User-Id

(Any valid User ID defined to your security system)

`

Examples:

```
//ABC     JOB  ACCT123,'PROGRAMMER NAME',NOTIFY=IBMUSER
```

This example will notify the user id MAIN006 when the job completes

```
//ABC     JOB  ACCT123,'PROGRAMMER NAME',NOTIFY=&SYSUID
```

The value '&SYSUID' indicates that the user id, which submitted the job, should be notified when the job completes.

The NOTIFY parameter can used be when someone needs to be notified when a job completes, along with its maximum return-code (discussed later) or Abend code.

Once the Job finishes, if the user specified in NOTIFY has logged on, he or she will get the message once he or she presses ENTER or any other Attention Identifier Key. If not logged on, the user will get the message(s) along with the logon prompts during the next login procedure.

Parameter – MSGLEVEL

Type: Keyword parameter accepting 2 positional parameters.

Purpose: To specify how much output has to be generated for this Job by the system.

Syntax:

```
//ABC     JOB  ACCT123,'PROGRAMMER,
              NAME',MSGLEVEL=(a,b)
```

MSGLEVEL=(a,b)

a: Specifies how much of the Job has to be printed in the output (job log).

Valid values are 0, 1 and 2

0 – Print only the Jobcard (JOB statement of the Job)

1 – Print all JCL and JES statements including all statements in procedures, if any

2 – Print only submitted JCL and JES statements. Statements in procedures are not printed.

b: Specifies which messages will be printed in the output (job log).

Valid values are 0 and 1

0 – Print the entire messages only if the Job abends (abnormal termination).

1 – Print all the messages regardless of the outcome of the Job how the job terminates

Examples:

```
//ABC    JOB  ACCT123,'PROGRAMMER NAME',MSGLEVEL=(1,1)

//ABC    JOB  ACCT123,'PROGRAMMER NAME',MSGLEVEL=(,1)

//ABC    JOB  ACCT123,'PROGRAMMER NAME',MSGLEVEL=(2,)
```

Parameter – REGION

Type: Keyword parameter

Purpose: To limit the maximum amount of memory that the Job can utilize.

Syntax:

```
//JOBNAME  JOB  ACCTINFO,'PROGRAMMER NAME',REGION=nnnnM
```

REGION=nnnnnnnK or REGION=nnnnM

n:Numeric value

Valid ranges:

0 thru 2047000 in case of K

0 thru 2047 in case of M

K: Kilobytes M: Megabytes

Examples:

```
//ABC    JOB  ACCT123,'PROGRAMMER NAME',REGION=1024K
//ABC    JOB  ACCT123,'PROGRAMMER NAME',REGION=1M
```

- REGION need not be explicitly coded unless specified in the documentation for certain utilities.
- If REGION=0K or REGION=0M is coded, the system assumes a default region size.

Parameter – TIME

Type: Keyword parameter

Purpose: The TIME parameter can be used to specify the maximum length of CPU time that a job or job step is to use the processor.

Syntax:

```
//JOBNAME   JOB   ACCTINFO,'PROGRAMMER NAME',TIME(m,ss)
```

TIME = (minutes, seconds)

ON JOB STATEMENT - Indicates the maximum processing time for the JOB.

ON EXEC STATEMENT - Indicates the maximum processing time for the STEP.

Special Values

Any combination of minutes and seconds can be coded. However, there are three values for the parameter that have a special meaning:

TIME=NOLIMIT means the job or step is not to be timed.

TIME=1440 TIME=1440 is equivalent to coding TIME=NOLIMIT, indicating that the job will not be timed. Thus 1440 is the only numerical value of TIME that is not to be interpreted literally.

TIME=MAXIMUM means the job or step will be allowed to use to 357912 minutes. (This is about 8.25 months of processor time)

- If coded on the JOB statement and any step causes the total time to be exceeded, the job is abnormally terminated. If not coded, the JES installation default is used.

Examples - TIME Parameter

//JOBNAME JOB TIME=(1,30)

TERMINATE JOB IF TOTAL TIME FOR ALL STEPS EXCEEDS 1 MINUTE AND 30 SECONDS

//STP1 EXEC PGM = ABC, TIME=1

//STP2 EXEC PGM=BAKER, TIME=(,20)

TERMINATE THIS STEP IF TIME EXCEEDS 20 SECONDS

DO NOT TIME THIS STEP

//STP3 EXEC PGM=CHARLTE, TIME=NOLIMIT

- The entire job will be limited to a maximum of 1 minute and 30 seconds. The first step will be limited to a maximum of 1 minute.

- Note that the submitter is apparently asking that the last step not be timed. However, coding TIME=NOLIMIT on the last step will NOT accomplish this objective. The TIME=(1,30) on the JOB statement will limit the maximum time for step 3 to 1 minute and 30 seconds.

- If the time consumed by a job (or step) exceeds the value specified for the TIME parameter for that job (or step), the job ABENDS.

Parameter – RESTART

Type: Keyword parameter

Purpose: The RESTART parameter allows you to begin the execution of the Job from a Step other than the first one.

Syntax:

//JOBNAME JOB ACCTINFO,'PROGRAMMER ,

NAME',RESTART=Stepname

Example:

//ABC JOB ACCT123,'PROGRAMMER NAME',RESTART=STEP2

· If you want to skip one or more of the initial steps in the Job before starting execution, use the RESTART parameter.

· For example, if you have run a Job with 2 steps, the first step runs and the second step abends. You make the changes in the second step to set right any error causing statement or parameter. Now, you will require running only the second step. Therefore, you change the JOB statement and add the RESTART parameter to start execution from the second step onwards.

Parameter – TYPRUN

Type: Keyword parameter

Purpose: The TYPRUN parameter controls the type of execution for the Job.

Syntax:

//JOBNAME JOB ACCTINFO,'PROGRAMMER NAME',TYPRUN=typerun,...

TYPRUN=SCAN TYPRUN=SCAN Only check the Job for syntax errors. Do not execute the Job.This is a common way to debug errors in your JCL. The output, including error messages, if any, is directly sent to the Output Queue.

TYPRUN=HOLD TYPRUN=HOLD Check the Job for syntax errors and, if no errors are present, hold it in the Input Queue. The job will execute when a person (usually an operator) or another program releases the hold.

TYPRUN=COPY TYPRUN=COPY With COPY, no syntax checking or execution takes place.

The source content of the Job immediately gets directed to the Output Queue. The JCL is copied as submitted to the sysout class specified in the MSGCLASS parameter (JES 2 ONLY).

Example: *//JOB1 JOB ACCT2254,'MARK JOHN',TYPRUN=SCAN,...*

//JOB2 JOB ACCT2112,'MARK JOHN',TYPRUN=HOLD,...

//JOB3 JOB ACCT634,'MARK JOHN',TYPRUN=COPY,...

Examples of JOB statements

Following are a few sample JOB statements

Sample1

```
//USERJOB  JOB  ACCT54,'JOE SMITH',TYPRUN=SCAN,
//      MSGCLASS=X,CLASS=A,
//      MSGLEVEL=(1,1)
```

Sample2

```
//MAINJOB1  JOB  ,,RESTART=STEP3,
//      REGION=4M
```

Sample3

```
//TECJOB41  JOB  ,'SMITH',
//     TIME=1440,
//     MSGLEVEL=(2,0)
```

- It is a good practice to code individual parameters on separate lines. If one of the parameters needs to be cancelled, all you need to do is comment out the line containing the parameter (by putting an '*' in column 3).

UNIT 3

The EXEC Statement.

The EXEC Statement

- Objectives.
- Purpose of the EXEC statement.
- Syntax.
- Parameters.

    - PGM
    - PROC
    - REGION
    - TIME
    - PARM

Objectives

- Understand the need for the EXEC statement.
- Learn the most important parameters for the EXEC statement.
- . Code different EXEC statement depending on the requirement.

Purpose of the EXEC statement

The EXEC statement is needed for the following reasons

To specify which programs need to be executed.

To specify which procedures (covered in later units) need to be executed.

To specify the system required parameters for each Step.

The main purpose of a Job is to execute programs. Therefore, any Job will have at least one Step (EXEC statement).

When speaking of the EXEC statements, Step and Exec are very frequently used interchangeably.

All parameters required by a Program are coded along with that step. Parameters may also be coded to override either the default parameter values and/or the parameters given in the JOB statement.

Syntax

The EXEC statement has one positional parameter and other keyword parameters.

Example:

```
//JONNAME  JOB  NOTIFY='userid'
//STEPNAME EXEC PGM=Program-name
                    or
 //JONNAME  JOB  NOTIFY='userid'
//STEPNAME EXEC PROC=Proc-name
                    or
 //JOBNAME  JOB  NOTIFY='Userid'
//STEPNAME EXEC Procedure-name
```

The program or procedure parameter must be the first parameter in the EXEC statement. Hence, it is 'Positional' in nature.

PGM= or PROC= is a positional parameter even though it is coded in keyword format.

Parameter – PGM

Type: Positional parameter

Purpose: To name the load module program which is to be executed in the EXEC statement

Syntax:

```
//JOBNAME   JOB  ACCTINFO,'PROGRAMMER NAME',CLASS=A,.
//STEP1     EXEC  PGM=Program-Name,..
```

Example:

```
//ABC     JOB   ACCT123,'PROGRAMMER NAME',CLASS=A,
//STEP1   EXEC  PGM=SAMPLE,.
```

The PGM parameter names the load-module you wish to execute in a particular Step. The name of the load-module is a character-string ranging from 1 to 8 characters.

Parameter – PROC

Type: Positional parameter

Purpose: To name the Procedure which is to be executed in the EXEC statement

Syntax:

```
//JOBNAME   JOB  ACCTINFO,'PROGRAMMER NAME',CLASS=A,.
//STEP1     EXEC  PROC=Proc-Name,..
```

Instead of coding PROC= procedure-name, one can code the procedure-name directly, without the 'PROC=' syntax, since this parameter is positional in nature.

Example:

```
//STEP1  EXEC PROC=USERPROC,...
```

Can also be coded as:

```
//STEP1  EXEC USERPROC,...
```

Parameter – REGION

Type: Keyword parameter

Purpose: To limit the maximum amount of memory that the Step can utilize.

Syntax:

//JOBNAME JOB ACCTINFO,'PROGRAMMER NAME',CLASS=A,.

//STEPNAME EXEC PGM=Prog-name,REGION=nnnnM,..

REGION=nnnnnnnK or REGION=nnnnM

n: Numeric value

Valid ranges:

0 thru 2047000 in case of K

0 thru 2047 in case of M

K: Kilobytes M: Megabytes

Examples:

```
//ABC     JOB  ACCT123,'PROGRAMMER NAME',CLASS=A,..
//STP1    EXEC PGM=MYPGM1,REGION=1024K
```

```
//ABC     JOB  ACCT123,'PROGRAMMER NAME',CLASS=A,..
//STEP1   EXEC PGM=MYPGM2,REGION=10M
```

As with the JOB statement, the REGION parameter can also be coded for the EXEC statement. If it is coded in both the statements, the value specified in the EXEC statement overrides that of the JOB statement. However, the value in the EXEC statement cannot be more than that of the JOB statement.

Parameter – TIME

Type: Keyword parameter

Purpose: The TIME parameter can be used to specify the maximum length of CPU time that a job or job step is to use the processor.

Syntax:

```
//JOBNAME   JOB   ACCTINFO,'PROGRAMMER NAME',CLASS=A,..
//STEPNAME EXEC PGM=PGMNAME,TIME=(mm,ss),.
```

TIME = (minutes, seconds)

Example:

```
//JOBNAME   JOB   TIME=(1,30)
//STP1      EXEC PGM=ABC, TIME=1
```

As with the JOB statement, the TIME parameter can also be coded with the EXEC statement.

If it is coded in both the statements, the value specified in the EXEC parameter overrides

that of the JOB statement. But, the value specified in the EXEC statement cannot be more than that of the JOB statement.

Parameters – PARM

Type: Keyword parameter

Purpose: To pass data to the program that is being executed in the step.

Syntax:

```
//JOBNAME  JOB ACCTINFO,'PROGRAMMER NAME'
//STEPNAME EXEC PGM=PGMNAME,PARM='Parm'
```

Example:

```
//JOBNAME  JOB  ,,NOTIFY='JSMITH'
//STEP1    EXEC PGM=SAMPLE,PARM='ABCDEFGH'
```

The maximum length of the value specified to be passed to the program should be 100 bytes. But, the system attaches a 2-byte binary value at the beginning of the data-block passed. This filler contains the length of the data passed. Hence, the program should compensate for the additional 2 bytes at the beginning of the data-block into which it receives the value and refer the actual data beginning only from the 3rd byte onwards.

Parameter – PARM (Continued)

Relationship of PARM to Cobol Program

```
//JOB1     JOB  ACA123),'ANDREW', CLASS=A,MSGLASS=A
//JOBLIB   DD   DSN=MAIN006.X100.LOADLIB,DISP=SHR
//STEP1    EXEC PGM=Sample,PARM='PRINT'
//DDIN     DD   DSN=MAIN086.INFILE,DISP=SHR
//DDOUT    DD   SYSOUT=*
```

Take a look at linkage Section of a COBOL source code.

```
LINKAGE SECTION.
01   PARM-FIELD.
       05   PARM-LENGTH      PIC     S9(04)  COMP.
       05   PARM-INDICATOR   PIC     X(05).
PROCEDURE DIVISION USING PARM-FIELD
A000-CHECK-PARM.
IF PARM-INDICATOR  = 'PRINT'
      NEXT SENTENCE
ELSE
     PERFORM 0100-CLOSE-FILES
```

Parm Indicator is equal to the string ‘PRINT’.

Parm-length is the length of the parm field.

Examples of EXEC statements

Following are a few sample EXEC statements in a multi-step JOB

```
//JOBNAME JOB  NOTIFY='userid'
//STEP1   EXEC PGM=MYPROG,REGION=4M,
//             TIME=NOLIMIT
//STEP2   EXEC PROC=MYPROC
//STEP3   EXEC PGM=MYPROG,PARM='E001BROWN'
```

UNIT 4

THE DD STATEMENT

The DD statement

Objectives.

Purpose of the DD statement.

Syntax.

Parameters.

a. DSN

b. DISP

c. SPACE

d. VOL

e. UNIT

f. DCB

g. SYSOUT

Temporary datasets

Refer backs

Special DD names

Dataset concatenation

Objectives

- Understand the purpose of the DD statement
- Learn the most important parameters of the DD statement
- Be able to create, use and manipulate datasets
- Code different DD statements depending on the requirement

Purpose of the DD statement

The DD statement is required for the following reasons

1. To define the input and output datasets that are to be used by a particular Step
2. To create new datasets if required by a Step
3. To specify all the dataset attributes required for creating new datasets
4. To manipulate and specify attributes for existing datasets so that they meet the requirements of the executing Step
5. To provide input data other than dataset input to programs that are executing in the Step
6. To direct the output created onto an output device or onto the spool

Most DD statements appear after at least one EXEC statement. But, some special DD statements, as we shall see, appear before any EXEC statement in the Job.

