Copyright © Micro Focus 2015-2016. All rights reserved.

Table of Contents

01 COBOL Course Introduction 12
Introduction 12
Module Objectives 12
Student Assumptions 12
Course structure 12
What is COBOL? 13
COBOL Features 13
Brief history of COBOL 14
Why use COBOL? 14
About the Author 15
Class setup 15
Samples 15
Module Summary 16
Quick Quiz 16
02 COBOL Introduction 17
Introduction 17
Module Objectives 17
The structure of a COBOL program 17
Divisions 17
Simple Example 17
Exercise 18
Module Summary 19
Quick Quiz 19
03 Basic COBOL Structure 20
Introduction 20
Module Objectives 20
Programming Layout 20
Identification Division 23
Environment Division 23
Data Division 24
File Section 25
Working-Storage Section 25

Copyright © Micro Focus 2015-2016. All rights reserved.

Linkage Section 25
Procedure Division 26
Use of periods 26
Data File Assignment 27
Module Summary 27
Exercise 27
Quick Quiz 28
4 Data Representation 30
Introduction 30
Module Objectives 30
Defining Data 30
Data Names 31
Data Name restrictions 31
Reserved Words 32
Data Hierarchy 32
The PICTURE Clause 33
Handling numeric data 35
Exercise 1 36
The FILLER clause 36
The USAGE clause 37
Usage DISPLAY 37
Usage BINARY 38
Usage Packed Decimal 38
The VALUE Clause 39
Figurative Constants 40
The REDEFINES clause 41
The COPY statement 42
The COPY REPLACING statement 42
Exercise 2 43
Data item naming 43
Module Summary 44
Further Optional Exercises 44
Quick Quiz 44
05 Basic Verbs 46

Copyright © Micro Focus 2015-2016. All rights reserved.

Introduction 46
Module Objectives 46
Simple Example 46
Statement Termination 47
Commonly used verbs 50
DISPLAY verb 50
ACCEPT verb 50
MOVE verb 51
PERFORM Verb 53
Using Comments 57
STOP Verb 57
COBOL program execution 58
Mixing GO TO and PERFORM 61
Module Summary 62
Exercises 62
Quick Quiz 62
06 Best Practice 63
Introduction 63
Module Objectives 63
Designing a COBOL Program 63
Structure Diagrams 65
Example 66
Truth tables 68
Input and Output files 68
Module Summary 71
Exercise 1 71
Exercise 2 71
Exercise 3 72
Quick Quiz 73
07 Handling Sequential Data Files 75
Introduction 75
Module Objectives 75
Files and Records 75
Program statements required 76

Copyright © Micro Focus 2015-2016. All rights reserved.

Connecting Files 76
Identifying Files 76
Defining file layouts 78
The File Description FD entry 78
File Record structure 78
COBOL verbs for sequential file access 79
Exercise 1 82
Extending the verbs 82
The full READ verb 82
The full WRITE verb 83
Points to remember. 83
File records of different lengths 84
Module Summary 84
Exercise 2 84
Exercise 3 85
Quick Quiz 85
08 Decision Logic 87
Introduction 87
Module Objectives 87
The IF statement 87
Condition phrases 88
The EVALUATE statement 92
Simple EVALUATE 92
Condition EVALUATE 93
Compound EVALUATE 93
The CONTINUE clause 93
Infinite loops 94
Module Summary 95
Exercise 1 95
Exercise 2 96
Exercise 3 – Fixing compilation errors 96
Further Simple Exercises 97
Exercise 4 – Fixing loop and file end problems 97
Quick Quiz 97

Copyright © Micro Focus 2015-2016. All rights reserved.

09 Data Manipulation 99
Introduction 99
Module Objectives 99
Manipulating Data 99
The INITIALIZE verb 99
Arithmetic verbs 100
ON SIZE ERROR clause 104
Verbs used for string handling 105
INSPECT Statement 105
STRING Statement 107
UNSTRING Statement 108
Reference Modification 109
Module Summary 109
Exercise 1 109
Exercise 2 110
Exercise 3 110
Optional Exercises 112
Quick Quiz 113
10 Repeating Data 114
Introduction 114
Module Objectives 114
Representing Repeating Data 114
Keeping a subscript in range 115
Look up tables 116
Exercise 1 116
Indexes 116
Subscripts v Indexes 117
Using the SEARCH verb 117
Exercise 2 117
Modifying index values 117
Multi-dimensional tables 118
Variable length tables 118
Module Summary 119
Exercise 3 119

Copyright © Micro Focus 2015-2016. All rights reserved.

Quick Quiz 119
11 Printing and Reports 121
Introduction 121
Module Objectives 121
Edited Fields 121
Leading Zeros 121
Blank when zero 122
Other leading characters 123
Adding Commas and decimal point 123
Currency symbols 123
Plus and Minus Signs 124
Credit and Debit Signs 124
Insertion characters 125
Coding a Print Program 125
Setting up a print line 126
Writing a print line 127
Designing a Print Program 127
Using Report Writer 129
Module Summary 130
Exercise 130
Quick Quiz 131
12 Using Indexed Files 133
Introduction 133
Module Objectives 133
Indexed File Structure 133
Accessing an indexed file 134
Random Access 135
Exercise 1 137
Accessing an indexed file sequentially 140
Accessing an indexed file dynamically 141
Using Alternate Keys 141
Alternate Key READ 142
File Errors using file status clause 142
Use of Declaratives 143

Copyright © Micro Focus 2015-2016. All rights reserved.

Module Summary 144
Further Exercises 144
Exercise 2 144
Exercise 3 144
Exercise 4 144
Quick Quiz 146
13 Modular Programming 148
Introduction 148
Module Objectives 149
The CALL statement 149
Call using a data name 149
Call Nesting 150
Passing parameters 150
Exercise 1 150
Using a return code 151
Exercise 2 152
Exercise 3 152
Module Summary 152
Quick Quiz 152
14 Screen Handling 154
Introduction 154
Module Objectives 154
Basic Display and Accept 154
Enhancements to Display and Accept 154
Exercise – Screen section syntax 155
Module Summary 155
Quick Quiz 156
15 Database use 157
Introduction 157
Module Objectives 157
Database connection 157
Sample database 157
Accessing the database from COBOL 162
Module Summary 167

Copyright © Micro Focus 2015-2016. All rights reserved.

Exercise ... 167
Quick Quiz ... 167
16 Object Oriented COBOL ... 168
Introduction ... 168
Module Objectives ... 168
Program v Class ... 168
Quick Start Scenario ... 168
Class Structure ... 169
Evolving demonstrations ... 169
Exercise 1 ... 170
Exercise 2 ... 171
Exercise 3 ... 172
Exercise 4 ... 175
Exercise 5 ... 176
Module Summary ... 176
Quick Quiz ... 176
17 Further JVM Features ... 178
Introduction ... 178
Module Objectives ... 178
Project detail ... 178
18 Course Conclusions ... 179
Course Follow-on ... 179
Course Examination ... 179
19 Appendix I - Managed COBOL Review ... 180
Introduction ... 180
Managed COBOL refresher ... 180
Classes & Methods ... 180
Objects ... 183
Creating an instance of a class ... 184
Constructors ... 185
Recap ... 186
Properties ... 186
Method Visibility ... 187
Local Data ... 188

Copyright © Micro Focus 2015-2016. All rights reserved.

Recap 188
Data Types 188
Inheritance 189
Casting 192
Interfaces 192
Class names 195
Intrinsic types 195
The .NET and JVM frameworks 196
Reflection 198
What Next? 198
Further Reading 198
20 Appendix II – Further Features 200
Introduction 200
Module Objectives 202
Other Data File types 202
Report writing 203
Sorting data files 203
Local-Storage Section 203
Intrinsic functions 203
Library routines 203
Module Summary 204

Copyright © Micro Focus 2015-2016. All rights reserved.

Copyright © Micro Focus 2015-2016. All rights reserved.

01 COBOL Course Introduction

Forward

First and Foremost, Welcome. We are glad you have selected Visual COBOL: Modern COBOL for the Next Generation.

We believe in the unique power and flexibility of the COBOL language.

Today, 85% of the world's business applications and 70% of business data processing is dependent on COBOL. COBOL is everywhere and touches our daily lives in many ways. Did you know that the average person interacts with COBOL at least 10 times per day and is typically unaware. The perfect example of this interaction is at the ATM. Each of us, who withdraw money from our banking provider, interacts with COBOL. Yes, there is a fancy, interactive interface at the ATM, but behind the scenes, COBOL is the driving engine, delivering account balance information, depositing funds, or withdrawing money from your account.

But just as ATM interfaces have changed, so has the enterprise application development market. COBOL, once considered, the exclusive 'enterprise' language for business is now challenged by more modern languages such as C#, or Java. Today's business applications are moving forward, seeking to harness new technologies. So, what does this movement mean for COBOL? Can the COBOL language evolve and embrace these next generation technologies as well?

It's time for a step change forward--It's time for Visual COBOL.

This book is designed to teach its reader the Visual COBOL language. With Visual COBOL, you will unlock your potential to assist today's business organizations in maintaining, supporting, and enhancing their critical business applications. You'll also gain a valuable market skill expanding your repertoire as a developer.

I'd like to welcome your on this journey and encourage you to seize this opportunity. Visual COBOL connects the business application world to the exciting, modern, next generation of technologies such as Visual Studio, .NET, mobile computing, and more. You will be amazed by what Visual COBOL can do!

Thanks again for choosing Visual COBOL: Modern COBOL for the Next Generation.

Ed Airey

Product Marketing Director of COBOL Solutions

Micro Focus

Introduction

This course takes you through the structure and features of the COBOL programming language.

It starts with "traditional" COBOL and then moves through the implementation of Object Oriented COBOL in the 1990s and then on to more recent developments; providing fully featured Visual COBOL using all the powers of the Java framework.

Copyright © Micro Focus 2015-2016. All rights reserved.

Examples will be provided, showing how Java code can be used to fully integrate with legacy COBOL code, as well as more modern COBOL applications.

The course will not contain every single detail of COBOL coding but will cover all the major language features that you will most commonly be using. At the end of the course, there is an appendix, which will list further features of COBOL and point to other reading material to extend your knowledge.

Module Objectives

At the end of this module you will be familiar with:

- The reason for COBOL.
- A brief history of COBOL.
- The main features of COBOL.

You will also have set up the sample programs and projects that you will use during this course. In addition you will have configured a number of settings in Visual COBOL which you will need.

Student Assumptions

It is assumed that you, the student:

- Have some programming skills.
- Have little or no knowledge of COBOL.
- Are comfortable with the use of an **I**ntegrated **D**evelopment **E**nvironment (IDE) such as Eclipse.

Course structure

During each module there will be a series of student exercises. The end of each module there will be a short quiz with mainly multiple-choice questions (Or True/False questions). The answers to these quiz questions are provided to your instructor, in a separate document, which can be given to you, as required.

At the end of the course, there will be a final course examination, which is also mainly a set of multi-choice questions. A set of answers are also provided for the instructor.

At the end of the regular course materials you will find 2 Appendices:

- **Appendix 1** – Contains a refresher of Object Oriented COBOL programming with some additional description and features that have not been fully covered in the course.
- **Appendix 2** – Contains descriptions of some less used COBOL features, which may be of interest to you.

What is COBOL?

COBOL (COmmon Business Oriented Language) is one of many high-level computer programming languages.

COBOL was originally designed to solve common business problems, which often require collection, processing and reporting on large quantities of data. For example, COBOL programs generate payroll

Copyright © Micro Focus 2015-2016. All rights reserved.

information, store and report on personnel information, provide stock control and produce profit and loss statements etc.

Some languages are geared toward solving scientific problems that require complex algorithms. Some languages are targeted at user interface features such as Windows Forms or Web Forms. COBOL, in its earliest form, was best suited for financial and business data processing and reporting. In later evolutions of COBOL the use of Windows Forms, Web Forms and Web Services has become very simple to implement, entirely in COBOL.

COBOL still remains very much the language of choice for complex business and financial applications as well as providing very strong simple interfaces to data storage; whether it is regular data storage or integration to databases such as relational, hierarchical or pointer driven databases etc.

COBOL is an easy-to-understand language. It uses English-like structures, such as paragraphs, statements, and verbs. Paragraphs contain statements; statements can include verbs. It also is a highly structured language, where different types of statements must be included in specific parts of the program. This structure helps in the analysis of COBOL programs.

For the above reasons COBOL is a very easy language to maintain. Most commercial applications pass though many update processes in their life span. Ease of maintenance is generally a very high priority when choosing a suitable programming language.

There are hundreds of thousands of COBOL programs in use today; which began life 30 or 40 years ago. These legacy programs represent over 50% of the world's program source, containing billions of line of code. This legacy code is an extremely valuable asset to the Organizations to which they belong.

What we would like to do is to retain as much of this legacy code as possible, while moving forward to more modern user and data interfaces. COBOL provides us with many mechanisms to do just this.

COBOL Features

COBOL includes the following features:

- Representation of commercial concepts of files and records, such as employee name and salary information in personnel records.
- Excellent data definition — data fields customized for each program.
- A wide range of data (numeric and string) manipulation facilities.
- Comprehensive file-handling capabilities, such as methods to open and close files, store file data, and update file records.
- Database handling capability.
- High-level data processing commands (for example, sort, merge, and table manipulation commands).

Copyright © Micro Focus 2015-2016. All rights reserved.

- Conformity to an agreed standard, so that different companies can offer COBOL programming tools with the same features.
- Support for common mathematical functions, such as trigonometric and logarithmic functions, although COBOL was not initially designed to solve highly complex scientific problems.
- Tight integration with Microsoft .NET programming languages.
- The ability to code for Windows Forms, Web Forms and Web Services.

Brief history of COBOL

COBOL was developed by Rear Admiral Grace Hopper in 1959 as a language for commercial implementation. It has a long and steady history, with continued developments, that make it a viable choice for many applications.

COBOL has continually evolved, embracing new technologies with its basic business logic remaining intact.

All through this evolution, COBOL still meets today's computing needs and remains true to strict standards as developed by the American National Standards Institute (ANSI). (Although recent developments in the .NET world have accelerated faster, meaning that the standards, in these areas, have moved forward faster also).

In the past, Mainframe computers processed large quantities of data. Since then, many companies now consider client/server technology for their processing needs. COBOL fully supports this technology, with its basic business logic remaining sound.

COBOL can be viewed as the internal combustion engine of the computing community; it is familiar, perceived as unglamorous, and largely taken for granted. Just like the internal combustion engine, it is and will continue to be, a vital part of many of our society's applications.

However today, with the Object Oriented features and the integration with .NET (and JVM), much of the glamour has returned to COBOL.

Why use COBOL?

Compared with other programming languages, COBOL results in programs that are regarded by some as "verbose". This dramatically differs with languages such as C, where programs can be more obscure. COBOL, on the other hand, encourages coding that is easy to read and therefore, simple to maintain.

Because modern COBOL compilers produce tight, efficient code, wordiness in a program does not reduce performance. COBOL is one of the most readable and self-documenting programming languages in use today. This makes maintenance of COBOL code much simpler than most other languages.

Recent developments of Object Oriented COBOL and integration with the .NET framework have made COBOL the language of choice for many Organizations. This is particularly true for

Copyright © Micro Focus 2015-2016. All rights reserved.

Organizations who have much legacy code that they require to interface to, while moving into more modern user and data interfaces.

About the Author

The author of this training course lectured in commercial application design and programming at the University of Manchester, in England and has presented at many conferences world-wide, both for the University and later for Micro Focus.

He joined Micro Focus as a development team leader and has since worked in various capacities within Micro Focus as a developer, customer consultant, software development team leader, project manager, training development manager and training courseware developer.

He has been a COBOL programmer and advisor for the last 30 years.

Class setup

You can use this course very simply, as though you were reading a book. However this is not the best way to learn a programming language.

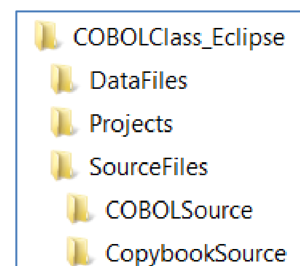
The course uses many illustrated examples and exercises and these are best demonstrated with the use of a program development environment.

So in order to effectively use the materials, provided throughout this course, you are advised to install:

- Micro Focus Visual COBOL
- The set of sample programs supplied with the course
- You will also find it valuable to install the Micro Focus data tools add-pack to allow you to view and edit your data files

Samples

A variety of sample projects, programs and data files are provided. These are installed by executing the self-extracting compressed file **COBOLClass_Eclipse.exe**. These files should be extracted to the root of your **C:** drive to give the following folders:



These folders and files will be used extensively during the running of this class.

Module Summary

At the end of this module you should now be familiar with:

- The reason for COBOL
- A brief history of COBOL
- The main features of COBOL

You have set up the sample programs and projects that you will be using during this course.

In addition you have configured a number of settings in Visual COBOL.

Copyright © Micro Focus 2015-2016. All rights reserved.

Quick Quiz

1. When was COBOL first used?
    a. 1949
    b. 1959
    c. 1969
    d. 1979
2. Traditional legacy COBOL is good at:
    a. JVM integration
    b. Object Orientation
    c. Business Logic
    d. Complex arithmetic
3. Modern COBOL is good at:
    a. JVM integration
    b. Object Orientation
    c. Business Logic
    d. All of these
    e. None of these

Copyright © Micro Focus 2015-2016. All rights reserved.

02 COBOL Introduction

Introduction

In this module we will take you through the basic structure of a traditional COBOL program.

For now we will be looking at COBOL programs which many would regard as traditional "legacy" programs.

Later modules will deal with the later extensions to COBOL which include Object Oriented COBOL and JVM COBOL.

Module Objectives

At the end of this module you will be familiar with:

- The basic structure of a COBOL program.
- A simple example of a traditional program.

The structure of a COBOL program

A COBOL program is divided into **Divisions**.

Each **Division** can be divided into **Sections**.

Each **Section** can be divided into **Paragraphs**.

Each **Paragraph** can contain a number of **Statements.**

Divisions

A COBOL program includes four **Divisions**. Although some compilers will permit the omission of some of the divisions, they must appear in the following sequence.

IDENTIFICATION DIVISION – the first division. Statements here define the name of the program and comments that describe the program's function.

ENVIRONMENT DIVISION – the second division. The particular system environment under which the program will run is defined here.

DATA DIVISION – the third division. This is where all data is defined. All programs manipulate data in some way.

PROCEDURE DIVISION – the final division. This division contains all the statements that determine what the program does. Any data items referred to must have been already defined in the Data Division.

Simple Example

The following COBOL program illustrates the breakdown of the structure of a simple COBOL program:

Copyright © Micro Focus 2015-2016. All rights reserved.

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.   DIVEX.
      *-----------------------------------------
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SOURCE-COMPUTER. IBM-370.
       OBJECT-COMPUTER. IBM-370.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT INFILE     ASSIGN  DATIN.
           SELECT OUTFILE    ASSIGN  DATOUT.
      *-----------------------------------------
       DATA DIVISION.
       FILE SECTION.
       FD INFILE.
       01    INFILE-REC   PIC X(80).
       FD OUTFILE.
       01    OUTFILE-REC PIC X(132).
       WORKING-STORAGE SECTION.
       01    WORK-FIELD   PIC X(20)    Value Spaces.
       LINKAGE SECTION.
       01    LS-FIELD     PIC X(30).
      *-----------------------------------------
       PROCEDURE DIVISION USING LS-FIELD.
       SEC-ONE SECTION.
       PARA-ONE.
          DISPLAY    'Hello World'.
          DISPLAY    'Press <CR> to terminate'
          STOP ' '
          STOP RUN.
      *-----------------------------------------
```

There you can see the 4 **Divisions** of a COBOL program together with **Sections**, **Paragraphs** and **Statements** as appropriate.

Look for the four **Divisions**; don't worry about the specifics of the code for now.

Look for the **Sections** within each **Division.**

Look for **Paragraphs** within each **Section.** (This is not so obvious – there are 5 in total – one of them is contained in **Division**, without a **Section**).

Exercise

1. Start Visual COBOL for Eclipse and set the workspace to:
   C:\COBOLClass_Eclipse\Projects\02_01_Division_Example
2. Double click the **DIVEX.CBL** in the COBOL tab and see the COBOL program shown above.
3. What is the name (ID) of the program?
4. How many sections are there in the sample program? (This is not so obvious – there are 5 in total).

Copyright © Micro Focus 2015-2016. All rights reserved.

5. Run this program by pressing right-clicking on the program name and **selecting Run As/COBOL Application** (This program does not do anything useful except say **'Hello World'** and then asks you to press the **Carriage Return** key).

Module Summary

At the end of this module you will now be familiar with:

- The basic structure of a COBOL program
- A simple example of a traditional program

Quick Quiz

1. The divisions of a COBOL program in order are:
    a. Identification, Environment, Data and Procedure
    b. Identification, Configuration, Data and Procedure
    c. Identity, Environment, Data and Logic
    d. Environment, Data and Procedure
2. The name of a program is contained in:
    a. Data Division
    b. Procedure Division
    c. Identification Division
    d. Linkage Division
3. The original computer that this program was used on was:
    a. A Windows PC
    b. An IBM 360 mainframe
    c. A UNIX machine
    d. An IBM 370 mainframe
4. Which of the following is true:
    a. A section can contain paragraphs
    b. A paragraph can contain sections
    c. A section must contain paragraphs
    d. A paragraph must contain sections

Copyright © Micro Focus 2015-2016. All rights reserved.

03 Basic COBOL Structure

Introduction

In this module we will start to look at the Basic structure of a COBOL program and what each portion of the code will contain.

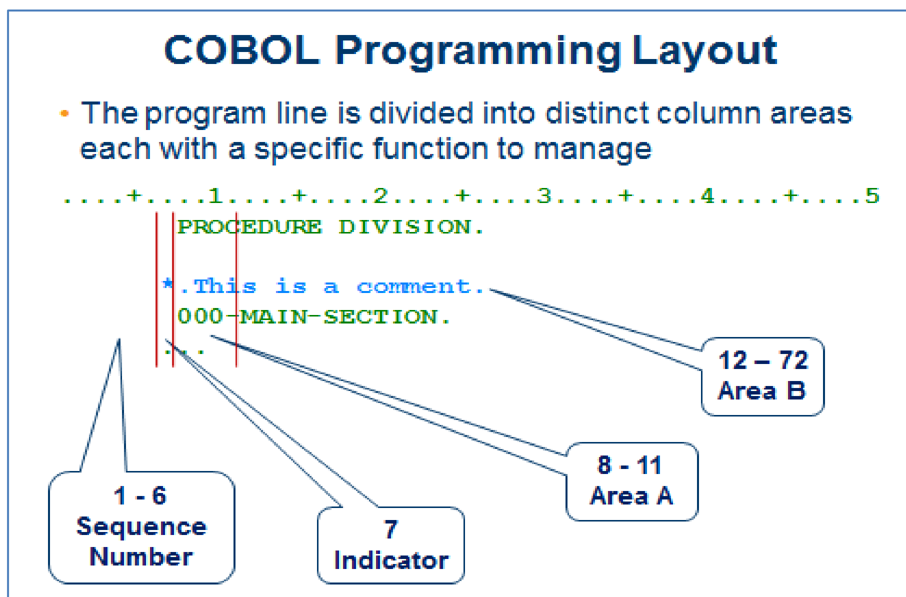
Module Objectives

At the end of this module you will be able to:

- View a basic COBOL program that includes the four divisions and their sections.
- Explain layout-positioning rules imposed by COBOL.
- Execute a simple program.
- See how physical files names can be assigned.

Programming Layout

A typical piece of legacy COBOL code is laid out as shown below:



Where the + indicates 5, 15, 25 etc

Note: We will see in later modules, how this strict layout does not apply so strongly with recent COBOL extensions.

Divisions, Sections, Paragraphs and Statements

From the previous module, recall that a COBOL program includes four divisions and those divisions can contain sections.

This module explores the divisions in more detail and analyzes statements written in each section. Because divisions and statements must be placed in a specific location, we must look at COBOL programming layout.

Copyright © Micro Focus 2015-2016. All rights reserved.

Program Column Layout Rules

COBOL programs expect statements to be confined to particular columns on an invisible page of 72 column lines. The figure below shows only 52 of those 72 columns and shows two lines of code.

```
....+....1....+....2....+....3....+....4....+....5..

       PROCEDURE DIVISION.
       000-MAIN SECTION.
```

Notice that the "P" in Procedure Division appears starting at the 8th column. Each 72-position line represents one line in a COBOL program. Some columns are reserved for specific items in categories called *areas*, as listed in the graphic.

These layout rules are referred to as the *COBOL reference format*.

COBOL Reference Format

- Col 1 – 6, Historically used to contain a line sequence number
- Col 7, Special usage column
  - * (Asterisk) used to denote a comment line
- Col 8 – 11, Area A all Division and Section names start in here
  - (Also known as Margin A)
- Col 12 – 72, Area B main body of program code appears here
  - (Also known as Margin B)
- Within Area B it is common practice to indent coding for clarity

Sequence Number Area

Columns 1–6 are not typically used today. They used to be reserved for line numbering, referred to as ***sequence numbering.*** Line numbers can help identify each line in a program. If you use sequence numbers, they must be six digits *(and since 1985 they can be alphanumeric)*. In the past only the compiler used these columns. So anything here is ignored. This means that if a statement begins, say in column 1, the first six characters of the statement will be ignored.

Indicator Area

Column 7 is a special column, where only certain characters can be inserted. By far the most common of these is an asterisk (*). Placing an asterisk here tells the compiler to see the whole line as a comment. Because COBOL programs may have a very long life, insert comments into your code whenever possible (anywhere in the program after the IDENTIFICATION DIVISION). Comments in the code help ensure that at least some documentation will be available for years to come.

Copyright © Micro Focus 2015-2016. All rights reserved.

The comment below starts with an asterisk on position 7.

```
....+....1....+....2....+....3....+....4....+....5..
      *    *** THIS IS A COMMENT ***
```

Area A - Columns 8-11

Columns 8-11 are known as **Area A.**

All **Division** names, **Section** names, and **Paragraph** headings must start here. Also, all **01** data items, which are discussed later, must start here. Statements under the headings appear indented, making programs easier to read.

NOTE: **Area A** is sometimes referred to as **Margin A.**

COBOL statements in Area A can begin in position 8, 9, 10 or 11. Again to make programs easier to read, however, programmers typically begin statements in Area A at column 8.

Additionally, all **Division** and **Section** headings must appear on one line without other entries and must end with a period.

Paragraph headings must begin in **Area A**, but can appear on a line with or without other entries. Although some paragraph headings do not require periods, adding periods after paragraph headings will never create errors. So, adding the period might be easier to remember.

NOTE: All the divisions can contain sections and paragraphs. In the case of the first 3 divisions the section names and paragraph names are **fixed** names, defined as part of the COBOL language. In the 4th division, procedure division, all the section and paragraph names are **user defined** by you, the programmer.

Area B Columns 12-72

Columns 12-72 are known as **Area B**.

The main body of the program appears here. Also, all entries that began in Area A, or comments, can continue here.

Area B, statements can begin anywhere starting position 12 through 72, but for ease of reading, programmers typically start Area B statements at position 12 and indent related statements.

NOTE: **Area B** is sometimes referred to as **Margin B**.

Statements in Area B end with a period if the statement ends a paragraph or a section.

Identifying Areas

The following code places all the divisions in their correct column layout. The bolded statements begin in Area A at position 8. All other statements appear in Area B starting at position 12.

```
....+....1....+....2....+....3....+....4....+....5..
       IDENTIFICATION DIVISION.
       PROGRAM-ID.           EXAMPLE.
       ENVIRONMENT DIVISION.
```


Copyright © Micro Focus 2015-2016. All rights reserved.

```
CONFIGURATION SECTION.
SOURCE-COMPUTER.       IBM-370.
OBJECT-COMPUTER.       IBM-370.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INFILE      ASSIGN TO 'EMPL.DAT'.
    SELECT OUTFILE     ASSIGN TO 'EMPLOUT.DAT'.
```

This all seems a little complicated!

Hopefully as you go through examples, it will prove to be less so.

The good news is we are still dealing with traditional legacy COBOL code. As we will see later in the course, when we talk of more modern implementations of COBOL, most of these positional rules are relaxed or removed completely.

Of course the colorization indicates different COBOL word types. The actual colors will vary depending on how you have configured the editor.

For now, for legacy COBOL code, we will need to understand these layout rules.

We will now look at each Division individually.

Identification Division

The **Identification Division** is the first division in the program.

It contains the name of the program in the "**Program-ID**" paragraph. (As we will see later in the course, the program-id can be replaced by **class-id** for Object Oriented COBOL).

There are other possible entries in this division, but all these additional entries are treated as comments. e.g.

```
Identification Division.
Program-id.     SimpleStructure.
Author.         Sally Rogers.
Installation.   MARS.
Date-written.   04th July.
Date-compiled.  30th February.
Security.       None.
```

All these entries start in column 8 (Area A).

***Note**: Very early COBOL had to be written in all upper case and many later programs are still written that way. For many years now, upper and lower case are treated as identical. So most of our examples will contain a mixture of case.*

Environment Division

This second division is used to identify any parts of the program that apply to specific computer hardware or devices. This section also specifies any data files used in the program.

This division can contain two sections.

Copyright © Micro Focus 2015-2016. All rights reserved.

CONFIGURATION SECTION contains particular entries that define the actual environment. This section is often omitted, if no such entries are necessary.

INPUT-OUTPUT SECTION is present whenever the program reads from or writes to data files. This section is covered in more detail in a later module, but is used to link the internal file names to external file locations.

e.g.

```
Environment Division.
Configuration Section.
source-computer.   ibm-370.
object-computer.   ibm-370.
Input-Output Section.
File-Control.
    Select InFile  assign datain.
    Select Outfile assign dataout.
```

Where **InFile** and **Outfile** are the internal program names and **datain** and **dataout** are the real physical file names and would have been defined as something like:
C:\COBOLClass_Eclipse\DataFiles\Mydatain.dat and
C:\COBOLClass_Eclipse\DataFiles\Mydataout.dat

Data Division

The Data Division describes the data items needed by the program.

Data can originate from input sources such as files on disk, or tables in databases, or data from internal working areas.

This division often contains the following Sections:

The **FILE SECTION**, which is used when accessing data files.

The **WORKING-STORAGE SECTION**, which holds all the data items that the program needs, e.g. counters and intermediate variables.

The **LINKAGE SECTION**, which is used when a program is called by another program and has data transferred to it.

e.g.

Copyright © Micro Focus 2015-2016. All rights reserved.

```
Data Division.
File Section.
fd  infile.
01  infile-record      pic x(80).
fd  outfile.
01  outfile-record     pic x(100).

Working-Storage Section.
01  Work-field         pic x(20).
01  Counter-field      pic 99.

Linkage Section.
01  ls-field           pic x(10).
```

File Section

This is used to define the data file contents, used during the execution of the program. E.g.

```
File Section.
fd  infile.
01  infile-record      pic x(80).
fd  outfile.
01  outfile-record     pic x(100).
```

In this example:

- The **infile** contains records of length 80 characters.
- The **outfile** contains records of length 100 characters
- The **fd** is **F**ile **D**escription or **F**ile **D**efinition and the names in here must match the names in the SELECT statement of the **input-output** section shown earlier.

Working-Storage Section

This section contains the data definitions of the variables that the program needs during it execution. E.g.

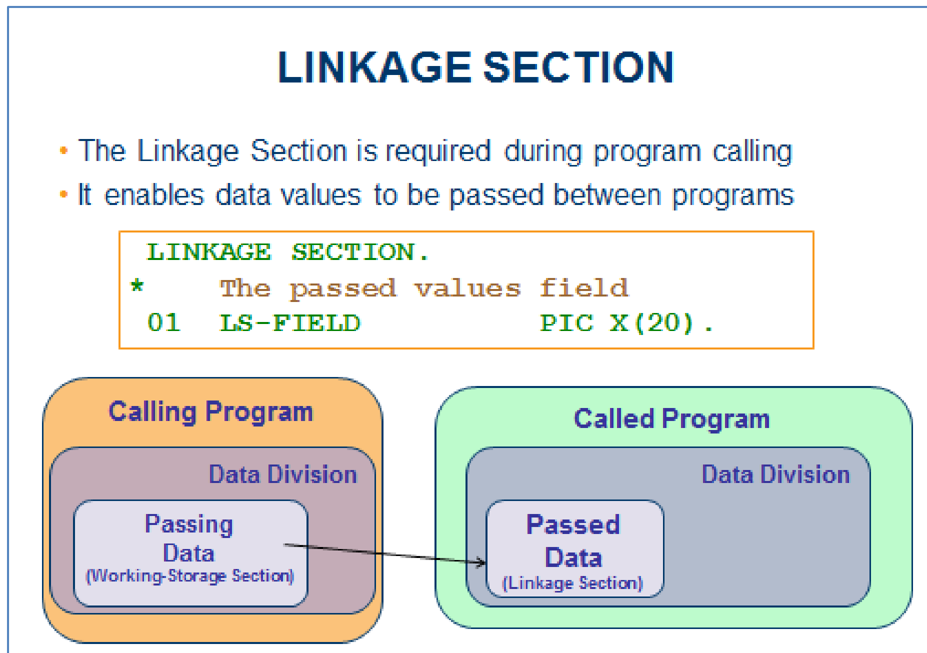
```
Working-Storage Section.
01  Work-field         pic x(20).
01  Counter-field      pic 99.
```

In this example:

- The **Work-field** is an alphanumeric variable of 20 bytes
- The **Counter-field** is an integer numeric field of 2 digits

Copyright © Micro Focus 2015-2016. All rights reserved.

Linkage Section



The **Linkage-Section** is contained in the program that is called, not in the calling program.

Procedure Division

The Procedure Division contains the program's processing statements. E.g.

```
procedure division using ls-field.
000-Main Section.
000-begin.
    open input infile
    open output outfile
    perform 010-read-write
    close infile outfile
    stop 'Press <CR> to terminate'
    stop run.
010-read-write Section.
010-begin.
    read infile
    move infile-record to outfile-record
    write outfile-record
    display outfile-record.
```

As stated earlier, all the section and paragraph names are user defined.

In the above simple example:

- The **infile** is opened ready to be read.
- The **outfile** is opened ready to be written to.
- The **infile** is then **read** (through the **perform** statement)
- The **infile-record** is moved to the outfile-record

Copyright © Micro Focus 2015-2016. All rights reserved.

- The **outfile-record** is written
- The contents of the **outfile-record** is displayed on the screen
- The files are then closed and the program stops.

Use of periods

The use of periods is important. Each division and section definition must end with a period (referred to as a "full stop"). Period is also used to terminate each statement within the first three divisions (IDENTIFICATION, ENVIRONMENT, and DATA).

If you omit a necessary period, the compiler may notice an error only at the beginning of the next line, causing a misleading error message to appear. If you ever see an error message that makes little sense, check missing periods first.

The rules for using periods in the PROCEDURE DIVISION are slightly different, and are discussed in a later module.

Data File Assignment

There are a number of ways to assign files within a program. Full details will be described later in the course. However in this example the mapping of internal file name to external file name has been done in a configuration file. See exercise below.

Module Summary

At the end of this module you are now able to:

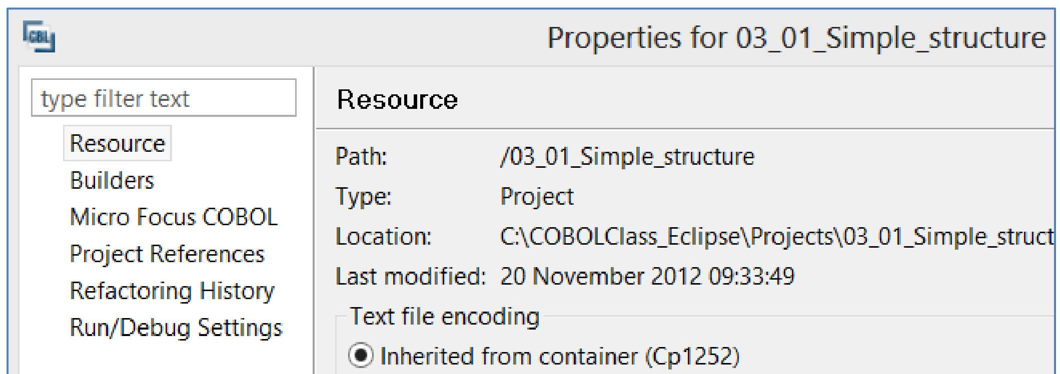
- View a basic COBOL program that includes the four divisions and their sections.
- Execute a simple program.
- Explain layout-positioning rules imposed by COBOL.
- See how physical files names can be assigned.

Exercise

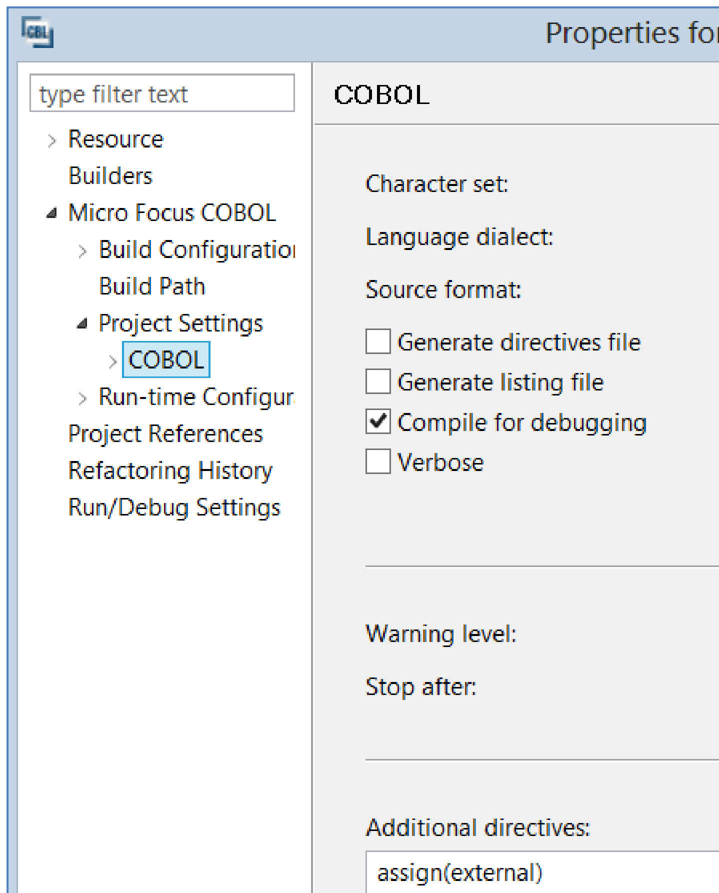
1. Inside Visual COBOL for Eclipse, switch the workspace to **C:\COBOLClass_Eclipse\Projects\03_01_Simple_structure**. (This will show the program you have seen above).
2. First of all examine this program to see the contents of the various divisions, sections and paragraphs.
3. Other than the program-id, change the entries in the Identification Division to be more sensible.
4. Run the program by right-clicking on the program name in the COBOL tab as before and you will see that it displays the details of the first record on the output file.
5. See how the COBOL program has been compiled and configured to look for its files externally. Do this by right-clicking on the project name in the **solution explorer** and

Copyright © Micro Focus 2015-2016. All rights reserved.

selecting **properties.**



6. Now selecting the tab shown. In here you can see that the additional directive **assign(external)** has been set.



7. You will see a new section here; **Linkage Section**. This will be discussed later in the course

Quick Quiz

1. What does **working-storage** section contain?
    a. Details of the file definitions
    b. Details of data used in the program
    c. Details of data passed to the program from another program
    d. Logical definitions of data file locations
2. What must be defined after the **Installation** paragraph?
    a. The name of the program
    b. The fixed value 'MARS'

Copyright © Micro Focus 2015-2016. All rights reserved.

   c. Anything you like.
   d. Nothing must be defined here.
3. In **File-Control** what are the 2 names
   a. The names of the 2 data files
   b. The relationship between internal file name and external file name
   c. The data items used by the file
   d. The procedure names to use in paragraphs
4. What is **Linkage Section** used for?
   a. To contain the file assignments
   b. To contain file data definitions
   c. To allow data to be passed between programs
   d. To contain working data definitions

Copyright © Micro Focus 2015-2016. All rights reserved.

4 Data Representation

Introduction

This module will show the way that data is represented inside a COBOL program. You will see also how data can be grouped into records. In addition, you will also see how copy files are used.

Before you can write COBOL programs, you need to learn about how data is defined and processed.

Data is defined in the same way in all three sections within the DATA DIVISION.

- File Section
- Working-Storage Section
- Linkage Section

Procedural statements to use that data are provided in the PROCEDURE DIVISION.

Data items are often referred to as fields. For example: a last name or a telephone number. In COBOL, rules govern how you can define data items. This module looks at:

- How data items are named.
- Names that are reserved and cannot be used to identify your own fields.
- How data item characteristics can be defined using the PICTURE clause.

Module Objectives

On successful completion of this module you will be able to:

- Explain the different ways in which data can be defined.
- Define data items in a basic COBOL program.
- Use the COPY statement and the COPY… REPLACING statement.
- Use the REDEFINES clause to redefine data.

Defining Data

There are a number of questions to be asked:

- Where is the program data defined?
    - The Data Division!
- Which sections are used?
    - File Section
    - Working-Storage Section
    - Linkage Section
- The lowest level of data is the FIELD (often called an **elementary item)**
- How do we name the data fields?
- How do we define the data field types?
- Are there any rules governing the definition?

Copyright © Micro Focus 2015-2016. All rights reserved.

Data Names

Unlike in some programming languages COBOL encourages you to use long meaningful names. This will help in the future when the program is maintained. For example you could choose names like:

Data Description	Data Name
The employee first name	Employee-First-name
The employee last name	Employee-Last-name
The employee identity number	Employee-ID
The employee salary	Employee-Salary

Data Name restrictions

There are a number of restrictions when choosing a data name:

- Names must be between 1-30 characters
- Names must include at least one alphabetic character
- Names cannot include spaces, so replace spaces with hyphens
- The name cannot start or end with a hyphen
- Names can consist of the following characters:
  - A-Z, a-z, 0-9, hyphen
- Ideally the name should be unique within the program, but it is possible to use the same name in more than one location.
  - It is a bad practice so please try to avoid any duplication
  - It is not always possible – so be careful
- Reserved words are not permissible (Reserved words are names that COBOL itself uses such as ADD, MOVE, READ etc.)

Examples of Valid Data Names

Here are examples of valid user-defined words.

- **EMPLOYEE-GENDER** – This is a meaningful name for an item.
- **WS-VALID-RECORD-COUNT** – Here the WS- prefix is being used to indicate that the item is in the WORKING-STORAGE SECTION of the DATA DIVISION. A prefix such as this is almost universally used.

Examples of Invalid Data Names

Here are examples of invalid data names.

- **PERSONS_SURNAME** Underline characters are not allowed.
- **INPUT** Reserved words are not allowed as a name. “INPUT” is a reserved word.
- **MY-ADDRESS-** You cannot terminate a data name with a hyphen
- **CUSTOMER NAME** Spaces in a data name are not allowed

Copyright © Micro Focus 2015-2016. All rights reserved.

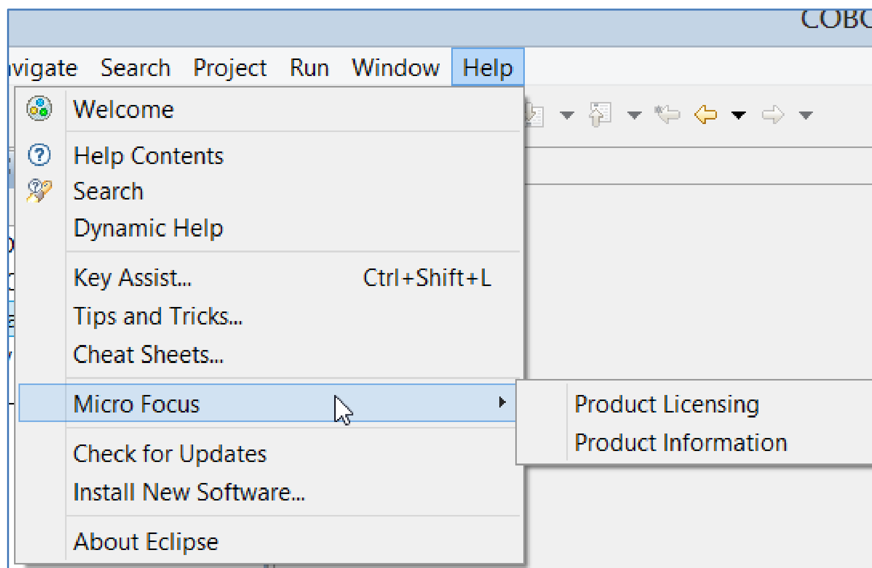
Reserved Words

COBOL uses some words for special processing instructions. These words are known as reserved words. The data item name must not be a reserved word, such as DATA. COBOL uses the word DATA for itself, so "DATA" cannot be used as a data name. However, you can use a reserved word as part of a larger name. "DATA-GROUP" would be acceptable.

The following give examples of some reserved words that cannot be used when naming data fields.

- All Division and Section names and the words "DIVISION" and "SECTION"
- AND, CLOSE, DISPLAY, FILE, INPUT, MOVE, MULTIPLY, NOT, OPEN, OR, PICTURE, RECORD, SPACE

Note: Online Product Information provides a complete list of reserved words.



Data Hierarchy

Defining a field in the DATA DIVISION requires a specific data hierarchy: either at a Group Level or a subordinate Elementary Level.

When describing data, we see not only the size of different data fields, but also how they relate to each other. This is done through level numbers.

- The level numbers aid in displaying the hierarchy
- Indenting aids in displaying the hierarchy is optional.
- Leaving gaps in the numbering allows later insertion of extra levels.
- The 01 level must be positioned in "Area A" and terminated by a period.
- The other data fields must be in "Area B" and terminated by a period.

The following example illustrates this:

Copyright © Micro Focus 2015-2016. All rights reserved.

