2. JOB CONTROL LANGUAGE

2.1 Introduction to Job Control Language

No one is expected to immediately understand all of the topics discussed here or, for that matter, all of the topics of Job Control Language, or JCL. The purpose of this section is to provide an overview of basic JCL used by application developers.

JCL is how we communicate with the mainframe telling it what we want to do. In order to understand the necessity for such a language, we must first cover some basic concepts.

The computer contains a series of programs called the operating system. The operating system executes computer programs, controls system resources, and manages data. The computer system's resources are the CPU (Central Processing Unit), main memory, and all peripheral equipment. Peripheral equipment includes all input, output, and storage devices like disk drives, printers, and anything attached via network to the mainframe.

The operating system manages all of the data sets on the system. It is important to understand exactly what a data set is because the term is used repeatedly throughout this course. A data set can be thought of as a collection of data records which is treated as one unit. The type of data set that you are probably most familiar with is a collection of records used as input to the COBOL and Assembler programs you have written. In COBOL we often refer to these data sets as files. Any output from a program, whether it is placed on disk or tape, or sent to the printer, is also considered a data set. As a matter of fact, your program itself is also a data set.

A job is a series of one or more programs to be executed. A job step is the unit of work associated with one program or procedure. A procedure is itself another series of one or more programs. To understand the concept of a job, consider the following example.

After writing a computer program, in order to execute the program a job must be run which involves two steps. The purpose of the first step is to convert the input program source code (called the source module) into machine language (machine code). The machine code is the only language recognized by the computer. The machine code output data set is called an object module. This step in the job is referred to as the compile step.

The purpose of the second step of the job is to obtain the addresses of any external subprograms called by the input program and any other external modules needed to execute the program "source code". Once these addresses are included with the program object code, the resulting data set, called a program object (or load module if using older technology), is ready to be executed. The program object will be saved as a member of a partitioned data set extended on disk (BINDER program) and executed in a later step or even in a later job (program fetch).

The following is a visual concept of a job that compiles and executes a program.

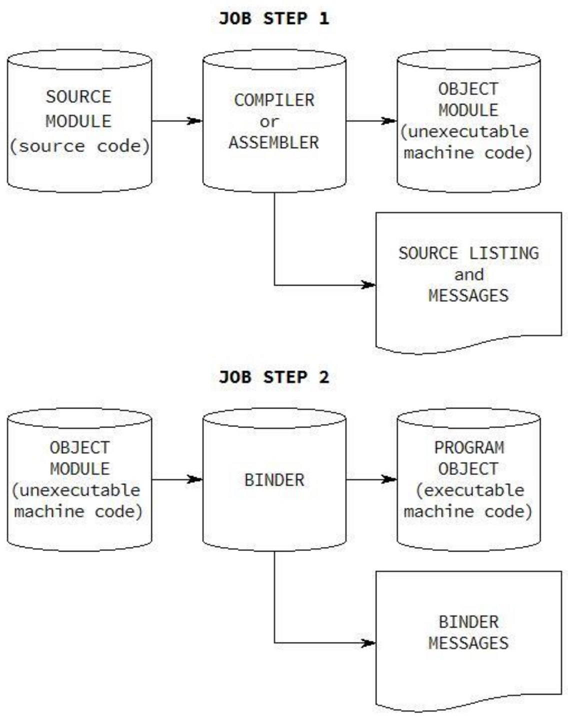
JOB STEP 1
SOURCE MODULE (source code)
COMPILER or ASSEMBLER
OBJECT MODULE (unexecutable machine code)
SOURCE LISTING and MESSAGES
JOB STEP 2
OBJECT MODULE (unexecutable machine code)
BINDER
PROGRAM OBJECT (executable machine code)
BINDER MESSAGES

Figure 2.1 Illustration of a two-step job

In order for the operating system to execute a job, it needs a little information. We must have some way of communicating-with the operating system, hence JCL. The information required by the operating system is divided into three categories:

1. Information about an entire job.
2. Information about a particular job step.
3. Information about a data set used within a job step.

