## The Stored Program Concept

## John von Neumann, a Hungarian-American mathematician, is considered the father of the modern general-purpose computer. His work from 1946 - 1951 established the basic relationship between the central processing unit (CPU) and main memory and formed the basis for the “stored-program” concept. Programs, along with a program’s data, are stored in memory in order for them to be executed. The fundamental principles of the Von Neumann architecture are:

## Memory should hold both computer code (programs) and data

## Each memory location has a unique numeric address

## Memory is addressed linearly i.e. addresses are numbered in sequential order

## Memory is addressed without regard for the contents stored at that address

## The von Neumann architecture makes it easy to temporarily store and execute programs in memory as needed and make the computer system general purpose in nature. Once programs receive computer hardware resources, like memory or CPU, they become processes. Computer processes are also referred to as workloads. There are three basic workloads: batch, online and real-time. Where real-time being a variation of an online workload. We’ll be exploring these workload types in future units.

**Program Preparation**

Before programs can be executed they need two very important things: 1) programs need to be prepared for execution and 2) they need to reside in memory so that the CPU has a place from which to fetch the program’s instructions. To prepare a program so that it is able to be executed it needs to be translated from “human readable” form called **source code** to a form that the computer recognizes, called **object code.** Object code is a preliminary form that the computer can’t really use because the computer doesn’t know where in memory the program will be placed so the object code needs to be made **re-locatable.** Once re-locatable, the program ready to run. Let’s take a look at each of the steps in more detail.

**Source Code -** Humans, us in this class, will write some computer source code. That is, we will be writing small programs that we will want the computer to run for us. Writing computer programs is a pretty cool way of getting the computer to do our work. We will be writing source code using a language called COBOL which is an acronym for **Co**mmon **B**usiness **O**riented **L**anguage. In this lab exercise we will write our COBOL source code by typing it into a file using an **editor**. Source code is only recognizable by humans. In fact, COBOL code is made up of commands that are really English words, making it easy for humans to read and understand. Unfortunately, not all programming languages use English words. So the first step in program preparation is, by using an editor, writing the source code and saving in a file.

**Object Code -** Unfortunately the computer doesn’t speak English, so we need to translate the source code into a machine-readable or intermediate language, called object code. We do this by submitting the source code as input to a compiler. A compiler is a language translator that converts your COBOL instructions to lower-level language that match the instruction set architecture of the target computer that you want your program to run on.

In addition to compilers, there is another kind of translator, called an **interpreter.** Interpreters translate one line of code at a time then execute that line of code. Unlike, COBOL, Java is a language that uses an interpreter. As a general rule, interpreted language implementations are less-efficient than compiled languages. So from an enterprise perspective consider compiled programs as part of your solution. But object code still isn’t ready for execution so we need one more step.

**Load Module -** In this last step we need to resolve all of the programs addresses. The object code is input into a Linkage Editor. The Linkage editor makes the object code re-locatable. This simply means that we want the operating system to have the flexibility to run our program from anywhere in the computer’s memory. So all our program’s instruction addresses can be adjusted based on the actual starting memory address where the program is placed in real memory. You can think of a load module as the functional equivalent to a Windows .EXE file. After the “link edit” step the program is now ready to be executed.

So the program preparation process looks something like this:

**COBOL Source Code:**

ID DIVISION

AUTHOR

etc

.

**COBOL Compiler**

**COBOL Editor**

**COBOL Object**

**Module**

**Linkage Editor**

**COBOL Load Module**

**Learning Objectives**

After this lab exercise the student will be able to:

* Explain the “stored program” concept
* Explain the equivalence and interchangeability of program code and data
* Explain the importance of the instruction set
* Explain the way in which instructions are formatted
* Explain the importance of the von Neumann Architecture
* Explain the steps involved in program preparation
* Compare and contrast compilers and interpreters

Use RDz and ISPF to create, compile and execute a COBOL program

**Lab Exercise Setup**

In this lab you will explore the COBOL language and associate it with Stored Program Concept. Using RDz, sign on to the enterprise system using your z/OS (SUSnnnn) account credentials and verify that you have all of the datasets you need to create, compile and run COBOL programs.

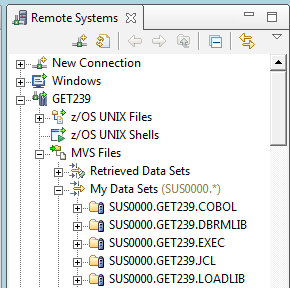
1. Before we can compile, link and run our COBOL programs we need to have places, or libraries, to save our **Source** and **Load** modules and the JCL to execute them. Using RDz look for the following PDS’. Use your SUS ID in place of SUSnnnn.