All DD statements related to a particular EXEC statement should be coded before coding the next EXEC statement. The next set of DD statements can now be coded for the new EXEC statement

Syntax

The DD statement accepts positional and keyword parameters.

```
//DDNAME DD DSN=X.Y.Z,DISP=NEW...
```

Example:

```
//DDIN   DD  DSN=MAINUSR.COBOL.INPUT,DISP=OLD
//DDOUT  DD  DUMMY
```

There are many keyword parameters that can be coded along with a DD statement, as we shall see. However, there is only one positional parameter - 'DUMMY'. The explanation for this parameter is discussed later in the unit.

Usually 'DUMMY' is the only parameter coded on a DD statement, but keyword parameters can follow. However if the keyword parameters are coded as the first parameters then there is no need to code a comma to signify the absence of a positional parameter.

Parameter – DSN (Data Set Name)

Type: Keyword parameter

Purpose: To specify the name of the dataset

Syntax:

```
//DDNAME    DD   DSN=MAINUSR.X.Y
```

Examples:

```
//DDIN    DD     DSN=MAINUSR.COBOL.FILE
//DDOUT   DD     DSN=MAINUSR.DB2.FILES(DCLGEN)
```

The DSN parameter is used to specify the name of the dataset, which is used by the Step. The Step will have a DD statement for each dataset used by the Program. The dataset may be used for input or output purposes as required by the program.

Parameter – DISP (Disposition)

Type: Keyword parameter that accepts positional parameters

Purpose: To specify

- The status of the dataset at the beginning of the step.
- What action to be performed on the dataset if the step completes normally.
- What action to be performed on the dataset if the step abends.

Example:

```
//JOBNAME    JOB   ,,CLASS=A
//STEP1      EXEC  PGM=SAMPLE1
//DD1        DD        DSN=MAINUSR.COBOL.INPUT,DISP=OLD
//DD2        DD        DSN=MAINUSR.COBOL.IN,DISP=(SHR,UNCATLG,DELETE)
//DD3        DD        DSN=MAINUSR.COBOL.OUT,DISP=(NEW,CATLG,DELETE),
//                    VOL=SER=SYSDA,SPACE=(TRK,(1,1))
//STEP2      EXEC      PGM=SAMPLE2
//DDIN       DD        DSN=MAINUSR.VSAM.IN,DISP=SHR
//DDOUT      DD        DSN=MAINUSR.VSAM.OUT,DISP=MOD,
//                    VOL=SER=SYSDA,SPACE=(TRK,(1,1))
```

The above example is a two Step JOB. In the first Step, the program SAMPLE1 requires the use of three datasets specified by the DD names DD1, DD2, and DD3.
In the first DD statement (DD1) the DSN parameter names the physical dataset and DISP parameter states that the dataset already exists and that we want exclusive use of this dataset (DISP=OLD).

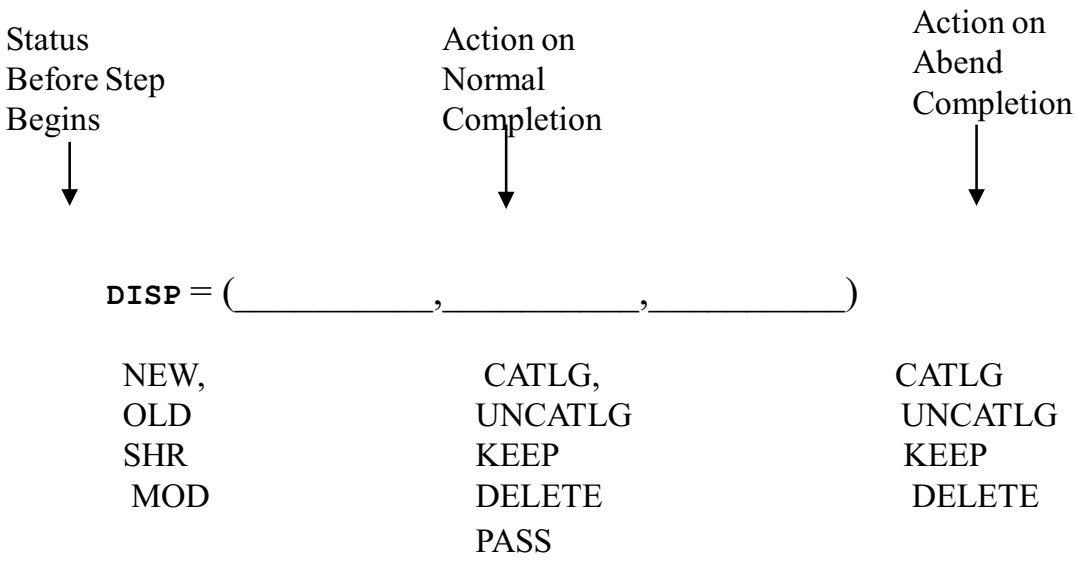
In the second DD statement (DD2) the DISP parameter states to use the existing dataset for shared use and that the dataset is to be uncataloged after completion of the program. However if the program terminates abnormally we want to delete this dataset.

In the third DD statement (DD3) the DISP parameter states that a new dataset is to be created. After completion of the program we want to catalog the dataset. However, if the program terminates abnormally than the dataset should be deleted.

The Disposition parameter is used to specify if the dataset already exists or if it is going to be a new dataset. It also determines what is to be done to the dataset once the program terminates normally, and if it terminates abnormally for some reason.

The DISP parameter refers to the entire dataset and not to members of a PDS.
The default DISP parameter is **`DISP=(NEW,DELETE,DELETE)`**

Parameter – DISP (Continued)



NEW – Dataset does not exist at the beginning of the step and is to be created.
This parameter requires coding of some additional parameters like Space, Volume and Data Control Block (DCB). Code DISP=NEW when creating a new data set. DISP=NEW allows only one user to access the data set at a time.

OLD – Dataset exists at the beginning of the step and should be opened for exclusive use by the step, i.e. no other process can access the dataset until this step finishes execution.
Also, if records are being written to the dataset, the new records will overwrite existing records..

Code DISP=OLD to reference an existing data set. DISP=OLD allows only one user to access the dataset at a time. The system places other users in a wait state. If DISP=OLD is coded in output, the system starts at the beginning of the data set and writes over existing data. The system writes an end-of-file (EOF) after writing the last record.

SHR – Dataset exists at the beginning of the step and should be opened for this step in shared mode, i.e. any other process can also access the same dataset while the current step is executing but, only in shared mode. This means that no other process can code DISP=OLD for this dataset.

Code DISP=SHR to reference an existing data set and allow other users to access the data set at the same time.

MOD – If dataset exists, open it in exclusive mode (DISP=OLD) and, if records are being written to it, add the records at the end of the dataset. If the dataset does not exist then the status defaults to NEW.

Code DISP=MOD to indicate records are to be appended to the logical end of an existing data set. If DISP=MOD is coded and the data set does not exist, the disposition is treated as DISP=NEW.

Parameter - SPACE

Type: Keyword parameter that accepts positional parameters

Purpose: To specify the Space related requirements for new datasets

Syntax: SPACE=(unit,(p,s,d),RLSE)

Where:

Unit - Specifies the unit type that space will be allocated.

Values are:

TRK – requests allocation of storage for the new dataset in terms of tracks

CYL – requests allocation of storage for the new dataset in terms of cylinders

p -Numerical value that indicates the primary quantity of space to be allocated in terms of the unit specified

s -Numerical value that indicates the secondary quantity to be allocated once the primary quantity is exhausted

d -Numerical value that specifies the number of directory blocks for a PDS (see below)

Directory Blocks Specifies the Directory Blocks the dataset is to have. Any value greater than zero creates a Partitioned Dataset. Coding a zero in this parameter or skipping it altogether creates a Physical Sequential dataset.

RLSE Releases any unused secondary space from the dataset for use by other processes once the step finishes execution.

Example:

```
//DDOUT   DD      DSN=MAINUSR.COBOL.OUT,DISP=(NEW,CATLG),
//          SPACE=(TRK,(2,3,1)RLSE),VOL=SER=SYSDA
```

In the above example, MAINUSR.COBOL.OUT is a new partitioned dataset (PDS) to be created since the directory block is a value of one, and the space required is in terms of tracks. The primary quantity is two tracks and secondary is three tracks.

Parameter - VOLUME

Type: Keyword parameter that accepts keyword and positional parameters

Purpose: To specify the Volume serial number on which the dataset exists or needs to be created.

Syntax:

VOL=SER=volume-serial-number

Volume serial number is the installation-dependent serial number of the volume on which the dataset resides or needs to be created.

or

VOL=REF=cataloged-dataset-name

When you want to specify the name of a cataloged dataset instead of specifying the volume-serial. Here, the dataset will be referenced from or created (depending on the disposition) on the same volume on which the cataloged dataset, mentioned in the parameter resides.

Example:

```
//DDOUT   DD   DSN=MAINUSR.COBOL.OUT,DISP=(NEW,KEEP),
//             SPACE=(TRK,(2,3,1)RLSE),VOL=SER=LP2WK1
```

Volume serial number is an important parameter to be mentioned for datasets that are given the Disposition of KEEP and for those datasets that are uncataloged.

For new datasets that are going to be cataloged there is no need to mention a volume serial number.

When you retrieve a cataloged existing dataset there is no need to mention the volume serial number.

The VOL parameter is a must while creating a dataset or accessing an uncataloged dataset.

Parameter - UNIT

Type: Keyword parameter which accepts positional parameters.

Purpose: To specify what type of device the dataset resides on or should be created.

Syntax:

```
UNIT=Generic-unit-name
                or
UNIT=Unit-model-number
                or
UNIT=/Machine-address
```

Generic-unit-name:

SYSDA	any Direct Access Device
TAPE	any Sequential Access Device
SYSSQ	any Direct or Sequential Access device

Unit-model-number:

9345	
3380	DASD
3390	
3490	Tape

Machine-address: The installation-specific address of the device

Example:

```
//DDOUT   DD     DSN=MAINUSR.COBOL.OUT,DISP=(NEW,CATLG),
//           SPACE=(TRK,(2,3,1)RLSE),VOL=SER=LP2WK1,UNIT=SYSDA
```

Parameter – DCB (Data Control Block)

Type: Keyword parameter that accepts keyword parameters

Purpose: DCB is used to tell the system about the characteristics of the dataset being referenced, such as the structure of the record and the Block size.

`Syntax: DCB=(LRECL=nnnn,BLKSIZE=nnnn,RECFM=record-format)`

n – a numeric value

LRECL – The Logical Record Length - Specifies the maximum length of a record in the dataset. Values can range from 1 to 32760 bytes.

BLKSIZE – The Block Size - Specifies the maximum length of a block in the dataset. Values can range from 18 to 32760 bytes.

RECFM – The Record Format - Specifies the format of the records in the dataset i.e. fixed, variable, blocked, undefined etc.

record-format -	F	Fixed length
	FB	Fixed length Blocked
	V	Variable length
	VB	Variable length Blocked
	U	Undefined

Example:

```
//DDOUT   DD      DSN=MAINUSR.COBOL.OUT,DISP=(NEW,CATLG),
//                SPACE=(TRK,(2,3,1)RLSE),VOL=SER=LP2WK1,
//                    DCB=(LRECL=80,RECFM=FB,BLKSIZE=8000)
```

For the RECFM parameter, some additional characters can be added at the end of the RECFM value. They are,

- A — If the records contain a carriage-return control character in the first byte.
- M — I f the records contain machine-specific device control characters other than the carriage-return control character in the first byte.
- S — If the records size may exceed the block size.
- T — If the records may be bigger than the track capacity of the device.

Also you can code each of these sub parameters separately without including them in the DCB parameter. However, if the DCB parameter is coded then all these sub parameters, if coded, must appear within the DCB parameter.

Parameter - SYSOUT

Type: Positional keyword parameter

Purpose: To send program output to the SYSOUT print spool

Syntax:

```
SYSOUT=class value
        or
SYSOUT = *   (refers to the MSGCLASS value on the JOB card).
```

Examples:

```
//DDOUT    DD    SYSOUT=A
//DDOUT1   DD    SYSOUT=*
```

If coded, the SYSOUT parameter should appear immediately after the DD statement.

Use the SYSOUT parameter if you want to direct the output meant for any dataset to the print spool maintained by JES.

Example:

```
//SYSPRINT DD SYSOUT=*
```

Directs the output meant for dataset SYSPRINT to the systems output spool.

Temporary Datasets

Temporary datasets are used as scratchpads for storing data only for the run of that step. After the end of the step, the temporary dataset is deleted unless Passed to the next step.

The default disposition for temporary datasets is (NEW,DELETE,DELETE)

A temporary data set is one that is created and deleted in the same job. There are three ways to request the creation of a temporary data set:

1.Omit the DSN

```
//A    DD UNIT=SYSDA,SPACE=(CYL,1)
```

In this case the system will create temporary data set having a 44-character name. The system builds date, time, and other information into this name to ensure the name is unique.

2 Code a name beginning with && such as DSN=&&WORK

```
//B    DD DSN=&&WORK,UNIT=SYSDA,SPACE=(CYL,1)
```

The system will create a temporary data set having a 44-character name. The system builds date, time, and other information into the name to ensure its uniqueness.

3 Code a name beginning with & such as DSN=&TEMP

```
//C    DD DSN=&TEMP,UNIT=SYSDA,SPACE=(CYL,1)
```

The system will create a temporary data set with a 44-character name. The system creates a unique name containing date, time and other information. Names prefixed by && are preferable to those prefixed by &. In a name of the form, DSN=&TEMP, the &TEMP can mean a temporary data set or a symbolic parameter (to be covered later).

Temporary Datasets (Continued)

To avoid ambiguity use the form, DSN=&&TEMP.

If you use two temporary data sets of the forms, DSN=&&WORK and DSN=&WORK, in the same job stream, the system will interpret these to be the same data set.

You can create temporary datasets by omitting the DSN parameter .

To create temporary datasets:

DSN=&Tempname

or

DSN=&&Tempname

Where Tempname is any name of 8 characters or less.

Example:

DSN=&&TEMPFILE

You can reference the dataset by coding &&TEMPFILE but, internally the system will generate a long dataset name of 5 qualifiers which include User ID, Timestamp etc. to maintain uniqueness.

To retain the dataset even after the current step ends, the temporary dataset has to be passed to the next step by coding PASS in the second sub-option of the DISP parameter.

It is not advisable to code the name preceded by a single ampersand (&) as the name might be substituted by the value of a symbolic-parameter (covered in later units) of the same name.