Before the operating system runs a job it needs information to schedule it, such as how much CPU time and how much memory it will need, which processor will be used, what the job's priority is, and who the user is. The JCL statement that supplies this information is the `JOB` statement.

Each program or procedure to be executed is initiated through an `EXEC` statement. Each `EXEC` statement is considered the beginning of a new step. The program(s) or procedure(s) will be executed in the order in which the `EXEC` statements are listed. The operating system reads the JCL stream in top-down fashion, step by step.

Within each step, `DD` statements are used to identify each and any data sets that are required by the program or procedure being executed in that step. The `DD` statements provide various information about the data sets, such as the data set name, whether it exists or will be created, the device and volume name for the data set, and, for a new data set, how much space is needed.

The JCL `JOB` stream begins with the `JOB` statement. The end of the JCL stream is represented by // beginning in column one with nothing on the rest of the line.

A conceptual model of a JCL job stream with two job steps is as follows:

```
JOB   statement   ( the beginning of the job )
EXEC  statement   ( the first job step )
DD    statements  ( the first step's data sets )
EXEC  statement   ( the second job step )
DD    statements  ( the second step's data sets )
//                ( the ending of the job )
```

(This illustration does not reflect actual JCL code.)

It is good to remember that each JCL file ***must*** have a single `JOB` card, or statement, ***must*** have at least one `EXEC` card, or statement, and each `EXEC` card, or statement, ***can*** have one or more `DD` cards, or statements.

It makes sense if you think about it. A JCL file that does not have at least one `EXEC` card does nothing as it executes NO programs!

2.2 Coding JCL

A set, or file, of JCL is called a "job" even though this is a bit of a misnomer. It doesn't really become a "job" until it is submitted for execution on the mainframe and the operating system assigns it a JOB number that might look something like `JOB07495` at Marist.

JCL must "surround" your source code to get it compiled (or assembled), for example, and executed. This JCL was provided to you in CSCI 360 but now you'll have to learn to write your own.

As stated above in 2.1, JCL supplies the operating system with critical information about:

1. the entire job
2. a specific step of the entire job
3. a data set used within a job step

As you will see, JCL is made up almost entirely of three types of statements. The thing that makes JCL hard to learn is the complexity and variety with which each of these statements can be used.

The three types of JCL statements:

1. `JOB` statement
2. `EXEC` statement
3. `DD` statement

JCL statements, or "cards", must be 80 bytes in length but you may use only the first 72 columns.

General Format of JCL Statements:

```
identifier [name] operation [parameters] [comments]
```

Explanation

identifier

Starts in column 1 and generally consists of two slashes (`//`). There are two exceptions: 1) a delimiter statement (`/*`) or 2) a comment (`//*`).

name

One to eight alphanumeric or national (`$,#,@`) characters, starting with a letter or national character. Must begin in column 3 if coded.

operation

A valid operation code, such as `JOB`, `EXEC` or `DD`. Must be preceded and followed by a space.

parameters

One or more parameters, depending on the operation. Individual parameters are separated from one another by commas, with no intervening spaces.

comments

Comments may follow the parameters, preceded by one space and not extending beyond column 71.

To continue a line of JCL, break where a comma is placed, put two forward slashes on the second line of the statement and code the continuation in columns 4 through 16, inclusive.

Now let's look at each of the three statements and how we should use them for the purposes of completing our class work on the Marist machine.

1) `JOB` Statement

Format:

```
//jobname JOB ,'your name',parameters
```

```
//jobname JOB ,yourname,parameters
```

- **jobname**

  The name of the job assigned by the programmer. 1 to 8 alphanumeric and/or national characters ($, @, #) and must NOT begin with a number.

  At Marist, the jobname should be your KC-ID and end with a capital letter A-Z.

- **'your name' or yourname**

  1 to 20 characters. This helps identify your job without having to look up the KC-ID.

