JCL 101: Writing your First Job

Job control language, also known as JCL, is a foundational piece of the puzzle in the world of mainframes. While the word ‘language’ may be in the name, JCL is quite different from what you would think of other programming languages in the traditional sense. This is because JCL does not support structures such as loops and variables, at least not how you would see them in other programming languages. This may lead one to wonder how a language like JCL can be so important and foundational for mainframes. And the answer to this is that JCL is used more in a script-like sense on IBM mainframes, for the purpose of communicating with the operating system, so it knows exactly what to do. An easy way to think of it is this: in JCL, you’re not writing a program to be executed, but rather, you’re writing a job, which contains all necessary information needed by the IBM mainframe operating system, to successfully execute another program, which could be HLASM, COBOL, Python, C++, or any other language that can be executed in a mainframe environment. This lesson is meant to serve as an introduction to JCL, as it can be difficult to work with and appear very complicated, especially when you are being introduced to it. In other words, this is not a deep dive into JCL, and if that is what you are seeking, then take CSCI 465, which will dive much deeper into JCL than this lesson will.

What is a Job?

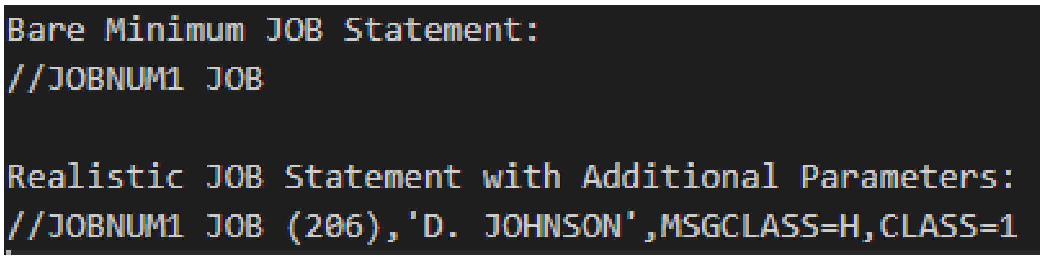
A job is a series of one or more programs and/or procedures to be executed. These programs/procedures become known as job steps within the job. One may wonder, if I just want to run a COBOL program, why do I need a whole job for that? Well, the answer is simple, which is that there a multiple parts that go into executing a COBOL program successfully. First, the COBOL source code would need to be compiled. This is done by executing the COBOL compiler, which is a program, and giving it source code as input. This COBOL compiler will produce what is known as an object module (an unexecutable file), which will then be used as input to another program known as the binder, to turn it into a load module, or in other words, an executable program. Actually executing this program would require a third job step, and all three job steps together would make up what is known as a job. Not all jobs are three steps like this; some could be more or less, but this is an example to illustrate why multiple job steps may be needed.

How is JCL Structured?

The structure of JCL will be the same in every job that is to be executed. Each job will have 3 types of statements, those being JOB statements, EXEC statements, and DD (or Data Definition) statements. Each JCL program will contain exactly 1 JOB statement, but can contain many EXEC statements and DD statements. The easiest way to introduce this structure is that the JOB statement contains EXEC statements within it, and each EXEC statement contains any relevant DD statements within it.

What is a JOB statement?

The JOB statement will always be the first statement in a JCL stream, and its purpose is to identify a job and how to manage it. The only required part of a JOB statement is the name of the job and the keyword ‘JOB’ (each statement must also start with two slashes // as well, but this is for JCL as a whole, not just the JOB statement). Different situations could require additional parts to be added to it. Some other common things seen within a JOB statement are the programmer’s name who is submitting the job, a MSGCLASS parameter (which tells the system where output should go), a CLASS parameter (used to group jobs, which helps the system manage resources), and job accounting information. However, these secondary commonalities within JOB statements are usually specific to whatever company you are working for, rather than being something required for the operating system. An example JOB statement will be shown below:



```
Bare Minimum JOB Statement:
//JOBNUM1 JOB

Realistic JOB Statement with Additional Parameters:
//JOBNUM1 JOB (206),'D. JOHNSON',MSGCLASS=H,CLASS=1
```

