

COBOL Programming with VSCode

A beginner's guide to COBOL using modern tooling



OPEN
MAINFRAME
PROJECT

COBOL Programming with VSCode

A beginner's guide to COBOL using modern tooling

Preface

Abstract

One computer programming language was designed specifically for business, Common Business-Oriented Language, COBOL. Today COBOL remains as relevant as ever, handling \$3 trillion in commerce every day.

This publication is aimed at beginners looking to build a working understanding of COBOL programming. It describes how to work with COBOL using modern tools including Visual Studio Code with Zowe and Z Open Editor extensions. It describes how to write, test, execute, and debug COBOL programs.

Authors

Michael Bauer is a development leader for the Open Mainframe value stream at Broadcom and is a squad lead for the Zowe open source initiative. Zowe, a popular framework of modern interfaces for z/OS, opens the mainframe to DevOps tools and practices. Mike leads the Command Line Interface (CLI) squad, which created and recently spun-off the successful Zowe Explorer extension for Visual Studio Code. A frequent speaker and blogger, Mike runs interactive workshops around the world for those interested in incorporating mainframe in their enterprise DevOps initiatives.

Zeibura Kathau is a technical writer for the Mainframe DevOps value stream at Broadcom. He works on the open-source projects Che4z and Code4z, which are IDE extension packages for mainframe developers. He has 8 years of experience in the Information Technology field.

Makenzie Manna is an IBM Redbooks Project Leader in the United States. She has 3 years of experience in the Computer Science Software Development field. She holds a Master's degree in Computer Science Software Development from Marist College. Her areas of expertise include mathematics, IBM Z and cloud computing.

Paul Newton is a Consulting IT Specialist in the United States. He has 40 years of experience in the Information Technology field. He holds a degree in Information Systems from the University of Arizona. His areas of expertise include IBM Z, z/OS, and LinuxONE. He has written extensively on implementation of z/OS based technology.

Jonathan Sayles is a technical educator at IBM, where he conducts presentations, seminars and training courses, as well as producing educational materials. His more than 40 years in the IT education and computer industries encompass work within both academic and corporate development organizations. He has also been engaged as a software developer/designer/consultant, educator, and author, with a focus on relational database, IDE, and object technologies. In addition to authoring/publishing 16 books, Jon has written and published more than 150 articles in technical journals, and served as technical editor for several IT magazines. He is also co-author of IBM Redbook publications Transitioning: Informix 4GL to Enterprise Generation Language (EGL), SG24-6673 and z/OS Traditional Application Maintenance and Support, SG24-7868.

William Yates is a Software engineer working for IBM UK. For the majority of his career he has worked on the CICS TS product mainly as a software tester and now as Test Architect. He has delivered technical content for many Redbooks, video courses and at conferences around the world. He is also one of the leaders of the Galasa project, building an open source integration test framework for hybrid cloud applications available at <https://galasa.dev>

Acknowledgements

Special thanks to the following people for participating in the residency to shape the content in this publication.

- Dr. Tak Auyeung, Professor, American River College
- Jeffrey Bisti, Z Ecosystem Architect, IBM

- Ilicena Elliott, IT Specialist II, Employment Development Department
- Martin Keen, Technical Content Services, IBM
- Sudharsana Srinivasan, z Influencer Ecosystem Program Coordinator, IBM
- Suzy Wong, Information Technology Specialist, DMV



Left-to-right: Ilicena, Suzy, Makenzie, Martin, Paul, and Tak

Contents

Part 1 - Getting started	10
1 Why COBOL?	10
1.1 What is COBOL?	10
1.2 How is COBOL being used today?	10
1.3 Why should I care about COBOL?	11
2 VSCode with Zowe Explorer	12
2.1 Introduction to Zowe Explorer	12
2.2 Using Zowe Explorer	12
2.3 Profiles in Zowe Explorer	13
2.3.1 Secure Credentials	13
2.3.2 Creating a New Profile	14
2.3.3 Editing Profiles	15
2.3.4 Deleting Profiles	17
2.4 Summary	20
3 VSCode with Z Open Editor	21
3.1 Introduction to the IBM Z Open Editor	21
3.1.1 What is the IBM Z Open Editor?	21
3.1.2 The role of the Language Server Protocol	21
3.1.3 Installing the IBM Z Open Editor for VS Code	22
3.2 Basic editing	22
3.2.1 Known file extensions	22
3.2.2 Margins	23
3.2.3 Variable expansion	23
3.2.4 Syntax highlighting	24
3.3 Navigation of code	24
3.3.1 Outline view	24
3.3.2 Breadcrumb view	24
3.3.3 Jump to declaration / reference	26
3.4 Code-completion	26
3.4.1 COBOL reserved word completion	27
3.4.2 Variable completion	27
3.4.3 CICS, MQ, DB2 API completion	27
3.5 Refactoring code	27
3.5.1 Renaming variables	29
3.5.2 Handling errors	29
3.6 Summary	31
4 VS Code with Code4z Open-Source Extension Package	32
4.1 What is Code4z?	32
4.2 Known File Extensions	32
4.3 Syntax Highlighting and Coloring	32
4.4 Syntax and Semantic Check	33
4.5 Navigation of Code	33
4.5.1 Breadcrumb View	33
4.5.2 Outline View	34
4.5.3 Shortcuts	34
4.5.4 Go To Definition	34
4.5.5 Find All References	34
4.6 Copybook Support	34
4.7 Autocomplete	35

4.8	Summary	35
5	Zowe CLI and Zowe CLI Plug-ins	36
5.1	What is a CLI and why would you use it?	36
5.2	What is Zowe CLI?	36
5.3	Zowe CLI interactive use	37
5.3.1	Installing Zowe CLI	37
5.3.2	Interactive Help	37
5.3.3	Zowe Profiles	37
5.3.4	Interacting with z/OS Data Sets	39
5.3.5	Interacting with z/OS Jobs	39
5.4	Automating tasks using Zowe CLI	41
5.4.1	Automated Job Submission	41
5.4.2	Using Other Programming Languages and Continuous Integration	43
5.4.3	Additional Examples	43
5.5	The world of modern open source tooling	43
5.6	Summary	45
6	Installation of VSCode and extensions	46
6.1	Install prerequisites	46
6.1.1	Install node.js	46
6.1.2	Install Java SDK	47
6.2	Install VSCode	47
6.3	Install VSCode extensions	48
6.3.1	Zowe Explorer	48
6.3.2	IBM Z Open Editor	49
6.3.3	Code4z	50
6.4	Summary	50
7	Installation of Zowe CLI and Plug-ins	52
7.1	Install prerequisites - Node.js	52
7.2	Install Zowe CLI	52
7.2.1	Install from Public npm Registry	52
7.2.2	Install from Bundled Package	53
7.3	Install Zowe CLI Plug-ins	53
7.3.1	Install from Public npm Registry	54
7.3.2	Install from Bundled Package	54
7.4	Summary	54
Part 2 - Learning COBOL		55
8	Basic COBOL	55
8.1	COBOL characteristics	55
8.1.1	Enterprise COBOL	55
8.1.2	Chapter objectives	56
8.2	What must a novice COBOL programmer know to be an experienced COBOL programmer?	56
8.2.1	What are the coding rules and the reference format?	56
8.2.2	What is the structure of COBOL?	57
8.2.3	What are COBOL reserved words?	57
8.2.4	What is a COBOL statement?	58
8.2.5	What is the meaning of a scope terminator?	58
8.2.6	What is a COBOL sentence?	58
8.2.7	What is a COBOL paragraph?	58
8.2.8	What is a COBOL section?	58
8.3	COBOL Divisions	58

8.3.1	COBOL Divisions structure	59
8.3.2	What are the four Divisions of COBOL?	59
8.4	PROCEDURE DIVISION explained	59
8.5	Additional information	60
8.5.1	Professional manuals	60
8.5.2	Learn more about recent COBOL advancements	60
8.6	Lab	61
8.7	Lab - Zowe CLI & Automation	70
8.7.1	Zowe CLI - Interactive Usage	70
8.7.2	Zowe CLI - Programmatic Usage	76
9	Data division	79
9.1	Variables / Data-items	79
9.1.1	Variable / Data-item name restrictions and data types	79
9.2	PICTURE clause	80
9.2.1	PIC clause symbols and data types	80
9.2.2	Coding COBOL variable / data-item names	80
9.2.3	PICTURE clause character-string representation	81
9.3	Literals	81
9.3.1	Figurative constants	81
9.3.2	Data relationships	81
9.3.3	Levels of data	83
9.4	MOVE and COMPUTE	83
9.5	Lab	83
10	File handling	87
10.1	COBOL code used for sequential file handling	87
10.1.1	COBOL inputs and outputs	88
10.1.2	FILE-CONTROL paragraph	88
10.1.3	COBOL external data source	88
10.1.4	Data sets, records, and fields	89
10.1.5	Blocks	89
10.1.6	ASSIGN clause	90
10.2	PROCEDURE DIVISION sequential file handling	90
10.2.1	Open input and output for read and write	91
10.2.2	Close input and output	91
10.3	COBOL programming techniques to read and write records sequentially	91
10.3.1	READ-NEXT-RECORD paragraph execution	92
10.3.2	READ-RECORD paragraph	93
10.3.3	WRITE-RECORD paragraph	93
10.3.4	Iterative processing of READ-NEXT-RECORD paragraph	93
10.4	Lab	93
11	Program structure	97
11.1	Styles of programming	97
11.1.1	What is structured programming	98
11.1.2	What is Object Orientated Programming	98
11.1.3	COBOL programming style	98
11.2	Structure of the Procedure Division	98
11.2.1	Program control and flow through a basic program	98
11.2.2	Inline and out of line perform statements	99
11.2.3	Using performs to code a loop	100
11.2.4	Learning bad behavior using the GO TO keyword	100
11.3	Paragraphs as blocks of code	102

11.3.1	Designing the content of a paragraph	102
11.3.2	Order and naming of paragraphs	102
11.4	Program control with paragraphs	104
11.4.1	PERFORM TIMES	104
11.4.2	PERFORM THROUGH	104
11.4.3	PERFORM UNTIL	105
11.4.4	PERFORM VARYING	105
11.5	Using subprograms	106
11.5.1	Specifying the target program	106
11.5.2	Specifying program variables	107
11.5.3	Specifying the return value	107
11.6	Summary	107
12	File output	108
12.1	Review of COBOL write output process	108
12.1.1	ENVIRONMENT DIVISION	108
12.2	FILE DESCRIPTOR	108
12.2.1	FILLER	109
12.3	Report and column headers	109
12.3.1	HEADER-2	112
12.4	PROCEDURE DIVISION	112
12.4.1	MOVE sentences	113
12.4.2	PRINT-REC FROM sentences	113
12.5	Lab	113
13	Conditional expressions	115
13.1	Boolean logic, operators, operands, and identifiers	115
13.1.1	COBOL conditional expressions and operators	115
13.1.2	Examples of conditional expressions using Boolean operators	117
13.2	Conditional expression reserved words and terminology	117
13.2.1	IF, EVALUATE, PERFORM and SEARCH	117
13.2.2	Conditional states	117
13.2.3	Conditional names	117
13.3	Conditional operators	119
13.4	Conditional expressions	119
13.4.1	IF ELSE (THEN) statements	119
13.4.2	EVALUATE statements	119
13.4.3	PERFORM statements	120
13.4.4	SEARCH statements	120
13.5	Conditions	120
13.5.1	Relation conditions	121
13.5.2	Class conditions	121
13.5.3	Sign conditions	121
13.6	Lab	121
14	Arithmetic expressions	123
14.1	What is an arithmetic expression?	123
14.1.1	Arithmetic operators	124
14.1.2	Arithmetic statements	124
14.2	Arithmetic expression precedence rules	124
14.2.1	Parentheses	125
14.3	Arithmetic expression limitations	125
14.4	Arithmetic statement operands	125
14.4.1	Size of operands	125

14.5 Examples of COBOL arithmetic statements	127
14.6 Lab	128
15 Data types	130
15.1 Data representation	130
15.1.1 Numerical value representation	130
15.1.2 Text representations	131
15.2 COBOL DISPLAY vs COMPUTATIONAL	132
15.3 Lab	132
16 Intrinsic functions	133
16.1 What is an intrinsic function?	133
16.1.1 Intrinsic function syntax	133
16.1.2 Categories of intrinsic functions	134
16.2 Intrinsic functions in Enterprise COBOL for z/OS V6.3	134
16.2.1 Mathematical example	135
16.2.2 Statistical example	135
16.2.3 Date/time example	135
16.2.4 Financial example	135
16.2.5 Character-handling example	136
16.3 Use of intrinsic functions with reference modifiers	136
16.4 Lab	136

Part 1 - Getting started

1 Why COBOL?

This chapter introduces COBOL, specifically with reference to its use in enterprise systems.

- **What is COBOL?**
- **How is COBOL being used today?**
- **Why should I care about COBOL?**

1.1 What is COBOL?

One computer programming language was designed specifically for business, Common Business-Oriented Language, COBOL. COBOL has been transforming and supporting business globally since its invention in 1959. COBOL is responsible for the efficient, reliable, secure and unseen day-to-day operation of the world's economy. The day-to-day logic used to process our most critical data is frequently done using COBOL.

Many COBOL programs have decades of improvements which includes business logic, performance, programming paradigm, and application program interfaces to transaction processors, data sources, and the Internet.

Many hundreds of programming languages were developed during the past 60 years with expectations to transform the information technology landscape. Some of these languages, such as C, C++, Java, and JavaScript, have indeed transformed the ever-expanding information technology landscape. However, COBOL continues to distinguish itself from other programming languages due to its inherent ability to handle vast amounts of critical data stored in the largest servers such as the IBM Z mainframe.

Continuously updated to incorporate modernized and proven programming paradigms and best practices, COBOL will remain a critical programming language into the foreseeable future. Learning COBOL enables you to read and understand the day-to-day operation of critical systems. COBOL knowledge and proficiency is a required skill to be a “full stack developer” in large enterprises.

1.2 How is COBOL being used today?

COBOL is everywhere. You have probably used an application written in COBOL today. For example, consider the following statistics:

- About 95% of ATM swipes use COBOL code.
- COBOL powers 80% of in-person transactions.
- Every day, COBOL systems handle \$3 trillion in commerce.

How pervasive is COBOL? Consider these mind-boggling facts:

- Every day there are 200 times more COBOL transactions executed than there are Google searches.
- There are over 220 billion lines of COBOL programs running today, which equates to around 80% of the world's actively used code.
- 1,500,000,000 lines of new COBOL code is written each year.

1.3 Why should I care about COBOL?

The COBOL programming language, COBOL compiler optimization, and COBOL run time performance have over 50 years of technology advancements that contribute to the foundation of world's economy. The core business logic of many large enterprises has decades of business improvement and tuning embedded in COBOL programs.

The point is - whatever you read or hear about COBOL, be very skeptical. If you have the opportunity to work directly with someone involved in writing or maintaining critical business logic using COBOL, you will learn about the operation of the core business. Business managers, business analysts, and decision makers come and go. The sum of all good business decisions can frequently be found in the decades of changes implemented in COBOL programs. The answer to "How does this business actually work?" can be found in COBOL programs.

Add the following to your awareness of COBOL. It is an absolute myth that you must be at least 50 years old to be good with COBOL. COBOL is incredibly easy to learn and understand. One of the many reasons financial institutions like COBOL, is the fact that it is not necessary to be a programmer to read and understand the logic. This is important because critical business logic code is subject to audit. Auditors are not programmers. However, auditors are responsible for ensuring the business financial statements are presented fairly. It is COBOL processing that frequently result in the business ledger updates and subsequent financial statements.

Now for a real-world lesson. A comment recently made in a well-known business journal by someone with a suspect agenda was quoted as saying, "COBOL is a computing language used in business and finance. It was first designed in 1959 and is pretty old and slow." A highly experienced business technology person knows the only true part of that last sentence was that COBOL was first designed in 1959.

It's no secret that lots of banks still run millions of lines of COBOL on mainframes. They probably want to replace that at some point. So why haven't they? Most banks have been around long enough to still feel the pain from the ~1960's software crisis. After spending enormous amounts of money, and time, on developing their computer systems, they finally ended up with a fully functional, well-tested, stable COBOL core system.

Speaking with people that have worked on such systems, nowadays they have Java front ends and wrappers which add functionality or more modern interfaces, they run the application on virtualized replicated servers, but in the end, everything runs through that single core logic. And that core logic is rarely touched or changed, unless necessary.

From a software engineering perspective, that even makes sense. Rewrites are always more expensive than planned, and always take longer than planned (OK, probably not always. But often.). Never change a running system etc., unless very good technical and business reasons exist.

2 VSCode with Zowe Explorer

Zowe Explorer is an open-source extension for VS Code that lets developers and system administrators interact with z/OS mainframes.

- **Introduction to Zowe Explorer**
- **Using Zowe Explorer**
- **Profiles in Zowe Explorer**
 - Secure Credentials
 - Creating a New Profile
 - Editing Profiles
 - Deleting Profiles
- **Summary**

2.1 Introduction to Zowe Explorer

The Zowe Explorer extension modernizes the way developers and system administrators interact with z/OS mainframes. Working with data sets and USS files from VS Code can be more convenient than using 3270 emulators. The extension provides the following benefits:

- Create, modify, rename, copy and upload data sets directly to a z/OS mainframe.
- Create, modify, rename and upload USS files directly to a z/OS mainframe.
- Streamlined process to access data sets, USS files and jobs.
- Easy interact with multiple z/OS systems

The Zowe Explorer can be installed into VS Code by searching the Extensions Marketplace inside VS Code for “Zowe Explorer” and selecting install. To see more detailed instructions on installing this extension, refer to “Installation of VSCode and Extensions”.

2.2 Using Zowe Explorer

Zowe Explorer allows you to work with data sets, Unix System Service (USS) files, and jobs.

Zowe Explorer offers the following functions:

Data sets

- Search for data sets matching desired filters and view their contents
- Download, edit, and upload existing PDS members
- Create and delete both data sets and data set members
- Interact with data sets from multiple systems simultaneously
- Rename data sets
- Copy data sets
- Submit JCL from a chosen data set member

USS Files

- View multiple Unix System Services (USS) files simultaneously
- Download, edit, and upload existing USS files
- Create and delete USS files and directories
- Interact with USS files from multiple systems simultaneously

- Rename USS files

Jobs

- View multiple jobs simultaneously
- Download spool content
- Interact with jobs from multiple systems simultaneously

For more information about Zowe Explorer and the different use cases, visit the [marketplace](#)

2.3 Profiles in Zowe Explorer

Profiles serve as a point-of-contact for Zowe Explorer and the Mainframe. Profiles contain the URL for the API services that you want to connect to, and your credentials. The main profile information that you need for Zowe Explorer is the z/OSMF Connection. If you have the Zowe Explorer installed, you can follow the steps in this section to connect to the mainframe.

2.3.1 Secure Credentials

Zowe Explorer has a built in Secure Credential Store. This enables you to encrypt the credentials that are stored in your machine, and as a result secure your connection to the Mainframe.

To enable this feature, follow these steps:

1. Click the **Gear Icon** at the bottom left and select **Settings**
2. Click **User Settings > Extensions > Zowe Explorer Settings** Look for the **Zowe Security: Credential Key** field

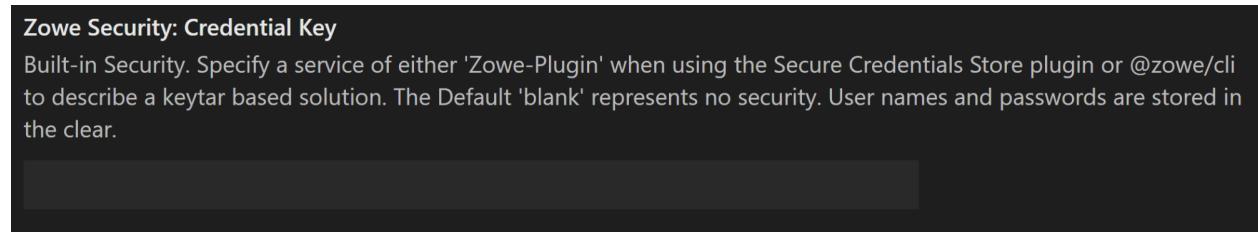


Figure 1:

3. Type **Zowe-Plugin** in the text box. This will trigger the Built-in Secure Credential Store.

Alternatively, to enable this feature by editing settings.json, hover over the gear icon and click “Copy Setting as JSON”. You can then paste that to settings.json and update the value to Zowe Plugin.

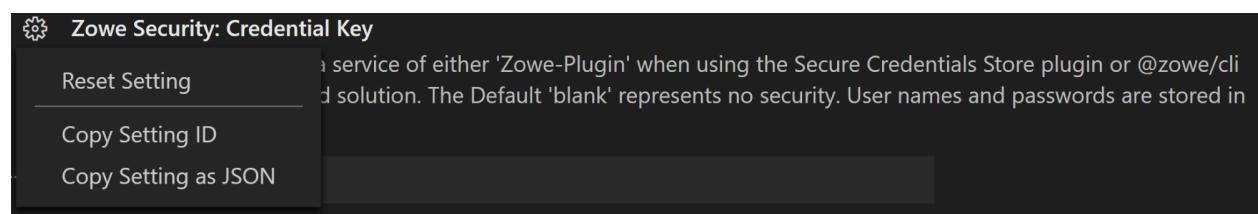


Figure 2:

Note: If you are using Zowe CLI and you've installed the Secure-Credential-Store Plugin, the steps to activate it will still be the same.

2.3.2 Creating a New Profile

Follow these steps:

1. Navigate to the Zowe Explorer tree on the right side and look for the + sign.

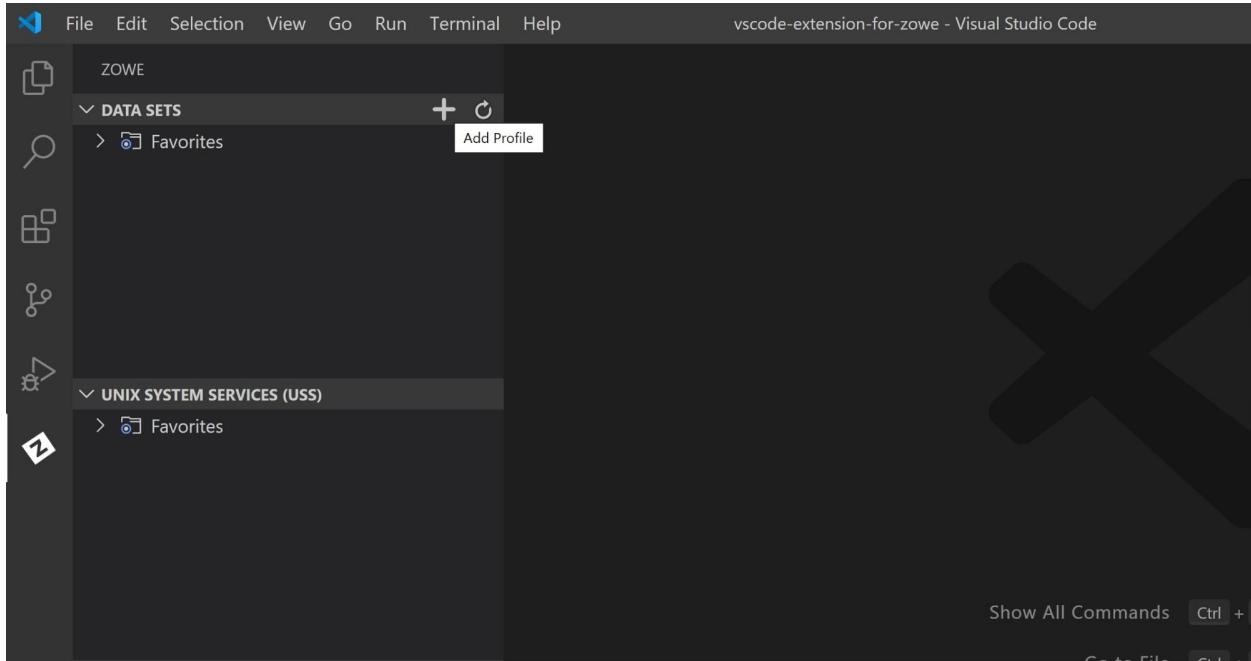


Figure 3:

2. Click on the + sign. A dialog box will appear and ask if you want to “Create a New Connection to z/OS”.
3. Press enter or click on that selection.

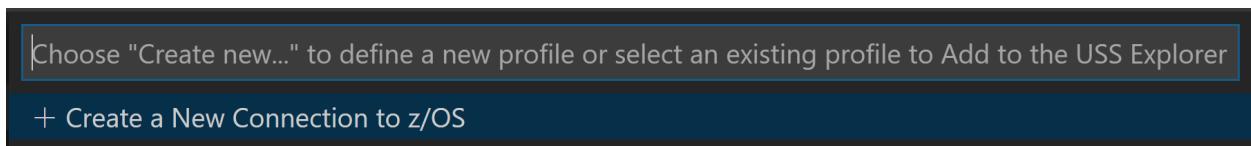


Figure 4:

4. Enter a Profile name in the “Connection Name” field.

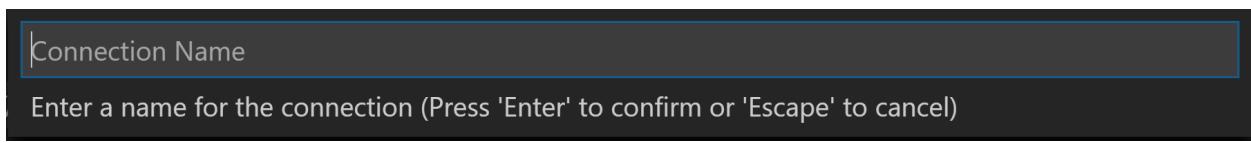


Figure 5:

5. Enter the URL and Port that you received by email when you registered for the COBOL Course. The connection information that you need has a title of “IP address for VSCode extension”.
6. Enter your Username. This is also included in the email.

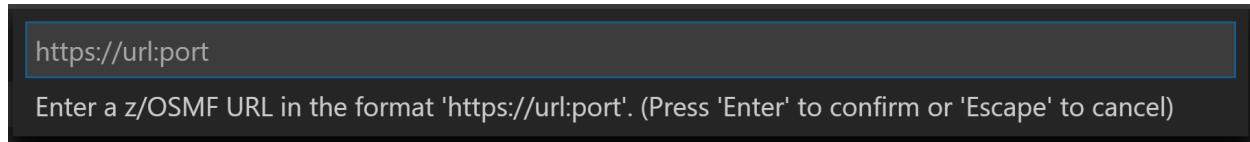


Figure 6:

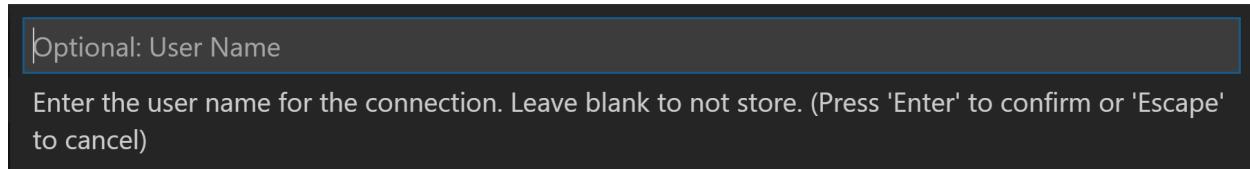


Figure 7:

Note: You can leave this blank if you do not want to save your credentials in your machine. You will be prompted for your username once you start using Zowe Explorer.

7. Enter your Password.

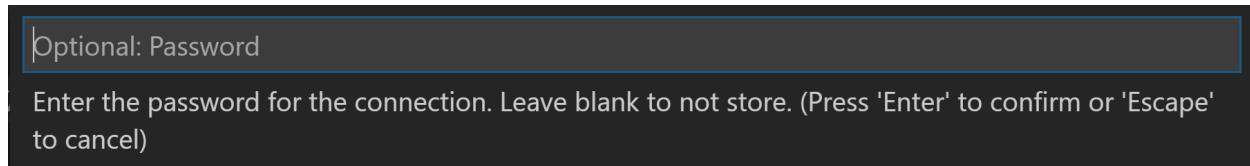


Figure 8:

Note: You can leave this blank if you do not want to save your credentials in your machine. You will be prompted for your username once you start using Zowe Explorer.

8. Select True/False if you want to accept or reject Self-Signed Certificates. For this course, please select false.

If you are successful, you will receive this informational message:

2.3.3 Editing Profiles

The Zowe Explorer v1.5.0 release introduces profile editing. This allows you to revise your existing profile information and continue using Zowe Explorer.

Follow these steps:

1. Add your profile to any of the Zowe Explorer Trees.
2. Click on the pencil icon to edit your profile. A dialog box opens displaying the current information in your profile, which you can edit as required.
3. Edit the URL information if changes are required, or enter to confirm the information is still correct.
4. Edit your Username.

Note: You can leave this blank if you do not want to save your credentials in your machine. You will be prompted for your username once you start using Zowe Explorer.

5. Edit your Password.

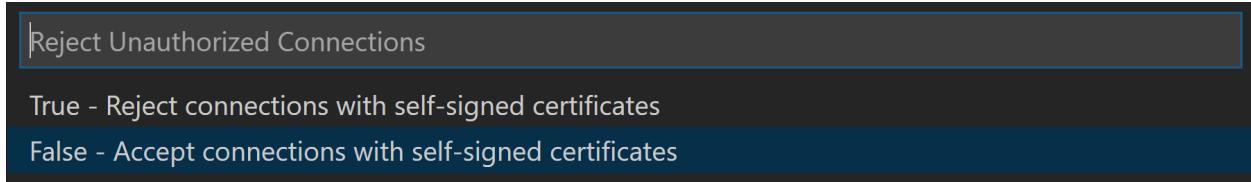


Figure 9:

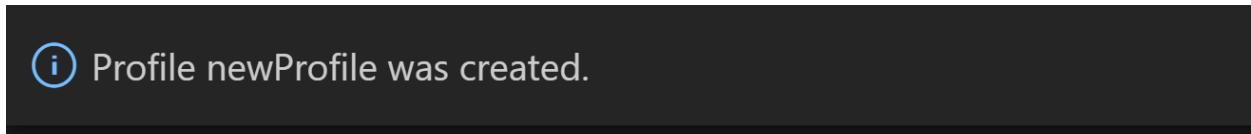


Figure 10:

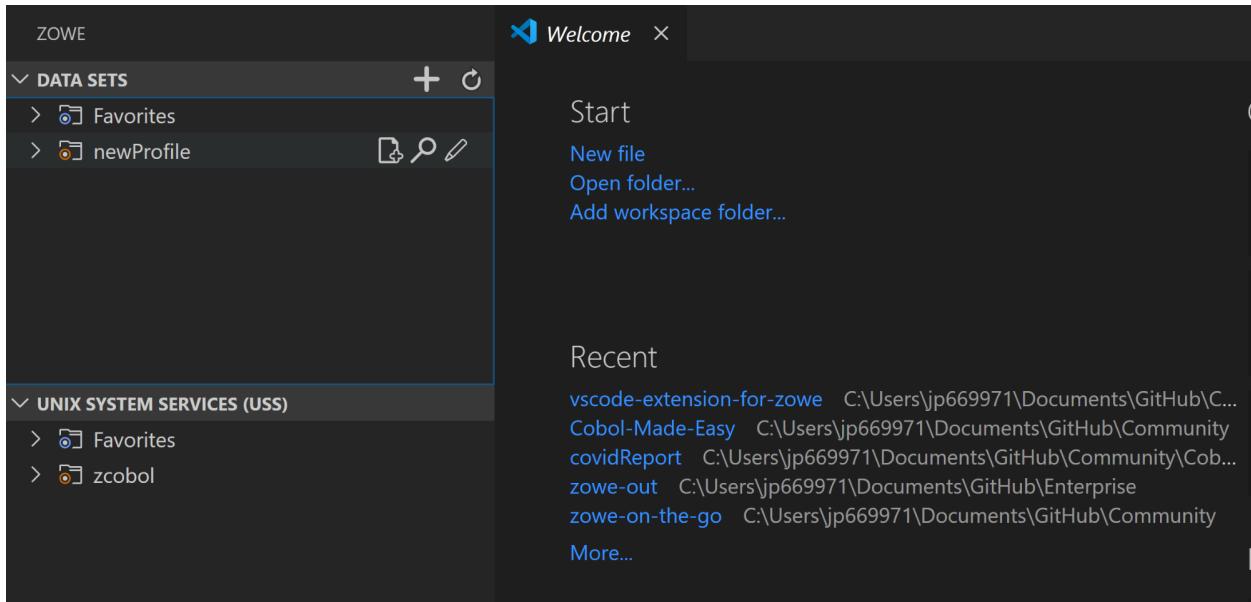


Figure 11:

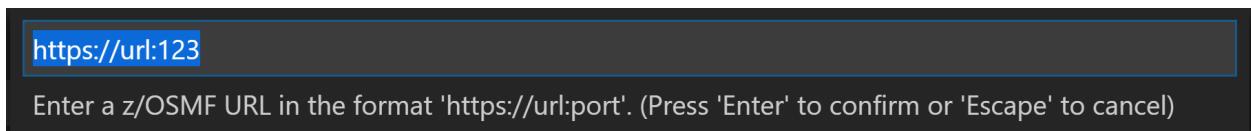


Figure 12:

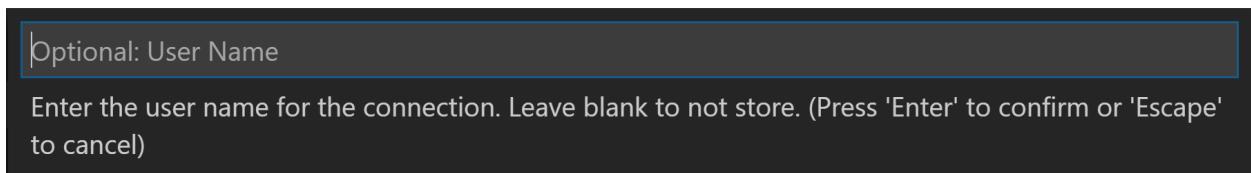


Figure 13:

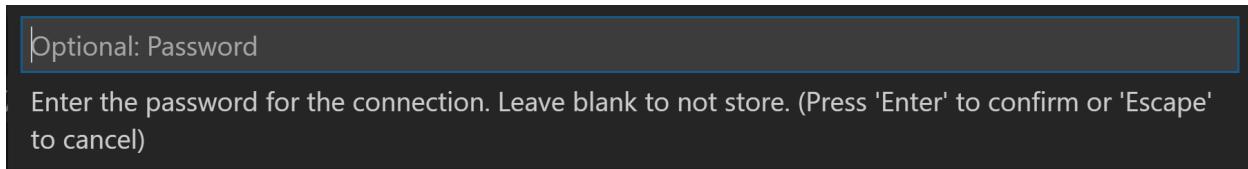


Figure 14:

Note: You can leave this blank if you do not want to save your credentials in your machine. You will be prompted for your username once you start using Zowe Explorer.

6. Edit your authorized connections

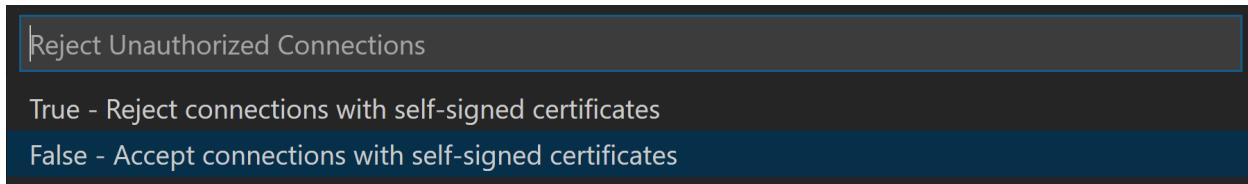


Figure 15:

If you are successful, an information message will appear:

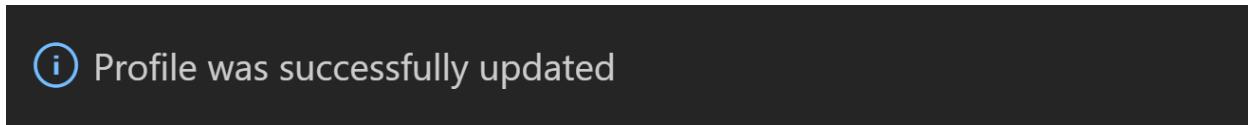


Figure 16:

2.3.4 Deleting Profiles

The Zowe Explorer v1.5.0 release introduces the option to delete profiles. This allows you to permanently delete unwanted profiles and clean up your files. You can delete profiles either using the command palette or in the tree.

Follow these steps:

Command Palette:

1. Press **CTRL+SHIFT+P** or Click **View > Command Palette** to open the Command Palette
2. Type “Zowe: Delete”. This command allows you to permanently delete a profile.
3. Select the Profile that you want to delete.
4. Confirm that you want to delete your profile.

