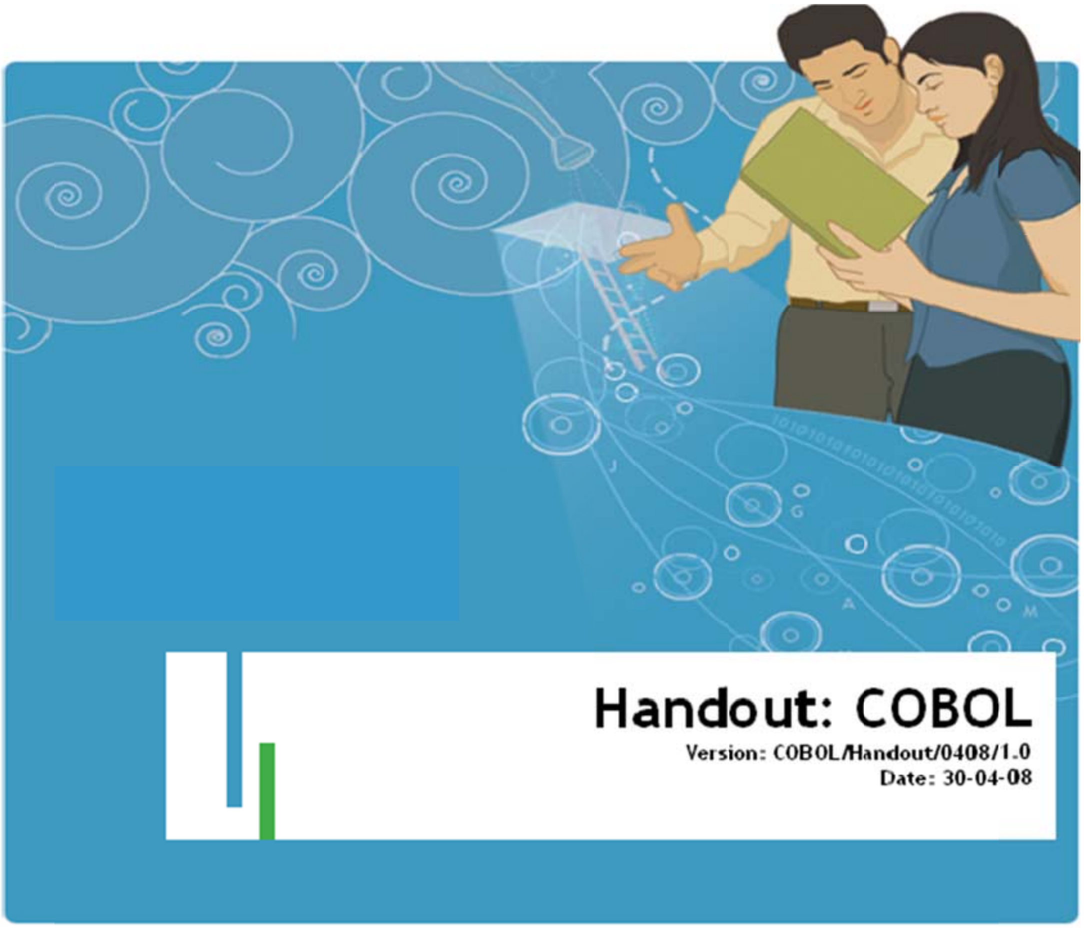
Cognizant
Passion for making a difference



Handout: COBOL

Version: COBOL/Handout/0408/1.0
Date: 30-04-08

Cognizant
500 Glen Pointe Center West
Teaneck, NJ 07666
Ph: 201-801-0233
www.cognizant.com

TABLE OF CONTENTS

Introduction

About this Module

This document provides an overview on the following topics:

❑ Phases involved in development of program

❑ Introduction to COBOL programming

❑ Basic concepts in COBOL programming

Target Audience

This module is designed for the entry level trainees.

Module Objectives

After completing this module, you will be able to:

❑ Explain the concepts involved in COBOL programming

❑ Identify how to code effectively in COBOL

❑ Describe the problem solving concepts in COBOL

Pre-requisite

To take-up the course in COBOL, the trainee should have a fair idea on the following topics:

❑ JCL

❑ VSAM

❑ File creation using option 3.2

❑ IBM Utilities like IEBGENER

Session 02: Introduction to COBOL

Learning Objectives

After completing this session, you will be able to:

- Explain the basic terminology used in programming
- Describe the significance of COBOL and its usage
- Explain the evolution of COBOL and the versions involved

Basic Terminology in COBOL Programming

- **Abend:** Abnormal termination of program.
- **ANSI (American National Standards Institute):** An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States.
- **Arithmetic Expression:** An identifier of a numeric elementary item, a numeric literal, such identifiers and literals separated by arithmetic operators, two arithmetic expressions separated by an arithmetic operator, or an arithmetic expression enclosed in parentheses.
- **Arithmetic Operation:** The process caused by the execution of an arithmetic statement, or the evaluation of an arithmetic expression, that results in a mathematically correct solution to the arguments presented.
- **Binary Search:** A dichotomizing search in which, at each step of the search, the set of data elements is divided by two; some appropriate action is taken in the case of an odd number.
- **Called Program:** A program that is the object of a CALL statement.
- **Calling Program:** A program that executes a CALL to another program.
- **Common Program:** A program which, despite being directly contained within another program, may be called from any program directly or indirectly contained in that other program.
- **Compiler:** A program that translates a program written in a higher level language into a machine language object program.
- **Condition:** A status of a program at run time for which a truth value can be determined. Where the term 'condition' (condition-1, condition-2,...) appears in these language specifications in or in reference to 'condition' (condition-1, condition-2,...) of a general format, it is a conditional expression consisting of either a simple condition optionally parenthesized, or a combined condition consisting of the syntactically correct combination of simple conditions, logical operators, and parentheses, for which a true value can be determined.
- **Copybook:** A file or library member containing a sequence of code that is included in the source program at compile time using the COPY statement. The file can be created by the user, supplied by COBOL, or supplied by another product.
- **Delimiter:** A character or a sequence of contiguous characters that identify the end of a string of characters and separate that string of characters from the following string of characters. A delimiter is not part of the string of characters that it delimits.

- **File:** A collection of logical records.
- **Function:** A temporary data item whose value is determined at the time the function is referenced during the execution of a statement.
- **Intrinsic function:** A pre-defined function, such as a commonly used arithmetic function, called by a built-in function reference.
- **Main Program:** In a hierarchy of programs and subroutines, the first program to receive control when the programs are run.
- **Nested Program:** A program that is directly contained within another program.
- **Object Code:** Output from a compiler or assembler that is itself executable machine code or is suitable for processing to produce executable machine code
- **Routine:** A set of statements in a COBOL program that causes the computer to perform an operation or series of related operations. In Language Environment, refers to a procedure, function, or subroutine.
- **Scope Terminator:** A COBOL reserved word that marks the end of certain Procedure Division statements. It may be either explicit (END-ADD, for example) or implicit (separator period).
- **Sentence:** A sequence of one or more statements, the last of which is terminated by a separator period.
- **Separator:** A character or two contiguous characters used to delimit character-strings.
- **Syntax:**
  - The relationship among characters or groups of characters, independent of their meanings or the manner of their interpretation and use.
  - The structure of expressions in a language.
  - The rules governing the structure of a language.
  - The relationship among symbols.
  - The rules for the construction of a statement.
- **Variable:** A data item whose value may be changed by execution of the object program. A variable used in an arithmetic expression must be a numeric elementary item.

Basic Development Process

- Initial analysis on program specifications:
  - **Identifying the requirements:** Once the objective is provided, the developer needs to perform an initial round of analysis and needs to come up with an understanding document.
  - **Input required and output to be generated:** Having a clear picture about the requirements, the next step is identifying the inputs that need to be processed and the outputs that needs to generated (or) updated.
- Design of the code:
  - **Detailed design:** Understanding the requirements in depth and provide specifications for the coder to proceed. The work products expected are Flowcharts, Pseudo Code

  - **High Level design:** To provide a snap shot of the requirements in the form of box diagrams, Hierarchy Charts. The idea here is to provide the overall business objective but does not focus on how on the code changes involved, modules impacted etc.
- **Coding:** To write programs / scripts to achieve the objective as per the analysis and design findings.
- **Compiling:** This process converts the source code to object code and performing initial validation such that there are no rule violations.
- Testing and Debugging:
  - Testing is to ensure that the code is executed as expected. Logical errors can be found out only by testing.
  - In order to correct the bugs found in testing, the code needs to be verified using one of the debugging techniques.
- **Documentation:** For future reference documentation is required

Debugging Techniques

- The following are the debugging techniques:
  - Manual verification of code after completing the program. This is done as part of the self-review where the developer validates the code.
  - Use Display or Print messages to ensure that a specific part of code is executed. This technique is adopted to identify the location of the bug.
  - Testing the code with specific test data for which the output is expected is already known. This technique is adopted if the scenario is known in which the program is not executing as expected.
  - Using debugging tools or utilities

About COBOL

There was a growing need in the early days to have a high level language that will be best suited for Business Applications. Hence In 1959, a new language COBOL – Common Business Oriented Language emerged.

COBOL is a High Level Structured Language. This is English like language, which is used to develop major business applications. Why most of business applications are developed using Mainframe COBOL?

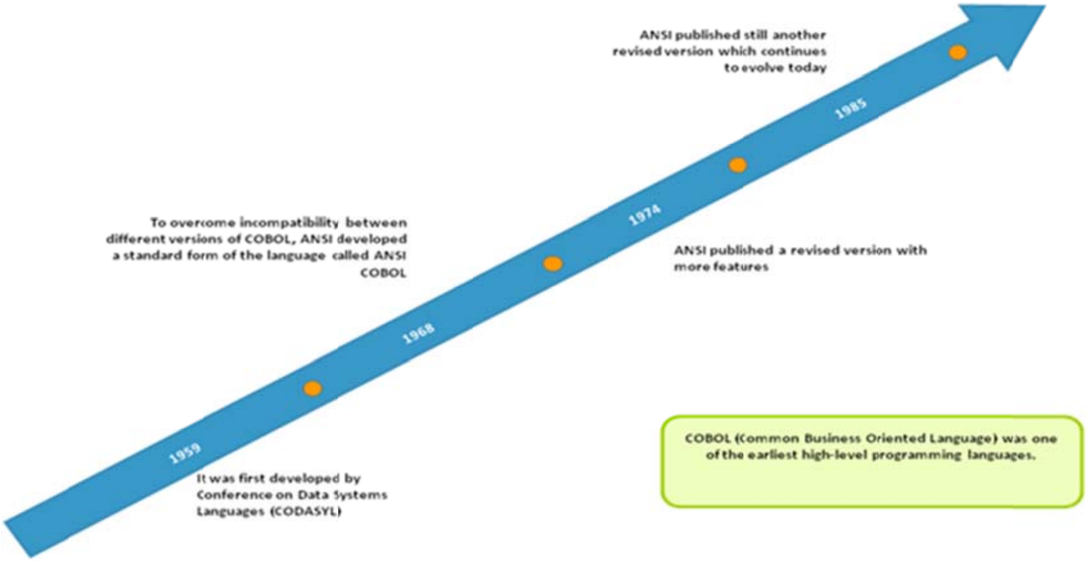
COBOL was the first programming language whose use was mandated by the Department of Defense (DoD)

- Cobol is an English like language and hence is easy to learn
- Can be used with any database like VSAM or DB2, IDMS
- Can handle huge volumes of Data
- COBOL applications can be easily maintained

History of COBOL

The following points highlight some salient facts about COBOL:

- Conceived in 1959
- Preliminary Version introduced in December 1959
- First Standard COBOL introduced in 1961
- ANSI Approved Versions introduced in 1968 and 1974
- Latest COBOL Version introduced in 1985
- Current version being used is COBOL II / COBOL 85



ANSI published still another revised version which continues to evolve today

To overcome incompatibility between different versions of COBOL, ANSI developed a standard form of the language called ANSI COBOL

ANSI published a revised version with more features

It was first developed by Conference on Data Systems Languages (CODASYL)

COBOL (Common Business Oriented Language) was one of the earliest high-level programming languages.

Great Moments in COBOL History

Year	Event	Description
1959	Origin of COBOL	A new English-like programming language, designed for writing business applications.
1962	Intended Next Release	First effort at a standardized version ... and look, it's only 2 years late!
1964	Release of COBOL 64	
1965	Intended Next Release	A few minor improvements over the original ... and look, it's only 3 years late!
1968	Release of COBOL 68	
1970	Intended Next Release	A few more minor improvements ... and look, it's only 4 years late!
1974	Release of COBOL 74	
1980	Intended Next Release	Somewhat controversial version - strayed from true English-like constructs ... and look, it's only 5 years late!
1985	Release of COBOL 85	
1990	Intended Next Release	Everything you could ever want and more ... and look, it's only??? Years late!
20??	Release of COBOL????	

Versions of COBOL

The following are the versions of COBOL:

- ❑ OS/VS COBOL:
  - No longer supported
  - The original COBOL on MVS systems
  - Limited focus on structured programming constructs
- ❑ VS Cobol II:
  - Very popular version of COBOL
  - Structured constructs such as scope terminators were introduced (END-IF, and so on)
  - Helped improved readability of programs
- ❑ COBOL for z/OS:
  - Also known as COBOL/370 and COBOL for OS/390
  - Provided more support for Language Environment and Object Oriented features
- ❑ Enterprise COBOL:
  - Current version on mainframe systems
  - Provide features like to support Web processing and Operating systems

Summary

- ❑ The basic terminologies applied in programming are Program, Software, and Applications Programs.
- ❑ The development process includes Analysis, Design, Coding, Compiling, Testing and Debugging, and Documentation.
- ❑ The debugging techniques consist of Manual Check, Print Statements, Using Test Data, and Tools.

Test Your Understanding

1. What is the expansion of ABEND?
2. Copy book can contain only file layouts. State true or false.
3. The latest version of COBOL is Enterprise COBOL. State true or false.
4. Which is the best debugging technique applicable when Test data and tools are not available?

Session 03: Program Structure

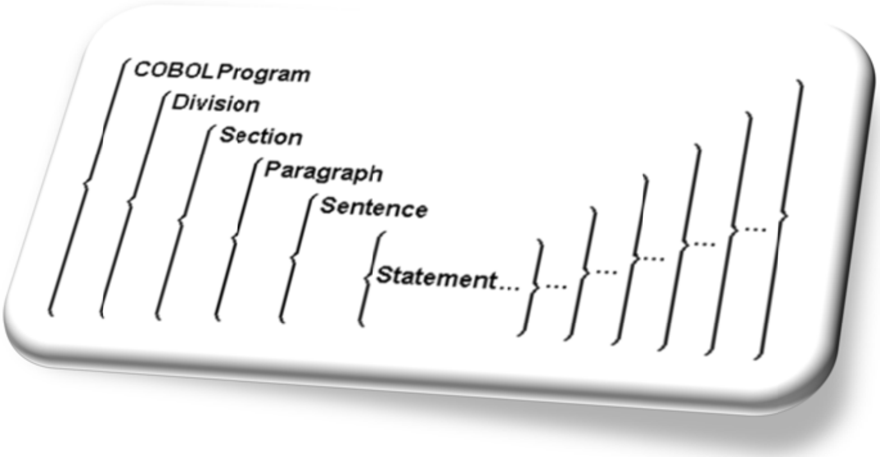
Learning Objectives

After completing this session, you will be able to:

- Describe the COBOL program layout
- Explain the structure of COBOL program

Structural Hierarchy

- COBOL programs are made up of familiar constructs such as paragraphs, sentences, statements, and clauses.
- These constructs, in turn contain elements such as words, names, verbs, and symbols.
- The hierarchy of a COBOL program is shown in the following diagram



- Division is a block of code, usually containing one or more sections:
  - It starts where the division name is encountered
  - It ends with the beginning of the next division or with the end of the program text
- A COBOL program is structured to have four divisions at a high level:
  - Identification Division
  - Environment Division
  - Data Division
  - Procedure Division

The COBOL divisions

COBOL language consists of 4 major Divisions namely:

- **IDENTIFICATION DIVISION:** The Identification Division names the program and, optionally, documents the date the program was written, the compilation date and other pertinent information.
- **ENVIRONMENT DIVISION:** This division contains machine dependent details such as Computer(s) used and peripheral devices.
- **DATA DIVISION:** The Data Division defines the nature and characteristics of all data the program is to process. This includes both Input/Output data and data used for internal processing.
- **PROCEDURE DIVISION:** The Procedure Division consists of executable statements that process the data in the manner the programmer defines. Unless the programmer defines some other order, the statements are executed in the order they are written.

The IDENTIFICATION DIVISION and ENVIRONMENT DIVISION are the first 2 divisions that need to be defined in the COBOL program. As their names indicate, these divisions contain entries that give the name of the program, author of the program, Source computer in which the program is written and the object computer in which the program may be run.

Let us consider a statement ADD PRINCIPAL, INTEREST TO AMOUNT.

The meaning of this statement is quite apparent. The principal and interest are added to get amount. The words PRINCIPAL, INTEREST, AMOUNT are called Data names. It is essential that all the data names which are used in PROCEDURE DIVISION needs to be defined at the outset.

Hence all the data definitions take place in DATA DIVISION. Hence DATA DIVISION appears before the PROCEDURE DIVISION and it is the third division in the COBOL program. The data definition includes the size of the data item; the type and any initial value, which needs to be set, are defined in DATA DIVISION entries.

The data that belongs to the input or output record is defined in FILE SECTION and the data names that holds the intermediate results are defined in the WORKING STORAGE SECTION.

Now that we have defined all the necessary data names and files to be used by the COBOL program, next comes the algorithm. The last division in the COBOL program is the PROCEDURE DIVISION where the actual algorithm of the program is written. The PROCEDURE DIVISION contains statements, which specify operations to be performed by the computer.

Thus these 4 divisions form the core structure of COBOL program. These divisions will be explained in detail in the following Sections.

Clauses and Statements

Clauses:

Written in Environment and Data Divisions: This specifies an attribute of an entry. A series of clauses, ending with period, is defined as an entry.

Statements:

Written in Procedure Division: Specify an action to be taken by the object program. A series of statements, ending with a period, is defined as a sentence.

Identification division

IDENTIFICATION DIVISION is the first division in the COBOL program. There may be several paragraphs in this division of which PROGRAM-ID is essential paragraph.

Each program has a Program id – Name of the program. Author – Person who is writing the program and other information like date written, date compiled. Since the information identifies a particular program, these are defined in **IDENTIFICATION DIVISION**, which is the First line of the COBOL program. Except PROGRAM-ID all the other paragraphs are optional and are mainly for documentation purposes.

Given below is the syntax for Identification Division:

```
{IDENTIFICATION DIVISION ID.}
PROGRAM-ID.  Program name
[AUTHOR.                      comment.]
[INSTALLATION.                comment.]
[DATE-WRITTEN.                comment.]
[DATE-COMPILED.
comment.]
[SECURITY.              comment.]
```

An explanation of the parts of Identification Division

PROGRAM-ID: The first paragraph of the Identification Division must be the PROGRAM-ID paragraph. The other paragraphs are optional, but, when written, must appear in the order shown in the format. This program names is used to identify the object program.

The Program name is a user-defined word that identifies your program. The system uses the first 8 characters of program-name of the outermost program as the identifying name of the program.

- The first 8 characters of program-name of the outermost program should be unique within the system. The first character must be alphabetic.
- If the first character is not alphabetic, it is converted as follows:
  - 1 through 9 are changed to A through I
  - Anything else is changed to J.
- If a hyphen is used in characters 2 through 8 of the program-name of the outermost program, it is changed to zero (0).

Apart from these statements, there can be comment statements describing the functionality of the program, and a change log block giving the details about programmers who changed the program as a part of enhancement to the program.

AUTHOR: This optional paragraph was removed from the standard in 1985, but most COBOL compilers still support its use. Code the programmer's name following the paragraph name. There are no formatting rules for the programmer's name. For example: AUTHOR. JANE DOE.

INSTALLATION: This optional paragraph was removed from the standard in 1985, but most COBOL compilers still support its use. Code the programmer's employer's name following the paragraph name. There are no formatting rules for the programmer's employer's name.

For example: INSTALLATION. MAJOR CORPORATION.

DATE-WRITTEN: This optional paragraph was removed from the standard in 1985, but most COBOL compilers still support its use. Code the date the program was written following the paragraph name. There are no formatting rules for the date.

For example: DATE-WRITTEN. JUNE 2001.

DATE-COMPILED: This optional paragraph was removed from the standard in 1985, but many COBOL compilers still support its use. Code the paragraph name only. The system date will be inserted in this area for documentation purposes.

For example: DATE-COMPILED.

SECURITY: This optional paragraph was removed from the standard in 1985, but most COBOL compilers still support its use. Code the security level of the program following the paragraph name. There are no formatting rules for the security level and this entry will have no actual effect on access to the program. The paragraph is for documentation purposes only.

For example: SECURITY. CLASSIFIED.

REMARKS: This optional paragraph was removed from the standard in 1985, but many COBOL compilers still support its use. Code the paragraph name followed by any desired comments, such as the primary purpose of the program. For example REMARKS. THIS IS THE MAIN SCREEN HANDLER FOR EMPLOYEE FILE MAINTENANCE.

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. SAMPLE.
 AUTHOR. COGNIZANT.
 DATE-WRITTEN.  NOVEMBER 22, 2007.
 DATE-COMPILED. NOVEMBER 24, 2007.
*================================================================*
*======             BRIEF PROGRAM DESCRIPTION              ======*
*================================================================*
* AUTHOR            : CHAITANYA (107592)
* DATE-WRITTEN      : NOVEMBER 22, 2007
* PROGRAM OBJECTIVE : BILL PROCESSING SYSTEM
* INPUT              : FROM JOB SYSIN
* OUTPUT             : TO JCL SYSOUT
* INPUT-OUTPUT       : NONE
* ENTRY FROM         : NOT APPLICABLE - MAIN PROGRAM
* REMARKS            : ELT CASE STUDY FOR PRESENTATION
*================================================================*
```

*** PROGRAM OBJECTIVE:** This Program Maintains the Employee file by updating the master file.

Environment division

The identification division gives introduction about the program. Now we need to define the Machine dependent details for our program. These details are given in ENVIRONMENT DIVISION. The ENVIRONMENT DIVISION must follow the IDENTIFICATION DIVISION in the COBOL program. Among all the divisions this one is the most machine dependent division. The computer and all the peripheral devices required by the program are described in this division.

The Environment Division consists of 2 sections:

- The Configuration Section
- The Input-Output Section

Syntax of Configuration Section

The configuration section appears first. The syntax is as follows:

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
[SOURCE-COMPUTER.       source m/c.]
[OBJECT-COMPUTER.       target m/c
      [PROGRAM COLLATING SEQUENCE IS alphabet-name].]
[SPECIAL NAMES.
      [, CURRENCY SIGN   IS literal-1]
      [, DECIMAL-POINT IS COMMA]
      [, alphabet-name IS  {STANDARD-1
                                STANDARD-2
                                EBCIDIC
                                NATIVE
                                lit-1 [ {THRU lit-2
                                     (ALSO lit-3)...}]...]...
         [, {SYSIN SYSIPT CONSOLE
          SYSPUNCH SYSPCH
          SYSOUT SYSLIST SYSLST
          C01 to C12 CSP S01 to S05}
                  IS mnemonic-name-1]...
      [, CLASS class-name-1 IS (lit-5 [THRU lit-6])...]...
      [, SYMBOLIC CHARACTERS
                  (symb-char-1 ... {IS ARE} int-1...) ...
                  [IN alphabet-name-1] ] ...
      .]
```

In the configuration section at least the source computer and object computer paragraph needs to be coded.

Paragraphs in Configuration Section

The paragraphs in the Configuration Section are given in the succeeding lines:

- SOURCE-COMPUTER Paragraph:
  - The Computer on which the source program is to be compiled.
  - The Computer Name in a system name in the form of IBM-3090
- OBJECT-COMPUTER Paragraph:
  - Identifies the computer on which the object program is to be executed.
  - The PROGRAM COLLATING SEQUENCE clause specifies that the collating sequence used in the program is the collating sequence associated with the specified alphabet-name, which must be defined in SPECIAL-NAMES Paragraph.
  - The program collating sequence is used to determine the truth value of following non-numeric comparisons:
    - Those explicitly specified in relation conditions.
    - Those explicitly specified in condition-name conditions.
- When the PRORAM COLLATING SEQUENCE clause is omitted, the EBCDIC Collating sequence is used.
- **The SPECIAL NAMES paragraph:** Can have entries, which are implementer dependent. Like the CURRENCY can be changed or the DECIMAL POINT can be COMMA.

For example:

```
SPECIAL NAMES. CURRENCY IS DOLLAR.
             DECIMAL POINT IS COMMA
```

Default collating sequence is EBCDIC on the IBM. Suppose we want ASCII.

Example 1:

```
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM 3090.
OBJECT-COMPUTER. VAX-6410
       PROGRAM COLLATING SEQUENCE IS ASCII-order.
SPECIAL-NAMES.
      ALPHABET ASCII-order IS STANDARD-1.
```

Example 2:

```
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM 3090.
OBJECT -COMPUTER. VAX-6410
PROGRAM COLLATING SEQUENCE IS ASCII-order.
SPECIAL-NAMES.
      ALPHABET ASCII-order IS STANDARD-1
      DECIMAL   IS COMMA
      CURRENCY IS  ‘#’
      SYMBOLIC CHARACTERS backspc IS 23
```

Input-Output Section

This section is primarily used to code the FILE-CONTROL paragraph.

FILE-CONTROL: This paragraph is used to associate the files to be used in the program with specific I/O devices. For example:

```
FILE-CONTROL.
SELECT PRINTER-FILE
ASSIGN TO SYS015-UR-1403-S.
```

In the example, PRINTER-FILE is the name of the file to be used throughout the program. SYS015-UR-1403-S is a system name, which will be used to link the file to something in the operating system. The method of such connection varies from system to system.

Data Division

The DATA DIVISION is where all of the data used by a program is defined. It is divided into five sections: the FILE, WORKING-STORAGE, LINKAGE, REPORT, and COMMUNICATION SECTIONs, although most programs only include the FILE and WORKING-STORAGE SECTIONs. The purpose of each of the sections follows:

FILE SECTION: The FILE SECTION is used to define the files that will be used in the program. There will be on file description (FD) for each file followed by a record description for each different record format associated with the file. For example:

```
FD    FILE-NAME.
01    RECORD-NAME.
      05    FIELD-NAME-1      PIC X(5).
      05    FIELD-NAME-2      PIC 99V99.
      05    FIELD-NAME-3      PIC X(20).
```

In the example, the file is named FILE-NAME-1. It is the same file name that was specified in the SELECT statement for the file. The 01 following the FD is called a record description entry. Multiple 01 entries can be included if there is more than one applicable record format for the file. See Data Items for further information on the record description and its subordinate items.

WORKING-STORAGE SECTION: The WORKING-STORAGE SECTION is used to define any data that will be used in the program that is not part of a file. These may include, but are not limited to accumulators, lookup tables, print line formats, save fields, etc. The structure of the items is as described in the Data Items section of this tutorial.

LINKAGE SECTION: The LINKAGE SECTION is used in a subprogram to define data that will be passed as arguments to the routine. The structure of the items is as described in the Data Items section of this tutorial. Each 01 level item in the LINKAGE SECTION should be included in the PROCEDURE DIVISION USING heading. For example:

```
PROCEDURE DIVISION USING      PARAMETER-1
                              PARAMETER-2
                              PARAMETER-3.
```

In this example, the subprogram would have 3 passed parameters called PARAMETER-1, PARAMETER-2, and PARAMETER-3. Each of them would be defined as an 01 level item in the LINKAGE SECTION. The order that the items are defined in the LINKAGE SECTION is unimportant.

REPORT SECTION: The REPORT SECTION can be used to produce standard control break driven reports. It is not commonly used.

COMMUNICATION SECTION: The COMMUNICATION SECTION is used for communicating between two programs running simultaneously on a computer which supports such operations. It is not commonly used.

Procedure Division

The PROCEDURE DIVISION consists of a series of procedures called paragraphs, each designed to perform a specific function. A paragraph consists of a paragraph name coded in Area A and a series of procedural statements designed to perform a desired function coded in Area B. Procedures frequently execute other procedures as indicated in the following example:

```
0000-MAIN-DRIVER.
     PERFORM 1000-INITIALIZATION-ROUTINE.
     PERFORM 2000-MAIN-PROCESSING
       UNTIL END-OF-FILE.
     PERFORM 3000-FINALIZE-ROUTINE.
     STOP RUN.
```

Notice the use of punctuation in the preceding paragraph. Each statement should be a sentence, that is, it should end with a period unless it is part of a conditional sentence. Sentences which are continued across multiple lines should be indented on the succeeding lines. Phrases which modify the behavior of a statement such as UNTIL in the PERFORM statement should be coded on a new line so that they stand out.