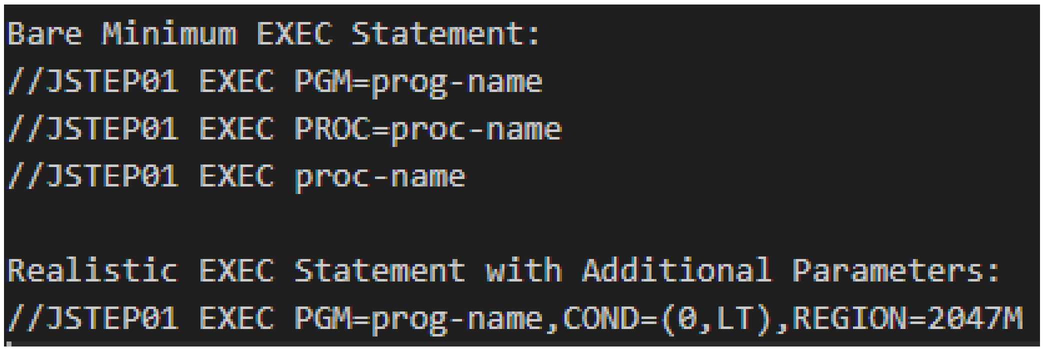
In the image above, JOBNUM1 refers to the job name, and JOB is a keyword. These are needed in every job statement. The additional parameter (206) is the accounting number, ‘D. JOHNSON’ is the programmer's name, MSGCLASS=H specifies which output to send to, and CLASS=1 specifies which class to group the job in. These secondary parameters all hold example values, as the values for each should differ between companies. For example, MSGCLASS=H may mean to send output to standard output for one company, but it could be MSGCLASS=Y for another company.

Programs vs Procedures

Before discussing EXEC statements, we need to understand what programs and procedures are, as they are often used interchangeably in EXEC statements, but are a little different. The term program refers to an executable load module that exists within a system library or what are known as load libraries. The term ‘load module’ simply means a copy of source code that has been successfully compiled and linked, and is simply awaiting a call to be executed. A procedure, on the other hand, is a reusable set of JCL statements that contains one or more steps to perform a task, rather than compiled source code. Procedures are typically stored in what are known as procedure libraries. To put it simply, Programs are compiled source code, and procedures are predefined blocks of JCL job steps to be called.

What is an EXEC Statement?

An EXEC statement is a statement in JCL that is often referred to as a job step, as within each job, each EXEC statement represents an individual part of the process along the way to completing an execution of a job. In each EXEC statement/job step (these two terms will be used interchangeably), there will always be a program or procedure being executed. Think of it as each EXEC step provides the operating system with the information about which program to perform, in the proper order of how the job should be executed. For example, if you have multiple job steps called JSTEP01, JSTEP02, and JSTEP03, they will be executed in the order they are listed. Each EXEC statement may contain DD statements within it, which provide further information for the EXEC step, but that will be covered in the next section of this lesson. For an EXEC step, the minimum syntax required is the name of the job step, the EXEC keyword, and a program/procedure parameter. Several other common parameters may be used, such as COND= (used as a conditional statement for whether or not to execute a job step) and REGION= (used to override the default amount of storage space allocated for a job step). An example EXEC statement will be shown below:



```
Bare Minimum EXEC Statement:
//JSTEP01 EXEC PGM=prog-name
//JSTEP01 EXEC PROC=proc-name
//JSTEP01 EXEC proc-name

Realistic EXEC Statement with Additional Parameters:
//JSTEP01 EXEC PGM=prog-name,COND=(0,LT),REGION=2047M
```

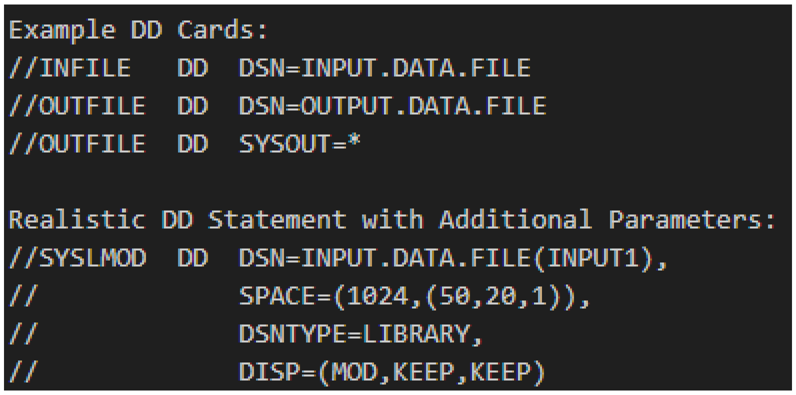
In the image above, you can see that for the "Bare Minimum" section, there are 3 different versions. This is because the syntax for using programs vs procedures can be done in a couple of different ways. If using a program, you must use PGM= followed by the name of the program. If using a procedure, you can either use PROC= followed by the name of the procedure, or simply write the procedure name as shown above. You will also notice the name of the job step, 'JSTEP01', as well as the EXEC keyword that is required for execution. For the additional parameters, the REGION= parameter tells the operating system to give whatever amount of space is needed to a step, which here I defined as 2047 megabytes. You will also notice the COND= statement, which I clarified in the next section, as it can be quite confusing to someone new to JCL to understand.