Refer back

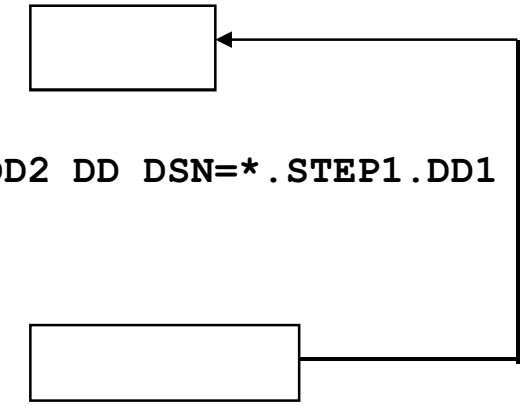
You can use refer backs if you don't want to explicitly code the value for parameters.

The presence of the '*' indicates that a backward reference is being used.

```
  DSN = *.DDNAME
        *.STEPNAME.DDNAME
//STEP1 EXEC ..............

//DD1 DD DSN=A.B.C, ..........

//STEP2 EXEC ..............
```



```
//DD2 DD DSN=*.STEP1.DD1
//
```

Syntax:

Parameter=*.STEPNAME.DDNAME

Refer back indicator

Many parameters in JCL statements can use a backward reference to fill in information. A backward reference (or refer back) is a reference to an earlier statement in the job or in a cataloged or instream procedure called by a job step.

A backward reference has the form:

***.NAME OR *.DDNAME**

***.STEPNAME.NAME OR *.STEPNAME.DDNAME**

***.STEPNAME.PROCSTEPNAME.NAME OR *.STEPNAME.PROCSTEPNAME.DDNAME**

Special DD names

You can use any name for your DD statements but there are certain names that have a special meaning to the system or to the program(s) that are being executed.

Below are some of the most common DD names that have special meaning on most installations.

STEPLIB	Default search library PDS for the load-module being executed in the step
JOBLIB	Default search library PDS for the load-modules of all the steps in the Job
SYSPRINT	Default output dataset for printable output
SYSOUT	Default output dataset for system messages
SYSIN	Default instream input dataset for most programs

Note that if a DD name has a special meaning to one program, it may mean nothing at all to another program

STEPLIB

The //STEPLIB DD statement alters the normal program search strategy to locate the program being executed.When included in a step, the library named on the //STEPLIB DD statement is searched before the system searches the default library, SYS1.LINKLIB

MY.LIB1SYS1.LINKLIB//LIBJOB JOB 8877,CLASS=T

```
//ONE          EXEC          PGM=FIRST
//STEPLIB      DD            DSN=MY.LIB1,DISP=SHR
        ----------
```

MY.LIB1
SYS1.LINKLIB

SYS1.LINKLIB

```
        ----------
//TWO   EXEC          PGM=SECOND
        ----------
```

SYS1.LINKLIB

MY.LIB2SYS1.LINKLIB

```
//THREE EXEC          PGM=THIRD
//STEPLIB      DD            DSN=MY.LIB2,DISP=SHR
```

MY.LIB2
SYS1.LINKLIB

You are permitted to concatenate libraries to a STEPLIB DD statement. If you do so the libraries are searched in their concatenation order.

STEPLIB overrides JOBLIB in the step, which has the STEPLIB statement.

JOBLIB

```
//ZEBRA JOB      66,CLASS=T
//JOBLIB         DD       DSN=MY.LIB,DISP=SHR
//ONE            EXEC     PGM=FIRST

//TWO            EXEC     PGM=SECOND
        --------
        --------

//THREE EXEC     PGM=THIRD
```

Code the //JOBLIB DD statement to identity a private library/libraries the system is to search for each program in the job.

The //JOBLIB DD statement must be coded before the first EXEC statement

You are permitted to concatenate libraries to a JOBLIB DD statement. If you do so the libraries are searched in their concatenation order.

Dataset Concatenation

```
//INDATA      DD DSN=USERID.INPUT.FILE1,DISP=SHR
//              DD DSN=USERID.INPUT.FILE2,DISP=SHR
//              DD DSN=USERID.INPUT.FILE3,DISP=SHR
//OUTDATA     DD DSN=USERID.OUTPUT.DATA1,DISP=OLD
//              DD DSN=USERID.OUTPUT.DATA2,DISP=OLD
//              DD DSN=USERID.OUTPUT.DATA3,
//                 DISP=(NEW,CATLG,DELETE),
//              SPACE=(TRK,(1,1),RLSE),LRECL=80,
//              BLKSIZE=6160,RECFM=FB,
//              VOL=SER=LP1WK1,UNIT=SYSDA
```

Concatenation is the method of combining 2 or more datasets and making them behave as a single file from the point of view of the program.

In case of input datasets records will be read in from files in the same order as specified in the concatenation i.e. start reading from the first file, once there are no more records in the first file, continue with the second file and so on until the last file is read. Only then will the system encounter the end-of-file condition.

In case of output datasets, writing will start from the first dataset. Once all the space in the first dataset is exhausted, the next dataset in the concatenation order will be written to and, so on.

UNIT 5

UTILITIES

Utilities

- Objectives
- OS/390 Utilities
- Utility Classification
- Utility Control Statement Syntax
- IEBGENER Utility
- IEBPTPCH Utility
- IEHLIST Utility
- IEBCOPY Utility
- IEHPROGM Utility
- IEBUPDTE Utility
- IDCAMS Utility
- SORT Utility

Objectives

- Understand the concept of a Utility.
- Know how to code different Utilities.
- Understand the functions of different Utilities.

OS/390 Utilities

- Utilities are IBM-supplied programs that are intended to perform certain routine and frequently occurring tasks in the OS/390 environment.
- Utilities are used in DASD, tape drives, print and punch operations.
- Utilities are used to allocate, update, delete, catalog and uncatalog data sets, and also to list the contents of VTOC (Volume Table of Contents).

Utility Classification

Utilities are broadly classified into two different categories:

- Data set Utilities (prefixed with IEB)
- System Utilities (prefixed with IEH)

Data set utilities are used to copy, print, update, reorganize, and compare data at the dataset and/or record level.

System utilities are used to list VTOC information, copy, delete, catalog and uncatalog datasets, to write tape labels and to add or delete dataset passwords.

Utility Control Statement (SYSIN)

During execution, utilities open and read the dataset described in the DD statement SYSIN. The Control statement parameters are given in this DD statement.
If no control statements are required then you have to use the DUMMY parameter of the DD statement, since the program expects some kind of input (in the form of control statement).

The format in which the SYSIN DD statement will appear is:

```
//SYSIN    DD    *
----Control statements-----
/*
```

Another way to code the SYSIN is to code the control statements in a member of a PDS.
Then the SYSIN DD would appear as follows:

```
//SYSIN   DD   DSN=MAINUSR.CTL.LIB(MEM),DISP=SHR
```

Utility Control Statement Syntax

The general format of a Utility Control Statement is :

[label] operation operands comments

Where:

Field

label is optional.

operation is required.

operands is required.

comments is optional.

Utility control statements must be coded in columns 1 thru 71.

If a statement exceeds column 71, then

1. Break the statement in column 71 or after a comma.
2. Code a non-blank character in column 72.
3. Continue the statement in column 16 of the following line.

The IEBGENER Utility

IEBGENER is a dataset utility used to create, copy, or print sequential data sets

Example:

```
//JOBNAME   JOB  ACCT,'APARNASANDEEP'
//STEP1     EXEC PGM=IEBGENER
//SYSPRINT  DD   SYSOUT=*
//SYSUT1    DD   DSN=MAINUSR.SEQ.INPUT,DISP=SHR
//SYSUT2    DD   DSN=ABC.SEQ.OUTPUT,
//                   DISP=(,CATLG,DELETE),
//               SPACE=(TRK,(1,1)),VOL=SER=LP1WK2,
//               RECFM=FB,LRECL=80,UNIT=SYSDA
//SYSIN     DD   DUMMY
```

In the above example, the dataset MAINUSR.SEQ.INPUT is being copied to a new file called ABC.SEQ.OUTPUT.

- The EXEC statement specifies the program to be executed (IEBGENER).
- The SYSPRINT DD statement defines the message dataset.
- The SYSUT1 DD statement defines the input dataset.
- The SYSUT2 DD statement defines the output dataset (can have multiple SYSUTn, where n should be 1,2,3…).
- The SYSIN DD statement defines the control dataset. This is where IEBGENER looks for any utility control statements. When DUMMY is specified, there are no control statements being used.

The IEBGENER Utility

This example shows how IEBGENER can be used for copying or printing selected portions of datasets. In this case, selected records from MAINUSR.ABC.REC will be printed.

Example:

```
 //JOBNAME  JOB  ACCT,'APARNASANDEEP'
//STEP1    EXEC PGM=IEBGENER
//SYSPRINT DD   SYSOUT=*
//SYSUT1   DD   DSN=MAINUSR.ABC.REC
//SYSUT2   DD   SYSOUT=*
//SYSIN    DD   *
       GENERATE  MAXFLDS=2
       RECORD FIELD=(10,20,,1),FIELD=(10,1,,15)
 /*
//
```

This step executes IEBGENER to create the SYSUT2 output data and route it to a print class.

- 'SYSIN DD *' indicates that instream records or control statements follow this statement.
- The GENERATE statement specifies that editing of the input data is to be performed.
- MAXFLDS=2 indicates that no more than 2 fields will be described on the subsequent RECORD statement.
- The RECORD statement describes the input and output fields thru the FIELD parameter.
- Each FIELD parameter specifies the field's length, its location in the input record, what type of conversion is to be done on the field, and where it should be placed in the output record.

The format is:

FIELD=(LENGTH OF FIELD,POSITION IN INPUT, CONVERSION, POSITION IN OUTPUT)

The FIELD parameters in the above example state to:

- move the 10 bytes starting in position 20 of the input record to the 10 bytes starting in 1 in the output record;
- and to move the 10 bytes starting in position 1 of the input record to the 10 bytes starting in 15 in the output record

The IEBPTPCH Utility

IEBPTPCH is used to print or punch all or selected portions of datasets. Editing can be done on the data to format the output.

Example:

```
//STEP1     EXEC PGM=IEBPTPCH
//SYSPRINT  DD SYSOUT=*
//SYSUT1    DD   DSN=MAINUSR.ABC.REC,DISP=SHR
//SYSUT2    DD   SYSOUT=*
//SYSIN     DD   *
   PRINT  TYPROG=PS,MAXFLDS=2
   TITLE ITEM=('EMPLOYEES PROFILE',27)
   TITLEITEM=('NAME       ADDRESS',15)
   RECORD FIELD=(8,2,,10),FIELD=(5,10,,20)
```

- The PRINT control statement specifies that the dataset has an organization of physical sequential and that a maximum of two fields will be printed on output.
- The output titles are specified on the TITLE control statements.
- 'EMPLOYEES PROFILE' will be placed position 27 of the output file
- 'NAME' and 'ADDRESS' will be placed on the next title line, beginning in position 15
- The RECORD statement describes the input and output fields thru the FIELD parameter.
- Each FIELD parameter specifies the field's length, its location in the input record, what type of conversion is to be done on the field, and where it should be placed in the output record.

  The format is:

  FIELD=(LENGTH OF FIELD,POSITION IN INPUT, CONVERSION, POSITION IN OUTPUT)
- The FIELD parameters in the above example state to:

  move the 8 bytes starting in position 2 of the input record to the 8 bytes starting in 10 in the output record;

  and to move the 5 bytes starting in position 10 of the input record to the 5 bytes starting in 20 in the output record

The IEHLIST Utility

The IEHLIST utility is used to:

list entries in a DASD VTOC (Volume Table of Contents)

list entries in a PDS Directory.

list entries in a system catalog

Example 1:

```
//STEP1     EXEC  PGM=IEHLIST
//SYSPRINT  DD        SYSOUT=*
//DD1       DD    DISP=OLD,UNIT=SYSDA,VOL=SER=ABC
//DD2       DD    DISP=OLD,UNIT=SYSDA,VOL=SER=DEF
//SYSIN     DD    *
    LISTVTOC  FORMAT,VOL=SYSDA=ABC
    LISTVTOC  FORMAT,VOL=SYSDA=DEF                                    X
        DSNAME=(MTPL.FILE1,MTPL.FILE2)
/*
//
```

The above example uses IEHLIST to print two VTOC listings:
The IEHLIST looks for utility control statements coded below the
SYSIN DD statements:
The first LISTVTOC control statement requests an formatted
(FORMAT) VTOC listing for pack ABC. This includes DSCB and space
allocation information. If FORMAT is omitted, an abbreviated version is listed.
The second LISTVTOC control statement requests a formatted VTOC listing
for two datasets: MTPL.FILE1 and MTPL.FILE2.

Example 2:

```
//STEP1     EXEC PGM=IEHLIST
//SYSPRINT  DD     SYSOUT=*
//DD1       DD   DISP=OLD,UNIT=SYSDA,VOL=SER=ABC
//SYSIN     DD   *
    LISTPDS DSNAME=MTPL.FILE,VOL=SYSDA=ABC
/*
//
```

The above example uses IEHLIST to list entries in a PDS directory.
The LISTPDS control statement requests a listing of the directory for the PDS, MTPL.FILE.

NOTE: DSNAME cannot be abbreviated as DSN on a control statement

The IEBCOPY Utility

The IEBCOPY is used to copy members of partitioned datasets.

The COPY statement identifies the input and output files by referring to their DDNAMEs in the JCL. The format is:

```
COPY    OUTDD=output-ddname,INDD=input-ddname
```

The SELECT statement identifies the members of the PDS to be copied. The format is:

SELECT MEMBER=NAME (to specify a single member)

-or-

SELECT MEMBER=(NAME,NAME,NAME) (to specify multiple members)

-or-

SELECT MEMBER=(old,new,R) (to copy a member and change its name)

The EXCLUDE statement indicates that all members should be copied except those specified in the EXCLUDE statement. The format is:

EXCLUDE MEMBER=NAME (to specify a single member)

-or-

EXCLUDE MEMBER=(NAME,NAME,NAME) (to specify multiple members)

Copying an Entire PDS

```
//MODAL2      JOB  '0.2AMIP',…….
//STEP1       EXEC PGM=IEBCOPY
//SYSPRINT    DD         SYSOUT=*
//IN          DD   DSN=MTPL.FILE1,DISP=SHR
//OUT         DD   DSN=MTPL.FILE2,DISP=SHR
//SYSIN       DD   *
     COPY OUTDD=OUT,INDD=IN
/*
```

The above example copies all of the members from the PDS, '**MTPL.FILE1**' to an existing PDS, '**MTPL.FILE2**'.

The IN and OUT DD statements define data sets to be used by IEBCOPY.

The COPY control statement specifies the input and output DDNAMES.
No SELECT or EXCLUDE statements were used.

Copying Specific Members

```
//MODAL2      JOB  '0.2AMIP',.....
//STEP1       EXEC PGM=IEBCOPY
//SYSPRINT    DD         SYSOUT=*
//IN          DD   DSN=MTPL.FILE1,DISP=SHR
//OUT         DD   DSN=MTPL.FILE2,DISP=SHR
//SYSIN       DD   *
   COPY    OUTDD=OUT,INDD=IN
   SELECT MEMBER=ALLOCATE
   SELECT MEMBER=((PROD,TEST,R))
 /*
```

The above example copies the member called "ALLOCATE" from the PDS, '**MTPL.FILE1**' to an existing PDS,'**MTPL.FILE2**'.