Rules of User-Defines Words

- Max. Length 30 Chars
- Must contain only Digits, Letters and Hyphens
- Must not be a Reserved Word
- Include at least one Letter
- Hyphens are imbedded
- Must not have Spaces

Examples of Reserved Words

Following are some examples of reserved words:

```
RECORD CONTAINS 100 TO 300   CHARACTERS DEPENDING ON file-
len
```

- **Keywords:** RECORD, DEPENDING
- **Optional Words:** CONTAINS, CHARACTERS ON
- **User defined Words:** file-len

Literals

Following are the main features of Literals:

- Literals are Constants
- Literals can be Numeric or Non-Numeric

Examples of Literals:

Numeric	Non-Numeric
123.345 9999 -100.95 0.54E-78	“COBOL PRG” “MY NAME”

Rules for Numeric Literals

Following rules should be kept in mind while creating numeric literals:

- Contains a minimum of 1 digit and can have maximum of 18 digits
- Sign, if used, must be left-most character and must not be more than one
- Decimal, if used, may not be right-most character and must not be more than one

Rules for Non-Numeric Literals

Following rules should be kept in mind while creating Non-numeric literals

- ❑ Any character in the Character Set allowed
- ❑ Must be enclosed in Quotes
- ❑ 2 continuous Quotes to get 1 Quote
- ❑ Numeric Literal in Quotes is Non-Numeric
- ❑ Maximum Length is 160 Characters

Rules for Floating Point Literals

Following rules should be kept in mind while floating Point literals:

- ❑ Useful for very high or very small values
- ❑ Specified as Numeric Literal with decimal point, followed by
- ❑ Letter E
- ❑ Sign
- ❑ 2-Digit Number
- ❑ Must fall between 0.54E-78 and 0.72E+76.

Introduction to Display Statement

Syntax - Format 1

```
                            { mnemonic-name-1  }
DISPLAY identifier-1 [UPON  {                  } ] [WITH NO ADVANCING]
                            { environment-name }
```

Description: The DISPLAY statement is used to exhibit data upon the primary output device. The field(s) listed after DISPLAY is exhibited.

Tips:

- ❑ Use the DISPLAY statement to exhibit small amounts of data such as error messages.
- ❑ The chief advantage of exhibiting data with the DISPLAY statement is that no file needs to be established to use it.
- ❑ The chief disadvantage of exhibiting data with the DISPLAY statement is that there is no mechanism to supply carriage control.

DISPLAY – Example

```
COBOL statement
DISPLAY  ‘Return code:’  i-ret-code
DISPLAY  ‘Return code:’  i-ret-code
            UPON SYSLIST
JCL statement
//SYOUT  DD  applicable parameters
//SYLIST  DD  applicable parameters
Default LRECL for SYSOUT is 121 and RECFM is FBA
```

```
Records will be folded if length > LRECL - 1.
```

Display: Rules

- Identifier-1 is converted automatically to external format, if required
- Negative signed values cause a low-order sign over punch
- Pointers are converted to an external PIC 9(10)
- INDEX names can’t be specified

Introduction to Exit Statement

Syntax - Format 1:

```
EXIT
```

Syntax - Format 2

```
EXIT PROGRAM
```

Description: The EXIT statement is used to exit routines or programs. Each of the two formats works slightly differently and is described in the correspondingly numbered area.

- If the first format is used, it must be the only entry in a paragraph. The EXIT statement itself does nothing. The same effect can be accomplished by leaving the paragraph empty.
- The second format is used to exit a subprogram. If the statement is executed from within a called program, control is returned to the calling program. If the statement is executed from within a main program, it has the same effect as the EXIT statement, that is, it does nothing.

Tips:

- Do not use the EXIT statement.
- Use the EXIT PROGRAM statement only in a called subprogram.
- If your system supports the GOBACK statement, use it in place of the EXIT PROGRAM statement in a called subprogram.

EXIT – Examples:

```
EXIT – Example
PROCEDURE DIVISION.
....
     PERFORM B1-PROCESS
      UNTIL  A > 5.
.....
B1-PROCESS.
….
      ADD 1 TO A.

     IF NOT-VALID-ACTION
      PERFORM B1-PROCESS-EXIT
...
B1-PROCESS-EXIT
      EXIT.
```

Introduction to Stop Statement

Syntax - Format 1:

$$\underline{\texttt{STOP}} \left\{ \begin{array}{l} \underline{\texttt{RUN}} \\ \textit{literal-1} \end{array} \right\}$$

Description: The STOP statement is used to stop programs. STOP RUN terminates the program. STOP literal displays the literal and waits for a response before continuing with the next executable statement.

Tips:

- ❑ Use the STOP RUN statement to terminate execution of a program.
- ❑ Do not use the STOP RUN statement in a called subprogram, as it will cause the termination of the entire run unit. Instead, to terminate execution of a subprogram, use the EXIT PROGRAM statement

Sample COBOL Program Layout

Following is a sample of a COBOL program layout:

```
DENTIFICATION DIVISION.
PROGRAM-ID. SAMPLE1.
AUTHOR. XYZ.
DATE-WRITTEN. 1-JUN-2000.
DATE-COMPILED.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-PC.
OBJECT-COMPUTER. IBM-PC.

DATA DIVISION.

PROCEDURE DIVISION.
0000-MAIN.
       STOP RUN.
```

Summary

- The structural hierarchy of a COBOL program consists of Divisions, Sections, Paragraphs, Sentence, and Statements.
- The various divisions involved are:
  - Identification division
  - Environment division
  - Data division
  - Procedure division
- Block of code containing one or more paragraphs are referred to as Sections
- Code blocks that are formed based on the logic performed are referred to as Paragraphs
- Sentences and Statements: Sentence consists of one or more statements and is terminated by a period.

Test Your Understanding

1). Which of the four COBOL divisions contains machine dependent details?

   a) Identification Division

   b) Environment Division

   c) Data Division

   d) Procedure Division

2). Which of the following defines AREA A in a COBOL program?

   a) Column 7 – 11

   b) Column 12 – 72

   c) Column 1 – 7

   d) Column 8 – 11

3). Paragraph name can begin in Area __.

Session 06: Introduction to Program Layout, Constants, Literals and Variables

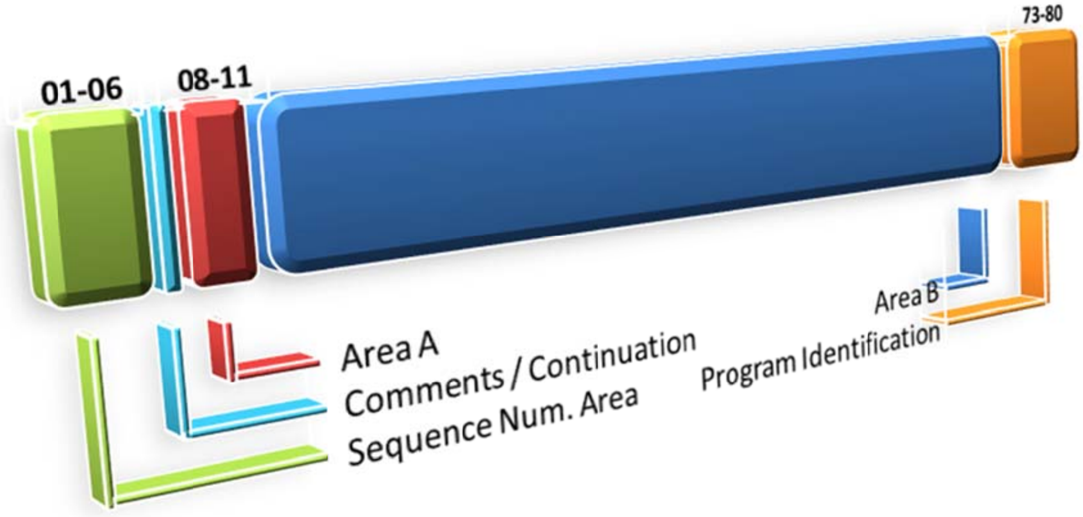
Learning Objectives

After completing this session, you will be able to:

- Identify the various data items in a COBOL program
- Declare data items
- Apply data items
- Write COBOL statements using the declared data items

Program Layout

COBOL programs must be written in the COBOL reference format. The below diagram illustrates the reference format for a COBOL source line.



The following areas are described below in terms of a 72-character line:

Sequence Number Area:

Columns 1 through 6: The sequence number area may be used to label a source statement line. The content of this area may consist of any character in the character set of the computer.

Indicator Area (comments and continuation)
Column 7:

- ❑ Any sentence, entry, clause, or phrase that requires more than one line can be continued in Area B of the next line which is not a
  - o Comment line
  - o Blank line
- ❑ Area A of a continuation line must be blank
- ❑ If no hyphen (-) in indicator Area
- ❑ Last character of the preceding line is assumed to be followed by a space
- ❑ **If hyphen is in Indicator Area:** First non-blank character of this continuation line immediately follows the last non-blank char of continues line
- ❑ If the continued line contains a non-numeric literal without a closing quotation mark:
  - o All spaces at the end of the continued line (through column 72) are part of the literal
  - o Continuation line must contain a hyphen in the indicator area
  - o First non-blank character must be a quotation mark and Literal
  - o Continues with the character following the quotation mark

Area A: Columns 8 through 11

The following items must begin in Area A:

- ❑ Division header
- ❑ Section header
- ❑ Paragraph header or paragraph name
- ❑ Level indicator or level-number (01 and 77)
- ❑ DECLARATIVES and END DECLARATIVES.
- ❑ End program header.

Area B: Columns 12 through 72

- ❑ Entries, sentences, statements, clauses
- ❑ Continuation lines.

Program Identification: Columns 73 through 80

- ❑ This area was originally designed for entering the program identification on each individual line of code
- ❑ Once cards were replaced with disk drives, this area fell into disuse

Picture Clause

The Picture clause is used to specify the type and size of an elementary data item. It consists of the word PIC or PICTURE followed by the actual picture clause, where the type and size are specified. The type is specified using a specific character selected from the table below. Each picture character represents one position in the data item. Different characters may be combined within the same Picture clause. Multiple occurrences of the same type of character are indicated by repeating the character or including a repetition factor after the character. For example, three alphabetic characters may be represented either with the clause PIC AAA or PIC A(3). Typically, repetitions of less than four characters are done by repeating the character and repetitions of four or more characters are done with the number enclosed in parentheses.

Character	Purpose	Notes
Data Representation Characters		
X	Any character	May contain any character including special or unprintable characters
9	Numeric character	May contain only the characters 0 through 9
A	Alphabetic character	May contain only the characters A - Z, a - z, and space
Arithmetic Positioning Characters		
S	Numeric sign	Does not use any actual storage space, without this character a numeric item would be treated as unsigned
V	Implied decimal point	Does not use any actual storage space, separates the integer portion of a numeric item from its decimal portion; used to keep track of the decimal point for arithmetic calculations; this character will not print or display; use the actual decimal point character, ".", for printing
P	Numeric place holder	Does not use any actual storage space; used to change precision, e.g. 99PPP will hold values in amounts of even 1,000 from zero to 99,000 without actually storing anything for the three lowest order positions; can also be used to represent positions after the decimal, e.g. PP9 will hold values in amounts in even thousandths, from 0 to .009 without actually storing anything for the two highest order positions
Numeric Editing Characters		
-	Minus sign	Used to display a sign in a printed item, the "-" character will be showed only if the numeric value is negative, the position will be left blank for positive values
+	Plus sign	Used to display a sign in a printed item, "-" will be shown for negative values, "+" will be shown for positive values
.	Actual decimal point	Separates the integer portion of a numeric item from its decimal portion; this character cannot be used in an item involved in a calculation; use the implied decimal point character, "V", for calculations
Z	Zero Suppress	Replaces leading zeros with blanks; has no effect on non-zero positions or zeros not in leading positions
,	Comma	Inserts a comma into the data item in this position; commonly used as a separator in numeric data

Character	Purpose	Notes
		items
$	Dollar sign	Inserts a dollar sign into the first position in the data item; commonly used for currency
*	Asterisk	Replaces leading zeros with asterisks; has no effect on non-zero positions or zeros not in leading positions; commonly used in check writing
CR	Credit	Used at the end of a PICTURE clause, the CR will appear in the position indicated if the item is negative; will be blank for a positive number
DB	Debit	Used at the end of a PICTURE clause, the DB will appear in the position indicated if the item is negative; will be blank for a positive number
/	Slash	Inserts a slash into the data item in this position; commonly used as a separator in date fields
0	Zero	Inserts a zero into the data item in this position; not commonly used

Floating PICTURE Characters: Certain of the characters above ($, -, +) have the ability to float. To float a character, simply repeat it in the leading positions of the item, along with separator commas, if desired. If the item has leading zeros, when it is filled, the leading positions will be filled with spaces, and the floating character will essentially move up to meet the actual data area. For example, filling an item with a picture of $$,$$$,$$$.99 with a value of 1047.53 will yield a value of $1,047.53 with four leading spaces.

SIGN Clause

```
[SIGN   IS     {LEADING
              TRAILING}]
       [SEPARATE CHARACTER]
```

- Specifies the position and mode of representation of the operational sign for a numeric entry.
- The Sign Clause can be specified for a signed numeric data description entry (that is, one whose PICTURE STRING contains an S)
- If a SIGN clause is specified in either an elementary or group entry subordinate to a group item for which a SIGN clause is specified, the SIGN clause for the subordinate entry takes precedence for the subordinate entry.
- The SEPARATE phrase indicates that sign is stores as separately and not in the zoned part.
- The TRAILING phrase indicates that sign is to be stored in the trailing position
- The LEADING phrase indicates that sign is to be stored in the leading position

SIGN Clause Examples:

```
05 l-V1 PIC S9(3) SIGN LEADING VALUE  K50
K stands for -2.
```

l-V1	K	5	0

```
05 l-V2 PIC S9(3) SIGN LEADING SEPARATE  VALUE  -250
```

l-V2	-	2	5	0

```
05 l-V2 PIC S9 (3) SIGN TRAILING VALUE  -250
The value stored internally in 3 characters as
```

l-V2	2	5	0

VALUE Clause

- The VALUE clause is used to set an initial value to the working storage variable.
- This can be specified only in the working storage variable.
- The VALUE Clause cannot be a part of RECORD DESCRIPTION entry in the FILE SECTION.

```
01 W04-G-VAR.
   03 W04-X-RUN-TIME              PIC X(20) VALUE SPACES.
   03 W04-X-PGM-NAME              PIC X(08) VALUE  'PXXBB290'.
01 W04-TEMP-J                     PIC 9(02) VALUE ZEROES.
```

In the above example the variable W04-X-PGM-NAME is set to an initial value of PXXBB290. The alphanumeric items can be initialized to Spaces and numeric items can be initialized to Zeros

Introduction to User defined variables

The main objectives of the chapter are:

- Identify the various data items in a COBOL program
- Can be declared in the Working Storage Section using PIC clause
- Example: 05 W01-STRING   PIC X(20).
  - The Variable W01-STRING is of Character Type(X) and of length twenty(20).
  - The prefix W denotes that the variable belongs to Working Storage section.
- Example: 05 W02-NUM   PIC 9(03) VALUE 100.
  - The Variable W02-NUM is of Numeric Type(9) and of length three(03).
  - The prefix W denotes that the variable belongs to Working Storage section.
  - The VALUE clause assigns the default value as 100.

Constants and Literals

Figurative Constants: COBOL has several built-in constants. These constants are called figurative constants. They are named according to their values. Most of them have alternative spellings available. Any of these spellings may be used to produce the same result. This is a list of all of the figurative constants:

ZERO:

- Alternate Spellings: ZEROS, ZEROES
- This figurative constant can be used to load a data item with as many zeros as it takes to fill the data item. ZEROS may be used to fill data items of any type.

SPACE:

- Alternate Spelling: SPACES
- This figurative constant can be used to load a data item with as many spaces as it takes to fill the data item. SPACES may not be used to fill numeric data items.

LOW-VALUE:

- Alternate Spelling: LOW-VALUES
- A LOW-VALUE is the lowest possible character in your machine's collating sequence. On most machines, that would mean that all of the bits in the character are set to 0's. This figurative constant can be used to load a data item with as many of this value as it takes to fill the data item. LOW-VALUES may not be used to fill numeric data items.

HIGH-VALUE:

- Alternate Spelling: HIGH-VALUES
- A HIGH-VALUE is the highest possible character in your machine's collating sequence. On most machines, that would mean that all of the bits in the character are set to 1's. This figurative constant can be used to load a data item with as many of this value as it takes to fill the data item. HIGH-VALUES may not be used to fill numeric data items.

QUOTE:

- Alternate Spelling: QUOTES
- A QUOTE is whatever character your compiler uses as a quote. In most COBOLs, this is the quotation mark. In some COBOLs, this is the apostrophe. This figurative constant can be used to load a data item with this character. QUOTES may not be used to fill numeric data items.

ALL literal: The literal is any non-numeric literal. This figurative constant can be used to load a data item with this character or character string. The character or character string will be repeated as often as necessary to fill the data item. The ALL literal may not be used to fill numeric data items.

Literals:

- **Numeric Literals:** Numeric literals can consist of any combination of numeric characters. No more than 18 total digits are allowed. One decimal point may be included and a sign (either a plus or minus sign) may be included in the front of the item. If the sign is not included, the value is assumed to be positive.
- **Non-Numeric Literals:** Non-numeric literals must be enclosed in quotes. Any character may be included except a quote. If it is necessary for a quote to appear in the literal, code two quotes adjacent to each other in the literal. They will be interpreted by the compiler as one quote and only one quote will appear in the resultant literal. Most COBOLs use the quotation mark for the quote. Some use the apostrophe. Some COBOLs allow either to be used; in such COBOLs, it possible to include a quotation mark in the literal by enclosing the literal in apostrophes and to include an apostrophe in the literal be enclosing the literal in quotation marks.

Data Items - Level Numbers

Levels 01-49 (Data Structuring Items):

- Levels 01-49 are used to create hierarchical data items. The highest level item in the hierarchy will always be the 01 item and every item contains the items beneath it with higher level numbers (up until the next item with a lower level number or the end of the section or division). Items which contain other data items are called group items and those which don't contain other data items are called elementary items.
- Although all of the numbers from 01-49 are available, by convention usually only the levels 01, 05, 10, 15, 20, 25, etc. are used.

```
Sample
01 A.
   05 B.
      10 C PIC X(20).
      10 D PIC X(10).
   05 E.
      10 F PIC X(5).
      10 G PIC 9(10).
```

In this example, there are three group level items, A, B, and E and there are four elementary items, C, D, F, and G. The sizes of the elementary items are determined by their picture clauses. See picture clauses below for more information on this. The sizes of the group items are determined by the sizes of the elementary items below them. Since B consists of the elementary items C (20 characters) and D (10 characters), B is 30 characters in length. Since E consists of the elementary items F (5 characters) and G (10 characters), E is 15 characters in length. Since A consists of the sub-groups B (30 characters) and E (15 characters), A is 45 characters in length.

Level 66 (Renaming Items):

Sometimes it is desirable to group items from two separate groups into a group of their own. Say we want to create a new group of items consisting of items D and F, but we don't want to interfere with these data items also belonging to groups B and E.

```
Sample
01 A.
   05 B.
      10 C PIC X(20).
      10 D PIC X(10).
   05 E.
      10 F PIC X(5).
      10 G PIC 9(10).
```

In this example, we could add a line after the last line, to create a new group from items D and F. To do this we would code:

```
66 H RENAMES D THRU F.
```

Following is the last line of the 01 level item. (In this case, the following line defines G)

Level 77 (Stand Alone Items):

- Level 77 items have been designated for deletion from the COBOL language and have been included here only in case you encounter one in an existing program. Level 77 items should not be coded in new programs.
- A level 77 item is used to define an item which is not to be subdivided any further. You can accomplish the same thing with a level 01 item, just don't subdivide it.

Level 88 (Condition Name Items): A level 88 item is used to give a name to a condition. Using a condition name has two primary benefits:

- It allows a descriptive name to be assigned to what otherwise might not be an obvious reason for a test.
- If a condition that is subject to change due to conditions beyond the control of the programmer and that condition is tested multiple places in the PROCEDURE DIVISION, using the condition name in the PROCEDURE DIVISION instead of the actual condition allows the programmer maintaining the program to change the values defining the condition in only one place in the DATA DIVISION rather than having to hunt down all of the individual occurrences in the PROCEDURE DIVISION.

```
Sample
05 COMPANY                        PIC 9(4).
   88 DOING-BUSINESS-IN-CALIFORNIA          VALUES 3
                                                   15
                                                   39 THRU 42
                                                   76.
```

To test for a company doing business in California, without the 88-level item, you would have to code:

```
IF COMPANY = 3 OR 15 OR 39 OR 40 OR 41 OR 42 OR 76
```

It would not be apparent to the person reading the code what condition you were testing for. With the 88 in place, you could code the test like this:

```
IF DOING-BUSINESS-IN-CALIFORNIA
```

Anybody reading the code could tell what condition you were testing for and if the check was done several times throughout the PROCEDURE DIVISION and the list of companies doing business in California changes, it is only necessary to change the list of values associated with the 88-level item.

While the data name associated with an 88-level item may have any value, the 88-level item itself can only have one of two values, true or false. The item is true if the data item immediately above the 88 has any of the values associated with the 88-item, otherwise it is false.

88-level items may be associated with either group or elementary items.

Introduction to MOVE Statement

Syntax - Format 1:

```
MOVE { identifier-1 } TO {identifier-2} ...
       literal-1
```

Syntax - Format 2:

```
MOVE { CORRESPONDING } identifier-1 TO {identifier-2} ...
       CORR
```

Description: The MOVE statement is used to copy data items to other data items. Each of the two formats works slightly differently and is described in the correspondingly numbered area.

- The first format is used to copy a field or value to another field(s). The field or value listed between MOVE and TO is copied to the value of the field(s) following the TO. For example in MOVE A TO B, the value in A is copied to B. The value in A is unchanged. Furthermore, in MOVE C TO E F, the values in C is moved to the value in E, storing the answer in E and the value of C is also moved to F storing the answer in F. The value in C is unchanged.
- The second format is used to copy subordinate field(s) of one group to subordinate field(s) in another. Those subordinate field(s) of the group item identifier-1 are copied to and those with exactly the same name in the group item identifier-2. The names of the subordinate items must be spelled exactly the same way in both groups to qualify to participate in the move.

Tips:

- If the receiving field is edited, any editing required will be done at the time of the MOVE.
- If the receiving field is unedited and the sending field is edited, the field will be de-edited at the time of the MOVE.
- If the sending field is shorter than the receiving field:
  - Numeric fields are padded on the left with zeros.
  - Non-numeric fields are padded on the right with spaces.
- If the sending field is longer than the receiving field:
  - Numeric fields are truncated on the left.
  - Non-numeric fields are truncated on the right.

Use of CORRESPONDING in MOVE Statement

- Both identifiers must be group items.
- Both identifiers following the keyword CORRESPONDING must name group items. In this discussion, these identifiers are referred to as identifier-1 and identifier-2.
- A pair of data items (subordinate items), one from identifier-1 and one from identifier-2, correspond if the following conditions are true:
- In a MOVE statement, at least one of the data items is an elementary item, and the move is permitted by the move rules.
- The subordinate items are not identified by the keyword FILLER.
- Neither identifier-1 nor identifier-2 is described as a level 66, 77, or 88 item, nor is either described as a USAGE IS INDEX item. Neither identifier-1 nor identifier-2 can be reference-modified.
- The subordinate items do not include a REDEFINES, RENAMES, OCCURS, or USAGE IS INDEX clause in their descriptions.
- However, identifier-1 and identifier-2, they can contain or be subordinate to items containing a REDEFINES or OCCURS clause in their descriptions.
- Neither identifier-1 nor identifier-2 nor the two subordinate items are described as USAGE IS POINTER items.
- Identifier-1 and/or identifier-2 can be subordinate to a FILLER item.

Valid and Invalid Elementary Moves:

Sending Item	Receiving Item				
	Alphabetic	Alphanumeric	Alpha numeric Edited	Numeric	Numeric Edited
Alphabetic and SPACE	Yes	Yes	Yes	No	No
Alphanumeric(2)	Yes	Yes	Yes	Yes(3)	Yes(3)
Alphanumeric-Edited	Yes	Yes	Yes	No	No
Numeric Integer and ZERO(5)	No	Yes	Yes	Yes	Yes
Numeric Non-integer (6)	No	No	No	Yes	Yes
Numeric-Edited	No	Yes	Yes	Yes	Yes

- Includes DBCS data items.
- Includes nonnumeric literals.
- Figurative constants and nonnumeric literals must consist only of numeric characters and will be treated as numeric integer fields.
- Figurative constants and nonnumeric literals must consist only of numeric characters and will be treated as numeric integer fields. The ALL literal cannot be used as a sending item.

- Includes integer numeric literals.
- Includes non-integer numeric literals (for example, 3.142).
- Includes floating-point literals, external floating-point data items (USAGE DISPLAY), and internal floating-point data items (USAGE COMP-1 or USAGE COMP-2).
- Includes DBCS data-items, DBCS literals, and SPACE.

Quite often it is required to move some of the data item of one group to some other data item of other group. If the names of the corresponding data items are distinct then separate MOVE statements have to be used. But if the corresponding data items have identical names, then instead of separate MOVE statements, MOVE CORRESPONDING can be used.

Example:

```
PAY-REC.
02    ID-NUMBER         PIC   9(5).
02    NAME              PIC   X(25).
02    DEPARTMENT  PIC   X(20).
02    BASIC-PAY         PIC   9999V99.
02    FILLER            PIC   X(24).

PRINT-REC.
02    FILLER            PIC   X(5).
02    ID-NUMBER         PIC   Z(5).
02    FILLER            PIC   X(5).
02    NAME              PIC   X(25).
02    FILLER            PIC   X(5).
02    DEPARTMENT  PIC   X(20).
02    FILLER            PIC   X(5).
02    BASIC-PAY         PIC   ZZZZ.99
02    FILLER            PIC   X(5).
02    DEDUCTIONS        PIC   ZZZZ.99.
02    FILLER            PIC   X(5).
02    ALLOWANCES        PIC   ZZZZ.99
02    FILLER            PIC   X(5).
02    NET-PAY           PIC   ZZZZ.99

MOVE CORR PAY-REC TO PRINT-REC
```

In the earlier example the ID-NUMBER, NAME, DEPARTMENT, BASIC-PAY of PAY-REC will be moved the same data item of PRINT-REC.

This is equivalent to 4 move statements

- `MOVE ID-NUMBER OF PAY-REC TO ID-NUMBER OF PRINT-REC.`
- `MOVE NAME OF PAY-REC TO NAME OF PRINT-REC.`
- `MOVE DEPARTMENT OF PAY-REC TO DEPARTMENT OF PRINT-REC.`

- ❑ MOVE BASIC-PAY OF PAY-REC TO BASIC-PAY OF PRINT-REC.

Introduction to PERFORM Statement

Syntax - Format 1:

```
                              THROUGH
PERFORM [procedure-name-1 [{         } procedure-name-2 ] ]
                              THRU

[imperative-statement-1 END-PERFORM]
```

Syntax - Format 2:

```
                              THROUGH
PERFORM [procedure-name-1 [{         } procedure-name-2 ] ]
                              THRU

  identifier-1
{              } TIMES [imperative-statement-1 END-PERFORM]
  integer-1
```

Syntax - Format 3:

```
                              THROUGH
PERFORM [procedure-name-1 [{         } procedure-name-2 ] ]
                              THRU

                BEFORE
[ WITH TEST  {         }] UNTIL condition-1
                AFTER

[imperative-statement-1 END-PERFORM]
```

Syntax - Format 4:

```
                                THROUGH
PERFORM [procedure-name-1  [ {         }  procedure-name-2 ] ]
                                THRU

                        }
               BEFORE
[ WITH TEST {         }
               AFTER
                        ]

                                       identifier-3            identifier-4
VARYING { identifier-2  } FROM {       index-name-2  } BY {               }
          index-name-1                 literal-1               literal-2

UNTIL condition-2

                                       identifier-5            identifier-5
[AFTER { identifier-4  } FROM {        index-name-4  } BY {               }
         index-name-3                  literal-3               literal-2

UNTIL condition-3] ...
[imperative-statement-1 END-PERFORM]
```

Description: The PERFORM statement is used to execute code. There are two different modes of operation.

The first mode is used to execute code in another area of the program and is invoked by mentioning a procedure or paragraph name. In this mode, the code in the other paragraph(s) is executed and control is returned to the statement following the PERFORM statement.

The second mode is called an in-line PERFORM, because the code to be executed is actually coded in place inside of the PERFORM statement. In the second mode, the code which is executed is that which lies between the PERFORM and the END-PERFORM.

Each of the four formats works slightly differently and is described in the correspondingly numbered area.