```
working-storage section.
01  EMPLOYEE-RECORD.
  03  EMPLOYEE-NAME.
    05  EMPLOYEE-TITLE        PIC X(3).
    05  EMPLOYEE-INITIALS     PIC X(4).
    05  EMPLOYEE-SURNAME      PIC X(30).
  03  EMPLOYEE-GENDER         PIC X.
  03  EMPLOYEE-ADDRESS.
    05  EMPLOYER-ADDRESS-LINE PIC X(20) OCCURS 4.
```

Data items defined above are either elementary items or group items.

Elementary items

EMPLOYEE-TITLE	is an elementary 3 character data item
EMPLOYEE-INITIALS	is an elementary 4 character data item
EMPLOYEE-SURNAME	is an elementary 30 character data item
EMPLOYEE-GENDER	is an elementary 1 character data item
EMPLOYEE-ADDRESS-LINE(1)	is an elementary 20 character data item
EMPLOYEE-ADDRESS-LINE(2)	is an elementary 20 character data item
EMPLOYEE-ADDRESS-LINE(3)	is an elementary 20 character data item
EMPLOYEE-ADDRESS-LINE(4)	is an elementary 20 character data item

Group items

EMPLOYEE-ADDRESS	is an 80 character group item containing 4 recurring fields
EMPLOYEE-NAME	is a 37 character group item containing 3 fields
EMPLOYEE-RECORD	is a 118 character group item containing 3 fields (2 of which are themselves group items)

The PICTURE Clause

COBOL provides a method of specifying the characteristics of data items. Data items referenced as Elementary Level items use the PICTURE clause to specify their characteristics. **PICTURE** is most often abbreviated as **PIC**.

PICTURE clauses are very closely related to the concepts of data hierarchy as PICTURE clauses are used only with Elementary Level items.

The PICTURE clause provides information about the type of data stored and the size of the storage area for the item. For example

PIC X(3) The **X** represents alphanumeric data.

PIC A(3) The **A** represents alphabetic data.

PIC 9(4) The **9** represent numeric data

The **(3)** and **(4)** represent the number of characters in the field.

Note: **PIC X(3)** is exactly the same as **PIC XXX**.

Copyright © Micro Focus 2015-2016. All rights reserved.

The following shows further examples of picture clauses:

```
03 DATA-FIELD1 PIC  A(3).                        [Alphabetic]
03 DATA-FIELD2 PIC  XXX.                       [Alphanumeric]
03 DATA-FIELD3 PIC  9(3).                           [Numeric]
03 DATA-FIELD4 PIC  X(6) value "Hello ".            [Literal]
```

Literals (both numeric and alphanumeric) and be applied to data items when defining these. For example is the code example the word "**Hello**" is a literal value of **DATA-FIELD4**.

Alphabetic Data Fields

Use the **PIC A** syntax to identify a field as alphabetic. You could use format 1 or format 2.

Format 1

```
01 DATA-FIELD1  PIC A(3).
```

Format 2

```
01 DATA-FIELD1  PIC AAA.
```

In Format 2 each of the three A's identifies the storage for one character. The 3 in the first format is merely another way to write this.

Alphanumeric Data Fields

Fields containing letters, numbers, and spaces are identified by PIC X.

```
01 DATA-FIELD2 PIC X(3). or
01 DATA-FIELD2 PIC XXX.
```

Numeric Data Fields

Define a field containing all numbers along with optional + or – sign as numeric data by using the PIC 9999 clause. Each 9 represents one digit.

```
01 DATA-FIELD3 PIC 9(4). or
01 DATA-FIELD3 PIC 9999.
```

Numeric fields can be up to a maximum of 31 digits.

Always make the definitions for numeric fields large enough to accommodate any possible number. If a field needs to hold 1 million, many programmers will at first define this in a PIC 9(6) field. Such a field will hold 999,999 with ease. However, adding one sets the field to zero, as the leading '1' is lost to the left – probably not what is needed.

Decimal Values

In the PICTURE clause, a V represents the decimal point, but takes up no space within the data item – it is an "implied" decimal point position.

```
01 EMPLOYEE-SALARY    PIC 99999V99.
```


Copyright © Micro Focus 2015-2016. All rights reserved.

Each 9 after the V represents a digit after the decimal point. So, an EMPLOYEE-SALARY would appear as $50000.00, two digits after the decimal point. Alternatively, you could write this as:

```
01 EMPLOYEE-SALARY    PIC 9(5)V9(2).
```

Negative Numbers

The notation of 9(5)V99 allows only for zero or positive values. However, if we needed to allow for negative numbers, for example, for a customer's account balance, we would say something like the following.

```
01 CUSTOMER-BALANCE   PIC S9(5)V99.
```

S stands for 'signed'. The sign is stored on the data item itself and does not take an extra byte.

Literals

Literals are used to give data items specific values, either numeric values (for example, 53 or 2) or alphanumeric values (for example, "Hello"). Literals represent the actual contents of the data rather than the name of the data item.

```
03 DATA-FIELD4     PIC X(6) Value 'Hello '.
03 AMOUNT-FIELD    PIC 9(4) Value 5678.
```

The following rules apply to using numeric literals in COBOL programs.

Numeric literals must be 1–31 digits.

- A + or – sign can be included, but only at the left.
- A decimal point can be included, but not at the end of the literal.

The following rules apply to using alphanumeric literals in COBOL programs.

- Alphanumeric literals can contain up to 268,434,912 characters.
- Surround the literal with either single or double quotation marks (for example, "USA" or 'USA').

Handling numeric data

Numeric data is quite variable, it can be held as:

- Integers; small and large number
- It can be held in decimal format
- The number to be stored may be positive or negative
- It may contain currency, dates, time or telephone numbers

Examples:

Copyright © Micro Focus 2015-2016. All rights reserved.

```
03 DATA-FIELD1 PIC  99999.
03 DATA-FIELD2 PIC  9(5).
03 DATA-FIELD3 PIC  9(5)v99.
03 DATA-FIELD4 PIC  S9(5)v99.
03 DATA-FIELD5 PIC  9(5)      VALUE 19856.
```

Exercise 1

1. In Visual Cobol to Eclipse switch the workspace to **C:\COBOLClass_Eclipse\Projects\04_01_Data_Representation**.
2. Use the **DataRep1.cbl** to add a record layout for a company employee. There is already a record there for CUSTOMER. So add a new record for EMPLOYEE to include the following data fields:
    a. Title (3 characters or Bytes)
    b. Initials (4 bytes)
    c. Surname (30 bytes)
    d. Gender (1 byte)
    e. Address lines (4 lots of 30 bytes)
    f. Postal code (8 bytes)
3. Add a Salary field for the employee to hold 5 bytes of before decimal point and 2 after
4. Change the layout to ensure the address lines are grouped
5. Change the layout to ensure the name can be accessed as one unit
6. How long is the record layout in bytes?

If you require a solution program to the above exercise, **DataRep2.cbl** is contained in the same folder as **DataRep1.cbl**.

The FILLER clause

Maybe you are interested only in parts of a Group Level data item.

Looking at our employee record again, let's say that we have to write a program that examines only the one-byte field containing the GENDER value, and we are not interested in any of the preceding or following fields.

However, as this is at byte 38 — the previous data fields were defined as PIC X(3), PIC X(4), and PIC X(30) — the structure below is **incorrect**. This will be pointing at byte 1 of the group field, not byte 38.

```
01  EMPLOYEE-RECORD.
    03  EMPLOYEE-GENDER      PIC X.
```

Since for this case we are not interested in those first 37 bytes, we can use a FILLER as shown next.

```
01  EMPLOYEE-RECORD.
    03  FILLER               PIC X(37).
    03  EMPLOYEE-GENDER      PIC X.
    03  FILLER               PIC X(135).
```


Copyright © Micro Focus 2015-2016. All rights reserved.

Use a FILLER clause to show that data is present, but you have no interest in accessing it. FILLER clauses can be found in almost every program.

Instead of saying FILLER, you can omit the name of the data item, as shown next. This is treated in exactly the same way by the compiler.

```
 01  EMPLOYEE-RECORD.
     03                       PIC X(37).
     03  EMPLOYEE-GENDER      PIC X.
     03                       PIC X(135).
```

The USAGE clause

Previously, we have defined data characteristics using the PICTURE clause.

Next we look at ways to indicate different storage methods when defining data items.

This is done with the USAGE clause. The USAGE clause tells the computer how to represent numbers internally. Here are some USAGE clauses:

```
03  FIELD1     USAGE DISPLAY     PIC 9(4).
03  FIELD2     USAGE DISPLAY     PIC 9(4) COMP.
03  FIELD3     USAGE DISPLAY     PIC 9(4) COMP-3.
```

By default, when usage is not specified, numeric data is held in the format we have seen previously, as shown in the following example.

```
01  RECORD-COUNTER          PIC 9(4).
```

This is known as **USAGE Display**. The number is stored as one digit per byte, in just the same way as an alphanumeric (PIC X) or alphabetic (PIC A) field. Such fields work perfectly well, in that you can carry out any arithmetical operations on them that you wish.

Yet, other methods of storing numeric data are more efficient (the calculations take place more rapidly) and the data takes up less room. Programmers tend to use such data types for internal data items (ones say in WORKING-STORAGE).

Let us look at three ways in which a data item that needs to hold up to 9,999 can be stored. (On some compliers there are other variants too such as **COMP-5** and **COMP-X**). Where the word COMP is an abbreviation of COMPUTATIONAL

```
Usage Display Format        - PIC 9(4)

Usage Binary Format         - PIC 9(4) COMP

Usage Packed Decimal Format - PIC 9(4) COMP-3
```

Usage DISPLAY

The base form is USAGE Display,

```
01  RECORD-COUNTER          PIC 9(4).
```

- Has a value range from 0000 to 9999
- It can be used for calculations and any other numeric operations

Copyright © Micro Focus 2015-2016. All rights reserved.

- This allocates 4 bytes to store the number

This method (Usage Display) allocates one byte per digit, which is what we are familiar with. If the number contained "1234", it would be stored like this.

```
00110001 00110010 00110011 00110100     in binary
31 32 33 34                             in hex
```

Usage BINARY

This format allocates the number of "Bits" to the field for storage

```
01  RECORD-COUNTER          PIC 9(4) USAGE COMPUTATIONAL.
```

Or more often defined as the abbreviated form

```
01  RECORD-COUNTER          PIC 9(4) COMP.
```

Or

```
01  RECORD-COUNTER          PIC 9(4) BINARY.
```

- Then rounds up to next whole byte
- So a number 9,999 can be held in 14 bits
- 8 bits to a byte
- On rounding up we will use 2 bytes
- Has a value range from 0000 to 9999
- It is more efficient for numeric operations

The Binary Format method allocates the actual number of bits needed to hold the maximum number (9,999) and then rounds that up to the next byte. So, 9999 (binary 10011010101011) can be held in 14 bits and would occupy 2 bytes (16 bits). The number "1234" would be stored as shown in the following.

```
10011010010        in binary
04D2               in hex
```

Storing data in binary numeric fields is the most efficient.

However, there is a tradeoff for gaining efficiency in this way.

It is more difficult to know how much space a COMP numeric field requires, for example, if the field is in a record.

The following lists the space required for various COMP fields on the PC (not mainframe).

PIC 9 or PIC 99 COMP	1 byte
PIC 999 or PIC 9(4) COMP	2 byte
PIC 9(5) or PIC 9(6) COMP	3 bytes
PIC 9(7), PIC 9(8) or PIC 9(9) COMP	4 bytes

Copyright © Micro Focus 2015-2016. All rights reserved.

PIC 9(10) or PIC 9(11) COMP 5 bytes
PIC 9(12) , PIC 9(13) or PIC 9(14) COMP 6 bytes
PIC 9(15) or PIC 9(16) COMP 7 bytes
PIC 9(17) or PIC 9(18) COMP 8 bytes

Signed fields are exactly the same size (the number is held in two's complement format). A decimal number is stored according to its total number of digits, for example a PIC 9(6)v9(4) will fit in 5 bytes.

Usage Packed Decimal

The third method (Packed-Decimal Format) is in some ways a hybrid between the first two.

The number is stored or *packed* more compactly than in display format, and it can be accessed with most of the extra efficiency of a binary field, yet it is still possible to see its value if you look at the field's hex representation.

The space allocated is one half byte for each digit and one half byte for the sign (whether the field is signed or not). If necessary, the number of bytes is then rounded up to the next whole number.

An example will make things clearer. A data item to hold **9(4) COMP-3** will consist of two and a half bytes – four half bytes for the digits and half a byte for the sign. The two and a half is then rounded up to three bytes. The number '1234' would be stored as follows.

```
00000001   00100011   01101111   in binary
01 23 4F                         in hexadecimal
```

In the example above note the 'F' for the sign. If a number is unsigned, as here, then this last half byte will always be hex F (binary 1111). If the number is signed, then the last half byte will be one of the following:

If the number is positive, hex C (binary 1100) with the "C" representing "Credit.")

If the number is negative, hex D (binary 1101) with the "D" representing "Debit."

To calculate the length of a COMP-3 field, divide the number of digits (PIC 9s) by 2, and then round up to the next whole byte if necessary.

The VALUE Clause

It is often useful to be able to preset the value of one or more data items, so that these values are in place before any code in the PROCEDURE DIVISION is executed. This is done with the VALUE clause along with literals or constants.

Let's look at a couple of examples.

```
01  WS-ITEMS.
    03  WS-NAME         PIC X(20)    VALUE 'SMITH'.
    03  WS-SALARY       PIC 9(6)V99  VALUE 27500.
    03  WS-AGE          PIC 99       VALUE ZERO.
```

The VALUES shown in lines 3 and 4 are known as *numeric literals.* In the case of PIC X or PIC A fields, the values must be enclosed in quotes.

Copyright © Micro Focus 2015-2016. All rights reserved.

In the above example, WS-NAME will now contain “SMITH” followed by fifteen spaces.

WS-SALARY will contain 027500.00. In other words, if the value of the literal does not fill up the field, then character fields are padded with spaces on the right and numeric fields padded with zeros on the left.

Entering too large a value into a field (for example, giving WS-NAME a value of “General Dwight D Eisenhower”) causes the COBOL compiler to identify an error.

Also, use a numeric literal that matches the PICture of the item. Since WS-SALARY is defined as PIC 9(6)V99, (the first 9 indicating a numeric data type), it would be wrong to assign SALARY a VALUE of “NONE” (which is an alphabetic value).

Figurative Constants

In addition to numeric literals, you can use VALUE clauses with figurative constants. Figurative constants preset data items to useful values. The following figurative constants are available.

- SPACE or SPACES (which mean the same thing)
- ZERO or ZEROS (which mean the same thing)
- LOW-VALUES
- HIGH-VALUES
- ALL “literal”

The SPACES Figurative Constant

SPACES, the easiest figurative constant to understand, is designed for use on alphanumeric or alphabetic fields. Ensure that such fields are cleared out before any data manipulation takes place. The following clause includes the SPACES figurative constant.

```
      03  WS-NAME       PIC X(20)   VALUE SPACES.
```

The next example illustrates a more problematic use of SPACES.

```
  01  WS-RECORD.
      03  WS-NAME      PIC X(20).
      03  WS-ADDRESS   PIC X(100).
      03  WS-SALARY    PIC 9(5)V99.
```

You could try to “clear” the record using:

```
MOVE SPACES TO WS-RECORD
```

Group fields are always regarded as being alphanumeric (even if all the Elementary Level fields are numeric). In this case, WS-SALARY will end up set to spaces, which might not be appropriate.

The way in which you can clear WS-RECORD, so that all alpha and alphanumeric items are cleared to SPACES and the numeric items are zeroed is:

```
INITIALIZE WS-RECORD
```


Copyright © Micro Focus 2015-2016. All rights reserved.

The ZERO Figurative Constant

ZERO (or ZEROES) can be similarly thought-provoking. ZERO will put that value in the field or fields referred to by the VALUE clause and format it using the PICture information.

The following clause will move “character zero” (hex 30) into each of the bytes of WS-NUMBER.

```
    03  WS-NUMBER     PIC 9(4) VALUE ZERO.
```

Another example will result in binary zero (hex 00) throughout the two bytes of the field, as shown next

```
    03  WS-BIN-NUMBER PIC 9(4) COMP VALUE ZERO.
```

This results in a nasty little trap that is very easy to fall into.

In the following sample, the programmer is trying to set all four items in the group to zero. Instead, ZERO gets moved to the group. Because the group is *alphanumeric,* hex 30 is moved to each of the eight bytes, setting all the items to 2336! (hex 3030). Note: Each field defined in the example as PIC 9(4) COMP is stored as 2 bytes.

```
 01  WS-NUMBERS.
     03  WS-NUM-1     PIC 9(4) COMP.
     03  WS-NUM-2     PIC 9(4) COMP.
     03  WS-NUM-3     PIC 9(4) COMP.
     03  WS-NUM-4     PIC 9(4) COMP.
```

So; `MOVE ZERO TO WS-NUMBERS` will **not** put 0 into each of the individual fields.

The correct way to do this is:

```
INITIALIZE WS-NUMBERS
```

The LOW-VALUES Figurative Constant

To avoid the previous dilemma, you can also use the LOW-VALUES figurative constant, which sets a field to its lowest possible value. Preset the group to “binary zero.” The following example shows the other correct way of achieving the goal.

```
01  WS-NUMBERS.
     03  WS-NUM-1     PIC 9(4) COMP.
     03  WS-NUM-2     PIC 9(4) COMP.
     03  WS-NUM-3     PIC 9(4) COMP.
     03  WS-NUM-4     PIC 9(4) COMP.
```

```
MOVE LOW-VALUES TO WS-NUMBERS
```

Use LOW-VALUES to set a field or fields to its lowest possible value hex 00. HIGH-VALUES is the opposite. HIGH-VALUES moves hex FF to every byte of the field or fields in question.

This of course would be no use to us if the group item contained some alphanumeric field(s). So as a general rule you should use INITIALIZE to clear out a group to its default empty values (i.e. SPACES in alpha and alphanumeric fields and zero in numeric fields).

The ALL “literal”

This fills a field with a character value, as shown in the following example.

Copyright © Micro Focus 2015-2016. All rights reserved.

```
   03  WS-STARS     PIC X(30) VALUE ALL '*'.
   03  WS-LINE      PIC X(80) VALUE ALL '_'.
```

Important note: Use the VALUE clause, as shown above, only in the Working-Storage Section. It has no relevance in the FILE section or the LINKAGE section since these fields are populated from outside (either by reading from a file or from a calling program).

The REDEFINES clause

The REDEFINES clause gives you a way to manipulate data with flexibility. It allows you to specify a different PICTURE clause for a data item defined previously.

Let's say that we have a group of items in Working-Storage, perhaps copied in from a record that was read. One of those fields is normally a numeric birthdate.

However, if the birthdate is not known, the field needs to be alphanumeric, so that it can contain some other value, as shown in the following code.

```
   01  WS-RECORD.
       03  WS-NAME        PIC X(30).
       03  WS-DOB         PIC 9(8).
       03  WS-DOB-XX      REDEFINES WS-DOB    PIC X(8).
```

The redefining item must immediately follow the one being redefined. Redefine items at any level (except 01 in the File Section, as we shall see later). A redefined *item* can have a VALUE clause, but the redefining *clause* cannot.

The COPY statement

What happens when a data item needs to be defined in more than one program?

Record layouts may be very complex and contain hundreds of coding lines. Is it wise to repeat them?

Sometimes Working-Storage items will be common across different programs. How do you handle these items?

Rather than repeat the definition in each case, we place these definitions in an external copy file and then we use the COPY statement in the program.

Let's look at an example where all programs in a payroll suite need the following same definitions for working storage counters.

```
 01  WS-COUNTERS.
     03  WS-NUMBER-OF-EMPLOYEES   PIC 9(4).
     03  WS-TOTAL-SALARY          PIC 9(8)V99.
     03  WS-TOTAL-TAX             PIC 9(7)V99.
     03  WS-TOTAL-DEDUCTIONS      PIC 9(6)V99.
```

When code needs to be shared in this way, the best solution is to store the relevant lines in a *copyfile*, that is, a text file that can be automatically copied into the program.

In the following example, the file was saved as **counters.cpy**. The COPY statement is typically inserted in Area A, though it is also valid in Area B.

```
   COPY COUNTERS.CPY
```


Copyright © Micro Focus 2015-2016. All rights reserved.

Sometimes full stops (periods) are absolutely necessary. The full stop at the end of a COPY statement must never be omitted. In the above example, such an omission would cause the compiler to be unable to compile the statement following the COPY.

The COPY REPLACING statement

You can create a “generic” copy file, where the actual names of the data items can be modified to suit the particular program (rather than having to use the names in the copy file). This is done using tags. The copy file can then be imported into a program, and the (TAG) element can be replaced with another value. The word **TAG** is just an example; it is not a keyword.

First, we set up a copy file as follows. We named it “ws.cpy”.

```
01  (TAG)-NUMBERS .
    03  (TAG)-NUM-1     PIC 9(4).
    03  (TAG)-NUM-2     PIC 9(6).
    03  (TAG)-NUM-3     PIC 9(5).
```

In the example below, the file is imported twice.

```
COPY WS REPLACING ==(TAG)== BY ==WS01==.
COPY WS REPLACING ==(TAG)== BY ==WS02==.
```

This makes the following data items available.

```
01  WS01-NUMBERS.
    03  WS01-NUM-1     PIC 9(4).
    03  WS01-NUM-2     PIC 9(6).
    03  WS01-NUM-3     PIC 9(5).

01  WS02-NUMBERS.
    03  WS02-NUM-1     PIC 9(4).
    03  WS02-NUM-2     PIC 9(6).
    03  WS02-NUM-3     PIC 9(5).
```

While copy files are commonly used to hold data hierarchies, such as records, they can also be used to hold executing code, that is, statements that are found in the Procedure Division.

Exercise 2

1. In Visual Cobol switch the workspace to **C:\COBOLClass_Eclipse\Projects\04_02_Data_Representation**.
2. Look at the program to see how the same copy file has been used twice in the program. In the 2 versions the ‘tag’ has been replaced with different values.
3. You can look at the contents of the copy file by double clicking the WS.CPY file the COBOL tab.
4. Execute the program as before to see how the different versions of the copy file are used.
5. How would you initialize the copy file field contents with zeroes at run-time?
6. Reset the values of the counters to zeroes during the execution to simulate a restart of the program.
7. What would be the effect of setting the **WS01-Numbers** group to **Low-Values**?

Copyright © Micro Focus 2015-2016. All rights reserved.

Data item naming

It is suggested that data names should be unique, and this is very often easy to achieve. Sometimes identical names *can* be used, as shown in the following example.

```
01  WS-INPUT-AREA-1.
    03  EMPLOYEE-NAME     PIC X(20).
    03  EMPLOYEE-ADDRESS  PIC X(60).
    03  EMPLOYEE-SALARY   PIC 9(6)V99.

01  WS-INPUT-AREA-2.
    03  EMPLOYEE-NAME     PIC X(20).
    03  EMPLOYEE-DOB      PIC 9(8).
```

The COBOL compiler tolerates any number of identically-named fields, until it has to distinguish between them. For example, the following statement from the Procedure Division will be flagged as an error.

```
MOVE 'VLAD THE IMPALER' TO EMPLOYEE-NAME
```

To anticipate a later portion of this course, the MOVE statement moves a value ("VLAD THE IMPALER") to a data item (EMPLOYEE-NAME).

Here the compiler cannot recognize the EMPLOYEE-NAME, as the data name is not unique.

The solution is to *qualify* the data name, making it unique, as shown by the addition of **OF WS-INPUT-AREA-2** e.g.

```
MOVE 'VLAD THE IMPALER' TO EMPLOYEE-NAME OF WS-INPUT-AREA-2
```

Module Summary

On completion of this module you can now:

- See the different ways in which data can be defined
- Define data items in a basic COBOL program
- Use the COPY statement and the COPY… REPLACING statement
- Use the REDEFINES clause to redefine data

Further Optional Exercises

You will find a number of further optional exercises which are appropriate at the end of this module.

When you decide to do these (either now or after, as a refresher) you should open the **Solutions** shown below and then work your way through the sample program(s) in the solution.

The recommended way is to execute in **Debug Mode** to fully understand what the code is doing and what the data item values are. To do this, right-click on the program name and select **Debug As/COBOL** Application

The workspaces which are appropriate are:

- **04_03_ Numeric initialization**
- **04_04_Field type moving**

Copyright © Micro Focus 2015-2016. All rights reserved.

- **04_05_Numeric moving**

Quick Quiz

1. Which of the following statements are true?
    a. There are no data name restrictions.
    b. Level Numbers are used to group records.
    c. "PIC S9(5)v9(3)" represents an integer field.
    d. COBOL Reserved Words can't be used to identify data names
    e. None of the above
2. Which statement(s) are true about the FILLER clause?
    a. FILLER can only be used in working-storage section.
    b. FILLER can be used in any of the data division sections
    c. The word FILLER can be replaced with spaces.
    d. You can move data into a FILLER in procedure division
    e. None of the above
3. Which is true of Numeric fields?
    a. Numeric fields are always signed
    b. Numeric fields are always stored as binary
    c. Numeric fields have a maximum size
    d. None of the above
4. Which of the following is true about figurative constants?
    a. Used in data division
    b. Not allowed
    c. Is one of a fixed number of values
    d. Can only be applied to numeric data items
5. Which of the following are valid data names?
    a. CUSTOMER-NUMBER
    b. Customer-Number
    c. CUSTOMER_NUMBER
    d. CUSTOMER-NUMBER-
    e. 1234
    f. C-1234
    g. All the above
    h. None of the above
6. Which is true of REDEFINES?
    a. Redefines is used only in the environment division
    b. Redefines is mainly used to use the same field with different pictures
    c. Redefines is used mainly in procedure division
    d. All the above
    e. None of the above
7. Which of the following is true about COPY?
    a. COPY is used to refer to an external file containing COBOL code
    b. COPY is used to redefine a data item
    c. COPY is used in procedure division to move one data item to another
    d. All the above

Copyright © Micro Focus 2015-2016. All rights reserved.

e. None of the above

Copyright © Micro Focus 2015-2016. All rights reserved.

05 Basic Verbs

Introduction

This module will deal with some of the basic “verbs” that you will find in the procedure division of a COBOL program.

Do you recall that the PROCEDURE DIVISION is the final division in the program?

It is where the logic code is written.

The PROCEDURE DIVISION heading starts in Area A

Code written in this division should be

- contained in sections and paragraphs
- each section and paragraph heading starts in Area A

In COBOL each line of procedural code begins with a “verb”.

A “verb” is effectively the start of an instruction to perform some kind of operation. There is a strict set of “verbs” that COBOL uses. Most of them we would understand as a “verb” in the English language, but others are not strictly English language verbs.

Module Objectives

On successful completion of this module you will be able to:

- Explain the DISPLAY, ACCEPT, STOP, MOVE, GO TO and PERFORM verbs
- Describe how the following verbs direct program logic
    - PERFORM ...
    - PERFORM … UNTIL
    - GO TO

Simple Example

We will start by using a very simple example program. This example can be found by setting the workspace to **05_01_Basic_Verbs**

The program here contains the following code:

Copyright © Micro Focus 2015-2016. All rights reserved.

```
IDENTIFICATION DIVISION.                      *> [Division]
PROGRAM-ID.    BasicVerb1.                    *> [Paragraph]
ENVIRONMENT DIVISION.                         *> [Division]
DATA DIVISION.                                *> [Division]
WORKING-STORAGE SECTION.                      *> [Section]
01  WS-MESSAGE     PIC X(20) VALUE SPACES.
01  WS-NAME        PIC X(20) VALUE SPACES.
PROCEDURE DIVISION.                           *> [Division]
THE-ONLY SECTION.                             *> [Section]
THE-ONLY-PARAGRAPH.                           *> [Paragraph]
    MOVE 'Hello world' TO WS-MESSAGE          *> [Statements]
    DISPLAY WS-MESSAGE
    DISPLAY 'Please enter your name:'
    ACCEPT WS-NAME
    DISPLAY 'Nice to meet you, ' WS-NAME
    MOVE 'Goodbye' TO WS-MESSAGE
    DISPLAY WS-MESSAGE
    STOP 'PRESS <CR> TO STOP'
    STOP RUN.
```

In this example, the Procedure Division contains a single section, which contains a single paragraph.

The paragraph contains 9 statements, which in turn use 4 “verbs”. The verbs being used are:

- MOVE — This copies the string to the target data item
- DISPLAY — This displays a value on the screen, at the next line
- ACCEPT — This requires you to enter a value on the screen and places the value into the data item
- STOP — The first Stop display a message requiring you to press <CR>
  The Stop Run, terminates the program

This program should be run by pressing **F5** to see the results of the execution. The result of running will be:

```
Hello world
Please enter your name:
George Washington
Nice to meet you, George Washington
Goodbye
PRESS <CR> TO STOP
```

Where “George Washington” was the name you entered on the screen.

Statement Termination

Up until the mid-1980s it was common practice to terminate each statement with a full stop (period). Indeed in many cases it was essential to do this.

Since the mid-1980s, additional syntax was introduced into COBOL which meant that statement termination was done by “statement terminators”.

Let’s see an example using the IF verb (Yes IF is a “verb” in COBOL).

Copyright © Micro Focus 2015-2016. All rights reserved.

Pre 1985

```
IF CUSTOMER-AGE < 21
    DISPLAY 'CUSTOMER TOO YOUNG'.
```

Post 1985

```
IF CUSTOMER-AGE < 21
    DISPLAY 'CUSTOMER TOO YOUNG'
END-IF
```

Another example might be:

Pre 1985

```
IF CUSTOMER-AGE < 21
    IF PARENTS-CONSENT = 'YES'
         DISPLAY 'CONSENT GIVEN'
    ELSE
         DISPLAY 'CUSTOMER TOO YOUNG'.
```

This period terminated both **IFs**.

Post 1985

```
IF CUSTOMER-AGE < 21
   IF PARENTS-CONSENT = 'YES'
        DISPLAY 'CONSENT GIVEN'
   ELSE
        DISPLAY 'CUSTOMER TOO YOUNG'
   END-IF
END-IF
```

On the face of it this does not seem to be much different, but consider how this post 1985 statement could have been coded pre 1985

```
IF CUSTOMER-AGE < 21
    IF PARENTS-CONSENT = 'YES'
         DISPLAY 'CONSENT GIVEN'
    END-IF
ELSE
    DISPLAY 'CUSTOMER TOO YOUNG'
END-IF
```

Where could you possibly place the period, since there are 2 nested IF statements?

Even today, there are still many relatively newly written COBOL programs using the period to terminate conditional statements.

This is **NOT** the way that we recommend, although we still have to be able to maintain programs that were written that way. So we need to be aware that a period can still be used to terminate a statement (In particular a conditional statement).

In answer to the question above about placing the period, one way that this used to be achieved is with the GO TO statement. A clumsy example would be:

```
 START-IF.
```


Copyright © Micro Focus 2015-2016. All rights reserved.

```
        IF CUSTOMER-AGE < 21
            GO TO TEST-PARENT
        ELSE
            DISPLAY 'CUSTOMER TOO YOUNG'.
   IF-CONTINUE.
              . . . .
   TEST-PARENT.
        IF PARENTS-CONSENT = 'YES'
            DISPLAY 'CONSENT GIVEN'.
        GO TO IF-CONTINUE.
```

Very messy !! (There are simpler ways, but none are so clear as shown in the post 1985 example above)

The use of the GO TO statement was prolific prior up to 1985. Since 1985 there is never a need to use a GO TO statement. There is always a better, structured way of achieving this.

The use of GO TO is a recipe for making error filled and hard to maintain programs. It is still supported, but our recommendations are that, if you must use it, then think very carefully of its implications for maintainability.

However the only consideration is that the COBOL compiler must be able to tell where one statement ends and the next begins. This yields the following rules.

The last statement in a paragraph or section *must* have a period or full stop, as shown in the following example.

```
SECTION-6 SECTION.                 *> [Section header]
PARA-6.                            *> [Paragraph header]
    DISPLAY 'I JUST MOVED 10 TO THE TOTAL'.
PARA-7.                            *> [End of paragraph]
    DISPLAY ' YOU ARE NOW ON PARA-7'
    IF A = B
        ADD 1 TO B
    END-IF.
PARA-8.
    MOVE A TO B
    ADD 8 TO B.
```

Post 1985, any statement that is immediately followed by another verb is implicitly terminated, so it does **not** need a period or full stop (*unless* it is the last statement in a paragraph or section).

My own personal preference (and that of many others) is to terminate paragraphs as shown below with a single period on a line of its own, at the end of a paragraph:

```
SECTION-6 SECTION.
PARA-6.
    DISPLAY 'I JUST MOVED 10 TO THE TOTAL'
    .
PARA-7.
    DISPLAY ' YOU ARE NOW ON PARA-7'
    IF A = B
        ADD 1 TO B
    END-IF
    .
PARA-8.
    MOVE A TO B
```


Copyright © Micro Focus 2015-2016. All rights reserved.

```
ADD 8 TO B
.
```

All statements can end with a period or full stop, so it is acceptable to terminate in this way, but not recommended in "Modern" COBOL.

Commonly used verbs

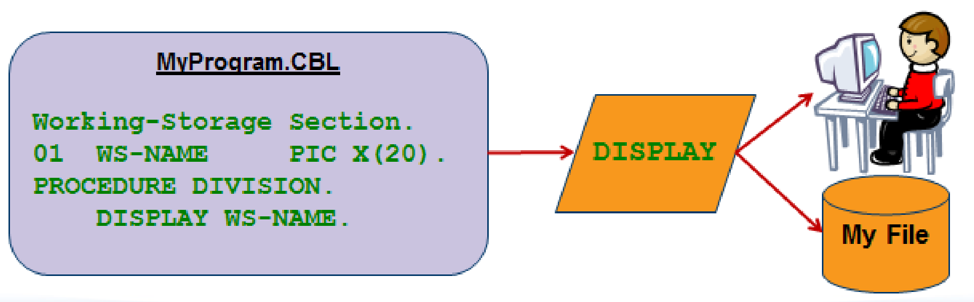
The following looks at the syntax and behavior of some commonly used verbs such as:

- DISPLAY
- ACCEPT
- MOVE
- PERFORM ...
- PERFORM ... UNTIL
- STOP

DISPLAY verb

DISPLAY can be used to output the value of any combination of data items and literal.

In the example below ("DISPLAY WS-NAME"), DISPLAY sends the contents of the field WS-NAME to the screen.

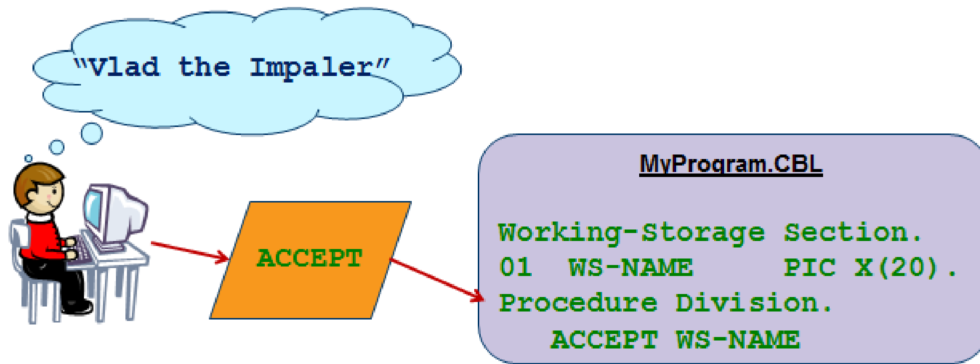


ACCEPT verb

ACCEPT is the opposite of DISPLAY. It is used to allow the person running the program to input a value into a data item.

In the example at below ("ACCEPT WS-NAME"), ACCEPT copies the value on the screen into the contents of the field named WS-NAME.

Copyright © Micro Focus 2015-2016. All rights reserved.



MOVE verb

Contrary to the English use of the word move, MOVE **copies** a value from a data item (or a literal or figurative constant) to one or more data items. The field which is the source of the 'MOVE' remains unchanged.

For example, if a program read a record and stored it, you could use MOVE to move the data from the input storage area to the output storage area. Here is a simple form of MOVE.

```
MOVE DATA-SEND-FIELD TO DATA-TARGET-FIELD
```

Then sending field could contain one of the following

- Numeric literal (e.g. 0 or 2004)
- Character literal (e.g. "Hello")
- Figurative constants (like the keywords SPACES or ZERO)
- Data item (for example, WS-NAME or WS-COUNTER)

Numeric literal MOVE

In the examples below the numeric literal zero is moved to field WS-TOTAL and 2004 is moved to the field WS-YEAR.

```
MOVE 0 TO WS-TOTAL
MOVE 2004 TO WS-YEAR
```

Character literal MOVE

In this example the character literal HELLO is moved to the field WELCOME-MESSAGE.

```
MOVE 'HELLO' TO WELCOME-MESSAGE
```

Figurative constant MOVE

These two examples use the figurative constants SPACES and ZERO.

```
MOVE SPACES TO WS-CHARACTER-FIELDS
MOVE ZERO   TO WS-TOTAL-2 WS-TOTAL-3
```

In the examples above spaces are moved to the field WS-CHARACTER_FIELDS, and zero is moved to two fields: WS-TOTAL-2 and WS-TOTAL-3.

Copyright © Micro Focus 2015-2016. All rights reserved.

Data item MOVE

Here are two examples showing how data items can be moved.

```
MOVE WS-NAME TO OUTPUT-NAME-1 OUTPUT-NAME-2
```

In this example the value of field WS-NAME is copied to two fields: OUTPUT-NAME-1 and OUTPUT-NAME-2.

```
MOVE WS-COUNTER to STORE-COUNTER
```

In this example the value of the field WS-COUNTER is copied to the field STORE-COUNTER.

MOVE verb logistics for character fields

MOVE works in a totally predictable way. MOVEing to a character field (whether from a character or numeric one) causes the target field to fill up from the left.

If the target field is longer, the remaining byte(s) on the right will be space-filled.

If the target field is too short, the original value will be truncated from the right. The following illustrates the how sending field will start filling up the target field starting from the ***left***.

Sending and Target Field Lengths – send longer than target

Consider the following example when the target field length is more restricted than the sending field length.

```
01  NAME-IN             PIC X(9) VALUE "ELIZABETH".
01  NAME-OUT            PIC X(3) VALUE "AMY".

    MOVE NAME-IN TO NAME-OUT
```

One would think that “ELIZABETH” becomes assigned to NAME-OUT; however, the PIC X(3) limits the target field to “ELI” only.

Sending and Target Field Lengths – send longer than target

Consider the following example when the target field length is longer than the sending field length.

```
01  NAME-IN             PIC X(9) VALUE "ELIZABETH".
01  NAME-OUT            PIC X(3) VALUE "AMY".

    MOVE NAME-OUT TO NAME-IN
```

After the MOVE, the NAME-IN field will contain "AMY   "

MOVE verb logistics for numeric fields

Basically MOVEing a numeric field to a numeric field, the move always aligns on the decimal place. So in some cases this will result in the truncation of the result, or the removal of the some of the decimal places. E.g.

```
01  NUM1  PIC 9(4)V99.
01  NUM2  PIC 99V99.
01  NUM3  PIC 9.
```


Copyright © Micro Focus 2015-2016. All rights reserved.

```
    01  NUM4  PIC V99.
    01  CHAR1 PIC XX.
    01  CHAR2 PIC X(10).

MOVE 1234.56 TO NUM1
MOVE NUM1 TO NUM2, NUM3, NUM4, CHAR1, CHAR2
```

These MOVEs will result in the data items containing the following values:

```
NUM1    1234.56
NUM2    34.56
NUM3    4
NUM4    .56
CHAR1   "12"
CHAR2   "123456    "
```

PERFORM Verb

Up until now the course has shown programs that execute in the order that the statements appear. However, you can control the order of execution sequence using the PERFORM verb. You can control execution using several formats of the PERFORM verb, including using it with an IF condition or using PERFORM UNTIL a condition occurs.

The following shows how to use PERFORM

```
PERFORM paragraph-header
```

PERFORM identifies the instruction to be done on paragraph-header, which is the name of a sub-process to be executed. The sub-process must also be written in your program. Here is an example.

In the first line of the procedure division in the following program, the verb PERFORM executes INIT-PARA, a sub-process that begins 3 lines further on. Upon completion of the INIT-PARA, the execution point returns to the statement after the PERFORM INIT-PARA

```
WORKING-STORAGE SECTION.
01  WS-TIMES          PIC 9(4).
01  WS-NUM1           PIC 9(4).
01  WS-NUM2           PIC 9(4).
01  WS-NUM3           PIC 9(6).
PROCEDURE DIVISION.
PROG.
    PERFORM INIT-PARA
    PERFORM LOOP-PARA WS-TIMES TIMES
    PERFORM END-PARA.
INIT-PARA.
    DISPLAY "A SIMPLE MULTIPLIER"
    DISPLAY "HOW MANY TIMES (1 TO 6)?"
    ACCEPT WS-TIMES
    IF (WS-TIMES > 6) OR (WS-TIMES < 1)
        MOVE 1 TO WS-TIMES
    END-IF.
LOOP-PARA.
    DISPLAY "FIRST NUMBER?"
    ACCEPT WS-NUM1
    DISPLAY "SECOND NUMBER?"
    ACCEPT WS-NUM2
    MULTIPLY WS-NUM1 BY WS-NUM2 GIVING WS-NUM3
```


Copyright © Micro Focus 2015-2016. All rights reserved.

```
    DISPLAY "PRODUCT OF " WS-NUM1 " AND " WS-NUM2
            " IS " WS-NUM3 .
END-PARA.
    DISPLAY "MULTIPLICATION DONE " WS-TIMES " TIME(S)"
    DISPLAY "THANK YOU FOR THAT"
    STOP RUN.
```

You can see this program in action if you change your workspace **05_02_Simple_Performs**

NOTE*: This program and later ones omit the IDENTIFICATION DIVISION and ENVIRONMENT DIVISION when they are not needed. You will also see that this program does not use section in procedure division. This is also fine.*

The PERFORM Verb and Processing Loops

The verb PERFORM executes the paragraph, INIT-PARA. Control passes to INIT-PARA and the program works through the INIT-PARA statements. The ACCEPT verb in INIT-PARA gets a value that the program needs. Here, ACCEPT gets the number of times the main program loop will occur.

The program also contains an IF statement, as shown here.

```
    IF (WS-TIMES > 6) OR (WS-TIMES < 1)
        MOVE 1 TO WS-TIMES
    END-IF.
```

This logic states that if a user enters 0 or a number greater than 6 ignore it and substitute 1. (A later module provides a more detailed discussion of the format of IF).

At the end of INIT-PARA we return to PROG, where the second statement appears: PERFORM LOOP-PARA WS-TIMES TIMES. This paragraph executes a particular number of times (at the end of which control returns to the PROG paragraph).

If we now look at the code in LOOP-PARA, we see two more ACCEPT statements, which give the program the numbers needed for the MULTIPLY statement. There is more to MULTIPLY than can be covered here, but we should be able to see that the statement calculates the product of two data items, places it in a third data item, and then DISPLAYs it.

Note: *Both the MULTIPLY and DIVIDE statements will be covered later. The Divide statement format is very similar to the MULTPLY; DIVIDE one data field by a second data field placing the result in a third data field.*

```
     DIVIDE  WS-NUM1 BY WS-NUM2 GIVING WS-NUM3
```

The whole paragraph will be PERFORMed as many times as WS-TIMES decrees.

Eventually, LOOP-PARA completes for the last time, and control passes to the last statement of the PROG paragraph, which is PERFORM END-PARA. This statement initiates END-PARA. The END-PARA paragraph contains two informational DISPLAYs and a STOP RUN. Note that there is no need to end the program in PROG, making the STOP RUN the last statement in END-PARA as perfectly acceptable.

The Verbs PERFORM … UNTIL and GO TO

Copyright © Micro Focus 2015-2016. All rights reserved.

COBOL offers two verbs, PERFORM and GO TO, to direct the logic of a program. For various reasons PERFORM is far more likely to be seen in modern COBOL. GO TO in COBOL, as in other languages, has acquired a bad reputation, because it has become associated with poor programming practice. In truth, it is possible to write well- or badly-designed programs with either verb, but PERFORM makes it easier to do the former.

You can use the PERFORM verb in several formats, including PERFORM…UNTIL. Use PERFORM … UNTIL according to the following syntax.

```
PERFORM PARAGRAPH-HEADER UNTIL [CONDITION]
```

We have looked at the simple use of PERFORM, which executes once. The PERFORM ... UNTIL executes one or many times until the condition is true.

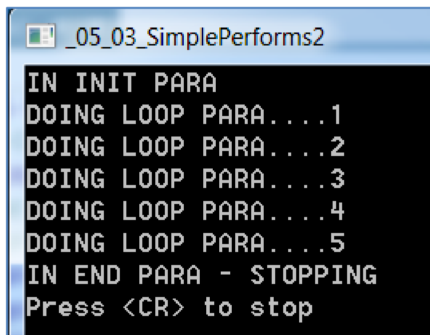
Here is an example of a program using PERFORM ... UNTIL. The output follows the program. In this case the PERFORMs are of section names rather than the paragraph names in the previous example. The choice is yours, but it is strongly recommended that you do not mix performs of paragraphs with performs of sections. This can lead to hard to maintain and error prone code.

***Note**: Use comments liberally to make the code more understandable. The compiler ignores any line with an asterisk (*) in column 7. Use this for comments.*

```
 WORKING-STORAGE SECTION.
 01  WS-COUNTER             PIC 9.
 PROCEDURE DIVISION.
 MAINLINE SECTION.
 START-UP.
     PERFORM INIT-SECT
     PERFORM LOOP-SECT UNTIL WS-COUNTER > 4
     PERFORM END-PARA
     STOP 'Press <CR> to stop'
     STOP RUN.
*------------------------------------------------------
 INIT-SECT SECTION.
 INIT-PARA.
     DISPLAY "IN INIT PARA"
     MOVE ZERO TO WS-COUNTER.
 INIT-EXIT.
     EXIT.
*------------------------------------------------------
 LOOP-SECT SECTION.
 LOOP-PARA.
     ADD 1 TO WS-COUNTER
     DISPLAY "DOING LOOP PARA...." WS-COUNTER.
 LOOP-EXIT.
     EXIT.
*------------------------------------------------------
 END-SECT SECTION.
 END-PARA.
     DISPLAY "IN END PARA - STOPPING".
 END-EXIT.
     EXIT.
```

The output is:

Copyright © Micro Focus 2015-2016. All rights reserved.



You can look at this program by changing your workspace to **05_03_Simple_Performs2**

This program illustrates the following interesting points.

PERFORM UNTIL is very commonly used, but the condition must become true at some point or the program could loop forever. In this case, if the ADD 1 TO WS-COUNTER were not present, this would happen.