- **parameters**

  Specify traits of the entire job separated by commas. The order does not matter as these are keyword parameters.

More About the Optional Parameters:

- `MSGCLASS=`

  Specifies what is to be done with the job output upon job completion of the job.

  Possible message classes at Marist:

  `H` - Hold the job in the queue (in SDSF) until purged by the user.
  `A` - Normal output queue and deleted from the system after an hour.
  `5` or `9` - Used to delete output immediately upon job completion.

- `TIME=(m,s)`

  Specifies the amount of time the entire job will need.

  `m` - number of minutes (integer)
  `s` - number of seconds (integer)

  `JOB` default at Marist: `TIME=(0,20) or TIME=(,20)`

  Note: Sometimes a job runs a very short time beyond that specified or the default time.

- `REGION=nK` or `nM`

  Specifies the maximum amount of memory for the entire job.

  `nK` – Kilobytes, a multiple of 4, in range $1 \le n \le 2097128$.

  `nM` – Megabytes within range of $1 \le n \le 2047$.

  `JOB` default at Marist: `REGION=1024K`

  Note: Sometimes a job can use a bit more space than specified or the default.

- `MSGLEVEL=(stmt,msg)`

  Controls which JCL statements and system messages are displayed by the job.

  `stmt` – A single digit that specifies which JCL statemetns should be printed.

  `0` – Print only the JOB statement.
  `1` – Print only JCL statements, including those that come from procedures (default)
  `2` – Print only JCL statements submitted through the input stream; exclude procedure JCL.

  msg – A single digit that specifies which system messages should be printed.

  `0` – Print step completion messages only; don't print allocation and deallocation messages unless the job jails.
  `1` – Print all messages (default).

- `NOTIFY=`

  Specifies the user-id of the TSO user to be notified when the job completes.

  At Marist, this would be the user's KC-ID.

- `TYPRUN=SCAN`

  Used to check your JCL for syntax errors without running it.

Examples of `JOB` Cards:

```
//KC0nnnnA JOB ,'ABE LINCOLN',MSGCLASS=H,MSGLEVEL=(0,0)

//KC0nnnnA JOB ,GEO.WASHINGTON,REGION=2048K,MSGCLASS=H

//KC0nnnnA JOB ,'ADAMS',TIME=(0,8),REGION=2048K,MSGCLASS=H
```

Although rarely necessary because hardly anyone prints output any more, to print multiple copies at Marist (3 in the example `JOB` card below):

```
//KC0nnnnA JOB ,'SUSAN B ANTHONY',(,,,,,,3),MSGCLASS=H
```

2) `EXEC` Statement

Used to specify which program or procedure to execute.

Each of these are called a "step" in the job. You can have up to 255 steps in a single job. Most will have between one and 5 or 10.

Format:

```
//stepname EXEC PGM=prog-name,parameters

//stepname EXEC PROC=proc-name,parameters

//stepname EXEC proc-name,parameters
```

stepname

Name assigned to a specific step of the job. 1 to 8 alphanumeric and/or national characters ($, @, #) and cannot begin with a digit.

`PGM=prog-name`

Specifies the name of a program (actually, the name of a program’s program object or load module) to execute.

`proc-name or PROC=proc-name`

Name of a cataloged procedure to execute. This is the default format.

Note: Although we will not use cataloged procedures this semester, you need to know how to identify when one is being used or referred to.

Cataloged procedures are very commonly used in the business world.

In fact, IBM has cataloged procedures for various compilations/assemblies and binding steps.

parameters

Specify traits of the specific step. These are separated by commas.

Parameters:

`TIME=(m,s)`

Specifies the amount of time the step will need.

`m` - number of minutes (integer)
`s` - number of seconds (integer)

Step default at Marist: `TIME=(0,5)` or `TIME=(,5)`

Note: Sometimes a job step runs a bit beyond that specified or the default time.

`REGION=nK` or `nM`

Specifies the maximum amount of memory for the step.

`nK` – Kilobytes, multiple of 4, in range of 1 ≤ `n` ≤ 2097128.