- The first format is used to execute the referenced code one time.
- The second format is used to execute the referenced code multiple times. The actual number of times is indicated by identifier-1 or integer-1.
- The third format is also used to execute the referenced multiple times. The actual number of times is dependent upon condition-1. The code will be repeatedly executed until condition-1 is true. The condition will be tested before the code is actually tested unless the WITH TEST AFTER phrase is included.
- The fourth format is used to execute the referenced multiple times with the actual number of times being dependent upon a condition like the third format. The primary difference between format 3 and format 4 is the VARYING phrase, which is used to count in a variable during the repetitions. The VARYING variable is set to the FROM value before the referenced code is executed. Before each subsequent pass, the VARYING variable is incremented by the amount of the BY value. The optional AFTER phrase may be included to vary the value of an additional variable. If this option is used, AFTER variable will be incremented through its entire cycle for each time the VARYING variable is cycled.

Tips:

- Use the PERFORM statement to access code which needs to be executed from more than one location.
- Use the PERFORM statement to structure your code. Make a main routine that PERFORMs other routines which in turn PERFORM other routines which themselves PERFORM other routines until the level of complexity of the PERFORMed routines is easily comprehended.
- Use the WITH TEST AFTER option to force the code to be PERFORMed at least once.
- Use the VARYING ... AFTER option to pass through all the entries in multiple dimensional tables.
- If the number of lines in the routine is relatively small, use an in-line PERFORM instead of PERFORMing a separate paragraph. The next programmer will appreciate not having to look all through the program to find the one or two lines of code in the paragraph you PERFORMed.

Perform – Overlap Rules:

- When the performed procedures executes another PERFORM the procedures associated with the 2nd level PERFORM must be totally included in or totally excluded from the procedures of the first level PERFORM statement.
- Two or more such active PERFORM must not have a common exit.

Perform (Times option) Rules:

- Once the perform statement is initiated any changes to the id-1 will have no effect on the number of times the Para is to be executed
- If identifier-1 is zero or a negative number at the time the PERFORM statement is initiated, control passes to the statement following the PERFORM statement.

PERFORM (TIMES):

```
Example
PERFORM B1-PROCESS
        THRU B1-PROCESS-EXIT
              10 TIMES
```

In this example the number of times perform statement is going to be executed is predetermined. Hence this perform statement will be executed 10 times.

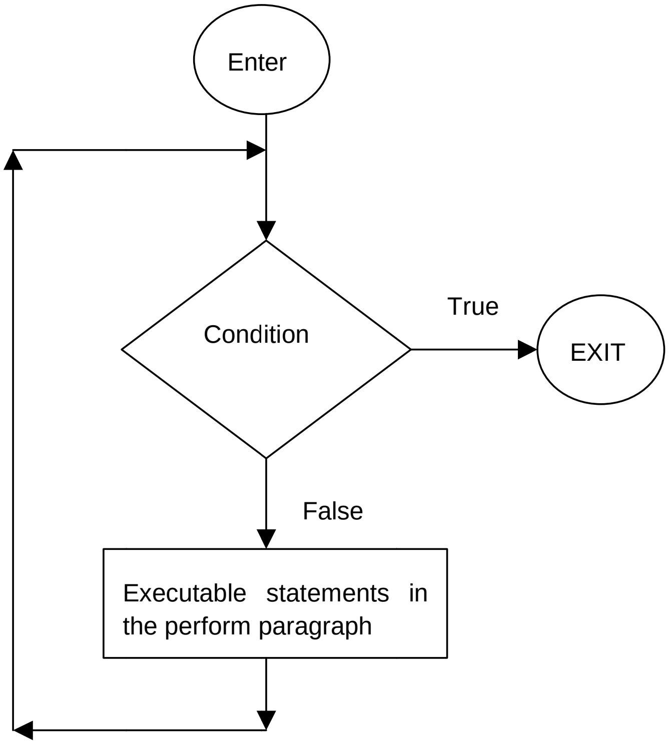
```
PERFORM I-CNT TIMES
    WRITE OUT-REC FROM I-ERR-REC (I-CNT)
    SUBTRACT 1 FROM I-CNT
END-PERFORM
```

- In this example the number of times the perform statement is to be executed is not predetermined. It depends on value of the variable I-CNT.
- If I-CNT is 0 or negative, this perform statement is not all executed

Perform – UNTIL Option:

- In the UNTIL phrase format, the procedure(s) referred to are performed until the condition specified by the UNTIL phrase is true. Control is then passed to the next executable statement following the PERFORM statement.
- Here the condition is tested before only at the beginning of each execution by default
- But this default is overridden by TEST AFTER phrase. If the TEST AFTER phrase is specified, the statements to be performed are executed at least once before the condition is tested (corresponds to DO UNTIL).
- In either case, if the condition is true, control is transferred to the next executable statement following the end of the PERFORM statement. If neither the TEST BEFORE nor the TEST AFTER phrase is specified, the TEST BEFORE phrase is assumed.

Flowchart:



PERFORM (UNTIL):

```
Example
MOVE 10 TO I-CNT
PERFORM UNTIL I-CNT = ZERO
       WRITE OUT-REC FROM I-ERR-REC (I-
CNT)
       SUBTRACT 1 FROM I-CNT
END-PERFORM
```

In the above example, the WRITE and SUBTRACT statements will be executed until the I-CNT becomes zero. Here perform will be executed 10 times.

Perform – Varying Option:

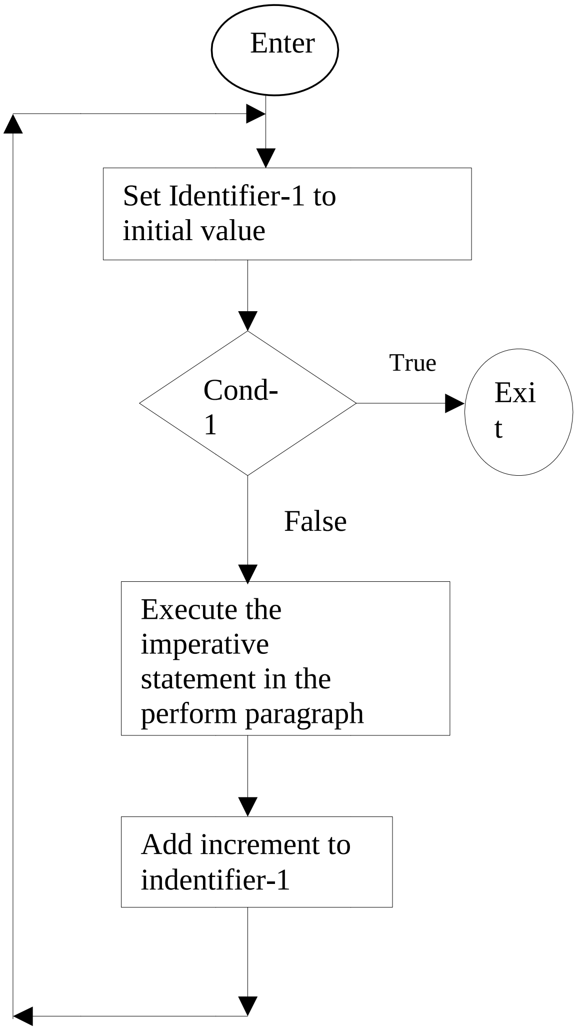
- After-phrase provides for varying more than one identifier
- If any of the operands specified in cond-1 or cond-2 is subscripted or reference modified, the subscript or reference-modifier is evaluated each time the condition is tested.
- When TEST BEFORE is indicated, all specified conditions are tested before the first execution, and the statements to be performed are executed, if at all, only when all specified tests fail. When TEST AFTER is indicated, the statements to be performed are executed at least once, before any condition is tested.

- Changing the values of identifiers and/or index-names in the VARYING, FROM, and BY phrases during execution changes the number of times the procedures are executed.

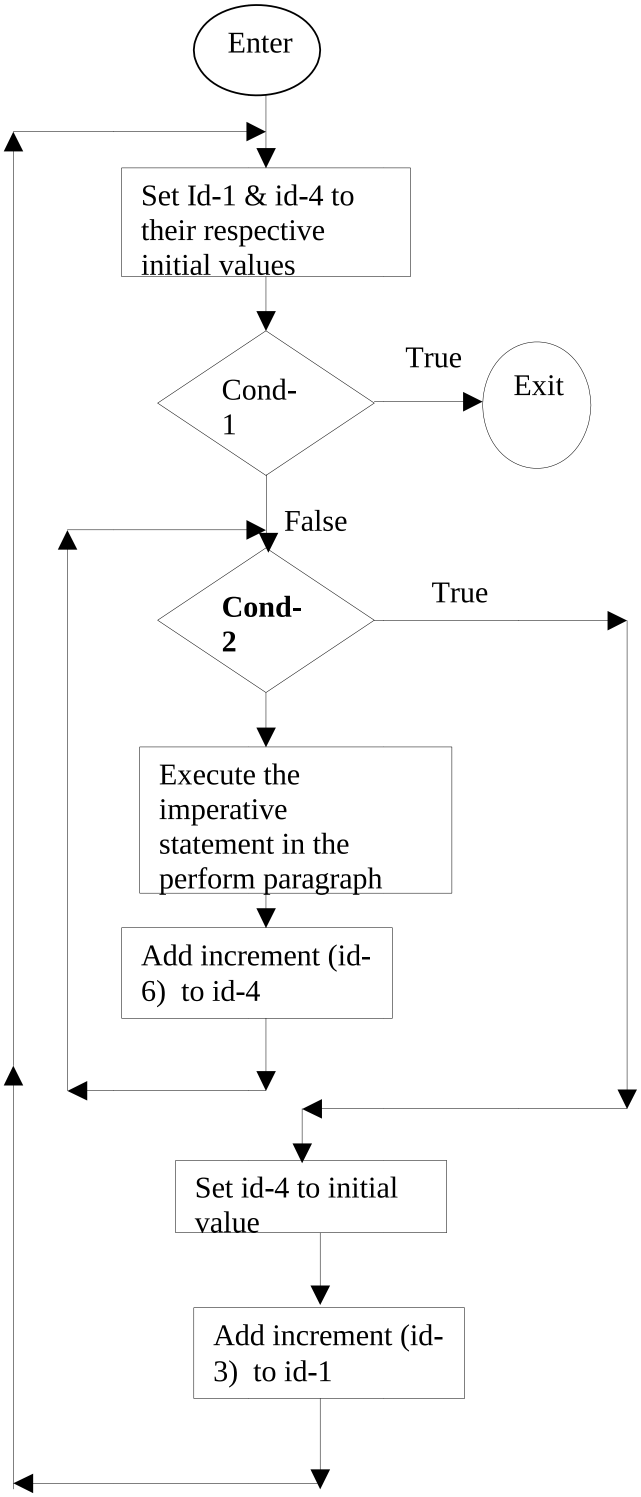
```
EXAMPLE
PERFORM PROC1 THRU PROC2
             VARYING I FROM 1 BY 1 UNTIL I > 50
             AFTER J FROM 1 BY 1 UNTIL J > 10
```

- In the above example the PROC-1 to PROC-2 will be executed 500 times.
- First with I as 1 and J varying from 1 to 10 in step of 1, then I as 2 and again J varying from 1 to 10 and so on. Every time I changes value, J must vary from 1 to 10.
- Each time the loop varying J is completed, J is initialized before changing the value of I.
- Thus after this perform statement is executed, value of I will 51 and that of J will 1 and not 11.

Flow Chart for Perform…varying… TEST BEFORE:


Enter
Set Identifier-1 to initial value
Cond-1
True
Exit
False
Execute the imperative statement in the perform paragraph
Add increment to indentifier-1

Flow Chart for Perform…varying….After….with TEST BEFORE:


Enter
Set Id-1 & id-4 to their respective initial values
Cond-1
True
Exit
False
Cond-2
True
Execute the imperative statement in the perform paragraph
Add increment (id-6) to id-4
Set id-4 to initial value
Add increment (id-3) to id-1

Try It Out

Problem Statement:

Try to display “HELLO WORLD” in the SYSOUT of the program from a sub-para called from the main para. Also the display should not contain any hard-coded string, instead use variables

Code:

```
 WORKING-STORAGE SECTION.
* INPUT VARIABLES
 01 W01-TEST-STRING                         PIC X(50).

PROCEDURE DIVISION.
*===============================================================*
* MAIN PARA - CALLS OTHERS PARA FOR PROCESSING
*===============================================================*
 0000-MAIN-PARA.
     PERFORM 1000-INITIALIZE-PARA
        THRU 1000-INITIALIZE-PARA-EXIT.
     STOP RUN.
 0000-EXIT.
     EXIT.

*===============================================================*
* INITIALIZE PARA - DISPLAYS ‘HELLO WORLD’
*===============================================================*
 1000-INITIALIZE-PARA.
*----MOVES & DISPLAYS HELLO WORLD
     MOVE 'HELLO WORLD' TO W01-TEST-STRING.
     DISPLAY W01-TEST-STRING.
 1000-INITIALIZE-PARA-EXIT.
     EXIT.
```

Refer File Name: Handout_Example_Code.txt under heading ‘Code for Session #6’ to obtain soft copy of the program code

How It Works:

- ❑ There are no Inputs (or) Outputs to this module
- ❑ The first statement calls the INITIALIZE Para
- ❑ In the INITIALIZE Para, the display statement is executed for displaying the PROGRAM SCOPE in the Job Dump (SYSOUT)
- ❑ The last statement “STOP RUN” indicates the end of execution.
- ❑ It is Good Practice to provide informative comments.

Summary

- Program Layout consists of:
  - Sequence Number Area (Position 01-06)
  - Comments / Continuation (Position 07)
  - Area A (Position 08-11)
  - Area B (Position 12-72)
  - Program Identification (Position 73-80)
- Identifiers in Variable Declaration are:
  - X for Any character
  - 9 for Numeric character
  - A for Alphabetic character
- Variable declaration examples: String & Numeric
- MOVE and PERFORM statements: Illustration with Code example and Result

Test Your Understanding

1. All Para names must start in _______.
2. What are the areas in disuse due to replacement of cards with disk drives?
3. What is Arithmetic Positioning Character for Implied decimal point?
4. What is the Data Representation Character for Numeric character?
5. What is the Numeric Editing Character for Zero Suppress?

Session 08: Arithmetic Expressions

Learning Objectives

After completing this session, you will be able to:

- ❑ Perform arithmetic operations in COBOL program
- ❑ Execute the arithmetic operations through statement involving clauses and expressions

Arithmetic Expressions

Arithmetic Expressions may consist of any of the following:

- ❑ An identifier described as a numeric elementary item.
- ❑ A numeric literal
- ❑ Identifiers and literals separated by arithmetic operators
- ❑ Two arithmetic expressions separated by an arithmetic operator.
- ❑ An arithmetic expression enclosed in parentheses

Arithmetic Operators

Following are the arithmetic operators:

- ❑ **BINARY:** + - * / **
- ❑ **UNARY:** + -
- ❑ Precedence
- ❑ Unary
- ❑ Exponentiation
- ❑ Multiplication and Division
- ❑ Addition and Subtraction

Arithmetic Statements

Arithmetic statements are of five kinds:

- ❑ `ADD`
- ❑ `SUBTRACT`
- ❑ `MULTIPLY`
- ❑ `DIVIDE`
- ❑ `COMPUTE`

Introduction to ADD Statement

Syntax - Format 1:

```
      identifier-
ADD { 1            } ... TO {identifier-2 [ROUNDED]} ...
      literal-1

[ON SIZE ERROR imperative-statement-1]
[NOT ON SIZE ERROR imperative-statement-2]
[END-ADD]
```

Syntax - Format 2:

```
      identifier-               identifier-
ADD { 1              } ...    { 2              } GIVING {identifier-3
      literal-1        TO       literal-2          [ROUNDED]}...

[ON SIZE ERROR imperative-statement-1]
[NOT ON SIZE ERROR imperative-statement-2]
[END-ADD]
```

Syntax - Format 3:

```
      CORRESPONDING
ADD {               } identifier-1 TO identifier-2 [ROUNDED]
      CORR

[ON SIZE ERROR imperative-statement-1]
[NOT ON SIZE ERROR imperative-statement-2]
[END-ADD]
```

Description: The ADD statement is used to add numeric items together. Each of the three formats works slightly differently and is described in the correspondingly numbered area.

- The first format is used to add field(s) or value(s) to another field. The field(s) or value(s) listed between ADD and TO are summed and added to the value of the field(s) following the TO, where the answer is stored.
    - For example, in ADD A TO B, the value in A is added to the value in B and the result is stored in B. The value in A is unchanged.
    - Furthermore, in ADD C D TO E F, the sum of the values in C and D is added to the value in E, storing the answer in E and the sum of C and D is also added to F storing the value in F. The values in C and D are unchanged.
- The second format is used to add field(s) or value(s) to one another, storing the answer in a different field. The field(s) or value(s) listed between ADD and GIVING are added and stored in the field following the field(s) following the GIVING. For example, in ADD A TO B GIVING C, the value in A is added to the value in B and the result is stored in C. The values in A and B are unchanged.
- The third format is used to add subordinate field(s) of one group to subordinate field(s) in another, storing the answer in those fields. Those subordinate field(s) of the group item identifier-1 are added to and stored in those with exactly the same name in the group item identifier-2. The names of the subordinate items must be spelled exactly the same way in both groups to qualify to participate in the addition.

Tips:

- The fields to be added must have numeric pictures, that is, they can only have the characters 9, S, and V in their PIC clauses.
- Receiving fields may be either numeric or numeric edited fields.
- Use the SIZE ERROR clause to detect field overflow on the receiving field.
- In all formats, the mathematically correct results are computed, but if the receiving field is too short in either the integer or decimal portion, the result will be truncated, the integer on the left and the decimal on the right. Including the ROUNDED phrase will result the answer field being rounded instead of truncating. Rounding is always done in the least significant portion of the answer.

ADD Corresponding:

For example, if two data hierarchies are defined as follows:

```
05  ITEM-1 OCCURS 6.
    10  ITEM-A PIC S9(3).
    10  ITEM-B PIC +99.9.
    10  ITEM-C PIC X(4).
    10  ITEM-D REDEFINES ITEM-C PIC 9(4).
    10  ITEM-E USAGE COMP-1.
    10 ITEM-F USAGE INDEX.
  05  ITEM-2.
    10  ITEM-A PIC 99.
    10  ITEM-B PIC +9V9.
    10  ITEM-C PIC A(4).
    10  ITEM-D PIC 9(4).
    10  ITEM-E PIC 9(9) USAGE COMP.
    10  ITEM-F USAGE INDEX.
```

Then, if ADD CORR ITEM-2 TO ITEM-1(X) is specified, ITEM-A and ITEM-A(X), ITEM-B and ITEM-B(X), and ITEM-E and ITEM-E(X) are considered to be corresponding and are added together. ITEM-C and ITEM-C(X) are not included because they are not numeric. ITEM-D and ITEM-D(X) are not included because ITEM-D(X) includes a REDEFINES clause in its data description. ITEM-F and ITEM-F(X) are not included because they are defined as USAGE IS INDEX. Note that ITEM-1 is valid as either identifier-1 or identifier-2.

If any of the individual operations in the ADD CORRESPONDING statement produces a size error condition, imperative-statement-1 in the ON SIZE ERROR phrase is not executed until all of the individual additions are completed.

ON SIZE Error Occurs:

- When the absolute value of the result of an arithmetic evaluation, after decimal point alignment, exceeds the largest value that can be contained in the result field
- When division by zero occurs
- If the ON SIZE ERROR phrase is specified and a size error condition occurs, the value of the resultant identifier affected by the size error is not altered--that is, the error results are not placed in the receiving identifier. After completion of the execution of

the arithmetic operation, the imperative statement in the ON SIZE ERROR phrase is executed, control is transferred to the end of the arithmetic statement, and the NOT ON SIZE ERROR phrase, if specified, is ignored.

- If the NOT ON SIZE ERROR phrase has been specified and, after execution of an arithmetic operation, a size error condition does not exist, the NOT ON SIZE ERROR phrase is executed.
- For ADD CORRESPONDING and SUBTRACT CORRESPONDING statements, if an individual arithmetic operation causes a size error condition, the ON SIZE ERROR imperative statement is not executed until all the individual additions or subtractions have been completed.
- If the ROUNDED phrase is specified, rounding takes place before size error checking.

ROUNDED Phrase:

- After decimal point alignment, the number of places in the fraction of the result of an arithmetic operation is compared with the number of places provided for the fraction of the resultant identifier.
- When the size of the fractional result exceeds the number of places provided for its storage, truncation occurs unless ROUNDED is specified. When ROUNDED is specified, the least significant digit of the resultant identifier is increased by 1 whenever the most significant digit of the excess is greater than or equal to 5.

ADD – Examples:

```
ADD A TO B
ADD 1 TO B
ADD A TO B ROUNDED
ADD A TO B, C
ADD A TO B ROUNDED
             C ROUNDED
ADD A TO B ROUNDED, C
ON SIZE ERROR
MOVE 1 TO ERR-FLAG
ADD A, B, C, D TO X, Y, Z
ADD A , B,  C,  D TO E GIVING X, Y
```

Add CORR X to Y (Example):

```
01 X.
     05 X-1 PIC 9(3) VALUE 1.
     05 X-2 PIC 9(3) VALUE 2.
     05 X-3 PIC 9(3) VALUE 3.
01 Y.
      05 X-1 PIC 9(5) VALUE 5.
      05 X-2 PIC 9(5) VALUE 9.
      05 X-3 PIC 9(5) VALUE 12.
After Add, Under group Y:
 X-1 = 6; X-2 = 11; X-3 =  15
```

Introduction to SUBTRACT Statement

Syntax - Format 1:

```
           identifier-1
SUBTRACT {              } ... FROM {identifier-2 [ROUNDED]} ...
           literal-1

[ON SIZE ERROR imperative-statement-1]
[NOT ON SIZE ERROR imperative-statement-2]
[END-SUBTRACT]
```

Syntax - Format 2:

```
           identifier-              identifier-
SUBTRACT { 1          } ...       { 2          } GIVING identifier-3
           literal-1    FROM        literal-2      [ROUNDED]

[ON SIZE ERROR imperative-statement-1]
[NOT ON SIZE ERROR imperative-statement-2]
[END-SUBTRACT]
```

Syntax - Format 3:

```
           CORRESPONDING
SUBTRACT {               } identifier-1 FROM identifier-2 [ROUNDED]
           CORR

ON SIZE ERROR imperative-statement-1]
[NOT ON SIZE ERROR imperative-statement-2]
[END-SUBTRACT]
```

Description: The SUBTRACT statement is used to subtract numeric items. Each of the three formats works slightly differently and is described in the correspondingly numbered area.

- The first format is used to subtract field(s) or value(s) from another field. The field(s) or value(s) listed between SUBTRACT and FROM are summed and subtracted from the value of the field(s) following the FROM. The answer is stored in the individual field. For example in SUBTRACT A FROM B, the value in A is subtracted from the value in B and the result is stored in B. The value in A is unchanged. Furthermore, in SUBTRACT C D FROM E F, the sum of the values in C and D is subtracted to the value in E, storing the answer in E and the sum of C and D is also subtracted from F, storing the value in F. The values in C and D are unchanged.
- The second format is used to subtract field(s) or value(s) from one another, storing the answer in a different field. The field(s) or value(s) listed between SUBTRACT and GIVING are added and stored in the field following the field(s) following the GIVING. For example, in SUBTRACT A FROM B GIVING C, the value in A is subtracted to the value in B and the result is stored in C. The values in A and B are unchanged.
- The third format is used to subtract subordinate field(s) of one group from subordinate field(s) in another, storing the answer in those fields. Those subordinate field(s) of the group item identifier-1 are subtracted from and stored in those with exactly the same name in the group item identifier-2. The names of the subordinate items must be spelled exactly the same way in both groups to qualify to participate in the subtraction.

Tips:

- The fields to be subtracted must have numeric pictures, that is, they can only have the characters 9, S, and V in their PIC clauses.
- Receiving fields may be either numeric or numeric edited fields.
- Use the SIZE ERROR clause to detect field overflow on the receiving field.
- In all formats, the mathematically correct results are computed, but if the receiving field is too short in either the integer or decimal portion, the result will be truncated, the integer on the left and the decimal on the right. Including the ROUNDED phrase will result the answer field being rounded instead of truncating. Rounding is always done in the least significant portion of the answer.

SUBTRACT – Examples:

```
SUBTRACT 1 FROM A
SUBTRACT A FROM B
SUBTRACT A B FROM C D
SUBTRACT A B FROM C GIVING D
SUBTRACT A FROM B
 GIVING CROUNDED
SUBTRACT A FROM B
      ON SIZE ERROR
      MOVE 1 TO ERROR-FLAG
```

Introduction to MULTIPLY Statement

Syntax - Format 1:

```
MULTIPLY { identifier-1 } ... BY {identifier-2 [ROUNDED]} ...
         { literal-1    }
[ON SIZE ERROR imperative-statement-1]
[NOT ON SIZE ERROR imperative-statement-2]
[END-MULTIPLY]
```

Syntax - Format 2:

```
MULTIPLY { identifier-1 } ... BY { identifier-2 } GIVING {identifier-3
         { literal-1    }        { literal-2    } [ROUNDED]}...
[ON SIZE ERROR imperative-statement-1]
[NOT ON SIZE ERROR imperative-statement-2]
[END-MULTIPLY]
```

Description: The MULTIPLY statement is used to multiply numeric items together. Both formats work slightly differently and each is described in the correspondingly numbered area.

- The first format is used to multiply a field or value by another field or value. The product of the field(s) or value(s) listed between MULTIPLY and BY is computed and multiplied by the value of the field(s) following the BY. The answer is stored in the individual field. For example, in MULTIPLY A BY B, the value in A is multiplied by the value in B and the result is stored in B. The value in A is unchanged. Furthermore, in

MULTIPLY C BY D E, the value in C is multiplied by the value in D, storing the answer in D and the value of C is also multiplied by E, storing the value in F. The value in C is unchanged.

- The second format is used by multiply field(s) or value(s) by one another field or value, storing the answer in a different field. The field(s) or value(s) listed between MULTIPLY and GIVING are multiplied and stored in the field(s) following the GIVING. For example, in MULTIPLY A BY B GIVING C, the value in A is multiplied by the value in B and the result is stored in C. The values in A and B are unchanged.

Tips:

- The fields to be multiplied must have numeric pictures, that is, they can only have the characters 9, S, and V in their PIC clauses.
- Receiving fields may be either numeric or numeric edited fields.
- Use the SIZE ERROR clause to detect field overflow on the receiving field.
- In both formats, the mathematically correct results are computed, but if the receiving field is too short in either the integer or decimal portion, the result will be truncated, the integer on the left and the decimal on the right. Including the ROUNDED phrase will result the answer field being rounded instead of truncating. Rounding is always done in the least significant portion of the answer

MULTIPLY – Examples:

```
MULTIPLY 5 BY A
MULTIPLY 5 BY A ROUNDED
MULTIPLY A BY B
MULTIPLY A BY B GIVING C D
MULTIPLY A BY B ON SIZE ERROR
 MOVE 1 TO ERROR-FLAG
MULTIPLY A BY B GIVING C
      ON SIZE ERROR
      MOVE 1 TO ERROR-FLAG
```

Introduction to DIVIDE Statement

Syntax - Format 1:

```
DIVIDE { identifier-1 } ... INTO {identifier-2 [ROUNDED]} ...
       { literal-1    }
[ON SIZE ERROR imperative-statement-1]
[NOT ON SIZE ERROR imperative-statement-2]
[END-DIVIDE]
```

Syntax - Format 2:

```
       { identifier- }      { identifier- }
DIVIDE { 1           } INTO { 2           } GIVING {identifier-3
       { literal-1   }      { literal-2   } [ROUNDED]}...
[ON SIZE ERROR imperative-statement-1]
[NOT ON SIZE ERROR imperative-statement-2]
[END-DIVIDE]
```

Syntax - Format 3:

```
       { identifier- }    { identifier- }
DIVIDE { 1           } BY { 2           } GIVING {identifier-3
       { literal-1   }    { literal-2   } [ROUNDED]}...
[ON SIZE ERROR imperative-statement-1]
[NOT ON SIZE ERROR imperative-statement-2]
[END-DIVIDE]
```

Syntax - Format 4:

```
       { identifier- }      { identifier- } GIVING identifier-3
DIVIDE { 1           } INTO { 2           } [ROUNDED] REMAINDER
       { literal-1   }      { literal-2   } identifier-4
[ON SIZE ERROR imperative-statement-1]
[NOT ON SIZE ERROR imperative-statement-2]
[END-DIVIDE]
```

Syntax - Format 5:

```
       { identifier- }    { identifier- }
DIVIDE { 1           } BY { 2           } GIVING identifier-3 [ROUNDED]
       { literal-1   }    { literal-2   } REMAINDER identifier-4
[ON SIZE ERROR imperative-statement-1]
[NOT ON SIZE ERROR imperative-statement-2]
[END-DIVIDE]
```

Description: The DIVIDE statement is used to divide numeric items together. Both formats work slightly differently and each is described in the correspondingly numbered area.