Once confirmed, the following message is displayed:

Zowe Explorer Tree:

1. Right click on the profile and select **Delete Profile**.
2. Confirm that you want to delete your profile.
3. Once confirmed, the following message is displayed:

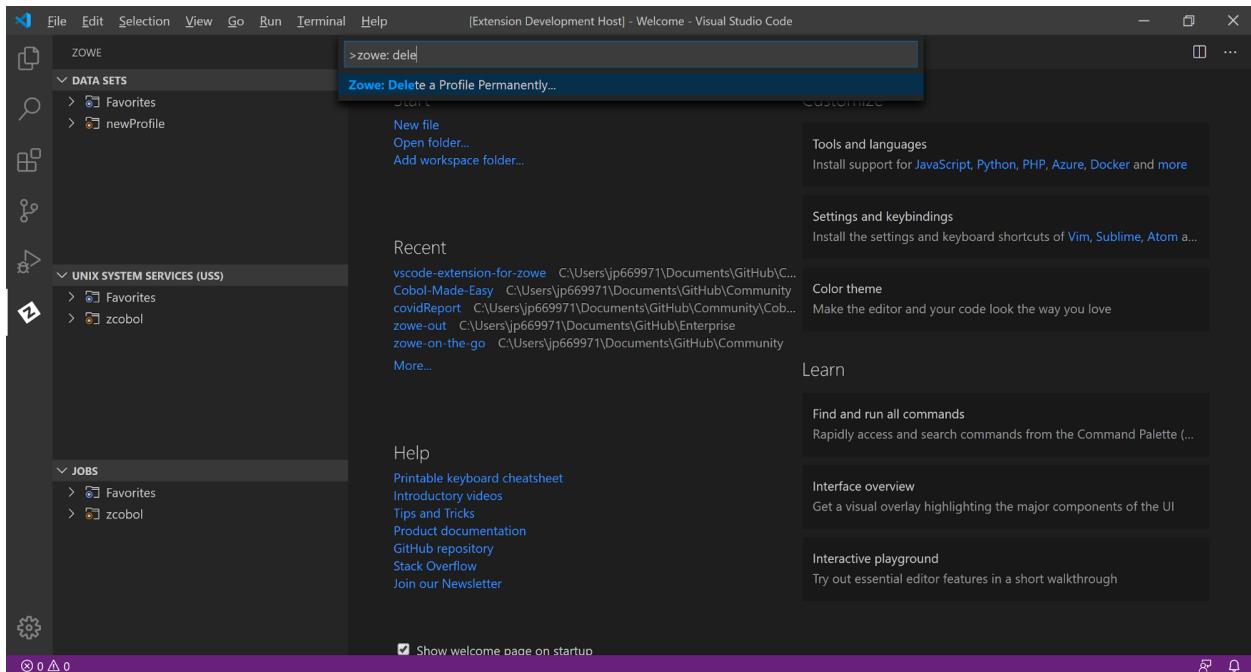


Figure 17:

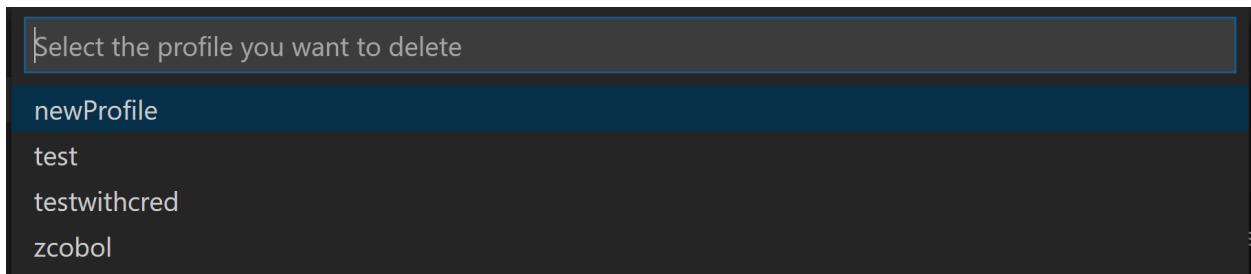


Figure 18:

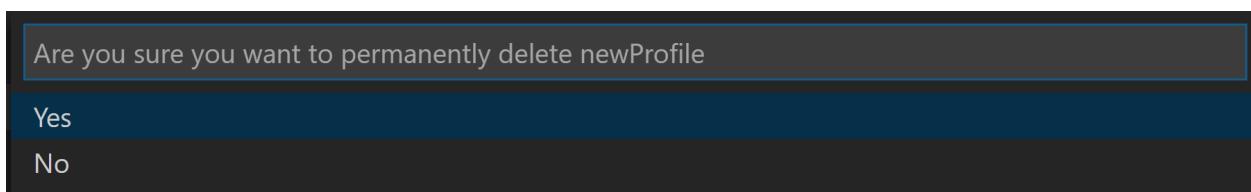


Figure 19:



Figure 20:

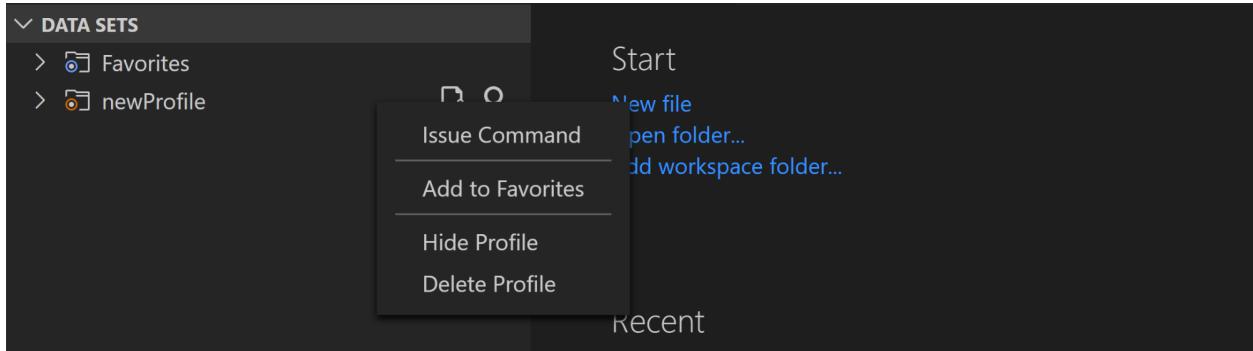


Figure 21:

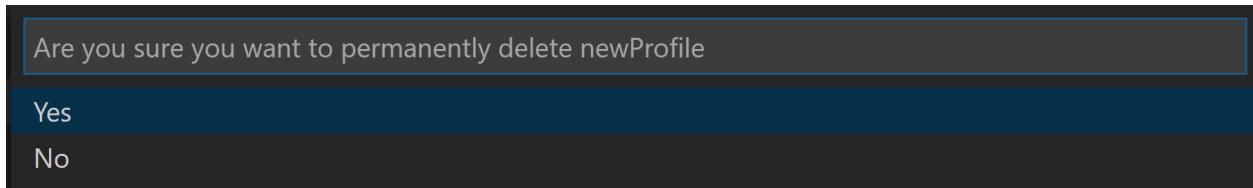


Figure 22:



Figure 23:

2.4 Summary

In this section you have learned the basic features of the Zowe Explorer extension and how to create and work with Zowe compatible `zosmf` profiles.

3 VSCode with Z Open Editor

In this chapter we will explain how to use the IBM Z Open Editor extension for VSCode and how using it can help you develop COBOL source code in a feature rich environment.

- **Introduction to the IBM Z Open Editor**
 - What is the IBM Z Open Editor?
 - The role of the Language Server Protocol
 - Installing the IBM Z Open Editor for VS Code
- **Basic editing**
 - Known file extensions
 - Margins
 - Variable expansion
 - Syntax highlighting
- **Navigation of code**
 - Outline view
 - Breadcrumb view
 - Jump to declaration / reference
- **Code-completion**
 - COBOL reserved word completion
 - Variable completion
 - CICS, MQ, DB2 API completion
- **Refactoring code**
 - Renaming variables
 - Handling errors
- **Summary**

3.1 Introduction to the IBM Z Open Editor

This section introduces the IBM Z Open Editor.

3.1.1 What is the IBM Z Open Editor?

The IBM Z Open Editor is a free extension for Visual Studio Code (VSCode) that provides language support for COBOL, PL/I and JCL languages. Along with this language support it also provides content assistance for applications that call CICS, MQ, IMS and DB2 APIs. The source code doesn't even need to reside on z/OS, it could be in a source code repository, locally in a file or on z/OS. Although this course focuses on COBOL as a source language, a lot of the functions we will discuss will also apply to PL/I and JCL.

3.1.2 The role of the Language Server Protocol

Integrated development environments always want to provide a rich platform for all supported programming languages, however, the proliferation of programming languages and the speed at which new editors reach the market makes keeping pace difficult. Each editor would need to provide an editor specific plugin for each language they wished to support, thus support for a certain language would differ between different editors.

Microsoft designed the Language Server Protocol (LSP) to act as a common description of how features like auto-complete should be implemented for a specific language. Languages which have an implemented LSP server can therefore be used within any editor that supports LSP. Many companies and the open source community have collaborated to provide LSP servers for an array of different languages.

The language server protocol defines six broad capabilities that should be implemented for a language server to be LSP compliant. These capabilities include code completion, hover, jump to definition, workspace symbols, find references and diagnostics. The IBM Z Open Editor provides compliant language servers for both the Enterprise Cobol and Enterprise PL/I for z/OS languages. In addition to being compliant, they also provide additional capabilities that we will discuss further on.

Note: More information on Language Server Protocol implementations can be found at: <https://langserver.org>

3.1.3 Installing the IBM Z Open Editor for VS Code

The IBM Z Open Editor can be installed into VS code by searching the Extensions Marketplace inside VSCode for “IBM Z Open Editor” and selecting install. Once installed, the default editor will be enabled to provide a rich editing experience for COBOL, PL/I and JCL. There is no need to use a specific editor for these languages. To see a more detailed instruction on installing this extension, refer to “Installation of VSCode and extensions”.

3.2 Basic editing

For the rest of this chapter we will use the CBL0001 sample program to demonstrate how rich of an experience editing COBOL in VSCode can be. So, let’s fire up VSCode, install IBM Z Open Editor (if it’s not already) open up CBL0001 and get started.

3.2.1 Known file extensions

For VSCode to use the capabilities of the IBM Z Open Editor, it needs to know that the file we are editing is in fact COBOL type. VSCode accomplishes this by comparing the location and name of the file being opened against a list of known extensions to map the file to a known language. For COBOL the following associations are used:

- *.COBOL*
- *.CBL*
- *.COB*
- *.COBCOPY*
- *.COPYBOOK*
- *.COPY*

These are applied to both local files and files held in a Partitioned Data Set or PDS on the mainframe, which for simplicity you can think of as a folder. Thus, a PDS called:

Z99994.COBOL

Or a file on the local file system called:

PROGA1.COBOL

Will be assumed to be COBOL code. This information is stored in the global VSCode settings.json file that can be accessed through VSCode preferences. This allows for a user to tailor VSCode's known extensions to a particular site naming convention.

3.2.2 Margins

The first thing you will notice when editing COBOL source code is that VSCode will have inserted five vertical lines down the file. These lines segment each line of code into the areas reserved for sequence numbers, comment / continuation characters, area A and area B. When coding without this basic aid I cannot recount the number of times I have made a compilation error because I started coding in the wrong column. This alone is a very useful aid to a new COBOL programmer. More information about COBOL syntax and in particular the columns will be discussed later

3.2.3 Variable expansion

As you browse through CBL0001 type **CTRL + G** to jump to a specific line of code. A small dialog will open asking you for the line you wish to jump to, type 68 and press the enter key. VSCode will highlight that line of code and navigate you directly to it, as shown in Figure 1.

```
test.hobit
Users > hobbit > Desktop > test.hobit:68 Go to line 68.
51
52     PERFORM READING.
53     PERFORM UNTIL LASTREC = "Y".
54         PERFORM WRITE-RECORD.
55         PERFORM READ-RECORD.
56     END-PERFORM.
57
58     CLOSE-STOP.
59         CLOSE ACCT-REC.
60         CLOSE PRINT-LINE.
61     STOP RUN.
62
63     READ-RECORD.
64         READ ACCT-REC.
65             AT END MOVE "Y" TO LASTREC.
66     END-READ.
67
68     WRITE-RECORD.
69         MOVE ACCT-NO      TO ACCT-NO-0.
70         MOVE ACCT-LIMIT   TO ACCT-LIMIT-0.
71         MOVE ACCT-BALANCE TO ACCT-BALANCE-0.
72         MOVE ACCT-NO      TO ACCT-NO-0.
73         MOVE LAST-NAME   TO LAST-NAME-0.
74         MOVE FIRST-NAME  TO FIRST-NAME-0.
75         MOVE COMMENTS    TO COMMENTS-0.
76     WRITE PRINT-REC.
77
```

A red arrow points from the word "MOVE" in the line "MOVE ACCT-NO TO ACCT-NO-0." to the first column of the line, indicating the start of the command.

Figure 24:

Figure 1. Navigating to a specific line of code

If you hover your mouse over the ‘ACCT-NO-O’ field a pop up will appear displaying the declaration of that variable, shown in Figure 2.

Figure 2. View declaration of variable

Since this field is a 05-level variable nested within a 01-level variable, the pop up shows the declaration of the field as an eight-byte picture variable, the name of the parent structure and the file definition that it is within. If you hold the CMD/Ctrl key while hovering over the field, then the pop up will additionally contain the line of code where the variable is defined as well as the following three lines of code. These features can be extremely helpful when analyzing existing code.

```

EXPLORER          settings.json      test.hobbit      Settings      test2.hobbit
OPEN EDITORS  UNSAVED: settings.json ~LibraryApplication.Slu...
X: test.hobbit ~Desktop
  Settings
  test2.hobbit ~Desktop
OUTLINE
  (1) PROGRAM: CBL0001
    IDENTIFICATION DIVISION.
  DATA DIVISION.
    FILE SECTION.
      FD ACCT-REC RECORDING MO...
        01 ACCT-FIELDS.
          05 ACCT-BALANCE
          05 ACCT-LIMIT
          05 ACCT-NO
          05 CITY-COUNTY
          05 COMMENTS
          05 FIRST-NAME
          05 LAST-NAME
          05 RESERVED
          05 STREET-ADDR
          05 USA-STATE
        FD PRINT-LINE RECORDING MO...
          01 PRINT-REC.

```

```

54      PERFORM READ-RECORD
55      END-PERFORM.
56
57      CLOSE-STOP.
58      CLOSE ACCT-REC.
59      CLOSE PRINT-LINE.
60      STOP RUN.
61
62      READ-RECORD.
63      READ ACCT-REC
64      AT END MOVE 'Y' TO LINE-NUMBER.
65      END-READ.
66
67      WRITE-RECORD.
68      MOVE ACCT-NO    TO ACCT-NO-0.
69      MOVE ACCT-LIMIT TO ACCT-LIMIT-0.
70      MOVE ACCT-BALANCE TO ACCT-BALANCE-0.
71      MOVE ACCT-NO    TO ACCT-NO-0.
72      MOVE LAST-NAME  TO LAST-NAME-0.
73      MOVE FIRST-NAME TO FIRST-NAME-0.
74      MOVE COMMENTS   TO COMMENTS-0.
75
76      WRITE PRINT-REC.
77

```

Figure 25:

3.2.4 Syntax highlighting

The COBOL code that you are editing will also be highlighted to help you understand the different elements of the COBOL language. Depending on the color theme that you have selected in VSCode, comments, COBOL reserved words, literal values and variables will be colored differently allowing you to spot obvious syntax issues early on before even submitting the code to a build.

3.3 Navigation of code

Although the code examples we are using in this section are fairly small, the code that you could be writing could have hundreds or thousands of lines. Being able to understand the general structure of the source code and being able to find your way around it without getting lost is a big advantage when editing COBOL. Fortunately, there are some great features to help you out, which we will discuss next.

3.3.1 Outline view

Within the explorer side bar of VSCode, there is an outline view that will be populated whenever you are editing a COBOL file. This view contains a collapsible look at each division, data structure and paragraph within your code. This allows you to easily view the structure of the source code. Clicking on a particular paragraph, division or data element will simultaneously move the editor to show that section of the code and highlight it, depicted in Figure 3. This makes jumping to a specific part of the code very easy.

Figure 3. Outline view in VSCode

3.3.2 Breadcrumb view

Similarly, the breadcrumb view across the top of the editor can show where the current line of code exists within the structure of the COBOL source code. As you navigate the source code in the editor, the breadcrumb trail will automatically update to reflect where you are in the structure of the program and provides you a mechanism to move to a different element of the code. Again, if you open CBL0001 in VSCode and jump to line 36, this line is a declaration of the field USA-STATE within the structure ACCT-FIELDS, in the FILE-SECTION of the DATA-DIVISION. Across the top of the editor the breadcrumb trail will show the information displayed in Figure 4.

```

test.hobbit X
Users > hobbit > Desktop > test.hobbit > CBL0001 > PROGRAM: CBL0001 > DATA DIVISION > FILE SECTION > FD ACCT-REC RECORDING MODE F
SELECT ACCT-REC ASSION TO ACCTREC
ORGANIZATION IS SEQUENTIAL.
DATA DIVISION.
FILE SECTION.
FD PRINT-LINE RECORDING MODE F.
 85 ACCT-NO-D PIC X(8).
 85 ACCT-LIMIT-D PIC $6,999,999.99.
 85 ACCT-BALANCE-D PIC $6,999,999.99.
 85 LAST-NAME-D PIC X(12).
 85 FIRST-NAME-D PIC X(15).
 85 COMMENTS-D PIC X(50).
FD ACCT-REC RECORDING MODE F.
 85 ACCT-FIELDS.
    85 ACCT-NO PIC X(8).
    85 ACCT-LIMIT PIC 5917VV99 COMP-3.
    85 ACCT-BALANCE PIC 5917VV99 COMP-3.
    85 LAST-NAME PIC X(12).
    85 FIRST-NAME PIC X(15).
    85 COMMENTS PIC X(50).
WORKING-STORAGE SECTION.
 85 FLAGS.
    85 LASTREC PIC X VALUE SPACE.
PROCEDURE DIVISION.
OPEN-FILES.

```

Figure 26:

```

zowe.vscode-extension-for-zowe-1.3.0 > resources > temp > _D > Redbook > Z99994.CBL(CBL0001).cbl > PROGRAM: CBL0001 > DATA DIVISION > FILE SECTION > FD ACCT-REC RECORDING MODE F > 85 05 USA-STATE
IDENTIFICATION DIVISION.
PROGRAM-ID. CBL0001
AUTHOR. Otto B. Fun.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT PRINT-LINE ASSION TO PRFLINE.
  SELECT ACCT-REC ASSION TO ACCTREC
  | ORGANIZATION IS SEQUENTIAL.
DATA DIVISION.
FILE SECTION.
FD PRINT-LINE RECORDING MODE F.
 85 ACCT-NO-D PIC X(8).
 85 ACCT-LIMIT-D PIC $6,999,999.99.
 85 ACCT-BALANCE-D PIC $6,999,999.99.
 85 LAST-NAME-D PIC X(12).
 85 FIRST-NAME-D PIC X(15).
 85 COMMENTS-D PIC X(50).
FD ACCT-REC RECORDING MODE F.
 85 ACCT-FIELDS.
    85 ACCT-NO PIC X(8).
    85 ACCT-LIMIT PIC 5917VV99 COMP-3.
    85 ACCT-BALANCE PIC 5917VV99 COMP-3.
    85 LAST-NAME PIC X(12).
    85 FIRST-NAME PIC X(15).
    85 COMMENTS PIC X(50).

```

Figure 27:

Figure 4. Breadcrumb trail in VSCode

Clicking on any of the items in the breadcrumb trail will highlight that element of the code in the editor, quickly showing you the location of that element within the code. It will also show a view of the code in a pop-up window, shown in Figure 5. , similar to the outline view previously discussed.

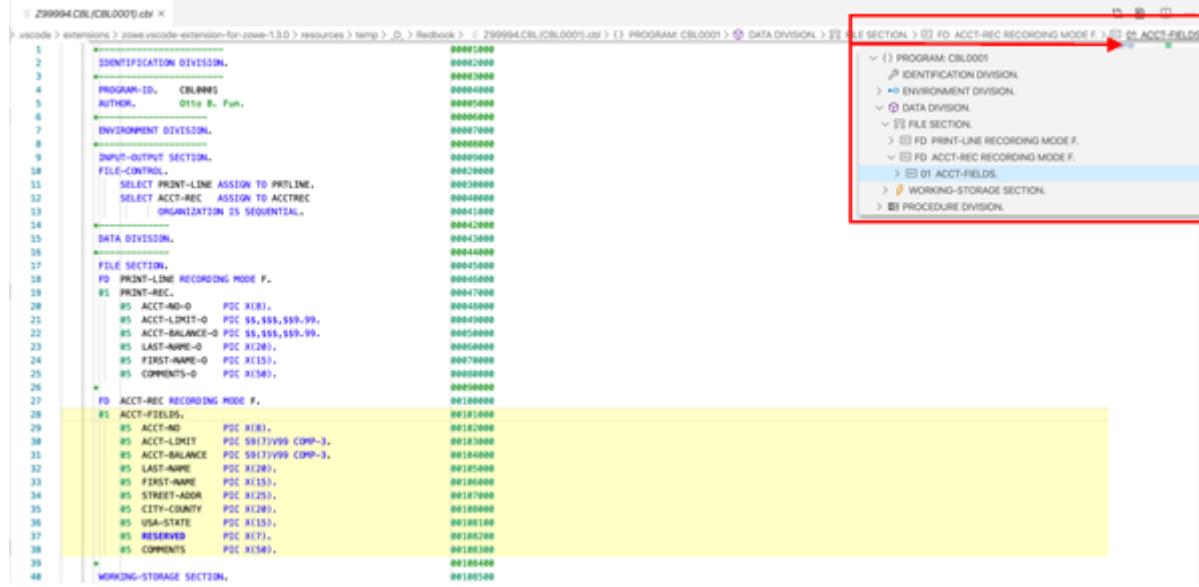


Figure 28:

Figure 5. Pop-up view of code via breadcrumb selection

3.3.3 Jump to declaration / reference

As you browse through code you will come across COBOL PERFORM statements or variable references. Often you will need to navigate to the definition of that paragraph or variable to follow the execution of the code. At line 50 of CBL0001 we see a set of perform statements. Place the cursor within the name, READ-RECORD, on line 51, right click and select **Go to Definition**. The editor then navigates to the READ-RECORD paragraph on line 62. Instead of the right click, the same function can be reached by using the F12 key.

“Go to References” does the reverse of this operation and allows you to navigate from the definition of a paragraph or variable to all the places within the application that reference that paragraph or variable. To demonstrate this, navigate to line 62 of CBL0001, which again is the declaration of the READ-RECORD paragraph. To see all of the places where this paragraph is called, right click and select **Go to References**, or hit the key combination **SHIFT+F12**. This will summon a new pop up dialog which shows all the references to this paragraph in the code, shown in Figure 6.

Note: If **SHIFT+F12** does not work for your machine, you may need to use the key combination, **Fn+F12** instead.

Figure 6. Finding paragraph/variable references in VSCode

3.4 Code-completion

Code completion isn't exactly a new concept in most IDEs. For example, the Eclipse editor has provided auto-completion for Java APIs for a long time. The same key combination, **CTRL+SPACE**, triggers this

```

57   CLOSE-STOP,
      CLOSE ACCT-REC,
      CLOSE PRINT-LINE,
      STOP RUN,
      READ-RECORD.
      READ-NEXT-RECORD.
      READ-RECORD.
      PERFORM READ-RECORD
      PERFORM UNTIL LASTREC = 'Y'
      PERFORM WRITE-RECORD
      PERFORM READ-RECORD
      END-PERFORM.
      CLOSE-STOP,
      CLOSE ACCT-REC,
      CLOSE PRINT-LINE,
      STOP RUN,
      READ-RECORD.
      READ ACCT-REC
      AT END MOVE "Y" TO LASTREC
      END-READ.
      WRITE-RECORD.

```

Figure 29:

auto-completion function while you are coding and can be used to help guide you through COBOL syntax and CICS, IMS API calls.

3.4.1 COBOL reserved word completion

As you are typing a COBOL reserved word, you can type **CTRL+SPACE** and the IBM Z Open Editor will present, in a pop-up, a list of possible COBOL reserved words that you might be trying to use. Using the cursor keys or the mouse allows you to select the correct recommended keyword and press enter to select it and the rest of the reserved word will be entered for you, aka auto-completed!

3.4.2 Variable completion

The same technique can be applied to variable completion. This can be particularly useful when you are referencing a variable that exists multiple times within different structures. In these cases, auto-completion can help you identify the variable you want to use. As an example, create a new line within the WRITE-RECORD paragraph. On the new line, enter the code `MOVE ACCT-BA` and then press **CTRL+SPACE** to invoke code auto-completion. You should see a pop up similar to the one shown in Figure 7. below.

Figure 7. Auto-completion in VSCode

You can see that not only is the variable `ACCT-BALANCE` prompted as a potential candidate, but it also presents `ACCT_BALANCE IN ACCT-FIELDS`.

3.4.3 CICS, MQ, DB2 API completion

The auto-completion of variables also extends to the CICS and DB2 APIs, known as EXEC CICS and EXEC SQL statements. Although COBOL programming for DB2 and CICS is not a primary focus here, note that if you find yourself programming for either of these APIs that the capability is available.

3.5 Refactoring code

Working with source code is seldom just about the initial creation, during a programs life cycle it will be changed and re-worked we often call this work refactoring. This section explores renaming variables and handling errors.

Z99994.CBL(CBL0001).cbl

```

34      05 STREET-ADDR    PIC X(25).          00107000
35      05 CITY-COUNTY   PIC X(20).          00108000
36      05 USA-STATE     PIC X(15).          00108100
37      05 RESERVED      PIC X(7).           00108200
38      05 COMMENTS      PIC X(50).          00108300
39
40      *WORKING-STORAGE SECTION.
41      01 FLAGS.        PIC X VALUE SPACE. 00108400
42      05 LASTREC       PIC X.              00108500
43
44      *PROCEDURE DIVISION.
45
46      OPEN-FILES.
47      OPEN INPUT  ACCT-REC.                00109000
48      OPEN OUTPUT PRINT-LINE.             00110000
49
50      READ-NEXT-RECORD.
51      PERFORM READ-RECORD.               00111000
52      PERFORM UNTIL LASTREC = 'Y'.
53      PERFORM WRITE-RECORD.              00112000
54      PERFORM READ-RECORD.               00112200
55      END-PERFORM.                    00112800
56
57      CLOSE-STOP.
58      CLOSE ACCT-REC.                  00113000
59      CLOSE PRINT-LINE.                00114000
60      STOP RUN.                      00115000
61
62      READ-RECORD.
63      READ ACCT-REC.                  00116000
64      AT END MOVE 'Y' TO LASTREC.    00117000
65      END-READ.                      00118000
66
67      WRITE-RECORD.
68      MOVE ACCT-NO      TO ACCT-NO-0.   00119000
69      MOVE ACCT-BA      TO ACCT-BA-0.   00120000
70
71      MOVE ACCT-LIMIT  TO ACCT-LIMIT-0. 00121000
72      MOVE ACCT-BALANCE TO ACCT-BALANCE-0. 00122000
73      MOVE LAST-NAME   TO LAST-NAME-0.  00123000
74      MOVE FIRST-NAME  TO FIRST-NAME-0. 00124000
75      MOVE COMMENTS    TO COMMENTS-0.   00125000
76
77      WRITE PRINT-REC.                00126000

```

ACCT-BALANCE

(e) ACCT-BALANCE IN ACCT-FIELDS

(e) ACCT-BALANCE-0

(e) ACCT-BALANCE-0 IN PRINT-REC

Figure 30:

3.5.1 Renaming variables

During maintenance of existing code, you might need to refactor variable names or paragraph names. Doing this manually can be a painful process, as you probably need to update both the original declaration and all the references within the source code. Fortunately, there is a function for that, let's work through an example. Back in CBL0001 hit **CTRL+G** to bring up the go to line function and go to line 29. This is the declaration of the variable ACCT-NO. Right click on the variable and select “**Find All References**”. From this we can see that apart from the declaration, the variable is also referenced on line 68. So, if we rename the variable, we probably need to update this reference as well. To perform the rename, ensure that the cursor is on the variable and then press **SHIFT/Fn+F2**. This will bring up a small pop-up asking you to provide a new variable name, as shown in Figure 8. Enter ACCT-NO-TEST and press **enter**.

The screenshot shows the IBM Z Open Editor interface. On the left, a sidebar displays 'REFERENCES: RESULTS (2)' and '2 results in 1 file' for 'Z99994.CBL(CBL0001).cbl'. The main editor area shows a COBOL program. At line 29, there is a declaration: '05 ACCT-NO PIC X(8).'. A red box highlights this declaration. At line 68, there is a reference: '05 ACCT-NO TEST'. Another red box highlights this reference. A tooltip 'Enter to Rename, ⌘Enter to Preview' is visible over the reference line. The code continues with FD, ACCT-REC RECORDING MODE F., and various data items like LAST-NAME, FIRST-NAME, STREET-ADDR, CITY-COUNTY, USA-STATE, and COMMENTS. The bottom section shows the WORKING-STORAGE SECTION with FLAGS and LASTREC.

Figure 31:

Figure 8. Renaming variables

You will note that both the declaration of the variable and the reference on line 68 have been updated to the new value. As stated previously, the same process also works for paragraph names. For example, go ahead and refactor the name of the paragraph READ-RECORD to be READ-NEW-RECORD.

3.5.2 Handling errors

The IBM Z Open Editor plugin also provides a level of syntax checking for local source code. Although not as thorough as the compiler, it is a method of quickly identifying basic errors in your code before submitting it for compilation. To demonstrate, let's create an error and then see how the editor shows it to us. First, open the problems view by selecting **View** and then **Problems** from the editor menu. The problems view should open at the bottom of the window, as depicted in Figure 9.

Figure 9. Problems view

Now we need to introduce an error into the code. After line 68, add the line:

```
MOVE ACCT-NO TO ACCT-N-0.
```

Note that this line incorrectly identifies the second variable, which doesn't exist. Once entering that line, you will notice that the invalid variable has been underlined in red to highlight it as an error. Also, the problems view has a new error. Clicking on the error will highlight the line of code at fault in the editor, shown in Figure 10. , allowing you to view the error directly.

Figure 10. Highlighting error in source code

```

      47  *PARA TRIMPFMT  PTC. PRINTF
      48  *COMMENTS=0  PDC X'581'.
      49
      50  FD ACCT-REC RECORDING MODE F.
      51  ACCT-FIELDS
      52
      53  05 ACCT-IDN          PDC X'831'.
      54  05 ACCT-LIMIT        PDC S'9C71100 COMP-3.
      55  05 ACCT-BALANCE      PDC S'9C71100 COMP-3.
      56  05 LAST-NAME        PDC X'26'.
      57  05 FIRST-NAME       PDC X'131'.
      58  05 STREET-ADDR       PDC X'261'.
      59
      60
      61
      62
      63
      64
      65
      66
      67
      68
      69
      70
      71
      72
      73
      74
      75
      76
      77

```

No problems have been detected in the workspace so far.

Figure 32:

```

      32  PERFORM UNTIL LASTREC = 'Y'
      33  PERFORM WRITE-RECORD
      34  PERFORM READ-RECORD
      35  END-PERFORM.
      36
      37  CLOSE-STOP.
      38  CLOSE ACCT-REC.
      39  CLOSE PRINT-LINE.
      40  STOP RUN.
      41
      42  READ-RECORD.
      43  READ ACCT-REC
      44  AT END MOVE 'Y' TO LASTREC
      45  END-READ.
      46
      47  WRITE-RECORD.
      48  MOVE ACCT-IDN    TO ACCT-IDN-0.
      49  MOVE ACCT-IDN    TO ACCT-ID-0.
      50
      51  MOVE ACCT-LIMIT   TO ACCT-LIMIT-0.
      52  MOVE ACCT-BALANCE  TO ACCT-BALANCE-0.
      53  MOVE LAST-NAME    TO LAST-NAME-0.
      54  MOVE FIRST-NAME   TO FIRST-NAME-0.
      55  MOVE COMMENTS     TO COMMENTS-0.
      56
      57  WRITE PRINT-REC.
      58
      59
      60
      61
      62
      63
      64
      65
      66
      67
      68
      69
      70
      71
      72
      73
      74
      75
      76
      77

```

Filter E.g.: test, **/*, **/*node_modules/**

Unable to resolve reference to ACCT-ID-0 symbol [68, 28]

Figure 33:

Now that you see where the error is located, it can now be corrected. As soon as the error has been rectified, the problem disappears from the problem view.

3.6 Summary

In this chapter you have been able to go through some of the editing features of the Z Open Editor for VSCode. These capabilities make editing COBOL, PL/I and JCL a lot friendlier and easier than some of the other editors in the market.

4 VS Code with Code4z Open-Source Extension Package

This section introduces the Code4z extension package, in particular the COBOL Language Support extension.

- **What is Code4z?**
- **Known File Extensions**
- **Syntax Highlighting and Coloring**
- **Syntax and Semantic Check**
- **Navigation of Code**
 - **Breadcrumb View**
 - **Outline View**
 - **Shortcuts**
 - **Go To Definition**
 - **Find All References**
- **Copybook Support**
- **Autocomplete**
- **Summary**

4.1 What is Code4z?

Code4z is an all-in-one, open-source mainframe extension package for Visual Studio Code. The Code4z package contains extensions which provide language support for COBOL and High Level Assembler language, a debugger for COBOL programs running on a CICS region, as well as tools which enable developers to access mainframe data sets and CA Endevor code repositories using the Visual Studio Code interface. This guide focuses on the COBOL Language Support extension. The Zowe Explorer extension is also included in the Code4z package.

The COBOL Language Support extension leverages the Language Server Protocol to provide autocomplete, highlighting, and diagnostic features for COBOL code. Together with Zowe Explorer, you can load COBOL code from a mainframe data set, and edit it leveraging the LSP features of the extension. Once you finish editing, you can save the file back on the mainframe, and store a copy locally.

The Code4z Extension Pack can be installed into VS Code by searching the Extensions Marketplace inside VS Code for “Code4z” and selecting install. The extension pack contains a number of extensions that can be leveraged when working with the mainframe, including the COBOL Language Support extension which provides similar functionality to the Z Open Editor extension discussed earlier. Therefore, ensure only one of these two extensions is enabled. Other extensions included in the pack will work with either COBOL Language Support or Z Open Editor. To see more detailed instructions on installing this extension, refer to “Installation of VSCode and Extensions”.

4.2 Known File Extensions

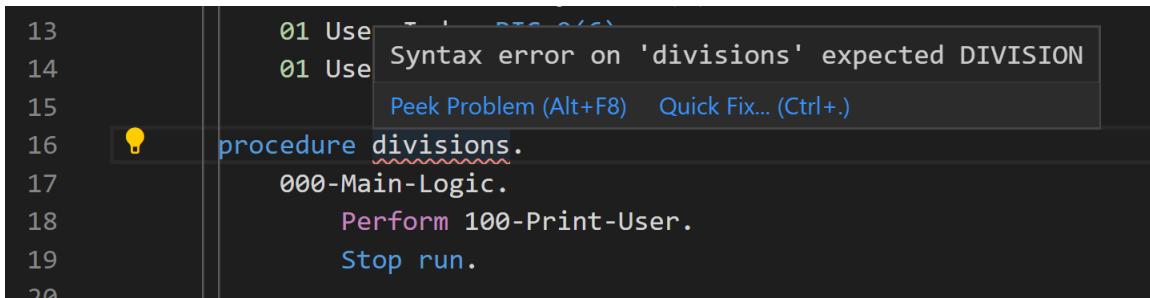
Code4z recognises files with the extension .COB and .CBL as COBOL files. This applies to both local files and files held in a PDS on the mainframe. COBOL Language Support features are automatically enabled when you open any file with an extension identifying it as a COBOL file.

4.3 Syntax Highlighting and Coloring

The Code4z package enables coloring of keywords, paragraphs, and variables in different colors to make the code easier to navigate.

4.4 Syntax and Semantic Check

The COBOL Language Support extension checks for mistakes and errors in COBOL code. The syntax check feature reviews the whole content of the code, highlights errors and suggests fixes.



A screenshot of a COBOL code editor. The code is as follows:

```
13      01 Use
14      01 Use
15      01 Use
16      procedure divisions.
17          000-Main-Logic.
18              Perform 100-Print-User.
19
20      Stop run.
```

A yellow lightbulb icon is positioned next to the word "procedure". A tooltip window is open at the cursor position, displaying the message "Syntax error on 'divisions' expected DIVISION". Below the message are two options: "Peek Problem (Alt+F8)" and "Quick Fix... (Ctrl+.)".

Figure 34:

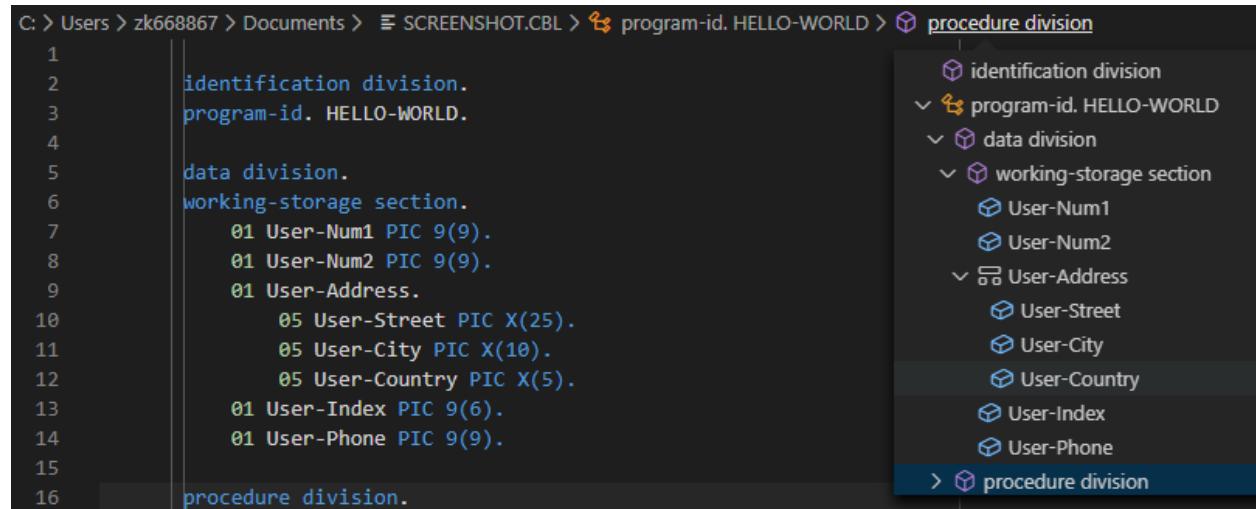
Figure 1. The syntax and semantic check feature highlights an error.

4.5 Navigation of Code

The Code4z package enables several features for ease of navigation through code.

4.5.1 Breadcrumb View

The breadcrumb view across the top of the editor shows where the current line of code exists within the structure of the COBOL source code. You can click each element on the bar to see that section of the code highlighted, or to select a code element within the section and navigate to it.



A screenshot of a COBOL code editor showing the breadcrumb view. The code is:

```
C: > Users > zk668867 > Documents > Screenshot.CBL > program-id. HELLO-WORLD > procedure division
1      identification division.
2      program-id. HELLO-WORLD.
3
4      data division.
5      working-storage section.
6          01 User-Num1 PIC 9(9).
7          01 User-Num2 PIC 9(9).
8          01 User-Address.
9              05 User-Street PIC X(25).
10             05 User-City PIC X(10).
11             05 User-Country PIC X(5).
12
13             01 User-Index PIC 9(6).
14             01 User-Phone PIC 9(9).
15
16      procedure division.
```

The breadcrumb view shows the following structure:

- identification division
- program-id. HELLO-WORLD
 - data division
 - working-storage section
 - User-Num1
 - User-Num2
 - User-Address
 - User-Street
 - User-City
 - User-Country
 - User-Index
 - User-Phone

Figure 35:

Figure 2. Breadcrumb view.

4.5.2 Outline View

The Code4z package enables the outline view tab in the sidebar. Outline view enables you to easily navigate through sections and individual variables and paragraphs.

4.5.3 Shortcuts

You can use the following shortcuts to navigate through your code:

- **CTRL+ALT+**, moves you one section back.
- **CTRL+ALT+**. moves you one section forward.
- **CTRL+ALT+D** takes you to the data division.
- **CTRL+ALT+P** takes you to the procedure division.
- **CTRL+ALT+W** takes you to the working storage section.

These commands are also available if you right click anywhere in the code.