`nM` – Megabytes within range of 1 ≤ `n` ≤ 2047.

Step default at Marist: `REGION=128K`

Note: Sometimes uses a bit more space than specified or the default.

If `REGION` is specified on both `JOB` and `EXEC` statements, then the `REGION` on the `JOB` statement has precedence and that on the step is ignored.

```
COND=(n,operator)
     (n,operator,stepname)
     (n,operator,stepname1,n,operator,stepname2,...)
     ONLY
     EVEN
```

Used to conditionally execute a job step.

n

The return code is used ***in comparison*** to that/those from previous step(s). The possibilities are:

0	The job step ran successfully.
4	The job step completed with a warning.
8	The job step issued an error (probable ABEND).
12	The job step issued a serious error (ABEND).
16	The job step issued a severe error (ABEND).

operator

Conditional operator to be used ***in comparison*** to that/those from previous step(s).

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

stepname

Name of the step whose return code is to be used in the comparison (if stepname is not used, the comparison is made to all preceding return codes from all the preceding steps).

`ONLY` The step is executed `ONLY` if a previous step has ABENDed.

`EVEN` The step is executed `EVEN` if a previous step has ABENDed.

Although it goes a bit against our programmer brains, if the COND condition evaluates to TRUE, then the step is NOT executed.

Examples:

```
COND=(8,LT)
```

If 8 is LESS THAN the return code from any one previous step, do not execute this step.

If the return code is 12 or 16, the condition is TRUE and the step does NOT execute.

If the return code is 0, 4, or 8, condition is FALSE and the step executes.

```
COND=(4,NE,JSTEP02)
```

If 4 is NOT EQUAL TO the return code from JSTEP02, then do not execute this step.

If the return code is 0, 8, 12 or 16, condition is TRUE and the step does NOT execute.

If the return code is 4, condition is FALSE and the step executes.

3) `DD` statement

Used to define a data set that will be used in a step of JCL. Note that there is usually more than a single `DD` card within a single job step.

Format:

```
//ddname  DD DSN=data-set-name,parameters
```

This format is used to specify a file to be used.

ddname

Used to link a `DD` statement with a specific file, or data set, that is used within program specified on the `EXEC` statement.

`DSN`

Specifies the name of the data set being referenced by the DD card.

Format:

```
DSN=data-set-name
```

The name of the data set as it is known by or will be known the system. It is made up of groups of 1-8 characters ("nodes") that are joined together by periods (maximum of 44 characters).

All data-set-names begin with your KC-ID at Marist. Also, please use exactly three nodes in every data-set-name you create at Marist.

Examples:

```
KC03B11.FILE1
KC03B12.ACCTNG.PAYROLL
KC03B13.DATA.PROCESS.DEPT.EMPLOYEE.DATA
```

The data-set-name may be a temporary data set that is created during the job and, unless deleted before, is deleted at the end of the job. A temporary data set name begins with two ampersands (&&) and is followed by 1 to 8 characters. The following are examples:

```
&&OBJMOD1
&&TEMP
&&TEMP3
```

Note that there is only one "node" in the temporary data set name.

parameters

Specify traits of the dataset separated by commas.

`SPACE`

Specifies the amount of space on the storage unit required for the data set being created, or allocated.

Format:

```
SPACE=(unit,allocation)
      (unit,allocation,RLSE)
```

```
    (unit,allocation,,CONTIG)
    (unit,allocation,RLSE,CONTIG)
```

unit

Specifies the unit, or measurement, of storage to be used in the allocation.

`TRK`	The data set is to be allocated in tracks.
`CYL`	The data set is to be allocated in cylinders.
`n`	The data set is to be allocated in blocks of n bytes.

allocation

Specifies the amount of storage to be allocated in the format `(p,s)`.

`p`	Primary, or initial, allocation provided once.
`s`	Secondary allocation provided as needed up to 15 times.