```
PERFORM LOOP-SECT UNTIL WS-COUNTER > 4
```

By default, the UNTIL clause is tested before the loop. The loop happened five times, because “greater than 4” becomes true only after the fifth iteration. When the program comes round for the sixth time, “greater than 4” is now true

Another common mistake is for the test not to be met because it has been incorrectly specified. For example, the following code would generate a perpetual loop if the salary paid never *exactly* equalled 20,000

```
PERFORM LOOP-PARA UNTIL SALARY-PAID = 2000
```

A better test would probably be greater than some figure.

```
PERFORM LOOP-PARA UNTIL SALARY-PAID >= 2000
```

There is a different format of the verb,

```
PERFORM <paragraph or section> WITH TEST AFTER UNTIL <condition>.
```

This means that the <paragraph or section> will always be PERFORMed at least once. The default version of PERFORM with the test before is very useful, as the loop will stop if the condition is never true. For example, if you are reading records from a file, you might write pseudo-code such as “perform process-record until there are no more records on the file.” The WITH TEST AFTER version of the verb would assume that at least one record existed. So, if there were no records on the file, a logic error would occur.

PERFORMing Paragraphs

Performing a single Paragraph is very efficient

Where a paragraph is a logical group of statements that perform a specific function

The paragraph is referenced by its heading name

Copyright © Micro Focus 2015-2016. All rights reserved.

When a series of paragraphs are to be executed then use the PERFORM...THRU... option

```
PERFORM B-INITIAL.    [Single]

PERFORM B-INITIAL THRU D-END.          [Series]
```

This Perform THRU option is not something which is used commonly in modern COBOL applications but you will find extensive use of it in older legacy programs.

PERFORMing Sections

Performing a single section is also very efficient.

All the paragraphs within the section will be performed in sequence. E.g.

```
MY-MAIN SECTION.
PARA-1.
      MOVE A TO B
      . . . .
PARA-2.
      . . . .
PARA-3.
      . . . .
```

The statement:

```
PERFORM MY-MAIN
```

Will execute PARA-1, PARA-2, PARA-3 in sequence.

You would get the same result with:

```
PERFORM PARA-1 THRU PARA-3
```

Using Comments

It has already been said that you should use comments as much as possible in a COBOL program to help with subsequent developers understanding the code.

A comment line is normally recognized by column 7 having an asterisk.

However, since 1985 you can also use what is known as the “in-line” comment.

In this case if you place the characters *> anywhere in the code, then the rest of the line, after *> is regarded as a comment. E.g.

```
MAINLINE SECTION.
START-UP.
    PERFORM INIT-SECT        *> This initializes the program
    PERFORM LOOP-SECT        *> Do this 4 times
        UNTIL WS-COUNTER > 4
    PERFORM END-PARA         *> This terminates the program
    STOP 'Press <CR> to stop'
```


Copyright © Micro Focus 2015-2016. All rights reserved.

STOP Verb

STOP is not issued on its own, it is most commonly written with RUN to terminate a program.

`STOP RUN` stops the current program

Any statements following STOP RUN in the same paragraph can never be executed

As you have seen in the preceding examples it is also used together with a literal. E.g.

```
STOP "Press <CR> to continue"
```

This causes a pause in the program, while the user presses the **<CR>** key

COBOL program execution

The procedure division of a COBOL program is executed in sequence from **first** line to **last** line. However the program execution can branch from place to place with the use of GO TO, IF, PERFORM and other conditional statements

Let us first look at a program with no branching

e.g.

```
PROCEDURE DIVISION.
PROG.
    DISPLAY "IN PROG".
PARA-1.
    DISPLAY "IN PARA 1".
PARA-2.
    DISPLAY "IN PARA 2".
PARA-3.
    DISPLAY "IN PARA 3".
PARA-4.
    DISPLAY "IN PARA 4".
PARA-5.
    DISPLAY "IN PARA 5".
```

In the above case the program will “drop through” the collection of paragraphs. The paragraph names have no meaning (other than documentation).

The results of executing this code would be the following values are displayed on the screen:

```
IN PROG
IN PARA 1
IN PARA 2
IN PARA 3
IN PARA 4
IN PARA 5
```

A similar scenario would happen with sections. e.g.

```
PROCEDURE DIVISION.
SECT-1 SECTION.
PROG.
    DISPLAY "IN PROG".
PARA-1.
```


Copyright © Micro Focus 2015-2016. All rights reserved.

```
        DISPLAY "IN PARA 1".
    PARA-2.
        DISPLAY "IN PARA 2".
    SECT-2 SECTION.
    PARA-3.
        DISPLAY "IN PARA 3".
    PARA-4.
        DISPLAY "IN PARA 4".
    PARA-5.
        DISPLAY "IN PARA 5".
```

Just as in the case of the paragraph only example above, the program will “drop through” the collection of sections and paragraphs. The section names and paragraph names have no meaning (other than documentation).

The results of executing this code would be the following values are displayed on the screen:

```
IN PROG
IN PARA 1
IN PARA 2
IN PARA 3
IN PARA 4
IN PARA 5
```

If we change the first example of paragraphs only to include a GO TO:

```
    PROCEDURE DIVISION.
    PROG.
        DISPLAY "IN PROG"
        GO TO PARA-3.
    PARA-1.
        DISPLAY "IN PARA 1".
    PARA-2.
        DISPLAY "IN PARA 2".
    PARA-3.
        DISPLAY "IN PARA 3".
    PARA-4.
        DISPLAY "IN PARA 4".
    PARA-5.
        DISPLAY "IN PARA 5".
```

The results of executing this code would be the following values are displayed on the screen:

```
IN PROG
IN PARA-3
IN PARA-4
IN PARA-5
```

If we change the first example of paragraphs only to include a PERFORM:

```
    PROCEDURE DIVISION.
    PROG.
        DISPLAY "IN PROG"
        PERFORM PARA-3.
    PARA-1.
        DISPLAY "IN PARA 1".
    PARA-2.
```


Copyright © Micro Focus 2015-2016. All rights reserved.

```
        DISPLAY "IN PARA 2".
    PARA-3.
        DISPLAY "IN PARA 3".
    PARA-4.
        DISPLAY "IN PARA 4".
    PARA-5.
        DISPLAY "IN PARA 5".
```

The results of executing this code would be the following values are displayed on the screen:

```
IN PROG
IN PARA-3
IN PARA-1
IN PARA-2
IN PARA-3
IN PARA-4
IN PARA-5
```

If we change the second example of sections to include a GO TO:

```
    PROCEDURE DIVISION.
    SECT-1 SECTION.
    PROG.
        DISPLAY "IN PROG"
        GO TO SECT-2.
    PARA-1.
        DISPLAY "IN PARA 1".
    PARA-2.
        DISPLAY "IN PARA 2".
    SECT-2 SECTION.
    PARA-3.
        DISPLAY "IN PARA 3".
    PARA-4.
        DISPLAY "IN PARA 4".
    PARA-5.
        DISPLAY "IN PARA 5".
```

The results of executing this code would be the following values are displayed on the screen:

```
IN PROG
IN PARA-3
IN PARA-4
IN PARA-5
```

If we change the second example of sections to include a PERFORM:

```
    PROCEDURE DIVISION.
    SECT-1 SECTION.
    PROG.
        DISPLAY "IN PROG"
        PERFORM TO SECT-2.
    PARA-1.
        DISPLAY "IN PARA 1".
    PARA-2.
        DISPLAY "IN PARA 2".
    SECT-2 SECTION.
    PARA-3.
        DISPLAY "IN PARA 3".
```


Copyright © Micro Focus 2015-2016. All rights reserved.

```
    PARA-4.
        DISPLAY "IN PARA 4".
    PARA-5.
        DISPLAY "IN PARA 5".
```

The results of executing this code would be the following values are displayed on the screen:

```
IN PROG
IN PARA-3
IN PARA-4
IN PARA-5
IN PARA-1
IN PARA-2
IN PARA-3
IN PARA-4
IN PARA-5
```

Mixing GO TO and PERFORM

Avoid if at all possible!!

It is very easy to get into a complete mess if you mix and match **Performs** and **Go Tos**. Please avoid if at all possible.

The example shown below (and in workspace **05_04_Mix_GOTO_Perform**) has a mixture which causes a number of problems.

```
    DATA DIVISION.
    WORKING-STORAGE SECTION.
    PROCEDURE DIVISION.
    PROG.
        PERFORM PARA-1 THRU PARA-3
        STOP "Press <CR> to terminate"
        STOP RUN.
    PARA-1.
        DISPLAY "IN PARA 1"
        GO TO PARA-4.
    PARA-2.
        DISPLAY "IN PARA 2".
    PARA-3.
        DISPLAY "IN PARA 3".
    PARA-4.
        DISPLAY "IN PARA 4".
    PARA-5.
        DISPLAY "IN PARA 5"
        GO TO PARA-3.
```

Even in this small piece of code, it has become difficult to follow the logic.

Although the PERFORM results in control jumping to PARA-1, the GO TO then takes us to PARA-4, and then we drop through to PARA-5.

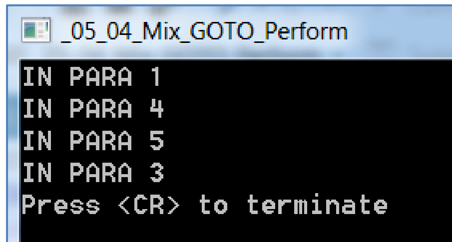
In fact, everything works out all right, as the GO TO PARA-3 takes us back to the last paragraph to be PERFORMed, but this was pure luck, overcoming terrible coding.

Control can therefore return to the STOP RUN.

Copyright © Micro Focus 2015-2016. All rights reserved.

If that GO TO pointed to, say a PARA-8 or there was no GO TO, the results would be unpredictable, to say the least.

Incidentally, this program produces the following output.



Module Summary

Now you have completed this module you are be able to:

- Explain the DISPLAY, ACCEPT, STOP, MOVE, GO TO and PERFORM verbs
- Describe how the following verbs direct program logic
    - PERFORM ...
    - PERFORM ... UNTIL
    - GO TO

Exercises

Make sure you execute the example solutions in debug mode (use **F5** to step through the execution of the code) to understand what is happening in the 4 programs inside these workspaces.

- **05_01_Basic_Verbs**
- **05_02_Simple_Performs**
- **05_03_Simple_Performs**
- **05_04_Mix_GOTO_Perform**

Quick Quiz

1. Which of these statements is always true?
    a. You can move any data type to any data type
    b. You can move character data of any size to character data of any size
    c. You can move character data of any size to numeric data of any size
    d. You can move numeric data of any size to numeric data of any size
    e. You can move numeric data of any size to character data of any size
2. True or false?
    a. It is compulsory to use END-IF to terminate an IF statement
    b. It is compulsory to use a period to terminate an IF statement
    c. It is compulsory to terminate an IF statement
3. True or False?
    a. The DISPLAY verb reads information from the screen
    b. The ACCEPT verb reads information from the screen
4. If the data items WS-COUNTER contains 14 and STORE-COUNTER contains 25, when you execute the statement:

Copyright © Micro Focus 2015-2016. All rights reserved.

```
MOVE WS-COUNTER TO STORE-COUNTER
```

the values of the two data items become?

a. WS-COUNTER 25, STORE-COUNTER 14
b. WS-COUNTER 25, STORE-COUNTER 25
c. WS-COUNTER 14, STORE-COUNTER 14
d. WS-COUNTER 0, STORE-COUNTER 14
e. WS-COUNTER 14, STORE-COUNTER 0

Copyright © Micro Focus 2015-2016. All rights reserved.

06 Best Practice

Introduction

Over the life of a typical COBOL application, many different people might make fixes and improvements at different times.

Often this maintenance work may take many times more than the initial effort needed to create the application in the first place.

Implementing best practices when creating new programs speeds up the process of getting a stable, high-quality application; yet also yields huge benefits during the maintenance phase.

We will look at ways in which a COBOL program can be written to support this long term objective of ease of maintenance.

Module Objectives

Upon successful completion of this module, you will be able to:

- Write meaningful comments when writing code.
- Design a typical COBOL file-handling program.
- Design well-structured programs.

Designing a COBOL Program

Does a COBOL program need to be designed?

- Of course it does, but it is surprising how many programmers simply embark upon coding without thought to overall design.

Could the code be written immediately with no thought being given to program structure?

- Yes it can, but what about long term issues?

It is always possible to write a program in this way, but in all but the simplest programs, it is rarely sensible.

- Most programs need to make decisions at some point and these need to be taken care of sensibly.
- Many programs will contain procedural logic, meaning, certain things must take place before other actions can be undertaken.
- Programs may have to execute a loop a particular number of times or until some condition is true, as we saw in the discussion of the PERFORM...IF or PERFORM...UNTIL verb.

If a program has to do any of these things, then bad design or not designing at all, will introduce logic errors. Logic errors cause the code to behave differently than the programmer expects.

Copyright © Micro Focus 2015-2016. All rights reserved.

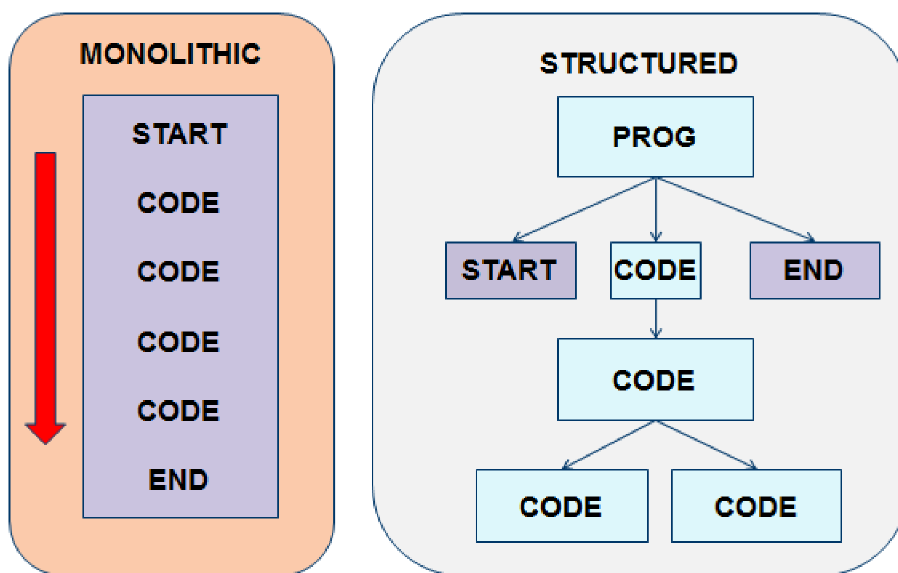
The COBOL compiler cannot find such logic errors, and they may remain undetected until the program goes live, surfacing in the future. A well-designed program very much reduces the number of logic errors. Spending time on the design avoids many problems later.

Usually the later an error is found, the more expensive it is to fix.

COBOL programs tend to be very long-lived, perhaps for twenty or thirty years. Over that time they may undergo many revisions, from the addition of minor features to almost a total rewrite.

A well-designed program from the start makes modifications considerably easier. If the modifications are in turn well designed, then the task is simplified for the next maintenance programmer, and so on.

Structured design over monolithic design gives us many benefits:



There are many methods employed for program design.

Two examples are:

- Monolithic programming
- Structured (Hierarchical) programming

Monolithic programming

- Entails the program logically starting at the beginning and flowing through to the end as though reading a novel.

Structured programming involves

- The analysis of the overall objective
- The division of the objective into functional sections
- The definition of each functional section
- The arrangement of the functions into logical steps

Copyright © Micro Focus 2015-2016. All rights reserved.

- Writing the designed steps

Object Oriented Design

We will not see this until later in the course, but Object design, by its very nature, encourages the use of good design techniques.

Design tools

The program designer has various tools to aid in the definition of the application solution

These include:

- Structure Diagrams
- Truth tables

Structure Diagrams aid the programmer in

- Isolating particular functions
- Simplifying the writing of the code
- Simplifying the test procedure

Truth Tables

- Highlight decision options prior to coding
- Simplify the data routes through the program
- Identify potential logic error traps
- Isolate the data item test options

Structure Diagrams

We will briefly indicate the use of structure diagrams in our program design.

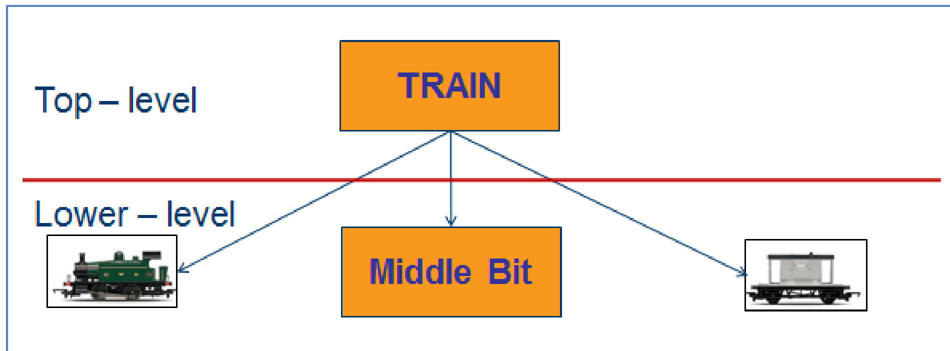
A logic block contains either the overall structure, or parts of the structure.

In our example we will use a goods train to view the structures possible.

A top-level block

- will have a meaningful name
- it will have further functional blocks beneath it
- two or more blocks at the same level are a **sequence**

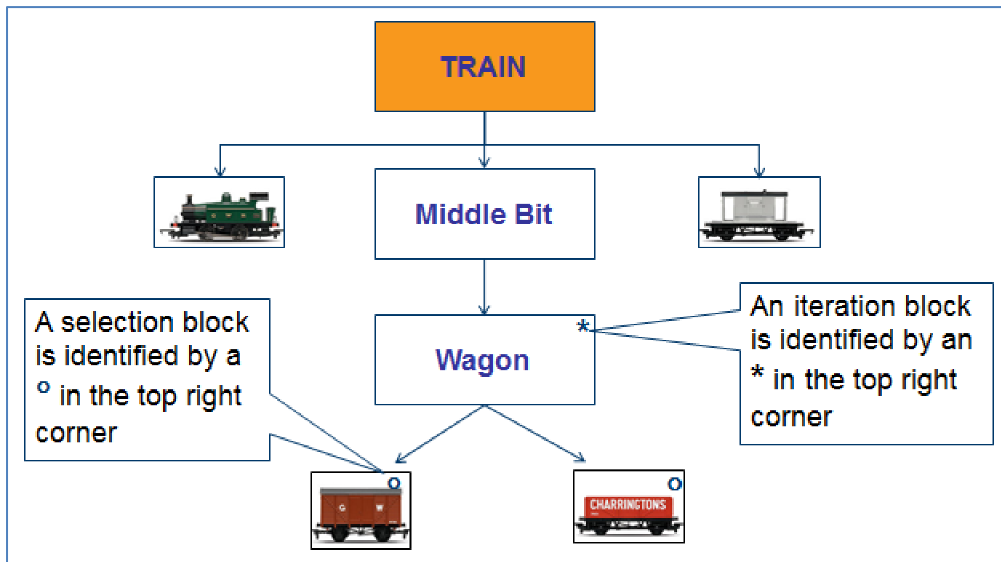
Copyright © Micro Focus 2015-2016. All rights reserved.



The train is now a sequence of blocks

- A locomotive
- A 'bit in the middle'
- A Guard Van

This can now be further refined as iterations and selections:



Example

As a more meaningful example you are asked to design a program that sequentially reads a file of records and then operates using the following rules:

- The input file can contain any number of records
- There are two types of input records, type A and type B
- Type A record, write an output record based on this record
- Type B record, increment a counter
- Display the total of type B records at the end of the program

Once the design phase is complete the coding can begin

If the structured design is adhered to then

- Each of the structure blocks will be either a Paragraph or a Section

Copyright © Micro Focus 2015-2016. All rights reserved.

- Each Section/Paragraph will be stand-alone
- Each Section/Paragraph can be tested separately

Each Section/Paragraph could contain a descriptive comment that instructs the next developer about the section function.

Sample Code

SECTION Header

COMMENTS – what the section will do – perhaps?

```
 PROCESS-RECORD SECTION.
*
*      THIS SECTION SELECTS THE RECORD TYPE TO PROCESS
*      RECORD TYPES A AND B ONLY - CHANGE HERE IF NEW
TYPES
*
 PR-START.
       IF [it's a type A record]
              PERFORM TYPE-A-ACTIONS
       ELSE [we're assuming here only type A or B]
              PERFORM TYPE-B-ACTIONS
       END-IF.
 PR-END SECTION
       EXIT.
```

STAND-ALONE - this section calls other sections but does not branch out.

During the design of the program you took into consideration various factors

The IF statement only tests for record type A or B

- What would happen if another record type was introduced?
- How could we get around this problem?

Read a record results

- You get a data record
- What if there was no record, or, you had reached the end of file
- How would you inform the program of this event?

Handling the Counter

- Incrementing was requested for the type B record
- Initialization of the counter may be required

Copyright © Micro Focus 2015-2016. All rights reserved.

Truth tables

- What are truth tables?
- How do they work?
- Do they really help us?
- Complete the exercise below:

> 1) There are 3 numbered statements in this box.
> 2) Two of the statements are false.
> 3) You have an aptitude for solving logic problems.

Consider the information in the box above
A) Is statement 3 true or false?
B) Explain why.

Truth Tables – Your Answer?

Question	A	B	C	D	E	F	G	H
1	Y	Y	Y	Y	N	N	N	N
2	Y	Y	N	N	Y	Y	N	N
3	Y	N	Y	N	Y	N	Y	N

In the table above, all possible truth combinations are shown
1) is true, therefore all "N" answers are false (red)
2) is false, since for it to be true 1) must be false (blue)
3) is true, since if false it would make 2) true (green)

Therefore it is proven that column C is the correct answer
- "You have an aptitude for solving logic problems."

We will see how we can use truth tables within COBOL later, using the **EVALUATE** statement.

Now back to our COBOL program to read the data file and produce the results we require!

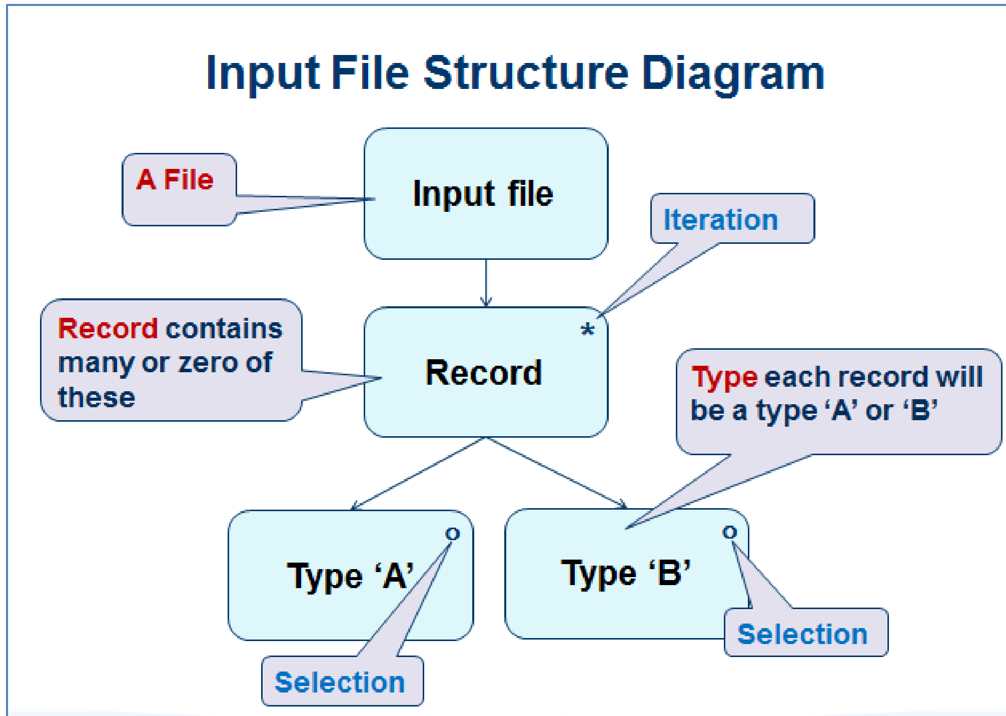
Input and Output files

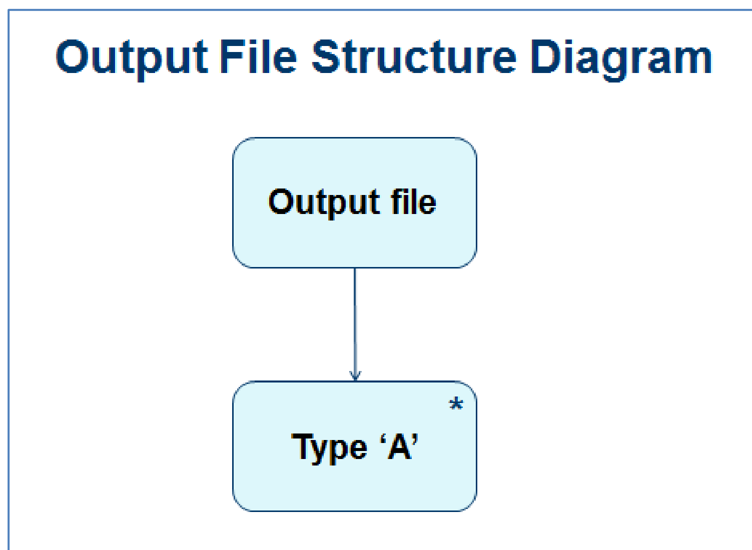
Considerations when handling files in our examples

- The input file can contain any number of records, including zero
- There are two types of input records, type A and type B. We will assume that there is a PIC X field on the record containing the vale "A" or the value "B".

Copyright © Micro Focus 2015-2016. All rights reserved.

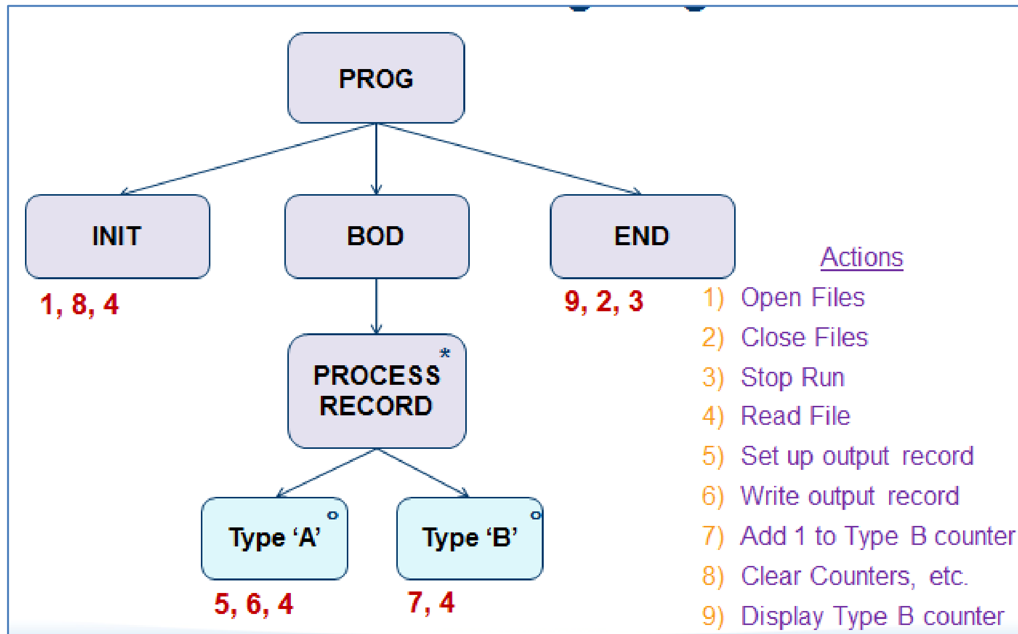
- If we find a type “A” record, create an output record based on that record, and write it out. (No details of record yet)
- If we find a type “B” record, just increment a type B counter, which will be displayed at the end of the program





The structure of the program to handle this processing is:

Copyright © Micro Focus 2015-2016. All rights reserved.



So the following is a sample program to implement this logic:

```
PROCEDURE DIVISION.
PROG.
    PERFORM INIT-PARA
    PERFORM BOD-PARA
    PERFORM END-PARA
INIT-PARA.
    DISPLAY "START OF PROGRAM".
    [open files]
    [read file, if no record,
        move 1 to flag]
BODY-PARA.
    PERFORM UNTIL FLAG = 1
        PERFORM PROCESS-REC
    END-PERFORM.
END-PARA.
    DISPLAY WS-TYPE-B-COUNTER
    [close files].
```

```
PROCESS-REC .
    IF [it's a type A record]
        PERFORM TYPE-A-ACTIONS
    ELSE
        PERFORM TYPE-B-ACTIONS
    END-IF.
TYPE-A-ACTIONS.
    [set up output record]
    [write output record ]
    [read file, if no record,
        move 1 to flag].
TYPE-B-ACTIONS.
    ADD 1 TO WS-TYPE-B-COUNTER.
    [read file, if no record,
        move 1 to flag].
```

In the example, there are a couple of things to note.

Firstly, the IF statement this time has an ELSE. The actions here will be carried out if the condition is false (in other words, any record which isn't a type A will automatically be treated as a type B). This may well be acceptable. Alternatively you may need to check for A, B, and 'anything else'.

Secondly, on this occasion the IF statement has been terminated with an END-IF. Normally this means that no period (full stop) is needed at the end of the IF. But, because this is the last statement in the paragraph, the period is necessary.

It is beyond the scope of this class to fully cover structured program design (as it will also be later in the course when we look at Object Oriented COBOL). The main objectives here are to show how

Copyright © Micro Focus 2015-2016. All rights reserved.

COBOL provides building blocks supporting structured design (and later to support Object Orientation).

Module Summary

Now you have completed this module, you will be able to:

- Write meaningful comments when writing code
- Design a typical COBOL file-handling program
- Design well-structured programs

Exercise 1

In relation to the program shown above, look at the following:

- **PROG** – Perform INIT. Perform BOD. Perform END.
- **INIT** – Includes three actions:

  Open the files **[1].** (We will take it for granted at the moment that we have to do this. It's the way of connecting the program to the files it needs.)

  Clear counters **[8].** (We could do this with VALUE clauses, but it's a useful reminder.)

  Read the input file **[4]** looking for a record. To anticipate, in COBOL you can check whether a READ failed because of the end-of-file indication. Then, set a flag automatically (either a PIC 9 set to 1 when end-of-file or PIC X set to 'Y' — or anything else you want). It's worth remembering that this flag will need to be cleared as one of the counters. Again, we can do this explicitly, or we can use a VALUE clause.
- **BOD** – Perform PROCESS-REC until the flag is true. If the file was empty, then the flag is already true and the PERFORM won't happen at all.
- **END** – Display the type B counter **[9]**, which must be getting updated elsewhere, close the files **[2]** (again, let's assume that this is a good thing to do), and stop the run **[3].**
- **PROCESS-REC** – If the record we have at the moment is a Type A, Perform Type-A; else Perform Type-B.
- **TYPE-A** – Set up the output record (presumably by copying values from the input record) **[5]**, and write an output record **[6]**. Try and read another record **[4].** This read will either succeed (in which case we go round the loop again) or fail because of the end-of-file (in which case we have finished the loop).
- **TYPE-B** – Add 1 to the type B counter **[7]**, and try and read another record **[4]**, as above. Again the same logic applies. If we find a record, we go round again; if not, we leave the PERFORM.

Exercise 2

Create a structure diagram for each example below.

1. A dinner party with a number of guests, each of whom is either male or female.
2. A dinner party, where only couples are invited.
3. The meal at the dinner party. The starter is quail's egg salad, langoustines or vegetable terrine. Next, include a main course of steak (well-done, medium rare, or rare) with

Copyright © Micro Focus 2015-2016. All rights reserved.

vegetables (and either French fries or salad) or spinach and aubergine bake. Guests may drink one or more glasses of wine with this portion of the meal. Peaches in Armagnac or lime and durian ice cream make the final course. (Both the first and third courses are optional).
Finally, everyone can have one or more cups of coffee (black or with cream) or a liqueur.

4. The train (now passenger, not goods) with first- and standard-class carriages, a buffet car, and an extra locomotive at the rear.
5. A holiday that consists of a flight at the beginning and a return flight at the end. Between those two events are days where the holidaymaker goes to the beach, takes a coach trip, or goes shopping. Allow for the luggage missing on the flight out.
6. An encyclopedia that consists of many volumes, each containing many articles. Each volume has a table of contents and an index. A supplementary volume also has a table of contents and index of all the volumes together with an atlas.
7. A person's life, defined in different ways. Firstly, draw a structure showing the person going to school (more than one, possibly) and college, getting a job, and having a working life, followed by retirement. After you have done this, produce a totally different structure, showing the person getting married, having children, and grandchildren.
8. Draw a structure of what you do in a typical week, showing events such as meals, working, watching television (or whatever you do instead), hobbies and so on. You will probably have to show workdays separately from weekends.

A COBOL program with a start, middle and end. The start and end don't have to show any detail at the moment, but in the middle we have to show that a number of records (possibly zero) will be processed. Each record is of type A, B or C.

Exercise 3

Set your workspace to **06_01_Program_Design** and look at the logic of the code in the program.

This program is incomplete since the full program code has not yet been "fleshed" out.

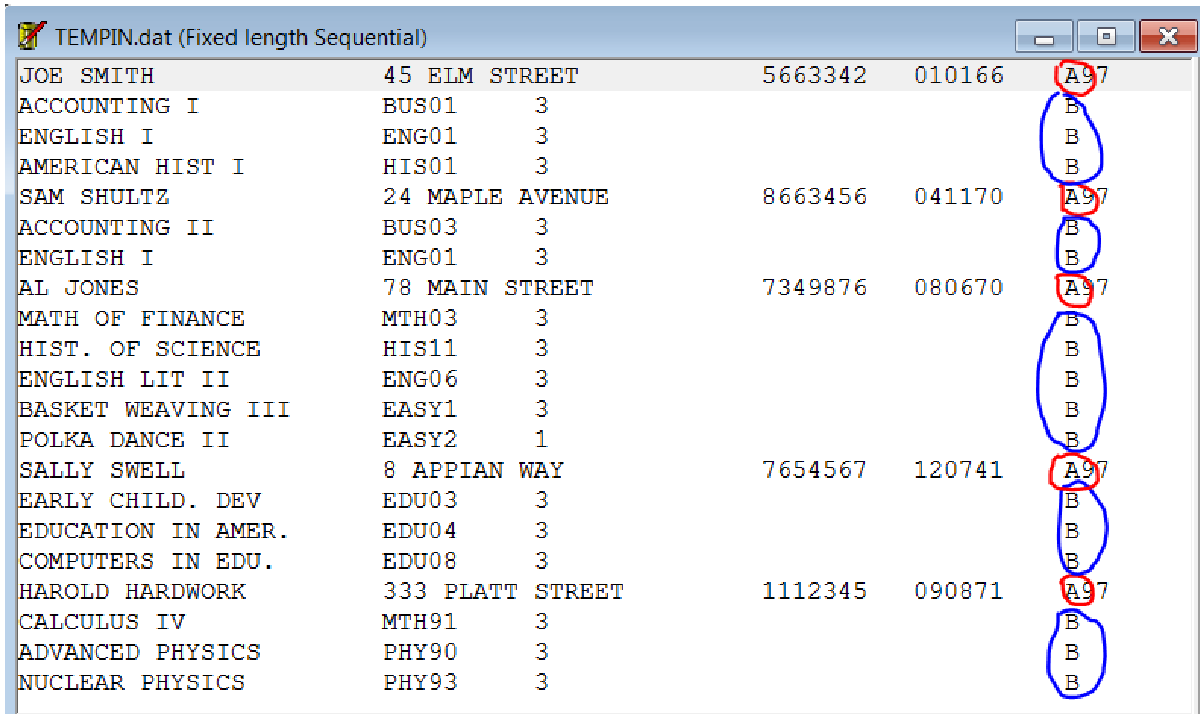
A fully "fleshed" out solution can be found in the workspace **06_02_Program_Design_full.** Inside this solution you will find code that we have not yet fully explored, but you may want to look through this code for now. Full details of what the code is doing will be discussed later.

The best was to look at this code is probably to debug the code using **F5**.

The only part of file handling you need to understand at this time is that when you "READ" a sequential file is that you read the next record on the file.

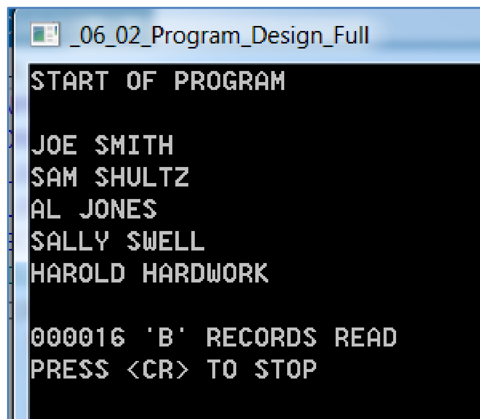
The contents of this file are:

Copyright © Micro Focus 2015-2016. All rights reserved.



```
TEMPIN.dat (Fixed length Sequential)
JOE SMITH              45 ELM STREET        5663342   010166   A97
ACCOUNTING I           BUS01     3                             B
ENGLISH I              ENG01     3                             B
AMERICAN HIST I        HIS01     3                             B
SAM SHULTZ             24 MAPLE AVENUE      8663456   041170   A97
ACCOUNTING II          BUS03     3                             B
ENGLISH I              ENG01     3                             B
AL JONES               78 MAIN STREET       7349876   080670   A97
MATH OF FINANCE        MTH03     3                             B
HIST. OF SCIENCE       HIS11     3                             B
ENGLISH LIT II         ENG06     3                             B
BASKET WEAVING III     EASY1     3                             B
POLKA DANCE II         EASY2     1                             B
SALLY SWELL            8 APPIAN WAY         7654567   120741   A97
EARLY CHILD. DEV       EDU03     3                             B
EDUCATION IN AMER.     EDU04     3                             B
COMPUTERS IN EDU.      EDU08     3                             B
HAROLD HARDWORK        333 PLATT STREET     1112345   090871   A97
CALCULUS IV            MTH91     3                             B
ADVANCED PHYSICS       PHY90     3                             B
NUCLEAR PHYSICS        PHY93     3                             B
```

The result of running this program will be:



```
_06_02_Program_Design_Full
START OF PROGRAM

JOE SMITH
SAM SHULTZ
AL JONES
SALLY SWELL
HAROLD HARDWORK

000016 'B' RECORDS READ
PRESS <CR> TO STOP
```

Quick Quiz

1. When designing a program the first task we undertake is:
    a. Make a list of actions the program must perform
    b. Start writing the code
    c. Think about the comments you can use
    d. Design the structure of the program
2. If a file is input to a program it must contain at least 1 record?
    a. TRUE
    b. FALSE
3. A file which is output from a program must already exist?

Copyright © Micro Focus 2015-2016. All rights reserved.

a. TRUE

b. FALSE

Copyright © Micro Focus 2015-2016. All rights reserved.

07 Handling Sequential Data Files

Introduction

At the end of the previous module you had a glimpse of how sequential files are handled in COBOL.

The ability to easily and flexibly handle data files is central to commercial computing needs and, not coincidentally, to COBOL.

The language regards files as either sequential or random access. Here, we look at the constructs and techniques for handling sequential files effectively.

Module Objectives

Upon successful completion of this module, you will:

- Use the Environment Division and Data Division entries in a COBOL program that uses input or output sequential files.
- Use the correct format of READ, WRITE, OPEN and CLOSE statements in the Procedure division for sequential files, including testing for an end-of-file condition.
- Describe how COBOL deals with multiple record types on the same file.
- Be able to design and code a typical COBOL program that handles sequential files.

Files and Records

In the earlier modules, we have written only COBOL programs that do not use files (although we briefly glimpsed some file access).

We shall now examine the extra entries necessary to use *sequential* files in a COBOL program.

When a program uses sequential files, the program must include the following entries and statements.

- **Environment Division entries** – to identify a file to your program.
- **Data Division entries** – to specify the file's layout for your program so that your program can read or write to it accurately.
- **Procedure Division statements** – to act on the file.

What are files and records?

- A file is a group of records
- A record is a group of fields, or data items. A record must be written as a Group Level item, which contains Elementary Level data items
- A record is also the unit of information that is read or written. (You do **not** read or write **fields**; you read or write **records**).

Files that COBOL can handle include:

- Sequential.

Copyright © Micro Focus 2015-2016. All rights reserved.

- Relative
- Indexed

In addition, on a PC or Unix machine, COBOL can also handle line-sequential files (which are workstation text files).

This module only deals with Sequential files

- A sequential file is a file containing records that must be retrieved in the order first to last in sequence.

We will take the program from the previous module and examine it in more detail.

We shall also examine the way that COBOL deals with multiple record types, within an input or output file.

Program statements required

What program entries and statements are required?

Environment Division entries

This notifies the program which files are to be accessed. This entry is used to map the internal program name of the file to the external file name on disk.

Data Division entries

This defines the data layout for each of the records in the files.

To enable accurate reading, writing and formatting of file data.

Procedure Division

This contains the statements for the actions to be performed on the files and its records.

Connecting Files

Before opening a file in your program, you must first establish its identity in your program and specify its layout so your program can access it appropriately. This is done in two steps and two places within a program.

Identify a file within the **ENVIRONMENT DIVISION** using a **SELECT** statement.

Specify a file's layout within the **DATA DIVISION**.

Identifying Files

The ENVIRONMENT DIVISION has not been used up until now. Now, we must identify the file to our program using a SELECT statement.

The SELECT statement must be included in the INPUT-OUTPUT SECTION of the ENVIRONMENT DIVISION.

The INPUT-OUTPUT SECTION contains an entry for each file.

The following code shows an entry for two files:

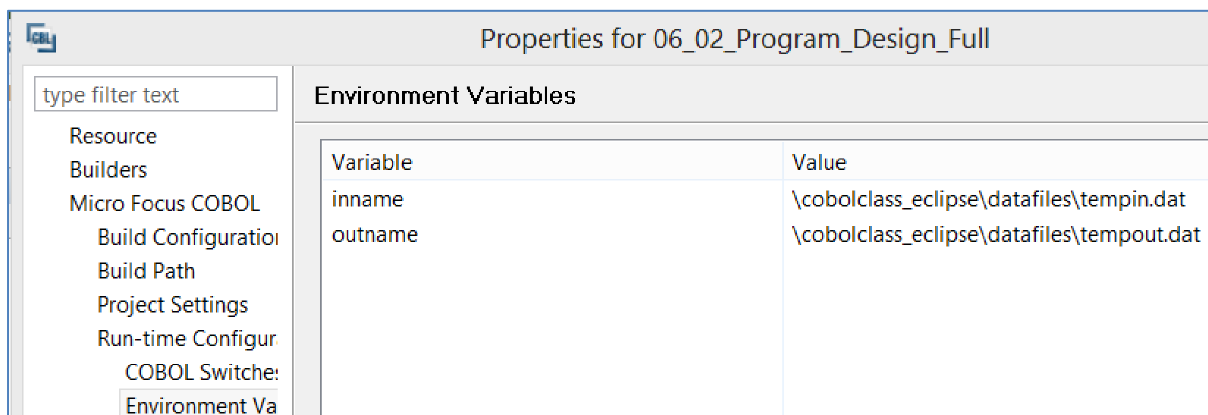
Copyright © Micro Focus 2015-2016. All rights reserved.

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INFILE  ASSIGN INNAME
        ORGANIZATION SEQUENTIAL.
    SELECT OUTFILE ASSIGN OUTNAME
        ORGANIZATION SEQUENTIAL.
```

This is the minimum entry for a file. However the phrase `ORGANIZATION SEQUENTIAL` can be omitted, since this is the default file organization. This would give:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INFILE  ASSIGN INNAME.
    SELECT OUTFILE ASSIGN OUTNAME.
```

The file is called **`INNAME`** in the outside world; however, your COBOL program knows it as **`INFILE`** within the program. The mapping between the internal file names and the external file names can be seen inside the project properties as shown below. (Switch to the workspace **06_02_Program_Design_Full** to see this).



This SELECT statement assigns the INNAME file with an internal file on your system.

While the Division and Section entries begin in Area A, the file entries should start in Area B.

An alternative method for assigning filenames, directly in the program, is shown below:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INFILE  ASSIGN 'C:\COBOLClass_Eclipse\DataFiles\TEMPIN.DAT'.
    SELECT OUTFILE ASSIGN 'C:\COBOLClass_Eclipse\DataFiles\TEMPOUT.DAT'.
```

Each file needs its own SELECT statement.

Each of them should be terminated with a period.

Any *input* file **must exist,** while an *output* file **might not exist**. If it does, by default the program *overwrites* it and loses existing records.

Although we have now told the program that we are going to access one or more files, the program needs detail of the data on the records within the file(s).

Copyright © Micro Focus 2015-2016. All rights reserved.

This information appears in the DATA DIVISION, inside the FILE SECTION.

Defining file layouts

We need to define each file to the program.

To do so we must provide entries in the FILE SECTION.

The definition includes the file layout, its attributes and the record formats.

- This sample shows two files – INFILE and OUTFILE
- Each with a single record layout

```
 FILE SECTION.
*     The input file
 FD   INFILE.                   [File Description]
 01   INREC.                    [Record Description]
      03   IN-REC-TYPE         PIC X.
      03   IN-NAME             PIC X(20).
      03   IN-DOB              PIC 9(8).
*     The output file
 FD   OUTFILE.
 01   OUTREC.
      03   OUT-NAME            PIC X(20).
      03   OUT-DOB             PIC 9(8).
```

The File Description FD entry

For every file name named in the SELECT statement within the INPUT-OUTPUT SECTION of the ENVIRONMENT DIVISION, you must include an **FD** (File Description) entry.