4.5.4 Go To Definition

While your cursor is placed on a variable or paragraph name, you can press **F12** or **CTRL+click** to use the **Go To Definition** functionality to display the point in the code where the variable or paragraph is defined.

```
32     Display "User-Address : " USER-STREET.  
SCREENSHOT.CBL C:\Users\zk668867\Documents - Definitions (2)  
5     data division.  
6     working-storage section.  
7         01 User-Num1 PIC 9(9).  
8         01 User-Num2 PIC 9(9).  
9         01 User-Address.  
10        05 User-Street PIC X(25).  
          05 User-Street PIC X(25).  
          05 User-Street PIC X(25).
```

Figure 36:

Figure 3. Go To Definition shows the point at which the USER-STREET variable is first defined.

4.5.5 Find All References

The **Find All References** functionality (**SHIFT+ALT+F12**) highlights all references to a variable or paragraph and displays them in a list in the sidebar, so that you can easily navigate between them.

Figure 4. Find All References lists all references to the USER-STREET variable in the code.

4.6 Copybook Support

Copybooks are pieces of source code stored in separate data sets which are referenced in a program. The COBOL Language Support extension enables you to download all copybooks referenced in your program from the mainframe to a folder on your machine. In order for this feature to work, you need to set up and configure a Zowe CLI `zosmf` profile.

The COBOL Language Support extension helps to ensure that copybooks called in the code remain compatible through semantic analysis of keywords, variables, and paragraphs within copybooks, and ensures the consistency of code by defining variables and paragraphs across copybooks. The extension also helps to protect against unwanted errors caused by recursive or missing copybooks.

```

c: > Users > zk668867 > Documents > ≡ SCREENSHOT.CBL > 📁
9
• 10
11
12
13
14
15
16
17
18
01 User-Address.
    05 User-Street PIC X(25).
    05 User-City PIC X(10).
    05 User-Country PIC X(5).
01 User-Index PIC 9(6).
01 User-Phone PIC 9(9).
procedure division.
    000-Main-Logic.
        Perform 100-Print-User.

```

Figure 37:

The **Go To Definition** and **Find All References** functionalities are also extended to work for occurrences of variables and paragraphs from copybooks called in the program as well as from the program itself.

4.7 Autocomplete

The COBOL Language Support extension provides live suggestions while you type for COBOL keywords, as well as variables and paragraphs which are already referenced in the code or in copybooks used by the program.

```

26 Move 'CZ' To USER-COUNTRY.
27
28 Move 11000 To USER-INDEX.
29
30 Move 770778774 To USER-.
31
32
33
34
35
36
37
[?] USER-ADDRESS
[?] USER-CITY
[?] USER-COUNTRY
[?] USER-INDEX
[?] USER-NUM1
[?] USER-NUM2
[?] USER-PHONE
[?] USER-STREET

```

Figure 38:

Figure 5. Autocomplete lists possible variables and keywords beginning with the typed string in a list.

4.8 Summary

In this chapter you have been introduced to all the COBOL language support features of the Code4z package of open-source extensions for VS Code.

5 Zowe CLI and Zowe CLI Plug-ins

In this chapter we will explain what a CLI is and why you would use it, how to use Zowe CLI interactively, how to abstract CLI commands into useful scripts, and how Zowe CLI enables the use of familiar open source tooling while developing COBOL applications on the mainframe.

- **What is a CLI and why would you use it?**
- **What is Zowe CLI?**
- **Zowe CLI interactive use**
 - Installing Zowe CLI
 - Interactive Help
 - Zowe Profiles
 - Interacting with z/OS Data Sets
 - Interacting with z/OS Jobs
- **Automating tasks using Zowe CLI**
 - Automated Job Submission
 - Using Other Programming Languages and Continuous Integration
 - Additional Examples
- **The world of modern open source tooling**
- **Summary**

5.1 What is a CLI and why would you use it?

CLI stands for Command Line Interface. It is a program that allows for user interaction through text based input. In the early days of computing, command line interfaces were the only means to interact with operating systems. The invention of the mouse and development of graphical user interfaces led to the experience we are familiar with today. Well-designed GUIs certainly yield an improved interactive experience. However, CLIs are still heavily used today and are very powerful. Windows shell and bash are common examples of terminals where command line tools are run interactively.

If well-designed GUIs yield an improved interactive experience, why would you use a CLI? Simply put, automation. Command line interfaces can be used interactively allowing for quick exploration of available commands. They are also usually self-guided and some even offer modern help displays by launching content in a browser. But, they are also programmatic interfaces where command sequences and tasks can be easily abstracted into scripts.

5.2 What is Zowe CLI?

Zowe CLI is an open source CLI for the mainframe. It is a tool that can be run on Windows, Linux, and Mac offering a means to interact with the mainframe from an environment where modern open source tooling is available. Cloud platforms like Amazon Web Services, Azure, and Google Cloud Platform all provide heavily used CLIs. The Zowe CLI helps make interacting with the mainframe like interacting with other cloud services.

At its core, Zowe CLI provides remote interaction with z/OS data sets & jobs, Unix System Services files, TSO and Console commands, and provisioning services. Zowe CLI is also an extensible technology and numerous plug-ins exist that extend its reach to z/OS subsystems and vendor software.

Zowe CLI is a bridge tool between distributed systems and the mainframe. Pick your favorite language or open source tool and leverage it for mainframe development with the assistance of the Zowe CLI. Want to

develop automation in Python? Want to write tests in Node? Want to run Jenkins pipelines for continuous integration? Want to use open source testing frameworks like Mocha or Facebook’s Jest? Want to leverage code quality tools like SonarQube? Go for it!

CLIs are useful for automating repeated tasks. For mainframe COBOL apps, Zowe CLI can help you automate your build, deployment, and testing processes. Check out this [blog](#) for more info and the sample code that made it possible! Zowe CLI can also help you to automate administrative tasks.

Most IDEs have integrated terminals as well so the CLI can be leveraged from your favorite distributed development environment, including VS Code!

5.3 Zowe CLI interactive use

The Zowe CLI can be leveraged for quick exploration of z/OS services as well as issuing commands that are not yet available in your IDE of choice. Before developing automation, it is common to first accomplish a commonly repeated task from the CLI interactively.

5.3.1 Installing Zowe CLI

The Zowe CLI is a node package and is one of over 1.2 million node packages available on the public npm registry. After Node.js and npm are installed on the client machine, the core CLI can be installed by simply issuing `npm install -g @zowe/cli@zowe-v1-lts`. There is an alternative installation method if your workstation does not have access to the public registry. More details on installing Zowe CLI and Zowe CLI plug-ins are provided in a future section titled “Installation of Zowe CLI and plug-ins”.

5.3.2 Interactive Help

To get started, you can simply open a terminal and issue `zowe`. This will yield the top level help.

Figure 1. Zowe CLI Help

In the example above, multiple extensions are installed. The structure of commands is `zowe <group> <action> <object>` followed by various parameters and options specific to the command. For example, a valid command is `zowe files list data-set "HLQ.*"`. This command will list data-sets matching a pattern of “HLQ.*”. You can append `-h` to any command to find out more information. Frequently referring to the help can be difficult and time consuming so if your environment has access to a web browser, simply append `--help-web` or `--hw` to any command to launch interactive web help.

Figure 2. Zowe CLI Web Help

Don’t have the CLI installed yet? You can also check out a copy of the web help for the core Zowe CLI and Zowe plug-ins [here](#).

5.3.3 Zowe Profiles

Zowe client technologies like Zowe CLI and the Zowe Explorer VS Code Extension store connection information in files commonly known as profiles. This provides convenience as after profiles for services are created, users do not have to constantly provide this information. For the secure storage of credentials, there is the Secure Credential Store plug-in which is discussed more in a later section titled “Installation of Zowe CLI and plug-ins”. The Secure Credential Store provides a means to store creds in the operating system’s secure credential vault.

When creating profiles you can also specify the `prompt*` keyword to be prompted for your username and password so they will be masked on the command line. Figure 3 shows a sample command to create a `zosmf` profile. This will eliminate the need to provide these details on future commands.

```

$ zowe
DESCRIPTION
-----
Welcome to Zowe CLI!

Zowe CLI is a command line interface (CLI) that provides a simple and
streamlined way to interact with IBM z/OS.

For additional Zowe CLI documentation, visit https://zowe.github.io/docs-site

For Zowe CLI support, visit https://zowe.org

USAGE
-----
zowe <group>

Where <group> is one of the following:

GROUPS
-----
cics           Interact with IBM CICS programs and
               transactions.
config         Manage configuration and overrides
db2           Interact with IBM Db2 for z/OS
endevor        CA Endevor SCM plug-in for listing Endevor
               environment information, working with elements
               and packages located in specified Endevor
               instance.
file-master-plus | fmp   Advanced file management and data manipulation
ims            Interact with IBM IMS programs and transactions.
mq             Interact with IBM MQ for z/OS.
ops            Zowe CLI Plug-in for CA® OPS/MVS®
plugins        Install and manage plug-ins
profiles       Create and manage configuration profiles
provisioning | pv  Perform z/OSMF provisioning tasks
secure-credential-store | sscs Work with securely stored credentials
zos-console | console Issue z/OS console commands and collect
               responses
zos-extended-files | files-e Extended capabilities to manage z/OS datasets
zos-extended-jobs | jobs-e CA Brightside plugin for z/OS jobs
zos-files | files  Manage z/OS data sets
zos-ftp | zftp    z/OS Files and jobs via FTP
zos-jobs | jobs  Manage z/OS jobs
zos-tso | tso    Interact with TSO
zos-uss | uss    Issue z/OS USS commands and receive responses
zos-workflows | wf  Create and manage z/OSMF workflows
zosmf          Interact with z/OSMF

```

Figure 39:

The screenshot shows the Zowe CLI web interface. On the left, there's a sidebar with a search bar and tabs for 'Tree View' (which is selected) and 'Flat View'. The main content area has a header 'zowe' and a 'Groups' section. Below the 'Groups' section, there's a bulleted list of available groups and their descriptions.

Group	Description
cics	Interact with IBM CICS programs and transactions.
config	Manage configuration and overrides
db2	Interact with IBM Db2 for z/OS
endevor	CA Endevor SCM plug-in for listing Endevor environment information, working with elements and packages located in specified Endevor instance.
file-master-plus fmp	Advanced file management and data manipulation
ims	Interact with IBM IMS programs and transactions.
mq	Interact with IBM MQ for z/OS.
ops	Zowe CLI Plug-in for CA® OPS/MVS®
plugins	Install and manage plug-ins
profiles	Create and manage configuration profiles
provisioning pv	Perform z/OSMF provisioning tasks
secure-credential-store sscs	Work with securely stored credentials
zos-console console	Issue z/OS console commands and collect responses
zos-extended-files files-e	Extended capabilities to manage z/OS datasets
zos-extended-jobs jobs-e	CA Brightside plugin for z/OS jobs
zos-files files	Manage z/OS data sets
zos-ftp zftp	z/OS Files and jobs via FTP
zos-jobs jobs	Manage z/OS jobs
zos-tso tso	Interact with TSO
zos-uss uss	Issue z/OS USS commands and receive responses
zos-workflows wf	Create and manage z/OSMF workflows
zosmf	Interact with z/OSMF

Figure 40:

```

user@ubuntu-base:~/Zowe-Workshop$ zowe profiles create zosmf learningSystem --host 192.86.32.250 --port 10433 --ru false --user prompt* --pass prompt*
"user" Description: Mainframe (z/OSMF) user name, which can be the same as your TSO login.
Please enter "user":
"password" Description: Mainframe (z/OSMF) password, which can be the same as your TSO password.
Please enter "password":
Profile created successfully! Path:
/home/user/.zowe/profiles/zosmf/learningSystem.yaml

host:          192.86.32.250
port:          10433
user:          managed by @zowe/secure-credential-store-for-zowe-cli
password:      managed by @zowe/secure-credential-store-for-zowe-cli
rejectUnauthorized: false

Review the created profile and edit if necessary using the profile update command.

```

Figure 41:

Figure 3. Zowe CLI z/OSMF Profile Creation Command

5.3.4 Interacting with z/OS Data Sets

Zowe CLI provides a significant suite of z/OS data set interaction functionality. See the following figures for details on available actions and a sample list command.

```

GROUPS
-----
copy | cp      Copy a data set
create | cre   Create data sets
delete | del   Delete a data set or Unix System Services file
download | dl  Download content from z/OS data sets and USS files
hMigrate | hmigr Migrate data sets.
invoke | call  Invoke various z/OS utilities
list | ls     List the details for data sets and the members in the data
              sets
mount          Mount file systems
rename         Rename a data set or member.
unmount | umount Unmount file systems
upload | ul    Upload the contents of a file to z/OS data sets

```

Figure 42:

Figure 4. Zowe CLI zos-files actions

Figure 5. Sample Zowe CLI zos-files list ds command

5.3.5 Interacting with z/OS Jobs

Zowe CLI provides a significant suite of z/OS jobs interaction functionality. See the following figures for details on available actions and a sample job submission command.

Figure 6. Zowe CLI zos-jobs actions

Figure 7. Sample Zowe CLI zos-jobs submit ds command

```
user@ubuntu-base:~/Zowe-Workshop$ zowe files list ds "CUST005.*"
CUST005.BRIGHT.MARBLES.COBOL
CUST005.BRIGHT.MARBLES.JCL
CUST005.BRIGHT.MARBLES.PARMLIB
CUST005.ISPF.ISPPROF
CUST005.MARBLES.JCL
CUST005.MTE.JCL
```

Figure 43:

```
GROUPS
-----
cancel | can   Cancel a job
delete | del   Delete a job
download | dl  Download job output
list | ls    List jobs and spool files
submit | sub  Submit z/OS jobs
view | vw    View details of a z/OS job
```

Figure 44:

```
user@ubuntu-base:~/Zowe-Workshop$ zowe jobs submit ds "CUST005.MARBLES.JCL(MARBDB2)" --wait-for-output
jobid: JOB02395
retcode: CC 0000
jobname: MARBDB2
status: OUTPUT
```

Figure 45:

5.4 Automating tasks using Zowe CLI

Running commands interactively is a great way to learn the capabilities of the Zowe CLI. However, creating custom automation for your commonly repeated tasks and making use of valuable development tooling is where significant value lies. For COBOL development, significant time can be spent reviewing compiler output and testing programs. These repetitive tasks are excellent candidates for automation.

5.4.1 Automated Job Submission

Let's investigate automating submitting a job and verifying the return code is 0. Of course, we could also parse the spool output for specific messages of interest but we will keep it simple for now. For this sample, we will leverage Node.js to develop a new automation suite. To start, I will create a package.json file to make it easy for others to manage and install the project. It will contain the list of dependencies for my project as well as the automation tasks I will develop. A quick way to create a package.json is to issue `npm init` and answer the prompts. Once created I will add a submitJob task. You can add whatever automation you want here. My final package.json is shown in the next figure. You can learn more about package.json files [here](#).

```
{  
  "name": "automation",  
  "version": "1.0.0",  
  "description": "Sample automation project",  
  "main": "index.js",  
  "scripts": {  
    "submitJob": "node ./submitJobAndVerifySuccess"  
  },  
  "author": "MikeBauerCA",  
  "license": "MIT"  
}
```

Figure 46:

Figure 8. Sample package.json

Then I will create a config.json file to store all the variables I may wish to change for my project. In this case, we will set the job to submit and the maximum allowable return code for that job.

```
{  
  "job": "CUST005.MARBLES.JCL(MARBDB2)",  
  "maxRC": 4  
}
```

Figure 47:

Figure 9. Sample config.json

Next we will write our automation. The Zowe CLI was built with scripting in mind and can output responses in JSON format which can be easily parsed.

```

user@ubuntu-base:~/automation$ zowe jobs submit data-set "CUST005.MARBLES.JCL(MARDBB2)" --wait-for-output --rfj
{
  "success": true,
  "exitCode": 0,
  "message": "Submitted JCL contained in \"dataset\": \"CUST005.MARBLES.JCL(MARDBB2)\"",
  "stdout": "\u001b[33mjobid: \u001b[39m JOB02486\n\u001b[39mretcode: \u001b[39mCC 0000\n\u001b[33mjobname: \u001b[39m OUTPUT\n",
  "stderr": "",
  "data": {
    "owner": "CUST005",
    "phase": 20,
    "subsystem": "JES2",
    "phase-name": "Job is on the hard copy queue",
    "job-correlator": "J00024865.....D7C70323.....",
    "type": "JOB",
    "url": "https://[REDACTED]:443/zosmf/restjobs/jobs/J00024865.....D7C70323.....%3A",
    "jobid": "JOB02486",
    "class": "A",
    "files-url": "https://[REDACTED]:443/zosmf/restjobs/jobs/J00024865.....D7C70323.....%3A/files",
    "jobname": "MARDBB2",
    "status": "OUTPUT",
    "retcode": "CC 0000"
  }
}

```

Figure 48:

Figure 10. Sample Zowe CLI response format JSON output

Now, instead of issuing this command and reviewing it to see if the retcode is less than or equal to 4, we want to automate it. See the implementation in a node script below.

```

const { exec } = require('child_process'),
      config = require('./config.json');

exec(`zowe jobs submit data-set ${config.job} --wait-for-output --rfj`, (err, stdout, stderr) => {
  if (err) {
    //some err occurred
    console.error(err)
  } else if (stderr) {
    console.log(new Error(`\nCommand:\n${command}\n\nstderr:\n${stderr}`));
  } else {
    data = JSON.parse(stdout).data;
    retcode = data.retcode;

    //retcode should be in the form CC nnnn where nnnn is the return code
    if (retcode.split("\n")[1] <= config.maxRC) {
      console.log("Job completed successfully");
    } else {
      console.log(new Error(`Job did not complete successfully. Additional diagnostics: ${JSON.stringify(data,null,1)})`));
    }
  }
});

```

Figure 49:

Figure 11. Sample code to submit job and verify output is less than or equal to a maximum allowable RC

I had to make the investment to write this automation but for future job submissions I can simply issue `npm run submitJob`. IDEs like VS Code can visualize these tasks making my commonly repeated tasks as easy as clicking a button :). This job could compile, link, and/or run a COBOL program.

Figure 12. Vizualization of npm script and sample run

More advanced code automating the compilation, deployment to test environment, and testing of a COBOL CICS application is described in this [blog](#).

```

NPM SCRIPTS
  package.json
    submitJob
      node ./submitJobAndVerifySuccess

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
> Executing task: npm run submitJob <

> automation@1.0.0 submitJob /home/user/automation
> node ./submitJobAndVerifySuccess

Job completed successfully

Terminal will be reused by tasks, press any key to close it.

```

Figure 50:

5.4.2 Using Other Programming Languages and Continuous Integration

Another good example of automating tasks using Zowe CLI is when you want to integrate other programming languages into your COBOL development. Similar to 3.4.1, you can use other languages such as Typescript to write a COBOL program generator and use Zowe CLI to create a “one-click” process for creating your program. The figure below is a representation of that “one-click” automated process where several tasks are executed such as creating your COBOL program, uploading it in mainframe, compiling it and running your JCL to test it.

Figure 13. “One Click” COBOL build process

You can then level-up this process by leveraging a CI/CD pipeline. What is a CI/CD pipeline? It is an automated way of building, testing, and deploying your application and you can do the same with your COBOL development. The figure below shows the pipeline for the same automated tasks that we did earlier.

Figure 14. CI/CD pipeline of the “one click” COBOL build process

To know more about this topic, check [this](#) out.

5.4.3 Additional Examples

If you are looking for an example on how to use Zowe Explorer and Zowe CLI with Db2 Stored Procedures, check out this [blog](#).

If you are interested in using open source tools in your development, you can review this [blog](#) where it talks about using Zowe CLI to leverage static code analysis tools when developing COBOL applications.

For additional blogs and articles on leveraging Zowe technologies, check out <https://medium.com/zowe/users/home>.

5.5 The world of modern open source tooling

We have only scratched the surface of using modern tools and languages for mainframe development and incorporating mainframe applications into enterprise DevOps pipelines. As a bridge tool, the Zowe CLI enables the use of a plethora of tools being developed by an enormous community for mainframe development. If you are new to mainframe, hopefully this offers some familiarity as you transition into this space. If you are an experienced mainframe, hopefully you find time to give some of these available technologies a try to see if they can help you.

```

> template@1.0.0 build C:\Users\jp669971\Documents\GitHub\Community\Cobol-Made-Easy
> npm run generate && npm run upload && npm run compile:cobol && npm run run:job

> template@1.0.0 generate C:\Users\jp669971\Documents\GitHub\Community\Cobol-Made-Easy
> node ./scripts/lib/append && node ./scripts/lib/generate

Created COBOL Template
Added Process
Added Closing
Generated custom JCL to ./build/compiler.jcl
Generated COBOL pgm to ./build/source.cbl
Generated run job to ./build/runjob.jcl

> template@1.0.0 upload C:\Users\jp669971\Documents\GitHub\Community\Cobol-Made-Easy
> zowe zos-files upload file-to-data-set "./build/source.cbl" "Z40607.SOURCE(CBLSRC)" --zosmf-p ztrial

success: true
from:   C:\Users\jp669971\Documents\GitHub\Community\Cobol-Made-Easy\build\source.cbl
to:     Z40607.SOURCE(CBLSRC)

file_to_upload: 1
success: 1
error: 0
skipped: 0

Data set uploaded successfully.

> template@1.0.0 compile:cobol C:\Users\jp669971\Documents\GitHub\Community\Cobol-Made-Easy
> zowe jobs submit lf "./build/compiler.jcl" --directory "./output" --zosmf-p ztrial

██████████ | 30% 0 | Waiting for JOB07745 to enter OUTPUT

```

Figure 51:

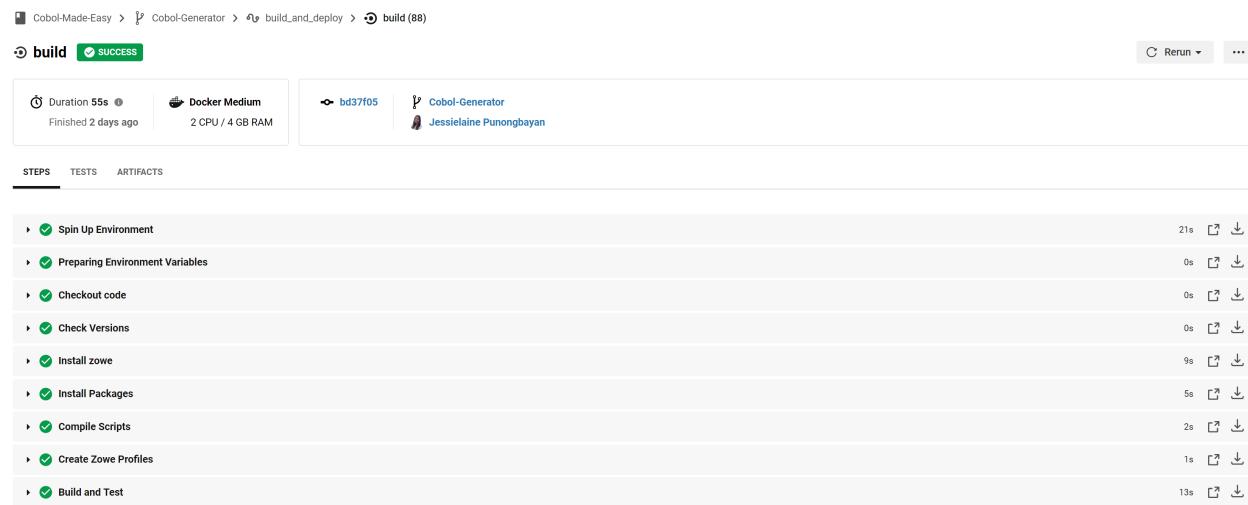


Figure 52:

5.6 Summary

As both a user and programmatic interface, command line interfaces offer significant value in simplifying complex repeatable processes into single tasks. CLIs are commonly used when developing on popular cloud platforms like Amazon Web Services. The Zowe CLI is the CLI for the mainframe that has been extended via numerous plug-ins. Zowe CLI acts as a bridge tool enabling the use of distributed development tooling while working with mainframe applications. Numerous resources and articles are available for using Zowe CLI to create custom automation, build CI pipelines, and incorporate static analysis into your COBOL development processes. Development tooling created by the distributed open source community can now be effectively leveraged for mainframe development.

6 Installation of VSCode and extensions

This chapter covers all aspects of download and installation of Visual Studio (VS) Code and any prerequisites that are needed. It includes:

- **Install prerequisites**
 - Install node.js
 - Install Java SDK
- **Install VSCode**
- **Install VSCode extensions**
 - Zowe Explorer
 - IBM Z Open Editor
 - Code4z
- **Summary**

6.1 Install prerequisites

This section will cover the necessary steps and information to download and install the prerequisites needed for the subsequent labs within this book. This software is needed for one or more of the applications we will be utilizing in our labs throughout the book. The prerequisites include:

- Install node.js
- Install Java SDK

6.1.1 Install node.js

1. Check for node.js installation and verify that the version number is v8 or higher.

Open your workstation's version of the command prompt (called Terminal on Mac OS X). Once the command prompt is open, use the command in Example 1. to check if your workstation currently has a version of node.js installed.

```
C:\Users\User> node -v  
v12.16.1
```

Example 1. Node.js version

If you do not see a version number after you submit the command, you do not have node.js installed, or if it shows a version less than v8, you should continue following these instructions. If you do see a version number and it is v8 or higher, you can move on to section Install Java SDK.

2. If node.js version is less than v8, or node isn't installed at all.

Updating node.js to the appropriate version number is a relatively simple process because the installer takes care of most of the “heavy lifting”. All you will need to do is visit the Node.js download site, provided below and follow the download and installation instructions for your specific workstation platform. Do this same process if you do not already have node.js installed.

<https://nodejs.org/en/download/>

This process will install the latest versions of Node.js and the node package manager (npm) and overwrite any older version files in your system. This removes the step of needing to manually uninstall the previous versions beforehand.

- Once completed, verify the installation and proper version number, as shown previously in Example 1.

Note : The version numbers in our examples are provided purely for reference and may not reflect the latest versions of the software.

6.1.2 Install Java SDK

- Check for Java installation and verify that the version number is v8 or higher.

Open your workstation's version of the command prompt, if not already open. Once the command prompt is open, use the command in Example 2. to check if your workstation currently has a version of Java installed. Java SDK 8 is the preferred version for these labs, however, any versions higher than that will suffice.

```
C:\Users\User> java -version

java version "1.8.0_241"

Java(TM) SE Runtime Environment (build 1.8.0_241-b07)

Java HotSpot(TM) 64-Bit Server VM (build 25.241-b07, mixed mode)
```

Example 2. Java version

If you do not see a version number after you submit the command, you do not have Java installed or if it shows a version less than v8, you should continue following these instructions. The display format of the version number for Java is slightly different than what is displayed for node.js. With Java, the second value in the displayed version number, i.e. the “8” in Example 2. , is the version number. So, our example is showing Java SDK version 8. If you do see a version number and it is v8 or higher, you can move on to section Install VSCode.

- If your version of Java displayed is less than v8, you need to uninstall the current version on your workstation and reinstall the correct version. Follow the link below to uninstall instructions that represent your workstation operating system (OS).

-Linux:

https://www.java.com/en/download/help/linux_uninstall.xml

-Mac:

https://www.java.com/en/download/help/mac_uninstall_java.xml

-Windows:

https://www.java.com/en/download/help/uninstall_java.xml

- Once Java is uninstalled from your workstation, you can click the Java JDK 8 download link below and follow the installation instructions for your specific OS.

<https://www.oracle.com/java/technologies/javase-jdk8-downloads.html>

- Verify the installation and proper version number as shown in Example 2.

Note : You will be prompted to register a new Oracle account in order to download the installation file, please do so. If you have an existing account, you may use that to log in and continue.

6.2 Install VSCode

If you do not already have VSCode installed on your workstation, please do so now by following the download and installation instructions at the link below:

<https://code.visualstudio.com/download>

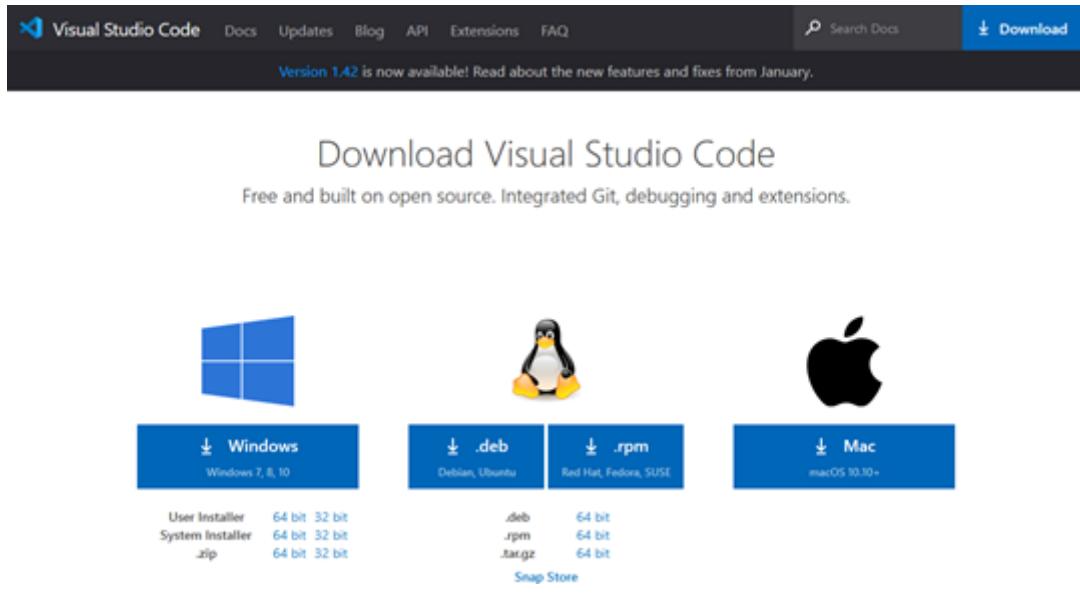


Figure 53:

Figure 1. VSCode download site

Note : Be sure to select the correct installation file for your workstations respective OS, shown in Figure 1.

6.3 Install VSCode extensions

This section introduces two VSCode extensions, Zowe Explorer and IBM Z Open Editor listed in Figure 2. , and instructions on how to install them.

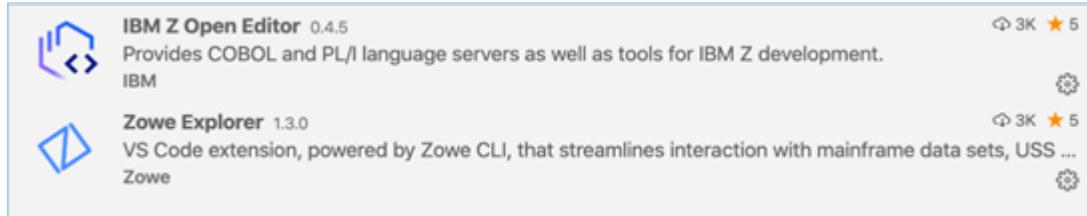


Figure 54:

Figure 2. VSCode required extensions

6.3.1 Zowe Explorer

Zowe is a new, and the first open source framework for z/OS and provides solutions for development and operations teams to securely manage, control, script and develop on the mainframe like any other cloud platform. Out of the box, the Zowe Explorer provides a lot of functionality allowing z/OS developers access to jobs, datasets and (USS) files on a z/OS server. Backed by the Zowe CLI and z/OSMF, developers now have powerful features that makes it easy to work with z/OS within the familiar VSCode environment. This extension can be used to edit COBOL and PL/I files opened on z/OS MVSTTM and USS using the Zowe

extension's Data Sets and USS views. It can even run JCL and lets you browse job spool files. For more information on Zowe Explorer and its interaction with z/OS please visit:

https://ibm.github.io/zopeneditor-about/Docs/zowe_interactwithzos.html

6.3.1.1 Install Zowe Explorer

Open VSCode and in the left side tool menu select **Extensions**. From there, in the “Search Extensions in Marketplace” search field, type **Zowe Explorer**. Search results will begin populating, select “**Zowe Explorer**” and click **install**, depicted in Figure 3.

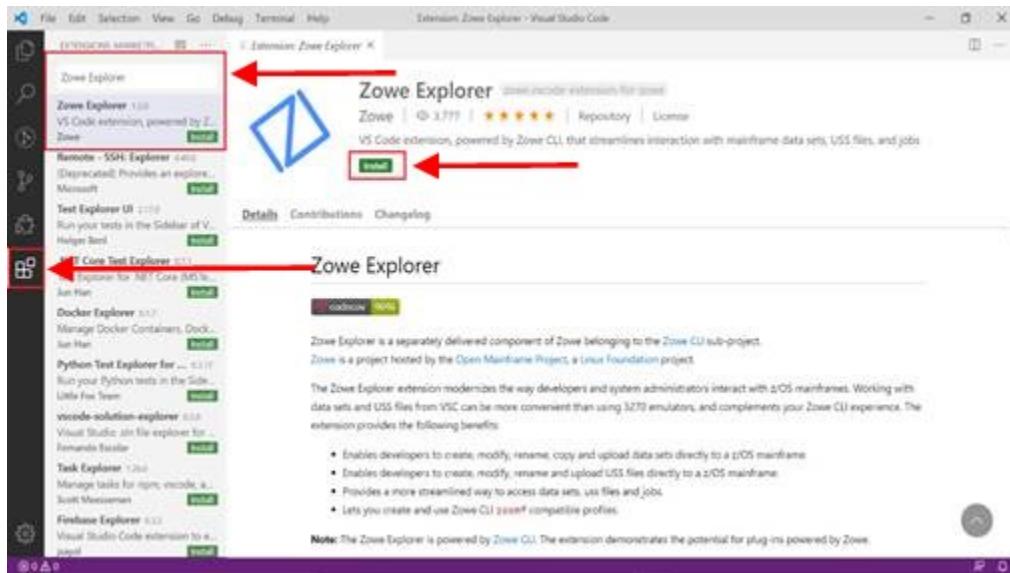


Figure 55:

Figure 3. Install Zowe Explorer in VSCode

The Zowe community have a number of on-line video that walk through the steps required to install, configure and operate the Zowe Explorer, see [Zowe Explorer VSC Extension \(part 1\)](#).

6.3.2 IBM Z Open Editor

IBM Z Open Editor brings COBOL and PL/I language support to Microsoft VSCode. It is one of the several next generation capabilities for an open development experience for z/OS®. It also works in association with the Zowe Explorer plugin. For more information on IBM Z Open Editor, please visit:

<https://ibm.github.io/zopeneditor-about/Docs/introduction.html#key-capabilities>

6.3.2.1 Install IBM Z Open Editor

Open VSCode and in the left side tool menu select **Extensions**. From there, in the “Search Extensions in Marketplace” search field, type **IBM Z Open Editor**. Search results will begin populating, select ** IBM Z Open Editor ** and click **install**, depicted in Figure 4.

Figure 4. Install IBM Z Open Editor in VSCode

Note : There may be some limitations with IBM Z Open Editor if running a 32-bit Java version on Windows.

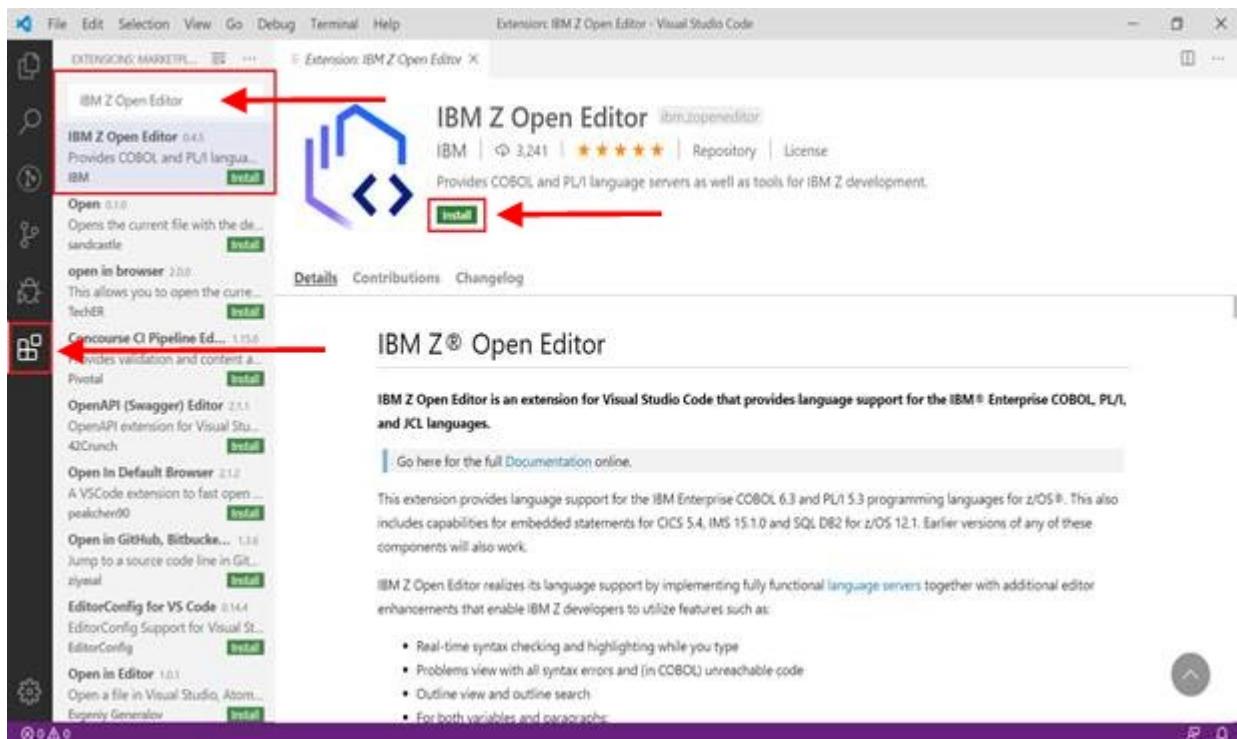


Figure 56:

6.3.3 Code4z

Code4z is an all-in-one, open-source mainframe extension package for developers working with z/OS applications, suitable for all levels of mainframe experience, even beginners. Mainframe application developers can use the Code4z package for a modern, familiar, and seamless experience, which helps to overcome some developers' reservations or concerns about the traditional mainframe user experience. To find out more about Code4z, please visit <https://github.com/BroadcomMFD/code4z>.

6.3.3.1 Install Code4z

Open VSCode and in the left side tool menu select **Extensions**. From there, in the “Search Extensions in Marketplace” search field, type **Code4z**. Search results will begin populating, select **" Code4z "** and click **install**.

The extension pack contains a number of extensions that can be leveraged when working with the mainframe, including the COBOL Language Support extension which provides similar functionality to the Z Open Editor extension. Therefore, ensure only one of these extensions is enabled. Extensions can be disabled within VS Code by locating the extension in the Extensions menu, clicking the settings gear, and selecting **Disable**. Other extensions included in the Code4z pack will work with either COBOL Language Support or Z Open Editor.