`RLSE` and `CONTIG`

`RLSE`	Specifies that any unused units of storage are to be released when the step ends.
`CONTIG`	Specifies that contiguous units of storage for the primary allocation are requested.

Both optional.

Example 1:

```
SPACE=(TRK,(7,3))
```

Start with 7 tracks and add 3 more tracks at a time as needed up to 15 times to a maximum of 52 tracks overall.

```
7 + 3 * 15 = 7 + 45 = 52
```

Example 2:

```
SPACE=(1024,(5,10),RLSE)
```

Start with 5 * 1024 bytes and, if needed, add 10 * 1024 bytes up to 15 times to a maximum of 158,720 bytes overall. This is unique, though, due to the `RLSE`. Specifying a `SPACE` parameter like this means that the data set is being allocated and written to all in the same job step. When the job step ends, any unused 1024 byte blocks, or units, will be released. In other words, this data set can have no more records written to it at a later time.

`DISP`

Specifies the disposition of the data set being referred to by the DD card.

Format:

`DISP=(status,normal,abnormal)`

status

Specifies the ***current*** status of a data set ***at the beginning of the job step.***

`NEW` The data set is to be created (default).

`OLD` The data set exists and is to be "locked" for access by the program running in the current job step.

`SHR` The data set exists and this and any other programs may access it concurrently.

`MOD` The data set can have records added to it. Also, if the data set does not exist, it will be created.

normal

Specifies what to do with the data set if the program, or job step, terminates normally and without an error.

`KEEP` The data set is to be kept at the end of the job step. Marist automatically catalogs it by default.

`PASS` The data set is to be passed to a later step within the job. **Must** be used for temporary data sets.

`DELETE` The data set is to be deleted at the end of the job step.

`CATLG` The data set is kept ***and*** cataloged at the end of the job step. At Marist, this is the default.

`UNCATLG` The data set is kept but uncataloged at the end of the job step. At Marist, this is the default.

Note: There is no need to ever use `CATLG` or `UNCATLG` at Marist as a "permanent" data set is automatically cataloged when allocated and automatically uncataloged when deleted.

abnormal

Specifies what to do with the data set if the job step terminates abnormally, or ABENDs.

`KEEP` Same as above.

`DELETE` Same as above.

`CATLG` Same as above.

`UNCATLG` Same as above.

Examples:

`DISP=(NEW,KEEP)` equivalent to `DISP=(NEW,KEEP,KEEP)`

`DISP=NEW` equivalent to `DISP=(NEW,DELETE,DELETE)`
`DISP=OLD` equivalent to `DISP=(OLD,KEEP,KEEP)`

Note that `DISP=OLD` locks the data set for access and no other user's program can access it until the job step ends.

`DISP=(,KEEP)` equivalent to `DISP=(NEW,KEEP,KEEP)`

Note: Can you see a bit of a pattern developing here?

DCB

Stands for **data control block** and specifies information about the data set that is being allocated. A `DCB` is necessary when allocating a data set where information about the data set's record length, the block size and record format – and possibly other necessary characteristics – cannot be determined otherwise.

Format:

```
DCB=(LRECL=l,BLKSIZE=m,RECFM=n)
```

`l`	The logical record length in bytes, commonly 80 if using data set for storing source code and JCL.
`m`	The size of a block, or physical record, in bytes. This must be a multiple of `LRECL`.
`n`	Most commonly this is `FB` for fixed, blocked which indicates whether records are fixed length or not.

Sometimes we will see `DSORG`, or data set organization, added to the `DCB` parameters. If `DSORG=PS`, the data set is physical sequential which means that it is a sequential file as opposed to a partitioned data set (PDS) or partitioned data set extended (PDSE).

If `DSORG=PO`, it is a PDS or PDSE. Of course, if `DSORG=PO` and there is a third integer in the inner parentheses in the `SPACE` parameter indicating the number of directory blocks to allocate, `DSORG=PO` is redundant.