The FD statement marks the beginning of the data description for that file.

The name following the FD must match that after the appropriate SELECT statement in the INPUT-OUTPUT SECTION.

Because two SELECT statements were included for two file names in the previous code, the program must include two FD entries.

Code your **FD** entries in Area A.

File Record structure

Each record begins with a Group Level Item which names the record

- Our sample shows **01 INREC** and **01 OUTREC**
  - **INREC** is our input record
  - **OUTREC** is our output record
- We only have one record layout defined for each file
- Field content
  - The contents of the fields are loaded from the file.

Copyright © Micro Focus 2015-2016. All rights reserved.

- Therefore VALUE clauses cannot be used to pre-set the data items in the File Section.

COBOL verbs for sequential file access

The COBOL verbs to access a sequential file are

- **OPEN**
- **READ**
- **WRITE**
- **CLOSE**

OPEN and **CLOSE** are always used before and after file access

READ and **WRITE** statements are program dependent.

- A file cannot be **read** unless it has been opened for **input**.
- A file cannot be **written** unless it has been opened for either **output** or **extend**

The OPEN verb

Before the program can access a file, it needs to be OPENed.

Input files are OPENed INPUT, and output files are normally OPENed OUTPUT.

Use the OPEN verb in the following format:

```
OPEN [Type of Access] Filename.
```

e.g.

```
OPEN INPUT  INFILE
OPEN OUTPUT OUTFILE
```

It is often written like this:

```
OPEN INPUT  INFILE
     OUTPUT OUTFILE
```

Two types of access include INPUT and OUTPUT; there are other types, but we need only these two for now.

Notice again that the names used with the verbs are those associated with both the SELECT and FD entries.

When opening for input

- The file should already exist

When opening for output

- The file would **not** normally exist, since the file will be created with the OPEN OUTPUT statement, thereby deleting the existing file.
- If it does exist, it can be opened to overwrite the current data if you require.

Copyright © Micro Focus 2015-2016. All rights reserved.

- Alternatively it can be opened to append to the current data using the **OPEN EXTEND** statement.

When opening to extend an output file

You may remember that we mentioned that by default COBOL clears output files if they exist.

This is what OPEN **OUTPUT** does. If you instead write OPEN **EXTEND**, a pointer is positioned at the end of any existing records, and new records are added to the end of the file. E.g.

```
OPEN EXTEND OUTFILE
```

Note: If you try to open a file that is already opened, you will get a run-time error.

The READ verb

The file to be READ must have been opened as input.

The READ of a sequential file reads the **next** record in the file (At the start of the file, it reads the first record on file).

The statement should also inform the program what to do when there are no more records to be read. In our sample solution **06_02_Program_Design_Full.sln** the result of a successful read:

- Returns a record into the Group Item defined – **RECORD-TYPE-A.**
- Since **RECORD-TYPE-B** is just a redefinition of **RECORD-TYPE-A** the record is also implicitly returned to this second definition.
- The result of an empty file or no more records to read, initiates the statements after the **"AT END"** e.g.

```
READ INFILE
    AT END
        MOVE 1 TO WS-END-OF-FILE
END-READ
```

- **If the READ *succeeds***, that is, if there is a record or another record, the contents of that record will go into all of the 01 descriptions specified.

In our case, the contents go into the record description of both **RECORD-TYPE-A** and **RECORD-TYPE-B**, as shown in our code previously. (This area is known as the *record buffer*.) **WS-END-OF-FILE** will not be touched.
If the read *fails*, **WS-END-OF-FILE** will be set to 1. The contents of **RECORD-TYPE-A** and **RECORD-TYPE-B** may be undefined (if there has never been a record) or those of the last record read (if there have been previous records). Logically, we should not look at them anyway, because there has been no new record read.

Why do we read a file name?

You will notice that the READ statement refers to the **file name**, not the **record name**. It is possible that a file may contain records of two or more types (as is the case for us here). Because this is an input file, we have no idea what record type is coming in next. All we can do is read the file name.

The WRITE verb

The converse of READ is WRITE.

Copyright © Micro Focus 2015-2016. All rights reserved.

While the verb READ moves data from the file to storage, WRITE moves data from storage to the next record in the file.

To WRITE to a file, the file must have been opened as either OUTPUT or EXTEND.

The WRITE to a sequential file writes to the next record position in the file (At the start of the file, it writes the first record on file).

Unlike READ's target, the program writes a **record name**, not a **file name**, since we know what record we are writing.

```
MOVE SR-NAME TO OUT-NAME
WRITE OUTREC
```

Why do we WRITE a record name?

Since this is output, we are creating the record. If there more than one type exists, we need to specify which one we are writing.

Notice above that prior to a WRITE, we need to have built up the output record in its record buffer.

The CLOSE verb

All files should be CLOSE'd before program termination

It is normally the last operation performed on a file before the program is finally halted.

The file must have been opened before it can be closed.

Logically, CLOSE statements usually belong at the end of the program logic; just as OPEN statements normally belong at the beginning.

The following code shows how we might use the CLOSE verb.

```
CLOSE INFILE
CLOSE OUTFILE
```

However, the following code shows how it would often be written.

```
CLOSE INFILE
      OUTFILE
```

While files can be opened in different ways, they are all closed in the same way.

So, the format of the CLOSE statement is simpler. Closing a file releases the file back to the operating system so that other programs can use it.

Note: If you try to close a file that is not open you will get a run-time error.

Sequential file access verbs summary

The following shows all the main sequential file verbs in summary:

Copyright © Micro Focus 2015-2016. All rights reserved.

```
PROCEDURE DIVISION.
PROG.
    OPEN    INPUT   INFILE
            OUTPUT  OUTFILE
[MORE CODING]
    READ   INFILE
      AT END
         MOVE 1 TO WS-END-OF-FILE
    END-READ
[MORE CODING]
    MOVE IN-NAME     TO    OUT-NAME.
    MOVE IN-DOB      TO    OUT-DOB.
    WRITE OUTREC
[MORE CODING]
    CLOSE   INFILE
            OUTFILE
```

Exercise 1

In the previous module we briefly saw the workspace **06_02_Program_Design_full.**

Now switch to this workspace and study the program's behaviour, in debug mode, using the knowledge that you have obtained so far in this module.

How would you alter the program to add to the end of the output file, each time the program is run?

Extending the verbs

Some of the sequential file verbs you have seen so far can be extended.

The full READ verb

The READ verb can be extended to

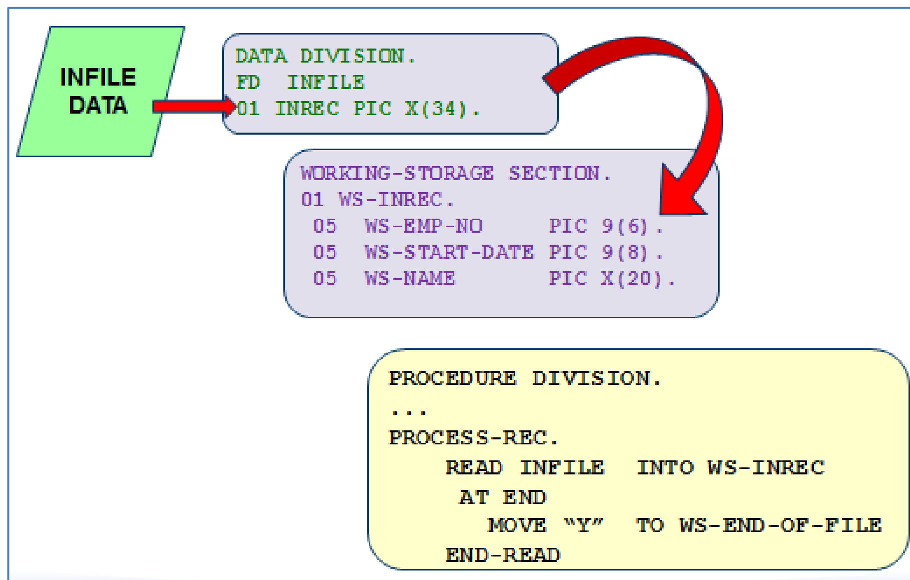
- Have **"AT END"**
- And **"NOT AT END"**
- READ the input record directly into the Working-Storage Section

e.g.

```
READ INFILE
    AT END
        MOVE 1 TO WS-EOF
    NOT AT END
        PERFORM GOT-A-RECORD
END-READ
```

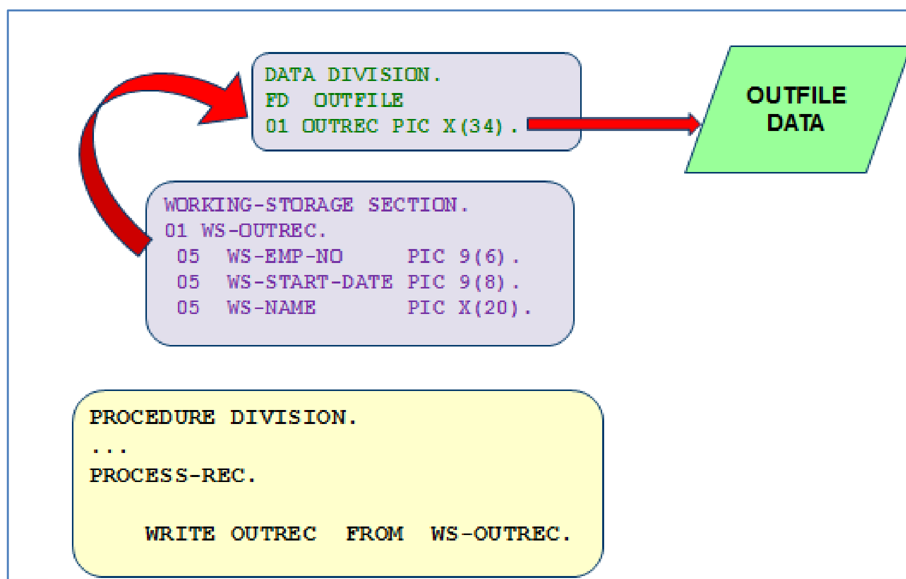
It can also be extended to READ a record directly into a working-storage data item

Copyright © Micro Focus 2015-2016. All rights reserved.



The full WRITE verb

This verb can also be extended to write from a working-storage data item.



Points to remember.

- When you OPEN a file for writing
  - **OPEN OUTPUT filename** – will overwrite existing file data
  - **OPEN EXTEND filename** – will append to existing file data
- You READ a FILE
  - **READ filename**
- You WRITE a RECORD
  - **WRITE recordname**

Copyright © Micro Focus 2015-2016. All rights reserved.

- You cannot initialize data items in the File Section
  - the data from the input/output operation is moved into this area

File records of different lengths

COBOL has no problem handling files with multiple record types. In our example so far we had a file with 2 record types. Both of these record types were the same length (74 bytes each). However there is no problem to COBOL in having records of different lengths.

If you do have different record lengths then there are some obvious things to be aware of:

- If you use READ INTO, make sure your working-storage item is big enough to handle the largest record.
- If you use WRITE FROM, make sure the working-storage item complies with the record length you require.

You can think of a READ INTO and WRITE FROM as 2 statements. In that way you will avoid any size mistakes.

So for a READ

```
READ INFILE INTO WS-INREC
```

is exactly the same as:

```
READ INFILE
MOVE INREC TO WS-INREC
```

For a WRITE

```
WRITE OUTREC FROM WS-OUTREC
```

is exactly the same as:

```
MOVE WS-OUTREC TO OUTREC
WRITE OUTREC
```

Module Summary

Now you have completed this module, you can:

- Use the Environment Division and Data Division entries in a COBOL program that uses input or output sequential files.
- Use the correct format of READ, WRITE, OPEN and CLOSE statements in the Procedure division for sequential files, including testing for an end-of-file condition.
- Describe how COBOL deals with multiple record types on the same file.
- Be able to design and code a typical COBOL program that handles sequential files.

Exercise 2

There is a workspace **07_01_multi_records** which you should switch to. The program in here illustrates many of the features we have defined above.

Copyright © Micro Focus 2015-2016. All rights reserved.

The purpose of this program is to read an employee file, create two output files and display a counter.

One file will contain the 'normal' employee records; the other file will contain records for each employee leaving the company and a control record at the end of the file.

The program will also count and display the number of records written to both output files.

Study this program to ensure that you understand all the processing.

Exercise 3

There is another workspace **07_02_multi_records2** which is similar to the last. You may find it useful to step through the code in this program to re-emphasise the way that sequential files are processed.

You will notice that there are a few comments on the program. You will see we have used a level 88 data name (This is a **condition name**). Also look at the SET statement to set the condition to true. We will discuss these later in the class.

Quick Quiz

1. Name the five verbs used in sequential data file handling

2. How is a data file appended to, rather than overwritten?

3. How is the empty file event handled when opening a file for input?

4. You wish to retrieve data from a file, which of the following statements would you most likely use?

    a. **READ INFILE INTO OUTREC**
    b. **READ INFILE AT END MOVE "Y" TO WS-EOF**
    c. **READ INREC AT END MOVE "Y" TO WS-EOF**
    d. **WRITE OUTREC FROM INREC .**
5. Which of the following is true of an AT END clause after a READ?
    a. It must be terminated with a period
    b. It must be terminated with an AT END
    c. It does not need to be terminated
    d. It must be terminated with something
6. AT END can be used with both a READ and WRITE?
    a. TRUE
    b. FALSE
7. Any sequential input file must have all its records the same length?
    a. TRUE
    b. FALSE
8. Any sequential output file must have all its records the same length?
    a. TRUE
    b. FALSE
9. Files are selected in:

Copyright © Micro Focus 2015-2016. All rights reserved.

   a. IDENTIFICATION DIVISION
   b. ENVIRONMENT DIVISION
   c. DATA DIVISION
   d. PROCEDURE DIVISION

10. When you write to a file, you write using?
    a. The file name
    b. The record name

Copyright © Micro Focus 2015-2016. All rights reserved.

08 Decision Logic

Introduction

Implementing the required decision logic in a program effectively means higher quality programs, shorter testing time, and better maintenance productivity.

Here, we understand the different conditional constructs available in COBOL and understand when and how to use them.

Most programming requires making a choice

- If one condition occurs, the program should do this task.
- If a different condition occurs, the program should do a different task.

In COBOL the flow of a program is controlled almost entirely by

- `IF-ELSE` or `EVALUATE` statements
- the `PERFORM` verb
- possibly the `GO TO` verb

Module Objectives

Upon successful completion of this module, you will be able to:

- Use the verb **EVALUATE** and the **IF** condition as well as the Level **88** construct in testing conditions
- Describe the different types of conditions
- Explain the advantages and disadvantages of different ways of condition testing
- Spot and fix infinite loops

The IF statement

The following is a simple IF Statement

```
IF [condition] (THEN)
    [do one or more statements]
(ELSE)
    [do one or more statements])
(END-IF)
```

The entries in round brackets ( ) are optional.

This code indicates that if a condition is true, then perform an action.

Otherwise (if the condition is false), do a different action.

Lastly, END-IF marks the end of the conditional statement. Prior to 1985 a period denotes the end of the IF).

Copyright © Micro Focus 2015-2016. All rights reserved.

Condition phrases

In the above sample IF statement, the "condition" phrase can take a number of forms:

- Relational conditions
- Class conditions
- Sign conditions
- Condition-name conditions (Level 88s)
- Compound conditions
- Nested conditions

Relational conditions

A relational condition compares two or more operands, for example:

```
IF CUSTOMER-AGE < 21
    DISPLAY 'CUSTOMER TOO YOUNG'
END-IF
```

You can also use the **NOT** to negate the condition, for example:

```
IF PARENTS-CONSENT NOT = 'YES'
    DISPLAY 'NO CONSENT GIVEN'
END-IF
```

In some cases we might want to determine whether a data is not equal to something.

Prior to 1985 no equivalent existed for **<>**, which means **not equal**; however, we could express it in one of the following ways.

```
IF WS-TOTAL NOT = 6
IF WS-TOTAL UNEQUAL 6
IF WS-TOTAL NOT EQUAL TO 6
IF WS-TOTAL <> 6
```

Relational conditions are clear when comparing numeric items. However relational comparisons are also completely valid for comparing alphanumeric items.

- Which is greater, "FRED" or "FREDA"?

F	R	E	D	A
F	R	E	D	

- Strings are evaluated left to right, looking for a 'no match'
- As soon as there is a 'no match', that decides the issue
- " " is less than "A"

Copyright © Micro Focus 2015-2016. All rights reserved.

For the first four bytes “FRED” and “FREDA” are equal.

At the fifth byte, “A” is compared with space, since “FRED” does not contain any more real characters that can be checked. (This still happens if the string holding “FRED” has no bytes that are unoccupied.) Since hex ‘20’ represents space in the ASCII character set, and hex ‘41’ represents “A”, “FRED” is less than “FREDA”.

***Note**: In modern COBOL running on a PC or UNIX platform, the character code set is normally ASCII. In mainframe platforms the character code set is normally EBCDIC. The character sequence in ASCII and EBCDIC are different. So that a comparison which works one way in EBCDIC, may work the opposite way in ASCII. This only becomes relevant if you are comparing alphanumeric items which can contain mixed upper and lower case, or items which contain a mixture of alphabetic and numeric data.*

Class Conditions

A class condition allows the programmer to test for a specific type of data, for example, **ALPHABETIC** or **NUMERIC**

```
IF EMPLOYEE-NAME IS ALPHABETIC
    PERFORM VALID-PROCESSING
ELSE
    PERFORM ERROR-ROUTINE
END-IF
```

or

```
IF EMPLOYEE-AGE IS NUMERIC
    PERFORM VALID-PROCESSING
ELSE
    PERFORM ERROR-ROUTINE
END-IF
```

Sign Conditions

Sign conditions allow the programmer to test numeric fields for POSITIVE, NEGATIVE, or ZERO. For example:

```
01  WS-CUSTOMER-BALANCE  PIC S9(5)V99.

IF WS-CUSTOMER-BALANCE IS NEGATIVE
   PERFORM REFUSE-LOAN
END-IF

IF WS-CUSTOMER-BALANCE IS ZERO
   PERFORM REFUSE-LOAN
END-IF

IF WS-CUSTOMER-BALANCE IS POSITIVE
   PERFORM ACCEPT-LOAN
END-IF
```

Condition Names

To make IF condition tests easier to write (and read), use Level 88, a special level number in COBOL programs

Copyright © Micro Focus 2015-2016. All rights reserved.

A level 88 is always associated with another variable. It may seem on first glance as if we are setting a VALUE clause; however, we are not.

Instead of moving a value to a field, we are declaring one or more values against which to test.

This makes the IF statement shorter and possibly more meaningful.

See the example:

```
01  INREC.
    03  INREC-AGE         PIC 99.
         88  VALID-AGE         VALUE 13 THRU 99.
         88  TEEN              VALUE 13 THRU 19.
         88  YOUNG-ADULT       VALUE 20 THRU 29.
         88  STILL-YOUNGISH    VALUE 30 THRU 39.
         88  PAST-IT           VALUE 40 THRU 99.

IF NOT  VALID-AGE
    DISPLAY 'TOO YOUNG TO JOIN FACEBOOK!'
END-IF
IF  PAST-IT
    PERFORM MAKE-TEXT-BIGGER-FOR-OLD-EYES
END-IF
IF  TEEN
    PERFORM TREAT-WITH-CARE
END-IF
IF  YOUNG-ADULT
    PERFORM NOT-TO-BAD
END-IF
IF  STILL-YOUNGISH
    PERFORM BUY-A-DRINK
END-IF
```

The 88 can be extended to contain multiple values. For example

```
01  WS-GENDER          PIC X.
    88  FEMALE                value "F" "f".
    88  MALE                  value "M" "m".

 01  WS-CREDIT-LIMIT  PIC 9999.
     88  LOW-BALANCE          value 0 thru 500.
     88  HIGH-BALANCE         value 8000 thru 9999.

 01  INREC.
     03  INREC-TYPE       PIC X.
         88  VALID-TYPES      VALUE "A" THRU "E" "G" "Z".
         88  CEO              VALUE "A".
         88  VICE-PRESIDENT   VALUE "B".
         88  SENIOR-MANAGER   VALUE "C".
         88  MANAGER          VALUE "D".
         88  TEAM-MEMBER      VALUE "E".
         88  JANITOR          VALUE "G".
         88  TRAINER          VALUE "Z".
```

Using a condition name to signal end of file:

One very common use of the 88 levels is on the flag used to signify end of file:

Copyright © Micro Focus 2015-2016. All rights reserved.

```
01  WS-END-OF-FILE      PIC 9 VALUE 0.
    88  NO-MORE-RECORDS       VALUE 1.

PERFORM PROCESS-REC UNTIL NO-MORE-RECORDS
    [other code]
READ INFILE
    AT END
        SET NO-MORE-RECORDS TO TRUE
END-READ
```

Using LEVEL 88's encourages more English-like statements in the PROCEDURE DIVISION.

The code:

```
PERFORM PROCESS-REC UNTIL NO-MORE-RECORDS
```

is clearer than:

```
PERFORM PROCESS-REC UNTIL WS-END-OF-FILE IS NOT EQUAL TO ZERO
```

Compound Conditions

To test more than one condition at once, use compound conditions in your IF statement

```
IF WS-TOT-3 = 5 OR WS-SUM-X > 22

IF WS-TOTAL > 23   AND WS-SUM = 200  OR WS-FINAL = 444
```

When using AND and OR in the same test always use brackets for clarity, so that the condition in brackets is evaluated before anything else. E.g. the following 3 **IF**s all behave differently. So make sure you use brackets for clarity:

```
IF  WS-TOTAL > 23  AND  WS-SUM = 200  OR  WS-FINAL = 444
IF (WS-TOTAL > 23  AND  WS-SUM = 200) OR (WS-FINAL = 444)
IF (WS-TOTAL > 23) AND (WS-SUM = 200  OR  WS-FINAL = 444)
```

Nested Conditions

It is possible to nest IF statements as you can see in this example:

```
IF INREC-JOB-TITLE = 'MANAGER'
    IF INREC-SALARY > 50000
        IF INREC-START-DATE < 20000101
            [actions1]
        ELSE
            [actions2]
        END-IF
    ELSE
        [actions3]
    END-IF
ELSE
    [actions4]
END-IF
```

Such statements, even when indented as shown, can be difficult to understand, and used to be avoided before END-IF became available after 1985.

Copyright © Micro Focus 2015-2016. All rights reserved.

The NEXT SENTENCE clause

The **next sentence** clause is used to exit an IF statement. E.g.

```
IF DATA-NAME = 'FRED'
     NEXT SENTENCE
ELSE
     ADD 1000 TO WS-NUMBER
END-IF
```

The EVALUATE statement

The **EVALUATE** verb has a different syntax to that of **IF**

Conditions that would be clumsy or complex as an IF statement can often be better expressed with EVALUATE.

EVALUATE exists in three formats:

- Simple EVALUATE statement
- EVALUATE condition statement
- Compound EVALUATE statement

Simple EVALUATE

This simple evaluate tests the values in a data item and behaves as appropriate.

```
EVALUATE INREC-TYPE
   WHEN 'A' THRU 'D'
        PERFORM MANAGER-STUFF
   WHEN 'E'
        PERFORM OTHER-CODE
   WHEN 'G'
        PERFORM JANITOR-CODE
   WHEN 'Z'
        PERFORM TRAINER-FUNCTIONS
   WHEN OTHER
        PERFORM INVALID-RECORD
END-EVALUATE
```

The **WHEN OTHER** clause is very useful in an **EVALUATE** statement, as it is a "catch-all" for when none of the conditions specified are met.

This often indicates invalid data or, at the very least, something the programmer did not expect to happen.

In the above example, on the `WHEN 'A' THRU 'D'`, if you just want this to be true for just `'A'` and `'C'` then you can write:

```
EVALUATE INREC-TYPE
   WHEN 'A'
   WHEN 'C'
        PERFORM MANAGER-STUFF
   WHEN 'E'
        PERFORM OTHER-CODE
   WHEN 'G'
        PERFORM JANITOR-CODE
   WHEN 'Z'
        PERFORM TRAINER-FUNCTIONS
```


Copyright © Micro Focus 2015-2016. All rights reserved.

```
      WHEN OTHER
          PERFORM INVALID-RECORD
   END-EVALUATE
```

Condition EVALUATE

The following code tests three conditions using the **Evaluate True** version:

- end of file using a level 88 condition name,
- when the counter equals 99, or
- when another condition is true (the catch all).

```
EVALUATE TRUE
     WHEN NO-MORE-RECORDS
         PERFORM UNEXPECTED-EOF
     WHEN WS-COUNTER = 99
         DISPLAY 'Too many records'
         STOP RUN
     WHEN OTHER
         PERFORM NORMAL-ACTIONS
END-EVALUATE.
```

When one of these conditions is true, then the EVALUATE is true and the condition-specific action is taken.

Once one condition is tested and found true, then the Evaluate terminates (the other tests are ignored). So in the above example, if **both** `NO-MORE-RECORDS` is true and `WS-COUNTER = 99` then only the first condition `NO-MORE-RECORDS` would be acted upon.

The EVALUATE statement can be much simpler to code and much easier to read than a series of nested IF statements.

Compound EVALUATE

To test multiple subjects in an EVALUATE statement, use an ALSO clause instead of AND as shown below:

```
EVALUATE IN-JOB-TYPE   ALSO TRUE
    WHEN 'MANAGER'     ALSO IN-SALARY > 250000
        PERFORM REWARD-TOP-EXECUTIVES
    WHEN 'TEAM LEADER' ALSO IN-SALARY > 80000
        PERFORM BENEFIT-MIDDLE-PEOPLE
    WHEN 'EMPLOYEE'    ALSO IN-SALARY < 50000
        PERFORM EMPLOYEE-BENEFITS
END-EVALUATE
```

This Compound Evaluate is used to implement Truth Tables.

The CONTINUE clause

To indicate that a program should do nothing when a particular condition is true, use the **CONTINUE** clause, which tells the program to exit the evaluate and execute the next statement after the END-EVALUATE. It is a little like the **NEXT SENTENCE** clause in an IF statement.

Copyright © Micro Focus 2015-2016. All rights reserved.

```
 01  CUSTOMER-TITLE      PIC X(4).
     88  MALE-TITLE        VALUE "MR".
     88  FEMALE-TITLE      VALUE "MRS" "MISS" "MS".

      EVALUATE TRUE
          WHEN MALE-TITLE
              CONTINUE
          WHEN FEMALE-TITLE
              PERFORM FEMALE-CUST
          WHEN OTHER
              PERFORM INVALID-TITLE
      END-EVALUATE
```

Infinite loops

As in any other programming language, you can produce loops that never terminate.

Example 1:

```
 01   MY-COUNTER     PIC 99.

 MAIN-PROCESS.
     PERFORM PROCESS-DATA
         VARYING MY-COUNTER FROM 1 BY 1 UNTIL MY-COUNTER > 100
```

In this example `MY-COUNTER` can never reach 100 since the data item is only 2 bytes. When it reaches 99 and then is increment by 1, the data item will revert to 0.

The solution would be to make `MY-DATA` a `PIC 999.`

Example 2:

```
01   MY-DATA         PIC 9(6)V99.

MAIN-PROCESS.
    MOVE 0 TO MY-DATA
    PERFORM UNTIL MY-DATA = 100
        ADD 0.37 TO MY-DATA
    END-PERFORM
```

`MY-DATA` will never become equal to `100`.

One solution might be to change the test to: `PERFORM UNTIL MY-DATA > 100`

Example 3:

This next example could cause an infinite loop on some compilers. On the Micro Focus compiler it does not cause an infinite loop, but does cause the program to crash if you try to read a record after the end of file has been reached.

```
MAIN-PROCESS.
    READ MY-FILE
```


Copyright © Micro Focus 2015-2016. All rights reserved.

```
    PERFORM PROCESS-FILE UNTIL END-OF-FILE.
    . . . .
PROCESS-FILE.
    MOVE A TO B
    ADD 1 TO RECORD-COUNT
    READ MY-FILE.
```

The `END-OF-FILE` condition is never set, so the read will continue to read a null record at the end of file forever.

The solution, in this case, would be to change the code to:

```
MAIN-PROCESS.
    PERFORM READ-MY-FILE
    PERFORM PROCESS-FILE UNTIL END-OF-FILE.
    . . . .
PROCESS-FILE.
    MOVE A TO B
    ADD 1 TO RECORD-COUNT
    PERFORM READ-MY-FILE.
READ-MY-FILE.
    READ MY-FILE
      AT END
        SET END-OF-FILE TO TRUE
    END-READ.
```

Module Summary

Now you have completed this module, you will be able to:

- Use the verb **EVALUATE** and the **IF** condition as well as the Level **88** construct in testing conditions
- Describe the different types of conditions
- Explain the advantages and disadvantages of different ways of condition testing
- Spot and fix infinite loops

Exercise 1

You should use the workspace **08_01_Decision_Making1.**

This solution contains 2 programs:

- **DecisionMakingProgram1.cbl**
- **DecisionMakingProgram2.cbl**

You should first examine **DecisionMakingProgram1.cbl** and execute it in debug mode to see how the IFs and EVALUATEs are working.

To debug this first program, right-click on the program name and select **Debug As/COBOL Program.**

Copyright © Micro Focus 2015-2016. All rights reserved.

When you are happy with an understanding of **DecisionMakingProgram1** then you need to do the same with **DecisionMakingProgram2**. This program is very similar to the previous program, but with slight changes.

In both cases you should ask yourself the following questions:

1. How is the record type validated?
2. Do any of the 88 levels overlap? Does this create any problems?
3. If you look at the output files, can you see anything wrong with the "Special" records? If you can, how would you fix this?

Exercise 2

You are going to write a program whose purpose program is to read the employee records and create three output files. Each will contain records of employee's within a range of salaries; either low medium or high.

This program should also count and display the number of employee's in each category.

Additionally, this program should evaluate the "TYPE" code on each record, rejecting and displaying invalid records.

The steps to follow are:

1. Switch your workspace to **C:\COBOLClass_Eclipse\Projects\08_02_Decision_Making2.**
2. This workspace contains a partly completed program. Use this to create a COBOL program to do the following:
    a. Read in a file of records, and then deal with them in different ways depending on the type of record found. The input file may have any number of records (including 0).
    b. All input records contain a one-character type, a name field (20 characters), an eight-digit start date, and an eight-digit salary field (six digits, two decimal places).
    c. If the type field is set to anything other than "A", "B", "C", "E", "R" or "T", reject the record and display the details. Similarly, if the name field is blank, reject the record in the same way. Keep a count of all such invalid records.
    d. If the salary is less than 25,000, write the record details unchanged to an output file. At the end of the program write a special record to that file containing the total number of records written (excluding that one). This record could contain the number zero.
    e. Include in another output file unchanged details of all records where the salary is between 25,000 and 64,999.99. Again, write a special record, containing the number of this type of record written.
    f. Include all unchanged details of all other salaries in a third output file. Again, write a special record at the end of the program.
3. Save the program.
4. Compile and test the program.

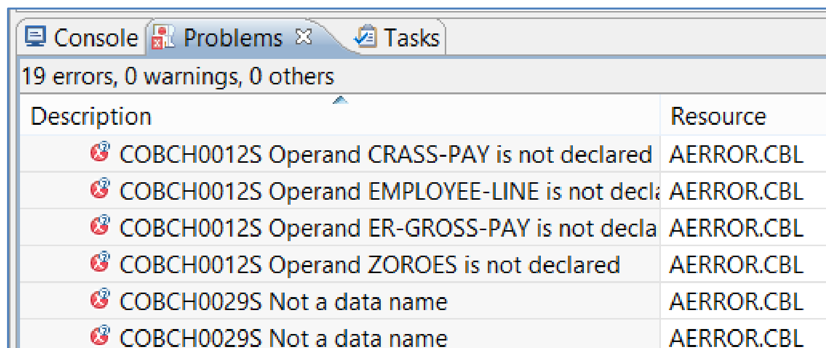
Copyright © Micro Focus 2015-2016. All rights reserved.

Exercise 3 – Fixing compilation errors

This exercise is not strictly a test of IF and EVALUATE, but is bringing together much of what you have seen so far.

Switch to the workspace **\COBOLClass_Eclipse\Projects\08_03_Bugs** and see if you can fix the compiler errors in this program.

There are a number of compiler errors in this program including:



Further Simple Exercises

There are a couple of other very simple solutions you can look at to extend your knowledge of EVALUATE. These can be found by switching your workspace to:

- **08_04_Evaluate_True**
- **08_05_Evaluate_positive**

Exercise 4 – Fixing loop and file end problems

The project in workspace **08_06_Infinite_Loop** has 3 "loop" problems, as described above. See if you can find and subsequently fix the 3 problems.

Quick Quiz

1. The EVALUATE statement must be terminated by?
    a. END-EVALUATE
    b. A period
    c. Either of the above
    d. Neither of the above
2. Which of the following is true?
    a. The NEXT SENTENCE clause is used in an IF statement
    b. The CONTINUE clause can be used in an EVALUATE statement
    c. Neither is true
3. In order to test a numeric field for positive or negative the field has to be a signed field?
    a. TRUE
    b. FALSE
4. Condition names can only apply to an elementary data item?
    a. TRUE
    b. FALSE

Copyright © Micro Focus 2015-2016. All rights reserved.

5. When comparing two PIC X(n) fields, they must both be the same size?
    a. TRUE
    b. FALSE
6. If you are using nested IF statements a period terminates ALL the Ifs?
    a. TRUE
    b. FALSE
7. You can change the value in a condition name at run time?
    a. TRUE
    b. FALSE
8. The “catchall” in an evaluate statement is:
    a. WHEN ANY
    b. WHEN OTHER
    c. WHEN NEITHER
    d. There is no “catchall” statement

Copyright © Micro Focus 2015-2016. All rights reserved.

09 Data Manipulation

Introduction

Understanding the full range of the data manipulation constructs available in COBOL means more efficient programming and maintenance.

This module examines the set of COBOL verbs available to manipulate both arithmetic and character data.

Module Objectives

By the end of this module you will be able to:

- Explain the functionality of the **INITIALIZE** verb.
- Use the various arithmetic COBOL verbs, such as **ADD, SUBTRACT, MULTIPLY, DIVIDE** and **COMPUTE.**
- Use the verbs for manipulating character data, such as **INSPECT, STRING** and **UNSTRING.**
- Explain the role of ***reference modification***.

Manipulating Data

The COBOL verbs available for data manipulation fall into three distinct groups:

- The **INITIALIZE** verb, which can be used on both numeric and character data
- The arithmetic verbs **ADD, SUBTRACT, MULTIPLY, DIVIDE** and **COMPUTE**
- The string handling verbs **INSPECT, STRING** and **UNSTRING**

This module also explains how to extract a substring from a string using "**reference modification**".

The INITIALIZE verb

The INITIALIZE verb, as its name implies, sets elementary or group data items to an initial value.

For example, if you write an income report and the program calculates the total income, you might want to clear values and reset them to "0" in order to begin subsequent income reporting.

INITIALIZE can be used anywhere in the program. So, it can be executed many times during the program.

Furthermore, it will always "intelligently" decide what constitutes a sensible initial value.

INITIALIZE sets alphanumeric data items to spaces and numeric data items to zeros, if no other value is specified.

Example of INITIALIZE verb

```
01 WS-DATA-ITEMS.
   03 WS-NAME         PIC X(20).
   03 WS-SALARY       PIC 9(6)V99.
   03 WS-RECORD-COUNT PIC 9(4) COMP.
   03                 PIC X(40).
   03 WS-ADDRESS      PIC X(80).
```


Copyright © Micro Focus 2015-2016. All rights reserved.

```
INITIALIZE WS-DATA-ITEMS
```

After the INITIALIZE:

- **WS-NAME** and **WS-ADDRESS** will be set to spaces
- **WS-SALARY** will be set to zeros
- **WS-RECORD-COUNT** will be set to binary zeros
- The forty-character filler will be left untouched

This use is less common:

```
INITIALIZE WS-DATA-ITEMS  REPLACING
           ALPHANUMERIC DATA BY ALL "?"
           NUMERIC DATA BY ALL "6"
```

After the INITIALIZE:

- **WS-NAME** and **WS-ADDRESS** will contain "?" characters
- **WS-SALARY** will contain 6666.00 (*not* 6666.66)
- **WS-RECORD-COUNT** will contain 6666
- The forty-character filler will be left untouched

Arithmetic verbs

There are five verbs that we shall consider:

- ADD (2 formats)
- SUBTRACT (2 formats)
- MULTIPLY (2 formats)
- DIVIDE (5 formats)
- COMPUTE

Many of these verbs have alternate formats. We will look though each briefly.

Before that, here are some general points:

- You must use only numeric data in arithmetic operations
- By default, there is no rounding, only truncation. Thus 35 divided by 12 will return 2, unless rounding is specified
- COBOL does not automatically check for the result of a calculation going out of range. However, you can specify that this check takes place.
- Should an out-of-range event occur, with no check in place, the results of the calculation are **undefined.**

ADD Format 1

Examples:

```
ADD WS-WEEKLY-SALES TO WS-MONTHLY-SALES
ADD WS-MONTHLY-SALES TO WS-YEARLY-SALES WS-CENTURY-SALES
ADD RESULT-1 RESULT-2 10000 TO WS-TOTAL
ADD 2000 TO EMPLOYEES-SALARY
```

This means that the operand or operands to the left of the "TO" are added to the field or fields to the right of the "TO," incrementing them.

Copyright © Micro Focus 2015-2016. All rights reserved.

The fields or literals to the left are unchanged.

The receiving field or fields (to the right of the "TO") are formatted according to their PICTURE clauses.

If you want to apply rounding to any of these, you add the word ROUNDED at the end of the line. E.g.

```
ADD RESULT-1 RESULT-2 10000 TO WS-TOTAL ROUNDED
```

Using ROUNDED, not surprisingly, rounds the answer according to the number of decimal places specified in the receiving field.

ADD Format 2

Examples:

```
ADD WS-WEEKLY-SALES TO WS-MONTHLY-SALES GIVING WS-TOTAL-SO-FAR
ADD WS-WEEK1 WS-WEEK2 WS-WEEK3 WS-WEEK4 250 TO WS-ALL-WEEKS
                                        GIVING WS-TOTAL1 WS-TOTAL2
ADD 100 200 300 400 GIVING WS-ADDITION
```

As above, if you want to apply rounding to any of these, you add the word **ROUNDED** at the end of the line. E.g.

```
ADD 10.01 202.34 305.11 498.32 GIVING WS-ADDITION ROUNDED
```

SUBTRACT Format 1

Examples:

```
SUBTRACT 10 FROM WS-TOTAL1 WS-TOTAL2
SUBTRACT WS-TOTAL1 WS-TOTAL2 FROM WS-SALARY
```

The value in the field or fields to the left of the "FROM" are added together if appropriate. That total value is subtracted from the field or fields to the right of the "FROM."

As above, if you want to apply rounding to any of these, you add the word **ROUNDED** at the end of the line.

SUBTRACT Format 2

Examples:

```
SUBTRACT 10 FROM WS-COUNT-1 GIVING WS-COUNT-2 WS-COUNT-3
SUBTRACT TAXES-1 TAXES-2 FROM WS-GROSS-SALARY GIVING WS-NET-SALARY
SUBTRACT 12 FROM 24 GIVING WS-NUM
```

Here the operand or operands to the left of the "FROM" are added together, and their combined value is subtracted from the one operand after the "FROM."

The result is moved to the field or fields after the GIVING.

Again, the word **ROUNDED** after a receiving field ensures that rounding takes place.

Note*: Ensure that all fields from which you are SUBTRACTing are signed fields. If not, results that should be negative, remain positive with potentially catastrophic results for your application.*

Copyright © Micro Focus 2015-2016. All rights reserved.

MULTIPLY Format 1

Examples:

```
MULTIPLY WS-INTEREST-RATE BY WS-SUM
MULTIPLY 1.01 BY EMPLOYEE-SALARY
MULTIPLY 1.01 BY EMPLOYEE-SALARY MANAGER-SALARY
```

In this format, the value of the operand (field or literal) before the “BY” is multiplied by the values in the field or fields after the “BY,” and the result is moved to the field(s) after the BY.

NOTE: The following would be incorrect using this format.

```
MULTIPLY EMPLOYEE-SALARY BY 1.01
```

Why not? Format 1 stores the results of the multiply in the operand **after** the BY. Also, 1.01 is a literal and cannot be used to store data.

As shown in the below, you can also include multiple fields after the “BY” so that the operand before the BY is multiplied by the fields after the “BY.”

```
MULTIPLY 1.01 BY EMPLOYEE-SALARY MANAGER-SALARY
```

The results are stored in both EMPLOYEE-SALARY and MANAGER-SALARY.

Rounding, of course, behaves the same way as in ADD and SUBTRACT.

MULTIPLY Format 2

Examples:

```
MULTIPLY WS-INTEREST-RATE BY WS-SUM GIVING WS-ANSWER
MULTIPLY WS-TOTAL BY 4 GIVING WS-TOTAL-2.
MULTIPLY EMPLOYEE-SALARY BY 1.01 GIVING EMPLOYEE-SALARY
```

In this format, only one field or literal can be before the “BY” and one after; however, the result can be moved to one or more fields (which can include a field already used in the MULTIPLY).

The word **ROUNDED** can be inserted after the name of a receiving field.

Preventing an Out-of-Range Condition

As a general rule, a **MULTIPLY** statement is more likely than an **ADD** statement to result in a field not being large enough for the calculation result.

Trap this out-of-range condition by using an **ON SIZE ERROR** clause, discussed later in this module.

DIVIDE

The DIVIDE statement has a total of five formats. In all cases **ROUNDED** can be used after any receiving operand.

- Format 1: The simple format (DIVIDE INTO)
- Format 2: The GIVING format (DIVIDE INTO… GIVING)
- Format 3: DIVIDE … BY
- Format 4: DIVIDE INTO… GIVING, yielding a remainder

Copyright © Micro Focus 2015-2016. All rights reserved.

- Format 5: DIVIDE … BY, yielding a remainder

DIVIDE Format 1

Examples:

```
DIVIDE WS-COUNT  INTO WS-TOTAL
DIVIDE WS-COUNT2 INTO WS-TOTAL2 WS-TOTAL3
DIVIDE WS-COUNT  INTO WS-TOTAL ROUNDED
```

In statement 1 above, this results in WS-TOTAL being divided by WS-COUNT.

There can be only one operand (data item or literal) before the INTO, but there may be many operands after the INTO.

Each operand will have the same division carried out on it.

DIVIDE Format 2

Examples:

```
DIVIDE WS-COUNT INTO WS-TOTAL GIVING WS-NUM-1
DIVIDE WS-COUNT INTO WS-TOTAL GIVING WS-NUM-1 WS-NUM-2
DIVIDE WS-COUNT INTO WS-TOTAL GIVING WS-NUM-1 ROUNDED
```

This time the result goes into the field or fields after the GIVING.

DIVIDE Format 3

This format reverses the operands used in DIVIDE...INTO.

Examples:

```
DIVIDE WS-TOTAL  BY WS-COUNT   GIVING WS-NUM-1
DIVIDE WS-ANNUAL BY 12         GIVING WS-MONTHLY-AVERAGE ROUNDED
```

This yields the same result as the Format 2 statement above. Again multiple receiving fields can exist after the GIVING.

DIVIDE Format 4

This format uses a remainder, so the fields need to be integers with no decimal places.

Example:

```
DIVIDE WS-COUNT INTO WS-TOTAL GIVING WS-NUM-1 REMAINDER WS-REM
```

DIVIDE Format 5

This format also uses a remainder, so the fields need to be integers with no decimal places as above.

Example:

```
DIVIDE WS-TOTAL BY WS-COUNT GIVING WS-NUM-1 REMAINDER WS-REM
```

COMPUTE Format

COMPUTE has the receiving field(s) on the left of the statement

It uses the standard operands +, -, /, * and ** (exponentiation)

Copyright © Micro Focus 2015-2016. All rights reserved.

COMPUTE used to be a comparatively inefficient verb and was not used too much in the past. This is no longer the case with modern compilers.

Examples:

```
COMPUTE WS-TOTAL = (WS-NET-PAY + (WS-OVERTIME * WS-OT-HRS)) * (100 - WS-TAX-RATE)
COMPUTE WS-MONTH ROUNDED = WS-YEAR / 12.
COMPUTE STAR-LUMINOSITY ROUNDED = 10 ** (0.4 * (4.85 - WS-ABS-MAG))
```

Order of Evaluation

The order in which items are evaluated makes a difference to the result of the expression.

Items are evaluated in the following order:

- Items within parentheses
- Exponentiation
- Multiplication and division (same priority)
- Addition and subtraction (same priority)
- Left to right, when the operators are of equal priority

In practice, use brackets, wherever possible, to clarify order to you and anyone else who may have to amend your code in the future.

NOTE: Specify **ROUNDED** after a receiving field in the normal way.

ON SIZE ERROR clause

The ON SIZE ERROR clause can be used with any arithmetic statement to trap a numeric calculation going out of range, including division by zero. (Often used if the receiving field is not big enough to hold the result of the calculation)

If the exception does not happen, the clause is ignored.

Example:

```
COMPUTE WS-INVOICE-AMT = WS-ITEM-PRICE * WS-QUANTITY
   ON SIZE ERROR
       DISPLAY 'ERROR – TOTAL IS TOO LARGE'
END-COMPUTE
```

In the above example the program will continue if the calculation is within range.

However, if the WS-INVOICE-AMT becomes larger than its PICTURE clause allows, ON SIZE ERROR displays the error message.

Alternatively, use the NOT ON SIZE ERROR clause as well. This would be necessary if the COMPUTE statement were within the scope of another conditional statement such as an IF statement.