The SELECT control statement specifies the member ALLOCATE is to be copied from the input to the output dataset.

Copying Multiple Specific Members

```
//MODAL2      JOB   '0.2AMIP',.....
//STEP1       EXEC  PGM=IEBCOPY
//SYSPRINT    DD          SYSOUT=*
//IN          DD    DSN=MTPL.FILE1,DISP=SHR
//OUT         DD    DSN=MTPL.FILE2,DISP=SHR
//SYSIN       DD    *
      COPY    OUTDD=OUT,INDD=IN
      SELECT MEMBER=(FILE1,FILE2,FILE3)
/*
```

The above example copies selected members from the PDS, '**MTPL.FILE1**' to an existing PDS, '**MTPL.FILE2**'.

The SELECT control statement specifies the members: FILE1, FILE2 and FILE3 to be copied.

Copying and Renaming Specific Members

```
//MODAL2      JOB  '0.2AMIP',.....
//STEP1       EXEC PGM=IEBCOPY
//SYSPRINT    DD       SYSOUT=*
//IN          DD   DSN=MTPL.FILE1,DISP=SHR
//OUT         DD   DSN=MTPL.FILE2,DISP=SHR
//SYSIN       DD   *
   COPY    OUTDD=OUT,INDD=IN
   SELECT MEMBER=(JOBA,(PROD,TEST,R))
/*
```

The above example copies the member called "PROD" from the PDS, '**MTPL.FILE1**' to an existing PDS, '**MTPL.FILE2**'.

The SELECT control statement specifies:

- Copy the member JOBA
- the member PROD is to be copied in the following manner
  - rename PROD to TEST,
  - copy the renamed member TEST to the output dataset,
  - if a member by that name exists in the output dataset replace it.

Copying Using EXCLUDE

```
 //MODAL2      JOB  '0.2AMIP',…….
//STEP1       EXEC PGM=IEBCOPY
//SYSPRINT    DD         SYSOUT=*
//IN          DD   DSN=MTPL.FILE1,DISP=SHR
//OUT         DD   DSN=MTPL.FILE2,DISP=SHR
//SYSIN       DD   *
   COPY    OUTDD=OUT,INDD=IN
   EXCLUDE MEMBER=ALLOCATE
/*
```

The above example copies all members '**MTPL.FILE1**' except the member 'ALLOCATE'

Compressing Data Sets

```
 //MODAL2      JOB  '0.2AMIP',…….
//STEP1       EXEC PGM=IEBCOPY
//SYSPRINT    DD        SYSOUT=*
//INPDS       DD   DSN=MTPL.FILE1,DISP=SHR
//SYSUT3      DD   UNIT=SYSDA,SPACE=(TRK,(1,1))
//SYSUT4      DD   UNIT=SYSDA,SPACE=(TRK,(1,1))
//SYSIN       DD   *
    COPY INDD=INPDS,OUTDD=INPDS
/*
```

The above example compresses the library '**MTPL.FILE1**'.

Notice that the same DD name is specified in both the INDD and OUTDD parameters.

The IEHPROGM Utility

The IEHPROGM is used to:

- Scratch (delete) a dataset
- Rename a member of a PDS
- Catalog or uncatalog datasets

Example 1:

```
//STEP1      EXEC PGM=IEHPROGM
//SYSPRINT   DD  SYSOUT=*
//NUM1       DD   UNIT=SYSDA,VOL=SER=ABC,DISP=OLD
//SYSIN      DD   *
     SCRATCH MEMBER=ALLOCATE,DSNAME=MTPL.FILE,                X
     VOL=SYSDA=ABC
     RENAME    MEMBER=XYZ,DSNAME=MTPL.FILE1,                  X
     VOL=SYSDA=ABC,NEWNAME=PQR
/*
```

The SCRATCH statement tells IEHPROGM is used to scratch the member ALLOCATE in the PDS, MTPL.FILE.

The RENAME control Statement tells IEHPROGM to rename member XYZ to PQR in the PDS, MTPL.FILE1.

Example 2:

```
//IEHPROGM2     JOB A123,'MAHESH',……..
//STEP1 EXEC  PGM=IEHPROGM
//SYSPRINT      DD    SYSOUT=A
//SYSUT1 DD      UNIT=3390,VOL=SER=LP2WK1
//SYSIN  DD      *
    UNCATLG DSNAME=ABC.FILE,VOL=SER=LP2WK1,UNIT=3390
/*
//
```

In the above example:

The UNCATLG statement tells IEHPROGM to uncatalog the dataset ABC.FILE

Use CATALOG to catalog a dataset

The IEBUPDTE Utility

The IEBUPDTE utility is used to create, update and modify sequential datasets and members of partitioned datasets.

Example

```
//STEP1       EXEC PGM=IEBUPDTE
//SYSPRINT    DD          SYSOUT=*
//SYSUT1      DD    DSN=MTPL.MYPDS,VOL=SER=ABC,UNIT=SYSDA,
       DISP=OLD
//SYSIN       DD    *
./              CHANGE NAME=MEM,UPDATE=INPLACE
     JAY     HARI   00000050
     MARY    CHRISTI      00000070
./            ENDUP
/*
```

When a PDS member is changed and then replaced or added to a PDS, it normally goes in the available space after the last member in the PDS. If several records in a member are replaced by an equal number of records, IEBUPDTE can update the member without changing its address in the PDS. This is called Update in Place.

In the above example, the PDS MTPL.MYPDS, contains a member named MEM, which contains two records with sequence numbers (00000050 and 00000070) in columns 73 thru 80. When the IEBUPDTE job is executed, those two records will be replaced by the two records in the SYSIN input stream. This is an Update in Place.

Before:

```
ELIZABETH   HENRY                00000030
THOMAS      JOHNSON              00000040
RICO        BROWN                00000050
TIMOTHY     SIMMS                00000060
MARY        BAKER                00000070
HARRIET     WILLIAMS             00000080
```

After:

```
ELIZABETH   HENRY                00000030
THOMAS      JOHNSON              00000040
JAY          HARI                00000050
TIMOTHY     SIMMS                00000060
MARY        CHRISTI              00000070
HARRIET     WILLIAMS             00000080
```

The IEFBR14 Utility

This utility is called a dummy utility, since its basic function is to do what the disposition parameter of the DD says.

To Uncatalog a Dataset

```
//UNCATLOG  JOB  A123,'SUSAN JOHN'
//STEP1     EXEC PGM=IEFBR14
//DD1       DD   DSN=MAINUSR.COBOL.FILE1,DISP=(OLD,DELETE)
```

To Delete The Dataset

```
//DELETE1   JOB    A123,' 'SUSAN JOHN'
//STEP1     EXEC   PGM=IEFBR14
//DD1       DD     DSN=MAINUSR.COBOL.FILE2,DISP=(OLD,DELETE)
```

To Catalog a Dataset

```
//CATALOG JOB A123,' 'SUSAN JOHN'
//STEP1   EXEC  PGM=IEFBR14
//DD1     DD   DSN=MAINUSR.COBOL.FILE3,DISP=(NEW,CATLG)
```

To Create A Dataset

```
//CREATE   JOB          A123, 'SUSAN JOHN'
//STEP3          EXEC   PGM=IEFBR14
//SYSIN    DD SYSOUT=*
//DD1      DD    DSN=MAINUSR.COBOL.FILE4,DISP=(NEW,KEEP),
//               UNIT=SYSDA,SPACE(TRK,(4,2)),
//               DCB=(LRECL=80,RECFM=FB,BLKSIZE=800),
//               VOL=SER=LP2WK1
//SYSIN DD DUMMY
```

The IDCAMS Utility

IDCAMS is a special utility to create, delete, copy and print the contents of VSAM and non-VSAM datasets.

Example:

```
//JOBNAME   JOB  (ACCT),'SUSAN JOHN'
//STEPNAME  EXEC PGM=IDCAMS
//SYSPRINT  DD   SYSOUT=A
//SYSIN   DD  *
                 Functional-Commands OR Control Statements
/*              (terminator)
//
```

Example:

```
//JOBNAME   JOB (ACCT),'SUSAN JOHN'
//STEP1     EXEC PGM=IDCAMS
//SYSPRINT  DD   SYSOUT=*
//SYSIN     DD   *
                 DELETE MTPL.SEQ.DATA
/*
//
```

In the above example when the step executes, IDCAMS deletes the non-VSAM dataset MTPL.SEQ.DATA.

Functions coded on IDCAMS Utility

The IDCAMS Utility can be executed for the following function on VSAM datasets.

- Define them
- Load records into them
- Print them

The IDCAMS Utility (Continued…)

Example:

```
//KSDLOAD1            JOB   (A123),'SUSAN JOHN',
//STEP1      EXEC PGM=IDCAMS
//SYSPRINT    DD    SYSOUT=*
//DDIN       DD    DSN=FILE1.TEST,DISP=SHR
//DDOUT      DD    DSN=VSAM1.KSDS.CLUSTER,DISP=OLD
//SYSIN      DD    *
         REPRO                        -
         INFILE(DDIN)      -
         OUTFILE(DDOUT)
//*
//
```

Alter Command to alter a PDS

```
//ALTER1         JOB     23,'SUSAN JOHN'
//STEP1  EXEC    PGM=IDCAMS
//SYSPRINT       DD      SYSOUT=A
//SYSIN  DD      *
         ALTER   -
         MAINUSR.COBOL.LOADLIB (PROGRAM1)                    -
         NEWNAME(MAINUSR.COBOL.LOADLIB (SAMPLE1))
/*
//
```

Here the MAINUSR.COBOL.LOADLIB (PROGRAM1) is changed to MAINUSR.COBOL.LOADLIB (SAMPLE1)

The SORT Utility

The SORT utility is used is sort the contents of a dataset depending on a key called the sort field. The sorted output is then written to another dataset
SORT is an alias or alternate name for ICEMAN. PGM=SORT or PGM=ICEMAN on the EXEC statement will invoke DFSORT program which is used for sorting an input dataset.
The SORT control statement is specified in the SYSIN DD statement.

Example:

```
//STEP1    EXEC PGM=SORT
//SYSOUT   DD  SYSOUT=*
//SORTIN   DD  DSN=MAINUSR.SEQ1.INPUT,DISP=OLD
//SORTOUT  DD  DSN=MAINUSR.SEQ2.OUTPUT,DISP=OLD
//SYSIN    DD  *
             SORT FIELDS=(21,2,CH,A)
/*
```

The above example will sort the records of the input dataset specified in the SORTIN DD statement based on the field specified in the control statement of the SYSIN DD.
The sorted dataset is copied to the output dataset specified in the SORTOUT DD statement.

The EXEC statement invokes the program.

- **The SORTIN DD statement defines the input data set.**
- **The SORTOUT DD statement defines the output data set.**
- **The SYSIN DD statement defines the control data set.**
- **The SORT control statement specifies the position, length, format and order of sort.**

The above example sorts the records in ascending (A) order, using the 2 bytes (2) of character data (CH) starting in location 21.

The SORT Utility (Continued…)

Example 2

```
//SORT1       JOB    (A123),'SUSAN JOHN',NOTIFY=&USERID
//STEP1       EXEC   PGM=SORT
//SYSOUT      DD     SYSOUT=A
//SYSPRINT    DD     SYSOUT=A
//SORTIN      DD     DSN=MAINUSR.ADDRESS.BOOK1,DISP=SHR
//SORTOUT     DD     DSN=MAIN006.ADDRESS.BOOK2,
//             DISP=(NEW,CATLG,DELETE),
//             UNIT=SYSDA,
//             SPACE=(CYCL,(2,1),RLSE),VOL=SER=LP2WK1,
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=800)
//SORTWK01     DD  UNIT=SYSDA,SPACE=(CYL,(20,10),RLSE)
//SYSIN        DD            *
       SORT   FIELDS=(2,3,CH,A)
/*
```

The above example sorts ascending on the 3-byte string which starts in position 2.

Assume MAINUSR.ADDRESS.BOOK1 in the input file has three records

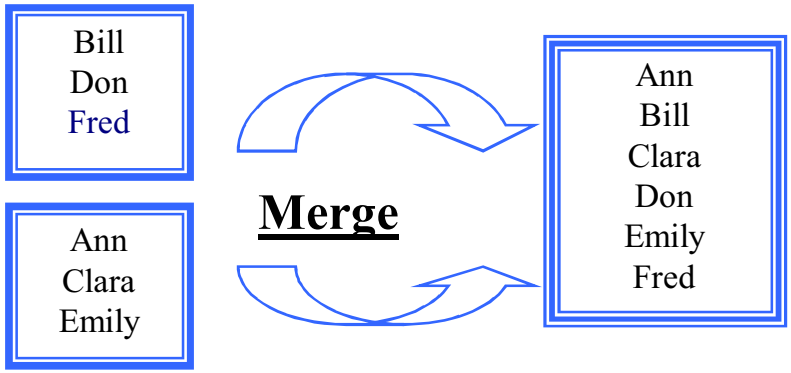
- 089134 - Ms. Patti Smith,Nevada
- 012345 - Mr.John Henley,Californa
- 042345 - Mr. Abraham Scott,NewYork

After successful completion the output file will look into this.

- 012345 – Mr. John Henley,California
- 042345 - Mr. Abraham Scott,NewYork
- 089134 - Ms. Patti Smith,Nevada

The MERGE UTILITY

The MERGE utility is used to combine two or more sorted files into a single sorted file



- This process takes records from up to 16 sorted data sets and combines them in a single sorted output data set.
- Each of the input data sets must be previously sorted in the same sequence before the merge

- In the above example two sorted data sets are merged into a single sorted data set.

Example

```
//STEP1    EXEC PGM=ICEMAN
//SYSOUT   DD   SYSOUT=*
//SORTIN   DD   DSN=MTPL.SEQ1,DISP=OLD
//SORTIN2  DD   DSN=MTPL.SEQ2,DISP=OLD
//SORTOUT  DD   DSN=MTPL.SEQ3,DISP=OLD
//SYSIN        DD *
        MERGE FIELDS =(21,2,CH,A)
/*
```

The merge control statement format is similar to that of the sort. Input data sets must already be sorted in the same sequence.

If all fields have the same format, then the merge control statement can be written in the form:

MERGE FIELDS =(21,2,CH,A)

Unit 5 Exercises

1. **The _____ utility can be used to create, copy or print sequential datasets.**

2. **The _____ utility is used to allocate and delete VSAM, non-VSAM datasets.**

3. **The _____ utility is used to copy the contents from one PDS to another.**

4. **The _____ DD name is used to code the control statement(s) for a utility program.**

5. **The _____ utility is used to print the contents of a dataset.**

6. **The _____ utility is used to list the contents of a DASD VTOC.**

7. **The sort utility uses the _____ program to sort an input dataset.**

Unit 5 Lab Exercises

Logon to TSO/ISPF and perform the following exercises. Wherever you see "userid" in lower case, substitute your valid security userid.