Other Types of DD Cards:

```
//ddname  DD SYSOUT=class-name
```

Used to route data to an output device. It is commonly `SYSOUT=*`

```
//ddname  DD *
```
...instream data goes here
```
/*
```

Used for instream data (data not from a file that is embedded in the JCL).

The `/*` signals the end of the data.

```
//ddname   DD DATA
```

...instream data goes here
```
/*
```

Used for instream data that might contain `//` in columns 1 and 2. The `/*` signals the end of the data.

```
//ddname   DD DATA,DLM='xx'
```
...instream data goes here
```
xx
```

Used for instream data that might contain either `//` or `/*` in columns 1 and 2. `xx` can be any two characters.

VOL=SER

Specifies volume serial number where the data set is located. Each DASD storage unit, or pack, at an installation is given a unique identifier known as its "`VOL SER`" or volume number.

Format:

```
VOL=SER=volume-number
```

volume-number The unique identifier of the DASD volume, such as KCTR23 at Marist.

Because Marist uses SMS (Storage Management Subsystem), all data sets are cataloged when allocated and VOL=SER is not necessary. Below is IBM's product description:

The Storage Management Subsystem (SMS) is a DFSMS facility designed for automating and centralizing storage management. Using SMS, you can describe data allocation characteristics, performance and availability goals, backup and retention requirements, and storage requirements to the systems.

SMS improves storage space use, allows central control of external storage, and enables you to manage storage growth more efficiently. With SMS, you can easily manage conversion between device types and ultimately move toward system-managed storage.

2.3 Data Sets

Data sets are made up of one or more records, sometimes referred to as logical records ("`LRECLs`").

Records are stored in blocks which are often called physical records. There can be one record in a block but this is the least efficient due to inter-block gaps. This would lead to more inter-block gaps, i.e., one between each record!

A data set's blocking factor is determined by dividing the block size (in bytes) by the size of a single record (in bytes).

blocking factor = block size / record length

For example, if a data set's block size is 880 bytes and each of its records is 80 bytes in length, the blocking factor would be 11.

Remember that a block, or physical record, is made up of records, or logical records ("LRECLs").

Non-VSAM Data Set Types:

Sequential Data Sets, or "Flat Files"

A data set with one or more records that can most commonly be read from beginning record to end. The records are probably very closely related in some way, i.e., a file of Visa check card transactions.

Partitioned Data Sets, PDSs, or Partitioned Data Sets Extended (“Libraries”)

Instead of one single sequential file, a partitioned data set, or PDS, or partitioned data set extended, or PDSE, is subdivided into one or more members that are each essentially sequential files. Of course, each of the PDS or PDSE members should have some relation to one another. A good example is a source library that has members containing source code for COBOL programs.

A DD card that references a PDS or PDSE is almost exactly like one that references a sequential, or flat, file.

The differences:

When allocating a PDS or PDSE, you have to specify how many 256-byte directory blocks to allocate. An example follows:

```
SPACE=(TRK,(10,20,5))
```

This allocates space exactly as we have seen above except the third position within the inner set of parentheses indicates the number of 256-byte directory blocks to allocate. CONTIG and RLSE can be used with a PDS or PDSE just as they can with sequential files.

A directory block provides information about each of the members stored in the PDS or PDSE. It is a lot like a phone directory where a member's name is followed by the location within the PDS or PDSE where it begins and information about how many bytes the member has.

2.4 Example Job

Please note that the following JCL example is NOT properly documented!