The following example uses both the ON SIZE ERROR and the NOT ON SIZE ERROR clauses.

```
COMPUTE WS-INVOICE-AMT = WS-ITEM-PRICE * WS-QUANTITY
   ON SIZE ERROR
      PERFORM ERROR-ACTIONS
```


Copyright © Micro Focus 2015-2016. All rights reserved.

```
          NOT ON SIZE ERROR
             PERFORM SET-UP-INVOICE
       END-COMPUTE
```

The examples shown here are using the COMPUTE statement. This clause can be used in any of the other arithmetic operators also. Because this results in ADD, SUBTRACT, MULTIPLY and DIVIDE becoming conditional statements, the following terminators are appropriate:

```
    END-ADD
    END-SUBTRACT
    END-MULTIPLY
    END-DIVIDE
```

Verbs used for string handling

COBOL deals not just with numbers, but also with strings of alphanumeric data. (PIC X and PIC A)

Three verbs manipulate such data:

- INSPECT
- STRING
- UNSTRING

INSPECT Statement

This statement can be used to look at a string of data and optionally to manipulate it. There are four formats:

- Format 1 (INSPECT...TALLYING) counts occurrences of a character or characters within a field
- Format 2 (INSPECT...REPLACING) changes a character or characters
- Format 3 (INSPECT...TALLYING...REPLACING) combines the functionality of Formats 1 and 2
- Format 4 (INSPECT...CONVERTING) changes a character or characters

In the following examples, we shall use the same fields with the same initial values.

INSPECT Format 1…Tallying

Example:

```
01  WS-COUNTS.
    03  WS-COUNT-1      PIC 99.
    03  WS-COUNT-2      PIC 99.
    03  WS-COUNT-3      PIC 99.
01  WS-STRING           PIC X(20) VALUE "AARDVARK EXTRA".


      INITIALIZE WS-COUNTS
      INSPECT WS-STRING TALLYING WS-COUNT-1 FOR LEADING 'A'
                                 WS-COUNT-2 FOR ALL 'R'
                                 WS-COUNT-3 FOR CHARACTERS
```

After the INSPECT statement, the values of the data items will be:

```
WS-COUNT-1       2
WS-COUNT-2       3
```


Copyright © Micro Focus 2015-2016. All rights reserved.

```
WS-COUNT-3     14
WS-STRING   AARDVARK EXTRA
```

INSPECT Format 2…Replacing

Using the same data definitions as above, the INSPECT statement could be:

```
INITIALIZE WS-COUNTS
INSPECT WS-STRING REPLACING FIRST "A" BY "B"
                            ALL   "R" BY "S"
```

After the INSPECT statement, the values of the data items will be:

```
WS-COUNT-1       0
WS-COUNT-2       0
WS-COUNT-3       0
WS-STRING   BASDVARK EXTSA
```

INSPECT Format 3…Tallying … Replacing

Using the same data definitions as above, the INSPECT statement could be:

```
INITIALIZE WS-COUNTS
INSPECT WS-STRING TALLYING WS-COUNT-1 FOR ALL "A"
                     REPLACING LEADING "A" BY "B"
                               ALL     "R" BY "S"
```

After the INSPECT statement, the values of the data items will be:

```
WS-COUNT-1       4
WS-COUNT-2       0
WS-COUNT-3       0
WS-STRING   BBSDVASK EXTSA
```

INSPECT Format 4…Converting

Using the same data definitions as above, the INSPECT statement could be:

```
INITIALIZE WS-COUNTS
INSPECT WS-STRING CONVERTING "RVX" TO "QUW"
```

After the INSPECT statement, the values of the data items will be:

```
WS-COUNT-1       0
WS-COUNT-2       0
WS-COUNT-3       0
WS-STRING   AAQDUAQK EWTQA
```

Every “R” changes to “Q”, every “V” to “U” and every “X” to “W”.

BEFORE and AFTER clauses

It is possible to limit INSPECT (any format) to examine only part of a string by using the AFTER and/or the BEFORE clause. For example:

```
INITIALIZE WS-COUNTS
INSPECT WS-STRING TALLYING WS-COUNT-3 FOR CHARACTERS AFTER "D" BEFORE "T"
```


Copyright © Micro Focus 2015-2016. All rights reserved.

After the INSPECT statement, the values of the data items will be:

```
WS-COUNT-1       0
WS-COUNT-2       0
WS-COUNT-3       7
WS-STRING  AARDVARK EXTRA
```

STRING Statement

STRING can be used to concatenate character strings, removing unwanted spaces. For example:

```
01  WS-FIELDS.
    03  WS-FIRST-NAME       PIC X(12) VALUE 'BEOWULF'.
    03  WS-SURNAME          PIC X(20) VALUE 'SHAEFFER'.
    03  WS-SALARY           PIC 9(6)V99.
01  WS-OUTPUT-NAME          PIC X(30) VALUE SPACES.

     STRING WS-FIRST-NAME
            '*'
            WS-SURNAME
        DELIMITED BY SPACE
           INTO WS-OUTPUT-NAME
     INSPECT WS-OUTPUT-NAME REPLACING ALL '*' BY SPACE
```

The result of the STRING and INSPECT statements will place the following value into WS-OUTPUT-NAME:

```
BEOWOLF SHAEFFER
```

Here, “DELIMITED BY SPACE” means to process all the characters in the string until a space is encountered.

So, WS-FIRST-NAME and WS-SURNAME are truncated to the right length.

However, we want a space between the two. We can’t specify “ “, as DELIMITED BY SPACE will instantly remove it. Instead, we specify an asterisk or any other character that will not be found in either part of the name.

Because “*” does not contain a space, the whole thing is moved.

String Delimit options:

```
DELIMITED BY {character}

DELIMITED BY SIZE
```

POINTER and ON OVERFLOW

STRING does not pad the target field with spaces – you may want to initialize it

POINTER can keep track of where you are in the field. So

```
01  WS-POINTER    PIC 9(4).
01  WS-FIELDS.
    03  WS-FIRST-NAME       PIC X(12) VALUE 'BEOWULF'.
    03  WS-SURNAME          PIC X(20) VALUE 'SHAEFFER'.
    03  WS-SALARY           PIC 9(6)V99.
```


Copyright © Micro Focus 2015-2016. All rights reserved.

```
 01  WS-OUTPUT-NAME          PIC X(30) VALUE SPACES.

       INITIALIZE WS-OUTPUT-NAME
       MOVE 1 TO WS-POINTER
       STRING WS-FIRST-NAME
              "*"
              WS-SURNAME
           DELIMITED BY SPACE INTO WS-OUTPUT-NAME
           POINTER WS-POINTER
         ON OVERFLOW
           PERFORM TOO-BIG
         NOT ON OVERFLOW
           PERFORM OK-ACTIONS
       END-STRING
```

As you might recall from discussions about the **MOVE** verb, **MOVE** space fills a receiving field if the value received is too small.

Since **STRING** does not do this, it is best to initialize the receiving field, and then use **POINTER** and **ON OVERFLOW** clauses, as illustrated above.

After this statement, **POINTER** contains **17**, the number of the next free byte (which is why we set it to 1 at the beginning).

Note: Specifying **POINTER** in this way requires setting it to a positive value at the start, or nothing will appear in the output string.

The **ON OVERFLOW** clause is triggered if the **STRING** is impossible because the number of characters being transferred is too large for the receiving field.

ON OVERFLOW will also be triggered if a **POINTER** clause cannot occur, which would happen if the pointer field is less than 1 or greater than the number of bytes in the receiving field.

UNSTRING Statement

UNSTRING uses delimiters to divide up a string of characters. For example:

```
01  INPUT-NAME          PIC X(30) VALUE 'PATRICIA**HUMPHREYS'.
01  RECEIVING-FIELDS.
    03  R-FIELD-1       PIC X(12).
    03  R-FIELD-2       PIC X(12).
    03  R-FIELD-3       PIC X(12).

    UNSTRING INPUT-NAME DELIMITED BY "*"
        INTO R-FIELD-1 R-FIELD-2 R-FIELD-3
```

After this statement:

- **R-FIELD-1** contains “PATRICIA”
- **R-FIELD-2** is blank (there is nothing before the next delimiter)
- **R-FIELD-3** contains “HUMPHREYS”.

The UNSTRING format, as you might expect, works in the opposite way to STRING. Use it to divide up a string of characters.

It works by recognizing delimiter characters.

Copyright © Micro Focus 2015-2016. All rights reserved.

Reference Modification

Reference modification allows you to examine or modify part of a string. For example:

```
01  WS-ALPHABET        PIC X(26) VALUE 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'.
01  WS-SMALLER-STRING  PIC X(10) VALUE SPACES.
     MOVE WS-ALPHABET(12:6) TO WS-SMALLER-STRING.
```

This will result in WS-SMALLER-STRING containing "LMNOPQ"

The 12 in WS-ALPHABET(12:6) is the start position and the 6 is the length.

If you want to move from position 12 in WS-ALPHABET to the end of the data item then the following syntax does this, without having to define the length of the string, then you can miss out the length.

```
    MOVE WS-ALPHABET(12:) TO WS-SMALLER-STRING.
```

Module Summary

At the end of this module you should now be able to:

- Explain the functionality of the INITIALIZE verb
- Use the various arithmetic COBOL verbs, such as **ADD, SUBTRACT, MULTIPLY, DIVIDE** and **COMPUTE**
- Use the verbs for manipulating character data, such as **INSPECT, STRING** and **UNSTRING**
- Explain the role of *reference modification*

Exercise 1

Check out the following syntax.

```
UNSTRING [field]
    DELIMITED BY (ALL) [literal or field name]
    OR  (ALL) [literal or field name] etc.)
    INTO [field name]
    (DELIMITER [field name])
    (COUNT [field name])
    INTO [field name]
    (DELIMITER [field name])
    (COUNT [field name] etc)
    [previous three fields repeated as necessary]
    (POINTER [field name])
    (TALLYING IN [field name])
    (ON OVERFLOW [action(s)])
    (NOT ON OVERFLOW [action(s)])
(END-UNSTRING)
```

This can be explained as follows:

- UNSTRING can use many different potential delimiters to separate out each substring.
- In each case, the field specified by DELIMITER holds the actual delimiter used.
- The COUNT field records how many bytes were transferred?
- POINTER keeps track of the total number of bytes moved over. As with STRING, set it to a positive value before the statement takes place.

Copyright © Micro Focus 2015-2016. All rights reserved.

- TALLYING IN keeps count of the total number of receiving fields that contain data when the UNSTRING finishes.
- ON OVERFLOW and NOT ON OVERFLOW are as before

Further Questions

1. Why must POINTER be set to a positive value?
2. Which is likely to be the more useful clause, ON OVERFLOW or NOT ON OVERFLOW?

Exercise 2

Switch to workspace **09_01_DataManipulation.** This is bringing together much of what you have seen so far. Spend some time looking at the program in here. Debugging is a good way to study it. (You will see a feature we have not described yet – that is the use of PIC Z for zero suppression. This will be covered shortly.)

Exercise 3

1. Here is the specification for the program, followed by the code itself.
2. The program reads in a file of sales staff records and does the following:
    a. The input file may have any number of records (including 0).
    b. All input records contain a one-character region (“N”, “S”, “E” or “W”), and an eight-digit sales total field (six digits, two decimal places).
    c. Generate four totals, one for each region. As each record is read, update the appropriate total with the value in the sales total.
    d. At the end of the program, all totals display, along with average sales for each region. We therefore need a record count for each region.

Consider the following points regarding this code.

1. Before calculating any average, the program makes a check to ensure that records were found for that region. This avoids a “divide by zero” error.
2. STRING produces messages, which are stored in WS-MESSAGE and from there DISPLAYed. Record counts employ normal fields, and so 4 displays as “0004”. However, to show the difference in appearance, *edited fields* are used for the averages. (A later module discusses edited fields.) Briefly, edited fields make a number more readable. Each edited field uses the format Z(5)9.99, which replaces any leading zeros with spaces (hence the Z). An explicit period (full stop) is inserted before the decimal part of the field, thus giving a value of, for example, 41504.88 instead of 04150488. Edited fields are very useful in making numbers and results of calculations more legible. They are not designed for calculation; therefore, numbers are moved to them after calculations are complete.
3. All the working-storage counts are PIC 9 DISPLAY rather than COMP. This is because STRING manipulates only character data.

This Example is shown below

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INFILE ASSIGN "INFILE.DAT".
```


Copyright © Micro Focus 2015-2016. All rights reserved.

```
DATA DIVISION.
FD  INFILE.
01  INREC.
    03  REGION          PIC X.
    88  NORTH                VALUE "N".
    88  SOUTH                VALUE "S".
    88  EAST                 VALUE "E".
    88  WEST                 VALUE "W".
    03  IN-TOTAL        PIC 9(6)V99.
WORKING-STORAGE SECTION.
01  WS-EOF              PIC 9 VALUE 0.
88  END-OF-FILE               VALUE 1.
01  WS-NUMERICS               VALUE ZERO.
    03  WS-N-RECORDS    PIC 9(4).
    03  WS-S-RECORDS    PIC 9(4).
    03  WS-E-RECORDS    PIC 9(4).
    03  WS-W-RECORDS    PIC 9(4).
    03  WS-N-TOTAL      PIC 9(10)V99.
    03  WS-S-TOTAL      PIC 9(10)V99.
    03  WS-E-TOTAL      PIC 9(10)V99.
    03  WS-W-TOTAL      PIC 9(10)V99.
    03  WS-N-AVERAGE    PIC 9(6)V99.
    03  WS-S-AVERAGE    PIC 9(6)V99.
    03  WS-E-AVERAGE    PIC 9(6)V99.
    03  WS-W-AVERAGE    PIC 9(6)V99.
01  WS-EDITED-FIELDS.
    03  WS-EDITED-N     PIC Z(5)9.99.
    03  WS-EDITED-S     PIC Z(5)9.99.
    03  WS-EDITED-E     PIC Z(5)9.99.
    03  WS-EDITED-W     PIC Z(5)9.99.
01  WS-MESSAGE          PIC X(80).
PROCEDURE DIVISION.
PROG-PARA.
    PERFORM INIT-PARA.
    PERFORM BOD-PARA.
    PERFORM END-PARA.
INIT-PARA.
    DISPLAY "STARTING CALCULATION PROGRAM".
    OPEN INPUT INFILE.
    READ INFILE
        AT END MOVE 1 TO WS-EOF.
BOD-PARA.
    PERFORM PROCESS-REC UNTIL END-OF-FILE.
END-PARA.
    IF WS-N-RECORDS > 0
        DIVIDE WS-N-TOTAL BY WS-N-RECORDS
            GIVING WS-N-AVERAGE ROUNDED
    END-IF
    IF WS-S-RECORDS > 0
        DIVIDE WS-S-TOTAL BY WS-S-RECORDS
            GIVING WS-S-AVERAGE ROUNDED
    END-IF
    IF WS-E-RECORDS > 0
        DIVIDE WS-E-TOTAL BY WS-E-RECORDS
            GIVING WS-E-AVERAGE ROUNDED
    END-IF
    IF WS-W-RECORDS > 0
        DIVIDE WS-W-TOTAL BY WS-W-RECORDS
            GIVING WS-W-AVERAGE ROUNDED
    END-IF
    MOVE SPACES TO WS-MESSAGE.
```


Copyright © Micro Focus 2015-2016. All rights reserved.

```
       STRING WS-N-RECORDS " " "N RECORDS, "
              WS-S-RECORDS " " "S RECORDS, "
              WS-E-RECORDS " " "E RECORDS, "
              WS-W-RECORDS " " "W RECORDS"
              INTO WS-MESSAGE.
       DISPLAY WS-MESSAGE.
       MOVE WS-N-AVERAGE TO WS-EDITED-N
       MOVE WS-S-AVERAGE TO WS-EDITED-S
       MOVE WS-E-AVERAGE TO WS-EDITED-E
       MOVE WS-W-AVERAGE TO WS-EDITED-W
       MOVE SPACES TO WS-MESSAGE.
       STRING "N AVE = " WS-EDITED-N ", "
              "S AVE = " WS-EDITED-S ", "
              "E AVE = " WS-EDITED-E ", "
              "W AVE = " WS-EDITED-W
              INTO WS-MESSAGE
       DISPLAY WS-MESSAGE
       CLOSE INFILE
       STOP RUN.


PROCESS-REC.
    EVALUATE TRUE
        WHEN NORTH
            ADD 1 TO WS-N-RECORDS
            ADD IN-TOTAL TO WS-N-TOTAL
        WHEN SOUTH
            ADD 1 TO WS-S-RECORDS
            ADD IN-TOTAL TO WS-S-TOTAL
        WHEN EAST
            ADD 1 TO WS-E-RECORDS
            ADD IN-TOTAL TO WS-E-TOTAL
        WHEN WEST
            ADD 1 TO WS-W-RECORDS
            ADD IN-TOTAL TO WS-W-TOTAL
    END-EVALUATE
    READ INFILE AT END MOVE 1 TO WS-EOF.
```

This produces output similar to the following:

```
Starting calculation program
0004 N records, 0006 S records, 0008 E records, 0006 W records
N ave =  10022.98, S ave =  41504.88, E ave =  84961.77, W ave =  36787.15
```

1. What is the reason for the ‘IF WS-n-RECORDS > 0’ tests in END-PARA?
2. We know that STRING does not pad the receiving field with spaces, so why does this program not clear out WS-MESSAGE after the first time it is used, before the second STRING?
3. As stated, the PIC Z fields will hold the entire output of the PIC 9(6)v99 fields, and will also show the decimal point. Does this mean they are the same size as the PIC 9 fields, or are they longer?

Optional Exercises

There are a number of other simple projects that you may choose to study, to gain further familiarity with the features we have described so far. These are found by switching to workspaces:

Copyright © Micro Focus 2015-2016. All rights reserved.

- **09_02_Initialize_replacing**
- **09_03_Inspect_Tallying**
- **09_04_String_Inspect_Unstring**

Quick Quiz

1. What are the five verbs that can be used to perform arithmetic calculations?
2. Three of the verbs share the same number of formats – what is this number?
3. Is rounding on or off by default?
4. What are the three string manipulation verbs in COBOL?
5. What verbs use DELIMITED BY?
6. What does the POINTER clause do?
7. Which of the following arithmetic statements are syntactically correct?
    a. ADD WS-ITEM-1 WS-ITEM-2 TO WS-ITEM-4 AND WS-ITEM-5.
    b. COMPUTE WS-RESULT = 2.667 * 5.4334 ROUNDED.
    c. MULTIPLY CEO-SALARY BY 1.1.
    d. ADD 2.3 -11 TO WS-RESULT-1 WS-RESULT-2.

Copyright © Micro Focus 2015-2016. All rights reserved.

10 Repeating Data

Introduction

Intelligent use of multi-dimensional tables can rapidly speed up program development and add clarity to application logic for maintenance programmers.

Here we become familiar with the COBOL constructs for defining and manipulating these tables.

Module Objectives

By the end of this module you will be able to:

- Explain the need for representation of repeating data in COBOL.
- Use subscripts and indexes to manipulate tables of such data.
- Use PERFORM with repeating data.
- Use different forms of the SEARCH verb to access a table.

Representing Repeating Data

A very common business need is to be able to access and manipulate groups, or occurrences, of data that are of the same size and type. For example, a monthly salesperson record may need to show:

- Name
- Salary
- 5 sets of weekly sales figures

This group of data repeats. These could be expressed as shown below:

```
01  MONTHLY-SALES     PIC 9(8)V99
01  SALES-RECORD.
    03  SALES-NAME     PIC X(20).
    03  SALES-SALARY  PIC 9(6)V99.
    03  SALES-WEEK-1  PIC 9(5)V99.
    03  SALES-WEEK-2  PIC 9(5)V99.
    03  SALES-WEEK-3  PIC 9(5)V99.
    03  SALES-WEEK-4  PIC 9(5)V99.
    03  SALES-WEEK-5  PIC 9(5)V99.
```

However, this does not express the whole truth (it does not show that there is a repeating field).

If we wanted to calculate the monthly sales figures, we could write code like:

```
       ADD SALES-WEEK-1 SALES-WEEK-2
           SALES-WEEK-3 SALES-WEEK-4
           SALES-WEEK-5
           GIVING MONTHLY-SALES
```

A better way to represent the data is in a table as follows, using the **OCCURS** clause:

```
01  MONTHLY-SALES     PIC 9(8)V99
01  SALES-RECORD.
    03  SALES-NAME     PIC X(20).
    03  SALES-SALARY  PIC 9(6)V99.
    03  SALES-WEEK     PIC 9(5)V99 OCCURS 5.
```


Copyright © Micro Focus 2015-2016. All rights reserved.

A table holds a set of different data items that have the same definition. The items in the table can be accessed using a reference to the item, called a *subscript*. A table is defined using the OCCURS clause in the DATA DIVISION. We could define the information in the previous code using a table, as above. We now have a much cleaner way of calculating the monthly sales figures:

```
01  WS-SUB    PIC 9.

    PERFORM VARYING WS-SUB FROM 1 BY 1 UNTIL WS-SUB > 5
       ADD SALES-WEEK(WS-SUB) TO MONTHLY-SALES
    END-PERFORM
```

- The PERFORM verb starts by setting WS-SUB to 1, and as the subscript is not greater than 5, SALES-WEEK(1) is added to MONTHLY-SALES (which should have been initialized at some point).
- The loop goes around again. WS-SUB is incremented automatically (BY 1) and the second occurrence is added. This goes on until SALES-WEEK(5) has been added.
- The next time around the UNTIL is now true, so the PERFORM stops.
- Although this code takes up as many lines on the page as the explicit adding of all five occurrences, it is far more logical.
- Furthermore, it would be very straightforward to modify both the record and the code to cope with 52 entries.

This is using a variation of the PERFORM statement that we have not yet seen. Instead of PERFORMing a paragraph or section, this version of the PERFORM is called an **in-line PERFORM** and performs the embedded code up to the END-PERFORM (This is COBOL, post 1985).

Keeping a subscript in range

Unlike in some programming languages, in COBOL, an array subscript starts at 1, not 0. So a subscript should never go below 1, or above the maximum number specified in the OCCURS.

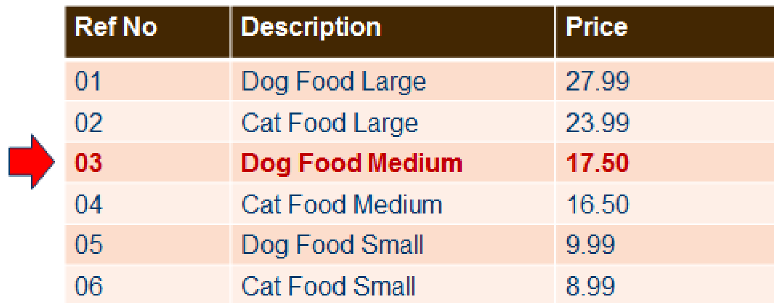
For example, when using OCCURS 10, the subscript pointing to the table should never go below 1 or exceed 10, or it will try and access memory used by something else. At runtime this could cause the program to crash.

A related error is to use a subscript that is too small. For example, a table that OCCURS 100 must have a subscript PIC 999. It is a common mistake to make the subscript PIC 99, which works perfectly until the subscript is 99 and 1 is added. The subscript then resets to zero. At the very least, the program will do something wrong.

Look up tables

Lookup tables are very common in COBOL programs.

Copyright © Micro Focus 2015-2016. All rights reserved.



Ref No	Description	Price
01	Dog Food Large	27.99
02	Cat Food Large	23.99
03	**Dog Food Medium**	**17.50**
04	Cat Food Medium	16.50
05	Dog Food Small	9.99
06	Cat Food Small	8.99

The program matches against the first column. It then reads values from the other columns.

The lookup table needs to use an OCCURS, and also VALUE clauses.

VALUE clauses **cannot** be used with OCCURS, but REDEFINES gets round this restriction. (of course, tables often get populated by reading from a data file). If the data above was to be hard coded in a COBOL program, then it would be done as follows:

```
01  WS-PETFOOD.
    03   PIC X(22) VALUE '01Dog Food Large  2799'.
    03   PIC X(22) VALUE '02Cat Food Large  2399'.
    03   PIC X(22) VALUE '03Dog Food Medium 1750'.
    03   PIC X(22) VALUE '04Cat Food Medium 1650'.
    03   PIC X(22) VALUE '05Dog Food Small  0999'.
    03   PIC X(22) VALUE '06Cat Food Small  0899'.
01  WS-PETFOOD-REDEF REDEFINES WS-PETFOOD.
    03  WS-PET-DETAILS OCCURS 6.
        05  WS-TABLE-NUM         PIC 99.
        05  WS-FOOD-DESCRIPTION PIC X(16).
        05  WS-PRICE             PIC 99V99.
```

Exercise 1

Switch to the workspace **10_01_Repeating_Data.**

There are 2 programs in this project. Study both of them and see the way in which repeating data is used.

Indexes

An index is another way of accessing a lookup table (it holds a relative address of the element it is looking at). Each index can only be associated with a single table. (So if you have multiple tables you will need to define a different index for each). You do not declare an index as a COBOL filed in data division. Instead by referring to it in the occurs definition, the declaration happens automatically.

Indexes have two advantages over subscripts:

1. Indexes hold a pointer to each table entry, providing more efficient access. (This is especially true in mainframe COBOL – less so more recently).
2. If an indexed table is sorted (as the pet food example was), then it can be searched very efficiently, using the so-called “binary chop” method.

Subscripts v Indexes

This table must be accessed using a subscript:

Copyright © Micro Focus 2015-2016. All rights reserved.

```
01  WS-PETFOOD-REDEF REDEFINES WS-PETFOOD.
    03  WS-PET-DETAILS OCCURS 6.
        05  WS-TABLE-NUM         PIC 99.
        05  WS-FOOD-DESCRIPTION  PIC X(15).
        05  WS-PRICE             PIC 99V99.
```

This next table may be accessed by an index or a subscript:

```
01  WS-PETFOOD-REDEF REDEFINES WS-PETFOOD.
    03  WS-PET-DETAILS OCCURS 6 indexed by pet-ix.
        05  WS-TABLE-NUM         PIC 99.
        05  WS-FOOD-DESCRIPTION  PIC X(15).
        05  WS-PRICE             PIC 99V99.
```

NOTE: To move a value to an index, do not use the **MOVE** statement; instead, use **SET**. We shall see this later.

Using the SEARCH verb

Provided that the table is indexed, you can use the SEARCH verb to find the entry you require.

- Use **SEARCH** for unsorted tables
- Use **SEARCH ALL** for sorted tables

SEARCH starts at the beginning of the table and searches to the end, which is not efficient across large tables.

SEARCH ALL is more efficient because it performs a binary search starting in the middle determining whether the value is greater or less than the item being searched. This splitting process continues until the **SEARCH ALL** is complete. To use **SEARCH ALL**, the table must be indexed and the elements of the table must be in **ascending or descending** sequence.

Exercise 2

Switch to the workspace **10_01_Repeating_Data2.**

There are 2 programs in this project, both using indexes. One program uses SEARCH and the other uses SEARCH ALL.

Study both of them and see the way in which repeating data is used. Look at them in the order:

- **PetFoodtablesearch.cbl**
- **PetFoodtablesearchall.cbl**

Modifying index values

Examples of changing the value of an index:

```
SET PET-IX TO 1
SET PET-IX UP BY 1
SET PET-IX DOWN BY WS-NUM
```

To set one index value to the value of another index:

```
SET IX-2 TO IX-1
```


Copyright © Micro Focus 2015-2016. All rights reserved.

Multi-dimensional tables

It is possible to have an OCCURS within an OCCURS (two-dimensional table), or even more dimensions, in COBOL. For example:

```
01  WS-SALES-DATA.
    03  WS-REGION                             OCCURS 10.
        05  WS-REGION-NAME        PIC X(20).
        05  WS-SALESPERSON                    OCCURS 20.
            07  WS-PERSONS-NAME   PIC X(20).
            07  WS-PERSONS-SALES  PIC 9(5)V99 OCCURS 52.
```

To access particular fields:

```
          WS-REGION(WS-SUB-1)
          WS-PERSONS-NAME(WS-SUB-1, WS-SUB-2)
          WS-PERSONS-SALES(WS-SUB-1, WS-SUB-2, WS-SUB-3)
```

You can have tables within tables, thus creating a multi-dimensional table. Multi-dimensional tables have multiple **OCCURS** fields. OCCURS fields might contain fields that might themselves occur, and so on.

The above example shows tables within tables. This provides information on ten regions, each with twenty salespeople and each of them having 52 sets of sales results. Everything is held under one 01 level.

Referencing data items in this sort of table is not difficult; for example WS-SALESPERSON will need two subscripts as shown above. The highest level index appears first. Use commas to make the text clearer if you wish.

Variable length tables

If your table has unknown length at compilation time, this size can be set at run time, using OCCURS...DEPENDING syntax. E.g.

```
01  OUTREC.
    03  OUT-NAME              PIC X(20).
    03  OUT-ADDRESS.
        05  OUT-ADDR-LINE     PIC X(20) OCCURS 3.
    03  OUT-COMMENT-COUNT     PIC 999.
    03  OUT-COMMENTS.
        05  OUT-COMMENT-BYTE PIC X OCCURS 0 TO 100  DEPENDING ON OUT-COMMENT-COUNT.
```

The OCCURS clause can be used to define a variable length table or field. Create a variable length table by using the DEPENDING ON clause in the OCCURS clause. The number of items in the table depends on a data item, which is used in the DEPENDING ON clause. When working with a variable length table, you must also specify the maximum and minimum number of items in the table in the OCCURS clause.

The following is an example of its use:

```
01  CUSTOMER-RECORD.
    03  CUSTOMER-NO           PIC 9(6).
    03  CUSTOMER-BALANCE      PIC S9(8)V99.
    03  CUSTOMER-INVOICE-CNT PIC 999.
    03  CUSTOMER-INVOICE    OCCURS 1 TO 500  DEPENDING ON CUSTOMER-INVOICE-CNT.
```


Copyright © Micro Focus 2015-2016. All rights reserved.

```
    05  INVOICE-NO        PIC 9(6).
    05  INVOICE-DATE      PIC 9(8).
    05  INVOICE-AMT       PIC 9(6)V99.

ADD 1 TO CUSTOMER-INVOICE-CNT
MOVE WS-INVOICE-NO  TO INVOICE-NO(CUSTOMER-INVOICE-CNT)
MOVE WS-TODAYS-DATE TO INVOICE-DATE(CUSTOMER-INVOICE-CNT)
MOVE WS-INVOICE-AMT TO INVOICE-AMT(CUSTOMER-INVOICE-CNT)
```

Module Summary

At end of this module you are now able to:

- Explain the need for representation of repeating data in COBOL
- Use subscripts and indexes to manipulate tables of such data
- Use PERFORM with repeating data
- Use different forms of the SEARCH verb to access a table

Exercise 3

There a couple of other brief projects that you may find useful. They are found by switching to workspaces:

- **10_03_Variable size array**
- **10_04_Table Indexing**

Quick Quiz

1. Would a PIC 99 subscript be suitable for an OCCURS 100 table?
    a. YES
    b. NO
2. What is the minimum value a subscript should ever contain while you are using it?
    a. 0
    b. 1
    c. 100
    d. There is no minimum
3. What is the difference between SEARCH and SEARCH ALL?
    a. SEARCH requires the table to be sorted
    b. SEARCHALL requires the table to be sorted
    c. There is no difference
4. What happens if you do not set the index on a non-sorted table SEARCH?
    a. You will not find the item you require
    b. You will start the search at an unknown place
    c. That is fine, no problem
5. What is the clause that tests for 'entry not found in the table'?
    a. IF FOUND
    b. IF data-item = . . .
    c. WHEN FOUND
    d. WHEN data-item = . . .
6. Both SET and MOVE can be used with indexes
    a. TRUE

Copyright © Micro Focus 2015-2016. All rights reserved.

   b. FALSE
7. Both SET and MOVE can be used with subscripts
   a. TRUE
   b. FALSE
8. An index can be used on more than one table
   a. TRUE
   b. FALSE
9. A subscript can be used on more than one table
   a. TRUE
   b. FALSE

Copyright © Micro Focus 2015-2016. All rights reserved.

11 Printing and Reports

Introduction

Any business programming system will almost certainly require report generation. Users frequently create reports from static data using interactive report-writing tools. Moving beyond this, COBOL provides very simple and flexible methods when the required data can be captured only during the program run itself. So print programs, which produce reports, rather than data files, are common.

Print programs present different challenges both in the design stage and the COBOL procedural coding.

This module examines the following main areas:

- The range of edited fields available in COBOL and their uses.
- The COBOL differences found in a print program.
- The design implications.

Module Objectives

By the end of this module you will be able to:

- List the different types of edited fields used for printing or display.
- List the COBOL considerations that are relevant when designing a print program.
- Set up heading, detail, and footer lines.
- Design a print program.

Edited Fields

Edited fields offer a range of ways to change a value (usually a number) so that it is easier to read, and more meaningful.

Originally intended for printed reports, edited fields are also useful for DISPLAYed fields

Edited fields fall into the following types:

- Handling leading zeros (with replacement), commas, and decimal points
- Currency symbols
- Plus (+) and minus (-) signs
- Credit and debit symbols
- Insertion characters such as / in a date or : in a time

Leading Zeros

Look at the following example:

```
01  WS-NO          PIC 9(6)V99.
01  WS-EDITED-NO   PIC Z(5)9.99.
```

The Picture clause has been changed to Z(5)9.99

Copyright © Micro Focus 2015-2016. All rights reserved.

In this example, this means that if there are any leading zeros in the data item, when it is displayed, up to 5 leading zeros are changed to spaces.

The use of the decimal point in this field, instead of the V means that the "implicit" decimal point has been replaced with a "real" one.

So if these data items contain any of the following values the result of displaying the value is:

Value/Picture	9(6)V99	Z(5)9.99
1234.56	00123456	1234.56
123456.78	12345678	123456.78
0	00000000	0.00
.34	00000034	0.34

A further example shows the picture strings 9(6)V99, Z(6).99, ZZZ999.99

Value/Picture	9(6)V99	Z(6).99	ZZZ999.99
1234.56	00123456	1234.56	1234.56
123456.78	12345678	123456.78	123456.78
0	00000000	0.00	000.00
.34	00000034	0.34	000.34
12.34	00001234	12.34	012.34

The main thing to note here is that numeric edited picture string, as illustrated above are **NOT** numeric fields and cannot be used in any calculations. Any calculation must be carried out on the numeric field and then the edited field used in any display statements by first doing something like:

```
MOVE WS-NO TO WS-EDITED-NO
DISPLAY WS-EDITED-NO
```

The MOVE is done, as we would hope, by aligning the data on decimal point.

Blank when zero

Normally we want to see when the first digit to the left of any decimal point is zero, hence Z(5)9.99 for example

If we want to suppress a zero number completely, we can use BLANK WHEN ZERO. E.g.

```
01  WS-EDITED-TOTAL  PIC Z(5)9.99 BLANK WHEN ZERO.
```

The same result for a zero could be achieved with:

```
01  WS-EDITED-TOTAL  PIC Z(6).ZZ.
```


Copyright © Micro Focus 2015-2016. All rights reserved.

However this will **not** give identical results; see below.

Value/Picture	9(6)V99	Z(5)9.99 BLANK WHEN ZERO	Z(6).ZZ
1234.56	00123456	1234.56	1234.56
123456.78	12345678	123456.78	123456.78
0	00000000		
.34	00000034	0.34	.34
.01	00000001	0.01	.01

Other leading characters

In some cases you may want to precede a numeric display item with asterisks. This is often used in check (Cheque) printing to help prevent fraud. So this is done with the picture clause:

```
01  WS-PROTECTED        PIC *(5)9.99.
```

Value/Picture	9(6)V99	*(5)9.99
1234.56	00123456	**1234.56
123456.78	12345678	123456.78
0	00000000	*****0.00
.34	00000034	*****0.34

Adding Commas and decimal point

Commas can be inserted into an edited field.

They 'disappear' like Zs if they are not needed.

They always take up a byte, whether they are visible or not, so WS-EDITED-BENEFITS always occupies thirteen bytes, regardless of the value it contains.

For example:

```
01  WS-BENEFITS          PIC 9(8)v99.
01  WS-EDITED-BENEFITS   PIC ZZ,ZZZ,ZZ9.99.
```

The following shows some examples:

Value/Picture	9(8)V99	ZZ,ZZZ,ZZ9.99
1234.56	0000123456	1,234.56
123456.78	0012345678	123,456.78
12345678.90	1234567890	12,345,678.90
0	0000000000	0.00
.34	0000000034	0.34

Currency symbols

Currency symbols are very similar to Zs.

Copyright © Micro Focus 2015-2016. All rights reserved.

The symbol will 'move up' to the first non-zero number, space filling to the left if appropriate

So always add one extra symbol, or you will lose a digit if the number field is full.

```
01  WS-BENEFITS          PIC 9(8)v99.
01  WS-EDITED-BENEFITS   PIC $$$,$$$,$$9.99.
```

Value/Picture	9(8)V99	$$$,$$$,$$9.99
1234.56	0000123456	$1,234.56
123456.78	0012345678	$123,456.78
12345678.90	1234567890	$12,345,678.90
0	0000000000	$0.00
.34	0000000034	$0.34

The actual currency symbol that is displayed is determined by a definition in the Environment Division.

This currency symbol is defined in a paragraph we have not discussed before: SPECIAL-NAMES

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
  CURRENCY SIGN IS '£'.
```

You can also swap the functions of decimal point and comma, as happens in some European countries.

For example:

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
   DECIMAL-POINT IS COMMA.
```

Plus and Minus Signs

You can define floating plus and minus signs as follows:

```
01  WS-SIGNED-NUMBER      PIC S9(8).
01  WS-ED-MINUS-1         PIC -(7)9.
01  WS-ED-PLUS-1          PIC +(7)9.
```

The results of the display of these data items would be:

Value/Picture	S9(8)	-(7)9	+7(9)
1234	00001234+	1234	+1234
-1234	00001234-	-1234	-1234

Copyright © Micro Focus 2015-2016. All rights reserved.

Credit and Debit Signs

The two fields CR and DB are both used to indicate negative values. These symbols are specifically available in COBOL numeric edited fields.

```
01  WS-CRED-1                PIC Z(7)9.99CR.
01  WS-DEB-1                 PIC Z(7)9.99DB.
```

The CR field is often for use on bills, to show when a customer owes a negative amount (i.e. when they are in credit). In both cases the CR or DB only appears when the field is negative.

Value/Picture	S9(8)	Z(7)9.99CR	Z(7)9.99DB
1234	00001234+	1234	1234
-1234	00001234-	1234CR	1234DB

Insertion characters

There are four insertion characters that you may wish to use

- The slash "/" character is commonly used with dates
- The colon "/" character is commonly used with time
- The "B" character allows you insert a space, or Blank
- It is also possible to insert a "0" into a field

Example 1

```
01  WS-DATE                PIC 9(8).
01  WS-EDITED-DATE         PIC 9999/99/99.
```

If WS-DATE contains 20120707 and is moved to WS-EDITED-DATE then WS-EDITED-DATE will contain 2012/07/07

Example 2

```
01  WS-CRED-CARD-NO         PIC X(16).
01  WS-EDITED-CRED-CARD-NO PIC X(4)BX(4)BX(4)BX(4).
```

If WS-CRED-CARD-NO contains 1234567890123456 and is moved to WS-EDITED-CRED-CARD-NO then WS-EDITED-CRED-CARD-NO will contain 1234 4568 9012 3456.

Example 3

```
01  WS-SOL                  PIC 999000.
```

If you execute the code:

```
   MOVE 300 TO WS-SOL
   DISPLAY 'Light Speed is: ' WS-SOL  ' mph'
```

You will get the result:

```
Light Speed is: 300000 mph
```


Copyright © Micro Focus 2015-2016. All rights reserved.

Coding a Print Program

Using print programs warrants different COBOL elements, including the following:

- Use of **LINE SEQUENTIAL** rather than **RECORD SEQUENTIAL** files (default)
- Output lines, to hold the different types of record you create

e.g.

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT INFILE   ASSIGN 'INFILE.DAT'
     ORGANIZATION IS SEQUENTIAL.
SELECT OUTFILE  ASSIGN 'OUTPRINT.TXT'
     ORGANIZATION IS LINE SEQUENTIAL.
```

COBOL includes another type of file, called the **LINE SEQUENTIAL** file, which is a PC text file. It can be read and modified by any PC text editor, such as Notepad.

Consequently, the file is described as LINE SEQUENTIAL in the Environment Division.

Up until now we have been using RECORD SEQUENTIAL files, which is the default setting. Such files as these are COBOL data files, and while you may be able to view them in a text editor, if you try to alter them, the layout will be damaged and can no longer be read by a COBOL program.

The following shows a typical example of a report file:

```
        REPORT DATED 2004/04/17     <---- Header
                                             1

 REGION: NORTHERN                   <---- New Region
                  MALCOLM ARNOLD                      12,345.67
                  CAMILLE SAINT-SAENS                 85,450.00
                  CLAUDE DEBUSSY                      37,267.00
                  ANTONIO VIVALDI                     66,660.00  <---- Detail
                  KARL DITTERS                        30,046.00
                  ROGER QUILTER                       92,929.00
                  PERCY GRAINGER                      44,449.44

                  TOTALS FOR THIS REGION ARE    369,147.11  <---- End of Region
```

Consider the range of output lines that need to be written. A print program will likely include a heading, a detail line, possibly a subtotal, and a grand total line.

The detail line consists of items from an input record and/or calculations, but the other lines show a mixture of preset values and items examined at runtime.

The preset items would best be coded with VALUE clauses; however, these cannot be used in the File Section.

Therefore it is normally best to include several versions of the print line in working storage, and WRITE the print record FROM the appropriate line.

Copyright © Micro Focus 2015-2016. All rights reserved.

Setting up a print line

A print line is defined as a record (for example) 132 characters. E.g.

```
FILE SECTION.
FD OUTFILE.
01 OUTREC  PIC X(132).
```

All output record types are set up in working storage, and then the WRITE...FROM verb is used to write out the appropriate type of record.

```
    WRITE OUTREC FROM WS-HEADING-LINE AFTER PAGE
    WRITE OUTREC FROM WS-DETAIL-LINE  AFTER 3
```

The AFTER clause positions the line on the page

An example of an output header line could be:

```
01  WS-HEADING-LINE.
    03                     PIC X(13) VALUE 'REPORT DATED'.
    03  WS-EDITED-DATE     PIC 9999/99/99.
    03                     PIC X(60) VALUE SPACES.
    03  WS-PRINT-PAGE-CNT PIC Z9.
```

Note

- The use of FILLERs and VALUE clauses
- The heading line is a mixture of these fields and of variable data (the date, and the page counter)
- The variable parts use edited fields

An example of a detail line could be:

```
01  WS-DETAIL-LINE.
   03                      PIC X(20) VALUE SPACES.
   03  WS-PRINT-NAME       PIC X(40).
   03  WS-PRINT-SALES      PIC ZZZ,ZZ9.99.
```

The FILLER is there purely to provide the correct spacing

Other fields are supplied, or calculated elsewhere and moved here, at runtime

Writing a print line

Examples of the syntax used to write a print line are:

```
   1. WRITE OUTREC FROM WS-HEADING-LINE AFTER PAGE
   2. WRITE OUTREC FROM WS-DETAIL-LINE  AFTER 2
   3. WRITE OUTREC FROM WS-DETAIL-LINE
```

Example 1 shows writing a heading line on a new page

Example 2 shows writing a detail line two lines below the current position (leave a blank line)

Example 3 shows writing a detail line on the next line

Copyright © Micro Focus 2015-2016. All rights reserved.

Designing a Print Program

The design will be driven largely by the format of the output report file.

```
        REPORT DATED 2004/04/17                                                  1

REGION: NORTHERN
                    MALCOLM ARNOLD                          12,345.67
                    CAMILLE SAINT-SAENS                     85,450.00
                    CLAUDE DEBUSSY                          37,267.00
                    ANTONIO VIVALDI                         66,660.00
                    KARL DITTERS                            30,046.00
                    ROGER QUILTER                           92,929.00
                    PERCY GRAINGER                          44,449.44

                    TOTALS FOR THIS REGION ARE    369,147.11
```

Header

New Region

Detail

End of Region

A print page typically consists of a number of pages (which, theoretically, may be zero).

A print page might include a header page, followed by “normal” pages, which will probably display some sort of heading.

In a simple report program, the heading precedes multiple detail lines, that is, lines that consist of data, probably drawn from records read in or calculations made at runtime.

When the program writes a specified number of detail lines, a new page begins, following the same format.

The program might produce subtotals at, say, a change of region.

Finally, the program might include grand totals or other one-off messages or calculations at the end of the report.

The design challenge exists on how to reconcile this fairly complex structure with the simple structure of an input file.

Copyright © Micro Focus 2015-2016. All rights reserved.

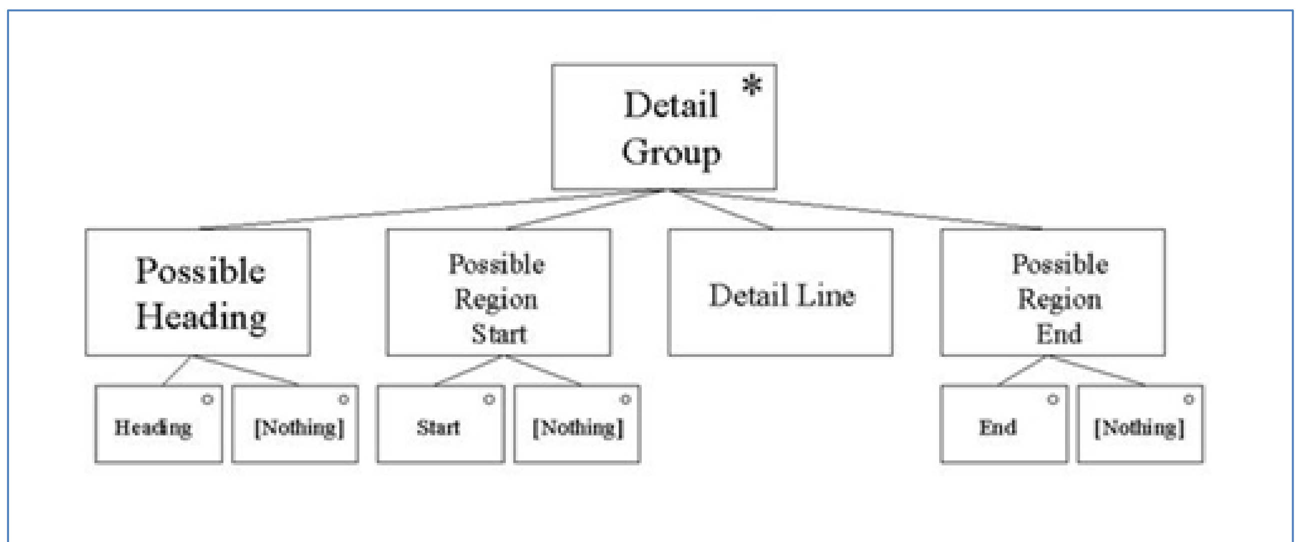
- If there are subheadings and subtotals (for the region in the example below), how and when do you test for a new region?
- If there are subheadings, how do you get the first one printed?
- If there are subtotals, how do you get the last one printed?

```
        REPORT DATED 2004/04/17                                    1

REGION: NORTHERN
                   MALCOLM ARNOLD                       12,345.67
                   CAMILLE SAINT-SAENS                  85,450.00
                   CLAUDE DEBUSSY                       37,267.00
                   ANTONIO VIVALDI                      66,660.00
                   KARL DITTERS                         30,046.00
                   ROGER QUILTER                        92,929.00
                   PERCY GRAINGER                       44,449.44

                   TOTALS FOR THIS REGION ARE   369,147.11
```

A possible view of a print program is shown below:



This view may at first seem contrived; however, it avoids any conflict between input and output files. “Pages” as such have disappeared; only an iteration of “detail group” exists.