Create a Physical Sequential Dataset

1. In your PDS called 'userid.JCL.CNTL', create a new member called JOBTEST5.

2. Write a JOB Statement using the following criteria.
   - Job name - Your Userid & an additional character
   - Account field - your valid account number
   - Programmers name - Userid
   - Job Log & system messages - Send to print class X
   - Messages should be sent to the TSO user when JOB processing is completed.
   - Maximum execution time 10 minutes.

IEBGENER

Use the JCL listed below as a guide to this lab.

3. Add a step called //STEP1 to execute the program IEBGENER.

4. Include one DD statement called SYSPRINT. It should put the SYSOUT on the same Print class as indicated in the JOB statement.

5. Include one DD statement called SYSUT1 referring to the member JOBTEST2 on the library 'userid.JCL.CNTL'

6. Include one DD statement named SYSUT2. It should allocate a new dataset called userid.JCL.LAB5A that JOBTEST2 will be copied to.

7. Include one DD statement named SYSIN for a dataset that does not exist.

8. Submit your job and review/debug the results.

```
//STEP1    EXEC PGM=IEBGENER
//SYSPRINT DD   SYSOUT=*
//SYSUT1   DD   DSN=userid.JCL.CNTL(JOBTEST2),DISP=SHR
//SYSUT2   DD   DSN=userid.JCL.LAB5A,
//                  DISP=(,CATLG,DELETE),
//              SPACE=(TRK,(1,1)),
//              RECFM=FB,LRECL=80,UNIT=SYSDA
//SYSIN    DD   DUMMY
```

IEBCOPY

Use the JCL listed below as a guide to this lab.

1. In your PDS called 'userid.JCL.CNTL', create a new member called JOBTEST6.
2. Copy the job card from JOBTEST5, and rename the job to userid6.
3. Add a step called //STEP1 to execute the program IEBCOPY.
4. For this step:

a. Copy the member 'JOBTEST2' from userid.JCL.CNTL to a new PDS called userid.JCL.CNTL1

b. Rename 'JOBTEST2' to 'COPY1 during the copy.

```
//STEP1    EXEC PGM=IEBCOPY
//SYSPRINT DD  SYSOUT=*
//IN       DD   DSN=userid.JCL.CNTL,DISP=SHR
//OUT      DD   DSN=userid.JCL.CNTL1,DISP=(NEW,CATLG),
//         SPACE=(TRK,(2,3,1)RLSE),VOL=SER=SYSDA
//SYSIN    DD   *
         COPY   OUTDD=OUT,INDD=IN
         SELECT MEMBER=((JOBTEST2,COPY1,R))
```

UNIT 6

PROCEDURES

Procedures

Objectives

- What is a Procedure?
- Instream and Cataloged Procedures
- The PROC and PEND statements
- Coding an Instream Procedure
- Coding a Cataloged Procedure
- The JCLLIB statement
- Modifying DD statements of a Procedure
- Modifying EXEC statements of a Procedure

Objectives

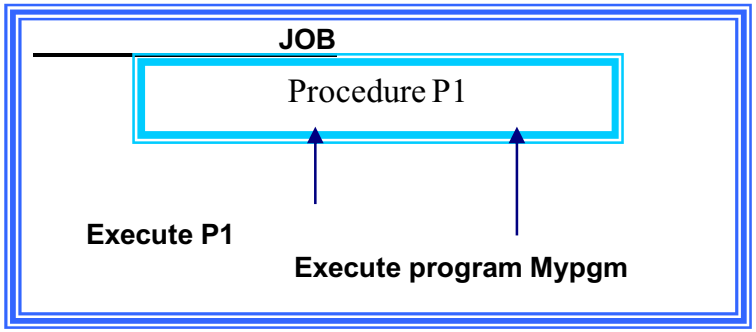
- Understand the concept of a Procedure
- Code instream procedures
- Create your own Procedure-Library with cataloged Procedures
- Use Symbolic parameters to code standard Procedures
- Create copy libraries for frequently used JCL code

What is a Procedure?

A Procedure is a JCL consisting of multiple executable steps but no JOB statement. This means that a Procedure consists of EXEC statements along with their respective DD statements.

A Procedure would generally be written for a set of programs that you want executed a number of times. Instead of coding these Steps repeatedly, you would make these Steps a part of a Procedure and would then execute the procedure as many times as required.

There are two types of procedures: Instream Procedures and Cataloged Procedures.



Benefits of using procedures:

1. Procedure saves time by reducing the time required to code JCL.
2. They save library storage by eliminating duplicate JCL.
3. They reduce JCL errors by providing access to debugged JCL.

To execute a Procedure, an EXEC statement is required along with the Procedure-name.

Example:

```
//JOBNAME   JOB    NOTIFY=&USERID
//STEP      EXEC   PROC=procedure name
```

A Procedure name is a positional parameter of the EXEC statement and therefore can be coded with out the 'PROC=' syntax, such as:

```
//JOBNAME   JOB    NOTIFY=&USERID
//STEP      EXEC   procedure name
```

Cataloged Procedures

A procedure which is part of a library is called a CATALOGED PROCEDURE.

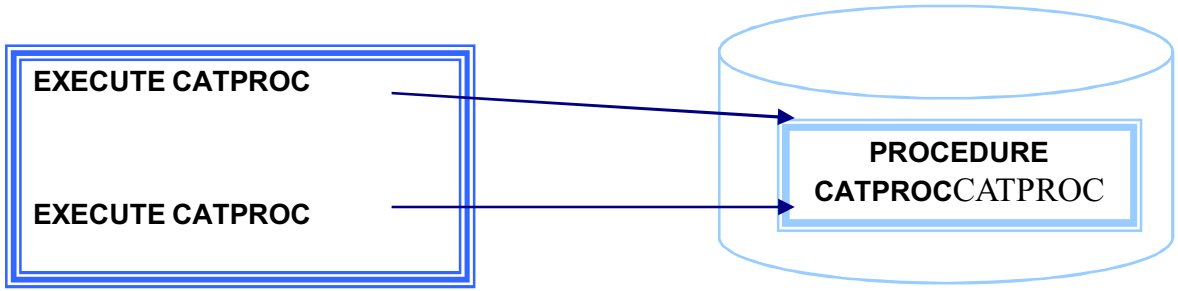


Figure 6-2

Since it a member of a Partitioned dataset, it can be used by multiple Jobs.

It is not mandatory to have PROC statement in a cataloged procedure to signify the beginning of the procedure. However, as we will see, this statement is coded to contain the default values of symbolic parameters.

The PEND statement is also not mandatory in a cataloged procedure.

Instream Procedure

When you code the procedure in the job input stream, it is called an Instream Procedure

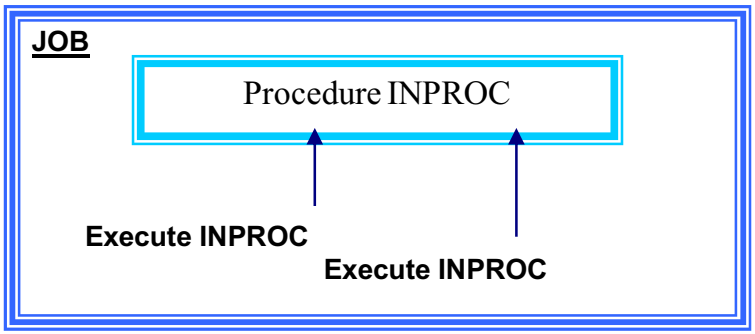


Figure 6-3

An Instream Procedure begins with a PROC statement and ends with a PEND statement.

An Instream Procedure can begin after coding the JOB statement.

Instream Procedures can only be executed by the Jobs in which they are coded.

Procedures cannot contain

- JOB statements
- DD * statements
- DD DATA statements
- Delimiter statements (‘/*’ followed by 78 blanks)
- Null statements (‘//’ followed by 78 blanks)
- Non-JCL statements (for example, JES or utility control statement)

The PROC and PEND statements in Instream Procedures

```
//JOBNAME   JOB NOTIFY=USERID
//CREPROC   PROC
//STEP1  EXEC PGM=IEFBR14
//DD1           DD      DSN=MAINUSR.JCL1.CNTL,
//                    DISP=(NEW,CATLG,DELETE),
//                    VOL=SER=LP2WK1,
//                    UNIT=SYSDA,SPACE=(TRK,(1,1,1),RLSE),
//                    DCB=(BLKSIZE=800,LRECL=80,RECFM=FB)
//SYSPRINT  DD SYSOUT=*
//              PEND
//*
//JCLSTEP       EXEC   PROC=CREPROC
//SYSIN     DD DUMMY
//*
```

The PROC statement signifies the beginning of a Procedure.

An Instream Procedure must begin with a PROC statement.

The name of the procedure is determined by the LABEL field of the PROC statement (CREPROC above). his is the Procedure name that will be used by the JOB when executing it.

The PEND statement specifies the end of a Procedure definition.

Coding an Instream Procedure

```
//MYJOB    JOB NOTIFY=USERID
//MYPROC   PROC
//STEP1    EXEC PGM=IEFBR14
//SYSPRINT DD SYSOUT=*
//         PEND
//*
//JOBSTEP EXEC PROC=MYPROC
//SYSIN  DD DUMMY
```

At Runtime:

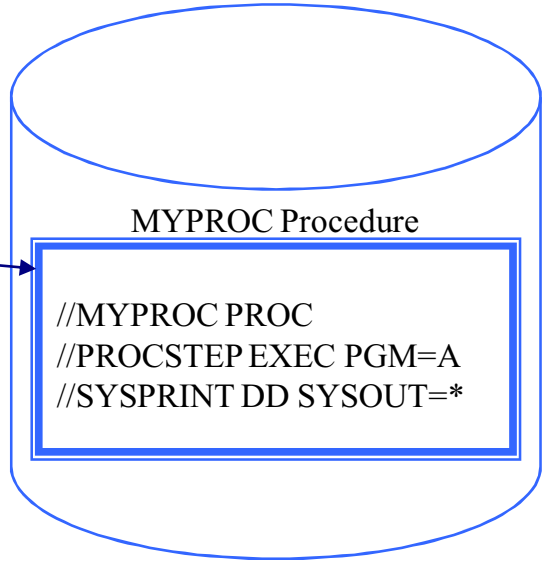
```
//MYJOB    JOB
//*JOBSTEP EXEC PROC=MYPROC
//*  This is what the system visualizes
//STEP1    EXEC PGM=IEFBR14
//SYSPRINT DD SYSOUT=*
//SYSIN    DD DUMMY
```

In the example, the instream procedure begins with the PROC statement and ends with the PEND statement.

The procedure name is ‘MYPROC’ which will be used to invoke the procedure in the EXEC statement.

Coding a Cataloged Procedure

```
//MYJOB    JOB
//STEP1    EXEC MYPROC
//SYSIN    DD DUMMY
```



During Run-time

```
//MYJOB    JOB
//PROCSTEP EXEC PGM=A
//SYSPRINT DD SYSOUT=*
//SYSIN    DD DUMMY
```

Note that the PEND statement is not necessary for a Cataloged Procedure

The member name in which the cataloged procedure is coded becomes the name of the procedure, which will be used to invoke it in a Job.

A procedure can be cataloged by placing it in one of three types of proclibs:

- SYS1.PROCLIB – IBM-supplied system procedure library.
- System PROCLIBs - defined by an installation.
- A user-defined PROCLIB – OS/390 or MVS/ESA SP V4 or Higher

Use the IEBUPDTE utility or ISPF/PDF to add a procedure to a proclib or modify a procedure. (IEBUPDTE is described in the next topic)

A PROC statement in a cataloged procedure is optional and its label is also optional. Its only function is to contain default symbolic overrides.

The JCLLIB Statement

The JCLLIB statement identifies a library (PDS) that contains catalogued procedures. Placing the JCLLIB statement in the JCL tells the JCL where to locate the procedures being executed in the job.

Example: **Procedure Library: `MAINUSR.JCL.CNTL(JCLPROC)`**

```
//JCLPROC          PROC
//STEP1  EXEC    PGM=IEFBR14
//DD1              DD          DSN=MAINUSR.PROC1.PROC,
//                             DISP=(NEW,CATLG,DELETE),VOL=SER=LP2WK1,
//                             UNIT=SYSDA,SPACE=(TRK,(1,1,1),RLSE),
//                             DCB=(BLKSIZE=800,LRECL=80,RECFM=FB)
//SYSPRINT         DD      SYSOUT=*
//          PEND
//*
```

JCL

```
//CATGC    JOB      A123,'SUSAN JOHN',.....
//DD1               JCLLIB  ORDER=(MAINUSR.JCL.CNTL,..)
//CREAT  EXEC         PROC=JCLPROC
//SYSIN  DD           DUMMY
//*
```

Rules for overriding a PROC

Parameters on the EXEC and DD statements of a PROC can be added, modified or nullified with the use of ***JCL overrides***.

Procedures are executed when invoked from a Job, in its EXEC statement. The programs coded in the procedure are executed as they are without any changes. However, JCL overrides allow you to change the parameters given in the EXEC statement and the DD statements of the Procedure. These changes are limited for that run of the JCL only.

Rules for EXEC statement overriding.

•To override an EXEC parameter, "parameter.stepname=value" must be coded when adding or replacing a parameter, and "parameter.stepname=" must be coded when nullifying a parameter.

•The PGM parameter cannot be overridden.

•All overriding EXEC parameters must be coded in the EXEC statement that invokes the procedure.

•All overrides to EXEC parameters must be completed before overriding parameters in a subsequent step.

RULES for DD statement overriding

- To override any parameters in a DD statement an independent DD statement must be coded in the following format:

- //stepname.ddname DD overriding parameters

- The sequence of overriding DD statements must be the same as the sequence of the corresponding overridden statements.

- An additional DD statement must be the last one coded in a step's overriding statements.

Overriding EXEC statements of a Procedure

```
//MYJOB  JOB  NOTIFY=USERID
//MYPROC PROC
//STEP1  EXEC PGM=IEBGENER,TIME=NOLIMIT,REGION=4M
//SYSUT1 DD DUMMY
//SYSUT2 DD DUMMY
//       PEND
//STEP2 EXEC MYPROC,TIME.STEP1=10,REGION.STEP1=
 AT RUNTIME
//MYJOB  JOB NOTIFY=USERID
//STEP1  EXEC PGM=IEBGENER,TIME=10
//SYSUT1 DD DUMMY
//SYSUT2 DD DUMMY
```

The syntax for overriding Procedure EXEC parameters is

Parameter.Procedure-stepname=value

In the example above:

§ The value of the TIME parameter has been changed from NOLIMIT to 10

§ The REGION parameter has been nullified

Overriding DD statements of a Procedure

```
//MYJOB  JOB
//MYPROC PROC
//STEP1  EXEC PGM=IEFBR14
//DD1    DD  DSN=MAINUSR.JCL.PDS,DISP=SHR,VOL=SER=LP1WK1,
//           LRECL=80
//       PEND
//STEP2 EXEC MYPROC
//STEP1.DD1 DD VOL=SER=LP2WK1,LRECL=
//STEP1.DD2 DD DSN=MAINUSR.COPYLIB,DISP=SHR
```

AT RUNTIME

```
//MYJOB JOB
//STEP1   EXEC PGM=IEFBR14
//DD1     DD   DSN=MAINUSR.JCL,PDS,DISP=SHR,VOL=SER=LP2WK1
//DD2     DD   DSN=MAINUSR.COPYLIB,DISP=SHR
```

The syntax for overriding Procedure DD parameters is

```
//PROC-STEPNAME.DDNAME DD Parameter-modifications
```

In the above example

- In DD1, LP1WK1 was changed to LP2WK1
- The LRECL parameter was nullified (discarded)
- A new DD statement DD2 was added

Symbolic parameters

A symbolic parameter can be coded in place of any parameter as part of an EXEC statement or DD statement. Symbolic parameters are variables, which hold values that can be changed when the Procedure is called.