**SUSnnnn.GET239.COBOL** - for your COBOL source code

**SUSnnnn.GET239.JCL** - for your Job Control Language needed compile and execute your programs

**SUSnnnn.GET239.LOADLIB** - for your executable programs

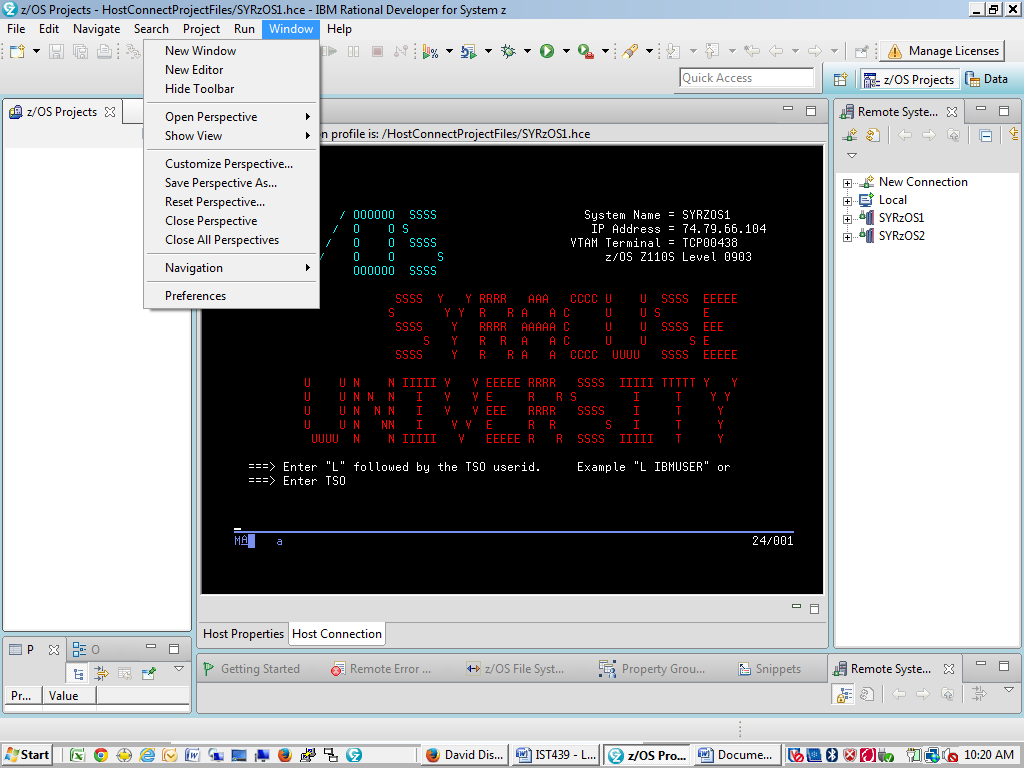
These PDS’ should have been created in a previous lab exercise, but if any of them are missing you can recreate them by going to **SHARE.GET239.JCL** and submitting **CREATEDS**. After you run the **CREATEDS** job verify that all of the PDS’ are now there. Your RDz Remote Systems Explore should look something like this:

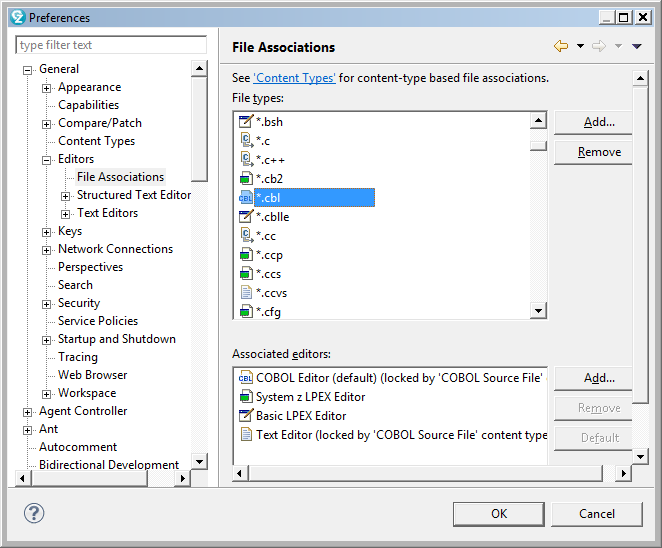


**You should see the following data sets in your account. Here I used SUS0000, your data sets will have your SUS ID as the high-level qualifier**