- The first format is used to divide a field(s) or value(s) by another field or value. The field or value listed between DIVIDE and INTO is divided into the value of the field(s) following the INTO. The answer is stored if the individual field. For example, in DIVIDE A INTO B, the value in A is divided into the value in B and the result is stored in B. The value in A is unchanged. Furthermore, in DIVIDE C INTO D E, the value in C is divided into the value in D, storing the answer in D; the value of C is also divided into E, storing the value in E. The value in C is unchanged.
- The second format is used to divide field(s) or value(s) by one another, storing the answer in a different field. The field(s) or value(s) listed between DIVIDE and GIVING are divided and stored in the field following the field(s) following the GIVING. For example, in DIVIDE A INTO B GIVING C, the value in A is divided into the value in B and the result is stored in C. The values in A and B are unchanged.
- The third format is also used to divide field(s) or value(s) by one another, storing the answer in a different field. The field(s) or value(s) listed between DIVIDE and GIVING are divided and stored in the field following the field(s) following the GIVING. For example, in DIVIDE A BY B GIVING C, the value in A is divided by the value in B and the result is stored in C. The values in A and B are unchanged.
- The fourth format is used to divide field(s) or value(s) by one another, storing the answer in a different field. The field(s) or value(s) listed between DIVIDE and GIVING are divided and stored in the field following the field(s) following the GIVING. For example, in DIVIDE A INTO B GIVING C REMAINDER D, the value in A is divided into the value in B and the results stored in C with the remainder being stored in D. The values in A and B are unchanged.
- The fifth format is also used by divide field(s) or value(s) by one another, storing the answer in a different field. The field(s) or value(s) listed between DIVIDE and GIVING are divided and stored in the field following the field(s) following the GIVING. For example, in DIVIDE A BY B GIVING C REMAINDER D, the value in A is divided by the value in B and the result is stored in C with the remainder being stored in D. The values in A and B are unchanged.

Tips:

- The fields to be divided must have numeric pictures, that is, they can only have the characters 9, S, and V in their PIC clauses.
- Receiving fields may be either numeric or numeric edited fields.
- Use the SIZE ERROR clause to detect field overflow on the receiving field.
- In all formats, the mathematically correct results are computed, but if the receiving field is too short in either the integer or decimal portion, the result will be truncated, the integer on the left and the decimal on the right. Including the ROUNDED phrase will result the answer field being rounded instead of truncating. Rounding is always done in the least significant portion of the answer

DIVIDE – Examples:

	BEFORE			**AFTER**		
	A	B	C	A	B	C
`DIVIDE A INTO B`	2	10	-	2	5	-
`DIVIDE A INTO B ROUNDED`	2	9	-	2	5	-
`DIVIDE A INTO B C`	2	10	14	2	5	7

DIVIDE A INTO B GIVING C	2	10	-	2	10	5
DIVIDE 2 INTO A GIVING B C	2	-	-	2	1	1
DIVIDE A BY B GIVING C	6	2	-	6	2	3
DIVIDE A BY 5 GIVING B REMAINDER C	8	-	-	8	1	3

Introduction to COMPUTE Statement

Syntax - Format 1:

```
                                              =
COMPUTE {identifier-2 [ROUNDED]} ...  {           }  arithmetic-expression
                                            EQUAL

[ON SIZE ERROR imperative-statement-1]
[NOT ON SIZE ERROR imperative-statement-2]
[END-COMPUTE]
```

Description: The COMPUTE statement is used to evaluate arithmetic formulas. The formula is evaluated and the result is stored in the identifier(s) between COMPUTE and =.

- ❑ The format for the formula is a standard algebraic expression with addition, subtraction, multiplication, and division represented as +, -, *, and /. The exponentiation operator is **.
- ❑ When evaluating the formula, standard rules of mathematical precedence apply, i.e. exponentiation first, then multiplication and division at equal precedence, then addition and subtraction at equal precedence. If some other order is preferred, operations may be enclosed in parentheses to give them precedence.

Tips:

- ❑ If only one mathematical operation is to be performed, use the statement that performs that operation (ADD, SUBTRACT, MULTIPLY, or DIVIDE). Reserve the COMPUTE statement for formulas or exponentiation, since it has no separate statement associated with it.
- ❑ Receiving fields may be either numeric or numeric edited fields.
- ❑ Use the SIZE ERROR clause to detect field overflow on the receiving field.
- ❑ In all formats, the mathematically correct results are computed, but if the receiving field is too short in either the integer or decimal portion, the result will be truncated, the integer on the left and the decimal on the right. Including the ROUNDED phrase will result the answer field being rounded instead of truncating. Rounding is always done in the least significant portion of the answer.

COMPUTE – Examples:

```
COMPUTE PROFIT = PRICE - COST,
      COMPUTE A            = B + C * 2.
```

Introduction to INITIALIZE Statement

Syntax - Format 1:

```
INITIALIZE {identifier-1}...

                  ALPHABETIC
                  ALPHANUMERIC
[REPLACING {   {  NUMERIC              }  DATA BY { identifier-2 } }...]
                  ALPHANUMERIC-EDITED               literal-1
                  NUMERIC-EDITED
```

Description: The INITIALIZE statement is used to initialize data items. Data items are initialized according to their type:

- ❑ Alphabetic and alphanumeric items are initialized to spaces.
- ❑ Numeric items are initialized to zeros.
- ❑ For group items, the subordinate elementary items are initialized depending on their individual data types. Only named items are initialized.
- ❑ The REPLACING phrase can be used to initialize data items to different values other than the default SPACES or ZEROS.

Tips:

- ❑ Use the initialize statement instead of moving ZEROS or SPACES to a group. That way, each subordinate item will be set to an appropriate value.
- ❑ Notice that unnamed and FILLER items are not initialized. This allows the variable portion of a detail line, for example, to be initialized without the constant values inserted into the unnamed fields being affected.

INITIALIZE - Example

```
01 A.
02    A1    PIC   9(5).
            02    A2    PIC   X(4).
02    A3    PIC   9(3).
02    A4    PIC   Z(3)9.99.

INITIALIZE A REPLACING NUMERIC DATA BY 50
In the above example on A1 and A3 will be initialized to 50.

INITIALIZE A.
In the above statement A1 and A3 will be initialized to zeroes and A2
and A4 will be    initialized to spaces.
```

INITIALIZE Rules:

- Category of Id-2 should be compatible that of CORR **REPLACING**
- Whether identifier-1 references an elementary or group item, all operations are performed as if a series of MOVE statements had been written, each of which had an elementary item as a receiving field.
- Same Category can't be repeated
- If the REPLACING phrase is specified:
  - If identifier-1 references a group item, any elementary item within the data item referenced by identifier-1 is initialized only if it belongs to the category specified in the REPLACING phrase.
  - If identifier-1 references an elementary item, that item is initialized only if it belongs to the category specified in the REPLACING phrase.
- All such elementary receiving fields, including all occurrences of table items within the group, are affected, with the following exceptions:
  - Index data items
  - Data items defined with USAGE IS POINTER
  - Elementary FILLER data items
  - Items that are subordinate to identifier-1 and contain a REDEFINES clause or any items subordinate to such an item. (However, identifier-1 can contain a REDEFINES clause or be subordinate to a redefining item.)
- The areas referenced by identifier-1 are initialized in the order (left to right) of the appearance of identifier-1 in the statement. Within a group receiving field, affected elementary items are initialized in the order of their definition within the group.
- SPACE is implied source for Alpha, Alpha-numeric and Alpha-numeric edited
- ZERO is implied source for Numeric and Numeric edited
- For Group initialize the following will not participate:
  - Index variables
  - Pointer variables
  - Filler
  - Redefines
  - Elementary items that are not of category mentioned in REPLACING

Introduction to ACCEPT Statement

Syntax - Format 1:

```
                          { mnemonic-name-1  }
ACCEPT identifier-1 [FROM {                  } ]
                          { environment-name }
```

Syntax - Format 2:

```
                         { DATE        }
ACCEPT identifier-1 FROM { DAY         }
                         { DAY-OF-WEEK }
                         { TIME        }
```

Description: The ACCEPT statement is used to acquire data from the primary input device or various system fields. Each of the two formats works slightly differently and is described in the correspondingly numbered area.

- ❑ The first format is used to accept data from the primary input or other specified device. The field(s) listed after ACCEPT are loaded
- ❑ The second format is used to accept one of various system value(s). Each of the items to be accept has a different format. They are:
  - o **DATE (YYMMDD):** Two-digit year of century followed by two-digit month of year followed by two-digit day of month. (Gregorian Date)
  - o **DAY (YYDDD):** Two-digit year of century followed by three-digit day of year. (Julian Date)
  - o **DAY-OF-WEEK (D):** One-digit day of week, where Monday is 1, Tuesday is 2, etc.
  - o **TIME (HHMMSSHH):** Two-digit hour of day followed by two digit minute of an hour followed by two-digit second of minute followed by two digit hundredths of a second. (Twenty-four hour clock)

Tips:

- ❑ Use the ACCEPT statement to acquire small amounts of data such as parameters to be entered at runtime.
- ❑ The chief advantage of acquiring data with the ACCEPT statement is that no file needs to be established to use it.
- ❑ The chief disadvantage of acquiring data with the ACCEPT statement is that there is no mechanism to recognize end of file.
- ❑ Many COBOLs have added enhancements to allow access to a four-digit year due to the Y2K problem. Check your system manual for details.

ACCEPT – Example

```
COBOL statement
ACCEPT in-parm
ACCEPT in-parm FROM card

JCL statement:
//SYSIN DD applicable parameters
LRECL of DCB parameter can be up to a
max of 256.

RECFM can be Fixed or Variable
Records are read until identifier is
filled or EOF reached
```

Try It Out

Problem Statement:

Declare two numeric variables and assign values from JCL to them using ACCEPT Statement. Perform all the arithmetic operations on the variables

Code:

```
*================================================================*
* PROCESS PARA - PROCESS INPUTS
*================================================================*
 2000-PROCESS-INPUT.
     ADD      W01-INPUT-01 TO   W01-INPUT-02 GIVING W02-ADD.
     SUBTRACT W01-INPUT-01 FROM W01-INPUT-02 GIVING W02-SUB.
     MULTIPLY W01-INPUT-01 BY   W01-INPUT-02 GIVING W02-MUL.
     DIVIDE   W01-INPUT-01 BY   W01-INPUT-02 GIVING W02-DIV.
     DISPLAY '--ARITHMATIC OPERATIONS DEMO RESULTS--'.
     DISPLAY W01-INPUT-01 ' + ' W01-INPUT-02 ' = ' W02-ADD.
     DISPLAY W01-INPUT-01 ' - ' W01-INPUT-02 ' = ' W02-SUB.
     DISPLAY W01-INPUT-01 ' * ' W01-INPUT-02 ' = ' W02-MUL.
     DISPLAY W01-INPUT-01 ' / ' W01-INPUT-02 ' = ' W02-DIV.

 2000-PROCESS-INPUT-EXIT.
     EXIT.

*================================================================*
* PROCESS PARA - PROCESS INPUTS
*================================================================*
 3000-COMPUTE-INPUT.
     INITIALIZE W02-ADD, W02-SUB, W02-MUL, W02-DIV.
     DISPLAY '------DATA AFTER INITIALIZATION--------'.
     DISPLAY 'W02-ADD = ' W02-ADD.
     DISPLAY 'W02-SUB = ' W02-SUB.
     DISPLAY 'W02-MUL = ' W02-MUL.
     DISPLAY 'W02-DIV = ' W02-DIV.
     COMPUTE W02-ADD = W01-INPUT-01 + W01-INPUT-02.
     COMPUTE W02-SUB = W01-INPUT-01 - W01-INPUT-02.
     COMPUTE W02-MUL = W01-INPUT-01 * W01-INPUT-02.
     COMPUTE W02-DIV = W01-INPUT-01 / W01-INPUT-02.
     DISPLAY '--ARITHMATIC OPERATIONS DEMO RESULTS--'.
     DISPLAY '------------USING COMPUTE-------------'.
     DISPLAY W01-INPUT-01 ' + ' W01-INPUT-02 ' = ' W02-ADD.
     DISPLAY W01-INPUT-01 ' - ' W01-INPUT-02 ' = ' W02-SUB.
     DISPLAY W01-INPUT-01 ' * ' W01-INPUT-02 ' = ' W02-MUL.
     DISPLAY W01-INPUT-01 ' / ' W01-INPUT-02 ' = ' W02-DIV.

 3000-COMPUTE-INPUT-EXIT.
     EXIT.
```

Refer File Name: Handout_Example_Code.txt under heading 'Code for Session #8' to obtain soft copy of the program code

How It Works:

- The job accepts the inputs from the JCL from the SYSIN using the ACCEPT statement.
- The output of the calculations is displayed in the JCL SYSOUT using the DISPLAY statements.
- The initial para performs the operations using ADD, SUBTRACT, MULTIPLY, DIVIDE.
- The subsequent para performs the same logic using COMPUTE after initializing the display variables
- The last statement “STOP RUN” indicates the end of execution.

Summary

- The basic arithmetic operation statements are `ADD`, `SUBTRACT`, `MULTIPLY`, and `DIVIDE`.
- The `COMPUTE` statement is used to handle complicated calculations.
- The `INITIALIZE` statement is used to initialize variables.

Test Your Understanding

1. The statement used to perform arithmetic operation in expression format is ______.
2. What is the significance of GIVING in the COBOL statements involving arithmetic operations?
3. The _______ statement would reset the values of string to spaces and numbers to zeroes.
4. The destination variable is mandatory in COMPUTE statement but not in other clauses discussed. State true or false

Session 09: String Handling

Learning Objectives

After completing this session, you will be able to:

- Execute the various string operations in a COBOL program
- Perform string functions to extract a part of a long string

Introduction to STRING Statement

Syntax - Format 1:

```
STRING {{ identifier-1 } ... DELIMITED BY { identifier-2 }} ...
       {{ literal-1    }                  { literal-2    }}
INTO identifier-3
[WITH POINTER identifier-4]
[ON OVERFLOW imperative-statement-1]
[NOT ON OVERFLOW imperative-statement-2]
[END-STRING]
```

Description:

- The STRING statement is used to concatenate non-numeric items together. Any number of items can be concatenated. Entire or partial strings may be concatenated. To use the entire string, delimit it by size. To use only a portion of a string, delimit it by whatever character indicates the end of the data you want to concatenate.
- The POINTER clause can be used to start in a position other than 1. For example, if the POINTER variable has a value of 3, character 1 and 2 of the receiving item will remain unchanged, and the stringing operation will proceed from the third character of the receiving field. As each character is inserted into the receiving field, the value of the POINTER variable will be incremented, so that the POINTER variable may be used to STRING in more values where the previous STRING operation left off.
- Only as much of the value in the receiving field will be changed as is necessary to hold the newly assembled string, for example, in the statement

  ```
  STRING A B DELIMITED BY SIZE INTO C.
  ```

- If A is a one character string with a value of A and B is a one character string with a value of B and C is a five character string with an original value of CCCCC, after the STRING statement is executed, C will have a value of ABCCC.

STRING – Example:

```
05    FIELD-1     PIC   X(04) VALUE ‘ABCD’.
05    FIELD-2     PIC   X(06) VALUE ‘MA IN ‘.
05    FIELD-3     PIC   X(09) VALUE ‘121,34,56’.
05    FIELD-4     PIC   X(14) VALUE SPACES.
```

```
STRING FIELD-1 FIELD-2 FIELD-3 DELIMITED BY ‘ ‘, ‘,’ INTO FIELD-4
FIELD-4 = ABCDMA121bbbbb
```

- In the above example, FIELD-1 does not contain any delimiters, all 4 characters are moved to FIELD-4
- First 2 characters of FIELD-2 are moved to FIELD-4 as third character is the delimiter
- First 3 characters of FIELD-3 are moved to FIELD-4 as 4th character is the delimiter

STRING Rules:

- Receiving fields must not be:
  - Edited field
  - With Justified clause
  - With reference modification
- Pointer variable must be:
  - Elementary and Numeric
  - Large enough to hold max length of receiving field
- ON OVERFLOW executed when pointer Value <= 0 or exceeds max length of receiving field

Introduction to UNSTRING Statement

Syntax - Format 1:

```
                                                 }
UNSTRIN  identifie  [ DELIMITE  { identifie      OR    { identifie  }   ..  ]
G        r-1          D BY        r-2            [ALL    r-3        ]   .
                      [ALL]       literal-1      ]       literal-2
                                                 [
```

```
INTO {identifier-4 [DELIMITER IN identifier-5] [COUNT IN identifier-
6]}...
[WITH POINTER identifier-7]
[TALLYING IN identifier-8]
[ON OVERFLOW imperative-statement-1]
[NOT ON OVERFLOW imperative-statement-2]
[END-UNSTRING]
```

Description:

- The UNSTRING statement is used to parse individual items from within a single string. Any number of items may be parsed. Entire or partial strings may be parsed. As many items as are provided as INTO operands will be parsed.
- The POINTER clause can be used to start in a position other than 1. For example, if the POINTER variable has a value of 3, character 1 and 2 of the string will not be included, and the unstringing operation will proceed from the third character. As each character is inserted into a receiving field, the value of the POINTER variable will be incremented, so that the POINTER variable may be used to UNSTRING more values from where the previous UNSTRING operation left off.

- The COUNT phrase can be used to determine the length of any parsed string.
- Only as much of the values in the receiving field will be changed as is necessary to hold the newly parsed string, for example, in the statement

```
UNSTRING A DELIMITED BY SPACES INTO B C.
```

- If A is a five character unstring with a value of "A A A" and B is a two character string with an original value of BB and C is a two character string with an original value of CC, after the UNSTRING statement is executed, B will have a value of AB and C will have a value of AC.

Tips:

- The operands should be non-numeric.
- The POINTER and COUNT operands, if any, must be positive integers that are their pictures should contain only 9's.
- INITIALIZE the receiving items before the UNSTRING, to remove unwanted characters that may be left from a prior operation.
- Use the OVERFLOW clause to detect field overflow on the receiving field(s).

To extract a part of the string

- A part of the string could be extracted provided the starting position and the length are known, for example MOVE W01-TEXT-01(4:5) TO W01-TEXT-02
- In the preceding example if the string W01-TEXT-01 has value “MISMATCH”, then the value of W01-TEXT-02 would be “MATCH”.
- This concept could be used to extract PREFIX and SUFFIX of a given string

Introduction to INSPECT Statement

Syntax - Format 1:

```
INSPECT identifier-1 TALLYING identifier-2 FOR

                                                 identifie
   CHARACTERS  [{  BEFORE    }  INITIA    {  r-3          }]  ...
                   AFTER        L            literal-1
                                                                                        ..
{                                                                                    }  .
                   identifi                              identif
   {  ALL      }{{ er-4       }[{ BEFORE  }  INITIAL  {  ier-5     }]  ..  }  ..
      LEADING      literal-      AFTER                   literal       .       .
                   2                                     -3
```

Syntax - Format 2:

```
INSPECT identifier-6 REPLACING

                                                       identif
  CHARACT      identifier                              ier-8
  ERS BY   {   -7          }[{   BEFORE  }  INITIA  {  literal  }] ...
               literal-4         AFTER      L          -5
                                                                            .
{                                                                        }  .
     ALL                                                      ident         .
     LEAD         identifi                                    ifier
  {  ING   }  {   er-9        }  BY {  identifie  }  [{ BEFORE } INITI  {  -11   } ..  } ..
     FIRS         literal-                 r-10          AFTER    AL        liter   ] .     .
     T            6                        literal-7                        al-8
```

Syntax - Format 3:

```
INSPECT identifier-12 TALLYING identifier-13 FOR

                                             identifier   }
  CHARACTER  [{  BEFORE   }  INITIA    {     -14          }  ...
  S              AFTER       L               literal-9    ]
                                                                             ...
{                                                                        }   REPLACIN
     ALL      }{  identifier   }[  BEFORE  }  INITIAL  {  identifier  }  ..  }  G
  {  LEADIN   {   -15          {   AFTER                  -16         ]  .      ..
     G            literal-10                              literal-11            .

                   identifier-                             identifier
  CHARACTERS   {   17          }[{  BEFORE  }  INITIA   {  -18         }  ...
  BY               literal-12       AFTER      L           literal-13  ]
                                                                                    ..
{                                                                                }  .
     ALL
     LEADIN  } {  identifier   }  B  {  identifier  }  [{  BEFOR  }  INITIAL  {  identifier  }  ..  }  ..
  {  G            -19             Y     -20               E                     -21         ]  .      .
     FIRST        literal-14            literal-15        AFTER                 literal-16
```

Syntax - Format 4:

```
INSPECT identifier-22 CONVERTING

                                                   [   BEFOR                                     }
  {  identifier  }  T  {  identifier  }            E      }  INITIA  {  identifier          ..
     -3             O     -23                              L            -24                 .
     literal-1            literal-17               {   AFTER                literal-18   ]
```

Description: The INSPECT statement is used to perform various operations on string data. Each of the four formats works slightly differently and is described in the correspondingly numbered area.

Cognizant
Passion for making a difference

- The first format is used to count occurrences of characters or strings within another string. The number of occurrences is stored in identifier-2. The value in identifier-1 is unchanged.
- The second format is used to replace occurrences of characters or strings within another string. The replacement strings must be of the same length as the original string.
- The third format is used to count while replacing occurrences of characters or strings within another string. It is basically a combination of the first two formats.
- The fourth format is used to replace occurrences of characters within another string using only one INSPECT. Any occurrence on the first character of the CONVERTING in the inspected string will be replaced by the first character in the TO string, while occurrences of the second character in the CONVERTING string will be replaced by the second character in the TO string, and so on. The CONVERTING string must be of the same length as the tostring.

Tips:

- Format 2 can be used to convert leading spaces to zeros, in a numeric item, e.g.

```
INSPECT LEADING-SPACE-ITEM REPLACING LEADING SPACES BY ZEROS.
```

- Format 4 can be used to convert lowercase to uppercase, e.g.

INSPECT MIXED-CASE-ITEM CONVERTING "abcde..." TO "ABCDE...".

INSPECT – Example:

```
77  countr    PIC   9  VALUE ZERO.
01  data-1    PIC X(6).
INSPECT data-1 TALLYING countr FOR CHARACTERS AFTER INITIAL "S"
                             REPLACING  ALL "A" BY "O"
DATA-1      COUNTR       DATA-1
Before           After   After

ANSELM            3     ONSELM
SACKET            5     SOCKET
PASSED            3     POSSED

INSPECT data-1  REPLACING CHARACTERS BY ZEROS BEFORE INITIAL
QUOTE.

DATA-1      COUNTR      DATA-1
Before      After       After

ANS”LM            0     000”LM
SACK”T            0     0000”T
```

```
INSPECT data-1  CONVERTING
”abcdefghijklmnopqrstuvwxyz" TO
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
 AFTER INITIAL "/"  BEFORE INITIAL"?"

 Before                  After

 a/five/?six                      a/FIVE/?six
 r/Rexx/RRRr             r/REXX/RRRR
 zfour?inspe             zfour?inspe
```

Intrinsic Functions

Date Functions: Probably the most useful intrinsic function is CURRENT-DATE which is a replacement for ACCEPT DATE and ACCEPT TIME. CURRENT-DATE is Y2K-compliant, having a 4-digit year. This function returns a 20-character alphanumeric field which is laid out as follows:

```
01  WS-CURRENT-DATE-FIELDS.
    05  WS-CURRENT-DATE.
        10  WS-CURRENT-YEAR    PIC  9(4).
        10  WS-CURRENT-MONTH   PIC  9(2).
        10  WS-CURRENT-DAY     PIC  9(2).
    05  WS-CURRENT-TIME.
        10  WS-CURRENT-HOUR    PIC  9(2).
        10  WS-CURRENT-MINUTE  PIC  9(2).
        10  WS-CURRENT-SECOND  PIC  9(2).
        10  WS-CURRENT-MS      PIC  9(2).
    05  WS-DIFF-FROM-GMT       PIC S9(4).
```

So not only can you get the time down to the millisecond, but you can get the difference between your time and Greenwich Mean Time.

The function is used in a MOVE:

```
MOVE FUNCTION CURRENT-DATE TO WS-CURRENT-DATE-FIELDS
```

But normally reference modification is used to only grab the part you want:

- * Get the current date in yyyymmdd format:

  ```
  MOVE FUNCTION CURRENT-DATE (1:8) TO WS-TODAY
  ```

- * Get the current time in hhmmss format:

  ```
  MOVE FUNCTION CURRENT-DATE (9:6) TO WS-TIME
  ```

The other intrinsic date functions deal with converting between either Gregorian dates or Julian dates and an internal Integer format. This Integer format is simply the number of days since some predetermined, fixed date like 1/1/0001. These four conversion functions are:

- * Convert from Gregorian to Integer formats:

  ```
  COMPUTE WS-INTEGER-DATE = FUNCTION INTEGER-OF-DATE (WS-DATE)
  ```

- * Convert from Integer to Gregorian formats:

  ```
  COMPUTE WS-DATE = FUNCTION DATE-OF-INTEGER (WS-INTEGER-DATE)
  ```

- * Convert from Julian to Integer formats:

  ```
  COMPUTE WS-INTEGER-DATE = FUNCTION INTEGER-OF-DAY (WS-JULIAN-
  DATE)
  ```

- * Convert from Integer to Julian formats:

  ```
  COMPUTE WS-JULIAN-DATE = FUNCTION DAY-OF-INTEGER (WS-INTEGER-
  DATE)
  ```

All Gregorian and Julian dates are expected to have 4-digit years.

You might be asking just what in the heck would you want with this integer date. Well, with it calendar math is a breeze. You need to know what date is 150 days from today (and this kind of stuff happens more often than you'd think)? Convert today to an integer date, add 150 to it and convert it back. No more checking which months you're going through to see if it's a 30 day or 31 day month. No more leap year calculations. It's pretty automatic:

```
01  WS-TODAY          PIC 9(8).
01  WS-FUTURE-DATE    PIC 9(8).
....
....
MOVE FUNCTION CURRENT-DATE (1:8) TO WS-TODAY.
COMPUTE WS-FUTURE-DATE = FUNCTION DATE-OF-INTEGER (FUNCTION
        INTEGER-OF-DATE (WS-TODAY) + 150)
```

COMPUTE is OK because we're only using integers here.

How many days between two dates?

```
COMPUTE WS-DAYS = FUNCTION INTEGER-OF-DATE (WS-DATE-1) -
FUNCTION INTEGER-OF-DATE (WS-DATE-2)
```

Converting between Gregorian and Julian formats used to be a pain also. Now:

```
COMPUTE WS-DATE = FUNCTION DATE-OF-INTEGER (FUNCTION
INTEGER-OF-DAY (WS-JULIAN))
```

Numeric Functions: It's doubtful that there will be much use for most of the numeric functions available. Some of the more useful ones:

- * Calculate the square root of a number

  ```
  COMPUTE WS-RESULT = FUNCTION SQRT (WS-NUMBER)
  ```

- * Get the remainder from dividing two integers

  ```
  COMPUTE WS-RESULT = FUNCTION MOD (WS-INTEGER-1, WS-INTEGER-2)
  ```

- * Get the remainder from dividing two non-integers

  ```
  COMPUTE WS-RESULT = FUNCTION REM (WS-NUMBER-1, WS-NUMBER-2)
  ```

There are also geometric functions (SIN, COS, TAN), log functions (LOG, LOG10), FACTORIAL and even RANDOM for generating random numbers.

Try It Out

Problem Statement:

Accept a sentence from the JCL SYSIN using ACCEPT command and perform the string functions to extract a part of the string from Left, from right and from anywhere in the middle.

Code:

```
Code for STRING Statement
STRING W01-INPUT-01
       ' - '
       W01-INPUT-04
       DELIMITED BY SIZE
       INTO W02-STRING
END-STRING.

Code for UNSTRING Statement
UNSTRING W01-INPUT
DELIMITED BY '  '
     INTO W02-UNSTRING-01
          W02-UNSTRING-02
          W02-UNSTRING-03.

Code for extracting a part of the string
MOVE W01-INPUT-02(3:W01-INPUT-04) TO W02-SUBSTRING.

Code for INSPECT Statement
INSPECT W01-INPUT-02 TALLYING W03-TEMP-01 FOR LEADING SPACES.
```

Refer File Name: Handout_Example_Code.txt Code for Session #9 to obtain soft copy of the program code

How It Works:

- The job accepts the inputs from the JCL from the SYSIN using the ACCEPT statement.
- The output is displayed in the JCL SYSOUT using the DISPLAY statements.
- The process para performs the string operations like STRING, UNSTRING, INSPECT and so on.
- The last statement “STOP RUN” indicates the end of execution.

Summary

- The STRING statement is used to concatenate items together that are not numeric.
- The UNSTRING statement is used to parse individual items from within a single string.
- The starting position and length as parameters are used to extract a part of a string, for example MOVE W01-TEXT-01(4:5) TO W01-TEXT-02.
- The INSPECT statement is used to count occurrences of characters or strings within another string.

Test Your Understanding

1. What will be present after the execution of following statement?

```
01  WS-REC-1.
    02      NAME        PIC   X(20).
    02      AMT-1 PIC   9(5)V99.
    02      AMT-2 PIC   9(5)V99.
    02      TOTAL PIC   9(6)V99.
INITIALIZE WS-REC-1.
INITIALIZE WS-REC-1 REPLACING NUMERIC BY 1000.
MOVE 2000 TO AMT-2.
COMPUTE TOTAL = AMT-1 + AMT-2
```

   a) What value will be present in TOTAL?