A symbolic parameter is a name preceded by an ampersand (&) and can be a maximum of 8 characters long.

The default values for the symbolic parameter can be coded in the PROC statement.
To override the default parameter you will have to code the values for the symbolic parameter in the EXEC statement that invokes the procedure.

Symbolic overrides can be used only when symbolic parameters have been coded inside the procedure.

```
//MYPROC PROC A=LP2WK1,B=SYSDA
//STEP1  EXEC PGM=IEFBR14
//DD1    DD DSN=MAINUSR.ABC.INPUT,DISP=SHR,
//          VOL=SER=&A,
//          UNIT=&B
//       PEND
//STEPX     EXEC MYPROC,A=LP1WK1,B=SYSSQ
```

AT RUNTIME

```
//STEP1 EXEC PGM=IEFBR14
//DD1     DD DSN=MAINUSR.ABC.INPUT,DISP=SHR,
//              VOL=SER=LP1WK1,
//              UNIT=SYSSQ
```

In the above example, A and B are symbolic parameters. Instead of hard-coding the values for the VOL and UNIT parameters, they have been assigned the values contained in the symbolic parameters.

In case we do not change the value of the Symbolic parameters when the Procedure is called, the default values specified at the Procedure-declaration statement (PROC) are taken.

The SET statement

The SET statement is another way of assigning values to Symbolic parameters.

```
//MYPROC PROC A=LP2WK1,B=SYSDA
//STEP1  EXEC PGM=IEFBR14
//DD1    DD   DSN=MAINUSR.INPUT.FILE1,DISP=SHR,
//              VOL=SER=&A,
//              UNIT=&B
//       PEND
//SET1   SET A=LP1WK1
//SET2   SET B=SYSSQ
//STEPX  EXEC MYPROC
```

AT RUNTIME

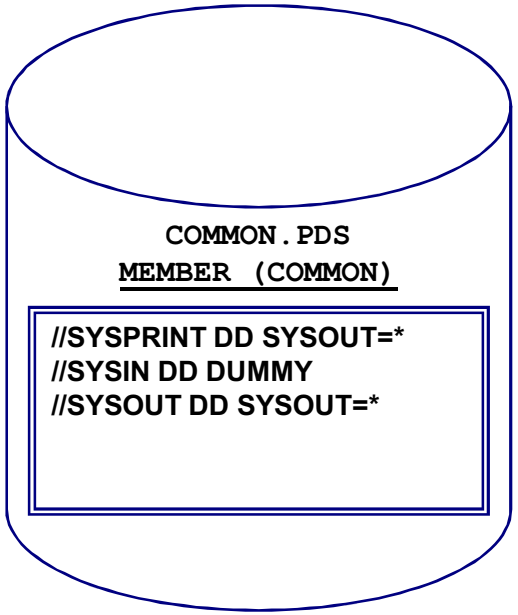
```
//STEP1 EXEC PGM=IEFBR14
// DD1 DD DSN= MAINUSR.INPUT.FILE1,DISP=SHR,
//           VOL=SER=LP1WK1,
//           UNIT=SYSSQ
```

The SET statement can appear anywhere in a JCL between the JOB statement and the first point where a SET –statement-assigned symbolic parameter is referenced.

The INCLUDE statement

The INCLUDE statement allows you to copy statements from any member of the PDS(s) listed in the JCLLIB statement.

Similar to the way PROCs are used, INCLUDE allows you to code a single set of JCL statements that you can use in multiple jobs.



```
//MYJOB JOB
//DD1   JCLLIB  ORDER=COMMOM.PDS
//STEP1 EXEC    PGM=MYPGM
//INDD  DD      DSN=A.B.C,DISP=SHR
//INC1  INCLUDE MEMBER=COMMON
```

AT RUNTIME

```
//MYJOB    JOB
//DD1      JCLLIB ORDER=COMMON.PDS
//STEP1    EXEC   PGM=MYPGM
//INDD     DD     DSN=A.B.C,DISP=SHR
//*INC1 INCLUDE MEMBER.COMMON
//SYSPRINT DD     SYSOUT=*
//SYSIN    DD     DUMMY
//SYSOUT   DD     SYSOUT=*
```

UNIT 7

CONDITIONAL EXECUTION

Conditional Execution

- Objectives
- COND Parameter
- IF-THEN ELSE Statement

Objectives

Understand the need for the Conditional statement Understand the different types of COND Parameter and their usage

Understand the IF-THEN ELSE Statement and how to use it

COND – Introduction

Once a JOB starts executing, all of the steps are executed in the order in which they are coded.

JES returns a condition code after the execution of each Job Step in a job

This condition code has a special meaning in the execution process. A condition code of zero specifies a successful execution, while a code other than zero indicates some kind of error. The error code can be a value ranging from 0 to 4095.

Some typical non-zero condition codes (and their meaning) are:

```
COND CODE   0000     Program execution was completely successful.
COND CODE   0004     Execution was ok but cause warning messages.
COND CODE   0008     Program execution was seriously flawed.
COND CODE   0012     Program execution was very seriously flawed.
COND CODE   0016     Program failed disastrously.
```

Condition codes are different from ABEND codes.

If a Job STEP abnormally terminates (ABENDS), then all subsequent Steps are bypassed.

If a step returns a non-zero condition code, processing continues.

The COND parameter gives you the ability to determine which steps of the job should execute, based on the condition codes returned in previous steps.

The COND parameter is used on the JOB and EXEC statements.

Another way to control step execution based on condition codes is the IF-THEN-ELSE-ENDIF statement, discussed later in this unit.

Why Use The COND Parameter?

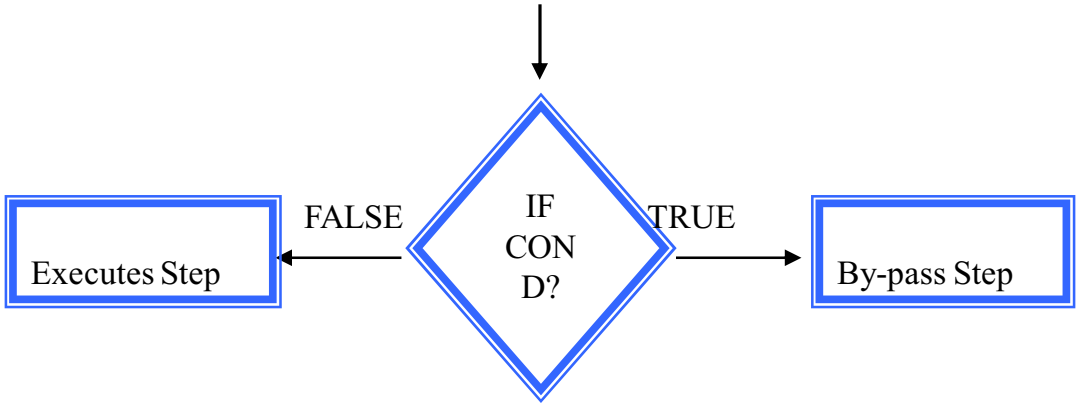
There are two reasons why we need the COND parameter:

1. To define conditions under which a step of a normally executing job is to be bypassed.

2. To process a step even if a previous step in a job has abended or only if a previous step has abended. Ordinarily MVS will terminate the entire job when a step abends. The COND parameter allows you to conditionally execute steps based on whether or not the job abended (such as executing a recovery step at the point of an abend).

Parameter –COND

The COND parameter can be coded on both JOB and EXEC statements. The flow chart below explains how the COND parameter works.



The COND parameter states that if a condition is true then the step in which this COND parameter is coded is to be bypassed. If the condition is false than the step is to be executed.

Syntax

The pattern to code COND parameter is

COND=(value, operator, stepname)

(0-4095) GT , The name of a previous step that returned

LT the condition code to be tested.

EQ

NE

GE

LE

COND OPERATORS

GT	greater than
LT	less than
EQ	equal to
NE	not equal to
GE	greater than or equal to
LE	less than or equal to

Example:

```
//STEP08 EXEC PGM=IEWL,COND=(4,LT,STEP06)
```

As mentioned earlier, the COND parameter states that if a condition is true then the step in which this COND parameter is coded is to be bypassed.

If the condition is false than the step is to be executed.

In the above example, if 4 is less than the condition code returned for STEP06, then STEP08 will be bypassed and will not execute.

If STEP06 has this CCODE	Will STEP08 Execute?	Why?
0000	Yes	Because 4 is not less than 0.
0001	Yes	Because 4 is not less than 1.
0002	Yes	Because 4 is not less than 2.
0003	Yes	Because 4 is not less than 3.
0004	Yes	Because 4 is not less than 4.
0005	No	Because 4 is less than 5.
0006	No	Because 4 is less than 6.

Compound Tests

You may specify up to eight individual tests for a single step.

The format is:

```
COND=(value,operator,stepname),(value,operator,stepname)
```

When the defined condition is satisfied it causes the step to be by-passed.

There is an implied OR relationship in any compound COND test.

Any one condition that is satisfied is sufficient to by-pass the step.

Example:

```
//LKED EXEC PGM=IEWL,COND=((4,LT,COB),(4,LT,PC))
```

In this example, the step 'LKED' is conditionally executed based on two separate steps – 'COB' and 'PC'.

Coding The COND Parameter On The JOB Statement

The JOB statement COND parameter performs the same return code tests for every step in a job. If the JOB statement's return code test is satisfied, the job terminates.

The JOB COND parameter performs its return code tests for every step in the job, even when EXEC statements also contain COND parameters.

If any step satisfies the return code test in the JOB statement, the job terminates. The job terminates regardless of whether or not any EXEC statements contain COND parameters and whether or not an EXEC return code test would be satisfied.

If the JOB statements return code test is not satisfied, the system then checks the COND parameter on the EXEC statement for the next step. If the EXEC statement return code test is satisfied, the system bypasses that step and begins processing of the following step, including return code testing.

Example 1:

```
//JOB1  JOB  ,'LEE BURKET',COND=((10,GT),(20,LT))
```

This example asks 'Is 10 greater than the return code or is 20 less than the return code? If either is true, the system skips all remaining job steps. If both are false the system executes all job steps.

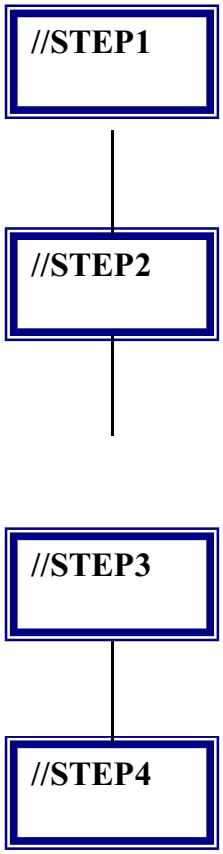
For example, if a step returns a code of 12, neither test is satisfied, the next step is executed. however, if a step returns a code of 25, the first test is false, but the second test is satisfied: 20 is less than 25. The system bypasses all remaining job steps.

Example 2:

```
//J2  JOB  ,'D WEISKOPF',COND=((50,GE),(60,LT))
```

This example says 'If 50 is greater than or equal to a return code, or 60 is less than a return code, bypass the remaining job steps.' In other words, the job continues as long as the return codes are 51 through 60.

COND parameter coded in the EXEC statement



EXEC PGM=A

EXEC PGM=B, COND=(0,LT, STEP1)

Program B is not executed if program A returns any non-zero COND CODE.

EXEC PGM=C, COND=(0,EQ, STEP1)

Program C is not executed if program A returns a COND CODE = 00

EXEC PGM=D, COND=(4,EQ)

Program D is not executed if program A or program B or program C returns any COND CODE = 04

The COND parameter can be coded on second and subsequent steps of a job stream.

You can define one or more conditions on a step.
If any condition in a COND is true the step is by-passed.

If you omit the stepname in the COND parameter,
the test applies to all previous steps (such as in //STEP4 in the example above).

EVEN and ONLY

The job can progress in any one of two modes

1. Normal processing mode.
2. Abnormal termination mode.

All jobs start up in normal processing mode, but when a program abends MVS changes the job to abnormal termination mode. In normal processing mode MVS recognizes each EXEC step and attempts to execute the program coded at it. If any COND conditions on an EXEC are true MVS does not run that step but goes on to the next step.

A job gets into an abnormal termination mode because a program has abended.

MVS dumps out the program that fails and flushes out the reminder of the job with no further processing. MVS will acknowledge the presence of the steps remaining after the step that failed, but will not process them. EVEN and ONLY are used under such cases.

EVEN and ONLY prevents MVS from flushing out the rest of the steps when an abend occurs and prevents exiting the program.

If COND=EVEN is coded on an EXEC statement, MVS will process the step even when the job was already put into abend mode by the failure of a previous step.

If COND=ONLY is coded, MVS will process the step *only* if the job is in abend mode. Such a step will be ignored in normal processing mode.

EVEN or ONLY can be coded in a step to gain MVS's attention in abend mode, but both can't be coded together.

```
//STEP2 EXEC PGM=IEBGENER,COND=EVEN
//STEP3 EXEC PGM=IEBGENER,COND=ONLY
```

EVEN or ONLY can be used along with normal COND CODE test.

```
//STEP4 EXEC PGM=IEBGENER,COND=((4,LT,STEP1),EVEN)
```

In this example, MVS will process STEP4 even if a previous step bends.

EVEN and ONLY do not guarantee that a Job step will be executed if a previous step has abended. Only control comes to this step and then a check of all JOB and EXEC level COND parameters are done before actually executing the step.

Effects of EVEN and ONLY

	Normal processing Mode	Abnormal termination mode
No COND code -or- COND return code tests only	**Step is processed, Return code tests can cause step to be skipped.**	**Step is acknowledged but not processed.**
COND=EVEN	**Step is processed, return code tests can cause step to be skipped.**	**Step is processed Return code tests can cause step to be skipped.**
COND=ONLY	**Step is acknowledged but not processed.**	• **Step is processed** • **Return code tests can cause step to be skipped.**

IF/THEN/ELSE/ENDIF

The way you code the IF/THEN/ELSE/ENDIF statement construct determines whether the statement construct tests all job steps, a single job step, or a procedure step.