This is an example job that is ready to run on the Marist mainframe.

```
 1.  //KC0nnnnA JOB ,'JCL EXAMPLE',MSGCLASS=H
     //*
     //**********************************************************
     //*                                                        *
     //*  JSTEP01 - HIGH-LEVEL ASSEMBLER (ASMA90)               *
     //*                                                        *
     //**********************************************************
     //*
 2.  //JSTEP01  EXEC PGM=ASMA90,PARM=ASA
     //*
 3.  //SYSLIB   DD DSN=SYS1.MACLIB,DISP=SHR
     //*
 4.  //SYSIN    DD *
        (Assembler code goes here)
 5.  /*
     //*
 6.  //SYSLIN   DD DSN=&&OBJMOD,SPACE=(3040,(40,40),,,ROUND),
 7.  //            DISP=(MOD,PASS)
     //*
 8.  //SYSUT1   DD SPACE=(16384,(120,120),,,ROUND)
     //*
 9.  //SYSPRINT DD SYSOUT=*
     //*
     //**********************************************************
     //*                                                        *
     //*  JSTEP02 - BINDER (HEWL)                               *
     //*                                                        *
     //**********************************************************
     //*
10.  //JSTEP02  EXEC PGM=HEWL,COND=(0,LT)
     //*
11.  //SYSLIB   DD DSN=CEE.SCEELKED,DISP=SHR
     //*
12.  //SYSLIN   DD DSN=&&OBJMOD,DISP=(OLD,DELETE)
     //*
13.  //SYSLMOD  DD DSN=KC0nnnn.CSCI465.LOADLIB(BITMANIP),
14.  //            SPACE=(1024,(50,20,1)),
15.  //            DSNTYPE=LIBRARY,
16.  //            DISP=(MOD,KEEP,KEEP)
     //*
17.  //SYSPRINT DD SYSOUT=*
     //*
     //**********************************************************
     //*                                                        *
     //*  JSTEP03 - PROGRAM FETCH STEP                          *
     //*                                                          *
     //*************************************************************
     //*
18.  //JSTEP03  EXEC PGM=BITMANIP,COND=(0,LT)
     //*
19.  //STEPLIB   DD DSN=KC0nnnn.CSCI465.LOADLIB,DISP=SHR
     //*
20.  //INDATA    DD DSN=KC01234.CSCI465.DATA(ENCODED1),
21.  //             DISP=SHR
     //*
22.  //PRINTOUT  DD SYSOUT=*
     //*
23.  //SYSUDUMP  DD SYSOUT=*
24.  //
```

Note that, in the above example, the documentation lines, each beginning with `//*` are not numbered as they require no explanation. Also, in the first days of computers, every line of a program and every line of JCL was saved on an 80-byte Hollerith card. The name has stuck with JCL with lines often called "cards".

Explanation:

1. The `JOB` card. The A at the end of the user's KC-ID can actually be any letter at Marist but use A.
2. `JSTEP01` is an `EXEC` card executing the high-level Assembler. Marist asks that the `PARM=ASA` be used. `ASA` is an abbreviation for American Standards Association. The `PARM=ASA` tells the Assembler that programs written by NIU students on the Marist mainframe are using `ASA` carriage control characters in the output produced so that downloading using mar_ftp.exe or another ftp program maintains line spacing designated in both Assembler and COBOL programs.
3. `SYSLIB DD` card. This data set is the standard IBM macro library.
4. `SYSIN DD *` indicates that instream data follows. The `SYSIN` card indicates the Assembler's input or the Assembler program source code. It works the same for the COBOL Compiler too.
5. The `/*` indicates the end of instream data.
6. The `SYSLIN DD` card indicates the name of data set into which the Assembler (or the COBOL Compiler) will write the program's object module. An object module is only an intermediary module and is NOT executable. This is usually a named temporary data set that is passed to a Binder step later in the job. The temporary data set in this example is named `&&OBJMOD`. Note the `DIS`Position values `(MOD,PASS)`. `PASS` is required indicating that, if the step is successful, the temporary data set named `&&OBJMOD` is to be `PASS`ed to a later step where it will be used in some manner.
7. This line is a continuation of the `SYSLIN DD` card above. Note the comma at the end of the line numbered 7. The continuation must not happen in the middle of a keyword parameter and the continuation must begin on the next line between columns 4 and 16, inclusive.
8. `SYSUT1 DD` card indicates a temporary data set used by the Assembler as a sort of "scratch pad" while creating the program's object module. This temporary data set will only be used in this step so does not require a name. Only the keyword parameter `SPACE` is necessary.
9. `SYSPRINT DD` card with `SYSOUT=*` indicates where the Assembler's (or COBOL Compiler's) output messages and source listing are to be written. `SYSOUT=*` indicates that the output will be spooled to standard output.
10. `JSTEP02` is an `EXEC` card executing the Binder. The `COND=(0,LT)` indicates that, if any previous job step (there is only one in this example, the Assembler step) has a return code greater than 0, do not execute this step. In other words, only a 100% clean assembly will be allowed through. Here is a breakdown of the evaluation of the `COND=` statement:

0 LT 0? False, **will** execute
0 LT 4? True, **will not** execute
0 LT 8? True, **will not** execute
0 LT 12? True, **will not** execute
0 LT 16? True, **will not** execute

11. `SYSLIB DD` card specifying the name of libraries containing any system load or object modules referenced by the `SYSLIN` object module(s).
12. `SYSLIN DD` card specifying the name of the object module(s) to become part of the program module. These will be linked together by the Binder, hence the name.
13. `SYSLMOD DD` the name of the load library where the program module that is created in this step will be stored "permanently". It can now be utilized at any point in the future by any other job as necessary.
14. The `SPACE=` card is included here in the case that the load library does not exist and thus needs to be allocated.
15. The `DSNTYPE=` card is included here in the case that the load library does not exist and thus needs to be allocated. It should be allocated as a PDSE which is exactly what `DSNTYPE=LIBRARY` indicates.
16. This line is a continuation of the `SYSLMOD DD` card above. Because the Binder will be adding a new program module (or replacing one previously created) to an existing and probably important – perhaps critical – load library, be sure to code the `DISP=` keyword parameter appropriately. If it is coded `DISP=MOD`, the entire library will be deleted if this one step fails(!). It is best to code it as `DISP=(MOD,KEEP,KEEP)` just to be sure.
17. `SYSPRINT DD` card with `SYSOUT=*` indicates where the Binder's output messages are to be written. Once again, `SYSOUT=*` indicates that the output will be spooled to standard output.
18. `JSTEP03` is known as a fetch step. It is an `EXEC` card executing a program module. In this case, it is the one created in the immediately previous step named `BITMANIP`. The `COND=(0,LT)` here has the same meaning as in `JSTEP02` but, in this case, execution will depend on the return codes issued by the previous **two** job steps. Fetch steps are most often done at a future time but, if the program module just created needs to be tested, a fetch can follow the Binder step as it does here. By the way, it is important to remember that the `CSECT` name in Assembler or `PROGRAM-ID` in COBOL must match the name of the program module itself. In this example, it is `BITMANIP`.
19. `STEPLIB DD` indicates the name of the load library from which the program module will be fetched. Note that the name of the load library member, `BITMANIP` in this example, is not indicated on the `STEPLIB DD`; it is already indicated on the `EXEC` card.
20. `INDATA DD` card indicates one of the data sets required by the program that is being fetched and executed. On this `DD` card in this example, it is the member named `ENCODED1` from the PDS or PDSE named `KC01234.CSCI465.DATA`. As will be illustrated later in Assembler QSAM and COBOL, the name INDATA is important so that the program can read data from or write to the data set(s) referred to by the `DD` card.
21. `DISP=SHR` is a continuation of the previous `DD` card.
22. `PRINTOUT DD` is another data set required by the program that is being fetched and executed. This one is obviously an output `DD` as whatever is being written by the program to this `DD` is being written to standard output.
23. `SYSUDUMP DD` card is used by the system to write a full dump if the program forces a dump by executing the ABEND macro or other or if an ABEND accidentally occurs. `SYSOUT=*` indicates the output is to be written to standard output.
24. `//` indicates the last card of a job. Be sure that this sequence of symbols (`//`) does not end up on a line of JCL with nothing following it unless it is indeed the last card of the job. If one is found in the middle of JCL, the JCL reader will discard the remainder of the job. This can sometimes cause a very confusing situation.