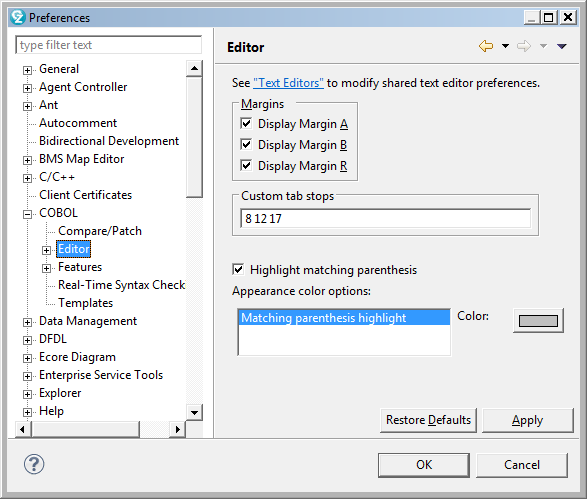
**RDz Smart Editors** – RDz has many different editors for accomplishing the same thing so we are going to set the COBOL editor as your default editor.

Go to the main tool bar 🡪Window 🡪 Preferences:



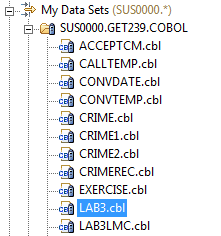
You will see a Preferences dialog box. Select General 🡪File Associations then find and highlight the .cbl file extension. Then under Associated editors click add, this will make the Cobol editor your default editor by selecting the Default option.

While still in Preferences, select COBOL 🡪 Editor then check the 3 Margins boxes, Display Margins A, B and C. These margins will be displayed when you open up the editor and help you format your COBOL code so that you can read and maintain code much easier.



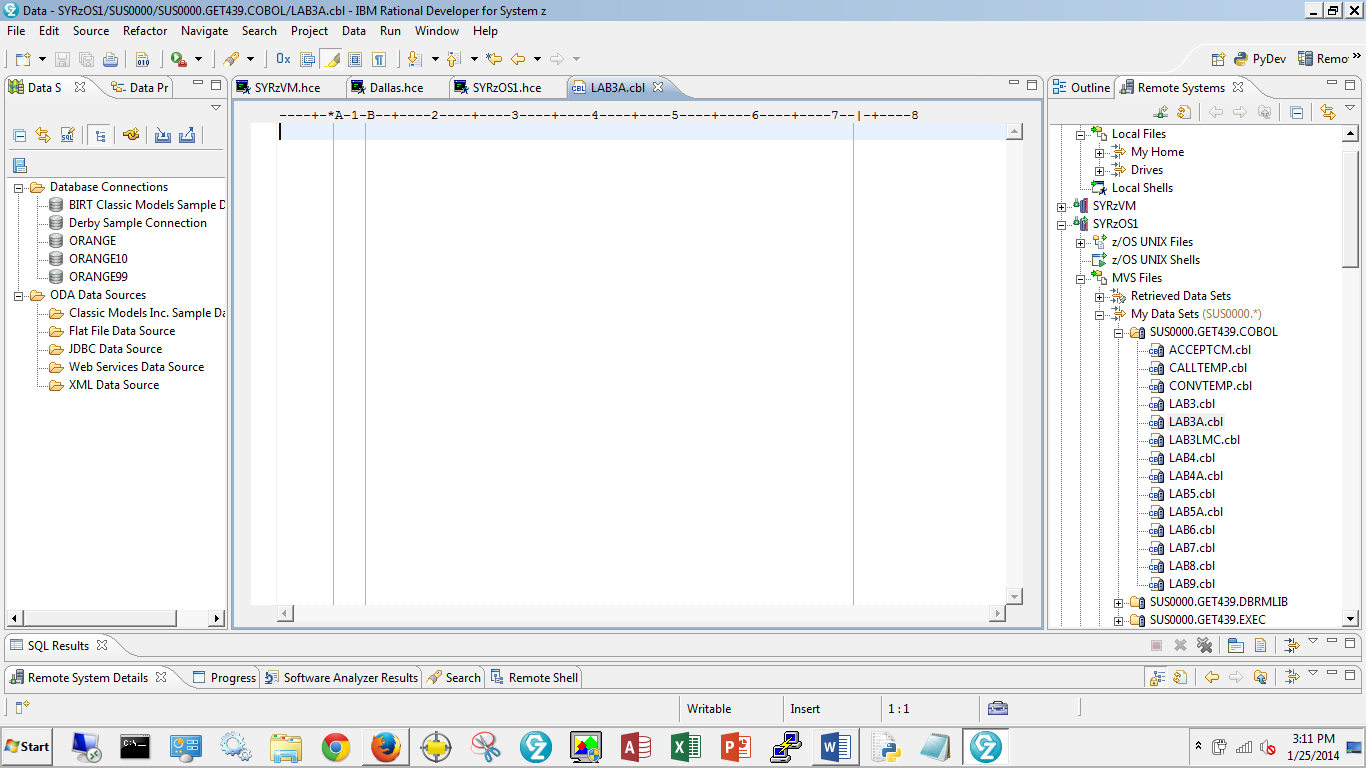
You should now be all set to start writing some COBOL code.