The COND= Parameter

As someone new to JCL, the COND= parameter can be one of the most complicated things to understand. All you need to know is that each job step returns a return code, depending on the results of its execution. If it executes with no warnings, then the return code will be 0; if there are warnings or errors, it will be either 4, 8, 12, or 16. That specific COND= statement in the example above compares 0 to whatever the return code is, meaning that it will evaluate the statement of (0 < return code). If that statement evaluates to false, the job step is executed. If that statement evaluates to true, then it is skipped. So in this example, the COND= parameter makes it so that the job step will only be executed if all previous job steps execute with no warnings or errors, since to make that statement evaluate to false, it would need to be 0 < 0, as if there were any warnings or errors, the return code would be at least a 4, which would change the statement to 0 < 4, which is true, which in turn will make the operating system ignore that job step.

What is a DD statement?

As mentioned before, DD statements will belong to a specific EXEC statement. To put it simply, it defines all the datasets a program will use to properly execute. Some information DD cards can provide is what file to use as input, what file to send the output to, or simply just send to standard output. The names of these DD statements are critical because they are what you use to connect your program logic to the data set you are attempting to process. This provides easy flexibility for using multiple different input data sets. If you have your input file DD statement titled INFILE, you could simply change the data set associated with it and execute the same program, but with a different data set. There are a couple of parameters needed for a DD statement, such as the name of the DD statement, the keyword 'DD', and any parameters that define the dataset. This can be a lot of varying things, but what is required is either a DSN= parameter, which is followed by the name of a data set, an asterisk or 'DATA' keyword to signify inline data, or a SYSOUT= parameter, which signifies to output to standard output (There are a couple more as well, but these are the only essential ones for a beginner to know). Examples of what DD statements could look like are shown below:


```
Example DD Cards:
//INFILE   DD  DSN=INPUT.DATA.FILE
//OUTFILE  DD  DSN=OUTPUT.DATA.FILE
//OUTFILE  DD  SYSOUT=*

Realistic DD Statement with Additional Parameters:
//SYSLMOD  DD  DSN=INPUT.DATA.FILE(INPUT1),
//             SPACE=(1024,(50,20,1)),
//             DSNTYPE=LIBRARY,
//             DISP=(MOD,KEEP,KEEP)
```

In the image above, you can see that the simpler DD statements simply list a data set. This is effective in situations when executing a program and using a DD statement to signify which data set to use as input or output. The second OUTFILE DD statement signifies what sending data to standard output looks like. The “Realistic DD statement” example is what a DD statement could look like in another situation. The SPACE=, DSNTYPE=, and DISP= parameters share their own significance, but for this lesson, I will not overcomplicate and discuss more specific details like those. If you are curious and want to dive further into what more parameters there are and dive deeper into JCL, take CSCI 465, as this class dives much deeper into JCL than this lesson does.

What makes DD statements so important in JCL?

The separation of the program code and the data definitions that go into it is one of the main strengths of JCL, and a key reason that mainframes have been able to flourish for over 60 years. This allows the same compiled program to be used in many different contexts, simply by updating the DD statements within the JCL, to use the same program to process varying data. As an example of this, think of a program like a payroll processor. You can write the program initially and compile it once, and then run it repeatedly over any number of data sets, as long as the data sets follow the expected structure of the program logic, by only changing the data sets in the DD statements.

This fundamental structure of JCL provides flexibility that enables high-volume batch processing, that have enabled mainframes to remain significant to many institutions all over the world. Companies can write programs once and reuse these programs repeatedly, by using the same logic, just with different data sets, with no need to even compile the source code again.

Useful Links:

- IBM JCL documentation: https://www.ibm.com/docs/en/zos-basic-skills?topic=collection-basic-jcl-concepts
- Final Note: I mentioned in the document, but if you are interested in learning more about JCL on a deeper level than discussed in this lesson, consider taking CSCI 465! This class will give a much deeper dive into JCL, and will give a great opportunity for learning to write your own JCL as well.