6.4 Summary

In this chapter you have been introduced to VSCode and some of the extension tools available to it. We have walked through the process of installing the pre-requisite software, Node.js and Java SDK, as well as VSCode, Zowe Explorer, IBM Z Open Editor and Code4z. You have also been briefly introduced to the utility of these

extensions in VSCode. In the subsequent chapters we will delve deeper into how and when to use them and get some practice through lab assignments.

7 Installation of Zowe CLI and Plug-ins

This chapter covers all aspects of the download and installation of Zowe CLI and Zowe CLI plug-ins.

- **Install prerequisites - Node.js**
- **Install Zowe CLI**
 - Public npm Registry
 - Package from Zowe.org
- **Install Zowe CLI Plug-ins**
 - Public npm Registry
 - Package from Zowe.org
- **Summary**

7.1 Install prerequisites - Node.js

Before installing Zowe CLI, please ensure an LTS version of Node v8.0 or higher is installed. Please refer back to the section titled “Install Node.js” if you have not already completed it. Please also verify that you have a version of Node Package Manager (npm) that is compatible with your version of Node.js. For a list of compatible versions, see <https://nodejs.org/en/download/releases/>. npm is included with the Node.js installation. Issue the command `npm --version` to view the version of npm that is installed.

7.2 Install Zowe CLI

There are two recommended methods for installing the Zowe CLI. If you have access to the public npm registry from your workstation, we recommend using that installation method as pulling updates is seamless. If you do not have access to this registry, we recommend downloading the package from zowe.org and installing from the bundled package.

7.2.1 Install from Public npm Registry

Issue the following command in your terminal (e.g. Command Prompt or if you are using VS Code, Terminal -> New Terminal):

```
npm install -g @zowe/cli@zowe-v1-lts
```

If the command returns an EACCESS error, refer to [Resolving EACCESS permissions errors when installing packages globally](#) in the npm documentation. If other issues are encountered in your environment, please review [known Zowe CLI issues](#) for solutions.

We also highly recommend installing the Secure Credential Store plug-in before using the CLI. The Secure Credential Store Plug-in for Zowe CLI lets you store your credentials securely in the default credential manager in your computer’s operating system. On Linux, libsecret will need to be installed.

If running Linux, please run the following command for your Linux distribution:

- Debian/Ubuntu: `sudo apt-get install libsecret-1-dev`
- Red Hat-based: `sudo yum install libsecret-devel`
- Arch Linux: `sudo pacman -S libsecret`

To install the Secure Credential Store Plug-in for Zowe CLI, issue the following command:

```
zowe plugins install  
@zowe/secure-credential-store-for-zowe-cli@zowe-v1-lts
```

User profiles, which contain connection information for interacting with various z/OS services, created after installing the plug-in will automatically store your credentials securely.

To securely store credentials in existing user profiles (profiles that you created prior to installing the SCS plug-in), issue the following command:

```
zowe sscs update
```

7.2.2 Install from Bundled Package

Navigate to [Zowe.org Downloads](#) and click the CLI Core button to download the core package which includes Zowe CLI and the Secure Credential Store plug-in. After accepting the EULA for Zowe, a package named `zowe-cli-package-v.r.m.zip` will be downloaded to your machine. Unzip the contents of `zowe-cli-package-v.r.m.zip` to a preferred location on your machine.

Open your terminal (e.g. Command Prompt or if you are using VS Code, Terminal -> New Terminal), change your working directory to wherever you unzipped the contents, and issue the following command:

```
npm install -g zowe-cli.tgz
```

If the command returns an EACCESS error, refer to [Resolving EACCESS permissions errors when installing packages globally](#) in the npm documentation. If other issues are encountered in your environment, please review [known Zowe CLI issues](#) for solutions.

The highly recommended Secure Credential Store Plug-in for Zowe CLI lets you store your credentials securely in the default credential manager in your computer's operating system. On Linux, libsecret will need to be installed.

If running Linux, please run the following command for your Linux distribution:

- Debian/Ubuntu: `sudo apt-get install libsecret-1-dev`
- Red Hat-based: `sudo yum install libsecret-devel`
- Arch Linux: `sudo pacman -S libsecret`

To install the Secure Credential Store Plug-in for Zowe CLI, issue the following command from where you unzipped the core CLI package contents:

```
zowe plugins install secure-credential-store-for-zowe-cli.tgz
```

User profiles, which contain connection information for interacting with various z/OS services, created after installing the plug-in will automatically store your credentials securely.

To securely store credentials in existing user profiles (profiles that you created prior to installing the SCS plug-in), issue the following command:

```
zowe sscs update
```

7.3 Install Zowe CLI Plug-ins

Zowe CLI is an extendable technology that can be enhanced by installing plug-ins. Zowe offers a number of [plug-ins](#). At the time of this writing, these include plug-ins for [CICS](#), [Db2](#), [FTP](#), [IMS](#), and [MQ](#). There are also many vendor plug-ins, many of which are available on the [public registry](#). At the time of this writing, these include plug-ins for [CA Endevor](#), [CA Endevor Bridge for Git](#), [CA File Master Plus](#), [CA OPS/MVS](#), [CA View](#), [IBM CICS Bundle Generation and Deployment](#), and [IBM z/OS Connect EE](#).

7.3.1 Install from Public npm Registry

To install a Zowe CLI plug-in from the registry, simply locate the plug-in you wish to install, e.g. `@zowe/cics-for-zowe-cli`, find the distribution tag for the distribution you want to install, e.g. `zowe-v1-lts`, and issue the following command:

```
zowe plugins install <name>@<distTag>
```

For example,

```
zowe plugins install @zowe/cics-for-zowe-cli@zowe-v1-lts
```

Multiple plug-ins can be installed in a single command. For example, to install all Zowe CLI plug-ins available from the Zowe organization, you could issue:

```
zowe plugins install @zowe/cics-for-zowe-cli@zowe-v1-lts  
    @zowe/ims-for-zowe-cli@zowe-v1-lts @zowe/mq-for-zowe-cli@zowe-v1-lts  
    @zowe/zos-ftp-for-zowe-cli@zowe-v1-lts  
    @zowe/db2-for-zowe-cli@zowe-v1-lts
```

Vendor plug-ins on the registry are installed in the same way. For example, to install the CA Endevor plug-in, you would issue

```
zowe plugins install @broadcom/endevor-for-zowe-cli@zowe-v1-lts
```

7.3.2 Install from Bundled Package

Navigate to [Zowe.org Downloads](#) and click the CLI Plugins button to download the package which includes all Zowe CLI plug-ins for the Zowe organization. After accepting the EULA for Zowe, a package named `zowe-cli-plugins-v.r.m.zip` will be downloaded to your machine. Unzip the contents of `zowe-cli-plugins-v.r.m.zip` to a preferred location on your machine. You can select which plug-ins you want to install. The IBM Db2 plug-in requires [additional configuration](#) when installing from a local package. To install all plug-ins you can issue:

```
zowe plugins install cics-for-zowe-cli.tgz zos-ftp-for-zowe-cli.tgz  
    ims-for-zowe-cli.tgz mq-for-zowe-cli.tgz db2-for-zowe-cli.tgz
```

For offline installation of vendor plug-ins, please reach out to the specific vendor for details.

7.4 Summary

In this chapter we walked through the process of installing the prerequisite software, Node.js and npm, as well as Zowe CLI and various plug-ins.

Part 2 - Learning COBOL

8 Basic COBOL

This chapter introduces the basics of COBOL syntax. It then demonstrates how to view and run a basic COBOL program in VSCode.

- COBOL characteristics
 - Enterprise COBOL
 - Chapter objectives
- What must a novice COBOL programmer know to be an experienced COBOL programmer?
 - What are the coding rules and the reference format?
 - What is the structure of COBOL?
 - What are COBOL reserved words?
 - What is a COBOL statement?
 - What is the meaning of a scope terminator?
 - What is a COBOL sentence?
 - What is a COBOL paragraph?
 - What is a COBOL section?
- COBOL Divisions
 - COBOL Divisions structure
 - What are the four Divisions of COBOL?
- PROCEDURE DIVISION explained
- Additional information
 - Professional manuals
 - Learn more about recent COBOL advancements
- Lab
- Lab - Zowe CLI & Automation
 - Zowe CLI - Interactive Usage
 - Zowe CLI - Programmatic Usage

8.1 COBOL characteristics

COBOL is an English-like computer language enabling COBOL source code to be easier to read, understand, and maintain. Learning to program in COBOL includes knowledge of COBOL source code rules, COBOL reserved words, COBOL structure, and the ability to locate and interpret professional COBOL documentation. These COBOL characteristics must be understood, to be proficient in reading, writing, and maintaining COBOL programs.

8.1.1 Enterprise COBOL

COBOL is a standard and not owned by any company or organization. “Enterprise COBOL” is the name for the COBOL programming language compiled and executed in the IBM Z Operating System, z/OS. COBOL details and explanations in the following chapters apply to Enterprise COBOL.

Enterprise COBOL has decades of advancements, including new functions, feature extensions, improved performance, application programming interfaces (APIs), etc. It works with modern infrastructure technologies with native support for JSON, XML, and Java®.

8.1.2 Chapter objectives

The object of the chapter is to expose the reader to COBOL terminology, coding rules, and syntax while the remaining chapters include greater detail with labs for practicing what is introduced in this chapter.

8.2 What must a novice COBOL programmer know to be an experienced COBOL programmer?

This section will provide the reader with the information needed to more thoroughly understand the questions and answers being asked in each subsequent heading.

8.2.1 What are the coding rules and the reference format?

COBOL source code is column dependent, meaning column rules are strictly enforced. Each COBOL source code line has five areas, where each of these areas has a beginning and ending column.

COBOL source text must be written in COBOL reference format. Reference format consists of the areas depicted in Figure 1. in a 72-character line.

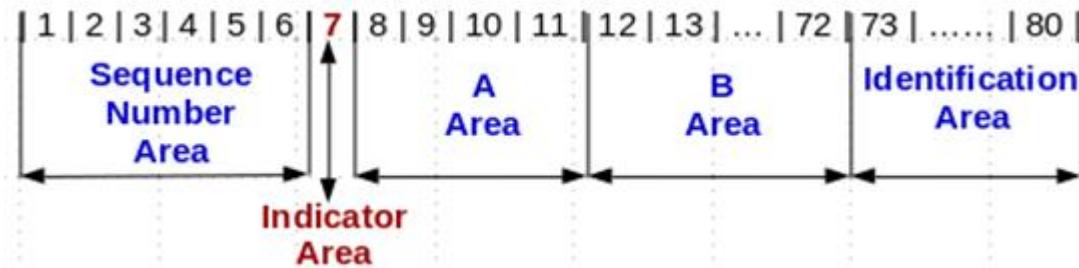


Figure 57:

Figure 1. COBOL reference format

The COBOL reference format is formatted as follows:

8.2.1.1 Sequence Number Area (columns 1 - 6)

- Blank or reserved for line sequence numbers.

8.2.1.2 Indicator Area (column 7)

- A multi-purpose area:
 - Comment line (generally an asterisk symbol)
 - Continuation line (generally a hyphen symbol)
 - Debugging line (D or d)
 - Source listing formatting (a slash symbol)

8.2.1.3 Area A (columns 8 - 11)

- Certain items must begin in Area A, they are:
 - Level indicators
 - Declarative
 - Division, Section, Paragraph headers
 - Paragraph names
- Column 8 is referred to as the A Margin

8.2.1.4 Area B (columns 12 - 72)

- Certain items must begin in Area B, they are:
 - Entries, sentences, statements, and clauses
 - Continuation lines
- Column 12 is referred to as the B Margin

8.2.1.5 Identification Area (columns 73 - 80)

- Ignored by the compiler.
- Can be blank or optionally used by programmer for any purpose.

8.2.2 What is the structure of COBOL?

COBOL is a hierarchy structure consisting and in the top-down order of:

- Divisions
- Sections
- Paragraphs
- Sentences
- Statements

8.2.3 What are COBOL reserved words?

COBOL programming language has many words with specific meaning to the COBOL compiler, referred to as reserved words. These reserved words cannot be used as programmer chosen variable names or programmer chosen data type names.

A few COBOL reserved words pertinent to this book are: PERFORM, MOVE, COMPUTE, IF, THEN, ELSE, EVALUATE, PICTURE, etc.. You can find a table of all COBOL reserved words is located at:

https://www.ibm.com/support/knowledgecenter/zh/SSZJPZ_9.1.0/com.ibm.swg.im.iis.ds.mfjob.dev.doc/topics/r_dmnjbref_COBOL_Reserved_Words.html

8.2.4 What is a COBOL statement?

Specific COBOL reserved words are used to change the execution flow based upon current conditions. “Statements” only exist within the Procedure Division, the program processing logic. Examples of COBOL reserved words used to change the execution flow are:

- IF
- Evaluate
- Perform

8.2.5 What is the meaning of a scope terminator?

A scope terminator can be explicit or implicit. An explicit scope terminator marks the end of certain PROCEDURE DIVISION statements with the “END-” COBOL reserved word. Any COBOL verb that is either, always conditional (IF, EVALUATE), or has a conditional clause (COMPUTE, PERFORM, READ) will have a matching scope terminator. An implicit scope terminator is a period (.) that ends the scope of all previous statements that have not yet been ended.

8.2.6 What is a COBOL sentence?

A COBOL “Sentence” is one or more “Statements” followed by a period (.), where the period serves as a scope terminator.

8.2.7 What is a COBOL paragraph?

A COBOL “Paragraph” is a user-defined or predefined name followed by a period. A “Paragraph” consists of zero or more sentences and are the subdivision of a “Section” or “Division”, see Example 1. below.

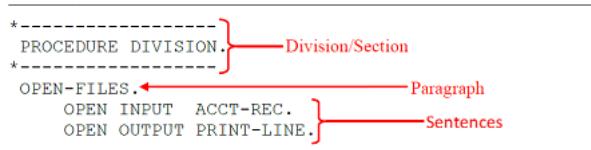


Figure 58:

Example 1. Division -> paragraph -> sentences

8.2.8 What is a COBOL section?

A “Section” is either a user-defined or a predefined name followed by a period and consists of zero or more sentences. A “Section” is a collection of paragraphs.

8.3 COBOL Divisions

This section introduces the four COBOL Divisions and briefly describes their purpose and characteristics.

8.3.1 COBOL Divisions structure

Divisions are subdivided into Sections.

Sections are subdivided into Paragraphs.

Paragraphs are subdivided into Sentences.

Sentences consists of Statements.

Statements begin with COBOL reserved words and can be subdivided into “Phrases”

8.3.2 What are the four Divisions of COBOL?

8.3.2.1 IDENTIFICATION DIVISION

The IDENTIFICATION DIVISION identifies the program with a name and, optionally, gives other identifying information, such as the Author name, program compiled date (last modified), etc.

8.3.2.2 ENVIRONMENT DIVISION

The ENVIRONMENT DIVISION describes the aspects of your program that depend on the computing environment, such as the computer configuration and the computer inputs and outputs.

8.3.2.3 DATA DIVISION

The DATA DIVISION is where characteristics of data are defined in one of the following sections:

- FILE SECTION:

Defines data used in input-output operations.

- LINKAGE SECTION:

Describes data from another program. When defining data developed for internal processing.

- WORKING-STORAGE SECTION:

Storage allocated and remaining for the life of the program.

- LOCAL-STORAGE SECTION:

Storage allocated each time a program is called and de-allocated when the program ends.

8.3.2.4 PROCEDURE DIVISION

The PROCEDURE DIVISION contains instructions related to the manipulation of data and interfaces with other procedures are specified.

8.4 PROCEDURE DIVISION explained

The PROCEDURE DIVISION is where the work gets done in the program. Statements are in the PROCEDURE DIVISION where they are actions to be taken by the program. The PROCEDURE DIVISION is required for data to be processed by the program. PROCEDURE DIVISION of a program is divided into sections and paragraphs, which contain sentences and statements, as described here:

- **Section** - A logical subdivision of your processing logic. A section has a header and is optionally followed by one or more paragraphs. A section can be the subject of a PERFORM statement. One type of section is for declaratives. Declaratives are a set of one or more special purpose sections. Special purpose sections are exactly what they sound like, sections written for special purposes and may contain things like description of inputs and outputs. They are written at the beginning of the PROCEDURE DIVISION, the first of which is preceded by the key word DECLARATIVES and the last of which is followed by the key word END DECLARATIVES.
- **Paragraph** - A subdivision of a section, procedure, or program. A paragraph can be the subject of a statement.
- **Sentence** - A series of one or more COBOL statements ending with a period.
- **Statement** - An action to be taken by the program, such as adding two numbers.
- **Phrase** - A small part of a statement (i.e. subdivision), analogous to an English adjective or preposition

8.5 Additional information

This section provides useful resources in the form of manuals and videos to assist in learning more about the basics of COBOL.

8.5.1 Professional manuals

As Enterprise COBOL experience advances, the need for the professional documentation is greater. An internet search for Enterprise COBOL manuals includes: “Enterprise COBOL for z/OS documentation library - IBM”, link provided below. The site content has tabs for each COBOL release level. As of April 2020, the current release of Enterprise COBOL is V6.3. Highlight V6.3 tab, then select product documentation.

<https://www.ibm.com/support/pages/enterprise-cobol-zos-documentation-library>

Three ‘Enterprise COBOL for z/OS’ manuals are referenced throughout the chapters as sources of additional information, for reference and to advance the level of knowledge. They are:

1. Language Reference - Describes the COBOL language such as program structure, reserved words, etc.

<http://publibfp.boulder.ibm.com/epubs/pdf/igy6lr30.pdf>

2. Programming Guide - Describes advanced topics such as COBOL compiler options, program performance optimization, handling errors, etc.

<http://publibfp.boulder.ibm.com/epubs/pdf/igy6pg30.pdf>

3. Messages and Codes - To better understand certain COBOL compiler messages and return codes to diagnose problems.

<http://publibfp.boulder.ibm.com/epubs/pdf/c2746481.pdf>

8.5.2 Learn more about recent COBOL advancements

- What’s New in Enterprise COBOL for z/OS V6.1:

https://youtu.be/N_Zsd1W8hWc

- What’s New in Enterprise COBOL for z/OS V6.2:

<https://youtu.be/H0iweEbVNfs>

- What’s New in Enterprise COBOL for z/OS V6.3:

<https://youtu.be/bRLKGeB6W2A>

8.6 Lab

In this lab exercise you will connect to an IBM Z system, view a simple COBOL hello world program in VSCode, submit JCL to compile the COBOL program, and view the output. Refer to “Installation of VSCode and extensions” to configure VSCode if you have not already done so. You can either use Z Open Editor and Zowe Explorer, or Code4z.

1. The lab assumes installation of VSCode with either Z Open Editor and Zowe Explorer extensions, as shown in Figure 2a, or the Code4z extension pack, as shown in Figure 2b.

Click the **Extensions** icon. If you installed Z Open Editor and Zowe Explorer, the list should include:

1. IBM Z Open Editor
2. Zowe Explorer

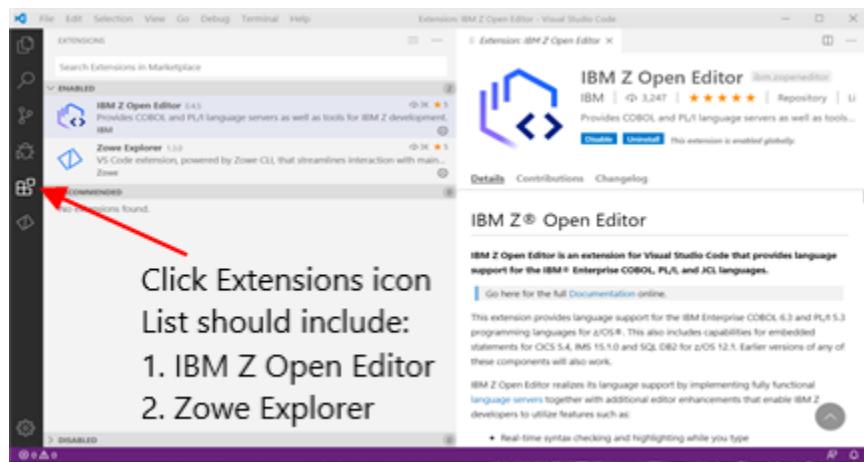


Figure 59:

Figure 2a. The Z Open Editor and Zowe Explorer VSCode extensions

If you installed Code4z, the list should include:

1. COBOL
2. COBOL Language Support
3. Code4z
4. Debugger for Mainframe
5. Explorer for Endevor
6. HLASM Language Support
7. Zowe Explorer

In these exercises, you will only use the COBOL Language Support and Zowe Explorer extensions.

Figure 2b. The Code4z package of VS Code extensions.

Note: If your list contains both Z Open Editor and COBOL Language Support, disable one of them, by clicking on the **cog** icon next to the extension in the extensions list, and selecting **disable**.

2. Click the Zowe Explorer icon as shown in Figure 3.

Figure 3. Zowe Explorer Zowe Explorer icon

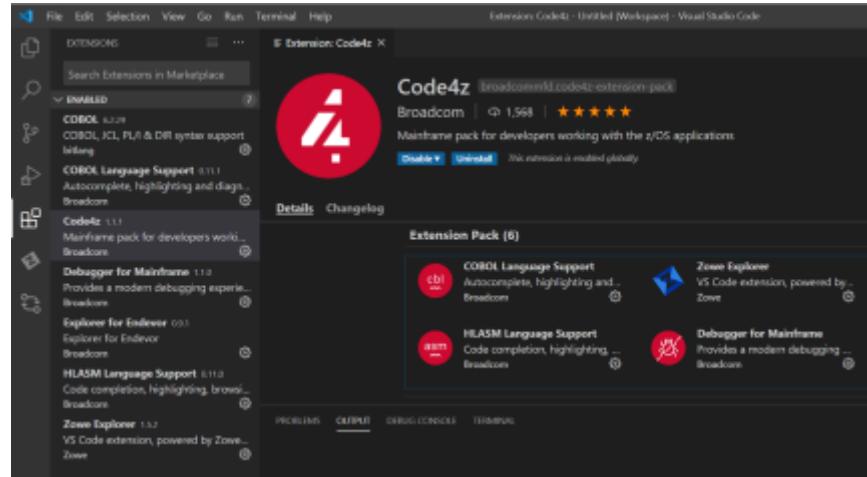


Figure 60:

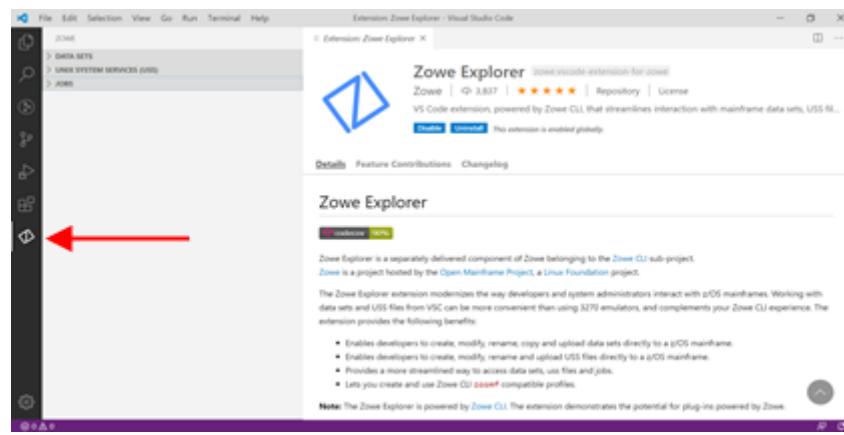


Figure 61:

- Zowe Explorer can list Data Sets, Unix System Services (USS) files, and Jobs output as shown in Figure 4.
- “+” will appear when hovering to the far right on the DATA SETS line. Click the “+” to define a VSCode profile.

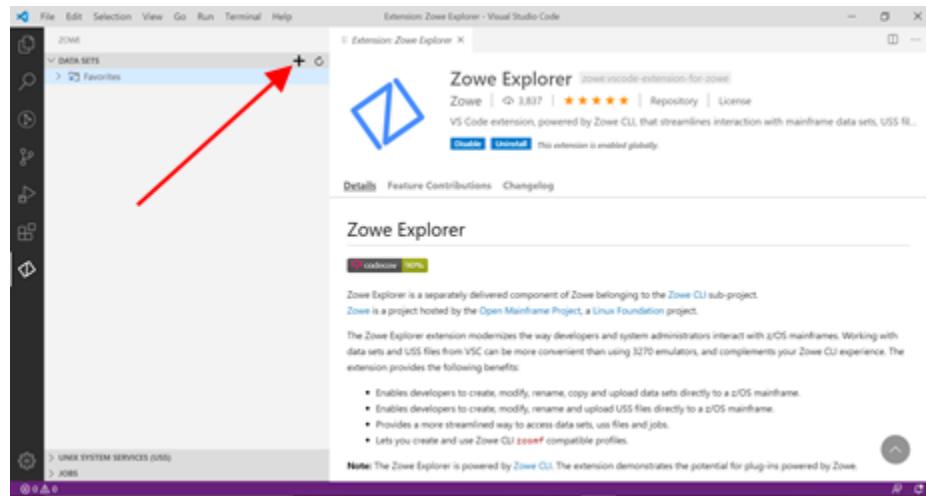


Figure 62:

Figure 4. Zowe Explorer

- A box appears to define a new profile. Click “+” to the left of Create a New Connection to z/OS as shown in Figure 5.



Figure 63:

Figure 5. Create a new connection to z/OS

- Select a name to enter, then enter. Figure 6. used LearnCOBOL as the selected connection name.



Figure 64:

Figure 6. Set connection name

- VSCode prompts for z/OSMF URL and port as shown in Figure 7. The z/OSMF URL and port will normally be provided by z/OS System Administrator.

Figure 7. z/OSMF URL

- A sample z/OSMF URL and port is entered as shown in Figure 8.

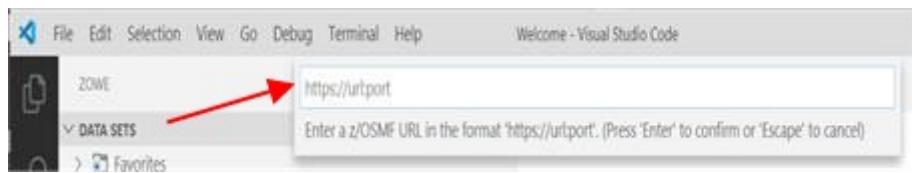


Figure 65:

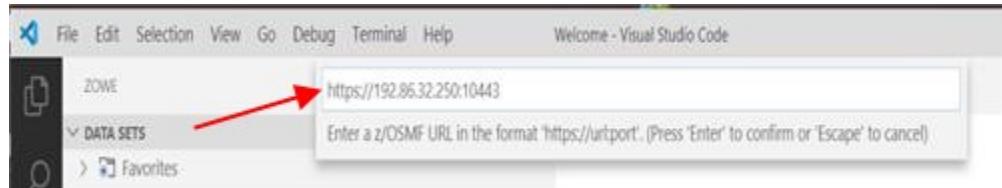


Figure 66:

Figure 8. Specified z/OSMF URL

8. The connection prompts for Username as shown in Figure 9.



Figure 67:

Figure 9. User name prompt

9. A sample username, Z99998, is entered as shown in Figure 10. The ID is assigned by the System Administrator.

Figure 10. Specified user name

10. The connection prompts for the Username Password as shown in Figure 11.

Figure 11. Password prompt

11. Enter the Username Password as shown in Figure 12.

Figure 12. Specified password

12. Select **False - Accept connections with self-signed certificates** to authorize workstation connection as shown in Figure 13.

Figure 13. Accept connections with self-signed certifications

13. Result is Favorites in the Data Sets, Unix System Services, and Jobs sections as shown in Figure 14.

Figure 14. Favorites

14. Again, click on the + to the far right on the Data Sets selection. Result is another prompt to Create a New Connection to z/OS. LearnCOBOL is in the connection list. Select **LearnCOBOL** for the Data Sets available to the previously defined LearnCOBOL connection to z/OS as shown in Figure 15.

Figure 15. LearnCOBOL connection



Figure 68:

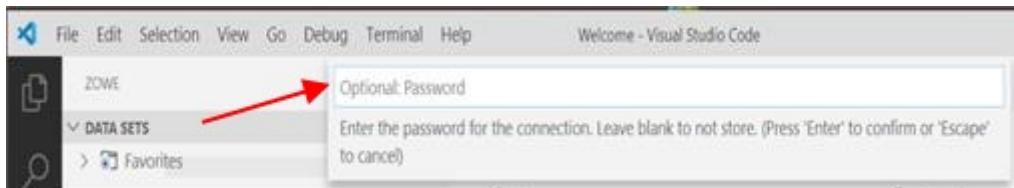


Figure 69:

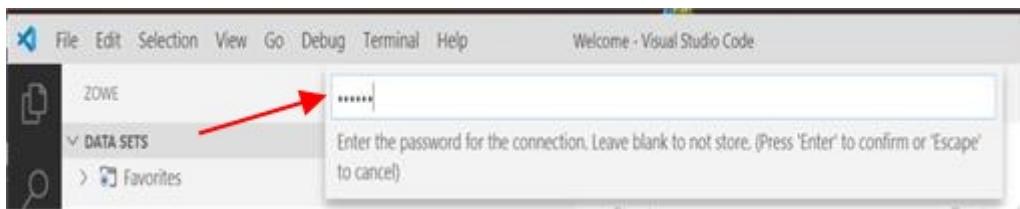


Figure 70:



Figure 71:

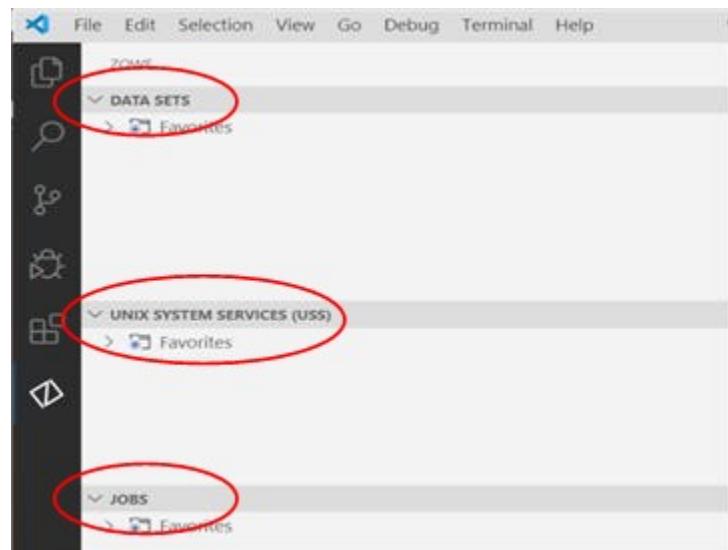


Figure 72:

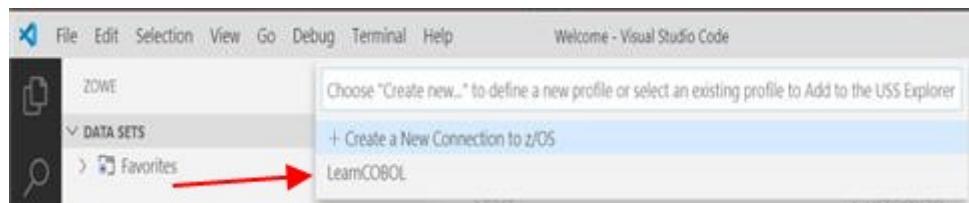


Figure 73:

15. Expansion of LearnCOBOL reads “Use the search button to display datasets”. Click the search button as shown in Figure 16.

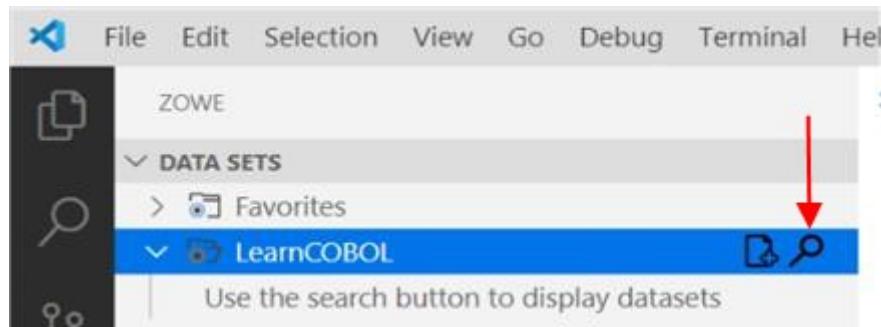


Figure 74:

Figure 16. Search button

16. A prompt to “Select a filter” appears for ID Z99998. Select the + to ‘Create a new filter’ as shown in Figure 17.

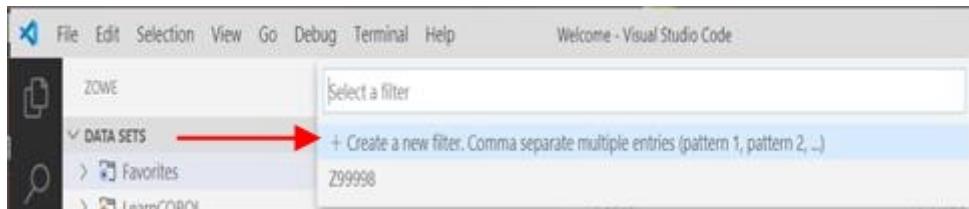


Figure 75:

Figure 17. Select a filter

17. A prompt appears to enter the filter name to be searched as shown in Figure 18.

Figure 18. Filter name to be searched

18. ID Z99998 has lab data set names that begin the Z99998. Therefore, Z99998 is entered as the filter to searched for ID Z99998 as shown in Figure 19.

Figure 19. Entered filter name

19. A list of data set names beginning with Z99998 for ID Z99998 from z/OS Connection LearnCOBOL appears as shown in Figure 20.

Figure 20. Filtered data set names

20. Expand **Z99998.CBL** to view COBOL source members, then select member **HELLO** to see a simple COBOL ‘Hello World!’ program as shown in Figure 21.

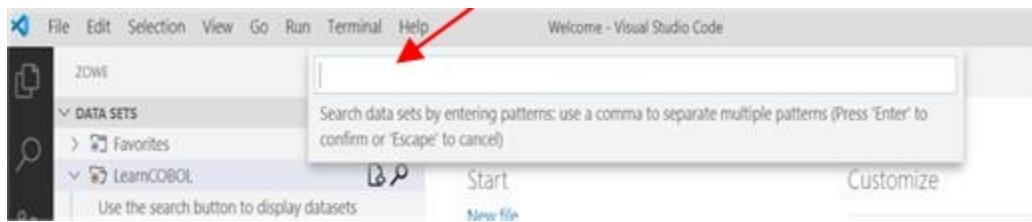


Figure 76:



Figure 77:

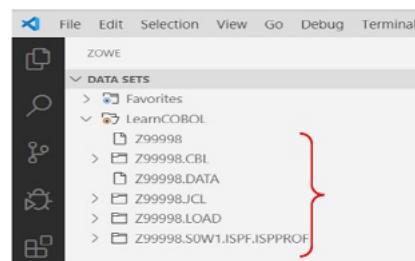


Figure 78:



Figure 79:

Figure 21. Z9998.CBL

21. Expand **Z9998.JCL** to view JCL and select member **HELLO** which is the JCL to compile and execute simple ‘Hello World!’ COBOL source code as shown in Figure 22.

```

1 //>HELLOCBL JOB 1,NOTIFY&$VSUID
2 //>COBRUN EXEC IGYNCLG,SRC=HELLO
3

```

Figure 80:

Figure 22. Z9998.JCL

22. Right click on JCL member **HELLO**. A section box appears. Select **Submit Job** for system to process HELLO JCL as shown in Figure 23. The submitted JCL job compiles the COBOL HELLO source code, then executes the COBOL HELLO program.

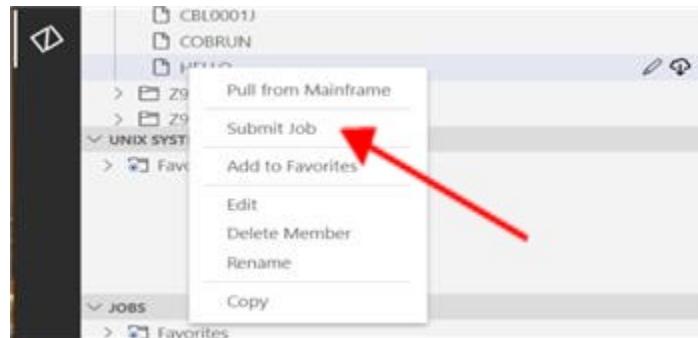


Figure 81:

Figure 23. Submit Job

23. Observe the ‘Jobs’ section in Zowe Explorer as shown in Figure 24.

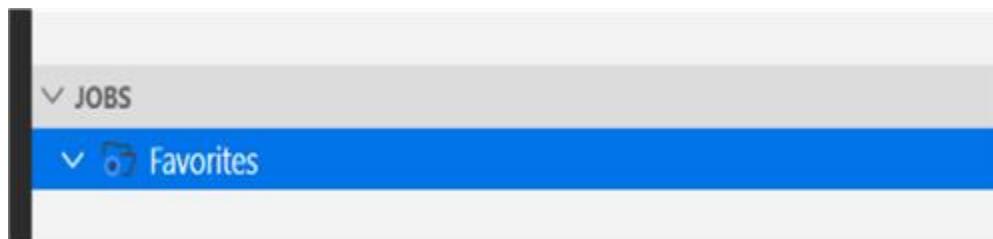


Figure 82:

Figure 24. JOBS section

24. Again, click on the + to the far right on the Jobs selection. Result is another prompt to ‘Create new’. Select **LearnCOBOL** from the list as shown in Figure 25.

Figure 25. + Select LearnCOBOL connection

25. As a result, the JCL jobs owned by ID Z9998 appears. HELLOCBL is the JCL job name previously submitted. Expand **HELOCBL** output to view sections of the output as shown in Figure 26.