1. We are going to write a very small COBOL program and save in your SUSnnnn.GET239.COBOL library. In RDz, right click on SUSnnnn.GET239.COBOL. From the drop down menu select **New 🡪 Member.** In the dialog box name your new member **Lab3**. Your Remote Systems Explorer should look like this:



**Notice that your new LAB3 member has a .cbl file extension. This allows the RDz to launch the COBOL smart editor**

1. Let’s open the COBOL smart editor. Double click on LAB3. The editor will open. You’ll notice there is nothing in LAB3 but it is here where you will code some COBOL instructions.



R

A

B

**Type the following COBOL commands here.**

**Here you will also see the margins A, B and R**

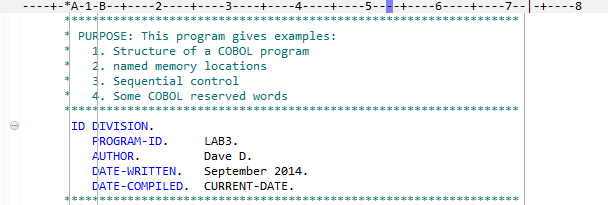
1. Let’s write COBOL. COBOL is a very easy programming language to learn since its instructions are English words. COBOL has a defined structure so once you understand where you place the English words you’ll have no problem building COBOL programs.

The COBOL language is designed to be easy to put the instructions in the correct places. It has four (4) divisions where code can be placed. The division names all start in column 8 in the COBOL editor. Let’s look at the purpose for each **DIVISION.**

**IDENTIFICATION DIVISION -** This division can be abbreviated **ID DIVISION**. It is always first instruction in your program and it is the place where you identify the program’s documentation. It contains the program name, the author, date written and date compiled. You can place comments before the ID DIVION statement if you choose; like I did in the example below.

Type these instructions in to your LAB3 editor view like the example below being mindful of the margins.

**1**



**2**

1. In Box 1 you’ll notice a guide that tells you where (the column) you are typing. The DIVISIONs always start in Margin A or column 8. All other commands start in Margin B or column 12. You’ll notice you can add your own documentation or comments. These start in column 7. I like to wrap my comments with asterisks so that they stand out for the reader to easily see them.

2. In Box 2 you see the actual COBOL code. Here I abbreviated the IDENTIFICATION DIVISION, named my program LAB3. These are the only two required statements. Everything else (AUTHOR, DATE-WRITTEN, DATE-COMPILED are treated as comments. In the COBOL editor all words in blue are COBOL reserved words and can only be used for their intended purpose. Reserved words cannot be used by you for any other purpose.

You’ll also notice that there are vertical gray lines that delineate the margins A, B and R. These are guidelines for allowing you to write code that is easy to read. Also notice that you can create general comments by putting a character in column 7 (here I used an asterisk.) The RDz COBOL editor uses a pretty teal color to identify comments.

**ENVIRONMENT DIVISION** - This division describes the input and output files needed by your program. In this lab we will only use the necessary commands for our program to run. We won’t be using any input or output files for this lab; but will be expanding this division more in future labs.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ENVIRONMENT DIVISION.

\* INPUT-OUTPUT SECTION. <to be used later>

\* FILE-CONTROL. <to be used later>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**DATA DIVISION -** This division is where you can describe your data in detail. Here you can describe input and output records and you can give memory locations real names. As you recall for the von Neumann architecture each memory location has an address. But who can remember memory addresses; especially, in systems that have terabytes of real memory. So for convenience you can give your memory locations meaningful names.

DATA DIVISION.

\* FILE SECTION. <to be used later>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

WORKING-STORAGE SECTION.