2. The string function used to concatenate is ______.
3. The Prefix, Suffix Sub-string operations could be performed using starting position and ______ as parameters.
4. The occurrence of a character is a string can be obtained using _______ statement.
5. The UNSTRING statement is used to split the string in several fragments based on a delimiter. State true or false.

Exercises

A field name STRING-1 contains 80 chars. The char (/) or (,) is used to indicate the end of a word within these 80 chars. Write the COBOL statement to find the number of words in STRING-1 and the length of the individual words

Session 11, 12 and 13: Compiling Concepts and Error Handling

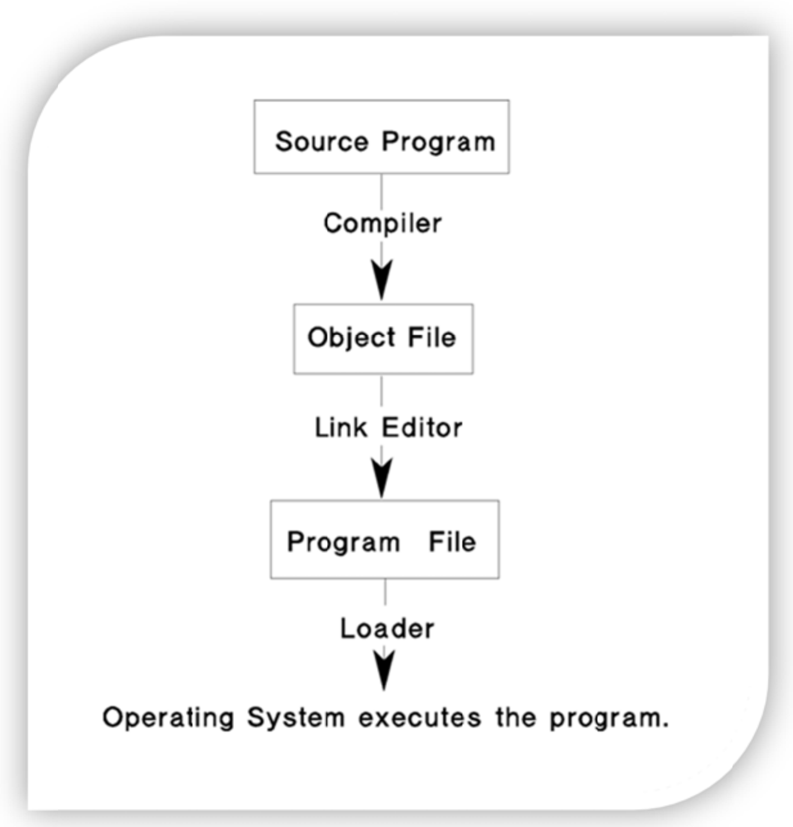
Learning Objectives

After completing this session, you will be able to:

- ❑ Describe the concept of compiling, linking, and executing
- ❑ Compile options that affect performance
- ❑ Identify common errors with regard to arithmetic operations and string functions

Compiling Concept

- ❑ On compiling a program, the COBOL compiler translates COBOL source program to object code and resolves calls to nested programs.
- ❑ On linking a program, the link editor resolves calls to subprograms that are to be bound at link time. The link editor uses object code to produce a program file.
- ❑ On executing a program, the loader loads the program file into memory and the operating system executes it.
- ❑ The default options are set up when compiler is installed.
- ❑ One can override them with other options in the compile JCL.
- ❑ To find out default compiler options in effect, run a test compilation without specifying any option. The output listing lists the default options specified by your installation.
- ❑ The compile options are handled in detail in the Compile Options Section



Steps involved in Compiling

- If the source program consists of a main program and one or more subprograms, then the main program and each subprogram must be compiled separately.
- The resulting object files must be linked together into a single program file.
- An object file cannot contain more than one program.
- Object module is the input for link-edit, which ends up in an executable module known as Load Module
- The coder needs to define the data sets needed for the compilation and specify any compiler options necessary for the program and the desired output
- The compiler translates your COBOL program into language that the computer can process (object code)
- Use compiler-directing statements and compiler options to control your compilation

Error Handling for Strings

STRING:

- When stringing or unstringing data, the pointer might fall out of the range of the receiving field.
- COBOL does not allow the overflow to happen and the STRING or UNSTRING operation will not be completed and the receiving field is not changed.

- If ON OVERFLOW clause is not present on the STRING or UNSTRINGS statement, then control passes to the next sequential statement, and notification is not provided for the incomplete operation.

Arithmetic Operations:

- The results of Arithmetic operations might have a value than what the fixed-point field can hold them. Example: Considering a 4 digit value attempted to store in a variable which can hold maximum 3 digits
- Possibility to have a division by zero
- In either case, the ON SIZE ERROR clause after the ADD, SUBTRACT, MULTIPLY, DIVIDE, or COMPUTE statements can handle the situation.
- The ON SIZE ERROR clause will execute the imperative statement and the result field will not be changed in the following five cases:
  - Fixed-point overflow
  - Division by zero
  - Zero raised to the zero power
  - Zero raised to a negative number
  - A negative number raised to a fractional power

Summary

- ❑ The basic compiling concepts include:
  - o Compiling
  - o Linking
  - o Executing
  - o Default Compiler Options
  - o Performance related Compile Options
- ❑ Common Errors:
  - o **String Functions:** Pointers falling out of Range, Overflow situation
  - o **Arithmetic Operations:** Possibility of division by Zero and Storing value beyond the capacity of the variable

Test Your Understanding

ON SIZE ERROR clause is not applicable to COMPUTE statement. State true or false.

Session 14: Picture Clause Editing

Learning Objectives

After completing this session, you will be able to:

- ❑ Explain the various types of Picture Clause Editing

Picture Clause

- ❑ Specified for every Elementary Item
- ❑ pic-string1 can contain a maximum of 30 Code Characters
- ❑ Code Character repetitions can be indicated by an integer within ()

Code Char	Meaning	Sample Picture
9	Numeric Data Item	9999
X	Any allowable character from the character set	XXXX
A	a Letter or Space	AAAA
V	Assumed Decimal Point	99V99
P	Position of Assumed Decimal Point when it lies outside	99PPV PPP99
S	Data Item is Signed	S9(4)
B	Inserts space	XXBXXX
0	Inserts zero	9(5)000
/	Inserts Slash	XX/XX/XX
,	Inserts Comma	9,999
.	Inserts Decimal Point	9,999.99
*	Replaces leading zeros	****9.99
$	Inserts Currency sign	$9,999.99
+ - CR DB	Inserts Sign	+99.99 -99.99 99.99DB 99.99CR

Valid Ranges for Signed Variables:

PICTURE	Valid Range of Values
9999	0 through 9999
S99	-99 through +99
S999V9	-999.9 through +999.9

Types of Picture clause editing

- ❑ Insertion:
  - o Simple insertion
  - o Special insertion
  - o Fixed insertion
  - o Floating insertion
- ❑ **Suppression and Replacement:** Zero suppression and replacement with asterisks or spaces
- ❑ EDITING ALLOWED ONLY for Categories:
  - o Numeric-edited (All edits)
  - o Alphanumeric-edited (Only Simple insertion)

Simple Insertion

- ❑ Done using 'B', '0', '/'
- ❑ Applicable only for numeric edited

PICTURE	Value of Data	Edited Result
X(2)BX(3)BX(4)	19Mar1995	19 Mar 1995
9(2)/9(2)/9(2)	190395	19/03/95
99,B999,B000	1234	01, 234, 000
99,999	12345	12,345

Special insertion

Period (.) is the special insertion symbol; it also represents the actual decimal point for alignment purposes

PICTURE	Value of Data	Edited Result
999.99	1.234	001.23
999.99	12.34	012.34
999.99	123.45	123.45
999.99	1234.5	234.50

Fixed insertion

- ❑ - $ (Currency symbol)
- ❑ + - CR DB (editing-sign control symbols)

PICTURE	Value of Data	Edited Result
999.99+	+6555.556	555.55+
+9999.99	-6555.555	-6555.55
9999.99	+1234.56	1234.56
-$999.99	-123.45	-$123.45
-$999.99	-123.456	-$123.45
-$999.99	+123.456	$123.45

PICTURE	Value of Data	Edited Result
$9999.99CR	+123.45	$0123.45
$9999.99DB	-123.45	$0123.45DB
$9999.99DB	+123.45	$0123.45
$9999.99CR	-123.45	$0123.45CR

Floating insertion

- Done using ‘$’, ‘+’, ‘-’
- Mutually exclusive

PICTURE	Value of Data	Edited Result
$$$$.99	.123	bbb$.12
$$$9.99	.12	bb$0.12
$,$$$,999.99	-1234.56	bbbb$1,234.56
+,+++,999.99	-123456.789	b-123,456.78
$$,$$$,$$$.99CR	-1234567	$1,234,567.00CR
++,+++,+++.+++	0000.00	Bbbbbbb

Zero suppression and replacement

- Z * (Mutually exclusive)
- as floating replacement ‘Z’, ‘*’, ‘+’, ‘-‘, ‘$’ are mutually exclusive

PICTURE	Value of Data	Edited Result
****.**	0000.00	****.**
ZZZZ.ZZ	0000.00	Bbbbbbb
$$$$.$$	0000.00	Bbbbbbb
$$$$.$$	0000.02	$.02
ZZZZ.99	0000.00	bbbb.00
ZZ99.99	0000.00	bb00.00
Z,ZZZ.ZZ+	+123.456	bb123.45+
*,***.**+	-123.45	**123.45-
,*,***.**+	+12345678.9	12,345,678.90+
$ZZ,ZZZ.ZZCR	+1234.56	$b1,234.56bb
$B*,***,***.**BBDB	-12345.67	$b ***12,345.67bbDB

Try It Out

Problem Statement:

ACCEPT an input from the JCL SYSIN and DISPLAY the value in all type of PICture clause editing.

Code:

```
01  W01-INPUT.
    05  W01-INPUT-01                       PIC 9(05).
01  W02-OUTPUT.
    05  W02-POS-SIGN                       PIC S9(05).
    05  W02-NEG-SIGN                       PIC S9(05).
    05  W02-NO-SIGN                        PIC 9(05).
    05  W02-POS-SIGN-DEC                   PIC S9(05)V9(02).
    05  W02-NEG-SIGN-DEC                   PIC S9(05)V9(02).
    05  W02-NO-SIGN-DEC                    PIC 9(05)V9(02).
    05  W02-NO-SIGN-DEC-DOT                PIC 9(05).9(02).
    05  W02-ZERO-SUPRESS                   PIC ZZZZZ.99.
    05  W02-DOLLAR                         PIC $$$$$.99.
    05  W02-PPP                            PIC 999PP.
```

Refer File Name: Handout_Example_Code.txt under heading 'Code for Session #14' to obtain soft copy of the program code.

How It Works:

- The job accepts the inputs from the JCL from the SYSIN using the ACCEPT statement.
- The various parameters are calculated and moved to the display variables.
- Based on the declaration type each of the variables are displayed in JOB SYSOUT.

Summary

You should remember the concepts on the following

- Signed and unsigned variables,
- Implied decimal point and placeholders
- Other representation like Zero suppress, dollar prefix, and so on

Test Your Understanding

1. Does 'Implied Decimal' consume extra memory for the decimal?
2. Do 'sign' consume extra memory for declaration PIC S9(02)?
3. 'Zero suppress' representation coverts ___________ to ______________.
4. What is length of the variable S9(05).9(02)?

Session 15: Conditional Expressions

Learning Objectives

After completing this session, you will be able to:

- Execute the Conditional Expression statements in a COBOL program

Conditional Expressions

- Types of Conditional Expressions:
  - Relation
  - Class
  - Sign
  - Condition name
- Cause the processing to select between alternate paths depending on the truth value of the expression

Relation Operators

- Causes a comparison of two operands. Operands can be an Identifier, Literal or Arithmetic Expression.

Syntax:

```
Operand-1 relop operand-2
```

- Operand-1, operand-2 can be an identifier, literal, arithmetic expression, index-name or a pointer.

Relational Operators:

```
IS [NOT] GREATER THAN                [NOT]  >
IS [NOT] LESS THAN                         [NOT]  <
IS [NOT] EQUAL TO                          [NOT]  =
IS GREATER THAN OR EQUAL TO     > =
IS LESS THAN OR EQUAL TO              < =
```

- For numeric operands the comparison is algebraic

Example:

012
12.00
12
+12 are all-equal

012

12.00

12

+12 are all-equal

Comparison permitted regardless of usage

Non-Numeric Operands or One Numeric and One Non-Numeric

- Comparison is made as per the collating sequence
- If both operands’ Usage are same and relational operator = or NOT = no conversion
- All other comparison operands with Usage other than display are converted to display
- Comparison starts from left most character
- For operands size unequal the comparison proceeds as if the shorter operand had been padded by blanks on the right

Example:

```
       l-A   “JOES”
       l-B   “JOE” taken as “JOE “
       l-A > l-B is TRUE
```

Class Condition

- Numeric or alphabetic or user defined class-name check of the operand
- Numeric test cannot be done for an item declared as alphabetic (PIC A)
- Alphabetic test cannot be done for item declared as Numeric (PIC 9)
- Packed-decimal allowed for numeric test

Syntax:

```
 IF id-1 IS [NOT]
      {NUMERIC
       ALPHABETIC
       ALPHABETIC-LOWER
       ALPHABETIC-UPPER
       class-name}
```

SIGN CONDITION:

```
Operand-1 IS [ NOT ]
            {POSITIVE
             NEGATIVE
             ZERO}
```

- Determines whether algebraic value is less than greater than or equal to zero
- Operand-1 must be a numeric identifier, or arithmetic containing at least one reference to a variable

Condition Name

- It is a subset of relational condition
- It tests a conditional variable to determine whether its value is equal to any value(s) associated with the condition-name.
- Condition-name is used in conditions as an abbreviation for the relation condition.

Syntax:

```
condition-name-1
```

Conditional expression:

Examples:

```
IF ASSOC-AGE > 80
    SET SENIOR-ASSOC TO TRUE
END-IF

IF ASSOC-SAL IS POSITIVE
    SET DEBIT-AMOUNT TO TRUE
END-IF
```

Complex Conditions

- Formed by:
  - Combining simple conditions
  - Nested IFs
  - Logical Connectives
- Logical Operators:
  - Used to combine simple conditions
  - Abbreviating conditions
  - For negation

Logical Operators

- **NOT:** Logical Negation
- **OR:** Logical Inclusive
- **AND:** Logical Conjunction

Evaluation Rules

- Precedence:
  - Arithmetic Expression
  - All Relational Operators
  - NOT
  - AND
  - OR
  - If a condition is put in parenthesis, it is evaluated and its truth value depends on the truth value of its constituents

Abbreviated Compound Conditions

`IF A = B AND A = C` is same as

`IF A = B AND = C`

- Here identical subjects are omitted in a consecutive sequence of relational conditions
  - o `IF A = B AND A = C` is same as `IF A = B AND C`
- Here identical subjects and relational operators are omitted in a consecutive sequence of relational conditions
- As indicated in the examples compound conditions can be abbreviated by having implied subjects, or, implied subjects and relational operators.

Examples:

```
A > B OR NOT C OR D
      = (A>B) OR NOT (A>C) OR (A>D)

A > B OR NOT C OR NOT < D
      = (A>B) OR (A NOT > C) OR  (A NOT< D)
```

Introduction to IF Statement

Syntax - Format 1:

```
IFcondition-1 THEN
{ statement-1    }
{ NEXT SENTENCE  }
[ELSE
{ statement-1    }
{ NEXT SENTENCE  }]
[END-IF]
```

Description: The IF statement is used to test various conditions. The ELSE clause is used to provide code to be executed for the alternative condition. When the statement is executed, the condition after IF is evaluated. If the condition is true, the statement(s) between the condition and the ELSE clause are executed and control is then transferred to the statement following the END-IF. If the condition is false, the code between the ELSE and the END-IF, if any, is executed.

Tips:

- The IF statement can be used to test for various kinds of conditions, relational tests, class tests, sign tests, or condition name tests. See conditions for more detail.
- The logical operators AND, OR, and NOT may be used to combine conditions into complex conditions. If these operators are used, the order of precedence is from highest to lowest: AND, OR, and finally NOT, unless the order is overridden with the use of parentheses.

IF Statement - Example

```
IF (WS-NT91-CLT NOT EQUAL ‘99999’)
AND WS-NT91-SEQ NOT NUMERIC
      Statements 1
ELSE
     Statement 2
END-IF
```

Using conditional-names:

```
IF C08W-ERR-OPEN-FLAG
     GO TO 999-ERR-OPEN-PARA
END-IF
```

Introduction to Evaluate Statement

Syntax - Format 1:

```
            identifier-1                 identifier-2
            literal-1                    literal-2
EVALUATE  { arith-expr-1 } [  ALSO  {  arith-expr-2 } ] ...
            condition-1                  condition-2
            TRUE                         TRUE
            FALSE                        FALSE

         ANY
         condition-3
         TRUE
         FALSE
{{WHEN {                                                      }
                   identifier-
                   3
         [NOT] {{  literal-3  }[  { THROUGH } { literal-4 }]}
                   arith-expr-      THRU        arith-expr-
                   3                            4
                                                identifier-
                                                4

        ANY
        condition-4
        TRUE
        FALSE
[ALSO {                                                  }  ]
                                           identifier-      ...}...
                                           6
        [NOT]  {{ identifier-5 }[ { THROUGH } { literal-6 }]}
                  literal-5         THRU        arith-expr-
                  arith-expr-5                  6

      imperative-statement-1}...
[WHEN OTHER imperative-statement-
2]
[END-EVALUATE]
```

Description: The EVALUATE statement is used to test a variable for multiple different values. As such, it is an implementation of the standard case construct. The ALSO clause is used to test for values of additional variables as the same time. To execute the same code for multiple conditions, code all the applicable conditions on multiple WHEN clauses. When the statement is executed, the variable after EVALUATE is compared to the each of the values in the WHEN clauses in turn. When a match is found the statement(s) following the WHEN clause and before the next WHEN clause, are executed and control is then transferred to the statement following the END-EVALUATE. If no matching WHEN clause is found, the code following the WHEN OTHER statement is executed. If the ALSO clause is used, then the value following the ALSO in the EVALUATE must match the value in the ALSO associated with the WHEN clause.

For example:

```
EVALUATE FILE-STATUS
WHEN "00"
    PERFORM SUCCESSFUL-ROUTINE
WHEN "10"
    PERFORM END-OF-FILE-ROUTINE
WHEN OTHER
    PERFORM IO-ERROR-ROUTINE
END-EVALUATE.
```

In this example, if the file status has a value of "00", SUCCESSFUL-ROUTINE will be performed, but if the file status has a value of "10", the END-OF-FILE-ROUTINE will be executed. If neither value is in file status, then IO-ERROR-ROUTINE will be executed. Here is the same test using condition names instead of relational tests:

```
EVALUATE TRUE
WHEN I-O-OK
    PERFORM SUCCESSFUL-ROUTINE
WHEN END-OF-FILE
    PERFORM END-OF-FILE-ROUTINE
WHEN OTHER
    PERFORM IO-ERROR-ROUTINE
END-EVALUATE.
```

Tips:

- Use multiple WHEN clauses to implement code that works like an OR.
- Use the ALSO clause to implement code that works like an AND.
- Use ANY if the value of one of the variables is not needed in one of the conditions.

EVALUATE Statement:

Example 1

```
EVALUATE proc-type ALSO
            cust-type
WHEN 1 ALSO 1
```

```
              MOVE 1 TO RESULT
WHEN 3 ALSO 1 THRU 2
              MOVE  2 TO RESULT
WHEN 2 ALSO 1
              MOVE 3 TO RESULT
WHEN OTHER
             MOVE 0 TO RESULT
END-EVALUATE
```

In this example subject of Evaluation is an Expression. Depending on these values the control passes to the corresponding WHEN.

```
EVALUATE TRUE
WHEN l-VALUE < 100
              MOVE 1 TO RESULT
WHEN l-VALUE < 1000
              MOVE  2 TO RESULT
WHEN l-VALUE < 10000
WHEN OTHER
             MOVE 0 TO RESULT
END-EVALUATE
```

In this example, the subject is TRUTH value. Depending on the values the WHEN phrase satisfying the TRUTH value is executed.

If a WHEN does not have an associated imperative statement it is a do-nothing condition (same as coding CONTINUE)

Creating Complex Conditions with Logical Operators

Any of the conditions described above can be combined to form more complex conditions through the use of any of the following logical operators. These logical operators may be combined as needed to create a multitude of complex conditions.

AND: If two conditions are combined with an AND, then both conditions must be true for the combined expression to be true. For example,

```
ABC > DEF AND GHI IS NUMERIC
```

Will true only if the current value of the data-item ABC exceeds the current value of the data item DEF and GHI contains only values in the range 0 - 9.

OR: If two conditions are combined with an OR, then either condition must be true for the combined expression to be true. For example,

```
JKL < MNO OR PQR IS ALPHABETIC
```

is true if either the current value of the data-item MNO exceeds the current value of the data item JKL or PQR contains only values in the range A - Z, a - z, or spaces or both.

NOT: Any condition may be preceded with the word NOT in order to negate that condition. In relational and class tests, the word NOT may be moved into the middle of the condition for readability. For example,

```
STU NOT NUMERIC
```

Will true if there are any characters not in the range 0 - 9 in the data item.

Try It Out

Problem Statement:

Declare a variable and assign a value to it using MOVE Statement and validate the value and a specific condition and display the result accordingly.

Code:

```
PROCEDURE DIVISION.
*================================================================*
* MAIN PARA - CALLS INPUT AND PROCESS PARA
*================================================================*
 0000-MAIN-PARA.
     PERFORM 1000-IF-STATEMENT
        THRU 1000-IF-STATEMENT-EXIT.

     MOVE ZEROES  TO W01-INPUT-NUM.
     DISPLAY '--CONDITIONAL EXPRESSIONS (EVALUATE)--'.
     PERFORM 2000-EVALUATE-STATEMENT
        THRU 2000-EVALUATE-STATEMENT-EXIT
       UNTIL W01-INPUT-NUM > 2.
     STOP RUN.

 0000-MAIN-PARA-EXIT.
     EXIT.

*================================================================*
* CONDITIONAL EXPRESSIONS - IF STATEMENT
*================================================================*
 1000-IF-STATEMENT.
     MOVE 'TRUE ' TO W01-INPUT-FLAG.
     DISPLAY '----CONDITIONAL EXPRESSIONS (IF)----'.
     IF W01-INPUT-FLAG-TRUE THEN
        DISPLAY 'FLAG = TRUE'
     ELSE
        DISPLAY 'FLAG = FALSE'
     END-IF.
     MOVE 'FALSE' TO W01-INPUT-FLAG.
     IF W01-INPUT-FLAG-TRUE  THEN
        DISPLAY 'FLAG = TRUE'
     ELSE
        DISPLAY 'FLAG = FALSE'
     END-IF.
     DISPLAY ' '.

 1000-IF-STATEMENT-EXIT.
     EXIT.

*=================================================================*
* CONDITIONAL EXPRESSIONS - EVALUATE STATEMENT
*=================================================================*
 2000-EVALUATE-STATEMENT.
     EVALUATE  W01-INPUT-NUM
        WHEN  0
            DISPLAY 'VARIABLE VALUE IS = ZERO'
        WHEN  1
            DISPLAY 'VARIABLE VALUE IS = ONE'
        WHEN  OTHER
            DISPLAY 'VARIABLE VALUE IS > ONE'
     END-EVALUATE.
     COMPUTE W01-INPUT-NUM = W01-INPUT-NUM + 1.

 2000-EVALUATE-STATEMENT-EXIT.
     EXIT.
```

Refer File Name: Handout_Example_Code.txt under heading 'Code for Session #14' to obtain soft copy of the program code

How It Works:

- The job accepts the inputs from the JCL from the SYSIN using the ACCEPT statement.
- The input parameters are validated based on the pre-defined value of the flag.
- Based on the declaration type each of the variables is displayed in JOB SYSOUT.

Summary

You should remember the concepts on the following

- Simple Conditions like `IF` statement
- Complex Conditions like `EVALUATE` statement
- Looping Structures like `PERFORM UNTIL` statement
- Logical Operators

Test Your Understanding

1. When does the code following the OTHER clause be executed in EVALUATE statement?
2. How to implement code that works like an OR logical operator?
3. How to implement code that works like an AND logical operator?
4. In an IF statement what is the order of precedence assuming that the order is not overridden with the use of parentheses

Session 19 and 21: File Handling

Learning Objectives

After completing this session, you will be able to:

- Explain the basic concepts of file handling in COBOL
- Declare files in the Environment Division and Data Division
- Access sequential files
- Explain the concepts of VSAM file handling in COBOL
- Use copy books for defining Record formats for Sequential and VSAM Files
- Handle different File Status Codes for Sequential and Indexed Sequential Files

File Management in COBOL

COBOL has support to three principle file organizations:

- Sequential organization
- Relative organization (Random organization or Direct Access)
- Indexed Sequential organization

All these three organizations can be used on DISK files; only the SEQUENTIAL organization can be used with TAPE files.

All files must be declared in the ENVIRONMENT division by proper SELECT statements which should contain a phrase describing to the compiler what type of organization the file should have. A second important declaration should be placed in the FILE-SECTION where the programmer describes the record format associated to the file in FD blocks.

Finally, the PROCEDURE division should have proper OPEN/CLOSE and READ/WRITE etc. statements used in compliance with the logic (algorithm) of the program.

The Sequential File Organization

In this file organization, the records of the file are stored one after another both physically and logically. That is, record with sequence number 16 is located just after the 15th record.

A record of a sequential file can only be accessed by reading all the previous records.

The records are discriminated from one another using the record length declared in the associated FD statement of the FILE-SECTION. For example, if the record structure that the programmer has declared is 52 bytes, blocks of 52 byte data (records) are assumed to place one after another in the file. If the programmer is reading the data in a sequential file, every READ statement brings 52 bytes into the memory.

If the file contains, say, 52 byte records; but the programmer tries to read this file with a program which has declared 40 byte records (that is, the total length of the FD structure is 40 bytes), the program will certainly read some pieces of information into the memory but the after the first READ statement, some meaningless pieces of records will be brought into memory and the program will start processing some physical records which contain logically meaningless data.

It is the programmer's responsibility to take care of the record sizes in files. You must be careful when declaring record structures for files. Any mistake you make in record sizes will cause your program to read/write erroneous information. This is especially dangerous if the file contents are being altered (changed, updated).

Since the records are simply appended to each other when building SEQUENTIAL files, you simply end up with a STREAM of byte. If this string does not contain any "Carriage Return/Line Feed" control characters in it, the whole file will appear as a single LINE of character and would be impossible to process with regular text editors. As you should know by now, text editors are good in reading/writing/modifying text files. These programs will assume that the file consists of LINES and expect the lines to be separated from each other by a pair of control characters called "Carriage Return/Line Feed" (or CR/LF).

COBOL has a special type of sequential file organization, which is called the LINE SEQUENTIAL ORGANIZATION which places a CR/LF pair at the end of each record while adding records to a file and expects such a pair while reading. LINE SEQUENTIAL files are much easier to use while developing programs because you can always use a simple text editor to see the contents of your sequential file and trace/debug your program.

Please note that LINE SEQUENTIAL files have two extra characters for each record. For files, which have millions of records, this might use up a significant amount of disk space.

SEQUENTIAL files have only one ACCESS MODE and that is "sequential access". Therefore you need not specify an ACCESS MODE in the SELECT statement. Typical SELECT statements for SEQUENTIAL files are:

```
SELECT MY-FILE ASSIGN TO MYFILE
       ORGANIZATION IS SEQUENTIAL.
SELECT MY-FILE-2 ASSIGN TO MYFILE2
       ORGANIZATION IS LINE SEQUENTIAL.
```

In the FILE-SECTION, you must provide FD blocks for each file; hence for a sequential file you could have something like:

```
   FD MYFILE.
       01  MYFILE-REC.
           02 M-NAME PIC X(16).
           02 M-SURNAME PIC X(16).
           02 M-BIRTHDATE.
              03 M-BD-YEAR PIC 9999.
              03 M-BD-MONTH PIC 99.
              03 M-BD-DAY PIC 99.
```

Note: You must NOT provide record fields for the extra two CR/LF bytes in record descriptions of LINE SEQ files. Once you declare the file to be a LINE SEQ file, these two extra bytes are automatically taken in consideration and added for all new records that are added to a file.

Procedure Division Considerations: Like all files, sequential and line sequential files must be OPENed before they can be processes and CLOSEd when they are not needed anymore.