Job Level Evaluation

If you do not code a stepname, the IF/THEN/ELSE/ENDIF statement construct evaluates the return code, abend condition, or run condition of every previous step in the job. If the

condition (return code, abend condition or run condition) is satisfied, based on the steps in the job that have executed thus far, the system executes the THEN clause.

Step Level Evaluation

To test a single step, code the stepname of the step you want to test. To test a procedure step, code the stepname.procstepname of the procedure step you want to test. If the step or procedure step that you are evaluating did not execute, was cancelled or ended abnormally, the result of the evaluation is false.

You can code the IF/THEN/ELSE/ENDIF statement construct anywhere in the job after the JOB statement. Code it as follows:

```
//[Name]                IF   (relational expression) THEN
//STEPTRUE       EXEC
//[Name]                ELSE
//STEPFALS       EXEC
//                      ENDIF
```

Example:

```
//MODAL2    JOB         '0.2AMIP',…..
//STEP1     EXEC      PGM=IDCAMS
//SYSPRINT  DD          SYSOUT=*
//SYSIN     DD            *
```

For the above example given, first it does the repro (copy) for the files cobol.source.file2, cobol.source.file4 from the files cobol.source.file1, cobol.source.file3, if MaxCC is greater than zero then set that to 16 i.e. (MaxCC=16), else it will delete the input datasets.

The relational expression (Continued)

Example 1: This example tests the return code for a step.

```
//RCTEST     IF   (STEP1.RC GT 20|STEP2.RC = 60)   THEN
//STEP3      EXEC PGM=U
//ENDTEST    ENDIF
//NEXTSTEP   EXEC
```

The system executes STEP3 if

•The return code from STEP1 is greater than 20, or the return code from STEP2 equals 60.

•If the evaluation of the relational expression is false, the system bypasses STEP3 and continues processing with step NEXTSTEP.

Example 2: This example tests for an ABEND condition in a procedure step.

```
//ABTEST      IF      (STEP4.LINK.ABEND=FALSE) THEN
//BADPROC     ELSE
//CLEANUP     EXEC    PGM=ERRTN
//ENDTEST     ENDIF
//NEXTSTEP    EXEC
```

The relational expression tests that an ABEND did not occur in procedure LINK, called by the EXEC statement in STEP4. If the relational expression is true, no ABEND occurred. The null THEN statement passes control to step NEXTSTEP. If the relational expression is false, an ABEND occurred. The ELSE clause passes control to the program called ERRTN. This example implies the following naming convention – STEP4.LINK.ABEND=FALSE – is how you would test for an ABEND in STEP4.LINK – meaning that you type the job step and procstep followed by a dot (.) and the work ABEND.

Example 3: This example tests for a user abend completion code in the job.

```
//CCTEST   IF   (ABENDCC = U0100) THEN
//GOAHEAD  EXEC  PGM=CONTINUE
//NOCC     ELSE
//EXIT     EXEC  PGM=CLEANUP
//ENDIF
```

If any job step produced the user abend completion code 0100, the EXEC statement GOAHEAD calls the procedure CONTINUE. If no steps produced the completion code, the EXEC statement EXIT calls program CLEANUP.

```
REPRO                              -
INDATASET(COBOL.SOURCE.FILE1)      -
OUTDATASET(COBOL.SOURCE.FILE2)

REPRO                              -
INDATASET(COBOL.SOURCE.FILE3)      -
OUTDATSET(COBOL.SOURCE.FILE4)
   IF   MAXCC GT 0
   THEN
   SET   MAXCC=16
 ELSE
   DELETE(COBOL.SOURCE.FILE1)
   DELETE(COBOL.SOURCE.FILE3)
```

For the above example given, first it does the repro (copy) for the files cobol.source.file2, cobol.source.file4 from the files cobol.source.file1, cobol.source.file3, if MaxCC is greater than zero then set that to 16 i.e. (MaxCC=16), else it will delete the input datasets.

The Relational Expression

There are four types of relational operators:

- Comparison operators
- Logical operators
- Not (¬) operators
- Relational expression keywords.

Comparison operators (GT, LT, etc.) compare a relational expression keyword to a numeric value. The comparison results in a true or false condition.

The logical operators & (AND) and | (OR) indicate that the system should evaluate the Boolean result of two or more relational expressions.

The ¬ (NOT) operator reverses the testing of the relational expression.

Relational expression keywords indicate that you are evaluating a return code, ABEND condition, or ABEND completion code.

The THEN clause or ELSE clause must contain at least one EXEC statement. The EXEC statement indicates a job step that the system executes based on its evaluation of the relational expression. A THEN or ELSE clause that does not contain an EXEC statement is a null clause.

You can nest IF/THEN/ELSE/ENDIF statement constructs up to 15 levels of nesting.

Example 1: This example tests the return code for a step.

```
//RCTEST     IF   (STEP1.RC GT 20|STEP2.RC = 60)   THEN
//STEP3      EXEC PGM=U
//ENDTEST    ENDIF
//NEXTSTEP   EXEC
```

The system executes STEP3 if:

•The return code from STEP1 is greater than 20, or

•The return code from STEP2 equals 60.

If the evaluation of the relational expression is false, the system bypasses STEP3 and continues processing with step NEXTSTEP.

UNIT 8

GDG

GDG (Generation Data groups)

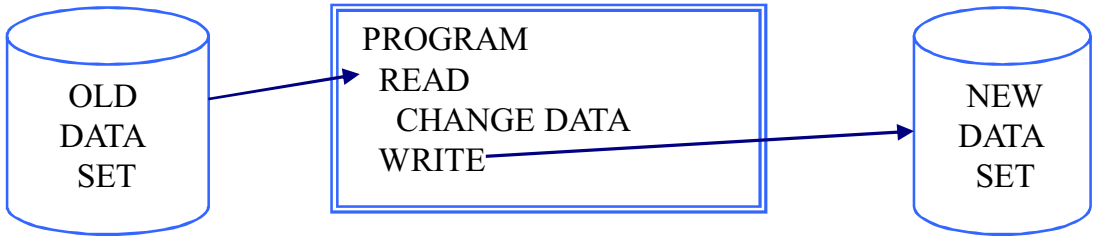
- Generation Data Sets
- Relative Generation Numbers
- Building a GDG Base Entry
- Defining Attributes for Generation Data Sets
- Creating a Generation Data Set
- Retrieving a Generation Data Set

Objectives

- Understand the need for the GDGs

- Ways of Coding GDGs

Generation Data Groups



A Generation Data Group (GDG) consists of like-named data sets that are chronologically or functionally related. A data set in a GDG is called a generation.

Why use Generation Data Groups (GDGs)?

You can only catalog one entry of a particular dataset name. In other words, once a dataset (ABC.DATA for example) is cataloged, you cannot catalog a new version of that dataset (another ABC.DATA).

Case1: You attempt to catalog a new data set with the same name as an existing data set.

CATALOG ERROR1 NAME ALREADY IN THE CATALOG.

Case2: You keep the new data set with the same name.

JCL ERROR! DUPLICATE NAME ON VTOC.

Case3: You catalog or keep the data set with different names.

JCL STATEMENT MUST BE CHANGED TO THE NEW NAMES

PURPOSE OF GDG's:

For datasets that get created on a daily basis, GDG's allow you to create new versions of a dataset where the base name of the dataset remains the same, and a new *generation* number gets tagged onto the end of each new version of the dataset name.

One example of GDG use might be a daily transaction log. Each day a new file is created showing the transactions of that day. The base name of the transaction log might be TRANS.LOG, where the actual generations would be called TRANS.LOG.G0001V00, TRANS.LOG.G0002V00, TRANS.LOG.G0003V00, etc.

GDG- DSNAME specification

Generation data sets (generations) have two types of names: Relative and Absolute

```
DSN=ABD.DAILY.TRANS(+n)DSN=ABC.DAILY.TRANS.GxxxxVyy

(+n) Add a new generationGxxxxVyy:
(+0)   Use   current   generation   numberxxxxGeneration
number
(-n) Use an old generationyyVersion number
```

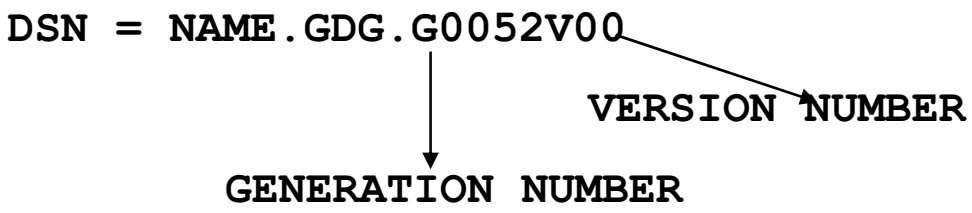
```
EXAMPLE
DSN=ABC.DAILY.TRANS(+1)DSN=ABC.DAILY.TRANS.G0053V00
```

Absolute Data Set Name:

The absolute data set name is the true data set name you would see on the volume table of contents

(VTOC).

The Format is:



Relative Data Set Name:

The relative name is the most common form of GDG name used in JCL coding. It refers to a generation number relative to the most current generation in the catalog.

The format is:

`DSN=NAME.GDG(+n)` **`or`** **`DSN = NAME.GDG(-n)`**

where 'n' is the relative generation number of the GDG.

Possible Formats

`(+n)` Create a new generation by adding 'n' to the current generation in the catalog

`DSN=NAME.GDG(+1)`

`(0)` Use the current generation in the catalog **`DSN=NAME.GDG(0)`**

`(-n)` Use the generation that is 'n' less than the current generation

`DSN=NAME.GDG(-1)`

Example

Current Catalog Entries

```
ABC.TRANS.DAILY.G0053V00

ABC.TRANS.DAILY.G0052V00

ABC.TRANS.DAILY.G0051V00
```

With the current catalog entries shown above, the following GDG's would be translated as shown:

Dataset	Translates to
DSN=ABC.TRANS.DAILY(-1),DISP=OLD	**ABC.TRANS.DAILY.G0052V00**

The current generation in the catalog is G0053V00. Since 53 – 1 = 52, G0052V00 is the generation requested. This is common when reading the previous cycle's file.

The current generation in the catalog is G0053V00. This is used to read the most current file available.

```
DSN=ABC.TRANS.DAILY(+1),DISP=(NEW,CATLG)
```

```
ABC.TRANS.DAILY.G0054V00
```

The current generation in the catalog is G0053V00. Since 53 + 1 = 54, G0054V00 is the generation requested. This is used when creating a new file for the current cycle.

GDG- DSNAME specification (Continued)

The generation number starts with G0001 for the first generation and is incremented by the value coded in the relative data set name.

For Example:

RELATIVE NAME	ABSOLUTE NAME	
`NAME.GDG(+0)`	`NAME.GDG.G0003V00`	current generation
`NAME.GDG(-1)`	`NAME.GDG.G0002V00`	current generation -1
`NAME.GDG(-2)`	`NAME.GDG.G0001V00`	current generation -2

IBM does not use the V00 or Version number. The version number is set to V00 with the first generation. If a generation is damaged and needs to be replaced, an installation can code the absolute name with a new version number, to replace the damaged generation, such as

```
//INDD      DD DSN=NAME.GDG.G0052V00,DISP=OLD
//OUTDD     DD DSN=NAME.GDG.G0052V01,DISP=(,CATLG)
```

GDG DSNAME – BASE CATALOG ENTRY

Before a generation data set can be created by a job, a GDG base Catalog entry and model DSCB (or

pattern DSCB) must be defined to the catalog. (SMS does not support model DSCB's).

This is a sample IDCAMS job stream, which can be used to create both the catalog entry and build a model DSCB.

```
//JOBNAME  JOB, ,.....
//STEP1    EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=*
//DSCB     DD DSN=TEST.MODELDSCB.GDG1,SPACE=(TRK,0),
            DCB=(RECFM=FB,LRECL=80,BLKSIZE=800),
            UNIT=SYSDA,DISP=(,KEEP)
//SYSIN    DD *
      DEFINE GENERATIONDATAGROUP  -
      (NAME (TEST.GDG)            -
      NOSCRATCH                   -
      NOEMPTY                     -
      LIMIT (3))
```

There are five control statement parameters which are used in creating a GDG base catalog entry. They are:

LIMIT Maximum number of generations allowed for this GDG entry

EMPTY When limit is exceeded, uncatalog all generations.

NOEMPTY When limit is exceeded, uncatalog oldest entry only

SCRATCHDelete any uncataloged generation.

NOSCRATCH Do not delete any uncataloged generation.

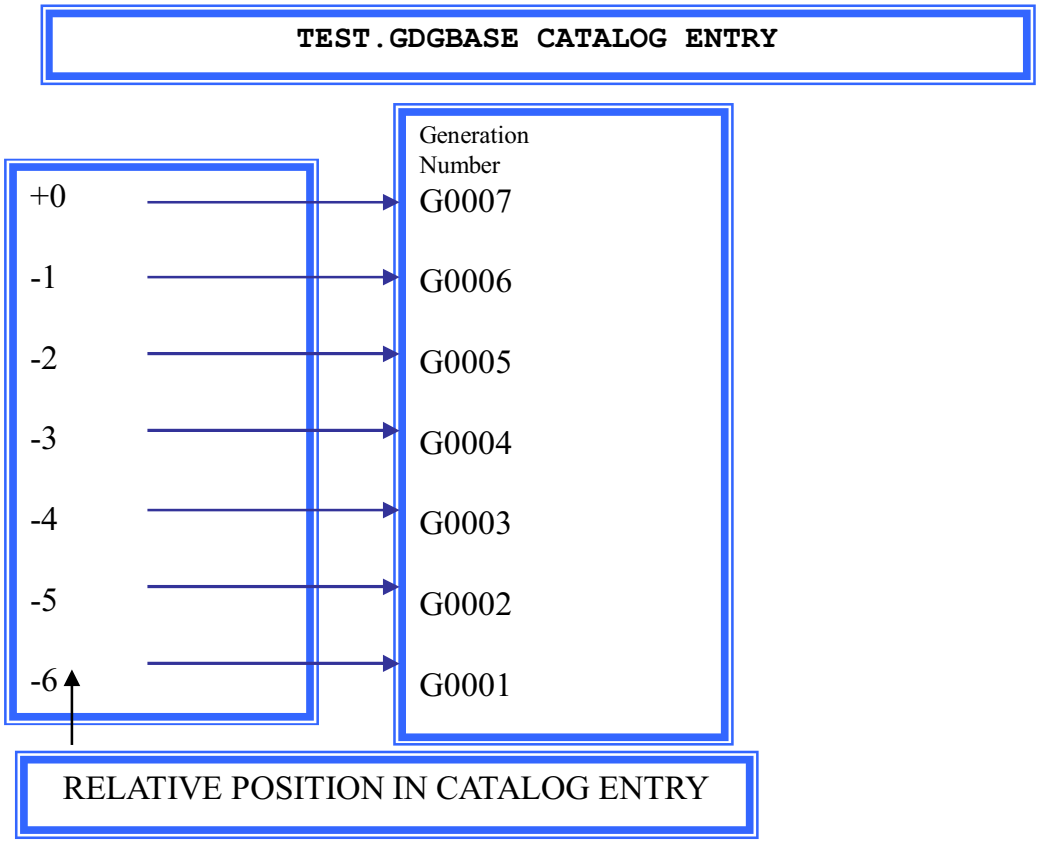
The model DSCB (//DSCB in the example above) ensures that the same DCB and EXPDT information is used to create all generations. This ensures greater consistency among generations.