This ensures that detail lines display and also other events – headings or potential start-of or end-of regions – occur automatically. A sample program, later in this module, uses this approach.

Two special points exist at which we must make sure that particular code takes place: at the very start to ensure that the first heading appears and at the end to make sure that the last region finishes.

We could achieve this in two ways. Firstly, the special code could be explicitly called in each case.

Alternatively, the program could force special values into fields at this time.

Using Report Writer

If you are producing many reports you may want to make use of the **COBOL Report Writer** facility.

Copyright © Micro Focus 2015-2016. All rights reserved.

This mechanism is very powerful for producing reports with headings footings, subheadings, sub footings, totals and subtotals.

Report writer is not used very often these days, since there are many other report writing mechanisms, outside the scope of this course. So for the purpose of this training course, this is not covered in any detail.

We have supplied a workspace **11_02_Report_Writer** which contains a program taken from a standard program supplied by the ANSI COBOL committee in 1983. It was subsequently modified by Jay Moseley in 2008. http://www.jaymoseley.com/hercules/compiling/crwex06o.htm

You may want to look at this.

You will find fuller examples at the following:

- http://www.pgrocer.net/Cis52/rptwritr.html
- ftp://ftp.software.ibm.com/software/websphere/awdtools/cobolreportwriter/c2643013.pdf

Module Summary

At the end of this module you are now able to:

- List the different types of edited fields used for printing or display
- List the COBOL considerations that are relevant when designing a print program
- Set up heading, detail, and footer lines
- Design a print program

Exercise

You should first switch to the workspace **11_01_Printing.sln**.

The program deals with an input file, consisting of records each with

- a one-character region code.
- a twenty-character field for sales person name.
- an eight-digit numeric field (six digits before the decimal point, two after) for that person's sales.

```
03  IN-REGION   PIC X .
03  IN-NAME     PIC X(20) .
03  IN-SALES    PIC 9(6)V99 .
```

The records will arrive in sorted order, that is, all records of the same region will be grouped together. The print report output looks something like the following.

Copyright © Micro Focus 2015-2016. All rights reserved.

```
      REPORT DATED 2004/04/17                                        1


REGION: NORTHERN
                    MALCOLM ARNOLD                          12,345.67
                    CAMILLE SAINT-SAENS                     85,450.00
                    CLAUDE DEBUSSY                          37,267.00
                    ANTONIO VIVALDI                         66,660.00
                    KARL DITTERS                            30,046.00
                    ROGER QUILTER                           92,929.00
                    PERCY GRAINGER                          44,449.44


                    TOTALS FOR THIS REGION ARE    369,147.11


REGION: SOUTHERN
                    ANTONIO ROSETTI                          7,725.68
                    JOSEF KRAUS                            102,938.47
                    ANTON BRUCKNER                          57,483.92
                    SIEGFRIED WAGNER                        18,356.74
                    PIOTR TCHAIKOVSKY                       98,201.80


                    TOTALS FOR THIS REGION ARE    284,706.61
```

You should first of all run the program and look at the results. The results can be seen by looking at the output text file **outprintprt.txt**. This file is shown in the solution explorer and can be opened directly from there, by double-clicking.

You should now examine the COBOL code and debug that code to see how it operates.

Consider the following points about the program.

- The program ACCEPTs the date in and moves to the heading line. When the program finds a new region, the program SEARCHes for the region in the table WS-REGION-TABLE-REDEF. (If an unknown region is found, the program terminates immediately.) The program moves the value found to the "start of region" line. The value found is also stored, so that "new region" is not encountered on every record.
- Normally, "start of region" immediately follows "end of region;" however, this is not true at the very start of the program. This explains the pre-setting of WS-REGN to LOW-VALUES.
- The program has to make sure that a page is thrown and headings are printed, at the end of every page, and at the very start of the program. How is this achieved?
- Is 'end of region' always indicated by the start of a new region?

Quick Quiz

1. What is the another way of writing PIC Z,ZZZ,ZZZ.ZZ?
   a. PIC ZZZ,(3)
   b. PIC Z,ZZZ,(3)VZZ
   c. PIC Z,Z(3),Z(3).Z(2)
2. Difference between a floating plus and a floating minus? Which of the following is true?
   a. No difference
   b. A minus always shows in both cases if the field is negative
   c. A plus always shows in both cases if the field is negative
   d. A plus always shows only if the field is positive in the case of floating plus
3. Why is B so called?

Copyright © Micro Focus 2015-2016. All rights reserved.

   a. B stands for Blank
   b. B stands for Bold
   c. B has no real meaning
4. Why must you always insert one more currency symbol than you need for the size of the field?
   a. To allow for longer data values
   b. To make space for the currency symbol if the field is full
   c. No need to add an extra character in the field
   d. Currency symbols require extra storage space than normal numeric characters
5. What organization would you expect for a print file?
   a. SEQUENTIAL
   b. TEXT
   c. LINE SEQUENTIAL
   d. INDEXED
6. How many blank lines do you get if you WRITE AFTER 4?
   a. 1
   b. 2
   c. 3
   d. 4
   e. 5
7. How do you make the print program throw a page?
   a. Use BEFORE PAGE THROW
   b. Use AFTER PAGE
   c. Use AFTER PAGE THROW
   d. Use AFTER 25 LINES

Copyright © Micro Focus 2015-2016. All rights reserved.

12 Using Indexed Files

Introduction

The data file handling capability within COBOL is extremely powerful.

Any business application programmer's skill set should include a full understanding of how to take advantage of these capabilities.

Up until now we have been using sequential files, where records are accessed, or written, one after another.

It is not easy to insert a new record, or delete a record, in the middle of such a file.

Sequential files are very common, but not suitable for applications needing to find a particular record, perhaps to amend or delete it.

The only way to select a chosen record on a sequential file is to read through the whole file until the correct one is found – hardly an efficient or easy process.

Similarly, inserting a new record into a sequential file might prove difficult.

Indexed files, on the other hand, allow more flexible access.

Finding any record quickly by using its ***key*** becomes easier.

Module Objectives

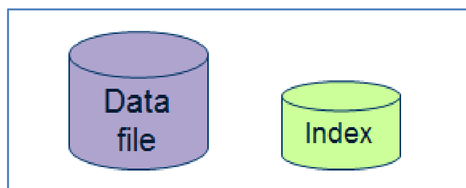
By the end of this module you will be able to:

- Explain the uses of indexed files
- Describe the different key structures
- List the different methods of access and use the associated COBOL statements
- Explain the use of File Status and Declaratives

Indexed File Structure

Each record has a primary key (which is almost always unique) and optional secondary (or alternate) keys.

A standard indexed file often consists of two physical files, the data file and the index file.



Other variations on an indexed file are for the Data part and the index part to be stored in the same physical file.

Copyright © Micro Focus 2015-2016. All rights reserved.

The file handler looks in the index part to determine which record of the data file to access.

You, the COBOL programmer, don't need to know the details of how this is done – you just 'go and get the record'

Indexed File Keys and Structure

Each record on an indexed file has its own *primary* key, which is stored as a data field (or fields) in the record.

If an indexed file contains, for example, employee records, a suitable key would be employee number. This is the main record key or *primary* key. We shall see later in this module how to use this key. This prime key is almost always unique, but COBOL does support non-uniqueness of prime key. Speaking personally I have never come across a useful non-unique prime key.

Other keys *or alternate keys* can exist on an indexed file record that can be used instead. These alternate keys may or may not be unique.

Again considering an employee file, a suitable alternate key would be employee surname. To find an employee called Smith where the employee number is not known, use the alternate key to find the appropriate key. There may be several Smiths to search until the program discovers the right one.

An indexed file often consists of two distinct parts: one part contains the data itself similar to a sequential file, and the other part contains the index information, which is used to access the right part of the data.

In practice, this may never be an issue. Yet, some situations might arise where this becomes important. For example, on some operating systems and within some versions of COBOL, these two parts are in fact two separate physical files.

So, to move the file to a different place, move both parts. Similarly, to calculate the amount of disk storage required for a certain file, consider the size of the index information for each record.

Thankfully, knowledge of this is not required when creating these files. The COBOL indexed file run-time system automatically creates them and your program will have only a logical view of a single file.

Accessing an indexed file

You can access an indexed file in one of three ways:

- **Random access** – go anywhere in the file, to insert, amend or delete a specific record
- **Sequential access** – read through, or write to, part or all of the file. The records will be processed in key order.
- **Dynamic access** – carry out both random and sequential access in the same program. A program that updates certain records on an employee file and then reads sequentially through all of them to produce paychecks, would be an example of the dynamic access.

File definition

The select statement for an indexed file contains information as shown below.

Copyright © Micro Focus 2015-2016. All rights reserved.

```
SELECT EMPLOYEE-FILE ASSIGN 'EMPLOYEE.DAT'
      ORGANIZATION IS INDEXED
      ACCESS IS RANDOM
      RECORD KEY IS EMPLOYEE-NO
      ALTERNATE RECORD KEY IS EMPLOYEE-SURNAME.
```

The `ACCESS IS RANDOM` can be replaced with:

`ACCESS IS SEQUENTIAL` or

`ACCESS IS DYNAMIC` as defined above

If ACCESS is SEQUENTIAL, then the whole clause can be omitted (since it is the default). However, it is better to leave the clause there for clarity.

The RECORD KEY refers to the unique primary key on the record that can be used to access any particular record.

The ALTERNATE RECORD KEY refers to the alternate key on the record that can be used to access any particular record.

The indexed file key (referred to as RECORD KEY) can be a compound of more than one field.

It is often stated that the key must be alphanumeric, meaning a PIC X elementary field or a group field.

This is not strictly true, but a MOVE to the key field (which is how the key is set) will always be treated as an alphanumeric move.

So, it is recommended that the key is alphanumeric. If the key really does have to be a number, then a redefinition can be used to “cheat.”

In this example the FD for the file could contain:

```
FD EMPLOYEE-FILE.
01  EMPLOYEE-REC.
    03  EMPLOYEE-NO         PIC X(5).
    03  EMPLOYEE-INITS      PIC X(4).
    03  EMPLOYEE-SURNAME    PIC X(16).
    03  EMPLOYEE-SALARY     PIC 9(6)V99.
    03  EMPLOYEE-DEPT       PIC X(10).
```

If the key field is actually numeric, this could be dealt with as shown below:

```
FD EMPLOYEE-FILE.
01  EMPREC.
    03  EMPLOYEE-NO.
        05  EMPLOYEE-NO-NUMERIC PIC 9(5).
    03  EMPLOYEE-INITS          PIC X(4).
    03  EMPLOYEE-SURNAME        PIC X(16).
    03  EMPLOYEE-SALARY         PIC 9(6)V99.
    03  EMPLOYEE-DEPT           PIC X(10).
```

The key is now EMPLOYEE-NO a group field, so it automatically has an alphanumeric picture.

Random Access

To open a file for Random access, you must first define the file as:

Copyright © Micro Focus 2015-2016. All rights reserved.

```
SELECT EMPLOYEE-FILE ASSIGN 'EMPLOYEE.DAT'
      ORGANIZATION IS INDEXED
      ACCESS IS RANDOM
      RECORD KEY IS EMPLOYEE-NO
      ALTERNATE RECORD KEY IS EMPLOYEE-SURNAME.
```

Or `ACCESS IS DYNAMIC`

The file OPEN statement is then:

```
OPEN I-O EMPLOYEE-FILE
```

There are three possible actions you may want to carry out:

- Amend an existing record
- Create a new record
- Delete an existing record

Amend an existing record

Firstly, find the record you wish to amend using the correct value in the key:

```
       MOVE IN-EMP-NO TO EMPLOYEE-NO.
       READ EMPLOYEE-FILE KEY IS EMPLOYEE-NO
              INVALID KEY
                     DISPLAY 'RECORD NOT FOUND'
                     STOP RUN
       END-READ.
```

Here we have moved the key value we want to the key field, and then carried out a READ.

If the record is not found, the INVALID KEY clause is invoked.

An AT END clause would have no relevance here (Why?).

Having retrieved the record, we can change any field or fields, except the **key field.**

We then REWRITE the record:

```
       REWRITE EMPREC KEY IS EMPLOYEE-NO
              INVALID KEY
                     DISPLAY 'FAILURE TO REWRITE '
                     STOP RUN
       END-REWRITE
```

Changing the key field would mean we were now dealing with a different record, so a REWRITE would not be appropriate.

The AT END clause ("no more records on the file") is not appropriate here.

Instead, use the INVALID KEY clause, which means "if the record specified with this key does not exist."

Assuming that the record has been found, all of the fields but one can be changed as appropriate. (The field that cannot be changed is the key.) The record is then written back to the file with REWRITE, as shown above.

Copyright © Micro Focus 2015-2016. All rights reserved.

Again INVALID KEY can be used to check that the REWRITE was successful. In this case, the program retrieved the record before rewriting it, as we needed the EMPLOYEE-SALARY value.

Because of that, the REWRITE should definitely work.

Create a new record

To create a new record, set up all the fields (including the key), and WRITE

```
MOVE '12345' TO EMPLOYEE-NO.
*> [set up other fields]_
WRITE EMPREC KEY IS EMPLOYEE-NO
       INVALID KEY
              DISPLAY 'CANNOT INSERT RECORD '
              STOP RUN
END-WRITE
```

If a record, with that key, already exists on the file, the INVALID KEY clause will be triggered

Delete an existing record

To delete a record, specify the key, and use the DELETE statement:

```
MOVE IN-EMP-NO TO EMPLOYEE-NO
DELETE EMPLOYEE-FILE KEY IS EMPLOYEE-NO
       INVALID KEY
              DISPLAY 'FAILURE TO DELETE'
              STOP RUN
END-DELETE
```

If the record does not exist on the file, the INVALID KEY clause will be triggered

Note that the clause is **DELETE [file name]** rather than **DELETE [record name]**

The KEY IS phrase

The phrase `KEY IS EMPLOYEE-NO` is needed if the file contains more than one key. If the file only has a single key, then this phrase can be omitted.

Closing the indexed file

The close is exactly the same as we have seen for sequential files:

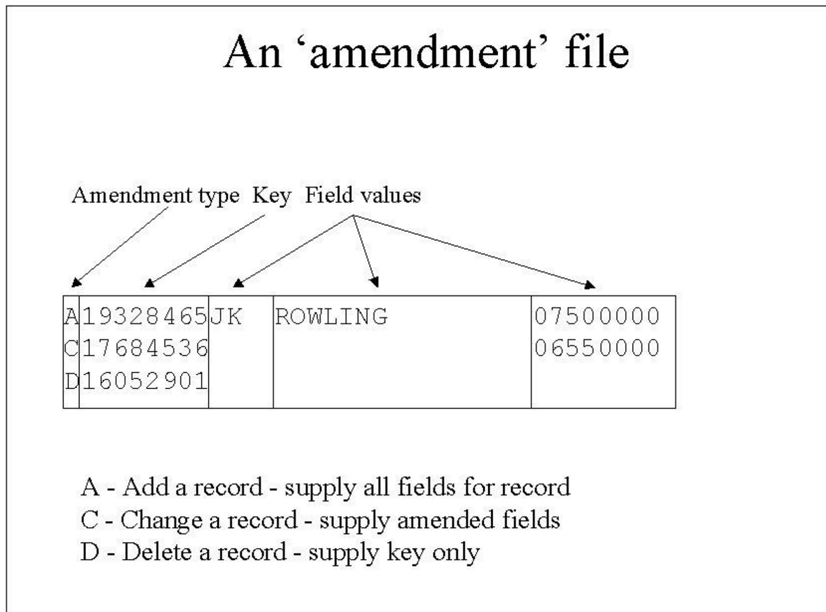
```
CLOSE EMPLOYEE-FILE
```

Exercise 1

For this exercise you should switch to the workspace **12_01_Indexed_Files.**

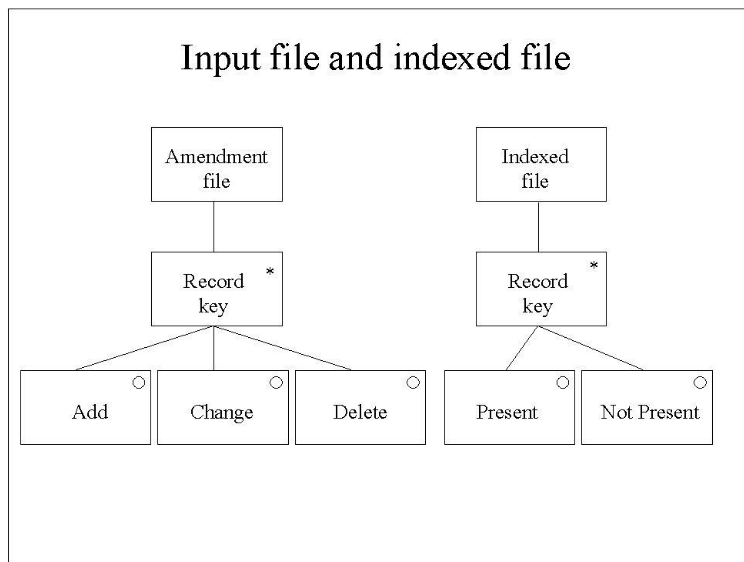
Most random-access programs need similar functionality. An indexed file needs changing, in that one or more records will be inserted, changed or deleted. One method for these amendments is to use a sequential input file containing details of the changes to be made. The following example uses a sequential input file.

Copyright © Micro Focus 2015-2016. All rights reserved.



The first byte of the amendment file can be interrogated to identify the type of amendment to which the record refers. If it is an addition, then all values need to be supplied. If a change, then perhaps only the changed fields will be specified (alternatively, they could all be supplied). If this is a delete, then all that is necessary is the record key.

Here is a view of both the amendment file and the indexed file.



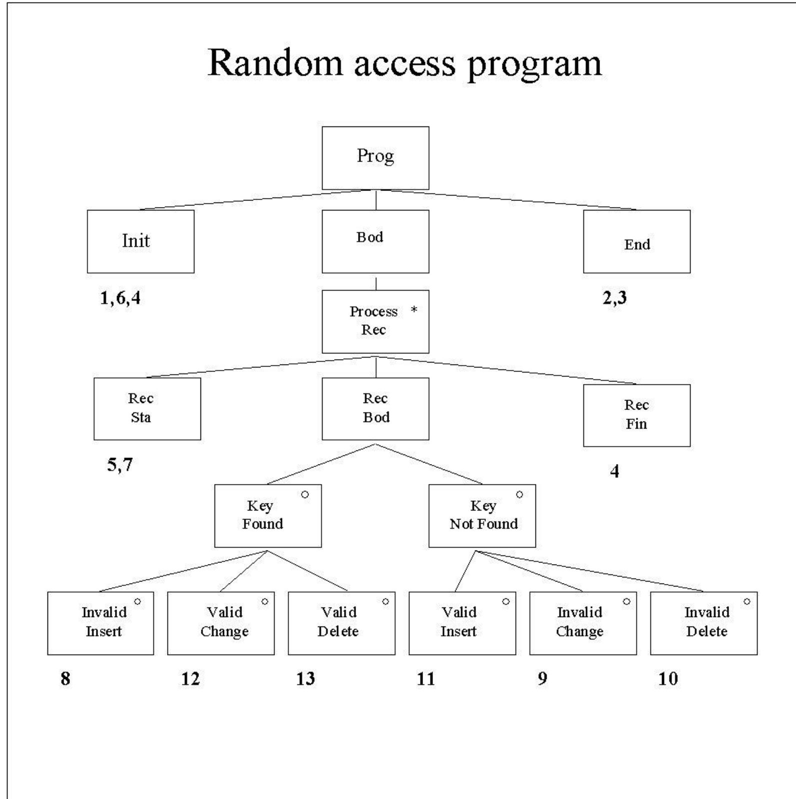
Each value in the amendment file will indicate an Add, Change or Delete. The key value may or may not exist on the indexed file. This yields six possibilities.

- The key value does exist, and this is an Add. This is an Invalid Add.
- The key value does exist, and this is a Change. This is a Valid Change and can be carried out.
- The key value does exist, and this is a Delete. This is a Valid Delete and can be carried out.
- The key vale does not exist, and this is an Add. This is a Valid Add and can be carried out.
- The key value does not exist, and this is a Change. This is an Invalid Change.
- The key value does not exist, and this is a Delete. This is an Invalid Delete.

Copyright © Micro Focus 2015-2016. All rights reserved.

The following program shows one way of tackling these choices. In “Record Start” the program seeks the key value on the indexed file, regardless of the amendment type. If the record does not exist, then the program sets a flag. (The flag must be cleared at a suitable point.) By “Record Body” the program knows whether or not the record exists and can act accordingly.

The actions and then the COBOL code appear after the program structure.



Actions

1. **Open Files**
2. **Close Files**
3. **Stop Run**
4. **Read amendment file**
5. **Clear found-it flag**
6. **Clear any other flags**
7. **Try and find indexed record**
8. **Display ‘Invalid Insert’**
9. **Display ‘Invalid Change’**
10. **Display ‘Invalid Delete’**
11. **Insert record**
12. **Rewrite amended record**
13. **Delete record**

You should study this program in debug mode to see how the program is behaving.

Copyright © Micro Focus 2015-2016. All rights reserved.

Accessing an indexed file sequentially

An indexed file can be accessed sequentially using any of its keys. In this example we will just use the prime key for now.

The SELECT...ASSIGN statement is used to set the access mode

```
SELECT EMPLOYEE-FILE ASSIGN 'EMPLOYEE.DAT'
      ORGANIZATION IS INDEXED
      ACCESS IS SEQUENTIAL
      RECORD KEY IS EMPLOYEE-NO
      ALTERNATE RECORD KEY IS EMPLOYEE-SURNAME.
```

You may still open the file **I-O**, allowing you to amend or delete records as you go.

To read the next record:

```
READ EMPLOYEE-FILE NEXT
      AT END MOVE 1 TO WS-EOF
END-READ
```

To change a record:

```
COMPUTE EMPLOYEE-SALARY = EMPLOYEE-SALARY * 1.01.
REWRITE EMPREC
```

To delete a record

```
DELETE EMPLOYEE-FILE
```

The START statement

Very often you want to start a sequential read at a specific point in an indexed file. The syntax to achieve this is:

```
MOVE '00100' TO EMPLOYEE-NO.
START EMPLOYEE-FILE KEY IS EMPLOYEE-NO
       KEY NOT < EMPLOYEE-NO
END-START
```

If there is a risk that the START will fail (if all the records have a smaller key), add an INVALID KEY clause to the statement.

Again the KEY IS phrase is only needed if the file contains more than 1 key.

START does not read, it merely positions.

The next sequential read, reads the next record.

```
READ EMPLOYEE-FILE NEXT
   AT END MOVE 1 TO WS-EOF
END-READ
```

This also moves the file record pointer forward by one.

Copyright © Micro Focus 2015-2016. All rights reserved.

The additional syntax (not available in some compilers) provides the reading the previous record

```
READ EMPLOYEE-FILE PREVIOUS
   AT END MOVE 1 TO WS-EOF
END-READ
```

Accessing an indexed file dynamically

An indexed file can be accessed both randomly and sequentially in the same program using any of its keys. The SELECT...ASSIGN statement is used to set the access mode

```
SELECT EMPLOYEE-FILE ASSIGN 'EMPLOYEE.DAT'
       ORGANIZATION IS INDEXED
       ACCESS IS DYNAMIC
       RECORD KEY IS EMPLOYEE-NO
       ALTERNATE RECORD KEY IS EMPLOYEE-SURNAME.
```

A dynamic access program consists of two distinct parts:

- the portion of the code where random accessing is carried out
- then sequential processing.

Treat these almost as two separate programs.

This allows 2 kinds of access

Random read

```
MOVE IN-EMP-NO TO EMPLOYEE-NO.
READ EMPLOYEE-FILE KEY IS EMPLOYEE-NO
        INVALID KEY
               DISPLAY 'RECORD NOT FOUND'
               STOP RUN
END-READ.
```

Sequential Read

```
READ EMPLOYEE-FILE NEXT
   AT END MOVE 1 TO WS-EOF
END-READ
```

It is common in a dynamic access program for the random element to come first, followed by the whole file being processed sequentially.

To get back to the beginning of the file (to begin sequential READs), we can use START:

```
MOVE LOW-VALUES TO EMPLOYEE-NO
START EMPLOYEE-FILE KEY IS EMPLOYEE-NO
        KEY NOT < EMPLOYEE-NO
END-STARTR
READ EMPLOYEE-FILE NEXT
        AT END MOVE 1 TO WS-EOF
END-READ
```

Using Alternate Keys

It is often desirable to have alternate keys on an indexed file.

Copyright © Micro Focus 2015-2016. All rights reserved.

In the case of “employee.dat,” alternate keys could help find a particular employee called “SMITH” if we did not know the primary key (or if the key we had was incorrect for some reason).

The SELECT…ASSIGN statement would look like:

```
SELECT EMPLOYEE-FILE ASSIGN 'EMPLOYEE.DAT'
      ORGANIZATION IS INDEXED
      ACCESS IS DYNAMIC
      RECORD KEY IS EMPLOYEE-NO
      ALTERNATE KEY IS EMPLOYEE-SURNAME WITH DUPLICATES.
```

In the case of “employee.dat,” alternate keys could help find a particular employee called “SMITH” if we did not know the primary key (or if the key we had was incorrect for some reason).

In this example, it is almost certain that the alternate key will have duplicates.

Alternate Key READ

To read a file using the alternate key, the syntax would be:

```
       MOVE 'SMITH' TO EMPLOYEE-SURNAME.
       READ EMPLOYEE-FILE KEY IS EMPLOYEE-SURNAME
              INVALID KEY
                     DISPLAY 'NO SURNAME'
       END-READ
```

To read the NEXT record, the syntax would be:

```
       READ EMPLOYEE-FILE NEXT
              AT END MOVE 1 TO WS-EOF
       END-READ
```

You will need to check yourself when the surname changes

AT END will only be triggered when the real EOF is encountered.

File Errors using file status clause

The AT END and INVALID KEY clauses are used to deal with 'expected' file behavior.

By default, the COBOL runtime deals with anything else that goes wrong.

It may be necessary to deal with other file errors explicitly.

For this, we need to use a **file status** field. E.g.

```
SELECT CHANGES-FILE  ASSIGN 'CHANGES.DAT'
  FILE STATUS WS-CHG-STATUS.
SELECT EMPLOYEE-FILE ASSIGN 'EMPLOYEE.DAT'
  ORGANIZATION IS INDEXED
  ACCESS IS RANDOM
  RECORD KEY EMPLOYEE-NO
  FILE STATUS WS-EMP-STATUS.
```

Where the status fields are defined as 2 byte fields.

```
01  WS-STATUSES.
```


Copyright © Micro Focus 2015-2016. All rights reserved.

```
03  WS-CHG-STATUS     PIC XX.
03  WS-EMP-STATUS     PIC XX.
```

Once you have defined file status for a file, then this must be used instead of AT END and INVALID key etc.

If the file operation is successful then the file status value will contain “00”

If there is an error, then certain other values will apply. E.g.

- "10" indicates "No next logical record" (equivalent of AT END)
- "22" indicates "Duplicate key condition" (trying to store a duplicate record when duplicates are not allowed)
- "23" indicates "No record found" (equivalent to INVALID KEY)

There are many other status code values.

If the first byte of the status code contains ‘9’ then the second byte will contain further information. To get to this information you will need to redefine the second byte as follows:

```
03  WS-EMP-STATUS     PIC XX.
03  WS-EMP-STATUS-EXTRA REDEFINES WS-EMP-STATUS.
    05  WS-EMP-STATUS-1 PIC X.
    05  WS-EMP-STATUS-2 PIC 99 COMP.
```

The values in this second byte will vary with whatever complier you are using. If you are using the Micro Focus compiler, then you would need to look in the help system to determine what these values are. For example a value of 9/65 indicates the file is locked. (Usually because some other user has opened the file for i-o).

Beware, that if you use file-status then you will need to recode the end-of-file; as shown below:

```
READ EMPLOYEE-FILE KEY IS EMPLOYEE-SURNAME
       INVALID KEY
              DISPLAY 'NO SURNAME'
END-READ
```

Will become

```
READ EMPLOYEE-FILE KEY IS EMPLOYEE-SURNAME
IF WS-EMP-STATUS NOT = '00'
       DISPLAY 'NO SURNAME'
END-IF
```

Use of Declaratives

Declaratives are *sections* of code, which if used, must appear first in the Procedure Division by inserting the word DECLARATIVES. (Note *SECTIONS*, not PARAGRAPHS.)

A Declaratives section will be PERFORMed automatically if a file status error occurs. Errors that you have specifically coded for, such as AT END, do *not* trigger Declaratives. This allows you to handle any other errors in file i-o which you have not specifically coded.

Copyright © Micro Focus 2015-2016. All rights reserved.

If using Declaratives, make sure that the rest of the Procedure Division code is also in a section and not the same one as the Declarative.

Sections consist of one or more paragraphs; by convention the last paragraph may be an exit paragraph, marking the end of the section. (The real end of the section is marked by the fact that another section has been found or by the physical end of the code.)

Example of Declarative:

```
PROCEDURE DIVISION.
DECLARATIVES.
CHANGES-ERROR SECTION.
     USE AFTER STANDARD ERROR PROCEDURE ON INFILE.
CHANGES-ERROR-ACTIONS.
    DISPLAY 'SERIOUS ERROR ON INPUT FILE '
            'STATUS IS ' WS-IN-STATUS
    CLOSE CHANGES-FILE
    STOP RUN.
*>[other Declaratives sections]

MAIN SECTION.
PARA.
 PERFORM INIT-PARA. *> ETC
```

Module Summary

At the end of this module you are now able to:

- Explain the uses of indexed files
- Describe the different key structures
- List the different methods of access and use the associated COBOL statements
- Explain the use of File Status and Declaratives

Further Exercises

There a number of solutions that you can look at to help re-enforce your knowledge of the use of indexed files and sequential files.

Exercise 2

Switch to the workspace **12_03_Indexed File update**

This project read a sequential file and applies appropriate update to and indexed file

Study this program and debug it to view the processing

Exercise 3

Switch to the workspace **12_04_Meaningful call with indexed file**

The project here shows one program “calling” another program. This is discussed in some detail in the next module. In this program you can also see a Microfocus extension to the display and accept

Copyright © Micro Focus 2015-2016. All rights reserved.

verbs, which allow you to position fields at certain positions on the screen. For now, just debug through the code to see the file handling in action. (Suitable code to use are 102 – 125)

Exercise 4

Switch to the workspace **12_05_Logic errors**

This project is not handling an indexed file but a sequential file with 2 record types. The program here is reading the input file **test44.dat** and producing a report **test44.txt**. However it contains 4 bugs.

You should use the knowledge you have gained so far to fix these 4 bugs.

The input file contains the following records:

```
JOE SMITH                45 ELM STREET             5663342   010166    197
ACCOUNTING I             BUS01      3                                   2
ENGLISH I                ENG01      3                                   2
AMERICAN HIST I          HIS01      3                                   2
SAM SHULTZ               24 MAPLE AVENUE           8663456   041170    197
ACCOUNTING II            BUS03      3                                   2
ENGLISH I                ENG01      3                                   2
AL JONES                 78 MAIN STREET            7349876   080670    197
MATH OF FINANCE          MTH03      3                                   2
HIST. OF SCIENCE         HIS11      3                                   2
ENGLISH LIT II           ENG06      3                                   2
BASKET WEAVING III       EASY1      3                                   2
POLKA DANCE II           EASY2      1                                   2
SALLY SWELL              8 APPIAN WAY              7654567   120741    197
EARLY CHILD. DEV         EDU03      3                                   2
EDUCATION IN AMER.       EDU04      3                                   2
COMPUTERS IN EDU.        EDU08      3                                   2
HAROLD HARDWORK          333 PLATT STREET          1112345   090871    197
CALCULUS IV              MTH91      3                                   2
ADVANCED PHYSICS         PHY92      3                                   2
NUCLEAR PHYSICS          PHY93      3                                   2
```

You will see there are 2 kinds of record indicated by the **red** and **blue** marks. **Type 1** is a STUDENT record and **Type 2** is a COURSE record

Each COURSE record belongs to the preceding STUDENT

The COURSE records contain a credit amount (Shown in **green**)

So, for example

- JOE SMITH is studying 3 courses with total credits worth 9 points
- AL JONES is studying 5 courses with total credits worth 13 points.

The report that is produced here is as follows:

Copyright © Micro Focus 2015-2016. All rights reserved.

```
           S T U D E N T   C R E D I T S   R E P O R T

      STUDENT NAME                COURSES        CREDITS
      JOE SMITH
      JOE SMITH                      3              9
      SAM SHULTZ                     2             15
      AL JONES                       5             28
      SALLY SWELL                    3             37

  TOTAL STUDENTS PROCESSED IS:       6
```

There are 4 logic errors here:

- The total number of student shows 6 – but there are only 5
- The Credits for each student are incorrect
- The student JOE SMITH appears twice in the list
- The last student on file, HAROLD HARDWORK is missing from the list

Most of these bugs are due to poor initial program design. Apart from a redesign fix the bugs as best you can. This is a problem you will face very often, where you will not have the time or budget to rewrite a program.

Please spend some time fixing this, but if you struggle and want to see a solution, then one can be found by switching to the workspace **12_02_data10.**

Quick Quiz

1. What is the verb to change a record on an indexed file?
   a. READ
   b. WRITE
   c. REWRITE
   d. AMEND
   e. WRITE OVER
   f. DELETE
2. What statement allows you to insert a new record?
   a. READ
   b. REWRITE
   c. WRITE
   d. AMEND
   e. WRITE OVER
   f. DELETE
3. Under what circumstances can a DELETE on an indexed record fail?
   a. If the file is not opened for i-o
   b. If the record already exists
   c. If the record does not exist
   d. If the record is greater than 500 bytes
4. Which READ statement(s) below are invalid?
   a. READ EMPLOYEE-FILE.
   b. READ EMPLOYEE-FILE KEY IS EMPLOYEE-SURNAME AT END...

Copyright © Micro Focus 2015-2016. All rights reserved.

   c. READ EMPLOYEE-FILE NEXT RECORD AT END...
   d. READ EMPLOYEE-FILE INVALID KEY...
5. If you READ NEXT on an alternate key, there is no way to detect, in the READ, that the alternate key has changed?
   a. TRUE
   b. FALSE
6. The syntax to READ the record before the current one in an indexed file is:
   a. READ FILE-NAME LAST
   b. READ FILE-NAME PREVIOUS
   c. READ RECORD-NAME LAST
   d. READ RECORD-NAME PREVIOUS

Copyright © Micro Focus 2015-2016. All rights reserved.

13 Modular Programming

Introduction

Modular programming is the name given to the procedure of tackling a programming problem by writing not one, but two or more programs, dividing the coding tasks between the separate programs.

Different programs can communicate with each other, passing data as parameters.

Modular programming is very common, for the following reasons:

- The programming task may be large. If so, dividing it up into discrete tasks (subroutines or modules) means that different people can be working simultaneously.
- You may want to separate the business logic of a program from the 'nuts and bolts' of how the data is retrieved or written.
- Several programs may need common code, for example tax calculation. That code can be written once, and ‘called’ from any program that needs it.

For example, processing the payroll of a company’s staff involves calculating different rates of tax and benefits.

- The program can be divided up logically if each discrete task is allocated to a separate program, often called a *subroutine*, *subprogram* or *module*. Different people can write each program separately, if necessary. If a task requires change, then only that module requires editing and recompiling.
- It might be advisable to keep certain actions away from the main program.

Let’s imagine that you designed a program that accesses and updates employee records on an indexed file.

- However, you later decide to access a database. If all the file access code is contained in a subprogram, then that could be replaced in the future with code to access a database — the code in the main program would not require changes.
- Instead, the subprogram can be “told” what to do — for example, read a record or update a record — by passing down a value, or *parameter*.
- The data fields in a record can be passed back similarly. We can also pass back file status values, so that the top-level program, as it is called, will “know” what happened in the module.

Several programs might require the same functionality. For example, it would be useful if some “common code” carried out end-of-year tax calculations. Rather than writing the same code many times, create a module that is called from many different programs.

The code becomes “common code.” Again, the appropriate data values can be passed back and forth as parameters.

Copyright © Micro Focus 2015-2016. All rights reserved.

In this module we shall discuss how to design and implement modular programming. In practice, modular programming makes program design easier, which is always good!

You saw in one of the examples in the previous module where there was a screen i-o program which "called" a file i-o program. This is a very common split of processing.

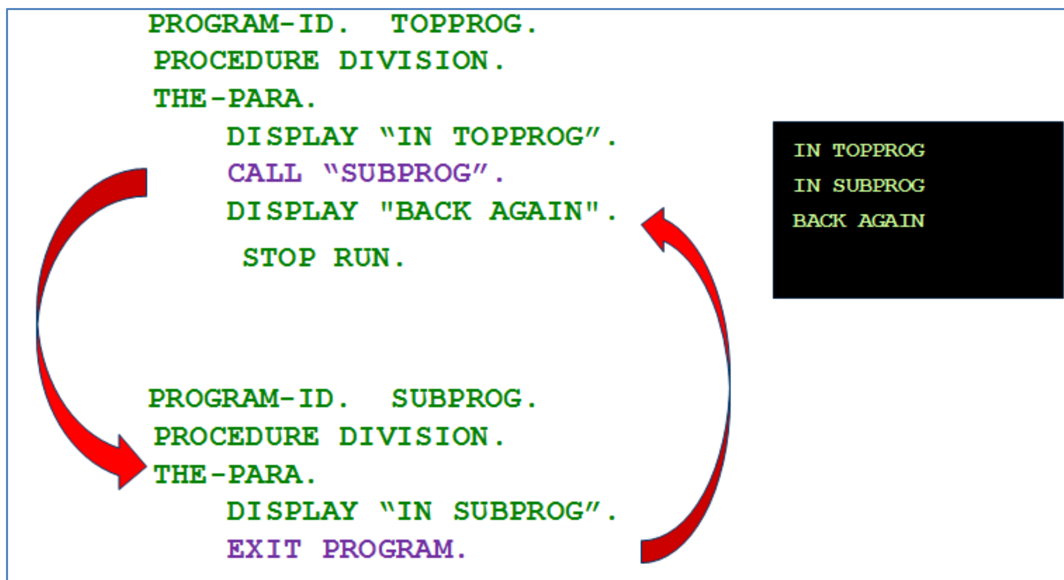
Module Objectives

By the end of this module you will be able to:

- Describe the basic concepts of modular programming
- Explain why modular programming is used
- List the Data Division entries
- Use the CALL and CANCEL verbs
- Pass parameters between modules

The CALL statement

The way in which one program invokes another program is through the CALL statement. E.g.



In this example the TOPPROG calls the SUBPROG. (SUBPROG must of course exist)

The SUBPROG carries out it processing and then returns to the calling program TOPPROG using the EXIT PROGRAM syntax (not STOP RUN).

When you call a sub-program, the name you use in the CALL statement is determined differently in different environments. For example on a Mainframe the name of the program you call is the PROGRAM-ID of the program. In Micro Focus COBOL, the name of the program you call is the name of the program file on disk. For that reason, it is strongly recommended that you save the program to disk with the ***same*** *name as the PROGRAM-ID to avoid confusion.*

Call using a data name

The above example showed TOPPROG calling the SUBPROG by calling a literal value. You can, of course, Call using a data name as shown below:

Copyright © Micro Focus 2015-2016. All rights reserved.

```
PROGRAM-ID.  TOPPROG.
WORKING-STORAGE SECTION.
01  WS-MODULE-NAME       PIC X(16).
PROCEDURE DIVISION.
THE-PARA.
    DISPLAY "HELLO WORLD"
    MOVE "SUBPROG" TO WS-MODULE-NAME.
    CALL WS-MODULE-NAME.
    STOP RUN.
```

Call Nesting

PROG1 can CALL PROG2, which can CALL PROG3, and so on.

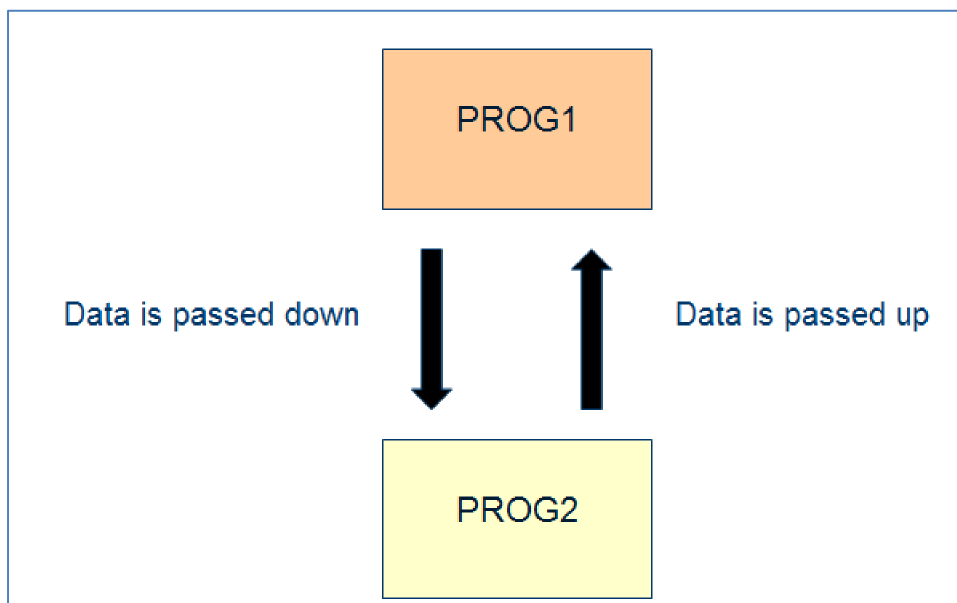
The level of call nesting will depend on the amount of stack space available.

Of course, this second level call is free to use any data items it wishes in the call, including its own linkage section items.

Recursion (e.g. PROG1 calling PROG1) is allowed, but is only useful in particular circumstances. It is very rarely in regular COBOL programs – but is used very commonly in Object Oriented Cobol as we will see later.

Passing parameters

A call can pass parameters to a called program. The called program can alter these values, which then get passed back on the EXIT PROGRAM.



For this to work, the called program must have a LINKAGE SECTION. For this to work, the called program must have a LINKAGE SECTION. This LINKAGE SECTION is used as a "holding" place for the data that is passed from the calling program. The names of the data items in the called program and the calling program need not be the same. But they must be the same formats. We often prefix the

Copyright © Micro Focus 2015-2016. All rights reserved.

linkage section versions with something like LS-, just to help us remember that it is linkage data we are referencing in our program.

The CANCEL Verb

After a program has called another program there is the option of cancelling the called program. This is mainly used in a PC (or Unix) environment. This will have the effect of freeing up memory that the called program was using. It does have the adverse effect that next time the program is called it will be loaded from disk again (with a slight time overhead) and any data values, that the called program was retaining, will be re-initialized. The decision to cancel is down to performance issues.

The syntax is very simple:

```
CANCEL "SUBPROG"
```

Exercise 1

Switch to the workspace **13_01_Modular_Programming.**

This project consists of 2 programs:

- **Program12** – this handles access to a sequential file and then calls Program12a
- **Program12a** – this handles an indexed file and updates this file according to the data that is passed to it by **Program12**.

The syntax in Program12 which is relevant is:

```
01  WS-EMPLOYEE-REC.
    03  EMPLOYEE-NO       PIC X(8).
    03  EMPLOYEE-INITS    PIC X(4).
    03  EMPLOYEE-SURNAME  PIC X(16).
    03  EMPLOYEE-SALARY   PIC 9(6)V99.
    03  EMPLOYEE-ADDRESS  PIC X(40).
    03  EMPLOYEE-DEPT     PIC X(10).
01  ws-action             pic X.
01  ws-status             pic xx.

call "program12a"
      using ws-employee-rec
            ws-action
            ws-status
```

The syntax in Program12a which is relevant is:

```
linkage section .
01  ls-EMPLOYEE-REC .
    03  ls-EMPLOYEE-NO       PIC X(8) .
    03  ls-EMPLOYEE-INITS    PIC X(4) .
    03  ls-EMPLOYEE-SURNAME  PIC X(16) .
    03  ls-EMPLOYEE-SALARY   PIC 9(6)V99 .
    03  ls-EMPLOYEE-ADDRESS  PIC X(40) .
    03  ls-EMPLOYEE-DEPT     PIC X(10) .
01  ls-action             pic x .
01  ls-status             pic xx .

procedure division using ls-employee-rec
                         ls-action
                         ls-status.
```


Copyright © Micro Focus 2015-2016. All rights reserved.

The calling program passes references to the data items contained in its own WORKING-STORAGE section.