Seq and Line Seq files can be opened for :

- OUTPUT
- INPUT
- EXTEND
- I-O (Input-Output)

Opening a sequential file for OUTPUT means that the program will only issue WRITE statements to a NON-EXISTING and JUST CREATED file. Therefore, when you open a file for OUTPUT, COBOL assumes that the file does not exist and try to create a new one. IF THE FILE EXISTS, ITS CONTENTS WILL BE CLEARED WHEN OPENED AND ASSUMED TO BE A BRAND NEW FILE INTO WHICH RECORDS WILL BE ADDED. You should be very careful when opening a file for OUTPUT. One small mistake and all your valuable records are lost forever.

Opening a sequential file for INPUT means that the program will only issue READ statements to an EXISTING file. Therefore, when you open a file for INPUT, COBOL assumes that the file exists and tries to access it. IF THE FILE DOES NOT EXIST, AN ERROR MESSAGE WILL BE ISSUED INDICATING THAT THE MENTIONED FILE COULD NOT BE FOUND.

Opening a sequential file for EXTEND means that the program will add NEW RECORDS to an EXISTING file. Therefore, when you open a file for EXTEND, COBOL assumes that the file exists and subsequent WRITE statements will try to ADD NEW RECORDS at the end of the existing file (in other words; the append mode). Records can only be added to the END of a sequential file.

Some examples are:

```
OPEN OUTPUT NEW-FILE.
OPEN INPUT OLD-FILE.
OPEN EXTEND OLD-FILE OLD-FILE2.
OPEN INPUT OLD-FILE OUTPUT NEW-FILE.
```

When you are finished with a file you must CLOSE it.

```
CLOSE NEW-FILE.
CLOSE OLD-FILE NEW-FILE.
```

You can close more than one file with a single CLOSE statement. When a COBOL program terminates, all files mentioned in the program are automatically closed; therefore you will not get an error message for those files you forget to close or do not close at all. Please note that relying on COBOL to close your files is not a proper programming style. Although not absolutely necessary, you should close your files when you are done with them.

The CLOSE statement has a very important function. Sometimes, your program needs to re-start reading a sequential file from the beginning. (Please note that every READ statement you execute brings the next record into memory and when you skip a record, you cannot backspace the file to previous records.) In such a situation, you will have to REWIND the file that is making the first record the active one in memory. You can achieve this only by closing the file and re-opening it for INPUT.

You should be careful not to open an already opened file. Closing already closed files usually do not cause any problems but should be avoided. You should make sure that your program opens and closes files at proper places in your logic flow so that no input-output is tried on closed files.

Simple INPUT-OUTPUT Statements for SEQ Files: Once you open a seq file, you can use READ or WRITE statements to read or append records to this file. You cannot use a READ statement for a file opened for OUTPUT or EXTEND and similarly you cannot use a WRITE statement for a file you have opened for INPUT.

A typical READ statement looks like:

```
READ MY-FILE.
```

A good programmer must check whether the READ statement could find a record to read successfully. That means; the programmer must check whether there were any records to READ when the statement was executed. The situation can be checked with the following construct:

```
READ MY-FILE AT END PERFORM NO-RECORDS-FOUND.
```

The construct tells the COBOL compiler to execute the paragraph labeled NO-RECORDS-FOUND when the READ statement cannot find any records to read.

Another example could be:

```
READ MY-FILE AT END DISPLAY "No more records!"
      CLOSE MY-FILE
      STOP RUN.
```

Once your program EXECUTES a successful READ statement, the information in the data record that was just brought into memory will be available in the corresponding variables mentioned in your record description declaration (the FD block).

When you want to put records in a seq file, you have to open it for OUTPUT or EXTEND (depending on the requirements of the algorithm of your program). The COBOL statement that is used to put records into a file is the WRITE statement. The important point that you should be careful with are that,

- The file should be opened
- New values for the field variables of the FD record description must be moved to proper variables of the record
- The RECORD NAME is specified in the WRITE statement and NOT the FILE NAME.

This will be best described by an example:

Suppose you want to add a new record to the file

```
       SELECT MY-FILE ASSIGN TO DISK MYFILE
             ORGANIZATION IS SEQUENTIAL.
       ....
       FD  MY-FILE.
       01  MY-FILE-REC.
             02 M-NAME PIC X(16).
             02 M-SURNAME PIC X(16).
             02 M-BIRTHDATE.
                   03 M-BD-YEAR PIC 9999.
                   03 M-BD-MONTH PIC 99.
                   03 M-BD-DAY PIC 99.
```

You must first open it, then move new values to the fields in MYFILE-REC and finally WRITE the record into the file.

```
        OPEN EXTEND MY-FILE.
        ....
        MOVE "UGUR" TO M-NAME.
        MOVE "AYFER" TO M-SURNAME.
        MOVE 1955 TO M-BD-YEAR.
        MOVE 1 TO M-BD-MONTH.
        MOVE 1 TO M-BD-DAY.
        WRITE MYFILE-REC.
        .....
```

Updating Records of a Sequential File: Sometimes you need to make some small changes in the contents of a file. Of course it possible to create a brand new file with the new, modified contents but this is not practical. COBOL provides you with an OPEN I-O mode with which you can modify only the required record in a file.

In other words, there is another file opening mode; the I-O mode; and another special REWRITE statement. Suppose that you want to change the surname field of the 20th record of a sequential file (originally "AYFER") into "AYFEROGLU".

The program that you can write could read like

```
        SELECT MY-FILE ASSIGN TO DISK MYFILE
               ORGANIZATION IS SEQUENTIAL.
        ....
        FD  MY-FILE.
        01  MY-FILE-REC.
            02 M-NAME PIC X(16).
            02 M-SURNAME PIC X(16).
            02 M-BIRTHDATE.
               03 M-BD-YEAR PIC 9999.
               03 M-BD-MONTH PIC 99.
               03 M-BD-DAY PIC 99.

        PROCEDURE DIVISION.
        ....
            OPEN I-O MY-FILE.
            ....
            MOVE 20 TO N.
            ....
            PERFORM READ-A-REC N-1 TIMES.
      *
      * TO MAKE SURE THAT WE ARE ON THE CORRECT RECORD
      *
            IF M-SURNAME NOT = "AYFER" DISPLAY "WRONG RECORD"
                                       DISPLAY "RECORD NOT UPDATED"
                                       CLOSE MYFILE
                                       STOP RUN.
            MOVE "AYFEROGLU" TO M-SURNAME.
            REWRITE MY-FILE-REC.
            .....
            CLOSE MY-FILE.
            ....
```

Deleting Records of a Sequential File: I is NOT possible to delete records of a sequential file. If you do not want a specific record to be kept in a sequential file any more, all you can do is to modify the contents of the record so that it contains some special values that your program will recognize as deleted (remember to open the file in I-O mode and REWRITE a new record).

Advantages and Disadvantages of using Sequential Files:

Advantages:

- Very easy to process,
- Can be easily shared with other applications developed using different programming languages

Disadvantages: Can be processed sequentially. If you need to read record number N, you must first read the previous N-1 records. It is not efficient especially for programs that make frequent searches in the file.

The Relative File Organization

In this file organization, the records of the file are stored one after another both physically and logically. That is, record with sequence number 16 is located just after the 15th record.

In contrast to SEQ files, records of a RELATIVE file can be accessed by specifying the record sequence number in the READ statement (the KEY) and without needing to read all the previous records.

It is the programmer's responsibility to take care of the record sizes in files. You must be careful when declaring record structures for files. Any mistake you make in record sizes will cause your program to read/write erroneous information. This is especially dangerous if the file contents are being altered (changed, updated).

Please note that you must provide the record structure to have a special field allocated to contain the KEY that is the record sequence number.

RELATIVE files can have RANDOM or SEQUENTIAL ACCESS MODEs. If the ACCESS MODE is declared to be RANDOM in the corresponding SELECT statement, the program can read the record with key value 2345 and then the record with key=3, then record 2344, and so on. In short, one can access any record of a relative file, in any order, provided that the KEY value is specified before the READ statement is executed.

If the ACCESS MODE is SEQUENTIAL, that means the records of the file will be accesses in their physical sequential order (just like SEQUENTIAL and LINE SEQUENTIAL files) and no specific KEY value be given for the READ statements; but instead, the NEXT clause will appear in READ statements meaning "Go get the record with the next consecutive key value.

```
        SELECT MY-FILE ASSIGN TO DISK MYFILE
               ORGANIZATION IS RELATIVE
               ACCESS MODE IS RANDOM
               RECORD KEY IS M-IDNO.

        SELECT MY-FILE-2 ASSIGN TO DISK MYFILE2
               ORGANIZATION IS RELATIVE
               ACCESS MODE IS SEQUENTIAL
               RECORD KEY IS M-IDNO.
```

In the FILE-SECTION, you must provide fields for the KEY, the numerical M-IDNO in this example, in the FD block of the RELATIVE file;

```
         FD   MY-FILE.
         01   MY-FILE-REC.
              02  M-IDNO        PIC 9999.
              02  M-NAME        PIC X(16).
              02  M-SURNAME     PIC X(16).
              02  M-BIRTHDATE.
                  03 M-BD-YEAR  PIC 9999.
                  03 M-BD-MONTH PIC 99.
                  03 M-BD-DAY   PIC 99.
```

Note: Since the key field will contain integer, record sequence numbers; you should declare the field to be numerical and take care that it is wide enough to carry all possible values for the key value. For example, if your file is expected to have 200,000 records in it, the key field should be declared at least 6 bytes (PIC 999999) so that it can hold the values 1 through 200,000. A key field declaration of "PIC 999" would let use a file of max 999 records in it.

Procedure Division Considerations: Like all files, relative files must be OPENed before they can be processes and CLOSEd when they are not needed anymore.

Relative files can be opened for:

- ❑ OUTPUT
- ❑ INPUT
- ❑ I-O (Input-Output)

Opening a relative file for OUTPUT means that the program will only issue WRITE statements to a NON-EXISTING and JUST CREATED file. Therefore, when you open a file for OUTPUT, COBOL assumes that the file does not exist and try to create a new one. IF THE FILE EXISTS, ITS CONTENTS WILL BE CLEARED WHEN OPENED AND ASSUMED TO BE A BRAND NEW FILE INTO WHICH RECORDS WILL BE ADDED. You should be very careful when opening a file for OUTPUT. One small mistake and all your valuable records are lost forever.

Once you open a relative file in Output mode, you are expected to write records in the INCREASING and CONSECUTIVE order of keys. That is, before writing the record with key 98 (98th record) you should first write the first 97 records.

Opening a relative file for INPUT means that the program will only issue READ statements to an EXISTING file. Therefore, when you open a file for INPUT, COBOL assumes that the file exists and tries to access it. IF THE FILE DOES NOT EXIST, AN ERROR MESSAGE WILL BE ISSUED INDICATING THAT THE MENTIONED FILE COULD NOT BE FOUND.

If you have declared the ACCESS MODE to be RANDOM, before each READ statement, you are supposed to move a valid KEY value to the record field variable that is declared as the RECORD KEY.

For instance;

```
OPEN INPUT MYFILE.
....
MOVE 23  TO M-IDNO.
READ MYFILE.
```

Good programmers should take precautions in their program to avoid error messages and subsequent abnormal terminations are an INVALID value for the record key specified before the READ statement.

The INVALID KEY clause handles this in COBOL.

```
MOVE 2300  TO M-IDNO.
READ MYFILE INVALID KEY  PERFORM OUT-OF-RANGE.
```

The INVALID KEY condition rises if the value in M-IDNO is zero, negative or has a value greater than the total number of records in the file.

If you have declared the ACCESS MODE as SEQUENTIAL, you should use the NEXT clause in the READ statement. Like

```
READ MYFILE NEXT AT END PEFORM NO-MORE-RECORDS.
```

When you are finished with a file you must CLOSE it.

```
CLOSE NEW-FILE.
CLOSE OLD-FILE NEW-FILE.
```

You can close more than one file with a single CLOSE statement. When a COBOL program terminates, all files mentioned in the program are automatically closed; therefore you will not get an error message for those files you forget to close or do not close at all. Please note that relying on COBOL to close your files is not a proper programming style. Although not absolutely necessary, you should close your files when you are done with them.

In order to rewind a relative file with ACCESS MODE RANDOM, you do not need to rewind it when you need to go the first record. Just move 1 to the record key and issue a READ statement.

Simple INPUT-OUTPUT Statements for RELATIVE Files: Once you open a relative file, you can use READ or WRITE statements to read or append records to this file. You cannot use a READ statement for a file opened for OUTPUT and similarly you cannot use a WRITE statement for a file you have opened for INPUT.

A typical READ statement for RANDOM access mode looks like:

```
MOVE 123 TO M-IDNO.
READ MY-FILE.
```

A good programmer must check whether the READ statement could find a record to read successfully. That means; the programmer must check whether there were any records with the indicated key value when the statement was executed. The situation can be checked with the following construct:

```
READ MY-FILE  INVALID KEY PERFORM  NO-RECORDS-FOUND.
```

The construct tells the COBOL compiler to execute the paragraph labeled NO-RECORDS-FOUND when the READ statement cannot find any record with the key value stored in the key field variable.

Once your program EXECUTES a successful READ statement, the information in the data record that was just brought into memory will be available in the corresponding variables mentioned in your record description declaration (the FD block).

When you want to put records in a relative file, you have to open it for OUTPUT. The COBOL statement that is used to put records into a file is the WRITE statement. The important point that you should be careful with are that,

- The file should be opened
- New values for the field variables of the FD record description must be moved to proper variables of the record
- The value of the KEY VARIABLE must have a valid value and this valid value should be one greater than the key value of the record written previously,
- The RECORD NAME is specified in the WRITE statement and NOT the FILE NAME.

Updating Records of a Relative File: Sometimes you need to make some small changes in the contents of a file. Of course it possible to create a brand new file with the new, modified contents but this is not practical. COBOL provides you with an OPEN I-O mode with which you can modify only the required record in a file.

In other words, there is another file opening mode; the I-O mode; and another special REWRITE statement. Suppose that you want to change the surname field of the 20th record of a relative file (originally "AYFER") into "AYFEROGLU".

The program that you can write could read like

```
    SELECT MY-FILE ASSIGN TO DISK MYFILE
                   ORGANIZATION IS RELATIVE
                   ACCESS MODE IS RANDOM
                   RECORD KEY IS M-IDNO.
     ....
     FD   MY-FILE.
     01   MY-FILE-REC.
          02  M-IDNO        PIC 9999.
          02  M-NAME        PIC X(16).
          02  M-SURNAME     PIC X(16).
          02  M-BIRTHDATE.
              03 M-BD-YEAR  PIC 9999.
              03 M-BD-MONTH PIC 99.
              03 M-BD-DAY   PIC 99.

     PROCEDURE DIVISION.
```

```
          ....
          OPEN I-O MY-FILE.
          ....
          MOVE 20 TO M-IDNO.
          READ MY-FILE INVALID KEY PERFORM KEY-ERROR.
     *
     * TO MAKE SURE THAT WE ARE ON THE CORRECT RECORD
     *
          IF M-SURNAME NOT = "AYFER" DISPLAY "WRONG RECORD"
                                     DISPLAY "RECORD NOT UPDATED"
                                     CLOSE MY-FILE
                                     STOP RUN.
          MOVE "AYFEROGLU" TO M-SURNAME.
          REWRITE MY-FILE-REC.
          .....
          CLOSE MY-FILE.
          .....
```

Deleting Records of a Relative File: The DELETE statement logically removes an existing record from a relative file. After successfully removing a record from a file, the program cannot later access it.

Two options are available for deleting relative records:

- ❑ Sequential access mode deletion
- ❑ Random access mode deletion

Deleting a relative record in sequential access mode involves the following:

1. Specifying ORGANIZATION IS RELATIVE in the Environment Division SELECT clause
2. Specifying ACCESS MODE IS SEQUENTIAL in the Environment Division SELECT clause
3. Opening the file for I-O
4. Using a START statement to position the record pointer, or sequentially reading the file up to the target record
5. Deleting the last read record

Deleting a relative record in random access mode involves the following:

1. Specifying ORGANIZATION IS RELATIVE in the Environment Division SELECT clause
2. Specifying ACCESS MODE IS RANDOM in the Environment Division SELECT clause
3. Opening the file for I-O
4. Moving the relative record number value to the RELATIVE KEY data name
5. Deleting the record identified by the relative record number

If the file does not contain a valid record, an invalid key condition exists.

The program that you can write to delete a relative record in Random:

```
SELECT MYFILE ASSIGN TO MYFILE
               ORGANIZATION IS RELATIVE
               ACCESS MODE IS RANDOM
               RELATIVE KEY IS M-IDNO.
```

```
DATA DIVISION.
FILE SECTION.
FD MY-FILE.
     01   MY-FILE-REC.
          02  M-IDNO        PIC 9999.
WORKING-STORAGE SECTION.
01  ID-KEY       PIC 99 VALUE 1.
PROCEDURE DIVISION.
    OPEN I-O MY-FILE.
    PERFORM A010-DELETE-RECORDS UNTIL ID-KEY = 00.
A010-DELETE-RECORDS.
    DISPLAY "TO DELETE A RECORD ENTER ITS RECORD NUMBER".
    ACCEPT ID-KEY.
    IF ID-KEY NOT = 00
       DELETE MY-FILE RECORD
              INVALID KEY DISPLAY "INVALID DELETE".
    CLOSE MY-FILE.
    STOP RUN.
```

Advantages of Relative Files:

- ❑ Quite easy to process,
- ❑ If you can know the key value of the record that you need to find, there is no need for a search and you can access the record almost instantaneously

Disadvantage of Relative Files:

- ❑ It can be only used in conjunction with consecutive numerical keys. If you key value range is 1-1000, you must have records for all 1000 possible keys, and that is, your file must have exactly 1000 records. If, for example, you are planning to use student ID numbers as numerical key values and the student id numbers are like 9653421232 (10 digits), you file must contain 9,999,999,999 records and this doesn't make sense. Of course, you should modify the student ID number structure to start from1 and extend till a few thousand.
- ❑ This disadvantage (only numerical and consecutive values for the key value) is overcome with a completely different file structure, namely the INDEXED SEQUENTIAL FILE.

The Indexed File Organization

In this file organization, the records of the file are stored one after another in the order they are added to the file.

In contrast to RELATIVE files, records of an INDEXED SEQUENTIAL file can be accessed by specifying an ALPHANUMERIC key in the READ statement (the KEY).

It is the programmer's responsibility to take care of the record sizes in files. You must be careful when declaring record structures for files. Any mistake you make in record sizes will cause an error and abnormal termination.

Please note that you must provide the record structure to have a special field allocated to contain the KEY.

INDEXED files can have RANDOM or SEQUENTIAL ACCESS MODEs. If the ACCESS MODE is declared to be RANDOM in the corresponding SELECT statement, the program can read the record with key value "ABC" and then the record with key="A12", then record with key "D2X", and so on. In short, one can access any record of an indexed file, in any order, provided that the KEY value is specified before the READ statement is executed.

If the ACCESS MODE is SEQUENTIAL, that means the records of the file will be accesses in their physical sequential order (just like SEQUENTIAL and LINE SEQUENTIAL files) and no specific KEY value be given for the READ statements; but instead, the NEXT clause will appear in READ statements meaning "Go get the record with the next consecutive key value.

```
SELECT MY-FILE ASSIGN TO DISK MYFILE
            ORGANIZATION IS INDEXED
            ACCESS MODE IS RANDOM
            RECORD KEY IS M-IDNO.


SELECT MY-FILE-2 ASSIGN TO DISK MYFILE2
            ORGANIZATION IS INDEXED
            ACCESS MODE IS SEQUENTIAL
            RECORD KEY IS M-IDNO.
```

In the FILE-SECTION, you must provide fields for the KEY, the M-IDNO in this example, in the FD block of the INDEXED file;

```
FD   MY-FILE.
01   MY-FILE-REC.
     02  M-IDNO          PIC XXXX.
     02  M-NAME          PIC X(16).
     02  M-SURNAME       PIC X(16).
     02  M-BIRTHDATE.
        03 M-BD-YEAR     PIC 9999.
         03 M-BD-MONTH   PIC 99.
         03 M-BD-DAY     PIC 99.
```

Note: Since the keys are not supposed to be numerical, you can use the X picture for key fields.

Procedure Division Considerations: Lie all files, Indexed files must be OPENed before they can be processes and CLOSEd when they are not needed anymore.

Indexed files can be opened for:

- ❑ OUTPUT
- ❑ INPUT

- I-O (Input-Output)

Opening an indexed file for OUTPUT means that the program will only issue WRITE statements to a NON-EXISTING and JUST CREATED file. Therefore, when you open a file for OUTPUT, COBOL assumes that the file does not exist and try to create a new one. IF THE FILE EXISTS, ITS CONTENTS WILL BE CLEARED WHEN OPENED AND ASSUMED TO BE A BRAND NEW FILE INTO WHICH RECORDS WILL BE ADDED. You should be very careful when opening a file for OUTPUT. One small mistake and all your valuable records are lost forever.

Once you open an indexed file in Output mode, you are expected to write records in the INCREASING order of keys. That is, before writing the record with key A98 you should first write all the records with key values alphanumerically less than "A98". Please note that indexed file keys are case sensitive; that is the key value "a89" is NOT EQUAL TO "A89"; in fact it is alphanumerically greater than "A89" because the ASCII value of "a" (97) is greater than the ASCII value of "A" (65).

Opening an indexed file for INPUT means that the program will only issue READ statements to an EXISTING file. Therefore, when you open a file for INPUT, COBOL assumes that the file exists and tries to access it. IF THE FILE DOES NOT EXIST, AN ERROR MESSAGE WILL BE ISSUED INDICATING THAT THE MENTIONED FILE COULD NOT BE FOUND.

If you have declared the ACCESS MODE to be RANDOM, before each READ statement, you are supposed to move a valid KEY value to the record field variable that is declared as the RECORD KEY.

For instance;

```
     OPEN INPUT MY-FILE.
     ....
     MOVE "X23"  TO M-IDNO.
     READ MY-FILE.
```

Good programmers should take precautions in their program to avoid error messages and subsequent abnormal terminations are an INVALID value for the record key specified before the READ statement.

The INVALID KEY clause handles this in COBOL.

```
     MOVE "2300"  TO M-IDNO.
     READ MY-FILE INVALID KEY PERFORM OUT-OF-RANGE.
```

The INVALID KEY condition rises if there is no record in the file with key value equal to the specified key value.

If you have declared the ACCESS MODE as SEQUENTIAL, you should use the NEXT clause in the READ statement. Like

```
     READ MY-FILE NEXT AT END PEFORM NO-MORE-RECORDS.
```

When you are finished with a file you must CLOSE it.

```
     CLOSE NEW-FILE.
     CLOSE OLD-FILE NEW-FILE.
```

You can close more than one file with a single CLOSE statement. When a COBOL program terminates, all files mentioned in the program are automatically closed; therefore you will not get an error message for those files you forget to close or do not close at all. Please note that relying on COBOL to close your files is not a proper programming style. Although not absolutely necessary, you should close your files when you are done with them.

REWINDING an indexed file with ACCESS MODE RANDOM is not meaningful. If the ACCESS MODE is SEQUENTIAL, you can CLOSE and then OPEN INPUT again to rewind a sequentially accessed indexed file.

Simple INPUT-OUTPUT Statements for INDEXED Files: Once you open an indexed file, you can use READ or WRITE statements to read or add records to this file. You cannot use a READ statement for a file opened for OUTPUT and similarly you cannot use a WRITE statement for a file you have opened for INPUT.

A typical READ statement for RANDOM access mode looks like:

```
       MOVE "1X3" TO M-IDNO.
       READ MY-FILE.
```

A good programmer must check whether the READ statement could find a record to read successfully. That means; the programmer must check whether there were any records with the indicated key value when the statement was executed. The situation can be checked with the following construct:

```
       READ MY-FILE  INVALID KEY PERFORM  NO-RECORDS-FOUND.
```

The construct tells the COBOL compiler to execute the paragraph labeled NO-RECORDS-FOUND when the READ statement cannot find any record with the key value stored in the key field variable.

Once your program EXECUTES a successful READ statement, the information in the data record that was just brought into memory will be available in the corresponding variables mentioned in your record description declaration (the FD block).

When you want to create an indexed file, you have to open it for OUTPUT. The COBOL statement that is used to put records into a new file is the WRITE statement. The important point that you should be careful with are that,

- The file should be opened
- New values for the field variables of the FD record description must be moved to proper variables of the record
- The value of the KEY VARIABLE must have a valid value and this valid value should be alphanumerically greater than the key value of the record written previously,
- The RECORD NAME is specified in the WRITE statement and NOT the FILE NAME.

Updating Records in an Indexed File: Sometimes you need to make some small changes in the contents of a file. Of course it possible to create a brand new file with the new, modified contents but this is not practical. COBOL provides you with an OPEN I-O mode with which you can modify only the required record in a file.

In other words, there is another file opening mode; the I-O mode; and another special REWRITE statement. Suppose that you want to change the surname field of a record in an indexed file (originally "AYFER") into "AYFEROGLU".

The program that you can write could read like

```
     SELECT MYFILE ASSIGN TO DISK "MYFILE.DAT"
                   ORGANIZATION IS INDEXED
                   ACCESS MODE IS RANDOM
                   RECORD KEY IS M-IDNO.
     ....
     FD   MYFILE.
     01   MYFILE-REC.
          02  M-IDNO      PIC XXXX.
          02  M-NAME      PIC X(16).
          02  M-SURNAME   PIC X(16).
          02  M-BIRTHDATE.
              03 M-BD-YEAR  PIC 9999.
              03 M-BD-MONTH PIC 99.
              03 M-BD-DAY   PIC 99.
     ...
     PROCEDURE DIVISION.
         ....
         OPEN I-O MYFILE.
         ....
         MOVE "X20" TO M-IDNO.
         READ MYFILE INVALID KEY PERFORM KEY-ERROR.
    *
    * TO MAKE SURE THAT WE ARE ON THE CORRECT RECORD
    *
         IF M-SURNAME NOT = "AYFER" DISPLAY "WRONG RECORD"
                                    DISPLAY"RECORD NOT UPDATED"
                                    CLOSE MYFILE
                                    STOP RUN.
         MOVE "AYFEROGLU" TO M-SURNAME.
         REWRITE MYFILE-REC.
         ....
         CLOSE MYFILE.
         .....
```

Important: The key field of a record in an indexed file CANNOT be updated. If the programmer has to update the key field, the only solution is to delete the record with the old key value and a add the same record to the file; this time with the new key value.

Adding Records to an Indexed File: When you need to add records to an indexed file, you should open the file in I-O mode and just write new records.

Note: Keys in an indexed file must be unique. That is to say, no two records can have the same key. There is, however, a technique to have duplicate keys in an indexed file, but this will be covered later.

Deleting Records of an Indexed File:

- If you do not want a specific record to be kept in an indexed file any more, you can use the DELETE statement to make the record inaccessible.
- The DELETE statement is used more or less like the WRITE statement. You must move a correct key value to the record key field and issue the DELETE statement.

```
MOVE "X12" TO M-IDNO.
DELETE MYFILE-REC.
```

- Please note that, when you delete a record, the physical record is NOT deleted. Only the record is rendered inaccessible. Suppose, for instance that an indexed file is 20 Mbytes in total length and you delete half of the records in this file. After all the deletions are completed, you shall see that the file is still 20 Mbytes in length. The only way you can recover the disk space used by deleted records is REORGANIZING the indexed file. (See section "REORGANIZING INDEXED FILES").

Advantages of Indexed Files:

- Quite easy to process,
- With proper selection of a key field, records in a large file can be searched and accessed in very quickly.
- Any field of the records can be used as the key. The key field can be numerical or alphanumerical.

Disadvantages of Indexed Files:

- Extra data structures have to be maintained (the COBOL run-time modules take care of these and it is not the programmers' concern). These extra data structures maintained on the disk can use up much disk space, especially for long key values.
- The indexed files have to be reorganized from time to get rid of deleted records and improve performance that gets gradually decreased with addition of new records.

Advanced Features of Indexed Sequential Files:

- **Reorganizing Indexed Files:** This operation is usually done by a utility program supplied by the manufacturer of the COBOL compiler that you use. Refer to the manuals of your compiler for details and instructions.
- **Multiple Keys (Primary & Alternate Keys): S**ometimes the programmers need to assign more than one key fields in their records. An example could be a database where one wants to keep records for vehicles registered to a certain country. The Turkish vehicle plate numbering system is NOT suitable for using relative files

because it contains alphabetic parts (06 AHT 08). As you might guess, this type of key values is very suitable to be organized as an INDEXED file.

Having declared the "PLATE" field as the key field, now, given a plate number, a program which has declared an indexed file with RANDOM access mode, can find the corresponding record very quickly (if the does exist, this condition will also be detected very quickly).

Knowing that no two vehicles have the same plate number (unique), the plate number is certainly a very good choice for the key.

However, suppose that the program you have to write is also expected to find records for vehicles of a certain brand very quickly. Since the key is the plate number, the only way you can find "OPEL" brand vehicles by going through all the records (declare the access mode to be sequential and READ NEXT all records till the AT END condition raises) testing the BRAND field against the value given by the coder ("OPEL" for instance).