\*\*\*\* NAMED MEMORY LOCATIONS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

01 WS-NAMED-MEMORY-LOCATIONS.

05 MAILBOX-1 PIC X(50) VALUE

'This is an example of PIC X character string'.

05 MAILBOX-2 PIC X(50).

\*\*\*\* EXAMPLE OF REPURPOSING THE SAME MEMORY ADDRESSES \*\*\*\*\*\*\*\*\*\*\*

05 MAILBOX-3 PIC 9(5) VALUE 15.

05 MAILBOX-4 REDEFINES MAILBOX-3 PIC XX.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

It is in the **WORKING-STORAGE SECTION** where you can give memory locations names. Each memory location can have a name and some characteristics that constrain the data; that is, metadata about what the user can expect to see in that particular memory location. This metadata is controlled by the **PICTURE** clause, or abbreviated **PIC**. Think of a picture or photograph that gives you a better idea of what things look like. A PICTURE 9 describes numeric data and a PICTURE X describes character data. You can also tell your program how much data you plan to use by providing a length in parenthesis. It is important to note that you can only perform arithmetic operations on data defined as PICTURE 9. Also, you can give memory locations starting values by using the **VALUE** clause. See the examples below:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

WORKING-STORAGE SECTION.

\*\*\*\* NAMED MEMORY LOCATIONS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

01 WS-NAMED-MEMORY-LOCATIONS.

05 MAILBOX-1 PIC X(50) VALUE

'This is an example of PIC X character string'.

05 MAILBOX-2 PIC X(50).

\*\*\*\* EXAMPLE OF REPURPOSING THE SAME MEMORY ADDRESSES \*\*\*\*\*\*\*\*\*\*\*

05 MAILBOX-3 PIC 9(5) VALUE 15.

05 MAILBOX-4 REDEFINES MAILBOX-3 PIC XX.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**PROCEDURE DIVISION** - this division is where you put your commands that you want your program to perform. This is where you read from and write to your files, move data, perform logical and arithmetic operations, just like you did with your little man computer program.

In the sample program below we defined four named memory locations. We define these locations in the **WORKING-STORAGE SECTION** of the **DATA DIVISION**. Here we named the memory locations MAILBOX-1 through MAILBOX-4 and used a variety of **PICTURE** clauses. Some are locations are character and others are numeric. We also gave some of the memory locations starting values by using the **VALUE** clause.

Now that we have some data to work with we can manipulate the data in the **PROCEDURE DIVISION** using COBOL commands. To copy data from one memory location to another we use the **MOVE** statement. To show data to the user we use the **DISPLAY** statement and to do arithmetic operations we use **ADD,** **SUBTRACT, MULTIPLY** and **DIVIDE**. In our sample program below we just used the **ADD** statement. To get our program to stop we use the **STOP RUN** command. Notice that I also introduced two paragraph names that start with numbers. Paragraph names start in margin A and can start with numbers or letters are a way to organize your code into logical units and giving you the capability to execute them at different times. As your programs get bigger and more complex you’ll see that paragraphs help you make complexity much more manageable. A tip about paragraphs, keep paragraphs small and make each paragraph perform only one task. For example, in a payroll program, use one paragraph to calculate the employee’s net pay.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

PROCEDURE DIVISION.

\*\*\*\* HERE IS WHERE YOUR EXECUTABLE INSTRUCTIONS GO \*\*\*\*\*\*\*\*\*\*\*\*\*\*

000-DISPLAY-MESSAGES.

MOVE 'GET239 Enterprise Technologies' TO MAILBOX-1.

DISPLAY MAILBOX-1

DISPLAY 'THIS IS GET239 LAB 3'

DISPLAY MAILBOX-3.

ADD 45 to MAILBOX-3.

MOVE MAILBOX-3 TO MAILBOX-4.

DISPLAY MAILBOX-4.

display 'Notice the COBOL commands are case-insensitive'

Display ' ... but the data is case-sensitive'.

100-END-PROGRAM.

STOP RUN.

1. **Let’s prepare the program then execute it**

Now that we have typed your COBOL program in source code format we want to demonstrate the **von Neumann “Stored Program” concept** by preparing it for execution and loading it into memory.

The next step is to compile the COBOL source code. The COBOL compiler will translate each line of COBOL code in to machine readable object code; but it will ignore any comments you added to document your program.