The called program defines this data in its own LINKAGE section.

These two definitions are effectively referencing the same data in the original calling program. You will see that the CALL and the PROCEDURE DIVISION using need to have the parameters in the same order.

So study these programs and debug the code to see how the call and exit program are working.

Using a return code

As soon as control of a program is transferred from one program to another, there is a risk that the top level program will not 'know' if anything has gone wrong in the called program.

Using a 'return code' is very common.

The common way to indicate success is to use a numeric field to return a value of 0.

Exercise 2

This example is similar to the one you looked at in the previous module.

Switch to the workspace **13_02_FileInquiry**

Study the code in here and debug the programs.

Suitable patron numbers to use are 101 thru 125.

Exercise 3

Switch to the workspace **13_02_Circle**

It consists of 2 programs:

- **Circle-IO** which handles the use interface which requests the radius if a circle and then calls the subprogram.
- **Circle-calculations** which very simply calculates the area and circumference of a circle

Study the behaviour of these 2 programs by first executing and then by debugging.

You will notice that the call to the subprogram is:

```
Call "circle-calculations" using
                               radius
                               circumference
                               circle-area
```

Module Summary

Now, at the end of this module you are able to:

- Describe the basic concepts of modular programming
- Explain why modular programming is used
- List the Data Division entries

Copyright © Micro Focus 2015-2016. All rights reserved.

- Use the CALL and CANCEL verbs
- Pass parameters between modules

Quick Quiz

1. The CALL to a subprogram must pass parameters to the subprogram
   a. TRUE
   b. FALSE
2. If parameters are passed, the called subprogram needs a LINKAGE section to receive the parameters
   a. TRUE
   b. FALSE
3. A calling program needs a LINKAGE section to send the parameters
   a. TRUE
   b. FALSE
4. The CALL to a subprogram must called using a literal.
   a. TRUE
   b. FALSE
5. The normal use of a return code value would expect a successful return code value to be:
   a. 0
   b. 1
   c. 9
6. If a called program is itself also a calling program, then it must use the same parameters that it was called with?
   a. TRUE
   b. FALSE

Copyright © Micro Focus 2015-2016. All rights reserved.

14 Screen Handling

Introduction

Standard COBOL offers little in the use of displaying and accepting from the screen.

For that reason, most COBOL compiler writers have provided additions to the COBOL syntax which allows a better ability to handle a character screen.

We will look briefly at some of the features which Microfocus has provided.

Module Objectives

By the end of this module you will be familiar with:

- Basic accept and display statements
- Some Microfocus extensions to accept and display
- Future directions

Basic Display and Accept

Display

Standard COBOL provides the following DISPLAY syntax:

```
DISPLAY data-name
DISPLAY 'literal'
```

These can be combined together as shown in this example:

```
DISPLAY 'Hello '
        USER-NAME
        ' I see you are '
        WS-AGE
        ' years old'
```

The DISPLAY displays on the **next** line of the screen.

Accept

Standard COBOL provides the following ACCEPT syntax:

```
ACCEPT data-name
```

The ACCEPT requires user input from the **next** line of the screen.

Enhancements to Display and Accept

As stated above, most COBOL vendors have added their own syntax extensions to support better display and accept usage.

Microfocus has added a variety of additional features, some of which are illustrated below:

Copyright © Micro Focus 2015-2016. All rights reserved.

Display:

- To clear the screen:
```
DISPLAY SPACES UPON CRT
```
- To display a data item at row 11, column 8
```
DISPLAY DATA-NAME AT 1108
```
or
```
DISPLAY DATA-NAME AT LINE 11 COLUMN 8
```
or
```
DISPLAY DATA-NAME AT LINE WS-LINE-NO COLUMN WS-COL-NO
```
- To display a data item at row 11, column 8 in high brightness
```
DISPLAY DATA-NAME AT 1108 WITH HIGHLIGHT
```

Accept:

- To accept a data item at row 11, column 8
```
ACCEPT DATA-NAME AT 1108
```
or
```
ACCEPT DATA-NAME AT LINE 11 COLUMN 8
```
or
```
ACCEPT DATA-NAME AT LINE WS-LINE-NO COLUMN WS-COL-NO
```
- To display a data item at row 11, column 8 in high brightness
```
ACCEPT DATA-NAME AT 1108 WITH HIGHLIGHT
```

There are a number of other variations, handling color, password protection, etc, which we will not describe here. (You can find extensive details in the Microfocus help system).

In addition many COBOL suppliers have provided very advanced screen handling. Microfocus has provided a new data division entry name SCREEN Section along with the advanced Dialog System.

This also shows how you can get function key information from COBOL.

We are not dwelling on these at this time, since we will shortly be describing the ways in which we can produce Windows forms and Web forms using COBOL integration with Microsoft .NET.

Exercise – Screen section syntax

If you wish, you can see an example of the use of SCREEN section by switching the workspace to **14_01_Screen_Section_use.**

You should first look at this application in run mode.

- See how the **tab** and **back tab** keys are working
- See how the **F1** key works
- See how the **<CR>** key works
- See how the **Esc**ape key works

Then debug your way through the code. You will see that how this example uses screen section syntax and uses of Function keys. This is NOT standard COBOL; it is Microfocus extensions to COBOL.

Copyright © Micro Focus 2015-2016. All rights reserved.

Module Summary

Now at the end of this module you will be familiar with:

- Basic accept and display statements
- Some Microfocus extensions to accept and display
- Future directions

Quick Quiz

1. Standard COBOL provides the ability to display and accept from the screen?
   a. TRUE
   b. FALSE
2. Displaying and accepting from the screen at predefined places on the screen are an extension to COBOL
   a. TRUE
   b. FALSE
3. COBOL providers have come up with their own display and accept syntax
   a. TRUE
   b. FALSE

Copyright © Micro Focus 2015-2016. All rights reserved.

15 Database Use

Introduction

Up to now we have been accessing data through sequential files and indexed files.

COBOL, of course, fully supports access to relational databases such as:

- SQL Server
- DB2
- Oracle
- Any database that is supplied with an ODBC Driver

The syntax used in COBOL to access a relational data base is name Embedded Structured Query Language (or ESQL).

It is not intended to be an exhaustive coverage of all the ESQL syntax, but just an illustration of the syntax involved.

Module Objectives

At the end of this module you will have an outline understanding of the way in which COBOL can handle access to a relational database.

Database connection

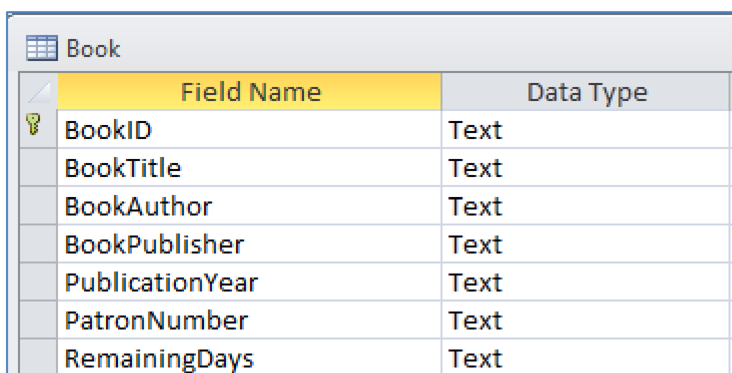
The mechanism you use to connect to a database, through ODBC, is almost the same for all databases. All that is required is for the database provider to supply an ODBC module and for your module to be registered.

Sample database

For the purpose of this training class we will show the access to a simple Microsoft Access data base, named **library**. The principles used here are exactly the same for all relational databases.

This sample database contains 2 tables **Book** and **Patron**

The **Book** table contains the following fields:



Book

Field Name	Data Type
BookID	Text
BookTitle	Text
BookAuthor	Text
BookPublisher	Text
PublicationYear	Text
PatronNumber	Text
RemainingDays	Text

Where **BookID** is the table's key

Copyright © Micro Focus 2015-2016. All rights reserved.

The **Patron** table contains the following fields:

Patron

Field Name	Data Type
PatronNumber	Text
FirstName	Text
LastName	Text
StreetAddress	Text
City	Text
State	Text
Zip	Text

Where **PatronNumber** is the table's key

The tables are joined through **Book.PatronNumber** and **Patron.PatronNumber**

Viewing the database

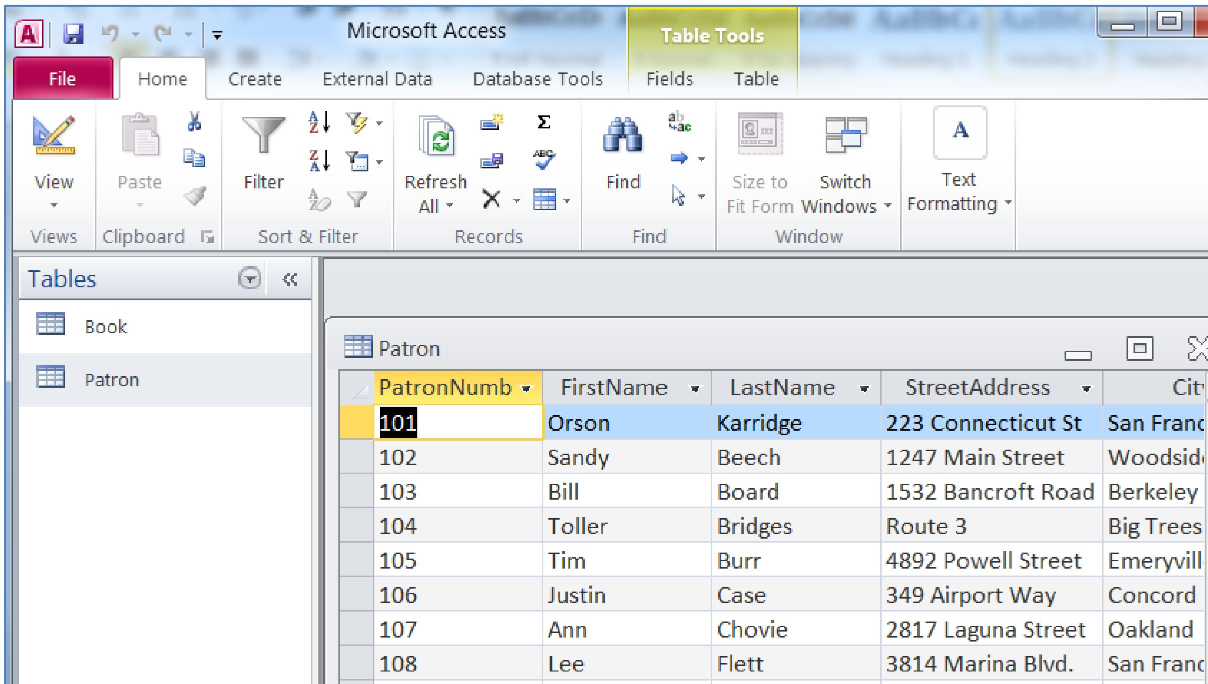
The database itself is **Library.mdb** and can be found at **C:\COBOLClass_Eclipse\DataFiles.** If you have Microsoft Access installed on your PC you can view this database by double clicking on **C:\COBOLClass_Eclpise\DataFiles\Library.mdb.**

Book

BookID	BookTitle	BookAuthor	BookPublisl
0001	Business Law	Force, Gayle	Legal Press
0002	Non-Business	Force, Gayle	Legal Press
0003	Contract Law	Peace, Warren	Legal Press
0004	Legal Secretar	Robe, Mike	Legal Press
0005	General Econo	Wheels, Helen	Jim Beam
0006	Specific Econo	Kerr, Ann	Jim Beam
0007	Economics Tod	Kerr, Ann	John Beam
0008	Silicon Econon	Mite, Dina	Ayixa Press
0009	Starring Silicor	Mite, Dina	Ayixa Press
0010	Introduction to	Flett, Lee	JP Publish.
0011	Computer Info	Key, Lee	JP Publish.
0012	Computers in I	Key, Lee	JP Publish.
0013	The 123s of Co	Zeen, Ben	JP Publish.
0014	The 456s of Lap	Tory, Vic	JP Publish.
0015	Life Without C	Zapple, Adam	Cobol Univ

Record: 1 of 25

Copyright © Micro Focus 2015-2016. All rights reserved.

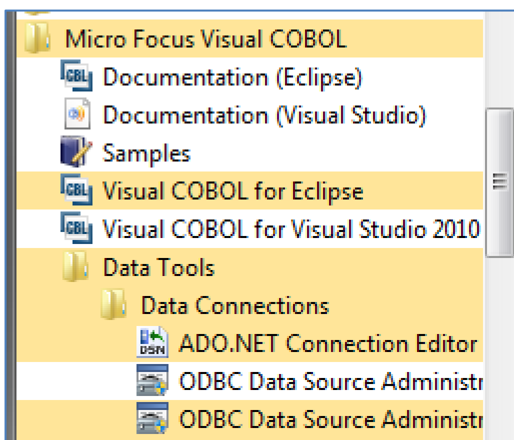


You do **not** need to have Microsoft Access installed on your PC in order to access this data with COBOL.

Registering the database with ODBC

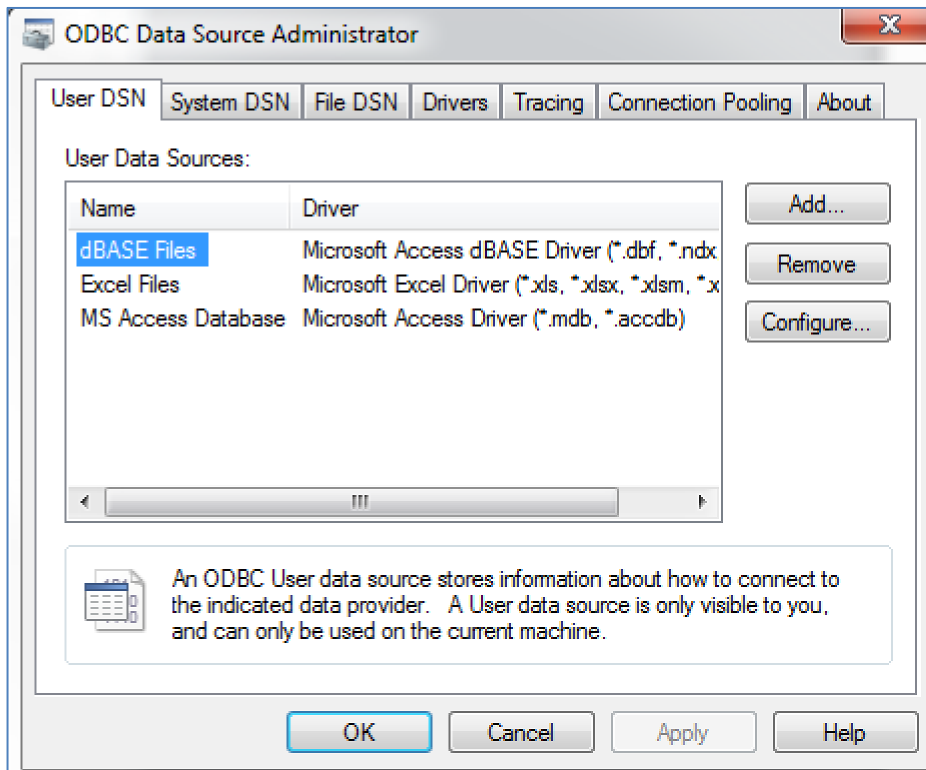
The mechanism we will be using to access this database is through ODBC, so we need to register this database correctly. Take the following steps:

1. From the Windows Start menu select **All Programs, Micro Focus Visual COBOL, Data Tools, Data Connections, ODBC Data Source Administration (32bit)**



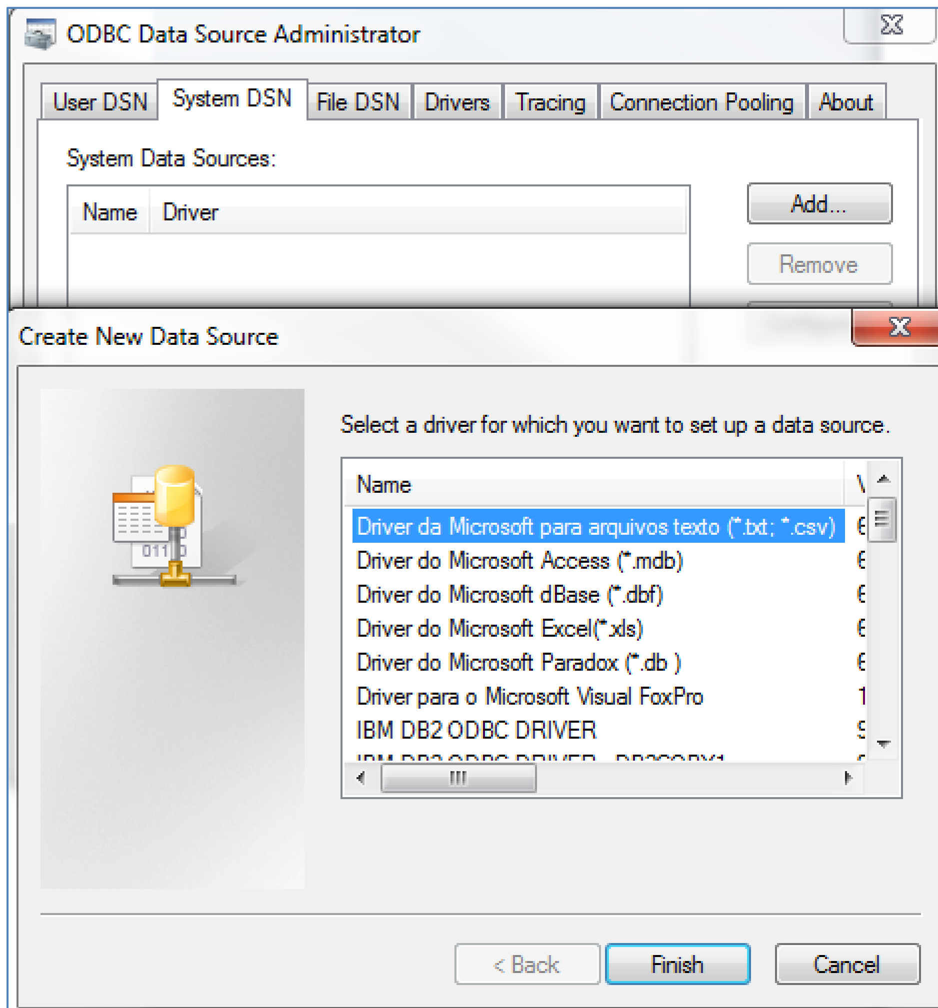
2. This will bring up the following window:

Copyright © Micro Focus 2015-2016. All rights reserved.

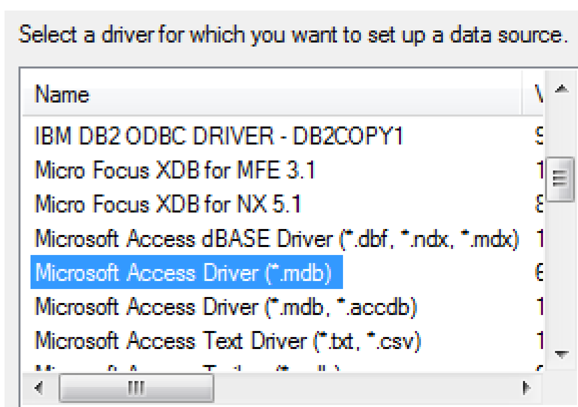


3. Select the **System DSN** tab and click **Add**.

Copyright © Micro Focus 2015-2016. All rights reserved.

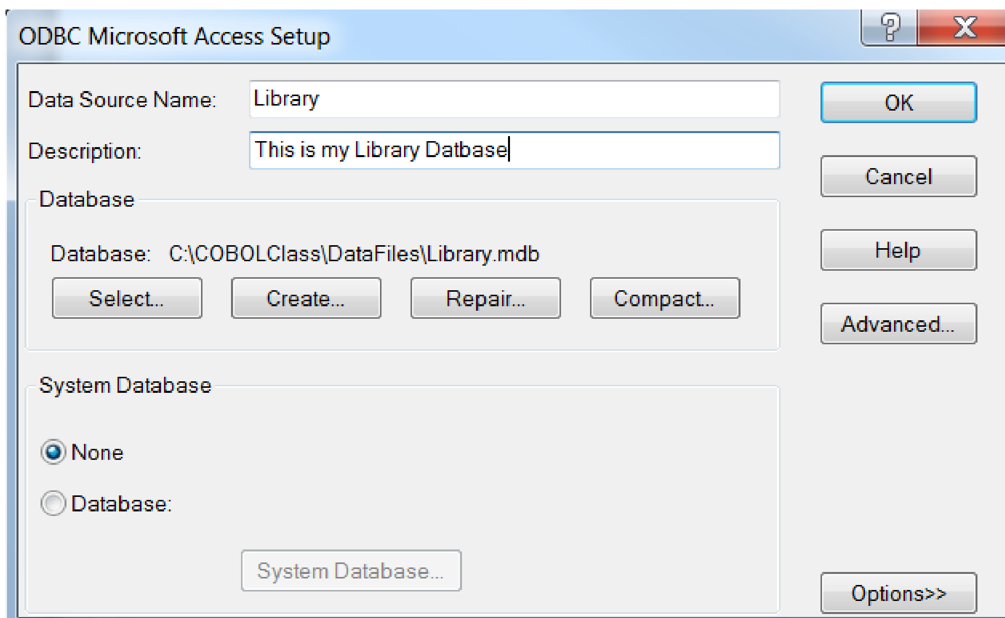


4. Select the entry for **Microsoft Access Driver (*.mdb)** and click **Finish**

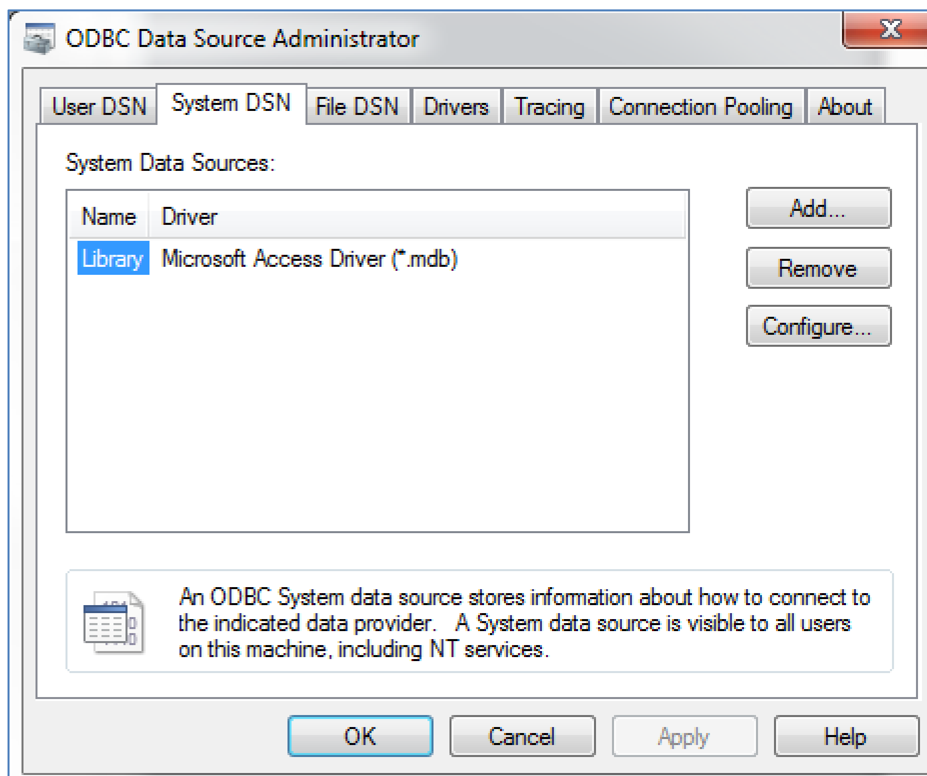


5. Enter the Data Source name to be **Library** and select the Database **C:\COBOLClass_Eclipse\DataFiles\Library.mdb.**

Copyright © Micro Focus 2015-2016. All rights reserved.



6. Click **OK** to finish.



7. Press **OK** once more to close the ODBC data source administrator.
   The connection is now a Windows ODBC data source connection named **Library,** pointing to the Access database.

Accessing the database from COBOL

A project has been provided for you to illustrate reading this database. The workspace to use is **C:\COBOLClass_Eclipse\Projects\15_01_SQLProject.**

Copyright © Micro Focus 2015-2016. All rights reserved.

COBOL program

There is one COBOL program which reads the 2 tables in the database using the Patron number as a link. It reads the tables and orders the results by BookAuthor.

```
EXEC SQL INCLUDE SQLCA  END-EXEC.

01 MFSQLMESSAGETEXT  PIC X(250).

EXEC SQL BEGIN DECLARE SECTION  END-EXEC
EXEC SQL INCLUDE Book END-EXEC.
EXEC SQL INCLUDE Patron END-EXEC.

EXEC SQL END DECLARE SECTION END-EXEC
EXEC SQL
      DECLARE CSR3 CURSOR FOR SELECT
             A.BookTitle
            ,A.BookAuthor
            ,A.PatronNumber
            ,B.FirstName
            ,B.LastName
      FROM Book A
            ,Patron B
      WHERE (A.PatronNumber = B.PatronNumber)
      ORDER BY A.BookAuthor
    END-EXEC.

PROCEDURE DIVISION.
RUN-START.
    EXEC SQL
```

There are 2 copy files – one for each table.

The cursor name “CSR3” in our example is just an automatically generated name, which is the next in the sequence of cursor names. You can rename this to anything you want.

Copyright © Micro Focus 2015-2016. All rights reserved.

Book copy file:

```
*> -----------------------------------------
*> DECLARE TABLE for Book
*> -----------------------------------------
   EXEC SQL DECLARE Book TABLE
   ( BookID                 VARCHAR(4)
   , BookTitle              VARCHAR(25)
   , BookAuthor             VARCHAR(13)
   , BookPublisher          VARCHAR(11)
   , PublicationYear        VARCHAR(4)
   , PatronNumber           VARCHAR(3)
   , RemainingDays          VARCHAR(3)
   ) END-EXEC.
*> -----------------------------------------
*> COBOL HOST VARIABLES FOR TABLE Book
*> -----------------------------------------
01  DCLBook.
    03 Book-BookID                  PIC X(4).
    03 Book-BookTitle               PIC X(25).
    03 Book-BookAuthor              PIC X(13).
    03 Book-BookPublisher           PIC X(11).
    03 Book-PublicationYear         PIC X(4).
    03 Book-PatronNumber            PIC X(3).
    03 Book-RemainingDays           PIC X(3).
*> -----------------------------------------
*> COBOL INDICATOR VARIABLES FOR TABLE Book
*> -----------------------------------------
01  DCLBook-NULL.
    03 Book-BookID-NULL             PIC S9(04)  COMP-5.
    03 Book-BookTitle-NULL          PIC S9(04)  COMP-5.
    03 Book-BookAuthor-NULL         PIC S9(04)  COMP-5.
    03 Book-BookPublisher-NULL      PIC S9(04)  COMP-5.
    03 Book-PublicationYear-NULL    PIC S9(04)  COMP-5.
    03 Book-PatronNumber-NULL       PIC S9(04)  COMP-5.
    03 Book-RemainingDays-NULL      PIC S9(04)  COMP-5.
```


Copyright © Micro Focus 2015-2016. All rights reserved.

Patron Copy file

```
*> -------------------------------------------
*> DECLARE TABLE for Patron
*> -------------------------------------------
   EXEC SQL DECLARE Patron TABLE
   ( PatronNumber          VARCHAR(3)
   , FirstName             VARCHAR(10)
   , LastName              VARCHAR(12)
   , StreetAddress         VARCHAR(20)
   , City                  VARCHAR(14)
   , State                 VARCHAR(2)
   , Zip                   VARCHAR(10)
   ) END-EXEC.
*> -------------------------------------------
*> COBOL HOST VARIABLES FOR TABLE Patron
*> -------------------------------------------
01  DCLPatron.
    03 Patron-PatronNumber            PIC X(3).
    03 Patron-FirstName               PIC X(10).
    03 Patron-LastName                PIC X(12).
    03 Patron-StreetAddress           PIC X(20).
    03 Patron-City                    PIC X(14).
    03 Patron-State                   PIC X(2).
    03 Patron-Zip                     PIC X(10).
*> -------------------------------------------
*> COBOL INDICATOR VARIABLES FOR TABLE Patron
*> -------------------------------------------
01  DCLPatron-NULL.
    03 Patron-PatronNumber-NULL       PIC S9(04)  COMP-5.
    03 Patron-FirstName-NULL          PIC S9(04)  COMP-5.
    03 Patron-LastName-NULL           PIC S9(04)  COMP-5.
    03 Patron-StreetAddress-NULL      PIC S9(04)  COMP-5.
    03 Patron-City-NULL               PIC S9(04)  COMP-5.
    03 Patron-State-NULL              PIC S9(04)  COMP-5.
    03 Patron-Zip-NULL                PIC S9(04)  COMP-5.
```

Processing details

The COBOL program uses the Cursor declaration, shown below, to fetch and display results on the screen, using the procedure division code:

Copyright © Micro Focus 2015-2016. All rights reserved.

```
FETCH-DATA.
    PERFORM UNTIL SQLSTATE >= "02000"
        EXEC SQL
            FETCH CSR3  INTO
                :Book-BookTitle:Book-BookTitle-NULL
               ,:Book-BookAuthor:Book-BookAuthor-NULL
               ,:Book-PatronNumber:Book-PatronNumber-NULL
               ,:Patron-FirstName:Patron-FirstName-NULL
               ,:Patron-LastName:Patron-LastName-NULL
        END-EXEC
        IF SQLSTATE < "02000"
            display Book-BookTitle ' '
                    Book-BookAuthor ' '
                    Book-PatronNumber ' '
                    Patron-FirstName ' '
                    Patron-LastName
        END-IF
    END-PERFORM
    .
```

Running the application

To execute the program, press **F5.**

This will give the results shown below:

```
SimpleAccess
Object Technology          Beech, Sandy  106 Justin     Case
Intro to Info Systems      Diss, John    103 Bill       Board
Introduction to Data Proc  Flett, Lee    107 Ann        Chovie
Non-Business Law           Force, Gayle  105 Tim        Burr
Business Law               Force, Gayle  101 Orson      Karridge
The NEXT Step              Job, Anita    101 Orson      Karridge
Computers in Management    Key, Lee      104 Toller     Bridges
Computer Information Sys   Key, Lee      103 Bill       Board
C                          Leets, Ethyl  102 Sandy      Beech
Database Processing        Lesse, Moira  106 Justin     Case
Starring Silicon Valley    Mite, Dina    108 Lee        Flett
Silicon Economics          Mite, Dina    101 Orson      Karridge
No One's Guide to dBASE    Ology, Archie 101 Orson      Karridge
Unix Hater's Reference     O'Shea, Rick  102 Sandy      Beech
Contract Law               Peace, Warren 101 Orson      Karridge
Legal Secretary Reference  Robe, Mike    101 Orson      Karridge
The 456s of Laptops        Tory, Vic     109 Ima        Fox
General Economics          Wheels, Helen 102 Sandy      Beech
The CORBA Standard         Yard, Bill    108 Lee        Flett
Life Without Computers     Zapple, Adam  101 Orson      Karridge

Press <CR> to terminate
```

You will see that it is ordered by BookAuthor as required.

Debugging the application

To debug the program, press **F11.** This behaves in exactly the same way as debugging any of the programs you have seen so far.

Although this sample program uses database access, it is completely OK to mix in regular COBOL file access in the same program.

Copyright © Micro Focus 2015-2016. All rights reserved.

```
PROCEDURE DIVISION.
RUN-START.
    EXEC SQL
        WHENEVER SQLERROR perform OpenESQL-Error
    END-EXEC
    PERFORM CONNECT-TO-DATABASE
    PERFORM OPEN-CURSOR
    PERFORM FETCH-DATA
    PERFORM SHUT-DOWN
    .
CONNECT-TO-DATABASE.
    EXEC SQL
        CONNECT TO 'Library'
    END-EXEC
    .
```

Step through the code with **F11**, as you have done before to see the code being executed.

Module Summary

Now you are at the end of this module you will have an outline understanding of the way in which COBOL can handle access to a relational database.

Exercise

If you have not already done so, load the solution **SQLProject.sln** from **\COBOLClass_Eclipse\Projects\15_01_SQLProject**

Examine the code inside this program and debug the code to get a feel for the way the syntax is working.

Quick Quiz

1. Relational database access is the same as indexed file access
    a. TRUE
    b. FALSE
2. In order to use relational database access you need to have a copy file for each of the database tables you are using.
    a. TRUE
    b. FALSE
3. Copy files relating to the tables include both COBOL data items and database items
    c. TRUE
    d. FALSE
4. You cannot mix database access and regular COBOL file access in the same program.
    e. TRUE
    f. FALSE

Copyright © Micro Focus 2015-2016. All rights reserved.

16 Object Oriented COBOL

The next few modules deal with Object Oriented COBOL and the JVM environment.

You may choose at this point, to follow on with these modules at a later date.

Copyright © Micro Focus 2015-2016. All rights reserved.

Introduction

This chapter introduces you to Object Oriented syntax in Visual COBOL through a series of evolving solutions.

The purpose of this chapter is to give you a high level overview of using Visual COBOL to code and debug Object COBOL programs. It is not the intention of this chapter to provide you with an exhaustive list of all the features and syntax of Object COBOL. Rather it introduces you to the main concepts and practices. Later, as you explore some of the extensive examples of Visual COBOL which are provided with the Micro Focus product, you should be aware of all the concepts behind the details.

Module Objectives

At the end of this module you will be familiar with:

- Coding Classes using Visual COBOL.
- Creating Class and Instance methods.
- Coding and debugging a program using a class and instance.

Program versus Class

The first thing to be familiar with is the difference between a Program and a Class.

The diagrams below show the structure of typical procedural COBOL program and the structure of an Object COBOL class.

Procedural Cobol Structure Strict structure	Object Cobol Structure Free Structure

`Identification Division.`
`Program-ID. Myprog.`
`Environment Division.`
`Data Division.`
`Procedure Division`
	`Class-id. Myclass.`
	`End Class.`

Most of the restrictions that traditional COBOL programs have is not the case when considering a Class, as you will see.

Quick Start Scenario

COBOL's object-oriented features, as incorporated into Visual COBOL, permit COBOL to function in the object world of other languages. To use these capabilities you need to become familiar with the new object-oriented syntax added to COBOL.

There is a very simple application, the circle example. While simple, it gives you important concepts that you will be employing when working with COBOL in the .NET environment.

Class Structure

Copyright © Micro Focus 2015-2016. All rights reserved.

```
Class-id. Myclass.

Method-id myClassMethod static.
01 My-data1    pic x.
Procedure Division.
 …. Program code
end method.

Method-id myInstanceMethod.
01 My-data2    pic x.
Procedure Division.
 …. Program code
end method.

End class.
```

"Static method definition"

If the word "static" is used against a method then that is a **Class** Method.

"Object instance method definition"

If the word "**static**" is NOT used against a method then that is an Object **Instance** Method. An instance method is a method which applies to an object instance. Object instances, as we will see later are instantiated by a class.

NOTE: in this example, there is no need for the words "**data division**" or "**working-storage section**"

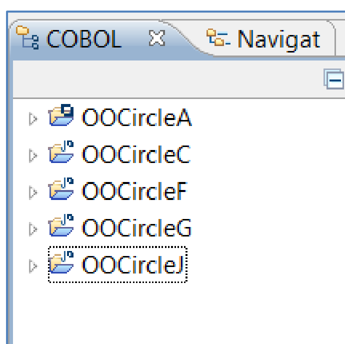
Note also that **Static** methods and **Instance** methods do not need to come in any particular order. However it is recommended, for clarity, that they are kept apart. E.g. Static methods first, Instance methods second.

Evolving demonstrations

There now follows a series of evolving demonstrations/exercises where Object COBOL syntax is used.

This is not shown as an example of what you need to do in practice, but rather demonstrates the structure and syntax of Object COBOL programs in a series of progressions.

If you switch to the workspace **16_01_OOCircles** there are a number of OO projects:



Copyright © Micro Focus 2015-2016. All rights reserved.

You will look at each of these projects in turn as you progress through this module.

What you need to do it to look at the code in each of these cases and then execute the code in debug mode to see how the various lines of code are set up and executed.

The sections shown below give you points to look out for. **Take your time doing this in order to understand what is happening.**

Exercise 1

The first thing to do is to look at a solution we already saw earlier in the course. This is **NOT** an example of Object Orientation but simply shows a procedural example that we will evolve into Object Oriented.

So, open the project **OOCircleA**

This solution contains 2 programs:

It consists of 2 programs:

- **Circle-IO** which handles the use interface which requests the radius if a circle and then calls the subprogram.
- **Circle-calculations** which very simply calculates the area and circumference of a circle

The circle-IO program is very simple COBOL code which just displays some textual information and then asks you to enter a value for the radius of a circle. It then calls the circle-calculations program, which calculates the area and circumference of the circle and passes these values back to the circle-IO program. The circle-IO program then displays these results on the screen and asks for the next radius. (entering a 0 radius terminates the program)

Study the behaviour of these 2 programs by first executing and then by debugging.

You will notice that the call to the subprogram is:

```
Call "circle-calculations" using
                          radius
                          circumference
                          circle-area
```

The circle-calculations program contains:

```
Identification Division.
Program-id. circle-calculations.

Data Division.
Linkage Section.
01  ls-radius          pic 99.
01  ls-circumference   pic 999v9.
01  ls-area            pic 99999v9.

Procedure division using ls-radius
                         ls-circumference
                         ls-area.
  Compute ls-circumference = 6.28 * ls-radius
  Compute ls-area = 3.14 * ls-radius * ls-radius
```


Copyright © Micro Focus 2015-2016. All rights reserved.

```
        exit program
        .
```

Notice that the call parameter order in the calling program matches the parameter order in the called program

Exercise 2

This next project changes the Circle-calculations program into an Object Oriented Class.

So, open the project **OOCircleC**

First of all execute this code to see that it is behaving in the same way that the previous project behaved.

Note1: In order to execute OO code inside Eclipse you are using the Java Virtual Machine (JVM) so to execute circle-io you need to right click on the program and select **Run As/COBOL JVM Application**.

Note2: You will see, that in the case of this OO project, the “display” and “accept” do not happen in a regular DOS window, but at the Console at the foot of the Eclipse Windows

Now let’s look at the code in circle-IO.

Inside the circle-IO program you will see the following in procedure division:

```
     Invoke type CircleCalculations::CalculateArea(radius) returning circle-area
     set circumference to type CircleCalculations::CalculateCircumference(radius)
```

The “Invoke” invokes a class method inside the circle-IO class. We know it is a class method that we are trying to invoke the word “**type**” in the INVOKE indicates that the method is a class method. We, of course, can only know it is a class method by looking in the class program itself.

So the way we can read the statement

```
Invoke type CircleCalculations::CalculateArea(radius)
                        returning circle-area
```

is:

- Invoke the class method **CalculateArea** inside the Class **CircleCalculations**
- Pass the value **radius** to that method and receive the value **circle-area** back from that method

The second statement set circumference . . . is exactly the same, but using a different syntax. If a method returns just a single value or object reference, then you would probably choose to use this set syntax.

So the following 2 statements are identical in behaviour:

```
Invoke type CircleCalculations::CalculateArea(radius) returning circle-area

Set circle-area to type CircleCalculations::CalculateArea(radius)
```

If a method returns more than 1 value or object reference then you would need to use the Invoke Syntax.

Copyright © Micro Focus 2015-2016. All rights reserved.

Now let's look at the code in circle-calculations.

```
Class-ID. CircleCalculations.
*>----------------------------------------------
Method-ID. CalculateArea static.
Linkage Section.
01 ls-radius          pic 99.
01 ls-area            pic 99999v9.
Procedure Division Using by value ls-radius
                          Returning ls-area.
   Compute ls-area = 3.14 * ls-radius * ls-radius
   Exit Method
   .
End Method CalculateArea.
*>----------------------------------------------------------
Method-ID. CalculateCircumference static.
Linkage Section.
      01 ls-radius         pic 99.
      01 ls-circumference  pic 99999v9.
Procedure Division Using by value ls-radius
                      Returning ls-circumference.
   Compute ls-circumference = 2 * 3.14 * ls-radius
   Exit Method
   .
End Method CalculateCircumference.
*>----------------------------------------------------------
END CLASS CircleCalculations.
```

1. You will see that the Class begins with the **Class-ID** statement and ends with the **END CLASS** statement.
2. Inside this class there are 2 methods, each beginning with a **Method-ID** and ending with an **End Method**
3. The word **static** alongside a method indicates that this is a **class** method. If the word **static** is missing then the method is an object **instance** method.
4. Inside the method is the normal COBOL syntax you would expect to find, except the end of the method has the **Exit Method** syntax. This is the equivalent of Exit Program in procedural code. The good news is that you do not even need to use this. The End Method is sufficient for the run time system to know that it should return to whatever invoked the method.

Now debug your way through the code to ensure that you understand what is happening.

Note: In the case of Object Oriented programs, since JVM is being used, you will need to set a breakpoint at the start of the program so that debugging will be seen.

Exercise 3

This next solution uses the Circle-calculations Class, but in this case, the methods have become Object **Instance** methods.

So, load the project **OOCircleF**

First of all execute this project to see that it is behaving in the same way that the previous project behaved.

Now let's look at the code in circle-IO.

Copyright © Micro Focus 2015-2016. All rights reserved.

Inside the circle-IO program you will see the following in data division.

```
    01  circleObject  type  CircleCalculations.
```

This is declaring a data item **circleObject**. It does **not** have a picture clause. It is declared to be an object instance reference to an object type **CircleCalculations.**

In procedure division, this object instance must be created. The object needs to be created by the class, using the "new" method. The syntax to create this object is:

```
Invoke type CircleCalculations::New() returning circleObject
```

From this point on, it is **not** the **class methods** which are invoked, but the **object instance methods**:

```
      Invoke circleObject::SetRadius(radius)
      Invoke circleObject::Calculate()
```

We know these are object instance methods in 2 ways:

- We are not using the word "type" in the invoke. (If we did we would get a compiler error).
- We are referring to the object Instance **circleObject**, not the class **CircleCalculations**.

Now let's look at some of the code in circle-calculations.

```
Identification Division.
Class-id. CircleCalculations.
Data Division.
Working-Storage Section.
01 radius              pic 99.
01 circle-area         pic 99999v9.
01 circumference       pic 999v9.
```

The first thing to notice is that the Class has a data division.

The contents of this data division are the Object Instance data. Data defined here can be either Object **Instance** data or **Class** data. If it is **class data**, it will have the word "**Static**" alongside it.

Further down the code you will find several methods e.g.

```
*>----------------------------------------------
Method-ID. SetRadius.
*>----------------------------------------------
Linkage Section.
01 ls-radius          pic 99.
Procedure Division Using by value ls-radius.
    Move ls-radius to radius
    .
End Method SetRadius.
*>----------------------------------------------
Method-ID. Calculate.
*>----------------------------------------------
Procedure Division.
   Compute Circle-area = 3.14 * radius * radius
   Compute circumference = 6.28 * radius
   .
End Method Calculate.
```


Copyright © Micro Focus 2015-2016. All rights reserved.

All these methods are object **instance** methods, since there is no "**static**" defined alongside the method.

The first instance method **SetRadius**, simply takes the **ls-radius** that is passed to it and places it into the object instance data item **radius**.

The method **Calculate** does the required calculations and places the results into the object instance data items. It does **not** return anything to the invoking program.

To get these values back to the invoking program, extra instance methods are defined:

```
Method-ID. GetCircumference.
Linkage Section.
01 ls-circumference      pic 999v9.
Procedure Division Returning ls-circumference.
   Move circumference to ls-circumference
   .
End Method GetCircumference.

Method-ID. GetCircleArea.
Linkage Section.
01 ls-circle-area      pic 99999v9.
Procedure Division Returning ls-circle-area.
   Move circle-area to ls-circle-area
   .
End Method GetCircleArea.
```

These methods get the data values from the instance data and pass them back to the invoking program. This sounds very clumsy! Just wait a moment and we will see how we can make this much, much easier.

For now, let's see how the circle-IO program behaves

```
      set circleObject to type CircleCalculations::New()
      Invoke circleObject::SetRadius(radius)
      Invoke circleObject::Calculate()
```

In this:

- An object instance **circleObject** is created
- The radius is then passed to the object instance using the **SetRadius** method
- The object instance is then asked to use its **Calculate** method

But how do we get these values back to Circle-IO?

This is done with the 2 statements:

```
    Move circleObject::GetCircumference() to edit-circumference
    Move circleObject::GetCircleArea() to edit-circle-area
```

Now debug through the code and watch how the application is behaving.

Copyright © Micro Focus 2015-2016. All rights reserved.

Exercise 4

We have gone from a fairly simple procedural program to what appears to be a more complex Object Oriented solution. We have done this to gain an understanding of the way in which OO works.

Let's now make life simpler.

Look at the project **OOCircleG**

First of all execute this project to see that it is behaving in the same way that the previous projects behaved.

Let's first look at the CircleCalculations program

```
Identification Division.
Class-id. CircleCalculations.
Working-Storage Section.
01 Radius               pic 99      property.
01 CircleArea           pic 99999v9 property.
01 Circumference        pic 999v9   property.
Method-ID. Calculate.
Procedure Division.
    Compute CircleArea = 3.14 * radius * radius
    Compute Circumference = 6.28 * radius
    .
End Method Calculate.
End Class CircleCalculations.
```

This looks like a very simple program. It has just one object instance method **Calculate**.

It also has 3 data items which are object instance data items. However there is the word **property** alongside each data item.

What this means is that if an object instance data item has the word **property** alongside it, then there are hidden default **get** and **set** methods for that data item. They have the singular function of receiving or returning object instance data values.

Let's look inside the circle-IO program to see how this is used

To set the radius instance data value we use:

```
    Move ip-radius to circleObject::Radius
```

To do the calculation we use:

```
    Invoke circleObject::Calculate
```

To get the instance values back we use:

```
    Move circleObject::Circumference to circumference

    Move circleObject::CircleArea to circle-area
```

Now debug through the code and watch how the application is behaving.