With indexed files, there is a better solution: You can declare the BRAND field also to be a key; but AN ALTERNATE key!. The PRIMARY key will still be the plate numbers. Since there are many vehicles manufactured by OPEL in your database, you know that the ALTERNATE KEY values will not be unique. THIS IS ALLOWED IN COBOL. ONLY THE PRIMARY KEY HAS TO BE UNIQE. ALTERNATE KEYS CAN HAVE DUPLICATE VALUES.

You can declare as many alternate keys your program requires. More alternate keys you have, more extra data structures will be required and slower be your program. Therefore certain attention should be paid before assigning alternate keys. You should avoid unnecessary alternate keys.

When you have alternate keys, you declare them in the SELECT statement together with the PRIMARY key.

```
SELECT VEHICLES ASSIGN TO VEHICLES
          ORGANIZATION IS INDEXED
          ACCESS MODE IS RANDOM
          RECORD KEY IS PLATE
          ALTERNATE KEY IS BRAND.
....
FD   VEHICLES.
01   VEHICLE-REC.
     02 PLATE           PIC XXXXXXXXXX.
     02 BRAND           PIC X(20).
     02 OWNER           PIC X(32).
     02 DATE-REGISTERED PIC 99999999.
     02 ENGINE-NO       PIC X(20).
     02 CHASSIS-NO      PIC X(20).
     ...
```

When you need to access the file with the primary key, you just move the key value to the primary key field of the record and issue a READ statement.

```
MOVE "06 AHT 08" TO PLATE.
READ VEHICLES INVALID KEY PERFORM NOT-FOUND.
```

When you need to access the file using an alternate key, you move an appropriate value to the alternate key you want to use and issue a READ statement in which you specify which key you want to use

```
MOVE "CHEVROLET" TO BRAND.
READ VEHICLES KEY BRAND INVALID KEY PERFORM NOT-FOUND.
```

The START Statement: Sometimes, it is necessary to find a set of records for which you want to specify a criterion. For example, in our "vehicles" example, you might want to find the records of "TOYOTA" brand vehicles. If the criterion is related to one of the keys, you will not have to go through the whole file, testing the fields against a certain value. Instead, you can use the START statement to find the first vehicle with "TOYOTA" recorded as the brand.

An example is:

```
              START VEHICLES KEY BRAND IS EQUAL TO "TOYOTA"
                             INVALID KEY NO-SUCH-BRAND.
          LOOP1.
              READ VEHICLES NEXT AT END PERFORM END-OF-BRAND.
              ...
              GO LOOP1.
              ...
```

Important: Please note that when you use the START statement, the file is accessed as a RANDOM file; whereas the subsequent READ NEXT statements access the file in SEQUENTIAL mode. This dilemma is solved by the ACCESS MODE IS DYNAMIC phrase in the SELECT statement. When you declare the access mode to be DYNAMIC, you can access the file both sequentially the NEXT option of the READ statement) and randomly by specifying a certain key value. When you want to use the START statement you must open an index file either as INPUT or I-O. The START statement is not allowed and in fact not meaningful when the file is opened for OUTPUT.

Sorting Files in COBOL

Line sequential and Sequential files are used very frequently in data processing applications.

The records in these files are usually needed to be put in ascending or descending order for proper and easy handling. (Remember that searching records in an ordered file (a file sorted in respect to the field which is the search key; you can stop reading records when you find a record which has the key field having greater value than the value you are searching for).

Sorting a set of data which will fit into arrays (therefore fitting into the memory) is quite easy. (Remember the Internal Sort techniques you have learned in previous courses). Sorting large files which will not fit into arrays in memory, on the other hand, is not easy. This requires special algorithms to be implemented and generally called "EXTERNAL SORT TECHNIQUES".

Many COBOL compilers provide facilities to solve this "External Sort" problem. RM-COBOL, for instance, has a very powerful SORT statement which will let the user SORT and MERGE sequential files very easily.

SORTING does not need much explanation here. The MERGE process however, needs a little bit explanation.

MERGING DATA is simply combining two or more SORTED sequential files together into a single file so that the resulting file is also sorted. We shall not cover the MERGE facility of the COBOL SORT statement in this course.

The RM-COBOL SORT Statement:

The general syntax of the RM-COBOL SORT statement is:

```
SORT sort-work-file
ON {ASCENDING,DESCENDING} KEY data-name1, data-name2, ...
ON {ASCENDING,DESCENDING} KEY data-name1, data-name2, ...
                     USING file-to-be-sorted
                     GIVING sorted-file
```

In order to use this SORT statement, you will have to declare 3 files;

- ❑ The WORK FILE for the SORTING Process
- ❑ The file to be sorted (input file)
- ❑ The file which will contain the sorted records (output file)

The WORK FILE is not an actual file; it is a special declarative file structure with which you tell the compiler that you will perform an external sort on a file and also indicate the fields on which you shall set the sort criteria. The concept will become clearer with an example:

Suppose you have sequential file with personnel records (PERSONEL FILE) and you want to sort this file on the persons' names and surnames. In order to use the SORT statement, you must declare a SORT WORK file which will declare the structure of the records to be sorted and indicate the lengths and positions of the fields which contain the names and surnames:

Suppose that the structure of the PERSONEL file is:

```
       01  PERS-REC.
           02 ID-NO              PIC X(8).
           02 NAME               PIC X(16).
           02 SURNAME            PIC X(16).
           02 GENDER             PIC X.
           02 DEPT-CODE          PIC X.
           02 NCHILDREN          PIC 99.
           02 HOME-ADR1          PIC X(25).
           02 HOME-ADR2          PIC X(25).
           02 HOME-ADR3          PIC X(25).
           02 HOME-TEL           PIC X(12).
           02 EMPLOYMENT-DATE.
              03 R-DAY             PIC 99.
              03 R-MONTH           PIC 99.
              03 R-YEAR            PIC 9999.
           02 LEAVE-DATE.
              03 R-DAY             PIC 99.
              03 R-MONTH           PIC 99.
              03 R-YEAR            PIC 9999.
           02 LEAVE-REASON       PIC X.
```

The corresponding SORT WORK file declaration might look something like:

```
       SD  SORT-WORK-FILE.
       01  PERS-REC.
           02 ID-NO              PIC X(8).
           02 NAME               PIC X(16).
           02 SURNAME            PIC X(16).
           02 FILLER             PIC X(107).
```

This declaration will enable the user to issue a SORT statement using the NAME and SURNAME fields; into either ascending or descending order.

Please note that 108 byte filler is used in the SORT-WORK-FILE record description so that the work file's record length matches the record length of the PERSONEL file. Please also note the "SD" indicator used in place of "FD" in FILE SECTION.

You can perform sort on more than one field at time; and also you can sort into ascending order (increasing) on one field, and into descending order (decreasing) on another. For example, while sorting the PERSONEL file, you can declare the NAME field to be the primary sort field and the SURNAME to be the secondary sort field; so that records with identical names will be sorted into surnames among themselves. (Just like the entries in a telephone directory).

Referring to our PERSONEL file example, typical file declarations and a SORT statement would look like:

In the ENVIRONMENT DIVISION:

```
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT SORT-FILE ASSIGN TO DISK SORTWORK.
           SELECT UNSORTED-PERSON ASSIGN TO PERSONEL
                  ORGANIZATION IS LINE SEQUENTIAL.
           SELECT SORTED-PERSON   ASSIGN TO SPERSONL
                  ORGANIZATION IS LINE SEQUENTIAL.
```

In the DATA DIVISION:

```
       FILE SECTION.
       SD  SORT-FILE.
       01  SORT-RECORD.
           02  FILLER                     PIC X(8).
           02  S-NAME                     PIC X(16).
           02  S-SURNAME                  PIC X(16).
           02  FILLER                     PIC X(108).

       FD  UNSORTED-PERSON.
       01  FILLER                         PIC X(148).

       FD  SORTED-PERSON.
       01  FILLER                         PIC X(148).
```

In the PROCEDURE DIVISION:

```
           SORT SORT-FILE
                   ON ASCENDING KEY S-NAME S-SURNAME
                   ON DESCENDING S-EMP-DATE
                   USING UNSORTED-PERSON
                   GIVING SORTED-PERSON.
```

Notes:

- The lengths of records in the work, input, and output files should match.
- You do not need to declare the details of input and output file records if you do not need these details in your program.
- You should not OPEN or CLOSE the work, input and output files before, after or during the SORT operation.
- You can specify more than one sort field and the fields may have different ordering (ascending/descending).

A complete RM-COBOL program which sorts the PERSONEL file, creating the sorted file SPERSONL.DAT might look like:

Please study the SORT statement and its relevant file declarations carefully!

```
IDENTIFICATION DIVISION.
    PROGRAM-ID.  "SORT DEMO".

    ENVIRONMENT DIVISION.
    CONFIGURATION SECTION.
    SOURCE-COMPUTER.  RMCOBOL-85.
    OBJECT-COMPUTER.  RMCOBOL-85.

    INPUT-OUTPUT SECTION.
    FILE-CONTROL.
        SELECT SORT-FILE ASSIGN TO SORTWORK.
        SELECT UNSORTED-PERSON ASSIGN TO PERSONEL
               ORGANIZATION IS LINE SEQUENTIAL.
        SELECT SORTED-PERSON   ASSIGN TO SPERSONL
               ORGANIZATION IS LINE SEQUENTIAL.
   *
    DATA DIVISION.
    FILE SECTION.
    SD  SORT-FILE.
    01  SORT-RECORD.
        02  S-ID-NO                       PIC X(8).
        02  S-NAME                        PIC X(16).
        02  S-SURNAME                     PIC X(16).
        02  S-GENDER                      PIC X.
        02  S-DEPT-CODE                   PIC 9.
        02  FILLER                        PIC X(89).
        02  S-EMP-DATE.
            03 S-EMP-DATE-DD              PIC 99.
            03 S-EMP-DATE-MM              PIC 99.
            03 S-EMP-DATE-YY              PIC 9999.
        02  S-LEAVE-DATE.
            03 S-LEAVE-DATE-DD            PIC 99.
            03 S-LEAVE-DATE-MM            PIC 99.
            03 S-LEAVE-DATE-YY            PIC 9999.
        02  S-LEAVE-REASON                PIC X.



    FD  UNSORTED-PERSON.
    01  FILLER                            PIC X(148).

    FD  SORTED-PERSON.
    01  FILLER                            PIC X(148).
   *
    PROCEDURE DIVISION.
```

```
       MAIN-PGM.

           SORT SORT-FILE
                   ON ASCENDING KEY S-NAME S-SURNAME
                   ON DESCENDING S-EMP-DATE
                   USING UNSORTED-PERSON
                   GIVING SORTED-PERSON.

           STOP RUN.
```

File operations

This section Covers File Handling, Input- output Statements for file, Sort and Merge of records.

OPEN

```
     { INPUT {file-name-1} ...   }
OPEN { OUTPUT {file-name-2} ...  } ...
     { I-O {file-name-3} ...     }
     { EXTEND {file-name-4} ...  }
```

- Opens the file for processing
- OUTPUT clears the file of its existing records

Open - File position indicator:

- The successful execution of OPEN INPUT or OPEN I-O statement sets the file position indicator to:
  - 1 for non-empty QSAM file
  - For VSAM sequential and indexed files, to the characters with the lowest ordinal position in the collating sequence associated with the file.
  - For VSAM relative files, to 1.

OPEN - Extend Rules:

- When the EXTEND is specified the file positioned immediately after the last record written in the file:
  - For ESDS or RRDS file the added records are placed after the last existing records
  - For KSDS your add must have a record key higher than the highest record in the file

CLOSE

```
                           { REEL } [ WITH NO REWIND ]
CLOSE { file-name-1  [     { UNIT } [ FOR REMOVAL    ]       ] } ...
                                    { NO REWIND }
                       WITH         { LOCK      }
```

- Closes the open file
- Terminates the file processing
- COBOL performs automatic closing if not closed
    - At termination of run-unit
    - CANCEL command
    - Return from program with INITIAL attribute

READ

Format-1 - Sequential Read:

```
READ file-name-1 [ NEXT     ] RECORD [{| WITH [NO] LOCK         |}]
                 [ PREVIOUS ]          | INTO identifier-1      |
```

```
       [AT END imperative-statement-1]
       [NOT AT END imperative-statement-2]
       [END-READ]
```

- NEXT phrase is optional for SEQUENTIAL access mode & a must for DYNAMIC
- When the READ NEXT statement is the first statement to be executed after the OPEN statement on the file, the next record is the first record in the file

USAGE:

- The sequential read is used when the exact key value is not known and if we want to search the VSAM file for a particular key for processing.
- This sequential read is also used when Alternate key is used as key to the indexed file. This is because the alternate key may not be unique.
- When we need to process records with part of the key matching, we use sequential read. For E.G, The key field of the EMP-MATER file is Emp-no, Designation., date-of-joining, if we want to process all employees belonging to designation “Associates”, we use sequential read of the indexed file

Format-2 - Random read

```
READ file-name-1 RECORD [{| WITH [NO] LOCK   |}] [KEY IS {data-name-1}]
                          | INTO identifier-1 |
```

```
[INVALID KEY imperative-statement-1]
[NOT INVALID KEY imperative-statement-2]
[END-READ]
```

- Data-name-1 is either a RECORD KEY or ALTERNATE RECORD KEY of the Indexed file
- The result of the execution of a READ statement with the INTO phrase is equivalent to the application of the following rules in the order specified:
- The execution of the same READ statement without the INTO phrase.
- The current record is moved from the record area to the area specified by identifier-1 according to the rules for the MOVE statement without the CORRESPONDING phrase. The size of the current record is determined by rules specified for the RECORD clause. If the file description entry contains a RECORD IS VARYING clause, the implied move is a group move. The implied MOVE statement does not occur if the execution of the READ statement was unsuccessful. Any subscripting or reference modification associated with identifier-1 is evaluated after the record has been read and immediately before it is moved to the data item. The record is available in both the record area and the data item referenced by identifier-1.
- This format of READ is used when the ACCESS mode is either random or dynamic

USAGE: Te random reading of indexed file is used when the key value for the file is exactly known. The random read can be used in real time systems like Patient monitoring system in hospitals. Here there might be necessary to retrieve a particular patient's record. The full key value for the Patient master file is known. Hence we use random read in this case. The random read of indexed file is faster.

READ Random - Indexed file:

- Key of reference**:**
  - Data name specified in KEY phrase
  - The prime RECORD KEY data name if KEY phrase not specified
- The KEY IS phrase can be specified only for indexed files. Data-name-1 must identify a record key associated with file-name-1
- When dynamic access is specified, this key of reference is used for subsequent executions of sequential READ
- Execution of a Format 2 READ statement causes the value of the key of reference to be compared with the value of the corresponding key data item in the file records, until the first record having an equal value is found. The file position indicator is positioned to this record, which is then made available. If no record can be so identified, an INVALID KEY condition exists, and READ statement execution is unsuccessful.
- If the KEY phrase is not specified, the prime RECORD KEY becomes the key of reference for this request. When dynamic access is specified, the prime RECORD KEY is also used as the key of reference for subsequent executions of sequential READ statements, until a different key of reference is established.
- **KEY Phrase**: When the KEY phrase is specified, data-name becomes the key of reference for this request. When dynamic access is specified, this key of reference is used for subsequent executions of sequential READ statements, until a different key of reference is established.

READ Random - Relative file:

- Execution of a Format 2 READ statement reads the record whose relative record number is contained in the RELATIVE KEY data item,
- The KEY phrase must not be specified

Multiple Record Processing

If more than one record description entry is associated with file-name-1 in the File Description, these records automatically share the same storage area; that is, they are implicitly redefined. After a READ statement is executed, only those data items within the range of the current record are replaced; data items stored beyond that range are undefined. The below example illustrates this concept. If the range of the current record exceeds the record description entries for file-name-1, the record is truncated on the right to the maximum size. In either of these cases, the READ statement is successful and an I-O status (04) is set indicating a record length

Conflict has occurred.

Example:

```
The FD entry is:

FD INPUT-FILE LABEL RECORDS OMITTED.
01  RECORD-1  PICTURE X(30).
01  RECORD-2  PICTURE X(20).
```

Contents of RECORD-1 when READ statement is executed:
ABCDEFGHIJKLMNOPQRSTUVWXYZ1234

When the next record from the file is read into RECORD-2,
Contents of record being read in RECORD-2:

01234567890123456789

But since the record description contains two 01 levels and it’s a variable length file tents of record area after second READ are executed:

01234567890123456789??????????
The last 10 characters of the record area are undefined. I.e. This storage area can hold any characters depending on the sequence of reads and any process done inside the program using RECORD-1. In the above example, assume there was no processing then the last 10 bytes would contain “UVWXYZ1234”. But this area cannot be accessed, when we use RECORD-2 layout because this is implicitly redefined item

More on File Position Indicators:

- The file position indicator is a conceptual entity used to facilitate exact specification of the next record to be accessed within a given file during certain sequences of input-output operations. Only the OPEN, CLOSE, READ and START statements affect the setting of the file position indicator. The concept of a file position indicator has no meaning for a file opened in the output or extends mode.
- Indicates the next record to be accessed for sequential COBOL requests
- You do not specify them anywhere in your program
- It is set by successful:
- OPEN, START, READ, and READ NEXT statements
- The successful execution of an OPEN INPUT or OPEN I-O statement for sequential files sets the file position indicator to 1.
- Execution of an OPEN INPUT or OPEN I-O statement sets the file position indicator:
- For VSAM sequential and indexed files, to the characters with the lowest ordinal position in the collating sequence associated with the file.
- For VSAM relative files, to 1.
- Subsequent READ or READ NEXT requests then uses & updates it
- It is not used or affected by the output statements:
- WRITE, REWRITE, or DELETE.
- If the record is no longer accessible, i.e. it has been deleted, the file position indicator is updated to point to the next existing record in the file
- The file position indicator has no meaning for random processing

READ – Example:

```
File- Indexed, Dynamic Access
Record Description:
01 ASSOC-REC
    05 ASSOC-NO-KEY PIC 9(4).
    05 ASSOC-NAME     PIC X(10).
Assoc #     Assoc Name
0004  RAJESH
0010  RAMESH
0015  GOPAL
0016  RAGHAVAN
0100  SARITA
0401  RANI
```

File position indicator example:

Operation	File Position Pointer	Record Area
OPEN input	0004	Not defined
READ next	0010	0004 Rajesh
READ Key 0100	0401	0100 Sarita
READ next	end of file	0401 Rani
READ key 0015	0016	0015 Gopal
READ key 0090	Undefined	Undefined (Invalid key)
READ next	Undefined	Undefined

START

- Enables the positioning of the pointer at a specific point in an indexed or relative file
- File should be opened in Input or I-O mode
- Access mode must be Sequential or Dynamic
- Does not read the record

```
START file-name
[ KEY IS  { EQUAL TO          data-name]
                    =
                   GREATER THAN
                    >
                   NOT LESS THAN
                   NOT < THAN }
[ INVALID KEY imperative-stmt-1 ]
[ NOT INVALID KEY imperative-stmt-2 ]
[ END-START ]
```

START – Rules:

- Invalid Key arises if the record position is empty
- When KEY phrase is not specified KEY EQUAL TO primary key is implied
- File position indicator points to the first record in the file whose key field satisfies the comparison
- When the KEY phrase is specified, the file position indicator is positioned at the logical record in the file whose key field satisfies the comparison.
- When the KEY phrase is not specified, KEY IS EQUAL (to the prime record key) is implied.
- When the START statement is executed, a comparison is made between the current value in the key data-name and the corresponding key field in the file's index.
- PROGRAM COLLATING SEQUENCE clause, if specified, has no effect
- The file position indicator points to the first record in the file whose key field satisfies the comparison. If the operands in the comparison are of unequal lengths, the comparison proceeds as if the longer field were truncated on the right to the length of the shorter field. All other numeric and nonnumeric comparison rules apply

START

File- Indexed, Dynamic Access

Record Description:

```
01 ASSOC-REC
   05 ASSOC-NO-KEY   PIC 9(04).
   05 ASSOC-NAME     PIC X(10).

 Assoc #   Assoc Name
 0004      RAJESH
 0010      RAMESH
 0015      GOPAL
 0016      RAGHAVAN
 0100      SARITA
 0401      RANI
```

OPERATION	POINTER VALUE	REC-AREA
OPEN I-O	TOP OF FILE	NOT DEFINED
START ASSOC.NO-KEY = 0004	0004	NOT DEFINED
READ NEXT	0010	0004 RAJESH
START ASSOC-NO-KEY > 0016	0100	0004 RAJESH
READ NEXT	0401	0100 SARITA
START ASSOC-NO-KEY > 0401	UNKNOWN	0100 SARITA

WRITE

```
WRITE record-name-1 [ FROM { identifier-1 } ]
                           { literal-1    }

                            { { identifier-2 } [ LINE  ] }
[ { BEFORE } ADVANCING      { { integer-1    } [ LINES ] } ]
  { AFTER  }                {                            }
                            { { mnemonic-name-1 }        }
                            { { PAGE            }        }

[AT { END-OF-PAGE } imperative-statement-1]
    { EOP         }

[NOT AT { END-OF-PAGE } imperative-statement-1]
        { EOP         }

[END-WRITE]
```

Note:

- Releases a record onto the output file
- ADVANCING phrase is only for Printer Files

When this phrase is specified, the following rules apply:

- When BEFORE ADVANCING is specified, the line is printed before the page is advanced.
- When AFTER ADVANCING is specified, the page is advanced before the line is printed.
- When identifier-2 is specified, the page is advanced the number of lines equal to the current value in identifier-2. Identifier-2 must name an elementary integer data item.
  - File Position is not affected by this statement
  - When END-OF-PAGE is specified, and the logical end of the printed page is reached during execution of the WRITE statement, the END-OF-PAGE imperative-statement is executed

WRITE for Seq. Files: From its Record Area defined at 01 Level

```
       WRITE ASSOC-REC
```

First Copy into File Area and then Write

```
WRITE ASSOC-REC FROM COPY-REC
```

Note that File Name is not mentioned in the Write Statement:

- Facilitates Multiple-Record Type Files also known as Non-Homogeneous Files.
- The various 01 levels defined for the same files implicitly redefine each other. That is they share the same record area. Therefore, in case of Non-Homogeneous Files the Record Layouts must contain a Record Type Field for identification.
- The WRITE statement can only be executed for a sequential file opened in OUTPUT or EXTEND mode for a sequential file.

WRITE for Printer Files:

- Facilities for Paper Movement are provided:
  - Print and Advance
  - Advance and Print
  - Position to New Page
  - Advance by given number of lines
  - End of Page Logic (FD entry for this file must contain a LINAGE clause)
- When ADVANCING option is used:
  - First char of record reserved for printer control
  - If compile has ADV the LRECL of file should be one more than FD entry record area length
  - If compile option is NOADV then LRECL of file is same as FD entry record area length
  - Leave the 1st char for printer control

WRITE for Disk Files:

```
WRITE record-name-1 [ FROM { identifier-1 } ]
                           { literal-1    }
      [INVALID KEY imperative-statement-1]
      [NOT INVALID KEY imperative-statement-2]
      [END-WRITE]
```

WRITE - Invalid key:

- ❑ Not Allowed for ESDS
- ❑ Attempt to write beyond file boundary
- ❑ Record with specified key already present (KSDS, RRDS)
- ❑ **FROM phrase:** The result of the execution of the WRITE statement with the FROM identifier-1 phrase is equivalent to the execution of the following statements in the order specified.

```
MOVE identifier-1 TO record-name-1.
      WRITE record-name-1.
```

WRITE - Rules for VSAM:

- ❑ **When access is sequential:** or KSDS, records must be released in ascending order of RECORD KEY values
- ❑ **Else INVALID KEY condition will be raised:** For RRDS system returns the RELATIVE KEY if clause is specified in SELECT stmt
- ❑ When access is random or dynamic:
  - o For KSDS populate RECORD key data item and ALTERNATE key data item (if any)
  - o For RRDS populate the RELATIVE key data item
- ❑ When an attempt is made to write beyond the externally defined boundaries of the file, the execution of the WRITE statement is unsuccessful, and an EXCEPTION/ERROR condition exists. The contents of record-name are unaffected. If specified, the status key is updated, and, if an explicit or implicit EXCEPTION/ERROR procedure is specified for the file, the procedure is then executed; if no such procedure is specified, the results are unpredictable.
- ❑ Before the WRITE statement is executed, you must set the prime record key (the RECORD KEY data item, as defined in the File-Control entry) to the desired value.
- ❑ If the ALTERNATE RECORD KEY clause is also specified in the File-Control entry, each alternate record key must be unique, unless the DUPLICATES phrase is specified. If the DUPLICATES phrase is specified, alternate record key values may not be unique. In this case, the system stores the records so that later sequential access to the records allows retrieval in the same order in which they were stored.
- ❑ When ACCESS IS SEQUENTIAL is specified in the File-Control entry, records must be released in ascending order of RECORD KEY values.

REWRITE

Syntax – Format1:

```
REWRITE record-name-1 [ FROM { identifier-1 } ]
                               literal-1
```

Syntax – Format2:

```
REWRITE record-name-1 [ FROM { identifier-1 } ]
                               literal-1
 [INVALID KEY imperative-statement-1]
 [NOT INVALID KEY imperative-statement-2]
 [END-WRITE]
```

Description: The REWRITE statement is used to replace records in a file. The record is replaced within the file associated with record-name-1. If the FROM clause is included, the FROM item is copied into record-name-1 before the data record is rewritten. If the record was locked during the READ, the REWRITE will unlock the record automatically. Each of the two formats works slightly differently, and is described below in its correspondingly numbered item.

- The first format performs a sequential rewrite of the file. The last record read is replaced by the contents of record-name-1.
- The second format performs a random rewrite of the file. The last record read, will be rewritten. If the record key has been changed, no record is replaced and the code between INVALID KEY and NOT INVALID KEY, if any, will be executed. If the rewrite is successful, the code between NOT INVALID KEY and END-REWRITE, if any, will be executed.

Tips:

- If there is a file status associated with the file, it will be set after the REWRITE is executed to indicate the result of the REWRITE. A value of "00" means successful and "23" means the record key was modified prior to the execution of the REWRITE.
- Any file mentioned must be the subject of a SELECT statement and a file description (FD). Files must be open for input/output and read prior to executing the REWRITE statement.
- When used in a program with a SORT statement, the sort file itself can never be rewritten.
- Be sure to READ the record you wish to update and change the desired fields before executing the REWRITE statement.

REWRITE - Invalid Key:

- When access mode is sequential, and the value of RECORD KEY of the record to be replaced not = RECORD KEY data item of the last-retrieved record
- Value contained in RECORD KEY not = any record in file
- Duplicate ALTERNATE RECORD KEY when duplicates are not allowed

REWRITE – Rules:

- File should be opened in I-O
- After REWRITE record is not available in the rec area
- After a REWRITE statement with the FROM phrase is executed, the information is still available in identifier-1

- ❑ File position ind not affected
- ❑ For Sequential files
  - o INVALID KEY not allowed
  - o Record length can't change
- ❑ For random or dynamic access the record to be rewritten need not be read first

DELETE

Syntax – Format1:

```
DELETE file-name-1 RECORD
[INVALID KEY imperative-statement-1]
[NOT INVALID KEY imperative-statement-2]
[END-DELETE]
```

Description:

- ❑ Removes a record from an indexed or relative file
- ❑ For indexed files, the key can then be reused for record addition.
- ❑ For relative files, the space is then available for a new record with the same RELATIVE KEY value.
- ❑ If the FILE STATUS clause is specified in the File-Control entry, the associated status key is updated when the DELETE statement is executed.
- ❑ The file position indicator is not affected by execution of the DELETE statement.
- ❑ For a file in sequential access mode, the last input/output statement must have been a successfully executed READ statement. When the DELETE statement is executed, the system removes the record retrieved by that READ statement.
- ❑ For a file in sequential access mode, the INVALID KEY and NOT INVALID KEY phrases must not be specified. However, an EXCEPTIONERROR procedure can be specified.
- ❑ When the DELETE statement is executed, the system removes the record identified by the contents of the prime RECORD KEY data item for VSAM indexed files, or the RELATIVE KEY data item for VSAM relative files. If the file does not contain such a record, an INVALID KEY condition exists.