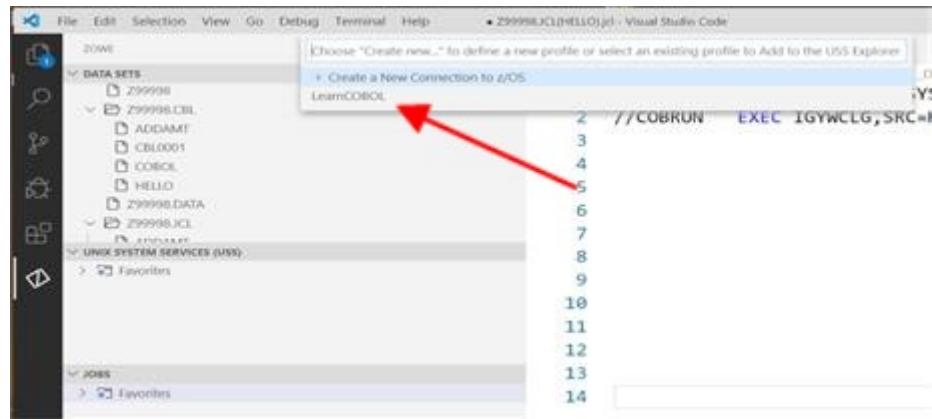


Figure 83:

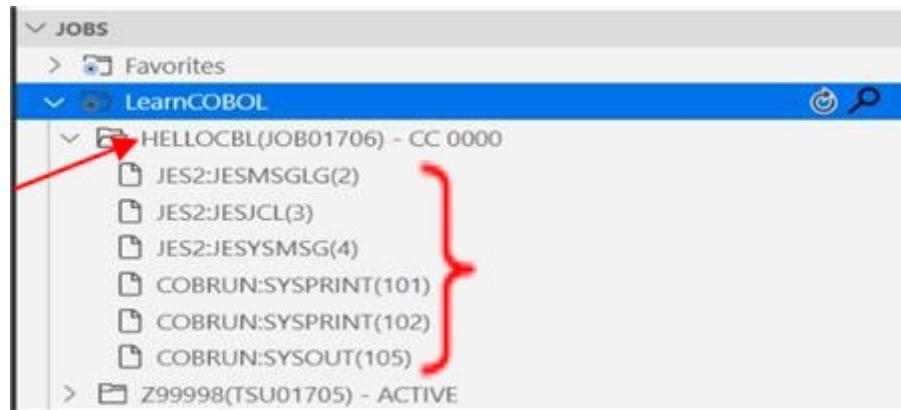


Figure 84:

Figure 26. HELLOCBL output

26. Select **COBRUN:SYSPRINT(101)** to view the COBOL compiler output. Scroll forward in the COBOL compile to locate the COBOL source code compiled into an executable module as shown in Figure 27. Observe the Indicator Area in column 7, A Area beginning in column 8, and B Area beginning in column 12. Also, observe the period (.) scope terminators in the COBOL source.

```

File Edit Selection View Go Debug Terminal Help
HELOCBL(JOB01706.SYSPRINT - Visual Studio Code
HELOCBL(JOB01706.SYSPRINT X
ZONE
DATA SETS
  Z99998
  Z99998.CBL
    ADDAMIT
    CBL0001
    COBOL
    HELLO
    Z99998.DATA
  Z99998.JCL
  UNIX SYSTEM SERVICES (USS)
    Favorites
JOBS
  Favorites
  LearnCOBOL
  HELLOCBL(JOB01706) - CC.0000
    JES2JESMSG(G2)
    JES2JESACLI(3)
    JES2JESMSG(4)
    COBRUN:SYSPRINT(101) (highlighted by a red arrow)
    COBRUN:SYSPRINT(102)
    COBRUN:SYSOUT(105)
  100 1PP 5655-EC6 IBM Enterprise COBOL for z/OS 6.3.0 P200115
  101 0An "M" preceding a data-name reference indicates that th
  102
  103 93 Defined Cross-reference of data names References
  104 94
  105 95 1PP 5655-EC6 IBM Enterprise COBOL for z/OS 6.3.0 P200115
  106 96 0 Defined Cross-reference of programs References
  107 97
  108 98 2 HELLO
  109 99 -* Statistics for COBOL program HELLO:
  110 *   Source records = 5
  111 *   Data Division statements = 0
  112
  113
  114
  115
  116
  117
  118
  119
  120
  121
  122
  123
  124
  125
  126
  127
  128
  129
  130
  131
  132
  133
  134
  135
  136
  137
  138
  139
  140
  141
  142
  143
  144
  145
  146
  147
  148
  149
  150
  151
  152
  153
  154
  155
  156
  157
  158
  159
  160
  161
  162
  163
  164
  165
  166
  167
  168
  169
  170
  171
  172
  173
  174
  175
  176
  177
  178
  179
  180
  181
  182
  183
  184
  185
  186
  187
  188
  189
  190
  191
  192
  193
  194
  195
  196
  197
  198
  199
  200
  201
  202
  203
  204
  205
  206
  207
  208
  209
  210
  211
  212
  213
  214
  215
  216
  217
  218
  219
  220
  221
  222
  223
  224
  225
  226
  227
  228
  229
  230
  231
  232
  233
  234
  235
  236
  237
  238
  239
  240
  241
  242
  243
  244
  245
  246
  247
  248
  249
  250
  251
  252
  253
  254
  255
  256
  257
  258
  259
  260
  261
  262
  263
  264
  265
  266
  267
  268
  269
  270
  271
  272
  273
  274
  275
  276
  277
  278
  279
  280
  281
  282
  283
  284
  285
  286
  287
  288
  289
  290
  291
  292
  293
  294
  295
  296
  297
  298
  299
  300
  301
  302
  303
  304
  305
  306
  307
  308
  309
  310
  311
  312
  313
  314
  315
  316
  317
  318
  319
  320
  321
  322
  323
  324
  325
  326
  327
  328
  329
  330
  331
  332
  333
  334
  335
  336
  337
  338
  339
  340
  341
  342
  343
  344
  345
  346
  347
  348
  349
  350
  351
  352
  353
  354
  355
  356
  357
  358
  359
  360
  361
  362
  363
  364
  365
  366
  367
  368
  369
  370
  371
  372
  373
  374
  375
  376
  377
  378
  379
  380
  381
  382
  383
  384
  385
  386
  387
  388
  389
  390
  391
  392
  393
  394
  395
  396
  397
  398
  399
  400
  401
  402
  403
  404
  405
  406
  407
  408
  409
  410
  411
  412
  413
  414
  415
  416
  417
  418
  419
  420
  421
  422
  423
  424
  425
  426
  427
  428
  429
  430
  431
  432
  433
  434
  435
  436
  437
  438
  439
  440
  441
  442
  443
  444
  445
  446
  447
  448
  449
  450
  451
  452
  453
  454
  455
  456
  457
  458
  459
  460
  461
  462
  463
  464
  465
  466
  467
  468
  469
  470
  471
  472
  473
  474
  475
  476
  477
  478
  479
  480
  481
  482
  483
  484
  485
  486
  487
  488
  489
  490
  491
  492
  493
  494
  495
  496
  497
  498
  499
  500
  501
  502
  503
  504
  505
  506
  507
  508
  509
  510
  511
  512
  513
  514
  515
  516
  517
  518
  519
  520
  521
  522
  523
  524
  525
  526
  527
  528
  529
  530
  531
  532
  533
  534
  535
  536
  537
  538
  539
  540
  541
  542
  543
  544
  545
  546
  547
  548
  549
  550
  551
  552
  553
  554
  555
  556
  557
  558
  559
  560
  561
  562
  563
  564
  565
  566
  567
  568
  569
  570
  571
  572
  573
  574
  575
  576
  577
  578
  579
  580
  581
  582
  583
  584
  585
  586
  587
  588
  589
  590
  591
  592
  593
  594
  595
  596
  597
  598
  599
  600
  601
  602
  603
  604
  605
  606
  607
  608
  609
  610
  611
  612
  613
  614
  615
  616
  617
  618
  619
  620
  621
  622
  623
  624
  625
  626
  627
  628
  629
  630
  631
  632
  633
  634
  635
  636
  637
  638
  639
  640
  641
  642
  643
  644
  645
  646
  647
  648
  649
  650
  651
  652
  653
  654
  655
  656
  657
  658
  659
  660
  661
  662
  663
  664
  665
  666
  667
  668
  669
  670
  671
  672
  673
  674
  675
  676
  677
  678
  679
  680
  681
  682
  683
  684
  685
  686
  687
  688
  689
  690
  691
  692
  693
  694
  695
  696
  697
  698
  699
  700
  701
  702
  703
  704
  705
  706
  707
  708
  709
  710
  711
  712
  713
  714
  715
  716
  717
  718
  719
  720
  721
  722
  723
  724
  725
  726
  727
  728
  729
  730
  731
  732
  733
  734
  735
  736
  737
  738
  739
  740
  741
  742
  743
  744
  745
  746
  747
  748
  749
  750
  751
  752
  753
  754
  755
  756
  757
  758
  759
  760
  761
  762
  763
  764
  765
  766
  767
  768
  769
  770
  771
  772
  773
  774
  775
  776
  777
  778
  779
  780
  781
  782
  783
  784
  785
  786
  787
  788
  789
  790
  791
  792
  793
  794
  795
  796
  797
  798
  799
  800
  801
  802
  803
  804
  805
  806
  807
  808
  809
  810
  811
  812
  813
  814
  815
  816
  817
  818
  819
  820
  821
  822
  823
  824
  825
  826
  827
  828
  829
  830
  831
  832
  833
  834
  835
  836
  837
  838
  839
  840
  841
  842
  843
  844
  845
  846
  847
  848
  849
  850
  851
  852
  853
  854
  855
  856
  857
  858
  859
  860
  861
  862
  863
  864
  865
  866
  867
  868
  869
  870
  871
  872
  873
  874
  875
  876
  877
  878
  879
  880
  881
  882
  883
  884
  885
  886
  887
  888
  889
  890
  891
  892
  893
  894
  895
  896
  897
  898
  899
  900
  901
  902
  903
  904
  905
  906
  907
  908
  909
  910
  911
  912
  913
  914
  915
  916
  917
  918
  919
  920
  921
  922
  923
  924
  925
  926
  927
  928
  929
  930
  931
  932
  933
  934
  935
  936
  937
  938
  939
  940
  941
  942
  943
  944
  945
  946
  947
  948
  949
  950
  951
  952
  953
  954
  955
  956
  957
  958
  959
  960
  961
  962
  963
  964
  965
  966
  967
  968
  969
  970
  971
  972
  973
  974
  975
  976
  977
  978
  979
  980
  981
  982
  983
  984
  985
  986
  987
  988
  989
  990
  991
  992
  993
  994
  995
  996
  997
  998
  999
  1000
  1001
  1002
  1003
  1004
  1005
  1006
  1007
  1008
  1009
  1010
  1011
  1012
  1013
  1014
  1015
  1016
  1017
  1018
  1019
  1020
  1021
  1022
  1023
  1024
  1025
  1026
  1027
  1028
  1029
  1030
  1031
  1032
  1033
  1034
  1035
  1036
  1037
  1038
  1039
  1040
  1041
  1042
  1043
  1044
  1045
  1046
  1047
  1048
  1049
  1050
  1051
  1052
  1053
  1054
  1055
  1056
  1057
  1058
  1059
  1060
  1061
  1062
  1063
  1064
  1065
  1066
  1067
  1068
  1069
  1070
  1071
  1072
  1073
  1074
  1075
  1076
  1077
  1078
  1079
  1080
  1081
  1082
  1083
  1084
  1085
  1086
  1087
  1088
  1089
  1090
  1091
  1092
  1093
  1094
  1095
  1096
  1097
  1098
  1099
  1100
  1101
  1102
  1103
  1104
  1105
  1106
  1107
  1108
  1109
  1110
  1111
  1112
  1113
  1114
  1115
  1116
  1117
  1118
  1119
  1120
  1121
  1122
  1123
  1124
  1125
  1126
  1127
  1128
  1129
  1130
  1131
  1132
  1133
  1134
  1135
  1136
  1137
  1138
  1139
  1140
  1141
  1142
  1143
  1144
  1145
  1146
  1147
  1148
  1149
  1150
  1151
  1152
  1153
  1154
  1155
  1156
  1157
  1158
  1159
  1160
  1161
  1162
  1163
  1164
  1165
  1166
  1167
  1168
  1169
  1170
  1171
  1172
  1173
  1174
  1175
  1176
  1177
  1178
  1179
  1180
  1181
  1182
  1183
  1184
  1185
  1186
  1187
  1188
  1189
  1190
  1191
  1192
  1193
  1194
  1195
  1196
  1197
  1198
  1199
  1200
  1201
  1202
  1203
  1204
  1205
  1206
  1207
  1208
  1209
  1210
  1211
  1212
  1213
  1214
  1215
  1216
  1217
  1218
  1219
  1220
  1221
  1222
  1223
  1224
  1225
  1226
  1227
  1228
  1229
  1230
  1231
  1232
  1233
  1234
  1235
  1236
  1237
  1238
  1239
  1240
  1241
  1242
  1243
  1244
  1245
  1246
  1247
  1248
  1249
  1250
  1251
  1252
  1253
  1254
  1255
  1256
  1257
  1258
  1259
  1260
  1261
  1262
  1263
  1264
  1265
  1266
  1267
  1268
  1269
  1270
  1271
  1272
  1273
  1274
  1275
  1276
  1277
  1278
  1279
  1280
  1281
  1282
  1283
  1284
  1285
  1286
  1287
  1288
  1289
  1290
  1291
  1292
  1293
  1294
  1295
  1296
  1297
  1298
  1299
  1300
  1301
  1302
  1303
  1304
  1305
  1306
  1307
  1308
  1309
  1310
  1311
  1312
  1313
  1314
  1315
  1316
  1317
  1318
  1319
  1320
  1321
  1322
  1323
  1324
  1325
  1326
  1327
  1328
  1329
  1330
  1331
  1332
  1333
  1334
  1335
  1336
  1337
  1338
  1339
  1340
  1341
  1342
  1343
  1344
  1345
  1346
  1347
  1348
  1349
  1350
  1351
  1352
  1353
  1354
  1355
  1356
  1357
  1358
  1359
  1360
  1361
  1362
  1363
  1364
  1365
  1366
  1367
  1368
  1369
  1370
  1371
  1372
  1373
  1374
  1375
  1376
  1377
  1378
  1379
  1380
  1381
  1382
  1383
  1384
  1385
  1386
  1387
  1388
  1389
  1390
  1391
  1392
  1393
  1394
  1395
  1396
  1397
  1398
  1399
  1400
  1401
  1402
  1403
  1404
  1405
  1406
  1407
  1408
  1409
  1410
  1411
  1412
  1413
  1414
  1415
  1416
  1417
  1418
  1419
  1420
  1421
  1422
  1423
  1424
  1425
  1426
  1427
  1428
  1429
  1430
  1431
  1432
  1433
  1434
  1435
  1436
  1437
  1438
  1439
  1440
  1441
  1442
  1443
  1444
  1445
  1446
  1447
  1448
  1449
  1450
  1451
  1452
  1453
  1454
  1455
  1456
  1457
  1458
  1459
  1460
  1461
  1462
  1463
  1464
  1465
  1466
  1467
  1468
  1469
  1470
  1471
  1472
  1473
  1474
  1475
  1476
  1477
  1478
  1479
  1480
  1481
  1482
  1483
  1484
  1485
  1486
  1487
  1488
  1489
  1490
  1491
  1492
  1493
  1494
  1495
  1496
  1497
  1498
  1499
  1500
  1501
  1502
  1503
  1504
  1505
  1506
  1507
  1508
  1509
  1510
  1511
  1512
  1513
  1514
  1515
  1516
  1517
  1518
  1519
  1520
  1521
  1522
  1523
  1524
  1525
  1526
  1527
  1528
  1529
  1530
  1531
  1532
  1533
  1534
  1535
  1536
  1537
  1538
  1539
  1540
  1541
  1542
  1543
  1544
  1545
  1546
  1547
  1548
  1549
  1550
  1551
  1552
  1553
  1554
  1555
  1556
  1557
  1558
  1559
  1560
  1561
  1562
  1563
  1564
  1565
  1566
  1567
  1568
  1569
  1570
  1571
  1572
  1573
  1574
  1575
  1576
  1577
  1578
  1579
  1580
  1581
  1582
  1583
  1584
  1585
  1586
  1587
  1588
  1589
  1590
  1591
  1592
  1593
  1594
  1595
  1596
  1597
  1598
  1599
  1600
  1601
  1602
  1603
  1604
  1605
  1606
  1607
  1608
  1609
  1610
  1611
  1612
  1613
  1614
  1615
  1616
  1617
  1618
  1619
  1620
  1621
  1622
  1623
  1624
  1625
  1626
  1627
  1628
  1629
  1630
  1631
  1632
  1633
  1634
  1635
  1636
  1637
  1638
  1639
  1640
  1641
  1642
  1643
  1644
  1645
  1646
  1647
  1648
  1649
  1650
  1651
  1652
  1653
  1654
  1655
  1656
  1657
  1658
  1659
  1660
  1661
  1662
  1663
  1664
  1665
  1666
  1667
  1668
  1669
  1670
  1671
  1672
  1673
  1674
  1675
  1676
  1677
  1678
  1679
  1680
  1681
  1682
  1683
  1684
  1685
  1686
  1687
  1688
  1689
  1690
  1691
  1692
  1693
  1694
  1695
  1696
  1697
  1698
  1699
  1700
  1701
  1702
  1703
  1704
  1705
  1706
  1707
  1708
  1709
  1710
  1711
  1712
  1713
  1714
  1715
  1716
  1717
  1718
  1719
  1720
  1721
  1722
  1723
  1724
  1725
  1726
  1727
  1728
  1729
  1730
  1731
  1732
  1733
  1734
  1735
  1736
  1737
  1738
  1739
  1740
  1741
  1742
  1743
  1744
  1745
  1746
  1747
  1748
  1749
  1750
  1751
  1752
  1753
  1754
  1755
  1756
  1757
  1758
  1759
  
```

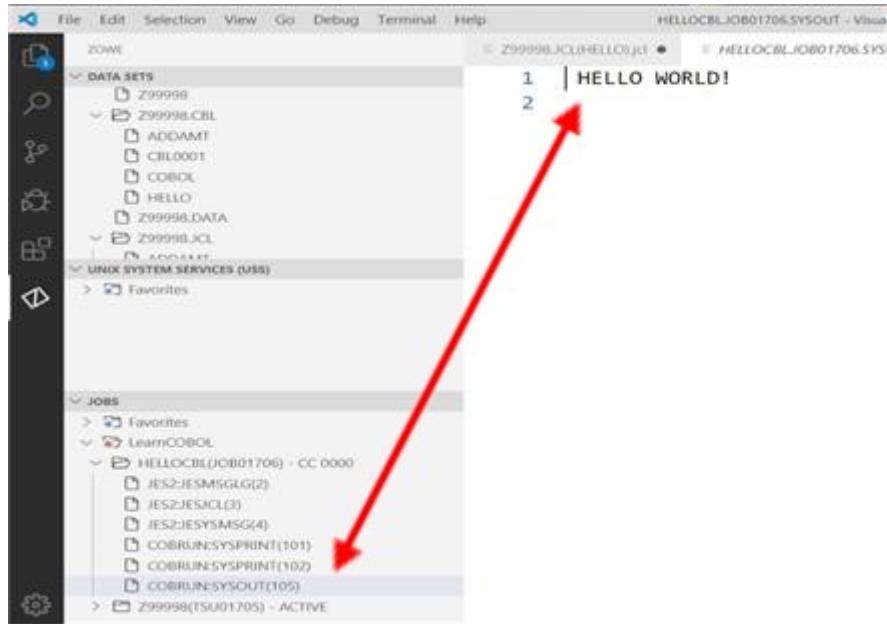


Figure 86:

the default shell selected (outlined in red) is `bash`. I would recommend selecting the default shell as either `bash` or `cmd` for this lab.

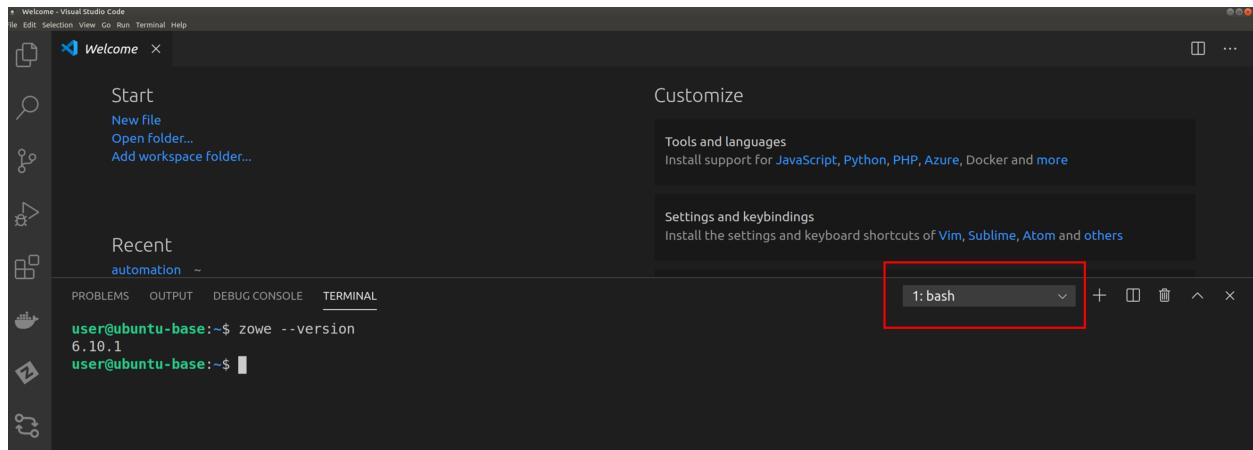


Figure 87:

Figure 29. `zowe --version` command in VS Code Integrated Terminal (Shell selection outlined in red)

2. In order for Zowe CLI to interact with z/OSMF the CLI must know the connection details such as host, port, username, password, etc. While you could enter this information on each command, Zowe provides the ability to store this information in configurations commonly known as profiles. Zowe CLI and the Zowe VS Code Extension share profiles. So if you created a connection profile in the first lab, you could naturally leverage it here.

To create a LearnCOBOL profile (and overwrite it if it already exists), issue the following command with your system details (using `prompt*` will prompt you for certain fields and not show input):

```
zowe profiles create zosmf LearnCOBOL --host 192.86.32.250 --port 10443
--ru false --user prompt* --pass prompt* --ow
```

Many profiles can be created for interacting with different z/OSMF instances. If this was not your first profile, you will want to set it as the default for the following lab exercises. Issue the following command:

```
zowe profiles set zosmf LearnCOBOL
```

The following figure demonstrates this sequence of commands.

The screenshot shows a terminal window titled "demo ~/Desktop". It displays the following command and its output:

```
user@ubuntu-base:~$ zowe profiles create zosmf LearnCOBOL --host 192.86.32.250 --port 10443 --ru false --user prompt* --pass prompt* --ow
"user" Description: Mainframe (z/OSMF) user name, which can be the same as your TSO login.
Please enter "user":
"password" Description: Mainframe (z/OSMF) password, which can be the same as your TSO password.
Please enter "password":
Overwrote existing profile for LearnCOBOL.
Profile created successfully! Path:
/home/user/.zowe/profiles/zosmf/LearnCOBOL.yaml

host:          192.86.32.250
port:          10443
user:          managed by @zowe/secure-credential-store-for-zowe-cli
password:      managed by @zowe/secure-credential-store-for-zowe-cli
rejectUnauthorized: false

Review the created profile and edit if necessary using the profile update command.
user@ubuntu-base:~$ zowe profiles set zosmf LearnCOBOL
The default profile for zosmf set to LearnCOBOL
user@ubuntu-base:~$
```

Figure 88:

Figure 30. Create and set z/OSMF profile (secure credential store plug-in is in use)

3. Confirm you can connect to z/OSMF by issuing the following command:

```
zowe zosmf check status
```

4. List data sets under your ID by issuing a command similar to (see sample output in the following figure):

```
zowe files list ds "Z80462.*"
```

You can also list all members in a partitioned data set by issuing a command similar to (see sample output in the following figure):

```
zowe files list am "Z80462.CBL"
```

Figure 31. zowe files list ds and am commands

5. Next, we will download our COBOL and JCL data set members to our local machine. First, create and open a new folder in your file explorer. Note that you could also create a workspace to manage multiple projects. See the following figure for help:

Figure 32. File explorer view to demonstrate opening a new folder

Once you have an empty folder opened, return to the integrated terminal, ensure you are in your folder, and issue commands similar to:

```
zowe files download am "Z80462.CBL" -e ".cbl"
zowe files download am "Z80462.JCL" -e ".jcl"
```

Then open hello.cbl in your file explorer. A completed example is shown in the following figure:

Figure 33. Download and view data set members using the CLI

6. Next, we will submit the job in member Z80462.JCL(HELLO). To submit the job, wait for it to complete, and view all spool content, issue:

```
user@ubuntu-base:~$ zowe files list ds "Z80462.*"
Z80462.CBL
Z80462.DATA
Z80462.JCL
Z80462.LOAD
Z80462.S0W1.ISPF.ISPPROF
user@ubuntu-base:~$ zowe files list am "Z80462.CBL"
ADDAMT
CBL0001
CBL0002
CBL0004
CBL0005
CBL0006
CBL0007
CBL0008
CBL0009
CBL0010
CBL0011
CBL0012
COBOL
HELLO
PAYROL0X
PAYROL00
```

Figure 89:

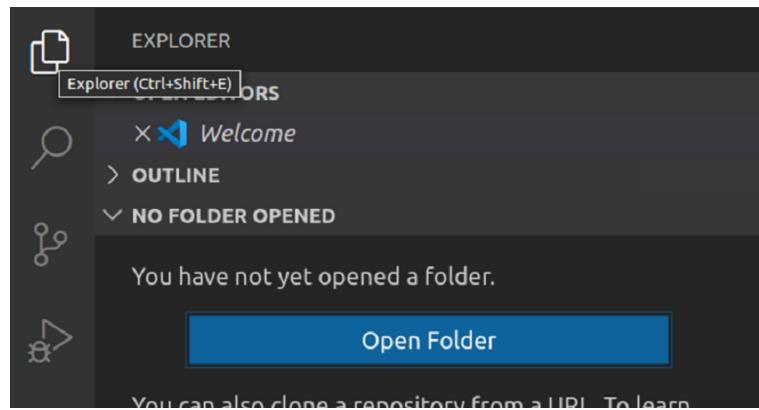


Figure 90:

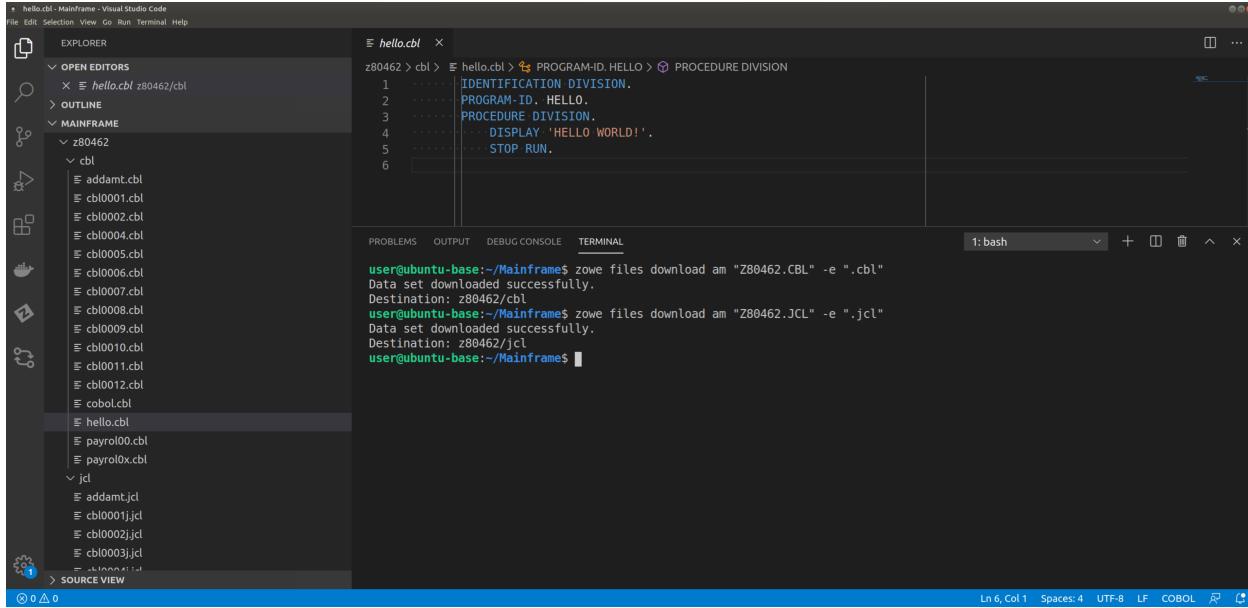


Figure 91:

```
zowe jobs submit ds "Z80462.JCL(HELLO)" --vasc
```

We could also perform this step in piecemeal to get the output from a specific spool file. See the next figure for an example of the upcoming commands. To submit the job and wait for it to enter OUTPUT status, issue:

```
zowe jobs submit ds "Z80462.JCL(HELLO)" --wfo
```

To list spool files associated with this job id, issue:

```
zowe jobs list sfbj JOB00906
```

where JOB00906 was returned from the previous command.

To view a specific spool file (COBRUN:SYSOUT), issue:

```
zowe jobs view sfbi JOB00906 105
```

where JOB00906 and 105 are obtained from the previous commands.

Figure 34. Submit a job, wait for it to complete, then list spool files for the job, and view a specific spool file

If desired, you can also easily submit a job, wait for it to complete, and download the spool content using the following command (see the following figure for the completed state):

```
zowe jobs submit ds "Z80462.JCL(HELLO)" -d .
```

Figure 35. Submit a job, wait for it to complete, download and view spool files

The Zowe CLI was built with scripting in mind. For example, you can use the **--rfj** flag to receive output in JSON format for easy parsing. See the next figure for an example.

Figure 36. The --rfj flag allows for easy programmatic usage

```

user@ubuntu-base:~/Mainframe$ zowe jobs submit ds "Z80462.JCL(HELLO)" --wfo
jobid: JOB00906
retcode: CC 0000
jobname: HELLOCBL
status: OUTPUT
user@ubuntu-base:~/Mainframe$ zowe jobs list sfbj JOB00906
2 JESMSGLG      JES2
3 JESJCL       JES2
4 JESYMSG      JES2
101 SYSPRINT COBOL COBRUN
102 SYSPRINT LKED COBRUN
105 SYSOUT    GO   COBRUN
user@ubuntu-base:~/Mainframe$ zowe jobs view sfbi JOB00906 105
HELLO WORLD!

```

Figure 92:

The screenshot shows a terminal window with the following content:

```

EXPLORER
  OPEN EDITORS: SYSOUT.txt (JOB00915/GO/COBRUN)
  OUTLINE
  MAINFRAME:
    JOB00915:
      COBOL
      GO / COBRUN
      SYSOUT.txt (selected)
      JES2
      LKED
    z80462

SYSOUT.txt X
JOB00915 > GO > COBRUN > SYSOUT.txt
1 | HELLO WORLD!
2

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
user@ubuntu-base:~/Mainframe$ zowe jobs submit ds "Z80462.JCL(HELLO)" -d .
jobid: JOB00915
retcode: CC 0000
jobname: HELLOCBL
status: OUTPUT
Successfully downloaded output to ../../JOB00915
user@ubuntu-base:~/Mainframe$ 

```

Figure 93:

```

user@ubuntu-base:~/Mainframe$ zowe jobs submit data-set "Z80462.JCL(HELLO)" -d . --rfj
{
  "success": true,
  "exitCode": 0,
  "message": "Submitted JCL contained in \"dataset\": \"Z80462.JCL(HELLO)\"",
  "stdout": "\u001b[33mjobid: \u001b[39m JOB01108\n\u001b[33mretcode: \u001b[39mCC 0000\n\u001b[33mjobname: OUTPUT\nSuccessfully downloaded output to ./JOB01108",
  "stderr": "",
  "data": {
    "owner": "Z80462",
    "phase": 20,
    "subsystem": "JES2",
    "phase-name": "Job is on the hard copy queue",
    "job-correlator": "J0001108SVSCJES2D7D0B7F3.....",
    "type": "JOB",
    "url": "https://192.86.32.250:10443/zosmf/restjobs/jobs/J0001108SVSCJES2D7D0B7F3.....%3A",
    "jobid": "JOB01108",
    "class": "A",
    "files-url": "https://192.86.32.250:10443/zosmf/restjobs/jobs/J0001108SVSCJES2D7D0B7F3.....%3A/files",
    "jobname": "HELLOCBL",
    "status": "OUTPUT",
    "retcode": "CC 0000"
  }
}

```

Figure 94:

8.7.2 Zowe CLI - Programmatic Usage

In this section, we will leverage the Zowe CLI programmatically to automate submitting the JCL to compile, link, and run the COBOL program and downloading the spool output. Once you have the content locally you could use any number of distributed scripting and testing tools to eliminate the need to manually review the spool content itself. Historically, in Mainframe we use REXX EXEC etc. for automation, but today we are going to use CLI and distributed tooling.

1. Since we already have Node and npm installed, let's just create a node project for our automation. To initialize a project, issue `npm init` in your project's folder and follow the prompts. You can accept the defaults by just pressing enter. Only the description and author fields should be changed. See the following figure.

Figure 37. Use of `npm init` to create `package.json` for the project

2. Now that we have our `package.json` simply replace the `test` script with a `clg` script that runs the zowe command (replace Z80462 with your high level qualifier):

```

zowe jobs submit ds 'Z80462.JCL(HELLO)' -d .

```

You can name the script whatever you want. I only suggested `clg` because the CLG in the `IGYWCLG` proc (which is what the JCL leverages) stands for compile, link, go. Now, simply issue `npm run clg` in your terminal to leverage the automation to compile, link and run the COBOL program and download the output for review. An example of the completed `package.json` and command execution are shown in the following figure.

Figure 38. Final `package.json` and `npm run clg` execution

3. If you prefer a graphical trigger, you can leverage VS Code as shown in the following figure. Essentially, the CLI enables you to quickly build your own buttons for your custom z/OS tasks. You could also invoke a script rather than a single command to accomodate more complex scenarios.

Figure 39. `clg` task triggered via button

```
user@ubuntu-base:~/Mainframe$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (mainframe)
version: (1.0.0)
description: Automation for COBOL Program
entry point: (index.js)
test command:
git repository:
keywords:
author: Michael Bauer
license: (ISC)
About to write to /home/user/Mainframe/package.json:

{
  "name": "mainframe",
  "version": "1.0.0",
  "description": "Automation for COBOL Program",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "author": "Michael Bauer",
  "license": "ISC"
}
```

Figure 95:

The screenshot shows the VS Code interface with the following details:

- EXPLORER** sidebar: Shows a tree view of files and folders. The 'MAINFRAME' folder is expanded, showing sub-folders like 'JOB01117', 'JOB01119', etc., and a file 'JOB02791'. This file is currently selected.
- PACKAGE.JSON** editor: Displays the contents of the package.json file. The 'scripts' section includes a 'clg' command that runs 'zowe jobs submit ds 'Z80462.JCL(HELLO)' -d .'.
- TERMINAL**: Shows the command being run: `user@ubuntu-base:~/Mainframe$ npm run clg`. The output of the command is displayed below, showing job details and a success message: `jobid: JOB02791
retcode: CC 0000
jobname: HELLOCBL
status: OUTPUT
Successfully downloaded output to ./JOB02791`.

Figure 96:

The screenshot shows the VS Code interface with the following details:

- EXPLORER** sidebar: Shows a tree view of files and folders. The 'MAINFRAME' folder is expanded, showing sub-folders like 'JOB01153', 'JOB01173', etc., and a file 'JOB02807'. This file is currently selected.
- PACKAGE.JSON** editor: Displays the contents of the package.json file. The 'scripts' section includes a 'clg' command.
- TERMINAL**: Shows the command being run: `> Executing task: npm run clg <`. The output of the command is displayed below, showing job details and a success message: `jobid: JOB02807
retcode: CC 0000
jobname: HELLOCBL
status: OUTPUT
Successfully downloaded output to ./JOB02807`. A note at the bottom of the terminal window states: `Terminal will be reused by tasks, press any key to close it.`

Figure 97:

9 Data division

Understanding COBOL variables and program processing of variables are essential to effectively learning the COBOL language. An experienced COBOL programmer must master characteristics of COBOL variables and the program processing using the variables introduced in this chapter. The objective is to introduce the reader to the basics of COBOL variables while exposing the reader to the many advanced COBOL variable options.

Following this chapter is a lab available to compile and execute the COBOL source code provided later in the chapter. Following the successful compile and execution of one provided program, a second provided COBOL program with a minor change is available to compile. The second program has an embedded error and on compile will fail. The failed compilation is an opportunity to identify the error associated with the significance of PICTURE clause data types associated with the operation of the COMPUTE statement (discussed in this chapter) and how to solve the error.

- **Variables / Data-items**
 - Variable / Data-item name restrictions and data types
- **PICTURE clause**
 - PIC clause symbols and data types
 - Coding COBOL variable / data-item names
 - PICTURE clause character-string representation
- **Literals**
 - Figurative constants
 - Data relationships
 - Levels of data
- **MOVE and COMPUTE**
- **Lab**

9.1 Variables / Data-items

A COBOL variable, also known as a data-item, is a name and is chosen by the COBOL programmer. The named variable is coded to hold data where the data value can vary, hence the generic term ‘variable’. A COBOL variable name is also known as ‘Data Name’. A COBOL variable name has restrictions.

9.1.1 Variable / Data-item name restrictions and data types

A list of COBOL variable name restrictions or rules are:

- Must not be a COBOL reserved word.
- Must not contain a space as a part of the name.
- Name contains letters (A-Z), digits (0-9), underscores (_) and hyphens (-).
- Maximum length of 30 characters.
- A hyphen cannot appear as the first or last character.
- An underscore cannot appear as the first character.

Note : A full list of COBOL reserved words can be found in the Enterprise COBOL Language Reference, Appendix E.

When COBOL source code is compiled into an executable program, the COBOL compiler is expecting a named COBOL variable to possess attributes such as a length and data type. During program execution, the variable represents a defined area of processing memory where the memory location has a maximum length and designated data type.

A list of the most common COBOL data types are:

- Numeric (0-9)
- Alphabetic (A-Z), (a-z), or a space
- Alphanumeric Numeric and Alphabetic Combination

9.2 PICTURE clause

The COBOL reserved word, PICTURE (PIC), determines the length and data type of a programmer selected variable name. Data types described by PIC are commonly referred to as a picture clause or pic clause. Some simple pic clauses are:

- PIC 9 - single numeric value where length is one
- PIC 9(4) - four numeric values where length is four
- PIC X - single alphanumeric (character) value where length is one
- PIC X(4) - four alphanumeric values where length is four

9.2.1 PIC clause symbols and data types

The maximum length of a picture clause is dependent upon the data type and compiler options. The PIC reserved word has many more data types beyond numeric (PIC 9) and alphanumeric (PIC X). As an example, an alphabetic only data type is defined as PIC A. Other PIC clause symbols are:

B E G N P S U V Z O / + - , . * CR DB cs

Where cs is any valid currency symbols such as the dollar sign (\$).

All PIC clause symbols are described in the [Enterprise COBOL for z/OS Language Reference manual](#).

9.2.2 Coding COBOL variable / data-item names

A PIC clause describes the data type of a variable/data-item name. Coding a variable/data-item is done in the DATA DIVISION. The COBOL code describing a variable/data-item name is accomplished using a level number and a picture clause.

- Level number - A hierarchy of fields in a record.
- Variable name / Data-item name - Assigns a name to each field to be referenced in the program and must be unique within the program.
- Picture clause - For data type checking.

Figure 1. below is an example of COBOL level numbers with respective variable/data-item names and picture clause.

9.2.3 PICTURE clause character-string representation

Some PIC clause symbols can appear only once in a PIC clause character-string, while other can appear more than once. For example:

- PIC clause to hold value 1123.45 is coded as follows, where the V represents the decimal position.