The model DSCB must be allocated with Zero space, and it cannot be cataloged.

The model DSCB must also lie on the same volume where the base catalog entry is cataloged.

GDG DSNAME – CATALOG ENTRY

The following example shows a view of a GDG Catalog entry.



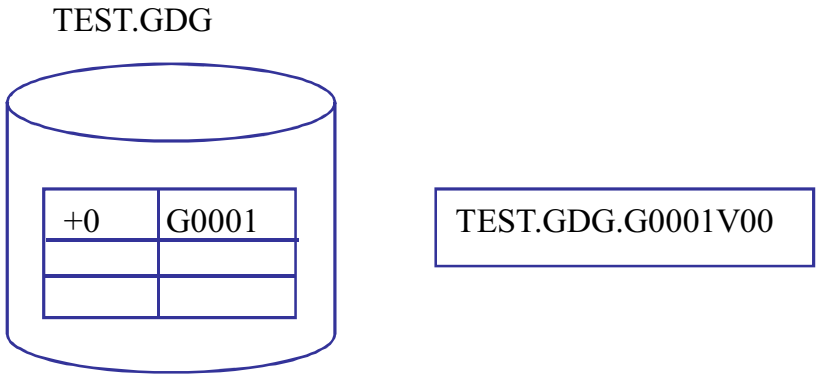

```
EXAMPLES:
//DD1 DD DSN=TEST.GDG(+0),DISP=OLD     locates current(top) entry
//DD2 DD DSN=TEST.GDG(-6),DISP=OLD     locates oldest entry
```

Notice that the catalog entry operates like a pushdown stack.
As new generations get created, the relative numbers for existing generations changes

GDG EXAMPLE - FIRST GENERATION

The following example shows what happens when the first generation of a GDG gets created.

```
//EXAMPLEJOB  378,SMITH,CLASS=T
//STEP1EXECPGM=USERPGM1
//FIRSTDDDSN=TEST.GDG.(+1),DISP=(,CATLG),
//INPUTDDDSN=INITIAL.DATA,DISP=OLD
```



As the first generation of the generation data group named TEST.GDG is created:

Note that the DISP=(,CATLG) is required when creating a new generation.

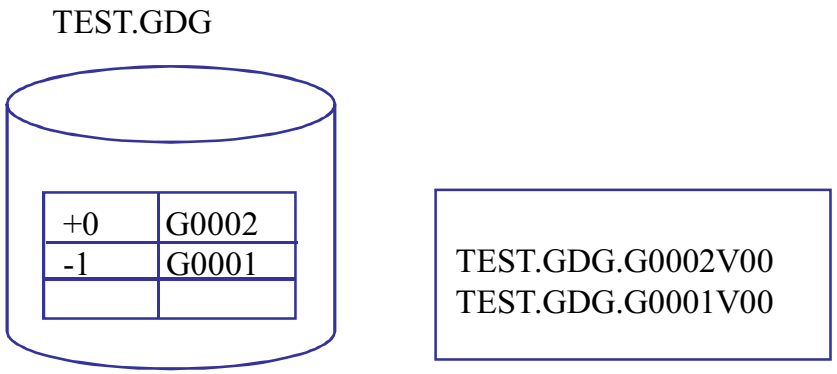
Relative generation (+0) is generation G0001V00.

The GDG Base must be created prior to creating any generation of the dataset

GDG EXAMPLE – SECOND GENERATION

The following example shows what happens when the second generation of a GDG gets created.

```
//EXAMPLEJOB378,SMITH, CLASS=G
//STEPXEXECPGM=MAINLINE
//GDGINDDDSN=TEST.GDG.(+0),DISP= OLD
//GDGOUTDDDSN=TEST.GDG(+1),DISP=(NEW, CATLG),
//
```



As the second generation of the generation data group named TEST.GDG is created:

The current generation (0) is used as input. You are not required to code the plus sign with the zero (+0), a zero without the plus sign (0) is acceptable and commonly used.

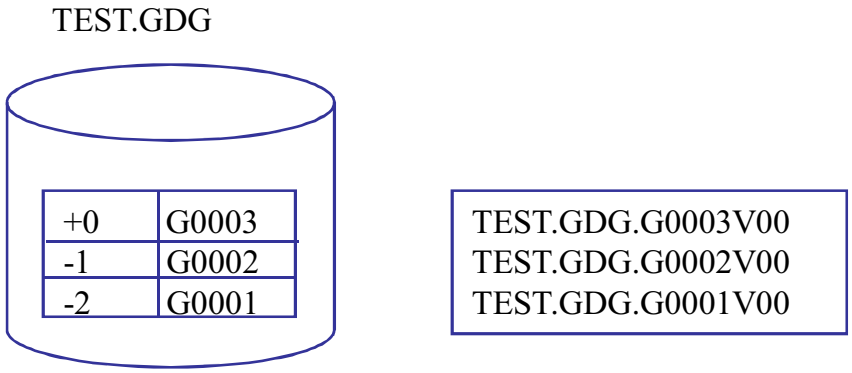
A new generation (+1) is created.

Notice that the catalog entry operates like a push-down stack. Generation G0001V00 is now the (-1) generation.

GDG EXAMPLE – THIRD GENERATION

The following example shows what happens when the third generation of a GDG gets created.

```
//EXAMPLEJOB378,SMITH, CLASS=G
//STEPXEXECPGM=MAINLINE
//GDGINDDDSN=TEST.GDG(+0),DISP= OLD
//GDGOUTDDDSN=TEST.GDG(+1),DISP=(NEW, CATLG),
//
```



As the third generation of the generation data group named TEST.GDG is created:

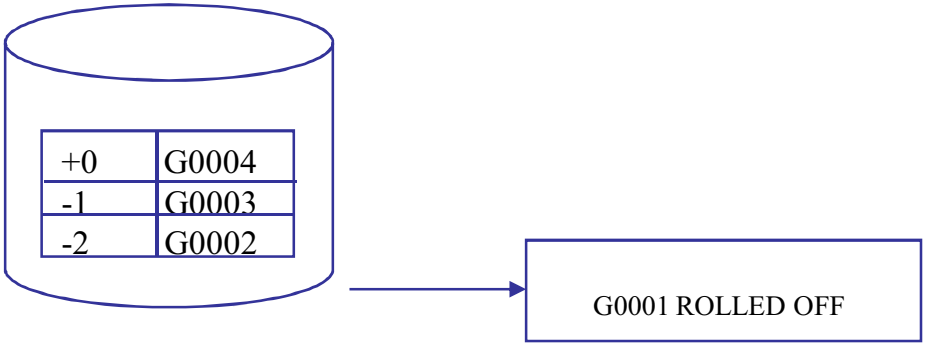
- The current generation (0) is used as input.
- A new generation (+1) is created.
- Notice that the catalog entry operates like a pushdown stack.

GDG EXAMPLE - LIMIT EXCEEDED

Once a GDG reaches the number of generations defined in its LIMIT, the oldest generation "rolls off" the catalog as a new generation gets created.

The old generation may automatically be kept or scratched, depending on the option you chose when you defined the GDG base.

TEST.GDG




```
//EXAMPLEJOB 378,SMITH, CLASS=G
//STEPXEXECPGM=MAINLINE
//GDGINDDDSN=TEST.GDG(+0),DISP= OLD
//GDGOUTDDDSN=TEST.GDG(+1),DISP=(NEW, CATLG),
//
```

As the fourth generation of the generation data group named TEST.GDG is created:

- The current generation (0) is used as input.
- A new generation (+1) is created
- The first generation, **`TEST.GDG.G0001V00`**, will be removed from the list of cataloged entries, since the LIMIT was defined as 3.
- Since NOSCRATCH was specified, this generation will be kept, though it will no longer be cataloged

Other Notes

- If the generations are SMS managed, this generation will remain in the catalog.
- If the generations were NN-SMS managed, the generation would have been uncataloged.
- The maximum no. of generations that MVS can manage for a data set is 255.

GDG's IN A MULTISTEP JOB

```
//STEP1  EXEC          PGM=ONE
//INBIG          DD              DSN=A.AIR,DISP=OLD
//OUT1     DD          DSN=A.AIR(+1),DISP=(,CATLG,DELETE)  …….

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

//STEP2 EXEC           PGM=TWO
//IN2            DD    DSN=PROD.DATA,DISP=OLD
//OUT2             DD  DSN=A.AIR(+2),DISP=(,CATLG,DELETE)  …

----------

//STEP3 EXEC           PGM=THREE
//IN3            DD    DSN=A.AIR(+1),DISP=OLD
//               DD    DSN=A.AIR(+2),DISP=OLD
```

STEP1

•A GDGALL request (DSN=A.AIR) is IBM's terminology for automatic concatenation of ALL generations currently in the catalog.

•The (+1) generation in DD 'OUT1' creates a new generation.

STEP3

•Notice that the (+1) generation is now OLD. Even though that generation was cataloged in STEP1, and the catalog pushed down, you do not adjust the relative generation numbers within the same job. The system does that internally.

•As additional generations are specified in this step, increment the relative generation number. An increment of 1 is shown, but any increment can be used.

Notes

Relative GDG numbers are not updated until end of job, therefore the relative numbers are held constant throughout a single job.

•Use DISP=OLD when processing a GDG to prevent other jobs (on the same system) from processing the same GDG at the same time.

Creating a Generation Data Set (Non-SMS)

When creating a new non-SMS-managed generation data set, always code the parameters

- DSNAME
- DISP
- UNIT

Optionally, code the parameters

- VOLUME
- SPACE
- LABEL
- DCB

In the DSNAME parameter, code the name of the GDG followed by a number, +1 to +255, in parentheses.

If this is the first dataset being added to a GDG in the job, code +1 in parentheses. Each time in the job you add a dataset to the same GDG, increase the number by one.

When referring to this data set in a subsequent job step, code the relative generation number used to create it on the DSNAME parameter. You cannot refer to the dataset in the same step in which it was created.

At the end of the job, the system updates the relative generation numbers of all generations in the group to reflect the additions.

Deleting a Generation Data Set

Use the DELETE parameter of the IDCAMS utility to delete a GDG from the catalog, such as:

```
//MODAL2     JOB   '0.2AMIP'
//STEP1      EXEC  PGM=IDCAMS
//SYSPRINT   DD    SYSOUT=*
//SYSIN      DD    *
DELETE   ABC.DATA.MONTHLY   FORCE
/*
//
```

In this example the GDG called **ABC.DATA.MONTHLY** is deleted. The FORCE parameter physically purges all entries related to GDG.

UNIT 9

SMS

SMS (Storage Management Subsystem)

Objectives

- Introduction
- Advantages of SMS
- New JCL Parameters
- SMS Constructs
- Specifying Constructs

Objectives

- Understand basic SMS Concepts
- Understand the Advantages of SMS
- Understand Overriding Attributes (New JCL PARM)
- Understand SMS Constructs
- Understand Migration and Backup with SMS

Why Use SMS (Storage Management Subsystem)

Storage Management Subsystem (SMS) is an optional feature of MVS, which is used to improve the management of available disk space in the data center.

In JCL, normally the developer must specify allocation attributes of datasets, such as the size and location about the data sets, to the operating systems. If they don't allocate files correctly, this can lead to wasted disk space, and makes it difficult to manage the DASD in the data center.

ADVANTAGES OF SMS

SMS managed data sets have several advantages:

Users are relieved of making decisions about the resource allocation of datasets, since it is handled by SMS.

SMS provides the capability of concatenating data sets of unlike devices.

SMS Managed data sets cannot be deleted unless they are first uncataloged

Due to this extra step, erroneous deletion of data sets is minimized.

Additional features are available in the use of IDCAMS in the SMS environment.

VSAM data sets created in an SMS environment offer more flexibility than those created through JCL in a non–SMS environment.

With SMS, the system obtains information about the attributes of a data set from the data class for the data set. In many cases, the attributes defined in the data class, selected by an installation-written automatic class selection (ACS) routine, are sufficient for the data sets you create with DD statements.

Additional JCL PARAMETERS

* RECFM (record format)

* LRECL (record length)

* SPACE (average record length, primary, secondary, and directory quantity)

*VOLUME (volume-count)

The above parameters were already discussed.

RECORG (record organization) or

KEYLEN (key length)

KEYOFF (key offset

AVGREC (record request and space quantity)

RETPD (retention period) or

EXPDT (expiration date)

DSNTYPE (data set type, PDS or PDSE

The above parameters are also JCL parameters, but which are used in VSAM.

The storage administrator at your installation defines the names of data classes and their data set attributes.

To view a list of data class names and their attributes, use the Interactive Storage Management Facility (ISMF).

SMS Constructs

With SMS, a new data set can have one or more of the following three constructs:

Data class - contains the data set attributes related to the allocation of the data set.

Management class - contains the data set attributes related to the migration and backup of the data set. A management class can only be assigned to a data set that also has a storage class assigned.

Storage class - contains the data set attributes related to the storage occupied by the data set. A data set that has a storage class assigned is defined as an "SMS-managed data set". The storage administrator at your installation writes the automatic class selection (ACS) routines that SMS uses to assign the constructs to a new data set.

For example, with SMS you can code the DDNAME, DSNAME, and DISP parameters to define a new data set:

```
//SMSDS0   DD   DSNAME=MYDS0.PGM,DISP=(NEW,KEEP)
```

and retrieve the data set with:

```
//SMSDSR   DD   DSNAME=MYDS0.PGM,DISP=MOD
```

In the example, installation-written ACS routines (possibly based on the data set name and information from your JOB, EXEC, and DD statements) can select a data class, management class, and storage class appropriate for the data set.

You code only the ddname, dsname, and disposition of the data set.

The constructs selected by the ACS routines contain all the other attributes needed to manage the data set

Specifying Constructs

In many cases, the constructs selected by the installation-written ACS routines are sufficient for your data sets.

However, when defining a new data set, you can select a data class, management class, or storage class by coding one or more of the following DD parameters:

- DATA class - specifies the data class
- MGMTCLAS - specifies the management class
- STORCLAS - specifies the storage class

The storage administrator has defined the names of the classes you can specify.
You can view the names and attributes defined in each of the named classes by using ISMF.

Unit 9 Exercises

1. SMS stands for _____ _____ _____.

2. _____ contains allocation attributes that describe the logical data format.

3. _____ contains desired performance and availability objectives.

4. _____ defines a list of volumes for data allocation.