To compile your program you need to run a compile job. That is, you need to submit your source code to the compiler. There is a compile job already setup and ready for you to use. Go to the **SHARE.GET239.JCL** folder and copy **COMPIL96.jcl** to your SUSnnnn.GET239.JCL folder. Open it up in the RDz editor by double-clicking on it. It will look like this:

//SUS0000C JOB (000000),'Dave D',

// NOTIFY=&SYSUID,

// CLASS=A

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//\* COMPILE PROGRAM

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//COBRUN EXEC IGYWCL,PARM.COBOL='RENT,LIST'

//COBOL.SYSIN DD DISP=SHR,DSN=&SYSUID..GET239.COBOL(LAB3)

//LKED.SYSLMOD DD DISP=SHR,DSN=&SYSUID..GET239.LOADLIB(LAB3)

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//\* END COMPILE PROGRAM

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

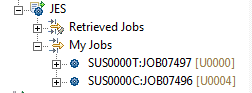
Make and save the following changes to the **COMPIL96.jcl** file. They are outlined in red above:

1. Change the SUSnnnnC to your z/OS SUS ID followed by a C for compile

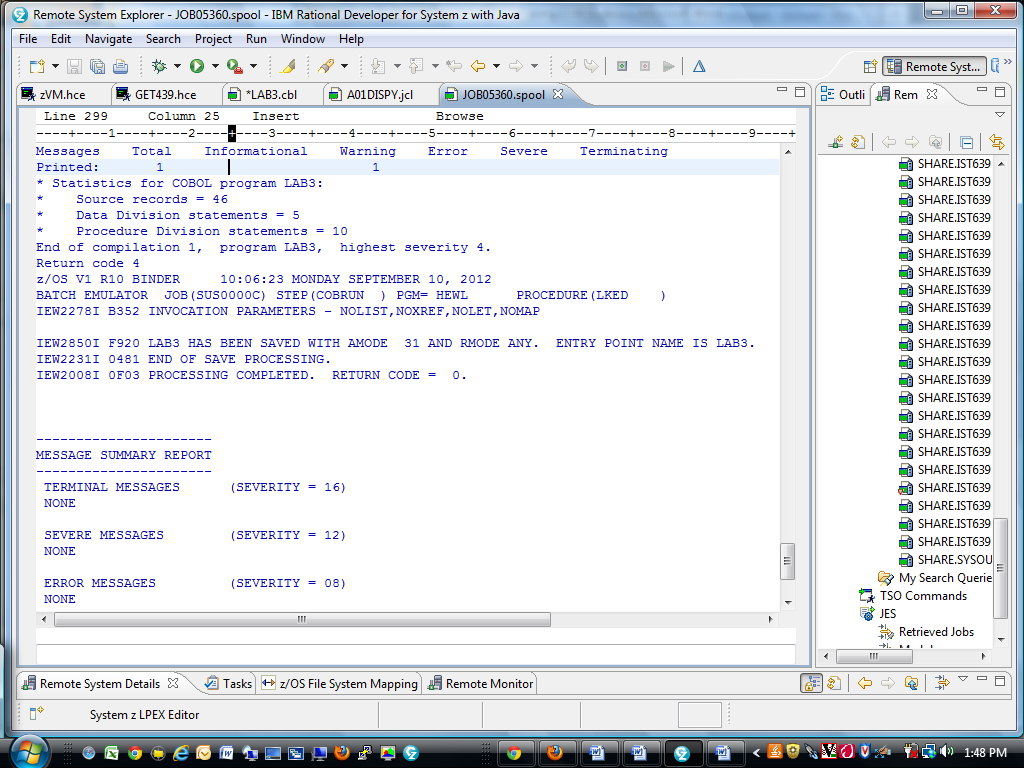
2. Change ‘your name’ to your first and last name; be sure to delineate it with the single quotes

3. Change the program name on the two DD statements to LAB3; be sure to enclose the LAB3 in parenthesis.

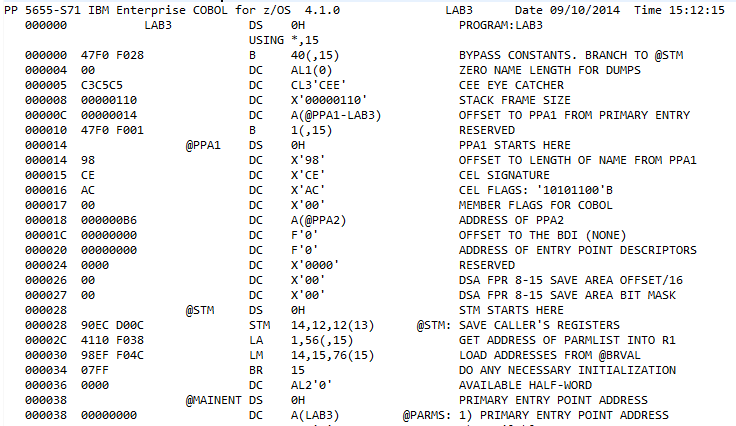
After you have made these changes, from the editor, right-click then select Submit the Compil96.jcl. Your job will be submitted to the Job Entry Subsystem, or JES, for processing. In RDz, open the JES folder located at the bottom of the Remote Systems Explorer. You’ll see your job listed when you open the My Jobs folder:



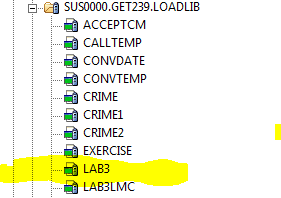
Double-click on the compile job to see if your program compiled with no errors. If the RDz editor didn’t find any errors then the compiler won’t find any errors either. Let’s look at the output from the compiler:



Here you can see that the Return Code is 0 which means the **LAB3** load module has been created. Also in the listing is the translated source code, called object code. The compiler translated each of your COBOL source code instructions into a language that the computer understands. See if you can find it. It should look like this:



If you go to your SUSnnnn.GET239.LOADLIB folder and open it you will find the executable program.



**You’ll notice that I have many more load modules in my library than you. By the end of the semester you will have many more as well**

1. Now we want to execute the **LAB3** load module program. We will follow the same procedure that we used to compile the program. To find the execute JCL go to **SHARE.GET239.JCL** and copy **LAB3JCL** to your SUSnnnn.GET239.JCL folder. Double-click to open the **LAB3JCL** file and it should look like this:

//SUSnnnnT JOB (000000),'your name',

// NOTIFY=&SYSUID,

// TIME=(,5),

// LINES=500,

// CLASS=A

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//\*STEP TO EXECUTE COBOL PROGRAM \*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//STEP01 EXEC PGM=LAB3

//STEPLIB DD DSN=&SYSUID..GET239.LOADLIB,

// DISP=SHR

//SYSPRINT DD SYSOUT=\*

//SYSOUT DD SYSOUT=\*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//

Make the following changes then submit the **LAB3JCL** to execute you new COBOL program:

1. On the JOB statement, change SUSnnnnT to your SUS z/OS ID followed by a T

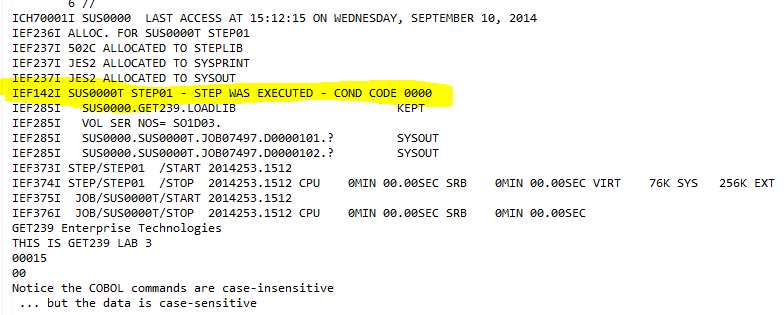
2. Also on the JOB statement change ‘your name’ to your first and last name and be sure your name is delineated by single quotes.

Notice there are other parameters on the JOB statement like:

* **NOTIFY=&SYSUID** will send job status information to your SUSnnn account. It’s a way for you to monitor your job
* **TIME=(,5)** limits your program to 5 CPU seconds of execution time. No need to worry, 5 CPU seconds is a lot of CPU time.
* **LINES=500** will limit the number of output created by your program
* **CLASS=A** is your way to enforce your environment standards and governs the conditions under which your program will run. CLASS=A on our system is intended for jobs that require no manual intervention and consume very little system resources. You will learn more about the CLASS parameter later in the semester

If LINES exceed 500 or CPU exceeds 5 seconds the operating system will terminate the job.

After your **LAB3JCL** runs you can find your execution in JES:



Notice that your **LAB3** program was executed successfully with a Return Code of 0 and the last five lines of print are the results of your COBOL DISPLAY statements.