PIC 9(4)V99

- PIC clause for a value such as \$1,123.45 is coded as follows:

PIC \$9,999V99

9.3 Literals

A COBOL literal is constant data value, meaning the value will not change like a variable can. The COBOL statement, DISPLAY "HELLO WORLD!", is a COBOL reserved word, DISPLAY , followed by a literal, HELLO WORLD!

9.3.1 Figurative constants

Figurative constants are reserved words that name and refer to specific constant values. Examples of figurative constants are:

- ZERO, ZEROS, ZEROES
- SPACE, SPACES
- HIGH-VALUE, HIGH-VALUES
- LOW-VALUE, LOW-VALUES
- QUOTE, QUOTES
- NULL, NULLS

9.3.2 Data relationships

The relationships among all data to be used in a program is defined in the DATA DIVISION, through a system of level indicators and level-numbers. A level indicator, with its descriptive entry, identifies each file in a program. Level indicators represent the highest level of any data hierarchy with which they are associated. A level-number, with its descriptive entry, indicates the properties of specific data. Level-numbers can be used to describe a data hierarchy; they can indicate that this data has a special purpose.

9.3.2.1 Level numbers

A structured level number hierachic relationship is available to all DATA DIVISION sections. Figure 1. shows the level number hierachic relationship with programmer chosen level numbers, variable names and PIC clauses in the File Section where “01 PRINT-REC” references the following “05”-level group of variables and the “01 ACCT-FIELDS” references the following “05”-level group of variables. Observe 05-level CLIENT-ADDR is further subdivided into several 10-level names. COBOL code referencing the name CLIENT-ADDR includes the 10-level names.

Figure 1. Level number hierachic relationship

```

-----+
DATA DIVISION.
*-----+
FILE SECTION.
FD PRINT-LINE RECORDING MODE F.
01 PRINT-REC.
    05 ACCT-NO-0      PIC X(8).
    05 ACCT-LIMIT-0   PIC $$,$$$,$$9.99.
    05 ACCT-BALANCE-0 PIC $$,$$$,$$9.99.
* PIC $$,$$$,$$9.99 -- Alternative for PIC on chapter 7.2.3,
* using $ to allow values of different amounts of digits
* and .99 instead of v99 to allow period display on output
    05 LAST-NAME-0   PIC X(20).
    05 FIRST-NAME-0  PIC X(15).
    05 COMMENTS-0    PIC X(50).
* since the level 05 is higher than level 01,
* all variables belong to PRINT-REC (see chapter 7.3.3)
*
FD ACCT-REC RECORDING MODE F.
01 ACCT-FIELDS.
    05 ACCT-NO          PIC X(8).
    05 ACCT-LIMIT        PIC S9(7)V99 COMP-3.
    05 ACCT-BALANCE      PIC S9(7)V99 COMP-3.
* PIC S9(7)V99 -- seven-digit plus a sign digit value
* COMP-3 -- packed BCD (binary coded decimal) representation
    05 LAST-NAME         PIC X(20).
    05 FIRST-NAME        PIC X(15).
    05 CLIENT-ADDR.
        10 STREET-ADDR     PIC X(25).
        10 CITY-COUNTY     PIC X(20).
        10 USA-STATE        PIC X(15).
    05 RESERVED          PIC X(7).
    05 COMMENTS          PIC X(50).
*
WORKING-STORAGE SECTION.
01 FLAGS.

```

Figure 98:

9.3.3 Levels of data

After a record is defined, it can be subdivided to provide more detailed data references as seen in Figure 1. A level number is a one-digit or two-digit integer between 01 and 49, or one of three special level numbers: 66, 77, or 88 where the variable names are assigned attributes different from the 01-49-level numbers. The relationship between level numbers within a group item defines the hierarchy of data within that group. A group item includes all group and elementary items that follow it until a level number less than or equal to the level number of that group is encountered.

9.4 MOVE and COMPUTE

MOVE and COMPUTE reserved word statements alter the value of variable names. Each MOVE shown in Figure 2. results in a literal stored in a 77-level variable name. The COMPUTE statement, also shown in Figure 2. , stores the value of HOURS * RATE in GROSS-PAY. All three variable names are assigned a numeric value data type using PIC 9, which is necessary for the operation of the COMPUTE statement.

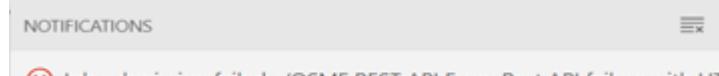
Figure 2. MOVE and COMPUTE example

9.5 Lab

Note : It may take a few seconds to load in all segments of this lab. If files are not loading, hit the refresh button on the list that appears when hovering over the section bar.

1. View the PAYROL00 COBOL source code member in the ‘id’.CBL data set.
2. Submit the JCL member, PAYROL00, from the id.JCL, where id is your id,dropdown. This is where id.JCL(PAYROL00) compiles and successfully executes the PAYROL00. program.

Figure 3. Submit PAYROL00 job



Note : If you receive this error message after submitting the job:

That is because you submitted the job from the .CBL data set and not the .JCL data set.

3. View both compile and execution of PAYROL00 job output, referenced in Figure 4.

Figure 4. PAYROL00 output

4. Next, view PAYROL0X COBOL source code member in id.CBL data set.
5. View and submit the JCL member, PAYROL0X, from the id.JCL dropdown. This is where id.JCL(PAYROL0X) compiles and executes the PAYROL0X program.
6. View the compile of PAYROL0X job output, notice there is no execution output.

Do you notice a difference between this compile and the previous job compile shown in Figure 5. ?

Figure 5. Compare job compiles

The difference is the return/completion code associated with each job output, located both next to the job output name within the JOBS section as shown above, or at the end of the compile output as, 0Return code ##. A return code of 12 means there was an error, but how do we know what that error was? Continue to find out!

7. Observe the text associated with IGYPA3146-S on line 137 within the job output (compile), illustrated in Figure 6.

```

*          COBOL reference format (Figure 1., page 32)
*Columns:   1       2       3       4       5       6       7
*89012345678901234567890123456789012345678901234567890123456789012
*<A-><-----B----->
*Area           Area
*---Sequence Number Area           Identification Area--->
*-----  

IDENTIFICATION DIVISION.  

*-----  

PROGRAM-ID. PAYROL00.  

*-----  

DATA DIVISION.  

*-----  

WORKING-STORAGE SECTION.  

***** Variables for the report  

* level number  

* | variable name  

* | picture clause  

* V V V  

77 WHO      PIC X(15).  

77 WHERE    PIC X(20).  

77 WHY      PIC X(30).  

77 RATE     PIC 9(3).  

77 HOURS    PIC 9(3).  

77 GROSS-PAY PIC 9(5).  

* PIC X(15) -- fifteen alphanumeric characters  

* PIC 9(3) -- three-digit value  

*-----  

PROCEDURE DIVISION.  

*-----  

***** COBOL MOVE statements - Literal Text to Variables  

MOVE "Captain COBOL" TO WHO.  

MOVE "San Jose, California" TO WHERE.  

MOVE "Learn to be a COBOL expert" TO WHY.  

MOVE 19 TO HOURS.  

MOVE 23 TO RATE.  

* The string "Captain COBOL" only contains 13 characters,  

* the remaining positions of variable WHO are filled with spaces  

* The value 19 only needs 2 digits,  

* the leftmost position of variable HOURS is filled with zero  

***** Calculation using COMPUTE reserved word verb  

COMPUTE GROSS-PAY = HOURS * RATE.  

* The result of the multiplication only needs 3 digits,  

* the remaining leftmost positions are filled with zeroes  

***** DISPLAY statements  

DISPLAY "Name: " WHO.  

DISPLAY "Location: " WHERE  

DISPLAY "Reason: " WHY  

DISPLAY "Hours Worked: " HOURS.  

DISPLAY "Hourly Rate: " RATE.  

DISPLAY "Gross Pay: " GROSS-PAY.  

DISPLAY WHY " from " WHO.  

GOBACK.

```

Figure 99:
84

The screenshot shows a Microsoft Visual Studio Code window with the following details:

- Title Bar:** 299995.CB\PARROD01.ps1 - Visual Studio Code
- File Path:** C:\Windows\system32\WindowsPowerShell\v1.0\parrodo1.ps1
- Code Content:**

```
param([string]$name,[string]$location,[int]$hoursWorked,[float]$hourlyRate,[float]$grossPay)

$rate = $hourlyRate * $hoursWorked
$tax = $rate * .15
$netPay = $rate - $tax

Write-Output "Hello $name"
Write-Output "Your location is $location"
Write-Output "You worked $hoursWorked hours"
Write-Output "Your hourly rate is $hourlyRate"
Write-Output "Your gross pay is $grossPay"
Write-Output "Your net pay is $netPay"
```
- Annotations:** A red box highlights the first line of code, and another red box highlights the entire parameter block. A red arrow points from the bottom right towards the parameter block.
- Status Bar:** Job submitted: J0020208

Figure 100:



Figure 101:



Figure 102:

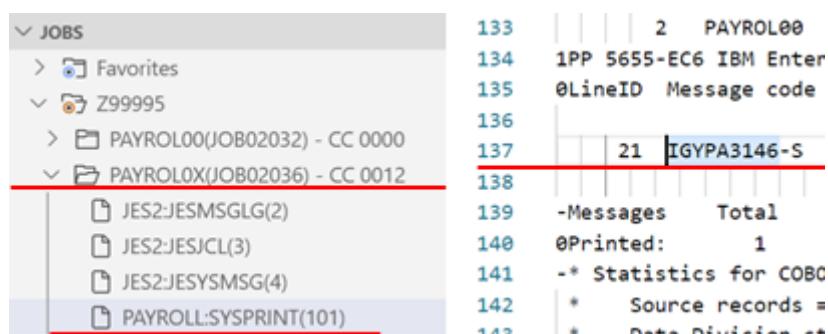


Figure 103:

Figure 6. IGYPA3146-S message

Notice that this line tells you to focus on the GROSS-PAY picture clause in order to identify the problem. Use this information, modify the PAYROL0X COBOL source code to fix the error. Be sure you are editing the correct code.

8. After modifying, re-submit the PAYROL0X JCL to verify the problem has been identified and corrected, resulting in a successful compile and execution with a return code of zero, shown in Figure 7.

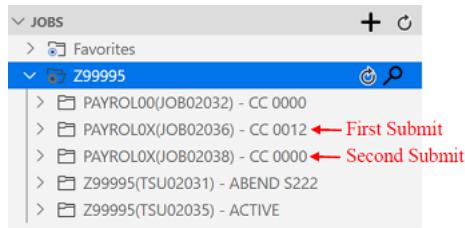


Figure 104:

Figure 7. Compare return codes

10 File handling

The previous chapter and lab focused on variables and moving literals into variables, then writing variable content using the COBOL DISPLAY statement. This section introduces reading records from files into variables, moving the variables to output variables, and writing the output variables to a different file. A simple COBOL program to read each record from a file and write each record to a different file is used to illustrate COBOL code necessary to read records from an input external data source and write records to an output external data source.

An experienced COBOL programmer can answer the question, “How does an Enterprise COBOL program read data from an input external data source and write data to an output external data source?” The objective of this chapter is to provide enough comprehensive information for the reader to be able to answer that question.

- COBOL code used for sequential file handling
 - COBOL inputs and outputs
 - FILE-CONTROL paragraph
 - COBOL external data source
 - Data sets, records, and fields
 - Blocks
 - ASSIGN clause
- PROCEDURE DIVISION sequential file handling
 - Open input and output for read and write
 - Close input and output
- COBOL programming techniques to read and write records sequentially
 - READ-NEXT-RECORD paragraph execution
 - READ-RECORD paragraph
 - WRITE-RECORD paragraph
 - Iterative processing of READ-NEXT-RECORD paragraph
- Lab

10.1 COBOL code used for sequential file handling

COBOL code used for sequential file handling involves:

- ENVIRONMENT DIVISION.
 - SELECT clauses
 - ASSIGN clauses
- DATA DIVISION.
 - FD statements
- PROCEDURE DIVISION.
 - OPEN statements
 - CLOSE statements

- READ INTO statement
- WRITE FROM statement

10.1.1 COBOL inputs and outputs

The ENVIRONMENT DIVISION and DATA DIVISION describes the inputs and outputs used in the PROCEDURE DIVISION program logic. Previous chapters introduced variable descriptions in the DATA DIVISION and literals were moved into the defined variables. The role of the ENVIRONMENT DIVISION and more specifically, the INPUT-OUTPUT SECTION, FILE-CONTROL paragraph introduces accessing external data sources where the data from external sources are moved into defined variables.

10.1.2 FILE-CONTROL paragraph

The FILE-CONTROL paragraph associates each COBOL internal file name with an external dataset name. Within the FILE-CONTROL paragraph, the SELECT clause creates an internal file name and the ASSIGN clause creates an external dataset name. Figure 1. shows the PRINT-LINE internal file name associated with the PRTLINE external dataset name and the ACCT-REC internal file name associated with the ACCTREC external dataset name. Section titled Assign Clause further explains the SELECT ASSIGN TO relationship.

```
*-----  
ENVIRONMENT DIVISION.  
*-----  
INPUT-OUTPUT SECTION.  
FILE-CONTROL.  
  SELECT PRINT-LINE ASSIGN TO PRTLINE.  
  SELECT ACCT-REC  ASSIGN TO ACCTREC.  
*SELECT clause creates an internal file name  
*ASSIGN clause creates a name for an external data source,  
*which is associated with the JCL DDNAME used by the z/OS  
*e.g. ACCTREC is linked in JCL file CBL0001J to &SYSUID..DATA  
*where &SYSUID. stands for Your z/OS user id  
*e.g. if Your user id is 254321,  
*the data set used for ACCTREC is 254321.DATA
```

Figure 105:

Figure 1. FILE-CONTROL

While SELECT gives a name to an internal file and ASSIGN gives a name to the external dataset name, a COBOL program needs more information about both. The COBOL compiler is given more information about both in the DATA DIVISION, FILE SECTION.

The COBOL reserved word ‘FD’ is used to give the COBOL compiler more information about internal file names in the FILE-SECTION. The code below the FD statement is the record layout. The record layout consists of level numbers, variable names, data types, and lengths as shown in Figure 2.

Figure 2. FILE-SECTION

10.1.3 COBOL external data source

Enterprise COBOL source code compiles and executes on IBM Z Mainframe hardware where z/OS is the operating system software. z/OS stores data in both data sets and Unix files. z/OS includes many data

```

*-----*
DATA DIVISION.
*-----*
FILE SECTION.
FD PRINT-LINE RECORDING MODE F.
01 PRINT-REC.
  05 ACCT-NO-0      PIC X(8).
  05 ACCT-LIMIT-0   PIC $$,$$$,$$9.99.
  05 ACCT-BALANCE-0 PIC $$,$$$,$$9.99.
* PIC $$,$$$,$$9.99 -- Alternative for PIC on chapter 7.2.3,
* using $ to allow values of different amounts of digits
* and .99 instead of v99 to allow period display on output
  05 LAST-NAME-0    PIC X(20).
  05 FIRST-NAME-0   PIC X(15).
  05 COMMENTS-0     PIC X(50).
* since the level 05 is higher than level 01,
* all variables belong to PRINT-REC (see chapter 7.3.3)

```

Figure 106:

storage methods. This chapter will focus on the z/OS sequential data storage method. A sequential dataset is a collection of records.

10.1.4 Data sets, records, and fields

A dataset has many records. A record is a single line in the dataset and has a defined length. Each record can be subdivided into fields where each field has a defined length. Therefore, the sum of all field lengths would equal the length of the record. Observe Figure 3.

10.1.5 Blocks

Each record read by the program can result in disk storage access. A program typically reads 1 record at a time in sequential order until all records are read. When a record is read, the record retrieved from disk is stored in memory for program access. When each next record read requires the need to retrieve the record from disk, system performance is impacted negatively. Records can be blocked where a block is a group of records. The result is when the first record is read, then an entire block of records is read into memory assuming the program will be reading the second, third, etc. records avoiding unnecessary disk retrievals and negative system performance. The memory holding a record or block of records to be read by the program is known as a buffer. COBOL BLOCK CONTAINS clause is available to specify the size of the block in the buffer. Observe Figure 3.

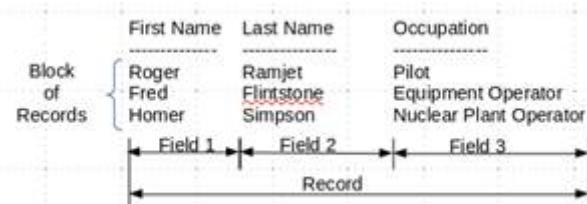


Figure 107:

Figure 3. Records, fields, and blocks

10.1.6 ASSIGN clause

While the SELECT clause name is an internal file name, the ASSIGN clause name is describing a data source external to the program. z/OS uses Job Control Language, JCL, operations to tell the system what program to load and execute followed by input and output names needed by the program. The JCL input and output names are called DDNAMEs. The JCL DDNAME statement includes a JCL DD operation where DD is an abbreviation for Data Definition. On the same DDNAME statement is the system-controlled data set name.

COBOL code “SELECT ACCT-REC ASSIGN TO ACCTREC” requires a JCL DDNAME ACCTREC with a DD redirecting ACCTREC to a z/OS controlled dataset name, MY.DATA. The COBOL program is shown in Example 1.

The purpose of the redirection of ACCT-REC, via ASSIGN TO, to JCL DDNAME, ACCTREC is flexibility. ACCT-REC is used in the program itself, ACCTREC is a bridge to JCL, shown in Example 1., and a DD JCL statement links ACCTREC to an actual dataset, shown in Example 2. This flexibility allows the same COBOL program to access a different data source with a simple JCL modification avoiding requirement to change the source code to reference the alternate data source.

```
SELECT ACCT-REC ASSIGN TO **ACCTREC**
```

Example 1. COBOL program

The JCL statement required by the compiled COBOL program during execution to redirect ACCTREC to the MY.DATA z/OS controlled dataset is shown in Example 2.

```
//**ACCTREC** DD DSN=MY.DATA,DISP=SHR
```

Example 2. JCL statement

In summary, ACCT-REC is the internal file name. ACCTREC is the external name where a JCL DDNAME must match the COBOL ASSIGN TO ACCTREC name. At program execution the JCL ACCTREC DDNAME statement is redirected to the dataset name identified immediately after the JCL DD operation.

```
ACCT-REC >>> ACCTREC >>> //ACCTREC >>> DD >>> MY.DATA
```

As a result, the COBOL internal ACCT-REC file name reads data records from sequential dataset named MY.DATA.

JCL is a separate z/OS technical skill. The introduction to COBOL explains just enough about JCL to understand how the COBOL internal file name locates the external sequential dataset name. To read more on JCL, visit the IBM Knowledge Center:

https://www.ibm.com/support/knowledgecenter/zosbasics/com.ibm.zos.zjcl/zjclc_basicjclconcepts.htm

10.2 PROCEDURE DIVISION sequential file handling

During COBOL program runtime, SELECT ASSIGN TO a JCL DDNAME is mandatory. If the ASSIGN TO name fails to associate with a JCL DDNAME of the same spelling, at runtime, then a program runtime error occurs when the OPEN operation is attempted. A message appears in the runtime output indicating the DDNAME was not found. READ and WRITE are dependent upon successful completion of the OPEN operation. The compiler cannot detect the runtime error because the compiler is unaware of the actual runtime JCL DDNAME dataset name subject to OPEN, READ, or WRITE. The FD, File Descriptor, mapping of the data record fields requires a successful OPEN to be populated by subsequent READ or WRITE operations.

10.2.1 Open input and output for read and write

COBOL inputs and outputs must be opened to connect the selected internal name to the assigned external name. Figure 4. opens the file name ACCT-REC as program input and file name PRINT-LINE as program output.

```
*-----  
ENVIRONMENT DIVISION.  
*-----  
INPUT-OUTPUT SECTION.  
FILE-CONTROL.  
    SELECT PRINT-LINE ASSIGN TO PRTLINE.  
    SELECT ACCT-REC  ASSIGN TO ACCTREC.  
*SELECT clause creates an internal file name  
*ASSIGN clause creates a name for an external data source,  
*which is associated with the JCL DDNAME used by the z/OS  
*e.g. ACCTREC is linked in JCL file CBL0001J to &SYSUID..DATA  
*where &SYSUID. stands for Your z/OS user id  
*e.g. if Your user id is 254321,  
*the data set used for ACCTREC is 254321.DATA
```

Figure 108:

Figure 4. OPEN-FILES

10.2.2 Close input and output

COBOL inputs and outputs should be closed at program completion or better yet when the program is done reading from or writing to the internal file name. Figure 5. closes the internal file name ACCT-REC and internal file name PRINT-LINE, then stops processing, STOP RUN.

```
*-----  
CLOSE-STOP.  
CLOSE ACCT-REC.  
CLOSE PRINT-LINE.  
STOP RUN.
```

Figure 109:

Figure 5. CLOSE-STOP

10.3 COBOL programming techniques to read and write records sequentially

When reading records, the program needs to first check for no records to be read or check for no more records to be read. If a record exists, then the fields in the read record populate variable names defined by the FD clause. COBOL uses a PERFORM statement for iteration. In computer programming, iterative is used to describe a situation in which a sequence of instructions or statements can be executed multiple times. One pass through the sequence is called an iteration. Iterative execution is also called a loop. In other programming languages, 'DO' or 'FOR' statements are used for iterative execution. COBOL uses a PERFORM statement for iterative execution. Figure 6. shows four programmer chosen paragraph names in the PROCEDURE DIVISION.

- READ-NEXT-RECORD

- CLOSE-STOP
- READ-RECORD
- WRITE-RECORD

READ-NEXT-RECORD repeatedly executes READ-RECORD and WRITE-RECORD until a last record is encountered. When the last record is encountered, then CLOSE-STOP is executed stopping the program.

```

READ-NEXT-RECORD.
  PERFORM READ-RECORD
*   The previous statement is needed before entering the loop.
*   Both the loop condition LASTREC = 'Y'
*   and the call to WRITE-RECORD depend on READ-RECORD having
*   been executed before.
*   The loop starts at the next line with PERFORM UNTIL
  PERFORM UNTIL LASTREC = 'Y'
    PERFORM WRITE-RECORD
    PERFORM READ-RECORD
  END-PERFORM
*
  CLOSE-STOP.
    CLOSE ACCT-REC.
    CLOSE PRINT-LINE.
    STOP RUN.
*
  READ-RECORD.
    READ ACCT-REC
    AT END MOVE 'Y' TO LASTREC
  END-READ.
*
  WRITE-RECORD.
    MOVE ACCT-NO      TO ACCT-NO-O.
    MOVE ACCT-LIMIT   TO ACCT-LIMIT-O.
    MOVE ACCT-BALANCE TO ACCT-BALANCE-O.
    MOVE LAST-NAME    TO LAST-NAME-O.
    MOVE FIRST-NAME   TO FIRST-NAME-O.
    MOVE COMMENTS     TO COMMENTS-O.
  WRITE PRINT-REC.

```

Figure 110:

Figure 6. Reading and writing records

Note: COBOL is English-like and COBOL reserved words are English-like. The programmer is free to use English-like variable names to help remember the purpose of the variable names. The PROCEDURE DIVISION structure is English-like. A paragraph contains one or more sentences. A sentence contains one or more statements. The implicit scope terminator, a period (.), terminates a sentence or terminates several consecutive statements which would be analogous to a compounded sentence where 'and' joins potentially independent sentences together. ###

10.3.1 READ-NEXT-RECORD paragraph execution

The READ-NEXT-RECORD paragraph is a COBOL programming technique used to read all records from a sequential file UNTIL the last record is read. The paragraph contains a compounded sentence

terminated by an implicit scope terminator, (.) period, on a separate line following the END-PERFORM statement. The PERFORM UNTIL through END-PERFORM, explicit scope terminator, is repeatedly executed until LASTREC variable contains Y. The first PERFORM READ-RECORD results in a branch to the READ-RECORD paragraph. Observe #1 in Figure 7.

10.3.2 READ-RECORD paragraph

The READ-RECORD paragraph executes the COBOL READ statement resulting in the external sequential file populating the variables associated with ACCT-REC internal file name. If ‘AT END’ of records read, then Y is moved into the LASTREC variable. The READ statement is terminated by an explicit scope terminator, END-READ. The paragraph is terminated by an implicit scope terminator, (.) period. Control is returned to READ-NEXT-RECORD paragraph to execute the next statement, PERFORM WRITE-RECORD.

10.3.3 WRITE-RECORD paragraph

The WRITE-RECORD paragraph contains several sentences terminated by an implicit scope terminator, (.) period. The MOVE statements result in each input file variable name moved to an output file variable name. The last sentence in the paragraph writes the collection of output file variable names, PRINT-REC.

PRINT-REC is assigned to PRTREC. JCL is used to execute the COBOL program. An associated JCL PRTREC DDNAME redirects the written output to a z/OS controlled data set name, etc. using JCL DD operation on the JCL DDNAME statement. Observe #2 in Figure 7.

10.3.4 Iterative processing of READ-NEXT-RECORD paragraph

Once all statements in the WRITE-RECORD paragraph are executed, then control is returned to the READ-NEXT-RECORD paragraph where the next sentence to be executed is the second PERFORM READ-RECORD statement.

Again, the READ-RECORD paragraph executes the COBOL READ statement, resulting in the external sequential file populating the variables associated with ACCT-REC internal file name. If ‘AT END’ of records read, Y is moved into the LASTREC variable, then returns control to READ-NEXT-RECORD paragraph. The READ-NEXT-RECORD paragraph would continue the iterative process UNTIL Y is found in the LASTREC variable. Observe #3 in Figure 7.

Figure 7. Iterative processing

10.4 Lab

The lab associated with this chapter demonstrates the ‘end-of-file’ COBOL coding technique for reading all data records from a sequential file. If a step has an asterisk (*) next to it, it will have a hint associated at the end of the lab content.

1. If not already, open VSCode and select Zowe Explorer from the left sidebar.

Note : If you are opening a new instance of VSCode (i.e. you closed out of it after the previous usage), you may need to ‘Select a filter’ again. You can do so by selecting the search icon  next to your named connection in the DATA SETS section and then reselecting the filter previously used. It should be in the listed filters after you have selected the search symbol.

2. View these COBOL source code members listed in the id.CBL data set:

- CBL0001
- CBL0002

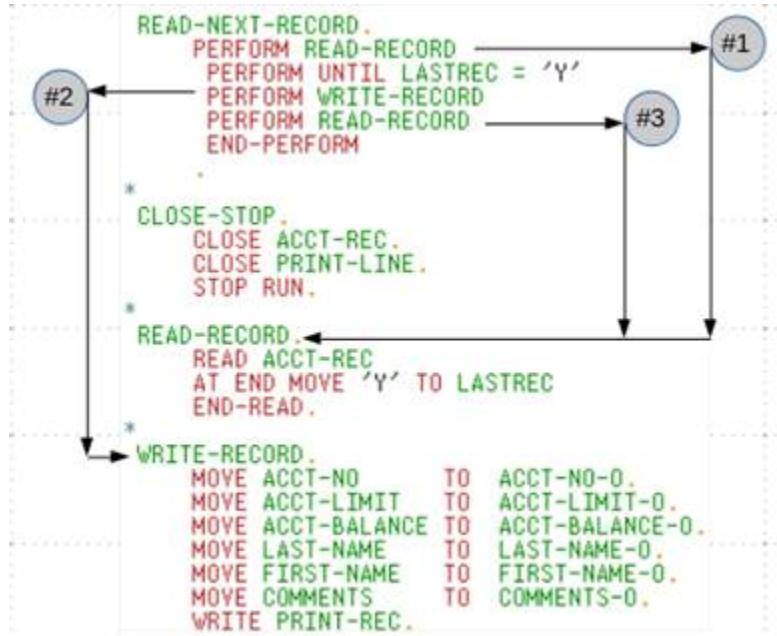


Figure 111:

3. View these three JCL members in the id.JCL data set:

- CBL0001J
- CBL0002J
- CBL0003J

Z9995.JCL(CBL0001J).jcl - Visual Studio Code

```

    Z9995.JCL(CBL0001J).jcl
    1 //CBL0001J JOB 1,NODYN=&SYSPID
    2 //*****
    3 //COBRUN EXEC IGWCL
    4 //COBOL.SYSIN DD DSN=&SYSPID..CBL(CBL0001),DISP=SHR
    5 //LKED.SYSLMOD DD DSN=&SYSPID..LOAD(CBL0001),DISP=SHR
    6 //*****
    7 // IF RC = 0 THEN
    8 //*****
    9 //RUN EXEC PGM=CBL0001
    10 //STEPLIB DD DSN=&SYSPID..LOAD,DISP=SHR
    11 //ACCTREC DD DSN=&SYSPID..DATA,DISP=SHR
    12 //PRTLINE DD SYSOUT=*,OUTLIM=15000
    13 //SYSOUT DD SYSOUT=*,OUTLIM=15000
    14 //CEEDUMP DD DUMMY
    15 //SYSDUMP DD DUMMY
    16 //*****
    17 // ELSE
    18 // ENDIF
    19
  
```

The screenshot shows the Visual Studio Code interface with the Z9995.JCL(CBL0001J).jcl file open. The left sidebar displays a tree view of data sets and jobs, including 'Z9995.JCL' which contains members like 'ADDAMT', 'CBL0001J', 'CBL0002J', 'CBL0003J', 'COBRUN', 'HELLO', and 'PAYROL00'. The main editor area shows the JCL code for job CBL0001J, which includes COBRUN, COBOL, and various DD statements for input, output, and system dump.

Figure 112:

Figure 8. Id.JCL(CBL0001J).jcl

4. Submit job, JCL(CBL0001J), within the DATA SET section.
5. View that job output using the JOBS section.

- COBRUN:SYSPRINT(101) - COBOL program compiler output
- RUN:PRTLINE(103) - COBOL program execution output, shown in Figure 9.

1	17891797	\$10,000.00	\$188.74WASHINGTON	George	longed to retire to his fields at Mount	
2	17971881	\$10,000.00	\$3,188.33ADAMS	John	retired to his farm in Quincy	
3	18011889	\$10,000.00	\$7,008.13JEFFERSON	Thomas	retired to Monticello	
4	18091817	\$10,000.00	\$503.13MADISON	James	retirement at Montpelier	
5	18171825	\$10,000.00	\$31,313.13MONROE	James	Russia must not encroach southward	
6	18251829	\$10,000.00	\$31,250.33ADAMS II	John Quincy	collapsed on the floor of the House fro	
7	18291837	\$10,000.00	\$3,318.38ACKSON	Andrew	that to the victory belong the spoils	
8	18371841	\$10,000.00	\$325.08VAN BUREN	Martin	independent treasury system	
9	18411841	\$10,000.00	\$313.58HARRISON	William Henry	ornate with classical allusions	
10	18411845	\$10,000.00	\$121.65TYLER	John	opposed the Missouri Compromise	
11	18451849	\$10,000.00	\$314.05POLK	James	health undermined from hard work	
12	18491850	\$10,000.00	\$828.20TAYLOR	Zachary	acted at times as though he were above	
13	18501853	\$10,000.00	\$373.10FILLMORE	Millard	signed the Fugitive Slave Act	
14	18531857	\$10,000.00	\$315.07PIERCE	Franklin	he tried to persuade Spain to sell Cuba	
15	18571861	\$10,000.00	\$7.90BUCHANAN	James	widening rift over slavery	
16	18611865	\$100,000.00	\$313.13LINCOLN	Abraham	new birth of freedom	
17	18651869	\$100,000.00	\$603.14JOHNSON	Andrew	freemen were beginning to appear	

Figure 113:

Figure 9. RUN:PRTLINE(103) for JCL(CBL0001J)

6. Submit job, JCL(CBL0002J), within the DATA SET section.
7. View that job output using the JOBS section.
 - COBRUN:SYSPRINT(101) - COBOL program compiler output

Locate COBOL compiler severe message IGYPS2121-S within the output file referred to in step 7, shown in Figure 10.

```
==000074==> IGYPS2121-S "PRINT-REX" was not defined as a data-name. The statement was discarded.
```

Figure 114:

Figure 10. IGYPS2121-S message

8. Edit CBL(CBL0002):
 - Determine appropriate spelling of PRINT-REX, correct it within the source code and save the updated source code.
9. Re-submit job, JCL(CBL0002J), using the DATA SET section and view the output in the JOBS section.
 - COBRUN:SYSPRINT(101) COBOL program compiler output
 - RUN:PRTLINE(103) is the COBOL program execution output (if correction is successful)
10. Submit job, JCL(CBL0003J), using the DATA SET section.
11. View CBL0003J ABENDU4038 output, using the JOBS section:
 - View the IGZ00355 abend message in RUN:SYSOUT(104) from the COBOL program execution output.

- IGZ00355 reads, program is unable to open or close ACCTREC file name, shown in Figure 11. guiding you to the root of the error.

The screenshot shows a Visual Studio Code interface. In the center, there is an error message window titled 'CBL0003J.JOB02349.SYSOUT - Visual Studio Code'. The message content is:

```

1  IGZ00355 There was an unsuccessful OPEN or CLOSE of file ACCTREC in program CBL0001 at relative location X'1A0'.
2  Neither FILE STATUS nor an ERROR declarative were specified. The status code was 35.
3  From compile unit CBL0001 at entry point CBL0001 at compile unit offset +000001A0 at entry offset +000001A0.
4
5

```

To the left of the message window, the left sidebar of the VS Code interface is visible. It shows a tree view of 'JOBS' under 'DATA SETS'. One node, 'RUN:SYSOUT(104)', is highlighted with a red box and has a red arrow pointing from it towards the error message window.

Figure 115:

Figure 11. RUN:SYSOUT(104) message

12. Fix this error by editing JCL(CBL0003J):

- Determine the DDNAME needed, but missing or misspelled.
- Correct it within the code and save

13. Re-submit job, JCL(CBL0003J), using the DATA SET section.

14. View CBL0003J output using the JOBS section, your output should look like Figure 12.

- RUN:PRTLINE - COBOL program execution output (if correction is successful)

The screenshot shows a Visual Studio Code interface with two panes. The left pane shows the 'JOBS' section of the JCL(CBL0003J). The right pane displays the output of the 'RUN:PRTLINE' command, which is a table of historical US Presidents and their achievements:

1	17891797	\$10,000.00	\$188.74WASHINGTON	George	longed to retire to his fields at P		
2	17971881	\$10,000.00	\$3,188.33ADAMS	John	retired to his farm in Quincy		
3	18011809	\$10,000.00	\$7,008.13JEFFERSON	Thomas	retired to Monticello		
4	18091817	\$10,000.00	\$503.13MADISON	James	retirement at Montpelier		
5	18171825	\$10,000.00	\$31,313.33MONROE	James	Russia must not encroach southward		
6	18251829	\$10,000.00	\$31,250.33ADAMS II	John Quincy	collapsed on the floor of the House		
7	18291837	\$10,000.00	\$3,318.38JACKSON	Andrew	that to the victors belong the spo		
8	18371841	\$10,000.00	\$325.00VAN BUREN	Martin	independent treasury system		
9	18411841	\$10,000.00	\$313.50HARRISON	William Henry	ornate with classical allusions		
10	18411845	\$10,000.00	\$121.65TYLER	John	opposed the Missouri Compromise		
11	18451849	\$10,000.00	\$314.05POLK	James	health undermined from hard work		
12	18491850	\$10,000.00	\$828.28TAYLOR	Zachary	acted at times as though he were at		
13	18501853	\$10,000.00	\$373.10FILLMORE	Millard	signed the Fugitive Slave Act		
14	18531857	\$10,000.00	\$315.07PIERCE	Franklin	he tried to persuade Spain to sell		
15	18571861	\$10,000.00	\$7,908BUCHANAN	James	widening rift over slavery		
16	18611865	\$100,000.00	\$313.13LINCOLN	Abraham	new birth of freedom		
17	18651869	\$100,000.00	\$603.14JOHNSON	Andrew	freemen were beginning to appear		
18	18691877	\$100,000.00	\$32,318.36GRANT	Ulysses S.	he brought part of his Army staff t		
19	18771881	\$100,000.00	\$5,860.55HAYES	Rutherford B.	appointments must be made on merit		

Figure 116:

Figure 12. RUN:PRTLINE(103) for JCL(CBL0003J)

Lab hints

13. The error is located on line 11, adjust 'ACCTREX' accordingly.

Figure 13. Error in id.JCL(CBL0003J).jcl

```

10 //STEPLIB    DD DSN=&SYSUID..LOAD,DISP=SHR
11 //ACCTREX   DD DSN=&SYSUID..DATA,DISP=SHR
12 //PRTLINE   DD SYSOUT=*,OUTLIM=15000

```

Figure 117:

11 Program structure

In this chapter we discuss the concept of structured programming and how it relates to COBOL. We highlight the key techniques within the COBOL language that allow you to write good well-structured programs.

- Styles of programming
 - What is structured programming
 - What is Object Orientated Programming
 - COBOL programming style
- Structure of the Procedure Division
 - Program control and flow through a basic program
 - Inline and out of line perform statements
 - Using performs to code a loop
 - Learning bad behavior using the GO TO keyword
- Paragraphs as blocks of code
 - Designing the content of a paragraph
 - Order and naming of paragraphs
- Program control with paragraphs
 - PERFORM TIMES
 - PERFORM THROUGH
 - PERFORM UNTIL
 - PERFORM VARYING
- Using subprograms
 - Specifying the target program
 - Specifying program variables
 - Specifying the return value
- Summary

11.1 Styles of programming

Before we discuss in more detail how to structure a program written in COBOL, it's important to understand the type of language COBOL is and how it's both different from other languages and how it affects the way you might structure your programs.

11.1.1 What is structured programming

Structured programming is the name given to a set of programming styles that could include functional, procedural amongst others. Structured programming technique results in program logic being easier to understand and maintain. Examples of structured programming languages are C, PL/I, Python and of course, COBOL. These languages, given specific control flow structures such as loops, functions and methods, allow a programmer to organize their code in a meaningful way.

Unstructured programming constructs, also known as spaghetti code, are concepts such as GOTO or JUMP which allow the flow of execution to branch wildly around the source code. Such code like this is hard to analyze and read. Although COBOL does contain these structures, it is important to use them sparingly and not as the backbone of well-structured code.

Well-structured code is both easy to understand and to maintain. It is highly likely that at some point in your career you will be required to read and work from someone else's code, often a decade after it was originally written. It would be extremely helpful to you if the original author structured their code well and likewise if it is your code someone else is reading.

11.1.2 What is Object Orientated Programming

Object Orientated programming, or OO programming, differs from structured programming, although it borrows a lot of the same concepts. In OO programming, code is split up into multiple classes, each representing an actor within the system. Each class is made up of variables and a sequence of methods. Instantiations of a class or objects can execute methods of another object. Each class within an OO program can be considered a structured program, as it will still contain methods and iteration constructs. However, it is the composition of the program from a set of individual classes that makes OO programming different. It is possible to write Object Orientated COBOL; however, it is not supported by some of the middleware products that provide COBOL APIs. It is not generally used within the market and so it is not covered in this course.