Copyright © Micro Focus 2015-2016. All rights reserved.

The implicit **get** and **set** methods are provided by using the **property** clause alongside the instance data items:

```
01 Radius              pic 99      property.
01 CircleArea          pic 99999v9 property.
01 Circumference       pic 999v9   property.
```

However we could have applied 2 variations to this property clause:

```
01 Radius              pic 99      property no get.
01 CircleArea          pic 99999v9 property no set.
01 Circumference       pic 999v9   property no set.
```

The **no get** clause means that that data item does not have an implicit **get** method.

The **no set** clause means that that data item does not have an implicit **set** method.

So in the example above:

- the radius can be "set" but cannot be "got"
- the CircleArea and CircleCircumference can be "got" but not "set"

Exercise 5

Finally look at the project **OOCircleJ.**

First of all execute the application. It will ask for a radius and not give any result. It will ask for another radius. Enter a few radius values and finally enter 0.

Multiple Object Instances have been created.

This will then show you the values in each of the multiple objects that it created!

Finally debug this code and see how it is working.

Module Summary

Now at the end of this module you should be familiar with:

- Coding Classes using Visual COBOL.
- Creating Class and Instance methods.
- Coding and debugging a program using a class and instance.

You should now have a high level overview of using Visual Studio to code and debug Object COBOL programs.

- It was not the intention of this chapter to provide you with an exhaustive list of all the features and syntax of Object COBOL. Rather it introduced you to the main concepts and practices.

Quick Quiz

1. In a class program do you distinguish class instance data from object instance data?
    a. You add the word "type" after the data definition.
    b. You add the word "static" after the data definition.

Copyright © Micro Focus 2015-2016. All rights reserved.

c. It is not possible to distinguish class data from object data

2. In a class program do you distinguish a class method from object instance method?
    a. You add the word “type” as part of the method definition.
    b. You add the word “static” as part of the method definition.
    c. It is not possible to distinguish class methods from object methods.
3. When invoking a method how do you ensure that it is a class method you are invoking?
    a. You add the word “type” as part of the method invocation.
    b. You add the word “static” as part of the method invocation.
    c. It is not possible to distinguish class methods invocation from object methods invocation.
4. When invoking a method how do you ensure that it is an object instance method you are invoking?
    a. You add the word “type” as part of the method invocation.
    b. You add the word “static” as part of the method invocation.
    c. You do not add anything to the invocation
    d. It is not possible to distinguish class methods invocation from object methods invocation.
5. How do you indicate that you want method **Calculate** in the object **MyObject?**
    a. MyObject.Calculate
    b. MyObject:Calculate
    c. MyObject::Calculate
    d. Calculate in MyObject
6. How do you create a new instance object MyObject from the class MyClass?
    a. Invoke MyClass::New() giving MyObject
    b. Invoke type MyObject::New() giving MyClass
    c. set MyObject to MyClass::New()
    d. set MyObject to type MyClass::New()

Copyright © Micro Focus 2015-2016. All rights reserved.

17 Further JVM Features

Introduction

This module provides further example of COBOL in the JVM environment

Module Objectives

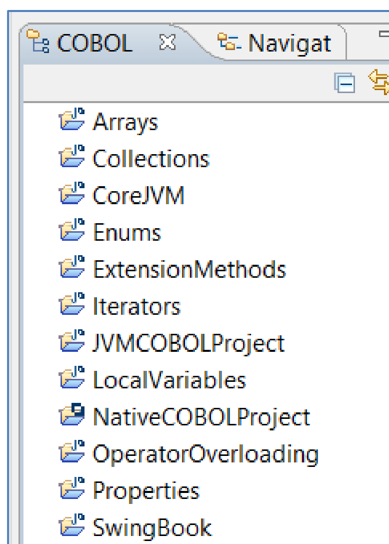
A workspace has been provided which contains a number of JVM projects.

These projects are not discussed in detail.

It is up to you to decide which of these projects are of interest to you.

Project detail

The workspace you should switch to is **17_01_JVM_Projects**.



These projects are taken from the examples supplied with the Visual COBOL product.

The names of the projects should be self-explanatory describing the purpose of the project.

Feel free to explore these samples as you see fit.

Copyright © Micro Focus 2015-2016. All rights reserved.

18 Course Conclusions

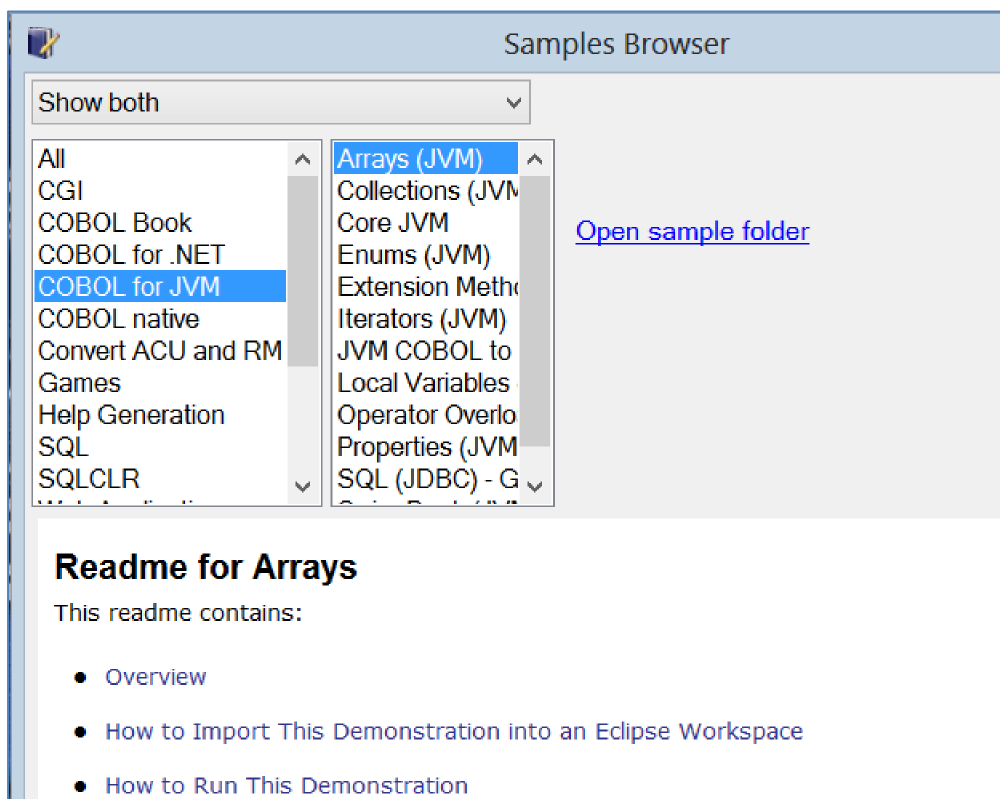
This course has taken you through the structures and features of the COBOL programming language.

It started with "traditional" COBOL and then moved through the implementation of Object Oriented COBOL in the 1990s and then on to more recent developments; providing fully featured JVM COBOL using all the powers of the Java Virtual Macine framework.

Course Follow-on

There are many samples provided by Microfocus, which you may want to examine. These samples can be found from the Windows Start menu under **Microfocus Visual COBOL, Samples.**

When you select this you will get the **Samples Browser**:



Just out of interest, this **Samples Browser** was written using Visual COBOL.

Feel free to explore the various samples, as you require.

Course Examination

You are now ready to take the course examination, as provided by your tutor.

Copyright © Micro Focus 2015-2016. All rights reserved.

19 Appendix I - Managed COBOL Review

Introduction

This appendix, re-introduces you to the features of Managed COBOL; particularly those which relate to Object Oriented COBOL.

This course has primarily been directed at the JVM environment, using Eclipse as the development IDE. However most of what has been covered applies equally to Microsoft .NET using Visual Studio as the development IDE. (Micro Focus supplies a version of Visual COBOL which is based around Visual Studio and .NET).

Managed COBOL refresher

Managed COBOL is the collective term for .NET COBOL and JVM COBOL. Managed COBOL is regular procedural COBOL with extensions to take advantage of the features of the managed frameworks. This includes object-oriented syntax (OO) that allows access to large libraries of functionality which you can use in your application and much more. To take full advantage of Managed COBOL, it is a great advantage to understand object-oriented concepts.

This summation guide serves as a basic review of object-oriented programming for COBOL developers.

Classes & Methods

At the heart of Object Oriented Programming (OOP) is the notion of a **class**. A class is said to encapsulate the information about a particular thing. A class contains data associated with the entity and operations, called methods, that allow access to and manipulation of the data. Aside from encapsulation of data, Classes are also very useful for bridging your existing procedural programs with managed code technologies.

Here's a simple COBOL class:

```
class-id. MyClass.

method-id. SayHello static.

linkage section.
01 your-name     pic x(10).
01 your-greeting pic x(20).

procedure division using your-name
            returning your-greeting.
    move "hello " & your-name to your-greeting
    .
end method.
end class.
```

Before we look at the details of the class, let's see how you would invoke the single method contained in this class.

Copyright © Micro Focus 2015-2016. All rights reserved.

```
program-id. TestMyClass.
working-storage section.
01  your-name      pic x(10).
01  your-greeting  pic x(20).
procedure division.
   Move 'Scot' to your-name
   invoke type MyClass::SayHello(your-name)
                returning your-greeting
   display your-greeting
   .
end program TestMyClass.
```

As you would expect, the result of this program is:

```
Hello Scot
```

In this example, you can see how a procedural COBOL program can also use object-oriented semantics even though it is itself not a class.

These two programs can be found by loading the workspace **18_01_Simple_Class**

Let's look at the details of the class:

```
class-id MyClass.
```

MyClass is the name of the class. When you reference a class, you do so by its name much in the same way you would reference a COBOL program.

Our class contains no data but it does have a single method named **SayHello**.

```
method-id SayHello static.
```

Notice there is STATIC clause associated with this method. This keyword is important in that is allows us to call the method without creating an instance of the class (That is, we are using a Class Method). Instances of classes are called objects which we'll come to later.

The remainder of the method declaration should be familiar as it is identical to a procedural program that two parameters as arguments to the program.

```
Method-id. SayHello static.

linkage section.
01 your-name     pic x(10).
01 your-greeting pic x(20).

procedure division using by reference your-name
            returning your-greeting.
    move "hello " & your-name to your-greeting
    .
end method
```


Copyright © Micro Focus 2015-2016. All rights reserved.

Let's look at the procedural program that invokes this method:

```
invoke type MyClass::SayHello(your-name)
                       returning your-greeting
```

The INVOKE keyword is synonymous with CALL but is used in the context of calling a method on a class.

The TYPE keyword allows us to specify the name of the class we are referring to.

The :: syntax allows us to refer to the specific method on the class we wish to invoke.

The using (and returning) statement allows us to pass in the parameters we need to supply to the method, as we would if this were a CALL to a COBOL program.

Before we go deeper let's review some more aspects of the syntax.

```
invoke type MyClass::SayHello(your-name)
```

The TYPE keyword is a new part of the COBOL language introduced with Visual COBOL and simplifies how you reference and invoke methods.

To illustrate this, here is the equivalent program conforming to ISO syntax:

```
program-id. TestMyClass
repository.
class MyClass as "MyClass".

procedure division.

      invoke MyClass "SayHello" using by reference "Scot"

end program.
```

Visual COBOL simplifies other aspects of the COBOL language, let's look at a couple of cases in our example:

```
invoke type MyClass::SayHello using(your-name)
```

Can become:

```
invoke type MyClass::SayHello("Scot")
```

If the method contained further arguments these might appear as:

```
invoke type MyClass::SayHello("Scot", 37, "Bristol Street")
```

In fact, even the commas separating parameters are optional.

In future examples, we'll use this abbreviated syntax.

The method can also be simplified as:

Copyright © Micro Focus 2015-2016. All rights reserved.

```
method-id. SayHello static.

procedure division using by reference your-name     as string
                                returning your-greeting as string.
    move "hello " & your-name to your-greeting
    .
end method.
end class.
```

Two important things have changed here:

First of all, the explicit linkage section has been removed and the linkage argument been defined in-line with the procedure division using statement.

Secondly, the PIC X(20) argument has been replaced by a reference to string.

String is a predefined COBOL type which maps onto the Java and .Net string class. Strings contain a variety of methods and are used to hold Unicode data of an arbitrary length. The compiler can convert between many of the pre-defined types such as string into COBOL types such as PIC X, we'll look at this in more detail later on.

For future examples, we'll adopt this convention of defining arguments inline. However, this is only possible when we use pre-defined managed types. COBOL records still need to be defined in the usual way.

You can see this example in use in the workspace **18_02_Simple_Class2**

Objects

Our simple example so far has helped demonstrate the basic concept of a class but the value of Object Oriented Programming (OOP) is not yet apparent. The power of OO really comes into play when we encapsulate data within a class, provide methods that perform actions on that data and then create instances of the class for use at runtime.

Creating an **instance** of a class results in the creation of an object. Each object maintains a separate set of data items that the methods act upon.

You can create many instances of a class, so therefore you can have many objects each with data distinct from other objects in the system.

For example, if we considered the kind of data we might need in a simple bank account class, we might think of such things as account number, balance and some way in which we could store transactions. At runtime, we could conceivably create a unique object for each customer we were dealing with where each object maintains distinct data from other customers at our bank.

In our first example, we did not define any data in our class and we didn't create an object at all but we were still able to invoke the method. This was due to the STATIC clause on the method which can call be applied to data.

```
method-id SayHello static.
```


Copyright © Micro Focus 2015-2016. All rights reserved.

We can invoke static methods directly of the class itself without creating an instance of the class.

Static methods and static data can be useful at times but there is only ever one instance of the static data. Static methods can only operate on static data.

Creating an instance of a class

Let's change our class a little and look at how we would create an object instance.

```
class-id. MyClass.
working-storage section.
01  your-name pic x(10) property.

method-id. SayHello.

procedure division returning your-greeting as string.
    move "hello " & your-name to your-greeting
    .
end method.
end class.
```

The **static** clause has been removed from the method.
The method no longer accepts an argument.
The class now has some data in the working-storage section.

To invoke the **SayHello** method, we now do this using an object rather than the class. Here's how we create that instance:

```
program-id. TestMyClass.
01  your-greeting  pic x(20).
01  an-obj type MyClass.
procedure division.
   set an-obj to new MyClass
   move 'Scot' to an-obj::your-name
   invoke an-obj::SayHello returning your-greeting
   display your-greeting
   stop "Press <CR> to terminate"
```

This is the declaration of the object, more formally known as an object reference.

```
01 an-obj type MyClass.
```

If we were to try and invoke the SayHello method on this class at this point, we would get a runtime error. That's because the object has not yet been created.

```
set an-obj to new MyClass
```

This is the line that creates the object. The keyword NEW is responsible for creating our object. NEW requires we specify the type of the object we want to create. This may seem strange as we have

Copyright © Micro Focus 2015-2016. All rights reserved.

already said what type our object is when we declared it but later on we'll see that an object can be declared as one type but at runtime, reference a different type.

The other thing we have done here is to set the value of my-name directly in the object instance data using:

```
move 'Scot' to an-obj::your-name
```

This is possible since the `property` clause has been added to the object instance data in the class.

```
01  your-name pic x(10) property.
```

This can be found in the workspace **18_03_Simple_object.**

The SET statement is frequently used in OOP and is synonymous with MOVE but applies to objects.

It is possible to declare another object reference and assign it the value of an-obj:

```
set another-obj to an-obj
```

In this case, another-obj now contains a reference to an-obj. It is important to note that whilst we have 2 object references, there is actually only 1 instance of type MyClass at this point and both another-obj and an-obj refer to it. If we invoked the SayHello method on `an-obj` and `another-object`, they would operate against the same data in the working-storage section.

The only way to create an entirely new object is to use the new keyword.

```
set another-obj to new MyClass
```

Our class has an issue at the moment. We have had to provide the name 'Scot' in a separate call to the object.

There are several ways we can fix this, one way is during the creation of the object which is otherwise known as construction. Right now, our class does not do anything during construction but we can do so if we create a method named new.

Constructors

```
method-id New.

procedure division using a-name as string.
      Set your-name to a-name
end method.
```

Whenever an object is created, the runtime system automatically invokes the New method on the class. If you didn't code one, the compiler automatically creates one for you.

In our method above, not only have we defined a constructor but we have also specified that it take a parameter. Given this, we need to change our code that creates the object:

```
set an-obj to new MyClass("Scot")
```


Copyright © Micro Focus 2015-2016. All rights reserved.

By the way, this code could also have been written as:

```
set an-obj to MyClass::New("Scot")
```

This can be found in the workspace **18_04_Simple_object2.**

What we have done is provide a way for our object to be initialized and ensured that we get an argument passed to the constructor any time an object of type MyClass is created.

However, it is possible to have multiple versions of the new method, each corresponding to different arguments that can be passed in when the object is created. This is called Method overloading because the method name remains the same but different arguments are accepted by each method.

We can also use this ability of Method overloading to re-instate the so-called default constructor otherwise known as the parameterless constructor. To do so, we just code a new New method.

```
method-id New.

procedure division.
   move all 'x' to your-name
end method.
```

This has allowed us to create the object by either supplying a parameter or using the default constructor which takes no arguments but still allows us to initialize our working-storage data.

Recap

So far we've seen how you can create a class with static methods and instance methods declare data, initialize our data using by defining a constructor and we've also looked at method overloading. We've also seen how you create an instance of a class and invoke a method on it.

If you have Visual COBOL installed, now would be a good time to either type this code yourself and step through in the debugger or to load an examine the 4 solutions already provided.

To do this, either create a JVM COBOL project in Eclipse or a managed COBOL console application in Visual Studio.

Properties

We already saw briefly how the property clause allows us to access object instance data directly. Let's look some more at this.

Our class has some data associated with it, a string called **your-name**. This data is not, by default, accessible directly by the program using the class just as the working-storage of one program is not accessible to another program.

Properties allow you to expose your data items to the user of your class.

Currently, our single data item looked like this:

```
01 your-name pic x(10).
```


Copyright © Micro Focus 2015-2016. All rights reserved.

We can turn this data item into a property as follows:

```
01 your-name pic x(10) property.
```

As such, you can now access this property through an object reference as we saw above

```
display an-obj::your-name
```

The property keyword allows us not only to get the value of a data item but we can also set it.

```
set an-obj::your-name to "Scot"
```

However, we can prevent anyone setting the value as follows:

```
01 your-name pic x(10) property with no set.
```

The **case** of your types and properties is important in the .NET framework especially when working with languages such as C#. The case of our property name is also taken from the declaration which is currently all lower case. We can change the name and case as follows:

```
01 your-name pic x(10) property as "Name".
…
display an-obj::Name
```

Whilst we're looking at properties, let's return to the subject of the STATIC clause which can also be applied to properties:

```
01 dataitem pic x(10) property as "DataItem" static.
```

If we recall, there is only ever 1 instance of a static data item regardless of how many objects have been created. Static data items are referenced through the class itself, we do not need an instance to access them:

```
set MyClass:DataItem to "some text"
```

Method Visibility

The Methods we have defined so far have all been public, the default for COBOL. A public method means that it can be invoked through the object reference. However, most classes have a need to for methods which we don't want to be visible outside of the class. Such methods are called private methods:

To declare a private method:

```
method-id ProcessData private.
```


Copyright © Micro Focus 2015-2016. All rights reserved.

This method could not be invoked through the object reference and if you tried you'd encounter a compiler error.

You can invoke this method from inside the class itself, say, from inside a public method.

```
method-id DoSomething public.
procedure division using ....
      invoke self::ProcessData          ...
end method.
```

Notice the use of the special keyword **self**. In this case, that just means invoke a method called ProcessData which is defined in this class.

Also note that we explicitly marked this method as public in its declaration. This isn't required as it is the default visibility but can be a useful reminder when first starting out.

Local Data

When writing procedural COBOL programs, we only have the choice of declaring all our data in the working-storage section. When working with classes, we still use the working-storage section for data that is associated with the class but we can also define data that is used only by a method.

```
method-id ProcessData private.
Local-storage section.

procedure division.

 perform varying counter as binary-long from 1 by 1 until counter > 10

...
```

In this example, our method has access to a local variable called counter. The lifetime and scope of this variable is entirely associated with the execution of this method. This field cannot be referenced outside of the method and on every invocation of the method, counter is set to its default value.

Recap

We've now covered properties and method visibility and we're probably 50% through the basics of OOP. One thing worth pointing out here is that classes can contain all of the regular COBOL semantics you use in procedural programming today.

For example, your class could contain a file section and methods can contain sections and paragraphs if desired and of course, you can use standard COBOL records within working-storage section.

Data Types

So far, our classes have used COBOL data types such as PIC X. All of the data types you use in COBOL today are supported in .NET but are only accessible to and understood by COBOL.

.NET and JVM do not have the concept of a PIC X or group record, let alone comp-1,-2,-3,-4,-5,-6,-X types. Whilst this is fine for your COBOL class and passing these types to other COBOL programs, they are not understood by other languages in the .NET framework.

Copyright © Micro Focus 2015-2016. All rights reserved.

To help transition COBOL types to other languages, there are a set of predefined types which are natively understood by .NET and JVM and map directly. These types are listed in the table at http://documentation.microfocus.com/help/index.jsp?topic=%2Fcom.microfocus.eclipse.infocenter.visualcobol.r4u2vc%2FH2TYRHTYPE01.html.

One example where all the following declarations refer to the same type:

```
01 val-1 binary-short.
01 val-2 pic s9(4) comp-5.
01 val-3 System.Int16
```

These are all different ways of saying the same thing, declaring a 16bit signed integer.

In C#, the equivalent declaration would use a type called **ushort**.

The point to remember here is that when working with classes, whatever data you expose to the caller of the class, be this as arguments to a method or as a property, it's generally best practice to use COBOL predefined types, as shown in the table.

However, in one of our previous examples we didn't do this, in fact we exposed a PIC X as a property. When we do this, the COBOL compiler is actually exposing the intrinsic String type, not the pic X field.

When a user of the property reads or sets it, the data is implicitly converted from native COBOL type to the .NET or JVM type, in this case a String.

Declaring a group item as a property actually exposes the whole group a .NET or JVM String type.

Native numeric types, such as comp-5, are coerced to the nearest managed code equivalent.

Inheritance

Inheritance is an important part of OOP. It allows us to create a new class by extending the functionality of an existing class. If we choose to, we can also change the behaviour of the class we are extending.

Let's consider a bank account example. We might imagine that accounts of any type, checking, savings, etc. share common data such as an account number field and a balance but the process of withdrawing money from an account might require different processing. A checking account may need to check whether an overdraft limit is in place and a savings account, which will not have an overdraft, will need to check other factors that affect interest earned, such as the amount of money that can be withdrawn within a given period.

An important consideration we'll look at later is that whatever is using these objects, let's say the ATM machine, should not need to determine the type of account it's dealing with and then perform different processing. It simply wants to perform a withdrawal action against whatever account object it is using.

For now though, let's just look at how we can both extend an existing class and customize behaviour.

Copyright © Micro Focus 2015-2016. All rights reserved.

Here's a simplistic account class:

```
Class-id BankAccount.

Working-storage section.
01 account-number 9(8) property as "AccountNumber".
01 balance float-long.


Method-id Withdraw.
Procedure division using amount as float-long
                   returning result as condition-value.

*> Process withdrawal
   …
End-method.
End-class.
```

This type of class, named BankAccount is often referred to as the base class as it forms the base of a hierarchy of classes that emanate from this one.

Let's create a new class to represent a specialization of the bank account, a savings account.

```
class-id SavingsAccount inherits BankAccount.

Method-id Withdraw override.
Procedure division using amount as float-long
                  returning result as condition-value.
End-method.
*> Specialized process for Savings withdrawal
End-class.
```

Besides defining a new class for savings accounts we have used the INHERITS clause to denote we are extending an existing class in the system. All public members (methods, properties, fields defined as PUBLIC) of the base class become part of the new class.

As such, an object that is of the type SavingsAccount, also has properties called AccountNumber, balance and a method named Withdraw which have been inherited from the base class BankAccount.

Our SavingsAccount class also has a method called Withdraw which will manage the different way in which money is withdrawn from a savings account. To indicate this is a change in behaviour to the method in the base class, we use the OVERRIDE keyword. The significance of this keyword will become more apparent later on.

```
class-id CheckingAccount inherits BankAccount.

…
```

Without fleshing out this new class, which also provides an override for the Withdraw method, we now have 3 classes in our class hierarchy.

Copyright © Micro Focus 2015-2016. All rights reserved.

Let's look at the effect of object instantiation and method invocation.

```
Program-id. TestBankAccounts.
01 account type BankAccount.

Procedure division.

  set account to new SavingsAccount
  set account::AccountNumber to "12345678"
  set account::Balance to 500.00

End-program.
```

The key point to notice is the declaration of our object's type, BankAccount, and the creation of it, as a SavingsAccount.

We can do this because SavingsAccount inherits (or descends) from BankAccount. The value of doing this is not so apparent in this example but this next might help:

```
Method-id PerformWithdrawal.

Procedure division using by value amount as float-long
                   account as type BankAccount.

 If Account::Withdraw(amount) not true
    *> perform error condition
   …
End-if
End-method.
```

In this case, a method receives an argument of type BankAccount from which it performs a withdrawal action. The method does not need to know about all the different types of accounts but whichever object type is passed in, the correct Withdraw method associated with that type is executed, be that a savings or checking account.

This is a very useful feature of OOP as it decouples implementation details from clients that use the classes. This in turn allows us to extend the system by adding new types of bank account but minimizing the impact on existing code.

Under both JVM and .NET, you can only inherit from one base class but, of course, the base class itself can inherit from a class and so on.

If a derived class needs to invoke the implementation of a method defined in the base class, it can do so using the **SUPER** keyword. For example, we can call the BankAccount WithDraw method from within the SavingsAccount class as follows:

```
invoke super::Withdraw(100)
```

SUPER can be used not only to invoke a method we have overridden in the derived class but to invoke any public method defined in the class hierarchy we have inherited.

Copyright © Micro Focus 2015-2016. All rights reserved.

Casting

At times, your application will need to work with the specific type of an object rather than the generic base type. If you find yourself with a reference to the base type, how do you get to the derived type?

```
Method-id…
local-storage section.
01 Savings type SavingsAccount.
Procedure division using by value account as type BankAccount.

Set Savings to account as SavingsAccount.
```

Here we cast the object passed as an argument to a specific derived type. Casting one object type to another must always be done with care as errors may occur if the type being cast at runtime is not the type stated at compile time.

The COBOL language includes syntax to help cater for these situations, such as TYPE OF, which can be used to test the type of an object before casting. More details of this syntax are referenced in the additional reading section.

Interfaces

Classes and inheritance allow us to decouple implementation details from the user of the class but there is another aspect of OOP that can help further decouple implementation, the interface.

An interface, like a class, defines a series of methods and possibly data too, but unlike a class, it does not provide any implementation within the methods. This is because the purpose of the interface is to merely define what behaviour a class will have – behaviour in this case being the methods and properties defined on the class.

Here's an example of an interface.

```
Interface-id ErrorHandler.

Method-id notifyError.
Procedure division using by value error-code as binary-short.
End-method.

Method-id notifyWarning.
Procedure division using by value warning-code as binary-short.
End-method.

End-interface.
```

This interface defines just two methods and from which we can probably deduce would be used for logging an error of some kind.

By defining a class that supports this interface, we are said to implement the interface.

```
Class-id MyErrorHandler implements ErrorHandler.

Method-id notifyError.
```


Copyright © Micro Focus 2015-2016. All rights reserved.

```
Procedure division using by value error-code as binary-short.
    *> display message box to the user
End-method.

Method-id notifyWarning.
Procedure division using by value warning-code as binary-short.
    *> depending on configuration, ignore this or print
    *> it to the console
End-method.

End-class.
```

The **IMPLEMENTS** keyword defines the interface we intend to provide an implementation for in this class and the compiler will check that all methods have been implemented correctly.

Unlike inheriting a class, which can only be done with a single class, you can implement as many interfaces as you like in a single class.

We can create an instance of our class and because we have implemented the ErrorHandler interface, we can pass an object reference of this class to any code that expects to be working with the ErrorHandler interface.

```
Class ProcessData.
Working-storage section.
01 error-handler-list as type List value null.

Method-id RegisterErrorHandler static.
Procedure division using error-handler type ErrorHandler.

    If error-handler-list = null
       Set error-handler-list to new List
    End-if

    Invoke error-handler-List::Add(error-handler)
End-method.

Method-id NotifyErrorHandlers static.
Local-storage section.
01 error-handler type ErrorHandler.

Procedure division using error-code as binary-short.
   Perform varying error-handler thru error-handler-list
          Invoke error-handler::NotifyError(error-code)
  End-perform
End-method.

Method-id DoProcessing.
Procedure division.
```


Copyright © Micro Focus 2015-2016. All rights reserved.

```
*> do something and possible call NotifyErrorHandlers when something
*> goes wrong
…
   Invoke self::NoitifyErrorHandlers(error-code)
…
End-method.

End-class.


Program-id TestProgram.
Working-storage section.
01 error-handler type MyErrorHandler.

Procedure division.
   Set error-handler to new MyErrorHandler
   Invoke ProcessData::RegisterErrorHandler(error-handler)
End-Program
```

Let's review this code as there are some new concepts as here.

First of all, we have a class, **ProcessData**, which during the method **DoProcessing** will at some point invoke interested parties that an error has occurred. It does this by invoking methods on the **ErrorHandler** interface.

This class has the capability to notify multiple parties as it allows clients to register their interface implementation using the **RegisterErrorHandler** method. Each interface is stored within a list object. We won't explore the list object now but let's assume such a class is provided to us by the .NET or JVM class frameworks.

When an error does occur and the **NotifyErrorHandlers** method is invoked, the code makes use of feature of the Visual COBOL syntax that allows it to iterate through the collection of error handler interfaces contained in the list. Each iteration results in the error-handler local-storage object reference being set to the next item in the list. The code simply calls the **NotifyError** method and the implement of this decides what to do about it.

The **TestProgram** constructs an instance of **MyErrorHandler** and passes this as an argument to the **RegisterErrorHandler** method. This call involves an implicit cast from the type **MyErrorHandler**, a class, to the type **ErrorHandler**, an interface.

Copyright © Micro Focus 2015-2016. All rights reserved.

Class names

So far, our classes have had simple names but this could soon lead to clashes with classes created by other people. To resolve this, we simply create classes with longer names and employ the use of **namespaces** which is nothing more than a convention for naming classes.

Here's a fully qualified class name:

```
com.acme.MyClass
```

`MyClass` is a different class from the follow:

```
com.yourcompany.MyClass
```

Everything leading up to the class name is considered a namespace if working in .NET or a package name if working in JVM. In this case, com.acme and com.yourcompany.

This convention allows us to create classes that do not conflict with others classes of the same name.

Whilst this is a naming convention, compilers provide directives and syntax to make working with namespace easier and in fact, there can be rules certain rules about the accessibility of classes within namespaces.

When you reference a class that has a namespace you need to use its fully qualified name. For example:

```
01 an-obj  type com.acme.MyClass.
```

```
01 another type obj com.yourcompany.MyClass.
```

The COBOL compiler provides a directive that allows you to use the abbreviated name of a class.

$set ILUSING(com.acme)

When using this directive in a source file, you can then reference the shortened name:

01 an-obj type MyClass.

Whilst this is generally accepted practice, as class names can otherwise become quite long, you should avoid needlessly importing lots of namespaces as it defeats the whole purpose of including classes in namespace and packages. Besides, you may find you encounter a clash of class names and in which case, you will need to disambiguate the class name by specifying the full class name.

Intrinsic types

The COBOL compiler is aware of several classes within the .NET and JVM frameworks and does not require you to specify its fully qualified name. The two we'll look at are String (System.String in .NET and java.lang.String in JVM) and Object (System.Object and java.lang.Object).

Object is important because all classes ultimately inherit from this type, whether you specify it or not. Therefore, any object can be cast to this type.

Copyright © Micro Focus 2015-2016. All rights reserved.

String is used commonly and is used for storing Unicode data. In both JVM and .NET, the string once created is considered immutable. Whichever method on the string class you invoke, the result is a new string object.

```
01 str-1 type System.String.
01 str-2 String.
01 str   string.
```

All of the above declarations in .NET are the same.

Notice there is no need to call the new method when creating a string:

```
set str-1 to "Hello World"
```

You can combine strings with regular pic X fields:

```
01 a-pic-x pic X(10) value "something".

display a-pic-x & str1 & "blah"
```

Here's an example of using one of the many string methods:

```
set str-1 to str-1::Replace("foo", "bar")
```

Notice how we assigned the result of this method to the original object. If we did not, str1 would have remained unchanged.

The .NET and JVM frameworks

.NET and JVM provide huge frameworks of classes that provide all manner of functionality. Learning all the classes in these frameworks can take a long time but there are many that you should get to know quickly, particularly the collection classes.

To help illustrate the usefulness of these frameworks, let's look at just one area, date and time arithmetic.

```
01 dt1 type System.DateTime.
01 dt2 type System.DateTime.
01 ts type System.TimeSpan.
…
Set dt1 to type System.DateTime::Now

Invoke System.Threading.Thread::Sleep(1000)

Set dt2 to type System.DateTime::Now
Set ts to dt2 - dt1

display ts
```

This example makes light work of date time arithmetic. Let's review what we've done:

First of all, we've declared 3 object references, 2 DateTime objects and 1 TimeSpan object.

Copyright © Micro Focus 2015-2016. All rights reserved.

The DateTime class provides an extensive set of routines for manipulating dates and times. To get an idea of its capabilities, take a look at the documentation:
http://msdn.microsoft.com/en-us/library/system.datetime.aspx

The TimeSpan class is used when calculating the difference between two DateTime objects.

In the first line of code, we initialize the dt1 object reference using a static method on the System.DateTime class, Now. There are many other ways of initializing a DateTime object but this is a convenient way of getting the current date and time.

```
Set dt1 to type System.DateTime::Now
```

In this next line, we again make use of a static method that causes the current thread to sleep for a specified period. You could invoke the Micro Focus CBL_THREAD_SLEEP routine to achieve the same result.

```
invoke System.Threading.Thread::Sleep(1000)
```

The next line initializes our second DateTime object following the sleep.

```
Set dt2 to type System.DateTime::Now
```

The next line demonstrates a feature of the managed code COBOL compiler called operator overloading.

```
set ts to dt2 - dt1
```

Operator overloading is an advanced feature of OOP and worth taking a quick look at:

When defining a class, it is also possible to provide implementation of some arithmetic operators such as add and subtract. The DateTime class defines several operators for date and time arithmetic and comparison.

Whilst you can perform arithmetic on objects by using the operator overloads, classes usually provide equivalent methods you can invoke directly, as is the case for DateTime. The following line would achieve the same:

```
Set ts to dt2::Subtract(dt1)
```

The result of either approach results in a TimeSpan object. This object contains the result of the arithmetic expression.

Finally, we display the result. Whenever you use the DISPLAY verb with an object reference, the compiler automatically invokes the ToString method that is defined on the base class, Object. If you remember, all classes ultimately descend from Object. Ordinarily, the ToString method simply

Copyright © Micro Focus 2015-2016. All rights reserved.

returns the name of the Type, but the TimeSpan class overrides the ToString method and returns a meaningful string about the TimeSpan object.

```
display ts
```

Reflection

When you compile a COBOL program for .NET or JVM, the compiler creates an executable that conforms to the specification of these platforms. This specification must be adhered to by any language supporting .NET or JVM.

Because all languages conform to a common underlying protocol, it allows classes written in any language to be easily integrated.

Another advantage of this commonality is something called reflection. Reflection is the ability to examine the underlying details of a given type.

With this ability, it is possible to inspect the methods that a given type provides, the arguments to each method and even the code in the body of the method. This is a very powerful feature of the framework and opens up many possibilities for application development. Although it may not seem immediately obvious as to how reflection can be valuable, understanding that it possible can help when considering how various technologies in managed code can do what they do without having prior knowledge of how you

One example of the use of reflection is something called Intellisense. Intellisense is a feature of the Visual Studio and Eclipse IDEs that assists the developer by showing a list of methods and properties available on a given object.

What Next?

This appendix has provided a basic introduction to object oriented programming and has covered many of the fundamental concepts. OOP is however an extensive subject, there are many other areas to cover including the many technologies provided by the .NET and JVM platforms before one could feel confident enough to build a .NET or JVM application with COBOL.

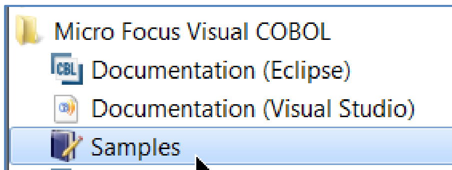
Aside from additional self-study, you should also consider a dedicated training course in C#, VB or Java. These courses will build upon your knowledge of OOP and enable you to build applications in Java, C# or COBOL as the principals remain the same across all of these languages – the key difference being syntax.

A great way to accelerate your understanding of OOP and managed code frameworks is to work directly with colleagues skilled in Visual Basic, C# or Java.

Further Reading

There are a series of examples that Micro Focus provides which you can explore as required. You will find these from your Windows start menu as:

Copyright © Micro Focus 2015-2016. All rights reserved.



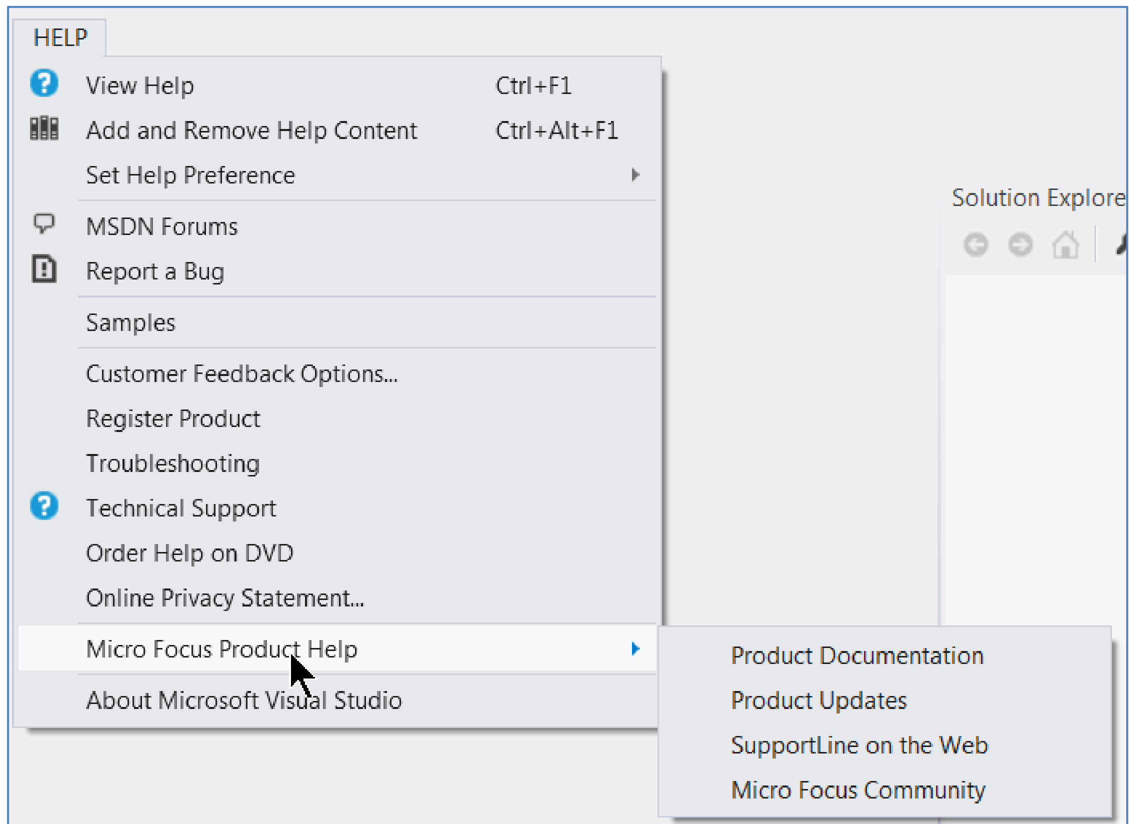
When you have got there you can explore many features:

Copyright © Micro Focus 2015-2016. All rights reserved.

20 Appendix II – Further Features

Introduction

This appendix lists a number of other language features of COBOL that have not been covered in the previous modules. These features will be covered very briefly in this appendix. If you require more detail then you will find this in the Visual Studio Help system:



If you select **Product Documentation** here you will be taken to the following:

Copyright © Micro Focus 2015-2016. All rights reserved.

- Help Viewer Home
  - Micro Focus Enterprise Developer
  - Micro Focus Visual COBOL for Visual Studio 2010
    - Welcome
    - Developing COBOL Applications
    - Programming
    - General Reference
      - C Functions for Calling COBOL
      - COBOL Language Reference
        - COBOL Language Supported and Copyrights
        - Notations
        - Part 1. Concepts
        - Part 2. Program Definition
        - Part 3. Additional Topics
        - Part 4: Appendices
      - Command Line Reference
      - Compiler Directives
      - Licensing
      - Environment Variables
      - Error Messages
      - File Handling Reference
      - Integrated Preprocessor Interface
      - Library Routines
      - Restrictions, Compiler Limits, and System Limits
      - Managed COBOL
      - Run-time System Configuration
    - Deployment
    - Copyright and Legal Information
    - Notations and Conventions

From here you will be able to explore the full features of COBOL.

For example, to get a full alphabetical list of all the reserved words you can look at the following:

Copyright © Micro Focus 2015-2016. All rights reserved.

- Help Viewer Home
  - Micro Focus Enterprise Developer
  - Micro Focus Visual COBOL for Visual Studio 2010
    - Welcome
    - Developing COBOL Applications
    - Programming
    - General Reference
      - C Functions for Calling COBOL
      - COBOL Language Reference
        - COBOL Language Supported and Copyrights
        - Notations
        - Part 1. Concepts
        - Part 2. Program Definition
        - Part 3. Additional Topics
        - Part 4: Appendices
          - Character Sets and Collating Sequences
          - ANSI File Status Summary
          - Reserved Words
            - Reserved Words Table
            - Context-sensitive Words Table
          - XML-CODE Exception Codes
          - Glossary

Module Objectives

The intention of this module is to briefly show you some additional features of COBOL which you may decide to use or may come across in other pieces of COBOL code.

Other Data File types

So far we have covered the use of:

- **Sequential data files**
- **Indexed data files**

COBOL also fully supports the use of:

- **Line sequential data files** (PC text files). These files are normally used for producing reports. The syntax for the SELECT statement is:
  ```
  SELECT REPORT-FILE ASSIGN REPNAME
         ORGANIZATION IS LINE SEQUENTIAL.
  ```
- **Relative data files**. These files are an alternative to using indexed files. They are not very commonly used. The syntax for the SELECT statement is:
  ```
  SELECT RE-FILE-FILE ASSIGN REPNAME
         ORGANIZATION IS RELATIVE.
  ```

Relative files do not need a key (although many do have keys stored on them). Without a key the access is physical offset in the file. If you want to read more about relative files, good starting points are:

- http://www.cse.ohio-state.edu/~sgomori/570/relcob.html or
- http://cayfer.bilkent.edu.tr/~cayfer/ctp108/relative.htm

Copyright © Micro Focus 2015-2016. All rights reserved.

Report writing

COBOL has extensive support of complex report writing. This was briefly mentioned in an earlier module and will not be covered here.

This is done in a new COBOL data division section called REPORT SECTION.

A good reference to check out the details of this is http://www.pgrocer.net/Cis52/rptwritr.html

Sorting data files

COBOL can directly SORT data files using the SORT verb and a new file definition for the sort file.

We will not look at this in any detail, but you may want to check out the following address for more details:

- http://theamericanprogrammer.com/programming/10-sortex1.shtml

Local-Storage Section

You will possible have seen that some methods have used LOCAL-STORAGE Section rather than WORKING-STORAGE section. To all intents the way we have used these so far is identical. For our needs so far, they are interchangeable.

However, this has another function. The presence of a LOCAL-STORAGE Section in a method indicates that this method can be used recursively. i.e. a method can call itself.

Intrinsic functions

COBOL supports a large number of “intrinsic” functions such as:

- Sine
- Square Root
- Cosine
- Tangent
- Annuity
- Date conversion
- Log
- Max, Min
- Random etc. etc.

The syntax for using an intrinsic function is illustrated by the use of the Square Root function:

```
COMPUTE WS-RESULT = FUNCTION SQRT (WS-NUMBER)
```

Library routines

In addition, Microfocus COBOL supports a number of library routines which you may find useful. E.g.

Copyright © Micro Focus 2015-2016. All rights reserved.

- Library Routines
  - Library Routines - Key
  - Application Subsystem Routines
  - Bit-packing Routines
  - Byte-stream File Routines
  - Character Set Conversion Routines
  - Consolidated Tracing Facility Routines
  - Container-Managed Services Routines
  - Debugging Routines
  - Display Attribute Routines
  - Enhanced ACCEPT and DISPLAY Syntax
  - Exit and Error Procedure Routines
  - File and Filename Routines
  - Keyboard Routines
  - Logical Operator Routines
  - Memory Allocation Routines
  - Mouse Routines
  - Multi-threading Routines
  - NLS Message-file Handling Routines
  - Operating System Information Routines
  - Portability Routines
  - Printer Routines
  - Program Canceling Routines
  - Program Information Routines
  - Run-unit Handling Routines
  - Screen Routines
  - State Maintenance Routines
  - Text Routines
  - Virtual Heap Routines
  - Windows Routines
  - Miscellaneous Routines
  - Alphabetical List of Library Routines

Module Summary

This appendix has briefly shown you some additional features of COBOL which you may decide to use or may come across in other pieces of COBOL code.

To explore these extra variants you should look inside the help system under Product Documentation.

Once you have selected Product Documentation. You can search for whatever you require.

In addition, there is a very healthy Micro Focus Community.

In particular you have also seen how Micro Focus provided you with a mechanism to generate COBOL code to access a relational database.

Copyright © Micro Focus 2015-2016. All rights reserved.