**This is a sample of questions to get the students thing about hardware architecture and its relationship to the software.**

1. **List the four (4) principles that describe von Neumann Architecture? (4)**

a.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

b. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

c. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

d.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. **Describe a. Source code, b. Object code and c. Load Module. (3)**

a.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

b.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

c.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. **a. Explain the “stored-program” concept. b. What part of this lab demonstrates the “stored-program” concept? What is the difference between a compiler and an interpreter? (3)**

a. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

b. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

c. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. **Write a COBOL program named LAB3LMC to do the following. This is the same Little Man Computer exercise we did in class.**
2. Display splash page (see example of output on next page) that has the following items:

* GET239 Enterprise Technologies LAB3LMC3
* Your name (First & Last)
* Today’s Date
* Surround the splash page with asterisks

1. Define the following memory locations:

MAILBOX-1, numeric, length 3, VALUE 15

MAILBOX-2, numeric, length 3, VALUE 45

MAILBOX-3, numeric, length 3, VALUE 40

MAILBOX-4, numeric, length 3, VALUE 100

MAILBOX-95, numeric, length 3, VALUE 0

MAILBOX-96, numeric, length 3, VALUE 0

MAILBOX-97, numeric, length 3, VALUE 0

MAILBOX-98, numeric, length 3, VALUE 0

MAILBOX-99, numeric, length 3, VALUE 0

1. Store the value from Mailbox-1 to Mailbox-95
2. Store the value from Mailbox-2 to Mailbox-96
3. Store the value from Mailbox-3 to Mailbox-97
4. Store the value from Mailbox-4 to Mailbox-98
5. Calculate the sum of mailboxes 95 thru 98
6. Store the sum in Mailbox-99
7. Display the contents of Mailboxes 95 - 99 be sure to zero suppress the leading zeroes
8. **Lab Deliverables -** submit to me – see next page for examples:
   * typed answers from questions above
   * LAB3LMC source code (Copy & paste from the RDz COBOL editor)
   * JES data sets to prove that your compile and the execution work.

1. **Deliverables needed to get full credit. (10)**
   1. A copy of your fully documented COBOL source code
   2. Your programs execute JCL output and programs results
   3. Screen print of your load library showing your load module

**In addition to your source code, your LAB3LMC program execution should look likes something like this.**

J E S 2 J O B L O G -- S Y S T E M S Y S 1 -- N O D E N 1

5b

06.37.42 JOB05421 ---- WEDNESDAY, 12 SEP 2012 ----

06.37.42 JOB05421 IRR010I USERID SUS0000 IS ASSIGNED TO THIS JOB.

06.37.42 JOB05421 ICH70001I SUS0000 LAST ACCESS AT 06:35:53 ON THURSDAY, SEPTEMBER 11, 2014

06.37.42 JOB05421 $HASP373 SUS0000T STARTED - INIT 1 - CLASS A - SYS SYS1

06.37.42 JOB05421 $HASP395 SUS0000T ENDED

------ JES2 JOB STATISTICS ------

11 SEP 2014 JOB EXECUTION DATE

15 CARDS READ

59 SYSOUT PRINT RECORDS

0 SYSOUT PUNCH RECORDS

3 SYSOUT SPOOL KBYTES

0.00 MINUTES EXECUTION TIME

1 //SUS0000T JOB (000000),'Dave D', JOB05421

// NOTIFY=&SYSUID,

// CLASS=A,

// MSGCLASS=X,

// REGION=4096K

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//\*STEP TO EXECUTE COBOL PROGRAM \*/ //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

IEFC653I SUBSTITUTION JCL - (000000),'Dave D',NOTIFY=SUS0000,CLASS=A,MSGCLASS=X,REGION=4096K

2 //STEP01 EXEC PGM=LAB3LMC

3 //STEPLIB DD DSN=&SYSUID..GET239.LOADLIB,

// DISP=SHR

IEFC653I SUBSTITUTION JCL - DSN=SUS0000.GET239.LOADLIB,DISP=SHR

4 //SYSPRINT DD SYSOUT=\*

5 //SYSOUT DD SYSOUT=\*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

6 //

ICH70001I SUS0000 LAST ACCESS AT 06:35:53 ON THURSDAY, SEPTEMBER 11,2014

**Here is the output of your COBOL program. Notice that I suppressed the leading zeroes before I displayed the contents of my mailboxes (i.e. memory locations)**

.

5b

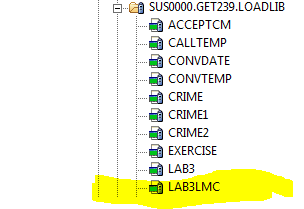
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* GET239 Enterprise Technologies LAB3LMC \*

\* Dave D \*

\* September 11, 2014 \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 15

45

40

100

---

SUM 200

5c

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* End of LAB3LMC \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*