11.1.3 COBOL programming style

COBOL doesn't directly have some of the components of a structured programming language as you may know them if you have studied a language like C or Java. COBOL doesn't contain for or while loops, nor does it contain defined functions or methods. Because COBOL is meant to be a language that is easy to read these concepts are embodied through the use of the PERFORM keyword and the concept of paragraphs. This allows the programmer to still create these structures, but in a way that is easy to read and follow.

11.2 Structure of the Procedure Division

As you already know, a COBOL program is split into several divisions, including identification, environment and data. However, this chapter concerns itself with how you structure the content of the procedure division to be easy to read, understandable and maintainable in the future.

11.2.1 Program control and flow through a basic program

Typically, execution in a COBOL program begins at the first statement within the procedure division and progresses sequentially through each line until it reaches the end of the source code. For example, take a look at Example 1. Snippet from TOTEN1. This is a simple program that displays a simple message counting to ten.

```

OPEN OUTPUT PRINT-LINE.

MOVE 'THE NUMBER IS: ' TO MSG-HEADER OF PRINT-REC.

ADD 1 TO COUNTER GIVING COUNTER.
MOVE COUNTER TO MSG-TO-WRITE.
WRITE PRINT-REC.

ADD 1 TO COUNTER GIVING COUNTER.
MOVE COUNTER TO MSG-TO-WRITE.
WRITE PRINT-REC.

...
CLOSE PRINT-LINE.
STOP RUN.

```

Example 1. Snippet from TOTEN1

Although this code is very simple to read, it's not very elegant, there is a lot of code repetition as the number is increased. Obviously, we want to provide some structure to the program. There are three keywords that we can use to transfer control to a different section of the source code and provide the structure we need. These keywords are PERFORM, GO TO and CALL.

11.2.2 Inline and out of line perform statements

The PERFORM keyword is a very flexible element of the COBOL language, as it allows functions and loops to be entered. At the most basic level, a PERFORM allows control to be transferred to another section of the code. Once this section has executed, control returns to the following line of code. Take the following example:

```

OPEN OUTPUT PRINT-LINE.

MOVE 'THE NUMBER IS: ' TO MSG-HEADER OF PRINT-REC.

PERFORM WRITE-NEW-RECORD.

CLOSE PRINT-LINE.
STOP RUN.

WRITE-NEW-RECORD.
ADD 1 TO COUNTER GIVING COUNTER
MOVE COUNTER TO MSG-TO-WRITE
WRITE PRINT-REC.

```

Example 2. Snippet from TOTEN2

In this example, the three lines of code that constructed a new line of output and printed it has been extracted into a new paragraph called WRITE-NEW-RECORD. This paragraph is then performed ten times by use of the PERFORM keyword. Each time the PERFORM keyword is used, execution jumps to the paragraph WRITE-NEW-RECORD, executes the three lines contained within that paragraph before returning to the line following the PERFORM statement. The concept of a paragraph will be covered later in this chapter in more depth.

11.2.3 Using performs to code a loop

The code we have built so far is still not optimal, the repetition of the perform statement ten times is inelegant and can be optimized. Observe the following snippet of code:

```
MOVE 'THE NUMBER IS: ' TO MSG-HEADER OF PRINT-REC.  
  
PERFORM VARYING COUNTER FROM 01 BY 1 UNTIL COUNTER EQUAL 11  
MOVE COUNTER TO MSG-TO-WRITE  
WRITE PRINT-REC  
END-PERFORM.  
  
CLOSE PRINT-LINE.  
STOP RUN.
```

Example 3. Snippet from TOTEN2

In this example, we are using the PERFORM keyword in a way that is similar to a for loop in other languages. The loop runs from the PERFORM keyword to the END-PERFORM keyword. Each time execution iterates over the loop, the value of COUNTER is incremented and tested by one. For comparison, the same loop would be written in Java like so:

```
for(int counter =0; counter<11; counter++){  
    //move counter to msg-to-write  
    //write print-rec  
}
```

Example 4. Java example

Although the COBOL version is perhaps more verbose than a for loop in other languages, it is easier to read, and remember you always have autocomplete (if you are using a good editor) to help you with the typing.

11.2.4 Learning bad behavior using the GO TO keyword

Programmers tend to have strong beliefs about choice of editor, tabs or spaces and many heated discussions have been had on such subjects. However, if there is one thing that we can agree on, it is that use of GO TO is usually a bad idea. To demonstrate why GO TO can be a poor idea, we will take a look at TOTEN2 again, and replace the second instance of the PERFORM keyword with a GO TO, shown in Example 5.

```
PERFORM WRITE-NEW-RECORD.  
GO TO WRITE-NEW-RECORD.  
PERFORM WRITE-NEW-RECORD.
```

Example 5. GO TO example

If we were to compile and run the program, you would see that although the job ABENDS (abnormally ends) with a 4038-abend code, it did execute some of the code and wrote the first two lines of the output. If you were to look at the output in more detail, you would see a message like the following:

```
IGZ0037S The flow of control in program TOTEN1 proceeded beyond the last  
line of the program.
```

Example 6. Abend from GO TO example

So, what went so terribly wrong when we used the GO TO command? To answer this, we need to understand the key difference between GO TO and PERFORM. On the first line we used the PERFORM keyword, that transferred control to the WRITE-NEW-RECORD paragraph. Once the execution reached the end of that paragraph, execution returned to the line following the PERFORM statement. The next line used the GOTO keyword to again transfer control to the WRITE-NEW-RECORD paragraph, which prints the second line of output. However, when that paragraph completed, execution continued to the next line following the WRITE-NEW-RECORD paragraph. Since there are no lines of code following that paragraph the processor tried to execute code beyond the program, z/OS caught this as a problem and abended the program.

As we can see, the use of GO TO causes a branch of execution that doesn't return to the line of code that issued it. Let's demonstrate how messy this code can get:

```
0 01 FLAG          PIC 9(1) VALUE 1.  
  
1 OPEN OUTPUT PRINT-LINE.  
2 GO TO SAY-HELLO-WORLD DEPENDING ON FLAG.  
3  
4 PRINT-NEW-MESSAGE.  
5 MOVE 2 TO FLAG  
6 GO TO SAY-HELLO-COBOL DEPENDING ON FLAG  
7 GO TO END-RUN.  
8  
9 SAY-HELLO-WORLD.  
10 MOVE "Hello World" TO MSG-TO-WRITE  
11 WRITE PRINT-REC  
12 GO TO PRINT-NEW-MESSAGE.  
13  
14 SAY-HELLO-COBOL.  
15 MOVE "Hello COBOL" TO MSG-TO-WRITE  
16 WRITE PRINT-REC  
17 GO TO END-RUN.  
18  
19 END-RUN.  
20 CLOSE PRINT-LINE  
21 STOP RUN.
```

Example 7. Messy code using GO TO

This example is using a mix of conditional and non-conditional GO TO statements, and there are included line numbers to make following the code easier. Line 2 executes and will branch to SAY-HELLO-WORLD on line 9, if the flag variable is set to 1. In this case, it is, so we progress through lines 9-12 and branch to lines 4-6 where the value of the flag is updated and tested again to see if we should jump to SAY-HELLO-COBOL. Since the value of flag is no longer 1, execution just continues to line 7 before jumping to line 19 and finishing the run. Take this program and comment out line 5 and run the program again. Track the execution of the program. Messy right?

Note: Both the TO and ON parts of the conditional GO TO statement can be omitted, giving a statement that looks like GO SAY-HELLO-WORLD DEPENDING FLAG. Which although is less verbose, is no less easy to understand.

So why teach you something that we have said is messy and not advised? Well, by giving you some understanding of its behavior, you will be better equipped when looking through existing code and maintaining

it.

11.3 Paragraphs as blocks of code

So far in this section we have used a few examples of paragraphs without really explaining what they are, how they work and what they can be used for. This section addresses that.

The most analogous way to think about a paragraph in COBOL is to think of a function or method in another language that accepts no parameters, returns no response and alters global variables. It is basically a block of code that performs a sequence of actions that could be used multiple times within the same program.

A paragraph is defined within the procedure division and starts at column eight and can have any name that the user likes, apart from a COBOL keyword, and the declaration of the paragraph is completed with a period (.). A paragraph can contain one to many COBOL sentences and is terminated either by the start of another paragraph or the physical end of the program.

Note: A paragraph can also be ended by END-PROGRAM, END-METHOD, END FACTORY OR END-OBJECT. Most of these are used within Object Orientated COBOL which is not discussed here.

Considering that a program can be made up of multiple paragraphs and that the PERFORM keyword can be used to call the paragraph, either conditionally or as part of a loop, it is easy to see that good paragraph design really helps makes your COBOL more structured and readable.

11.3.1 Designing the content of a paragraph

There are no restrictions as to what content can go inside a paragraph, however, there are two main reasons why you might want to refactor code to be inside a paragraph:

1. To group a sequence of COBOL sentences together that achieve a particular function or task, such as, open all the files that an application is using, calculate a particular function or perform some data validation. Grouping such sentences into a paragraph allows you to give them a name that explains the purpose of the lines of code.
2. The sequence of sentences will be used within a loop. Extracting these lines into a paragraph and then using the PERFORM keyword to create a loop can make for very comprehensible code.

Remember that you can also perform other paragraphs within existing paragraphs. This nested calling of paragraphs can again, help to structure your code.

11.3.2 Order and naming of paragraphs

There is no requirement about the order that paragraphs should appear within a COBOL program. A paragraph can be called from a point either before or after where it is declared. Although there are no restrictions enforced by the language, there are some techniques that you can follow that will make larger programs easier to follow and understand. Some of these techniques and best practices are:

- Name each paragraph to correspond with its function or behavior. A paragraph named OPEN-INPUT-FILES. is a lot more understandable than one named DO-FILE-STUFF.
- Order the paragraphs in the general order in which they will be executed at runtime. Doing this has two main advantages. Using the outline view in a modern IDE will allow you to ‘read’ the name of each paragraph from top to bottom, in doing so you will be able to establish the general structure of the program and its behavior.
- Some COBOL programmers prefix the name of paragraphs with a number that increases throughout the source code as per Example 8.

- Because the paragraphs are numbered and appear in the source code in that order, when a sentence references a paragraph it is easier to know where in the program that paragraph might appear. When initially structuring a program in this way, the numbers used would only increment the highest significant figure, allowing for new paragraphs to be inserted in between if needed. Although the rise of modern editors, which allow outlining and instant jumping to a reference or declaration, makes this technique of less necessity, it is still useful to understand.

```

        PERFORM 1000-OPEN-FILES .
        PERFORM 2000-READ-NEXT-RECORD .
        GO TO 3000-CLOSE-STOP .

1000-OPEN-FILES .
    OPEN INPUT ACCT-REC .
    OPEN OUTPUT PRINT-LINE .

*
2000-READ-NEXT-RECORD .
    PERFORM 4000-READ-RECORD
    PERFORM UNTIL LASTREC = 'Y'
    PERFORM 5000-WRITE-RECORD
    PERFORM 4000-READ-RECORD
    END-PERFORM .

*
3000-CLOSE-STOP .
    CLOSE ACCT-REC .
    CLOSE PRINT-LINE .
    STOP RUN .

*
4000-READ-RECORD .
    READ ACCT-REC
    AT END MOVE 'Y' TO LASTREC
    END-READ .

*
5000-WRITE-RECORD .
    MOVE ACCT-N -      TO ACCT-NO-0 .
    MOVE ACCT-LIMIT    TO ACCT-LIMIT-0 .
    MOVE ACCT-BALANCE  TO ACCT-BALANCE-0 .
    MOVE LAST-NAME    TO LAST-NAME-0 .
    MOVE FIRST-NAME   TO FIRST-NAME-0 .
    MOVE COMMENTS     TO COMMENTS-0 .
    WRITE PRINT-REC .

```

Example 8. Numbered paragraphs

- Lastly, it is common to explicitly end a paragraph by coding an empty paragraph following each paragraph, see Example 9. This empty paragraph does not contain any code, has the same name as the paragraph it is closing, suffixed with -END and is in turn closed by the starting of a following paragraph. But it can be used as a visual delimiter and is useful when using the PERFORM THRU keyword, which is discussed further on in this chapter. Some Java programmers who have learned COBOL have commented that it is equivalent to the closing brace ("}") at the end of a block of code.

```

1000-OPEN-FILES .
    OPEN INPUT ACCT-REC .
    OPEN OUTPUT PRINT-LINE .
1000-OPEN-FILES-END .

*
2000-READ-NEXT-RECORD .
    PERFORM 4000-READ-RECORD

```

```
PERFORM UNTIL LASTREC = 'Y'  
PERFORM 5000-WRITE-RECORD  
PERFORM 4000-READ-RECORD  
END-PERFORM.  
2000-READ-NEXT-RECORD-END .
```

Example 9. Explicitly closed paragraphs

11.4 Program control with paragraphs

So far in this chapter we have discussed the importance of using paragraphs to structure your code. In doing this, we have used the PERFORM keyword a few times to execute the paragraphs we had created. Specifically, we used the keyword by itself and used it with the VARYING keyword to construct a loop. In this section, we will discuss in more detail how the PERFORM keyword can be used.

11.4.1 PERFORM TIMES

Perhaps the simplest way of repeating a perform statement is to use the TIMES keyword to perform a paragraph or sections of code a static number of times, shown in Example 10.

```
PERFORM 10 TIMES  
MOVE FIELD-A TO FIELD-B  
WRITE RECORD  
END-PERFORM .
```

Example 10. TIMES

The required number of times that the code should be executed can either be a literal, as above, or the value of a numeric variable as shown in Example 11. where the PERFORM keyword is being used to execute a paragraph.

```
PERFORM MY-NEW-PARAGRAPH COUNTER TIMES .
```

Example 11. TIMES 2

11.4.2 PERFORM THROUGH

You may require a sequential list of paragraphs to be executed in turn, instead of performing them individually. The THROUGH or THRU keyword can be used to list the start and end paragraphs of the list. Execution will progress through each of the paragraphs as they appear in the source code, from beginning to end, before returning to the line following the initial perform statement, observe Example 12.

```
1000-PARAGRAPH-A .  
      PERFORM 2000-PARAGRAPH-B THRU  
            3000-PARAGRAPH-C .  
*  
2000-PARAGRAPH-B .  
      ...  
*  
3000-PARAGRAPH-C .  
      ...  
*  
4000-PARAGRAPH-D .  
      ...
```

Example 12. PERFORM THRU

Note: The use of the THRU keyword can also be used alongside the TIMES, UNTIL and VARYING keywords, to allow the list of paragraphs to be executed rather than just a single paragraph or blocks of code.

11.4.3 PERFORM UNTIL

Adding the UNTIL keyword to a perform sentence allows you to iterate over a group of sentences until the Boolean condition is met. Effectively allowing you to program while loops in COBOL, take this basic example:

```
MOVE 0 TO COUNTER.  
PERFORM UNTIL COUNTER = 10  
  ADD 1 TO COUNTER GIVING COUNTER  
  MOVE COUNTER TO MSG-TO-WRITE  
  WRITE PRINT-REC  
END-PERFORM.
```

Example 13. PERFORM UNTIL

This would be equivalent to the Java code:

```
while(counter != 10){  
    //counter++  
    //move counter to msg-to-write  
    //write print-rec  
}
```

Example 14. Java while loop

In this case, the Boolean condition is evaluated before the loop is executed. However, if you wish for the loop to be executed at least once before the condition is evaluated, you can alter the sentence to read:

```
PERFORM UNTIL COUNTER = 10 WITH TEST AFTER  
  ADD 1 TO COUNTER GIVING COUNTER  
  MOVE COUNTER TO MSG-TO-WRITE  
  WRITE PRINT-REC  
END-PERFORM.
```

Example 15. PERFORM UNTIL WITH TEST AFTER

This would be similar to a “do while” loop in Java:

```
do{  
    //counter++  
    //move counter to msg-to-write  
    //write print-rec  
}  
While(counter != 10);
```

Example 16. Java while loop

11.4.4 PERFORM VARYING

We've already used the VARYING keyword earlier in the section titled Using performs to code a loop, recall:

```
PERFORM VARYING COUNTER FROM 01 BY 1 UNTIL COUNTER EQUAL 11
...
END-PERFORM.
```

Example 17. Basic loop

In this example, the variable counter is tested to see if it equals 11, as long as it doesn't then it is incremented, and the sentences nested within the perform statement are executed. This construct can be extended, exemplified in Example 18.

```
PERFORM VARYING COUNTER FROM 01 BY 1 UNTIL COUNTER EQUAL 11
    AFTER COUNTER-2 FROM 01 BY 1 UNTIL COUNTER-2 EQUAL 5
...
END-PERFORM.
```

Example 18. Extended loop

This may seem complex, but compare it to this Java pseudo-code:

```
for(int counter = 0; counter < 11; counter++){
    for(int counter2 = 0; counter2 < 5; counter2++){
        //move counter to msg-to-write
        //write print-rec
    }
}
```

Example 19. Java extended loop

This is really, just two for loops nested within each other. This construct is very useful when iterating over tables or nested record structures. As for each loop of the outer varying loop, the inner loop will be executed five times. As mentioned previously, the test of the condition will be assumed by COBOL to be at the beginning of the loop, however, it can be specified to be evaluated at the end of the loop by adding the phrase WITH TEST AFTER to the initial perform sentence.

11.5 Using subprograms

So far, we have only examined the internal structure of a single COBOL program. As programs increase in function and number, it is common that a programmer might want certain aspects of a programs function to be made available to other programs within the system. Abstracting generic functions into their own program and allowing them to be called from other programs can reduce the amount of code duplication within a system and therefore decrease the cost of maintenance, as fixes to shared modules only need to be made once.

Note: Although here we will describe the COBOL native way of calling another program, note that some middleware products will provide APIs that might do this in an enhanced way.

When calling another program, we need to consider three main concerns: how we will reference the program we wish to call, the parameters we want to send to the target program and the parameter that we wish the target program to return.

11.5.1 Specifying the target program

To call a target program we will use the keyword CALL followed by a reference to the target program we wish to call. The two main ways to do this are by a literal value or by referencing a variable, shown in Example 20.

```
CALL 'PROGA1' ...
...
MOVE 'PROGA2' TO PROGRAM-NAME.
```

```
CALL PROGRAM-NAME . . .
```

Example 20. Basic CALL

It is also possible to reference the target platform by passing a pointer reference to the target program. If you thought that passing a pointer reference to a function was only something that ultra-modern languages had, nope COBOL got there first!

11.5.2 Specifying program variables

Now that we have identified the name of the program we wish to call; we must identify the variables that the calling program might want to send. These are individually specified by the USING keyword. COBOL provides support to both pass by reference and pass by copy, as well as a pass by value concept. Each of the supported passing techniques can be applied to all the data-items being passed or used selectively against different items.

By default, COBOL will pass data items by reference. This means that both the calling and target program will be able to read and write to the same area of memory that is represented by the variable. This means that if the target program updates the content of the variable, that change will be visible to the calling program once execution has returned.

The BY CONTENT phrase allows a copy of the passed variable to be passed to the target program. Although the target program can update the variable, those updates will not be visible to the calling program.

Note: When passing variables either BY REFERENCE or BY CONTENT, note you can send data items of any level. Which means you can pass entire data structures, handy for dealing with common records.

You might also see the phrase, BY VALUE, being used in a CALL sentence. BY VALUE is similar to BY CONTENT, as a copy of the content of the variable is passed. The difference is that only a subset of COBOL datatypes are supported and you can only specify elementary data-items. This is because BY VALUE is primarily used when COBOL is calling a program of another language (such as C).

11.5.3 Specifying the return value

Finally, the RETURNING phrase is used to specify the variable that should be used to store the return value. This can be any elementary data-item declared within the data-division. Note that this is optional. Some programs might not return anything, or you might have passed values BY REFERENCE to the target program in which case updates to those variables will be visible once the target program returns.

11.6 Summary

In summary, this chapter should provide the necessary foundation to understand structured programming and how it relates to COBOL and its importance to understanding and maintaining code. Many examples of how, when and why to implement key techniques have been provided and explained for further understanding. You should be able to identify the basic differences between structured programming (COBOL) and OO programming (Java). You should also understand the general concept of the best practices in the structure of the Procedure Division with reference to the design and content of paragraphs, program control options and ways to call other programs within the same system.

12 File output

Designing a structured layout that is easy to read and understand is required to format output. Designing a structured layout involves column headings and variable alignment using spaces, numeric format, currency format, etc. This chapter aims to explain this concept utilizing example COBOL code to design column headings and align data names under the such headings. At the end of the chapter you are asked to complete a lab that practices implementation of the components covered.

A capability of COBOL data output formatting that is worth noting but not covered in this chapter is that COBOL is a web enabled computer language. COBOL includes easy and quick transformation of existing COBOL code to write JSON (JavaScript Object Notation) where the output is sub-sequentially formatted for a browser, a smartphone, etc. Frequently, the critical data accessed by a smart phone, such as a bank balance, is stored and controlled by z/OS where a COBOL program is responsible for retrieving and returning the bank balance to the smart phone.

- **Review of COBOL write output process**
 - ENVIRONMENT DIVISION
 - **FILE DESCRIPTOR**
 - FILLER
 - **Report and column headers**
 - HEADER-2
 - **PROCEDURE DIVISION**
 - MOVE sentence
 - PRINT-REC FROM sentences
 - Lab

12.1 Review of COBOL write output process

This section briefly reviews certain aspects of the ENVIRONMENT DIVISION for the purpose of understanding how it ties together with the content of this chapter.

12.1.1 ENVIRONMENT DIVISION

The “File handling” section covered the SELECT and respective ASSIGN programmer chosen names, whereas this chapter focuses on output. Figure 1. shows a coding example using PRINT-LINE as the programmer chosen COBOL internal file name for output.

Figure 1. SELECT and ASSIGN

12.2 FILE DESCRIPTOR

The File Description (FD), previously described under the FILE-CONTROL paragraph section, entry represents the highest level of organization in the FILE SECTION. The FD entry describes the layout of the file defined by a previous FILE-CONTROL SELECT statement. Therefore, the FD entry connects the SELECT file name with a defined layout of the file name. An example file descriptor, FD, for PRINT-LINE is shown in Figure 2. What follows the file descriptor is a defined layout of PRINT-LINE.

```

*----- ENVIRONMENT DIVISION.
*----- INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT PRINT-LINE ASSIGN TO PRTLINE.
    SELECT ACCT-REC ASSIGN TO ACCTREC.
*SELECT clause creates an internal file name
*ASSIGN clause creates a name for an external data source,
*which is associated with the JCL DDNAME used by the z/OS
*e.g. ACCTREC is linked in JCL file CBL0001J to &SYSUID..DATA
*where &SYSUID. stands for Your z/OS user id
*e.g. if Your user id is 254321,
*the data set used for ACCTREC is 254321.DATA

```

Figure 118:

12.2.1 FILLER

Observe the data name FILLER. While most data fields have unique names, FILLER is a COBOL reserved word data name, that is useful for output formatting. This is in part because FILLER allocates memory space without the need for a name. Also, FILLER allocated memory has a defined length in the output line and may contain spaces or any literal. Figure 2. shows multiple VALUE SPACES for FILLER. SPACES create white space between data-items in the output which is valuable in keeping the code readable. More specifically in Figure 2. FILLER PIC X(02) VALUE SPACES, represents the output line containing two spaces.

Figure 2. FILLER

12.3 Report and column headers

Writing report or column headers requires a structured output layout designed by the programmer. Figure 3. illustrates such a structure. The designed output structure layout is implemented within the DATA DIVISION and includes the headers listed and defined below.

- **HEADER-1:**

- Writes a literal
- Example: ‘Financial Report for’

- **HEADER-2:**

- Writes literals
- Examples:
 - ‘Year’ followed by a variable name
 - ‘Month’ followed by a variable name
 - ‘Day’ followed by a variable name

- **HEADER-3:**

- Writes literals
- Examples:

```

*-----  

  DATA DIVISION.  

*-----  

  FILE SECTION.  

    FD PRINT-LINE RECORDING MODE F.  

*FD -- describes the layout of PRINT-LINE file,  

*including level numbers, variable names, data types and lengths  

*  

  01 PRINT-REC.  

    05 ACCT-NO-0      PIC X(8).  

    05 FILLER        PIC X(02) VALUE SPACES.  

*   FILLER -- COBOL reserved word used as data name to remove  

*   the need of variable names only for inserting spaces  

*  

    05 LAST-NAME-0   PIC X(20).  

    05 FILLER        PIC X(02) VALUE SPACES.  

*   SPACES -- used for structured spacing data outputs rather  

*   than using a higher PIC Clause length as in CBL0001.cobol,  

*   which makes a good design practice and a legible output  

*  

  01 WS-CURRENT-DATE-DATA.  

    05 ACCT-LIMIT-0  PIC $$,$$$,$$9.99.  

*   The repeated $ characters revert to spaces and then one $  

*   in front of the printed amount.  

*  

    05 FILLER        PIC X(02) VALUE SPACES.  

    05 ACCT-BALANCE-0 PIC $$,$$$,$$9.99.  

    05 FILLER        PIC X(02) VALUE SPACES.

```

Figure 119:

- ‘Account’ followed by FILLER spacing
 - ‘Last Name’ followed by FILLER spacing
 - ‘Limit’ followed by FILLER spacing
 - ‘Balance; followed by FILLER spacing
- **HEADER-4:**
 - Writes dashes followed by FILLER spacing

WORKING-STORAGE SECTION.

```

01 HEADER-1.
  05 FILLER      PIC X(20) VALUE 'Financial Report for'.
  05 FILLER      PIC X(60) VALUE SPACES.

01 HEADER-2.
  05 FILLER      PIC X(05) VALUE 'Year '.
  05 HDR-YR      PIC 9(04).
  05 FILLER      PIC X(02) VALUE SPACES.
  05 FILLER      PIC X(06) VALUE 'Month '.
  05 HDR-MO      PIC X(02).
  05 FILLER      PIC X(02) VALUE SPACES.
  05 FILLER      PIC X(04) VALUE 'Day '.
  05 HDR-DAY     PIC X(02).
  05 FILLER      PIC X(56) VALUE SPACES.

01 HEADER-3.
  05 FILLER      PIC X(08) VALUE 'Account '.
  05 FILLER      PIC X(02) VALUE SPACES.
  05 FILLER      PIC X(10) VALUE 'Last Name '.
  05 FILLER      PIC X(15) VALUE SPACES.
  05 FILLER      PIC X(06) VALUE 'Limit '.
  05 FILLER      PIC X(06) VALUE SPACES.
  05 FILLER      PIC X(08) VALUE 'Balance '.
  05 FILLER      PIC X(40) VALUE SPACES.

01 HEADER-4.
  05 FILLER      PIC X(08) VALUE '-----'.
  05 FILLER      PIC X(02) VALUE SPACES.
  05 FILLER      PIC X(10) VALUE '-----'.
  05 FILLER      PIC X(15) VALUE SPACES.
  05 FILLER      PIC X(10) VALUE '-----'.
  05 FILLER      PIC X(02) VALUE SPACES.
  05 FILLER      PIC X(13) VALUE '-----'.
  05 FILLER      PIC X(40) VALUE SPACES.

*
*HEADER -- structures for report or column headers,
*that need to be setup in WORKING-STORAGE so they can be used
*in the PROCEDURE DIVISION
*
```

Figure 120:

Figure 3. Designed output structure layout

12.3.1 HEADER-2

HEADER-2 includes the year, month, day of the report together with FILLER area, creating blank spaces between the year, month, and day, as you can see in Figure 3. Figure 4. is an example of the data name layout used to store the values of CURRENT-DATE. The information COBOL provides in CURRENT-DATE is used to populate the output file in HEADER-2.

```
01 WS-CURRENT-DATE-DATA.  
  05 WS-CURRENT-DATE.  
    10 WS-CURRENT-YEAR      PIC 9(04).  
    10 WS-CURRENT-MONTH    PIC 9(02).  
    10 WS-CURRENT-DAY     PIC 9(02).  
  05 WS-CURRENT-TIME.  
    10 WS-CURRENT-HOURS   PIC 9(02).  
    10 WS-CURRENT-MINUTE  PIC 9(02).  
    10 WS-CURRENT-SECOND  PIC 9(02).  
    10 WS-CURRENT-MILLISECONDS PIC 9(02).  
*   This data layout is organized according to the ouput  
*   format of the FUNCTION CURRENT-DATE.  
*
```

Figure 121:

Figure 4. CURRENT-DATE intrinsic function

12.4 PROCEDURE DIVISION

Figures 1 through 4 are a designed data layout that includes a data line and report headers. Using the storage mapped by the data line and report headers, COBOL processing logic can write the headers followed by each data line. Figure 5. is an example of an execution logic resulting used to write the header layout structure in a COBOL program.

```
WRITE-HEADERS.  
  MOVE FUNCTION CURRENT-DATE TO WS-CURRENT-DATE-DATA.  
*   The CURRENT-DATE function returns an alphanumeric value  
*   that represents the calendar date and time of day  
*   provided by the system on which the function is  
*   evaluated.  
  MOVE WS-CURRENT-YEAR  TO HDR-YR.  
  MOVE WS-CURRENT-MONTH TO HDR-MO.  
  MOVE WS-CURRENT-DAY   TO HDR-DAY.  
  WRITE PRINT-REC FROM HEADER-1.  
  WRITE PRINT-REC FROM HEADER-2.  
  MOVE SPACES TO PRINT-REC.  
  WRITE PRINT-REC AFTER ADVANCING 1 LINES.  
  WRITE PRINT-REC FROM HEADER-3.  
  WRITE PRINT-REC FROM HEADER-4.  
  MOVE SPACES TO PRINT-REC.
```

Figure 122:

Figure 5. Execution logic to write header layout structure

12.4.1 MOVE sentences

The COBOL MOVE sentence, on line 1, in the WRITE-HEADERS paragraph is collecting the current date information from the system and storing that information in a defined data name layout, WS-CURRENT-DATE-DATA. The use of the reserved word FUNCTION means whatever follows is a COBOL intrinsic function. The sentences on lines 2, 3, and 4 are storing the date information, year, month and day, in HEADER-2 defined data name areas, HDR-YR, HDR-MO and HDR-DAY. The sentence on line 11, the final sentence in the paragraph, writes spaces into the PRINT-REC area to clear out the line storage in preparation for writing the data lines.

12.4.2 PRINT-REC FROM sentences

PRINT-REC is opened for output resulting in PRINT-REC FROM following through with a write PRINT-REC FROM a different header or defined data name layout. The sentences on lines 5 and 6 write the PRINT-REC FROM defined header data names, HEADER-1 and HEADER-2, from Figure 3. The PRINT-REC file descriptor data names in Figure 2. are effectively replaced with the content of the header data names in Figure 3. written to output. The sentences on lines 7 and 8 result in a blank line written between headers. The sentences on lines 9 and 10 write the PRINT-REC FROM defined HEADER-3 and HEADER-4 data names from Figure 3. The PRINT-REC file descriptor data names in Figure 2. are effectively replaced with the content of the header data names in Figure 3.

12.5 Lab

This lab utilizes two COBOL programs, CBL0004 and CBL0005, located within your id.CBL data set, as well as two JCL jobs, CBL0004J and CBL0005J, located within your id.JCL data set. The JCL jobs are used to compile and execute the COBOL programs, as discussed in previous chapters.

12.5.0.1 Using VSCode and Zowe Explorer

1. Submit job: CBL0004J
2. Observe the report written with headers like Figure 6. below.

CBL0004J.JOB02732.PRTLINE X					
1	Financial Report for				
2	Year 2020	Month 03	Day 20		
3					
4	Account	Last Name		Limit	Balance
5	-----	-----		-----	-----
6	17891797	WASHINGTON		\$10,000.00	\$188.74

Figure 123:

Figure 6. Report with headers

3. Submit job: CBL0005J
4. Observe the report data lines are written without dollar currency symbol, illustrated in Figure 7.

Figure 7. No currency symbol in output

5. Modify id.CBL(CBL0005) to include the dollar currency symbol in the report.

Hint: Compare with CBL0004 line 25

Limit	Balance
-----	-----
10,000.00	188.74

Figure 124:

6. Re-submit job: CBL0005J
7. Observe the report data lines should now include the dollar currency symbol.

```
≡ CBL0005J.JOB02734.PRTLINE X
1 | Financial Report for
2 | Year 2020 Month 03 Day 20
3 |
4 | Account Last Name           Limit   Balance
5 | ----- ----- ----- -----
6 | 17891797 WASHINGTON        $10,000.00    $188.74
```

Figure 125:

Figure 8. Currency symbol added to output

13 Conditional expressions

This chapter dives into how programs make decisions based upon the programmer written logic. Specifically, programs make these decisions within the PROCEDURE DIVISION of the source code. We will expand on several topics regarding conditional expressions written in COBOL through useful explanations, examples and eventually practicing implementation through a lab.

- Boolean logic, operators, operands, and identifiers
 - COBOL conditional expressions and operators
 - Examples of conditional expressions using Boolean operators
- Conditional expression reserved words and terminology
 - IF, EVALUATE, PERFORM and SEARCH
 - Conditional states
 - Conditional names
- Conditional operators
- Conditional expressions
 - IF ELSE (THEN) statements
 - EVALUATE statements
 - PERFORM statements
 - SEARCH statements
- Conditions
 - Relation conditions
 - Class conditions
 - Sign conditions
- Lab

13.1 Boolean logic, operators, operands, and identifiers

Programs make decisions based upon the programmer written logic. Program decisions are made using Boolean logic where a conditional expression is either true or false, yes or no. A simple example would be a variable named ‘LANGUAGE’. Many programming languages exist; therefore, the value of variable LANGUAGE could be Java, COBOL, etc... Assume the value of LANGUAGE is COBOL. Boolean logic is, IF LANGUAGE = COBOL, THEN DISPLAY COBOL, ELSE DISPLAY NOT COBOL. IF triggers the Boolean logic to determine the condition of true/false, yes/no, applied to LANGUAGE = COBOL which is the conditional expression. The result of IF condition executes what follows THEN when the condition is true and executes what follows ELSE when the condition is false.

The Boolean IF verb operates on two operands or identifiers. In the example above, LANGUAGE is an operand and COBOL is an operand. A Boolean relational operator compares the values of each operand.

13.1.1 COBOL conditional expressions and operators

Three of the most common type of COBOL conditional expressions are:

1. General relation condition

2. Class condition

3. Sign condition

A list of COBOL Boolean relational operators for each of the common type of COBOL conditional expressions are represented in Figures 1, 2 and 3 below.

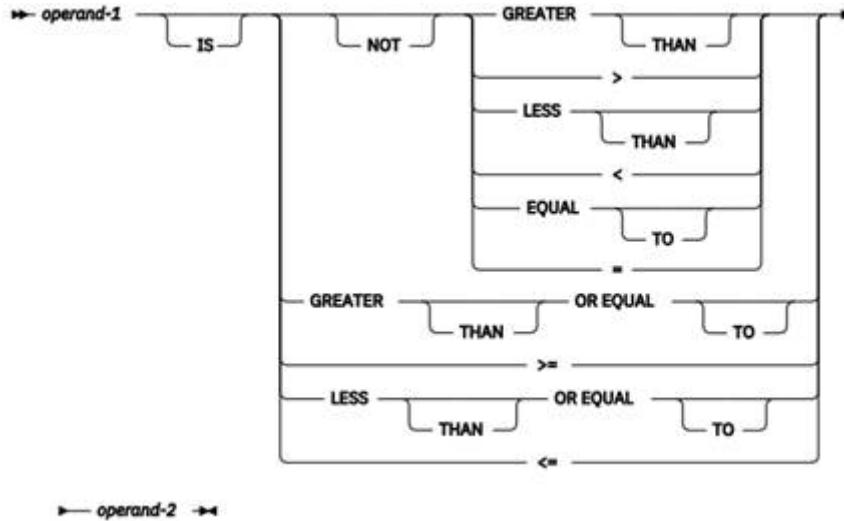


Figure 126:

Figure 1. General relation condition operators

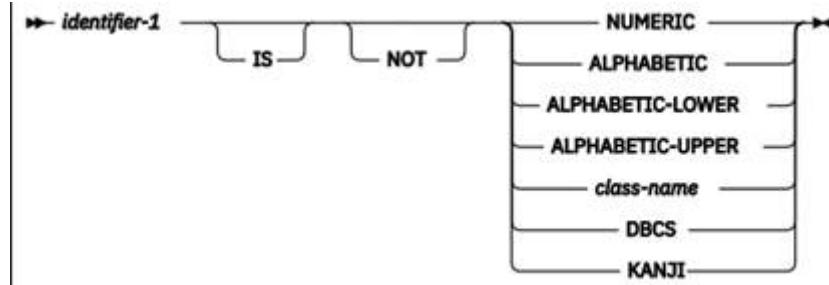


Figure 127:

Figure 2. Class condition operators

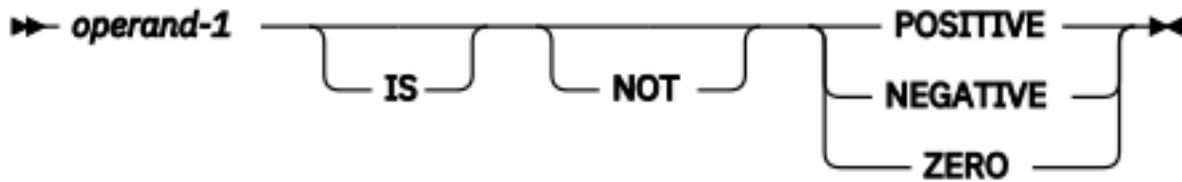


Figure 128:

Figure 3. Sign condition operators

13.1.2 Examples of conditional expressions using Boolean operators

A simple conditional expression can be written as:

```
IF 5 > 1 THEN DISPLAY '5 is greater than 1' ELSE DISPLAY '1 is greater than 5'.
```

Compounded conditional expressions are enclosed in parenthesis and their Boolean operators are:

AND

OR

The code snippet below demonstrates a compounded conditional expression using the AND Boolean operator.

```
IF (5 > 1 AND 1 > 2)THEN .... ELSE ....
```

This conditional expression evaluates to false because while $5 > 1$ is true, $1 > 2$ is false. The AND operation requires both expressions to be true to return true for the compounded condition expression. Let's show another implementation, this time using the OR Boolean operator.

```
IF (5 > 1 OR 1 > 2)THEN .... ELSE ....
```

This conditional expression evaluates to true because while $1 > 2$ is false, $5 > 1$ is true. The OR operation requires only one of the expressions to be true to return true for the entire compounded condition expression. More conditional operators used for relation, class, and sign conditions are discussed further on in the chapter.

13.2 Conditional expression reserved words and terminology

Thus far in this book, we have touched upon the necessity and use of COBOL reserved words. This section aims to expand on the topic of reserved words, specifically ones that are used when processing conditional expressions.

13.2.1 IF, EVALUATE, PERFORM and SEARCH

These are COBOL reserved words available for the processing of conditional expressions, where a condition is a state that can be set or changed.

13.2.2 Conditional states

TRUE and FALSE are among the most common conditional states.

13.2.3 Conditional names

A conditional-name is a programmer defined variable name with the TRUE condition state. Conditional names are declared in the WORKING STORAGE SECTION with an 88-level number. The purpose of 88-level is to improve readability by simplifying IF and PERFORM UNTIL statements.

The 88-level conditional data-name is assigned a value at compile time. The program cannot change the 88-level data-name during program execution. However, the program can change the data name value in the level number above the 88-level conditional data-name. 01-level USA-STATE in Example 1. can be changed. A program expression referencing the 88-level data-name is only true when the current value of the preceding level data name, USA-STATE, is equal to the WORKING-STORAGE 88-level conditional data-name assigned value.

Observe in Example 1. ‘The State is not Texas’ is written as a result of the first IF STATE because the value of USA-STATE is AZ which is not equal to the 88-level conditional data-name, TX. The second IF STATE writes, ‘The State is Texas’ because the value of USA-STATE is equal to the assigned 88-level value of TX.

```

WORKING-STORAGE.
01 USA-STATE      PIC X(2) VALUE SPACES.
    88 STATE        VALUE 'TX'.
    ...
    ...
PROCEDURE DIVISION.
    ...
    ...
MOVE 'AZ' TO USA-STATE.
    ...
    ...
IF STATE DISPLAY 'The State is Texas'
    ELSE DISPLAY 'The State is not Texas'
END-IF.
    ...
    ...
MOVE 'TX' TO USA-STATE.
    ...
    ...
IF STATE DISPLAY 'The State is Texas'
    ELSE DISPLAY 'The State is not Texas'
END-IF.

```

Example 1. Using 88-level conditional name

Numerous 88-level conditional data-names can follow an 01-level data-name. As a result an IF reference to 01-level data-name expression can have numerous values that would return true.

Other level number data-names require the condition expression to include a Boolean operator as shown in Example 2. , where a value can be stored in the 05-level STATE data name to be compared with some other stored value. Therefore, a little bit of extra coding is needed.

```

WORKING-STORAGE.
01 USA-STATE.
    05 STATE      PIC X(2) VALUE SPACES.
    ...
    ...
PROCEDURE DIVISION.
    ...
    ...
MOVE 'AZ' TO STATE.
    ...
    ...
IF STATE = 'TX' DISPLAY 'The State is Texas'
    ELSE DISPLAY 'The State is not Texas'
END-IF.
    ...
    ...
MOVE 'TX' TO STATE.
    ...
    ...
IF STATE = 'TX' DISPLAY 'The State is Texas'
    ELSE DISPLAY 'The State is not Texas'
END-IF.

```

Example 2. Without 88-level conditional name

13.3 Conditional operators

Relational operators compare numeric, character string, or logical data. The result of the comparison, either true (1) or false (0), can be used to make a decision regarding program flow. Table 1 displays a list of relational operators, how they can be written and their meaning.

Relational operator	Can be written	Meaning
IS GREATER THAN	IS >	Greater than
IS NOT GREATER THAN	IS NOT >	Not greater than
IS LESS THAN	IS <	Less than
IS NOT LESS THAN	IS NOT <	Not less than
IS EQUAL TO	IS =	Equal to
IS NOT EQUAL TO	IS NOT =	Not equal to
IS GREATER THAN OR EQUAL TO	IS >=	Is greater than or equal to
IS LESS THAN OR EQUAL TO	IS <=	Is less than or equal to

Table 1. Relational operator

13.4 Conditional expressions

A conditional expression causes the object program to select alternative paths of control, depending on the truth value of a test. Conditional expressions are specified in EVALUATE, IF, PERFORM, and SEARCH statements.

13.4.1 IF ELSE (THEN) statements

IF statements are used to implement or evaluate relational operations. IF ELSE is used to code a choice between two processing actions and inclusion of the word THEN is optional. When an IF statement is present, the statements following the IF statement are processed based on the truth of the conditional expression. Statements are processed until an END-IF or an ELSE statement is encountered. The ELSE statement can appear on any line before the END-IF. IF statements, regardless of the number of lines, are explicitly terminated using END-IF.

Consider this, during program processing something occurs to change the value in the data-name, FACIAL-EXP. Subsequent statements, the conditional expression, needs to check the value of the data-name to decide on how to proceed in the program. Exemplified in Example 3. by the THEN DISPLAY and ELSE DISPLAY statements.

```
IF FACIAL-EXP = 'HAPPY' THEN
    DISPLAY 'I am glad you are happy'
ELSE DISPLAY 'What can I do to make you happy'
END-IF .
```

Example 3. IF, THEN, ELSE, END-IF statement

13.4.2 EVALUATE statements

EVALUATE statements are used to code a choice among three or more possible actions. The explicit terminator for an EVALUATE statement is END-EVALUATE. The EVALUATE statement is an expanded form of the IF statement that allows you to avoid nesting IF statements, a common source of logic errors and debugging issues. EVALUATE operates on both text string values and numerical variables. Using the

FACIAL-EXP conditional-name, observe the COBOL code implementing an EVALUATE statement, shown in Example 4.

```
EVALUATE FACIAL-EXP
  WHEN 'HAPPY'
    DISPLAY 'I am glad you are happy'
  WHEN 'SAD'
    DISPLAY 'What can I do to make you happy'
  WHEN 'PERPLEXED'
    DISPLAY 'Can you tell me what you are confused about'
  WHEN 'EMOTIONLESS'
    DISPLAY 'Do you approve or disapprove'
END-EVALUATE
```

Example 4. EVALUATE statement

13.4.3 PERFORM statements

A PERFORM with UNTIL phrase is a conditional expression. In the UNTIL phrase format, the procedures referred to are performed until the condition specified by the UNTIL phrase evaluates to true. Using the FACIAL-EXP conditional-name, the SAY-SOMETHING-DIFFERENT paragraph is executed continuously UNTIL FACIAL-EXP contains 'HAPPY', observe Example 5.

```
PERFORM SAY-SOMETHING-DIFFERENT BY FACIAL-EXP UNTIL 'HAPPY'
END-PERFORM .
```

Example 5. PERFORM statement

13.4.4 SEARCH statements

The SEARCH statement searches a table for an element that satisfies the specified condition and adjusts the associated index to indicate that element. Tables, effectively an array of values, are created with an OCCURS clause applied to WORK-STORAGE data-names. A WHEN clause is utilized in SEARCH statements to verify if the element searched for satisfies the specified condition. Assuming FACIAL-EXP has many possible values, then SEARCH WHEN is an alternative conditional expression, observe Example 6.

```
SEARCH FACIAL-EXP
WHEN 'HAPPY' STOP RUN
END-SEARCH
```

Example 6. SEARCH WHEN statement

13.5 Conditions

A conditional expression can be specified in either simple conditions or complex conditions. Both simple and complex conditions can be enclosed within any number of paired parentheses; the parentheses, however, do not change whether the condition is simple or complex. This section will cover three of the five simple conditions:

- Relation
- Class
- Sign

13.5.1 Relation conditions

A relation condition specifies the comparison of two operands. The relational operator that joins the two operands specifies the type of comparison. The relation condition is true if the specified relation exists between the two operands; the relation condition is false if the specified relation does not exist. Provided, is a list of a few defined comparisons:

- Numeric comparisons - Two operands of class numeric
- Alphanumeric comparisons - Two operands of class alphanumeric
- DBCS (Double Byte Character Set) comparisons - Two operands of class DBCS
- National comparisons - Two operands of class national

13.5.2 Class conditions

The class condition determines whether the content of a data item is alphabetic, alphabetic-lower, alphabetic-upper, numeric, DBCS, KANJI, or contains only the characters in the set of characters specified by the CLASS clause, as defined in the SPECIAL-NAMES paragraph of the environment division. Provided below is a list of a few valid forms on the class condition for different types of data items.

- Numeric
 - IS NUMERIC or IS NOT NUMERIC
- Alphabetic
 - IS ALPHABETIC or IS NOT ALPHABETIC
 - IS ALPHABETIC-LOWER / ALPHABETIC-UPPER
 - IS NOT ALPHABETIC-LOWER / ALPHABETIC-UPPER
- DBCS
 - IS DBCS or IS NOT DBCS
 - IS KANJI or IS NOT KANJI

13.5.3 Sign conditions

The sign condition determines whether the algebraic value of a numeric operand is greater than, less than, or equal to zero. An unsigned operand is either POSITIVE or ZERO. When a numeric conditional variable is defined with a sign, the following are available:

- IS POSITIVE
- IS NEGATIVE
- IS ZERO

Note : To read more information about these conditions please visit the link:

[IBM Knowledge Center - Enterprise COBOL for z/OS 4.2.0](#)

13.6 Lab

This lab requires two COBOL programs, CBL0006 and CBL0007 and two respective JCL Jobs, CBL0006J and CBL0007J, to compile and execute the COBOL programs. All of which are provided to you in your VSCode - Zowe Explorer.

13.6.0.1 Using VSCode and Zowe Explorer:

1. Take a moment and look over the source code of the two COBOL programs provided: CBL0006 and CBL0007.
2. Compare CBL0006 with CBL0005 from the previous lab. Do you notice the differences?
 - a. Observe the new CLIENTS-PER-STATE line within the WORKING-STORAGE > PROCEDURE DIVISION.
 - b. Observe the new paragraph IS-STATE-VIRGINIA within that same division.
 - c. This paragraph checks whether the client is from Virginia. If that condition is met (true) then the program should add 1 to the clients from Virginia total.
 - d. Program writes “Virginia Clients = “, in last line of report.
3. Submit CBL0006J
4. View the job output from the JOBS section and verify the steps mentioned above were executed.

CBL0006J.JOB03135.PRTLINE X				
48	20012009	BUSH II	\$100,000.00	\$31,313.20
49	20092017	OBAMA	\$9,950,000.00	\$92,311.00
50	20172020	TRUMP	\$8,100,000.00	\$10.00
51	Virginia Clients = 008			

Figure 129:

Figure 4. Id.JCL(CBL0006J) output

5. Submit CBL0007J
6. Find the compile error, IGYPS2113-E, in the job output.
7. Go ahead and modify id.CBL(CBL0007) to correct the syntax error outlined by the IGYPS2113-E message.*
8. Re-submit CBL0007J
9. Validate that the syntax error was corrected by getting an error free output file.



Figure 130:

Figure 5. Successful compile

Lab Hints

```

IS-STATE-VIRGINIA.
  IF USA-STATE = 'Virginia' THEN
    ADD 1 TO VIRGINIA-CLIENTS
  END-IF.
* Boolean logic -- when the conditional expression
* USA-STATE = 'Virginia' is true, the program
* counts one more client from Virginia
* Note -- the inclusion of the word THEN is optional
* END-IF -- explicitly terminates the IF statement

```

Figure 131:

14 Arithmetic expressions

This chapter aims to introduce the concept of implementing arithmetic expressions in COBOL programs. We will review the basic concept of arithmetic expressions, operators, statements, limitations, statement operands, as well as precedence of operation within the expressions. You will be able to follow along with a comprehensive example exhibiting the usage of arithmetic expressions in a COBOL program that you have seen in previous chapters and labs. Following the chapter is a lab to practice the implementation of what you have learned.

- What is an arithmetic expression?
 - Arithmetic operators
 - Arithmetic statements
- Arithmetic expression precedence rules
 - Parentheses
- Arithmetic expression limitations
- Arithmetic statement operands
 - Size of operands
- Examples of COBOL arithmetic statements
- Lab

14.1 What is an arithmetic expression?

Arithmetic expressions are used as operands of certain conditional and arithmetic statements. An arithmetic expression can consist of any of the following items:

- An identifier described as a numeric elementary item (including numeric functions).
- A numeric literal.
- The figurative constant ZERO.
- Identifiers and literals, as defined in items 1, 2, and 3, separated by arithmetic operators.
- Two arithmetic expressions, as defined in items 1, 2, 3, or 4, separated by an arithmetic operator.
- An arithmetic expression, as defined in items 1, 2, 3, 4, or 5, enclosed in parentheses.
- Any arithmetic expression can be preceded by a unary operator.

Identifiers and literals that appear in arithmetic expressions must represent either numeric elementary items or numeric literals on which arithmetic can be performed. If the value of an expression to be raised to a power is zero, the exponent must have a value greater than zero. Otherwise, the size error condition exists. In any case where no real number exists as the result of an evaluation, the size error condition exists.

14.1.1 Arithmetic operators

Five binary arithmetic operators and two unary arithmetic operators can be used in arithmetic expressions. These operators are represented by specific characters that must be preceded and followed by a space. However, no space is required between a left parenthesis and unary operator. These binary and unary arithmetic operators are listed in Table 1.

Binary operator	Meaning	Unary operator	Meaning
+	Addition	+	Multiplication by +1
-	Subtraction	-	Multiplication by -1
	Multiplication		
/	Division		
	Exponentiation		

Table 1. Arithmetic operators

14.1.2 Arithmetic statements

Arithmetic statements are utilized for computations. Individual operations are specified by the ADD, SUBTRACT, MULTIPLY, and DIVIDE statements. These individual operations can be combined symbolically in a formula that uses the COMPUTE statement for ease of programming and performance. The COMPUTE statement assigns the value of an arithmetic expression to one or more data items. With the COMPUTE statement, arithmetic operations can be combined without the restrictions on receiving data items imposed by the rules for the ADD, SUBTRACT, MULTIPLY, and DIVIDE statements. When arithmetic operations are combined, the COMPUTE statement can be more efficient than the separate arithmetic statements written in a series. For these reasons, it is best practice to use the COMPUTE statement for most arithmetic evaluations rather than ADD , SUBTRACT , MULTIPLY , and DIVIDE statements. Often, you can code only one COMPUTE statement instead of several individual arithmetic statements. The COMPUTE statement assigns the result of an arithmetic expression to one or more data items, for example:

```
COMPUTE z      = a + b / c \*\* d - e
COMPUTE x y z = a + b / c \*\* d - e
```

Some arithmetic calculations might be more intuitive using arithmetic statements other than COMPUTE . You might also prefer to use the DIVIDE statement (with its REMAINDER phrase) for division in which you want to process a remainder. The REM intrinsic function also provides the ability to process a remainder.

14.2 Arithmetic expression precedence rules

Order of operation rules have been hammered into your head throughout the years of learning mathematics, remember the classic PEMDAS (parentheses, exponents, multiply, divide, add, subtract)? Arithmetic expressions in COBOL are not exempt from these rules and often use parentheses to specify the order in which elements are to be evaluated.

14.2.1 Parentheses

Parentheses are used to denote modifications to normal order of operations (precedence rules). An arithmetic expression within the parentheses is evaluated first and result is used in the rest of the expression. When expressions are contained within nested parentheses, evaluation proceeds from the least inclusive to the most inclusive set. That means you work from the inner most expression within parentheses to the outer most. The precedence for how to solve an arithmetic expression in Enterprise COBOL with parentheses is:

1. Parentheses (simplify the expression inside them)
2. Unary operator
3. Exponents
4. Multiplication and division (from left to right)
5. Addition and subtraction (from left to right)

Parentheses either eliminate ambiguities in logic where consecutive operations appear at the same hierarchic level or modify the normal hierarchic sequence of execution when necessary. When the order of consecutive operations at the same hierarchic level is not completely specified by parentheses, the order is from left to right.

An arithmetic expression can begin only with a left parenthesis, a unary operator, or an operand (that is, an identifier or a literal). It can end only with a right parenthesis or an operand. An arithmetic expression must contain at least one reference to an identifier or a literal.

There must be a one-to-one correspondence between left and right parentheses in an arithmetic expression, with each left parenthesis placed to the left of its corresponding right parenthesis. If the first operator in an arithmetic expression is a unary operator, it must be immediately preceded by a left parenthesis if that arithmetic expression immediately follows an identifier or another arithmetic expression.

14.3 Arithmetic expression limitations

Exponents in fixed-point exponential expressions cannot contain more than nine digits. The compiler will truncate any exponent with more than nine digits. In the case of truncation, the compiler will issue a diagnostic message if the exponent is a literal or constant; if the exponent is a variable or data-name, a diagnostic message is issued at run time.

Detailed explanation of fixed-point exponential expressions is an advanced topic and beyond the scope of the chapter. However, reference is made to fixed-point exponential expressions for your awareness as you advance your experience level with COBOL programming and arithmetic applied to internal data representations.

14.4 Arithmetic statement operands

The data descriptions of operands in an arithmetic statement need not be the same. Throughout the calculation, the compiler performs any necessary data conversion and decimal point alignment.

14.4.1 Size of operands

If the ARITH(COMPAT) compiler option is in effect, the maximum size of each operand is 18 decimal digits. If the ARITH(EXTEND) compiler option is in effect, the maximum size of each operand is 31 decimal digits.

The composite of operands is a hypothetical data-item resulting from aligning the operands at the decimal point and then superimposing them on one another. How to determine the composite of operands for arithmetic statements is shown in Table 2.

If the ARITH(COMPAT) compiler option is in effect, the composite of operands can be a maximum of 30 digits. If the ARITH(EXTEND) compiler option is in effect, the composite of operands can be a maximum of 31 digits.

Statement	Determination of the com- pos- ite of operands
SUBTRACT	Superimposing all operands in a given state- ment, ex- cept those fol- low- ing the word GIVING.
MULTIPLY	Superimposing all re- ceiv- ing data- items
DIVIDE	Superimposing all re- ceiv- ing data items ex- cept the RE- MAIN- DER data- item
COMPUTE	Restriction does not apply

Table 2. How the composite of operands is determined

In all arithmetic statements, it is important to define data with enough digits and decimal places to ensure the required accuracy in the result. Arithmetic precision details are available in the [IBM Enterprise COBOL Programming Guide Appendix A](#).

Additionally, in the IBM Enterprise COBOL Language Reference, Chapter 20. “PROCEDURE DIVISION Statements”, includes a detailed explanation of DIVIDE and COMPUTE statement capabilities applied to ROUNDING and ON SIZE ERROR handling.

14.5 Examples of COBOL arithmetic statements

In this section, COBOL source code used in previous labs will be modified to demonstrate arithmetic processing. Figure 1. shows level number data-items in the WORKING-STORAGE section. The data-items will be used to total client account limit and client account balance. Observe that the initial value is ZERO.

```
01 TLIMIT-TBALANCE.
  05 TLIMIT          PIC S9(9)V99 COMP-3 VALUE ZERO.
  05 TBALANCE        PIC S9(9)V99 COMP-3 VALUE ZERO.
*
```

Figure 132:

Figure 1. Number level data-items (1)

Shown in Figure 2. is another example of number level data-items in the WORKING-STORAGE section. These data-items are report trailer lines that are used to write a formatted total account limit and total account balance for all clients in the report. Observe the TLIMIT and TBALANCE data-items with large currency number picture clauses.

```
01 TRAILER-1.
  05 FILLER          PIC X(31) VALUE SPACES.
  05 FILLER          PIC X(14) VALUE '-----'.
  05 FILLER          PIC X(01) VALUE SPACES.
  05 FILLER          PIC X(14) VALUE '-----'.
  05 FILLER          PIC X(40) VALUE SPACES.
*
01 TRAILER-2.
  05 FILLER          PIC X(22) VALUE SPACES.
  05 FILLER          PIC X(08) VALUE 'Totals ='.
  05 FILLER          PIC X(01) VALUE SPACES.
  05 TLIMIT-0         PIC $$,$$$,$$9.99.
  05 FILLER          PIC X(01) VALUE SPACES.
  05 TBALANCE-0       PIC $$,$$$,$$9.99.
  05 FILLER          PIC X(40) VALUE SPACES.
*
```

Figure 133:

Figure 2. Number level data-items (2)

In Figure 3. the READ-NEXT-RECORD paragraph, located within the PROCEDURE DIVISION, includes a PERFORM LIMIT-BALANCE-TOTAL statement. The result of this statement is to transfer control to the LIMIT-BALANCE-TOTAL paragraph, located within the PROCEDURE DIVISION, to perform the COMPUTE statements.

```
READ-NEXT-RECORD.
  PERFORM READ-RECORD
  PERFORM UNTIL LASTREC = 'Y'
  PERFORM LIMIT-BALANCE-TOTAL ←
  PERFORM WRITE-RECORD
  PERFORM READ-RECORD
END-PERFORM
```

Figure 134:

Figure 3. READ-NEXT-RECORD.

Figure 4. is an example of two COMPUTE statements in the paragraph, LIMIT-BALANCE-TOTAL. Notice that the results of the COMPUTE statements are to add client ACCT-LIMIT to the current TLIMIT and add client ACCT-BALANCE to TBALANCE totals each time the paragraph is executed, which is one time for each client record read in our example.

```
LIMIT-BALANCE-TOTAL.
  COMPUTE TLIMIT = TLIMIT + ACCT-LIMIT END-COMPUTE
  COMPUTE TBALANCE = TBALANCE + ACCT-BALANCE END-COMPUTE
```

Figure 135:

Figure 4. COMPUTE statements

The WRITE-TLIMIT-TBALANCE paragraph shown in Figure 5. is positioned within the PROCEDURE DIVISION to be executed immediately after all records are read and before the final paragraph that closes the files and terminates program execution.

```
WRITE-TLIMIT-TBALANCE.
  MOVE TLIMIT TO TLIMIT-0.
  MOVE TBALANCE TO TBALANCE-0.
  WRITE PRINT-REC FROM TRAILER-1.
  WRITE PRINT-REC FROM TRAILER-2.
```

Figure 136:

Figure 5. WRITE-TLIMIT-TBALANCE

14.6 Lab

This lab requires two COBOL programs, CBL0008 and CBL0009 and two respective JCL Jobs, CBL0008J and CBL0009J, to compile and execute the COBOL programs. All of which are provided to you in your VSCode - Zowe Explorer.

14.6.0.1 Using VSCode and Zowe Explorer

1. Take a moment and look over the source code of the two COBOL programs provided: CBL0008 and CBL0009.
2. Submit CBL0008J
3. Observe report written with trailers consisting of limit and balance totals at the bottom of the output.

48	20012009	BUSH	11	\$100,000.00 \$51,315.20
49	20092017	OBAMA		\$9,950,000.00 \$92,311.00
50	20172020	TRUMP		\$8,100,000.00 \$10.00
51				Totals = \$47,500,000.00 \$23,004,207.47
52				

Figure 137:

Figure 6. Limit and balance totals

4. Submit CBL0009J
5. Was the job successful? If not, find the compile error message to understand why.
6. Modify id.CBL(CBL0009), correcting the compile error.*

```
==000141==> IGYPS2121-S "TLIMIT" was not defined as a data-name. The statement was discarded.
```

Figure 138:

Figure 7. IGYPS2121-S error message

7. Re-submit CBL0009J
8. Validate that the syntax error was corrected by getting an error free output file like in Figure 8. The correction should report written with trailers consisting of limit and balance totals, like Figure 6.

```
> [CBL0009J(JOB03243) - CC 0012]
> [CBL0009J(JOB03245) - CC 0000] ←
```

Figure 139:

Figure 8. Successful compile

Lab Hints

```
01 TLIMIT-TBALANCE.
  05 TLIMIT          PIC S9(9)V99 COMP-3 VALUE ZERO.
  05 TBALANCE        PIC S9(9)V99 COMP-3 VALUE ZERO.
```

Figure 140:

15 Data types

A COBOL programmer must be aware that the computer stored internal data representation and formatting can differ, where the difference must be defined in the COBOL source code. Understanding the computer's internal data representation requires familiarity with binary, hexadecimal, ASCII, and EBCDIC. Packed-Decimal is needed to explain COBOL Computational and Display data format. This chapter aims to familiarize the reader with these different "types" of data representation.

- **Data representation**
 - Numerical value representation
 - Text representations
- **COBOL DISPLAY vs COMPUTATIONAL**
- Lab

15.1 Data representation

Data such as numerical values and text are internally represented by zeros and ones in most computers, including mainframe computers used by enterprises. While data representation is a somewhat complex topic in computer science, a programmer does not always need to fully understand how various alternative representations work. It is important, however, to understand the differences and how to specify a specific representation when needed.

15.1.1 Numerical value representation

COBOL has five computational (numerical) value representations. The awareness of these representations is important due to two main reasons. The first reason being, when a COBOL program needs to read or write data, it needs to understand how data is represented in the dataset. The second reason is when there are specific requirements regarding the precision and range of values being processed. The following sections describe each computational representation keyword.

15.1.1.1 COMP-1

This is also known as a single-precision floating point number representation. Due to the floating-point nature, a COMP-1 value can be very small and close to zero, or it can be very large (about 10 to the power of 38). However, a COMP-1 value has limited precision. This means that even though a COMP-1 value can be up to 10 to the power of 38, it can only maintain about seven significant decimal digits. Any value that has more than seven significant digits are rounded. This means that a COMP-1 value cannot exactly represent a bank balance like \$1,234,567.89 because this value has nine significant digits. Instead, the amount is rounded. The main application of COMP-1 is for scientific numerical value storage as well as computation.

15.1.1.2 COMP-2

This is also known as a double-precision floating point number representation. COMP-2 extends the range of value that can be represented compared to COMP-1. COMP-2 can represent values up to about 10 to the power of 307. Like COMP-1, COMP-2 values also have a limited precision. Due to the expanded format, COMP-2 has more significant digits, approximately 15 decimal digits. This means that once a value reaches certain quadrillions (with no decimal places), it can no longer be exactly represented in COMP-2.

COMP-2 supersedes COMP-1 for more precise scientific data storage as well as computation. Note that COMP-1 and COMP-2 have limited applications in financial data representation or computation.

15.1.1.3 COMP-3

This is also known as packed BCD (binary coded decimal) representation. This is, by far, the most utilized numerical value representation in COBOL programs. Packed BCD is also somewhat unique and native to mainframe computers such as the IBM z architecture.

Unlike COMP-1 or COMP-2, packed BCD has no inherent precision limitation that is independent to the range of values. This is because COMP-3 is a variable width format that depends on the actual value format. COMP-3 exactly represents values with decimal places. A COMP-3 value can have up to 31 decimal digits.

15.1.1.4 COMP-4

COMP-4 is only capable of representing integers. Compared to COMP-1 and COMP-2, COMP-4 can store and compute with integer values exactly (unless a division is involved). Although COMP-3 can also be used to represent integer values, COMP-4 is more compact.

15.1.1.5 COMP-5

COMP-5 is based on COMP-4, but with the flexibility of specifying the position of a decimal point. COMP-5 has the space efficiency of COMP-4, and the exactness of COMP-3. Unlike COMP-3, however, a COMP-5 value cannot exceed 18 decimal digits.

15.1.2 Text representations

COBOL programs often need to represent text data such as names and addresses.

15.1.2.1 EBCDIC

Extended Binary Coded Decimal Interchange Code (EBCDIC) is an eight binary digits character encoding standard, where the eight digital positions are divided into two pieces. EBCDIC was devised in the early 1960's for IBM computers. EBCDIC is used to encode text data so that text can be printed or displayed correctly on devices that also understand EBCDIC.

15.1.2.2 ASCII

American Standard Code for Information Interchange, ASCII, is another binary digit character encoding standard.

15.1.2.3 EBCDIC vs ASCII

Why are these two standards when they seemingly perform the same function?

EBCDIC is a standard that traces its root to punch cards designed in 1931. ASCII, on the other hand, is a standard that was created, unrelated to IBM punch cards, in 1967. A COBOL program natively understands EBCDIC, and it can comfortably process data originally captured in punch cards as early as 1931.

ASCII is mostly utilized by non-IBM computers.

COBOL can encode and process text data in EBCDIC or ASCII. This means a COBOL program can simultaneously process data captured in a census many decades ago while exporting data to a cloud service utilizing ASCII or Unicode. It is important to point out, however, that the programmer must have the awareness and choose the appropriate encoding.

15.2 COBOL DISPLAY vs COMPUTATIONAL

Enterprise COBOL for z/OS by default utilizes EBCDIC encoding. However, it is possible to read and write ASCII in z/OS. The EBCDIC format representation of numbers and alphabetic characters is in a DISPLAY format. Packed decimal and zoned decimal are NOT in a DISPLAY format. COBOL can describe packed decimal and zoned decimal fields using COMPUTATIONAL, COMP-1, COMP-2, COMP-3, COMP-4, and COMP-5 reserved words.

15.3 Lab

Many of the previous COBOL lab programs you have worked with thus far are reading records containing two packed decimal fields, the client account limit and the client account balance. In the Arithmetic expressions lab, the total of all client account limits and balances used a COMPUTE statement, where the COMP-3 fields contained the packed decimal internal data.

What happens when an internal packed decimal field is not described using COMP-3? Without using COMP-3 to describe the field, the COBOL program treats the data as DISPLAY data (EBCDIC format). This lab demonstrates what happens during program execution without using COMP-3.

15.3.0.1 Using VSCode and Zowe Explorer

1. Submit the job, id.JCL(CBL0010J)
2. Observe that the compile of the COBOL source was successful, however, also observe that the execution of the job failed. How can you tell?

There's no CC code next to CBL0010J(JOB#), instead there is an ABENDU4038 message. U4038 is a common user code error typically involving a mismatch between the external data and the COBOL representation of the data.

3. Read the execution SYSOUT message carefully. The SYSOUT message mistakenly believes the records are 174 characters in length while the program believes the records are 170 characters in length.

Explanation: Packed decimal (COMP-3) expands into two numbers where only one number would typically exist. If the program reads a packed decimal field without describing the field as COMP-3, then program execution becomes confused about the size of the record because the PIC clause, S9(7)V99, is expecting to store seven numbers plus a sign digit when only three word positions are read. Therefore, execution reports a four-record length position discrepancy.

4. Edit id.CBL(CBL0010) to identify and correct the source code problem.*
5. Submit id.JCL(CBL0010J) and verify correction is successful with a CC 0000 code.

Lab Hints:

The ACCT-LIMIT PIC clause in the ACCT-FIELDS paragraph should be the same as the PIC clause for ACCT-BALANCE.

16 Intrinsic functions

Today's COBOL is not your parents COBOL. Today's COBOL includes decades of feature/function rich advancements and performance improvements. Decades of industry specifications are applied to COBOL to address the growing needs of businesses. What Enterprise COBOL for z/OS promised and delivered, is decades of upward compatibility with new releases of hardware and operating system software. The original DNA of COBOL evolved into a powerful, maintainable, trusted, and time-tested computer language with no end in sight.

Among the new COBOL capabilities is JSON GENERATE and JSON PARSE, providing an easy to use coding mechanism to transform DATA DIVISION defined data-items into JSON for a browser, a smart phone, or any IoT (Internet of Things) device to format in addition to transforming JSON received from a browser, a smart phone, or any IoT device into DATA DIVISION defined data-items for processing. Frequently, the critical data accessed by a smart phone, such as a bank balance, is stored and controlled by z/OS where a COBOL program is responsible for retrieving and returning the bank balance to the smart phone. COBOL has become a web enabled computer language.

Previous COBOL industry specifications included intrinsic functions, which remain largely relevant today. An experienced COBOL programmer needs to be familiar with intrinsic functions and stay aware of any new intrinsic functions introduced. This chapter aims to cover the foundation of intrinsic functions and their usage in COBOL.

- **What is an intrinsic function?**
 - Intrinsic function syntax
 - Categories of intrinsic functions
- **Intrinsic functions in Enterprise COBOL for z/OS V6.3**
 - Mathematical example
 - Statistical example
 - Date/time example
 - Financial example
 - Character-handling example
- Use of intrinsic functions with reference modifiers
- Lab

16.1 What is an intrinsic function?

Intrinsic functions are effectively re-usable code with simple syntax implementation and are another powerful COBOL capability. Intrinsic functions enable desired logic processing with a single line of code. They also provide capabilities for manipulating strings and numbers. Because the value of an intrinsic function is derived automatically at the time of reference, you do not need to define these functions in the DATA DIVISION.

16.1.1 Intrinsic function syntax

Written as:

`FUNCTION function-name (argument)`

Where function-name must be one of the intrinsic function names. You can reference a function by specifying its name, along with any required arguments, in a PROCEDURE DIVISION statement. Functions are elementary data items, and return alphanumeric characters, national characters, numeric, or integer values.

```

01  Item-1    Pic x(30)  Value "Hello World!".
01  Item-2    Pic x(30).

.
.
Display Item-1
Display Function Upper-case(Item-1)
Display Function Lower-case(Item-1)
Move Function Upper-case(Item-1) to Item-2
Display Item-2

```

Example 1. COBOL FUNCTION reserved word usage

The code shown in Example 1. above, displays the following messages on the system logical output device:

Hello World! HELLO WORLD! hello world! HELLO WORLD!

16.1.2 Categories of intrinsic functions

The intrinsic functions can be grouped into six categories, based on the type of service performed. They are as follows:

1. Mathematical
2. Statistical
3. Date/time
4. Financial
5. Character-handling
6. General

Intrinsic functions operate against alphanumeric, national, numeric, and integer data-items.

- **Alphanumeric** functions are of class and category alphanumeric. The value returned has an implicit usage of DISPLAY. The number of character positions in the value returned is determined by the function definition.
- **National** functions are of class and category national. The value returned has an implicit usage of NATIONAL and is represented in national characters (UTF-16). The number of character positions in the value returned is determined by the function definition.
- **Numeric** functions are of class and category numeric. The returned value is always considered to have an operational sign and is a numeric intermediate result.
- **Integer** functions are of class and category numeric. The returned value is always considered to have an operational sign and is an integer intermediate result. The number of digit positions in the value returned is determined by the function definition.

16.2 Intrinsic functions in Enterprise COBOL for z/OS V6.3

The current release of Enterprise COBOL for z/OS V6.3, includes 70 intrinsic functions. Each one of these functions falling into one of the aforementioned six categories. While an entire book could be written on intrinsic functions, a single example for each of the six categories are provided in this section.

16.2.1 Mathematical example

Example 2. is storing into X the total of A + B + value resulting from C divided by D. FUNCTION SUM enables the arithmetic operation.

```
Compute x = Function Sum(a b (c / d))
```

Example 2. Mathematical intrinsic function

16.2.2 Statistical example

Example 3. shows three COBOL functions, MEAN, MEDIAN, and RANGE where the arithmetic values are stored in Avg-Tax, Median-Tax, and Tax-Range using the data names with assigned pic clause values.

```
01 Tax-S          Pic 99v999 value .045.
01 Tax-T          Pic 99v999 value .02.
01 Tax-W          Pic 99v999 value .035.
01 Tax-B          Pic 99v999 value .03.
01 Ave-Tax        Pic 99v999.
01 Median-Tax     Pic 99v999.
01 Tax-Range       Pic 99v999.

.
.
.
Compute Ave-Tax      = Function Mean   (Tax-S Tax-T Tax-W Tax-B)
Compute Median-Tax    = Function Median (Tax-S Tax-T Tax-W Tax-B)
Compute Tax-Range     = Function Range  (Tax-S Tax-T Tax-W Tax-B)
```

Example 3. Statistical intrinsic function

16.2.3 Date/time example

Example 4. shows usage of three COBOL functions, Current-Date, Integer-of-Date, and Date-of-Integer applied to MOVE, ADD, and COMPUTE statements.

```
01 YYYYMMDD        Pic 9(8).
01 Integer-Form    Pic S9(9).

.
.
.
Move Function Current-Date(1:8) to YYYYMMDD
Compute Integer-Form = Function Integer-of-Date(YYYYMMDD)
Add 90 to Integer-Form
Compute YYYYMMDD = Function Date-of-Integer(Integer-Form)
Display 'Due Date: ' YYYYMMDD
```

Example 4. Date/time intrinsic function

16.2.4 Financial example

Example 5. shows application of COBOL function ANNUITY financial algorithm where values for loan amount, payments, interest, and number of periods are input to ANNUITY function.

```
01 Loan            Pic 9(9)V99.
01 Payment          Pic 9(9)V99.
01 Interest         Pic 9(9)V99.
01 Number-Periods   Pic 99.

.
.
```

```

Compute Loan = 15000
Compute Interest = .12
Compute Number-Periods = 36
Compute Payment = Loan * Function Annuity((Interest / 12)
    Number-Periods)

```

Example 5. Financial intrinsic function

16.2.5 Character-handling example

Example 6. shows used of COBOL function UPPER-CASE where an string or alphabetic variable processed by UPPER-CASE will translate any lower case characters to upper case.

```

MOVE FUNCTION UPPER-CASE("This is shouting!") TO SOME-FIELD
DISPLAY SOME-FIELD
Output - THIS IS SHOUTING!

```

Example 6. Character-handling intrinsic function

16.3 Use of intrinsic functions with reference modifiers

A reference modification defines a data item by specifying the leftmost character position and an optional length for the data item, where a colon (:) is used to distinguish the leftmost character position from the optional length, as shown in Example 7.

```

05 LNAME      PIC X(20).

LNAME(1:1)
LNAME(4:2)

```

Example 7. Reference modification

Reference modification, LNAME(1:1), would return only the first character of data item LNAME, while reference modification, LNAME(4:2), would return the fourth and fifth characters of LNAME as the result of starting in the fourth character position with a length of two. If LNAME of value SMITH was the data item being referenced in the intrinsic function, the first reference would output, S. Considering those same specs, the second reference would output, TH.

16.4 Lab

This lab contains data that includes a last name, where last name is all upper-case. It demonstrates the use of intrinsic functions together with reference modification to lower-case the last name characters, except the first character of the last name.

This lab requires two COBOL programs, CBL0011 and CBL0012 and two respective JCL Jobs, CBL0011J and CBL0012J, to compile and execute the COBOL programs. All of which are provided to you in your VSCode - Zowe Explorer.

16.4.0.1 Using VSCode and Zowe Explorer

1. Submit job, CBL0011J.
2. Observe the report output, last name, with first character upper-case and the remaining characters lower-case.

Figure 1. , below, illustrates the difference in output from the Data types lab compared to this lab. Notice that in the previous lab, the last names were listed in all capitalized characters, whereas, as previously stated, this lab output has only the first character of the last name capitalized.

Last Name	Last Name
-----	-----
Washington	WASHINGTON
Adams	ADAMS
Jefferson	JEFFERSON
Lab 8	Lab 7

Figure 141:

Figure 1. Current lab vs. Data types lab output

3. Observe the PROCEDURE DIVISION intrinsic function, lower-case, within the WRITE-RECORD paragraph. This intrinsic function is paired with a reference modification resulting in output of last name with upper-case first character and the remainder in lower-case.
4. Submit CBL0012J
5. Observe the compile error.

Previous lab programs made use of a date/time intrinsic function. The date/time intrinsic function in this lab has a syntax error that needs to be identified and corrected.

6. Modify id.CBL(CBL0012) correcting compile error.*
7. Re-submit CBL0012J
8. Corrected CBL0012 source code should compile and execute the program successfully. A successful compile will result in the same output as CBL0011J.

Lab Hints

Refer to CBL0011 line 120 for the proper formatting of the function-name causing the compile error.