DELETE – Rules:

- ❑ File should opened for I-O
- ❑ In sequential access mode, the last input/output statement must have been a successfully executed READ
  - o INVALID key not allowed
- ❑ For Random or Dynamic access system removes record pointed by the RECORD key (KSDS) or RELATIVE key(RRDS)

SORT

- While processing sequential files, it is sometimes necessary that the records should appear in some predetermined sequence. The process of sequencing records in file in some predetermined order on some fields is called SORTING
- The fields based on which the records are sequenced are called SORT KEYS.
- The sequencing can be ascending or descending order of the KEY.

```
SORT file-name-1 {ON { ASCENDING  } KEY {data-name-1}...}...
                     { DESCENDING }

[WITH DUPLICATES IN ORDER] [COLLATING SEQUENCE IS alphabet-name-1]

{ [INPUT PROCEDURE IS procedure-    { THROUGH } procedure-name- }
{ name-1 [                          { THRU    } 2]              }
{ [USING {file-name-2} ...                                      }

{ [OUTPUT PROCEDURE IS procedure-   { THROUGH } procedure-name- }
{ name-3 [                          { THRU    } 4]              }
{ [GIVING {file-name-3} ...                                     }
```

Description: The SORT statement is used to sort a data file. The keys to be sorted on must be in the sort file definition that is in the record associated with the SD (sort description). Two options are available for the input phase of the sort operation as well as two options for the output phase of the operation:

- If the USING phrase is specified, the records to be sorted are copied from the USING file into the SORT file before the data is sorted.
- If the INPUT PROCEDURE phrase is specified, the records to be sorted are entered into the sort file using the RELEASE statement.
- If the GIVING phrase is specified, the records to be sorted are copied in appropriate ASCENDING/DESCENDING KEY order into the GIVING file.
- If the OUTPUT PROCEDURE phrase is specified, the records to be sorted are copied in the appropriate ASCENDING/DESCENDING KEY order into the sort file. Records can be obtained from the file using the RETURN statement.

Tips:

- All of the files referenced must be closed when the SORT statement is executed.
- The key fields must all be defined in the record description(s) for the sort file.
- The sort file cannot be opened or closed.
- If the USING option is used, the USING file must have the same record layout as the sort file.
- If the GIVING option is used, the GIVING file must have the same record layout as the sort file.

Example:

Assume the following record description for the input file

```
FD    EMP-FILE.
EMP-REC.
02    ID-NUM            PIC 9(6).
02    NAME              PIC X(24).
02    DEPARTMENT        PIC X(10).
02    BASIC-PAY         PIC 9(5)V99.
02    ALLOWANCE         PIC 9(4)V99.
02    DEDUCTION         PIC 9(4)V99.
```

We want to sort this file on DEPARTMENT on ascending sequence and then within each

DEPARTMENT, arrange BASIC-PAY on descending sequence. This means that the records with same DEPARTMENT value are to be arranged from highest to lowest value of the BASIC-PAY. The work file and output files are SORT-FILE and OUTPUT-FILE. The FD and SD entries for these files are as follows:

```
SD    SORT-FILE
01    SORT-REC.
      02    FILLER                  PIC X(30).
      02    DEPARTMENT              PIC X(10).
      02    BASIC-PAY               PIC 9(5)V99.
      02    FILLER                  PIC X(12).
FD    OUTPUT-FILE
OUT-REC                             PIC X(59).
```

The sort statement for this is:

```
SORT  SORT-FILE ON ASCENDING KEY DEPARTMENT
      DESCENDING KEY BASIC-PAY
      USING EMP-FILE GIVING OUTPUT-FILE.
```

Sometimes it is necessary to edit records before sorting or after sorting before writing it to the output file. Editing in this context can mean selection of appropriate records from the input file for sorting.

For example, a file with variable length records will require some padding with some characters to make it fixed length for the purpose of sorting.

The sort verb allows user to specify the INPUT & OUTPUT PROCEDURE in the sort statement, which perform the necessary processing of the records.

The input and output procedures are nothing but sections placed outside the SORT statement.

INPUT PROCEDURE: The sort statement before the start of the sorting process implicitly performs the specified input procedure. These procedures reads the records from input file, performs the necessary processing, and then release the records to Sorting operation by RELEASE statement which is discussed later in this chapter

OUTPUT PROCEDURE: The output procedure is performed implicitly by sort statement in order to perform editing on the sorted records. The output procedure gets the sorted records by means of RETURN statement that is discussed later in this session.

```
SORT file-1 { ON
     {ASCENDING
      DESCENDING}       KEY id-1... }  ...
     [WITH DUPLICATES IN ORDER]
     [COLLATING SEQUENCE IS   alphabet-name-1]
     {USING file-2
       INPUT PROCEDURE IS
             proc-nm-1 [THRU proc-nm-2]}
     {GIVING file-3
       OUTPUT PROCEDURE IS
            proc-nm-3 [THRU proc-nm-4]}
```

```
SORT sorted-emp-file
      ASCENDING   s-last-nm
                  s-first-nm
      DESCENDING s-emp-nbr
      USING emp-file
      OUTPUT PROCEDURE
            write-emp-list THRU
            write-emp-list-exit

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT EMP-FILE ASSIGN TO EMPMSTR.
SELECT SORTED-EMP-FILE ASSIGN TO EMPSORT.
SELECT EMP-LIST ASSIGN TO REPORT.
DATA DIVISION.
FILE SECTION.

FD EMP-FILE
      LABEL RECORDS ARE OMITTED.
01 EMP-REC.
     05 EMP-NBR   PIC 9(4).
     05 LAST-NM   PIC X(20).
     05 FIRST-NM  PIC X(20).
     05 MID-NM    PIC X(20).

SD SORTED-EMP-FILE
LABEL RECORDS ARE OMITTED.
```

```
01 SORT-EMP-REC.
     05 S-EMP-NBR       PIC 9(4).
     05 S-LAST-NM       PIC X(20).
     05 S-FIRST-NM      PIC X(20).
     05 S-MID-NM  PIC X(20).

PROCEDURE DIVISION.
.....
     SORT sorted-emp-file
      ASCENDING   s-last-nm s-first-nm
      DESCENDING s-emp-nbr
      USING emp-file
      OUTPUT PROCEDURE write-emp-list THRU
                                write-emp-list-exit
...
WRITE-EMP-LIST.
...
WRITE-EMP-LIST-EXIT.
     EXIT.
```

Collating Sequences:

- COLLATING SEQUENCE option of the SORT statement that is defined in SPECIAL-NAMES paragraph
- PROGRAM COLLATING SEQUENCE if specified in the Configuration Section
- Default is EBCDIC

RELEASE

- Transfers records to the initial phase of sort operation
- It's like WRITE statement

```
                           ┌ identifier-1 ┐
RELEASE record-name-1 [ FROM {              } ]
                           └ literal-1    ┘
```

Description: The RELEASE statement is used within the context of a SORT INPUT PROCEDURE to release data records to the sort process. The released data is moved into the sort file associated with record-name-1. If the FROM clause is included, the FROM item is copied into record-name-1 before the data record is released.

Tips:

- The RELEASE statement is the equivalent of a WRITE statement, except that is used exclusively for SORT files.
- An INPUT PROCEDURE must contain at least one RELEASE statement.
- The RELEASE statement may only be used within the context of an INPUT PROCEDURE.

- ❑ The RETURN statement is used to read SORT files.

RELEASE Example:

```
MOVE  emp-rec     TO    sort-emp-rec
RELEASE sort-emp-rec

RELEASE sort-emp-rec
              FROM emp-rec
```

RETURN

- ❑ Transfers records from the final phase of a sorting or merging operation to an OUTPUT PROCEDURE.
- ❑ It's like READ statement

```
RETURN file-name-1 RECORD [INTO identifier-1]
[AT END imperative-statement-1]
[NOT AT END imperative-statement-2]
[END-RETURN]
```

RETURN – Rules: If OUTPUT PROCEDURE is used, at-least one RETURN stmt must be executed

RETURN - Examples

```
read-sorted-rec.
    RETURN sorted-emp-file
      AT END
      SET c-sort-eof TO TRUE
     END-RETURN
      .
read-sorted-rec-exit.
     EXIT.
```

```
WRITE-EMP-LIST.
      SET c-not-sort-eof TO TRUE
      PERFORM read-sorted-rec
      THRU  read-sorted-rec-exit
      PERFORM  UNTIL c-sort-eof
      PERFORM  string-emp-name
             THRU string-emp-name-exit
      MOVE w-emp-name TO rep-name
      MOVE s-emp-nbr  TO rep-emp-nbr
      WRITE  emp-list-rec
      PERFORM read-sorted-rec
```

```
             THRU  read-sorted-rec-exit
      END-PERFORM
      .
WRITE-EMP-LIST-EXIT.
      EXIT.
```

MERGE

Sometimes it becomes necessary to create a new output file from 2 input files. These 2 files needs to be merged and new file needs to be created

For example A Company has its marketing operations divided into 2 zones and for each zone there is file. Each of these files contains zone name, district name, salesman name, product name and amount of sales for a particular product. The files are sorted on product name. We need to merge these 2 files into a single file to get a single transaction file to update the master file.

The MERGE verb is used to merge 2 or more identical files sorted on the same field

Syntax – Format:

```
                      { ASCENDING  }
MERGE file-name-1{ON  { DESCENDING } KEY {data-name-1}...}...

[COLLATING SEQUENCE IS alphabet-name-1] USING file-name-2 {file-name-
3}...

{ [OUTPUT PROCEDURE IS procedure-  { THROUGH } procedure-name- }
{ name-1 [                         { THRU    } 2]              }
{ [GIVING {file-name-4} ...                                    }
```

Description: The MERGE statement is used to merge data files. The records must be in the files to be merged in the same order in which they are entered in the merged file that is in sequential order specified in the ASCENDING/DESCENDING KEY clause. Two options are available for the output of the operation:

- If the GIVING phrase is specified, the records to be merged are copied in appropriate ASCENDING/DESCENDING KEY order into the GIVING file.
- If the OUTPUT PROCEDURE phrase is specified, the records to be merged are copied in the appropriate ASCENDING/DESCENDING KEY order into the merge file. Records can be obtained from the file using the RELEASE statement.

Tips:

- All of the files referenced must be closed when the MERGE statement is executed.
- The key fields must all be defined in the record description(s) for the merge file.
- The merge file cannot be opened or closed.
- If the GIVING option is used, the GIVING file must have the same record layout as the merge file.

MERGE – Examples:

```
MERGE sorted-emp-ph-file
         ASCENDING      s-last-nm
                  s-first-nm
         DESCENDING s-emp-nbr
         USING   emp-ph-file
            emp-addon-ph-file
         GIVING new-emp-ph-file
```

MERGE – Rules:

- ❑ When USING / GIVING option is specified the input / output file(s) to merge must not be open
- ❑ The key used in the MERGE statement cannot be variably located.
- ❑ When the file referenced by filename-1 is merged control passes to first stmt in the OUTPUT PROCEDURE.
- ❑ Collating Sequences:
  - o COLLATING SEQUENCE option of MERGE statement that is defined in SPECIAL-NAMES paragraph
  - o PROGRAM COLLATING SEQUENCE specified in the Configuration Section
  - o Default is EBCDIC

Input/Output Error Handling Techniques

The following are techniques of intercepting and Handling input/output Errors.

- ❑ The end-of-file phrase (AT END)
- ❑ The EXCEPTION/ERROR declarative
- ❑ The FILE STATUS key
- ❑ The INVALID KEY phrase.

THE END-OF-FILE PHRASE (AT END): An end-of-file condition may or may not represent an error. In many designs, reading sequentially to the end of a file is done intentionally, and the AT END condition is expected.

For example, suppose you are processing a file containing transactions in order to update a master file:

```
  PERFORM UNTIL TRANSACTION-EOF = "TRUE"
    READ UPDATE-TRANSACTION-FILE INTO WS-TRANSACTION-RECORD
      AT END
        DISPLAY "END OF TRANSACTION UPDATE FILE REACHED"
        MOVE "TRUE" TO TRANSACTION-EOF
    END READ
     .
     .
     .
  END-PERFORM
```

In some cases, however, the condition will reflect an error. You code the AT END phrase of the READ statement to handle either case, according to your program design.

If you code an AT END phrase, upon end-of-file the phrase is executed. If you do not code an AT END phrase, the associated ERROR declarative is executed. Following an AT END condition, the contents of the associated record area are undefined, and attempts to move data to or from the associated record may result in a protection exception.

EXCEPTION/ERROR Declarative: You can code one or more ERROR declarative procedures in your program that will be given control if an input/output error occurs. You can have:

A single, common procedure for the entire program:

- Group procedures for each file open mode (whether INPUT, OUTPUT, I-O, or EXTEND)
- Individual procedures for each particular file

FILE STATUS key: The system updates the FILE STATUS key after each input/output statement executed for a file placing values in the two digits of the file status key. In general, a zero in the first digit indicates a successful operation and a zero in both digits means "nothing abnormal to report". Establish a FILE STATUS key using the FILE STATUS clause in the FILE-CONTROL and data definitions in the Data Division.

INVALID KEY phrase:

- This phrase will be given control in the event that an input/output error occurs because of a faulty index key. You can include INVALID KEY phrases on READ, START, WRITE, REWRITE, and DELETE requests for indexed and relative files.
- You can also include INVALID KEY on WRITE requests for QSAM files. In the case of QSAM files, however, the INVALID KEY phrase has limited meaning. It is used only when you attempt to write to a disk that is full.
- INVALID KEY phrases differ from ERROR declaratives in three ways:
  - INVALID KEY phrases operate for only limited types of errors, whereas the ERROR declarative encompasses all forms.
  - INVALID KEY phrases are coded directly onto the input/output verb, whereas ERROR declaratives are coded separately.
  - INVALID KEY phrases are specific for one single input/output operation, whereas ERROR declaratives are more general.
- If you specify INVALID KEY in a statement that causes an INVALID KEY condition, control is transferred to the INVALID KEY imperative statement. In this case, any ERROR declaratives you have coded are not executed.

Summary

- ❑ File organizations supported by COBOL:
  - o Sequential organization
  - o Relative organization (Random organization or Direct Access)
  - o Indexed Sequential organization
- ❑ Files are to be defined in the Environment Division using SELECT clause and in the Data Division for the Layout Records
- ❑ Files can be opened in either INPUT Mode, OUTPUT Mode, Input-Output Mode (or) EXTEND Mode
- ❑ File Operations that can be done are OPEN, START, READ, WRITE, REWRITE, CLOSE
- ❑ The result of the I/O operations can be validated against the status codes generated.

Test Your Understanding

1. Assume there is a Transaction sequential file with record length 80 characters. The following are the fields in the transaction file

Emp-no	10 chars
Emp-name	30 chars
Emp-birth-date	8 chars(ddmmyyyy)
Emp-designation	8 chars
Emp-department	6 chars
date-of-joining	8 chars
Emp-status	2 chars
Gross-salary	8 numeric with 2 decimal

The master file is the indexed file with the same fields as above.

This transaction file records needs to be inserted in the master indexed file if the emp-no is not present and if the emp-status id NA (new active). If the records are present and if the emp-status id is TT the emp record from the master file needs to be deleted as he is a terminated employee. If the record is present, and the emp status is OA which is old active, the record in the indexed file needs to be updated.

a) Write the COBOL code to do the processing of the transaction and master files. COBOL code includes environment division, data division and procedure division entries.

Session 31: Table Handling

Learning Objectives

After completing this session, you will be able to:

- ❑ Describe the various concepts table handling

Introduction

Sometimes it becomes necessary to handle group of data consisting of similar items. Such a group is called a Table or Array. An array is used to store similar items or elements. For example, a series of totals in the working storage with the same format.

A table is a list of stored fields that are looked up or referenced by the program. It also eliminates the need for separate entries for repeated data items. The usage of tables can be illustrated by the following example

For example when you need to list the months in normal way of representing it is:

```
MONTH-TABLE.
02    MON-1       PIC X(9)    VALUE  ‘JANUARYbb‘.
02    MON-2       PIC X(9)    VALUE  ‘FEBRUARYb ‘.
02    MON-3       PIC X(9)    VALUE  ‘MARCHbbbb‘.
02    MON-4       PIC X(9)    VALUE  ‘APRILbbbb‘.
02    MON-5       PIC X(9)    VALUE  ‘MAY bbbbbb‘.
02    MON-6       PIC X(9)    VALUE  ‘JUNEbbbbbb‘.
02    MON-7       PIC X(9)    VALUE  ‘JULYbbbbbb‘.
02    MON-8       PIC X(9)    VALUE  ‘AUGUSTbbb‘.
02    MON-9       PIC X(9)    VALUE  ‘SEPTEMBER‘.
02    MON-10      PIC X(9)    VALUE  ‘OCTOBERbb‘.
02    MON-11      PIC X(9)    VALUE  ‘NOVEMBERb‘.
02    MON-12      PIC X(9)    VALUE  ‘DECEMBERb‘.
```

In the above example, the same data month name is being repeated 12 times. Instead of defining the elements like this we can define it as a table with 12 elements. Each element of the table can be referred using a Subscript or an index. Now lets us look at how the table can be defined in COBOL

```
01    MONTH-TABLE.
02    MON-NAME    PIC X (9) OCCURS 12 TIMES.
```

This table can be fixed length tables and Variable length tables. The tables or Arrays are define using OCCURS in COBOL. Now let us look at Occurs clause in detail.

Occurs

- Used for defining Tables (Arrays)
- The OCCURS clause specifies tables whose elements can be referred to by indexing or subscripting.
- It also eliminates the need for separate entries for repeated data items.
- Occurs clause can be specified for elementary or group item.
- VALUE clause cannot be specified for associated data name defined with OCCURS clause or any sub item to the occurs clause
- REDEFINES and OCCURS clause cannot appear for the same data item. However REDEFINES clause can appear for a group item whose sub item contains occurs clause
- The ‘occurs’ clause is used to define single dimension and multi-dimensional tables.
- Tables can be Fixed Length or Variable Length
- Can’t be defined for a data item which:
  - Has a level number of 01, 66, 77, or 88.
  - Describes a redefined data item

Fixed Length Tables

```
OCCURS integer-2 TIMES

[ {ASCENDING           KEY  IS
     DESCENDING }
      dataname-2 [ ... ] ]

[ INDEXED BY index-name-1 [  ... ] ]
```

OCCURS - Index Name

- In COBOL, INDEXING can do the subscripting of table.
- An index can be defined for a particular table or dimension of table using INDEXED BY phrase.
- Required if item is **indexed** anywhere in the program
- Conceptually, an index similar to subscript. The internal representation of a subscript and index are different. A subscript is an integer data item. But index is referenced in a more efficient manner internally. INDEX is an offset or displacement of the element from the beginning of the table

Example

```
03    YEAR  PIC   9(4)  OCCURS      10 TIMES INDEXED BY I1.
```

In the above example the displacement of first element is zero, 2nd is 4, 3rd is 8 and so on. That is the index I1 assumes these values for occurrence numbers 1, 2, 3, and so on.

- The value of Index item is set by the occurrence number and not the internal value
- The value of index is set using SET clause. The statements like Move or Add cannot be used on index
- Index name is implicitly defined
- Index name cannot be associated with any other data hierarchy
- Must be initialized before it can be used using SET and PERFORM VARYING

Variable Length Tables Occurs - Depending On

- When a table of variable size needs to be defined, Occurs Depending clause may be used

Example:

```
77    A1    PIC   99

REC-1.
P1    PIC   XX.
02 P2  PIC  99 OCCURS 1 TO 90 TIMES DEPENDING ON A1.
```

In this example depending on the value of A1, the size of the table may vary from 1 to 90.

The syntax is as follows:

```
OCCURS int1 TO int2 TIMES
DEPENDING ON data-name-1

[{ASCENDING           KEY  IS
    DESCENDING }
      dataname-2 [ ... ] ]
[ INDEXED BY index-name-1 [  ... ] ]
```

- Object of the OCCURS DEPENDING ON (ODO) clause cannot be variably located;
- If subject of OCCURS is defined as EXTERNAL then ODO ought to be defined as EXTERNAL
- The int-1 and int-2 defined must be positive integers.
- The data name-1 can be defined at the same level as ODO but it cannot be defined as sub item to ODO.

Example:

```
01    REC-1.
02    A1    PIC   99.
02    P1    PIC   XX.
02    P2    PIC   99 OCCURS 1 TO 90 DEPENDING ON A1.
The above example is valid.
01    REC-1.
02    P1    PIC   XX.
02    P2    PIC   99 OCCURS 1 TO 90 DEPENDING ON A1.
02    A1    PIC   99.
```

This definition is invalid.

- GLOBAL definition needs rules similar to EXTERNAL
- ODO cannot be REDEFINED

Example - This is invalid

```
01    REC-1.
      02    A1    PIC   99.
      02    P1    PIC   XX.
      02    P2    PIC   99    OCCURS 1 TO 90 TIMES DEPENDING ON A1.
      02    P3    REDEFINES   P2
```

- Length of group item containing Occurs depending:
  - If ODO is outside the group the ODO value decides
  - If ODO is inside and group item is sending field then also ODO value decides
  - If ODO is inside and group item is receiving field then max length of group value used provided there are no non-subordinate to the group item

SET

Syntax – Format1:

```
                                  { index-name-2 }
SET { index-name-1 } ... TO       { identifier-2 }
    { identifier-1 }              { integer-1    }
```

Syntax – Format2:

```
                          { UP   }      { identifier-3 }
SET {index-name-3} ...    { DOWN }  BY  { integer-2    }
```

Syntax – Format3:

```
                                      { ON  }
SET { {mnemonic-name-1} ... TO        { OFF } } ...
```

Syntax – Format4:

```
SET {condition-name-1} ... TO TRUE...
```

Description: The SET statement is used to set values for indices and conditions. Each of the four formats works differently and is described below in the correspondingly numbered item.

- The first format of the SET statement is used to assign values to indices or assign the value of an index to another variable. The variable(s) between SET and TO is set to the value of the data item or literal following TO.
- The second format of the SET statement is used to increase or decrease the value of an index. The variable(s) between SET and UP or DOWN is INCREASED (for UP) or DECREASED (for DOWN) by the value of the data item or literal following BY.
- The third format of the SET statement is used to set the value of an external switch. The mnemonic(s) between SET and TO is set to ON or OFF according to the ON or OFF option supplied.
- The fourth format of the SET statement is used to set the condition-name conditions to true. The condition(s) between SET and TO is set to TRUE.

Tips:

- Regular MOVE and mathematical statements do not work on indices. Use the SET statement to manipulate an index.
- For format 4, if more than one value is associated with the condition-name being true, the first value listed is the one that will be used.

SET – Examples:

```
05  TABLE-ITEM OCCURS 10
        INDEXED BY INX-A PIC X (8).
```

- To set the index INX-A to 5th occurrence of TABLE-ITEM use stmt:
  ```
  SET INX-A   TO 5
  ```
- The Value in INX-A would be:
  ```
  (5 - 1) * 8 = 32.
  ```
- To refer to 6th element we could use relative indexing:
  ```
  -  TABLE-ITEM (INX-A+1)
  ```
- Or Set the index first using:
  ```
  SET INX-A  UP BY 1 and then refer as TABLE-ITEM(INX-A)
  ```

Set - Sending and Receiving Fields

	Receiving Fields		
Sending fields	**Index Name**	**Index Data Item**	**Integer Data Item**
Index Name	V	V*	V
Index data item	V*	V*	-
Integer data item	V	-	-
Numeric literal	V	-	-

Note: * - No Conversion

Complex Tables

- Data item with Occurs Depending On is followed by:
  - Non-subordinate item
  - Non-subordinate item with Occurs Depending On
- Data item with Occurs Depending On is nested

```
05    CTR1                              PIC S99.
05    CTR2                              PIC S99.
05    FIELD-A.
      1
      10    TABLE-1.
      15    REC-1 OCCURS 1 TO 5
            DEPENDING ON  CTR1          PIC X(3).
      10    EMP-NR                      PIC X (5).
2
      10    TABLE-2 OCCURS 5
3
            INDEXED BY INDX.
4
      15    TABLE-ITEM                  PIC 99.

5
      15    REC-2 OCCURS 1 TO 3 DEPENDING ON CTR2.
      20    DATA-NUM                    PIC S99.
```

- 1 -A group item of variable length
- 2 - An elementary data item following, and not subordinate to, a variable length table (variably located item)
- 3 - A group item following, and not subordinate to, a variable-length table
- 4 - An index name for a table that has variable length elements
- 5 - An element of a table that has variable-length elements

Watch out:

- The location of non-subordinate items following the item with OCCURS clause is affected by new value of the ODO object. f you want to preserve the contents of these items, prior to the change in the ODO object, save all non-subordinate items following the variable item and after the change in the ODO object, restore all the items back.
- Careful when using complex-ODO index names. In the example if the value of the ODO object CTR2 is changed then the offset in index INDX is no longer valid for the table TABLE-2.
- To avoid making this error save the value of index as an occurrence number before changing ODO object.

After changing ODO object, restore the value of index name from the integer data item

```
77 INT-DATA-1    PIC 99.
77 NEW-VALUE  PIC S99.
SET INT-DATA-1 TO INDX
MOVE NEW-VALUE TO CTR2
SET INDX TO INT-DATA-1
```

OCCURS – Example:

```
WORKING STORAGE SECTION.
01    TABLE-RECORD.
      05    EMP-TABLE OCCURS 100 ASCENDING KEY IS
            WAGE-RATE EMP-NO INDEXED BY A,  B.
            10    EMP-NAME                      PIC X (20).
            10    EMP-NO                        PIC 9(6).
            10    WAGE-RATE                     PIC 9999V99.
            10    WEEK-RECORD OCCURS 52 ASCENDING
                  KEY IS WEEK-NO INDEXED BY C.
                  15    WEEK-NO                 PIC 99.
                  15    ABSENCES                PIC 99.
```

Table Processing

Now we have seen how to define tables of various sizes and dimension, we now see how to process these tables using PERFORM verb.

Example:

Consider the following example:

```
77    A1                            PIC   99.
01    PRICE-1                       PIC S9 (7) V99.
DATA-REC.
05    PRODUCT-CODE            PIC X (6)
      05    PRICE-REC   OCCURS 36 TIMES.
10    PERIOD            PIC X (10).
10    PROD-PRICE        PIC S9 (7) V99.
```

Now we want to find the average price of the product for a period.

```
MOVE ZERO TO TOTAL.
MOVE 1 TO A1.
PERFORM ADD-PRICE-PARA 36 TIMES.
COMPUTE AVERAGE = TOTAL / 36

ADD-PRICE-PARA.
      COMPUTE     TOTAL = PROD-PRICE (A1) + TOTAL
      ADD 1 TO A1.
```

This is one form of PERFORM statement used for FIXED length tables.

Now you want to find the price of a product for period 011999 (mmyyyy)

```
MOVE 1 TO A1
PERFORM FIND-PRICE-PARA UNTIL
                                                  (PERIOD = ‘011999’ OR
                          A1 > 36)
FIND-PRICE-PARA
      IF PERIOD = ‘011999’
                MOVE PROD-PRICE (A1)        TO PRICE-1
      END-IF
          ADD 1 TO A1
```

In this form, the paragraph is executed repetitively until the period id 011999 or until the subscript of the table is greater than the maximum table limits.

The same fixed length table can be defined as variable length table.

In this case again you want to find the price of a product for period 011999(mmyyyy)

```
77 PRICE-1                    PIC S9 (7) V99.
01 DATA-REC.
      05 PRODUCT                    PIC X(10).
      05 TAB-CNT                          PIC S9(04) COMP.
      05 DATA-TABLE   OCCURS 1  TO  37 TIMES
                                 DEPENDING ON TAB-CNT
                                       INDEXED   BY     INX-1.
          07   PERIOD               PIC X(6).
          07  PROD-PRICE            PIC S9(7)V9(02).

PROCEDURE DIVISION.

SET INX-1 TO 1.
PERFORM FIND-PRICE-PARA VARYING INX-1 BY 1 UNTIL
                              INX-1 > TAB-CNT OR
                              PERIOD = ‘011999’

FIND-PRICE-PARA
      IF PERIOD = ‘011999’
                MOVE PROD-PRICE(INX-1)         TO PRICE-1
       END-IF
```

In the earlier statement the increment of the INDEX is not given explicitly.

It is done automatically:

- Either fixed length or variable length the table can be in the sorted order by defining a key for the table.

- The sorted table reduces the time of search, when a search is performed on the table.

The example for SORTED TABLES:

```
02 ASSOC-DATA
     OCCURS 10 TIMES
     ASCENDING KEY IS A-NAME
      INDEXED BY INX-1.
      05 A-NAME       PIC X(20).
      05 A-NUM         PIC 9(04).
```

- The data have to be sorted in the correct order (ascending or descending) by the programmer.

Search

- To search a table element that satisfies a specified condition
- Options
- SEARCH VARYING
- LINEAR SEARCH
- SEARCH ALL

Binary Search, valid only for sorted tables with key specified.

SEARCH - Sequential

Syntax – Format:

```
                       [          { identifier-2  } ]
SEARCH identifier-1    [ VARYING  { index-name-1  } ]

       [AT END imperative-statement-1]

       {                  { imperative-statement-2 } }
       {  WHEN condition-1 { NEXT SENTENCE          } } ...

       [END-SEARCH]
```

- This search is serial search or linear search.
- The table may sorted or not
- WHENs are performed one after another for each table entry until TRUE or NO more WHEN condition
- AT END executed if no match
- Set starting points of id-2 or index. The increment of the index or id-2 is taken care of by the search statement itself.
- The search statement can be performed on he table defined by OCCURS and INDEXED BY phrase.
- The search verb starts with initial value of index and tests whether condition stated in the when clauses have been satisfied or not. If none of the conditions are satisfied the index is incremented by 1 automatically. The process is continued until the index value exceeds the size of the table and the search is terminated. When one of conditions is

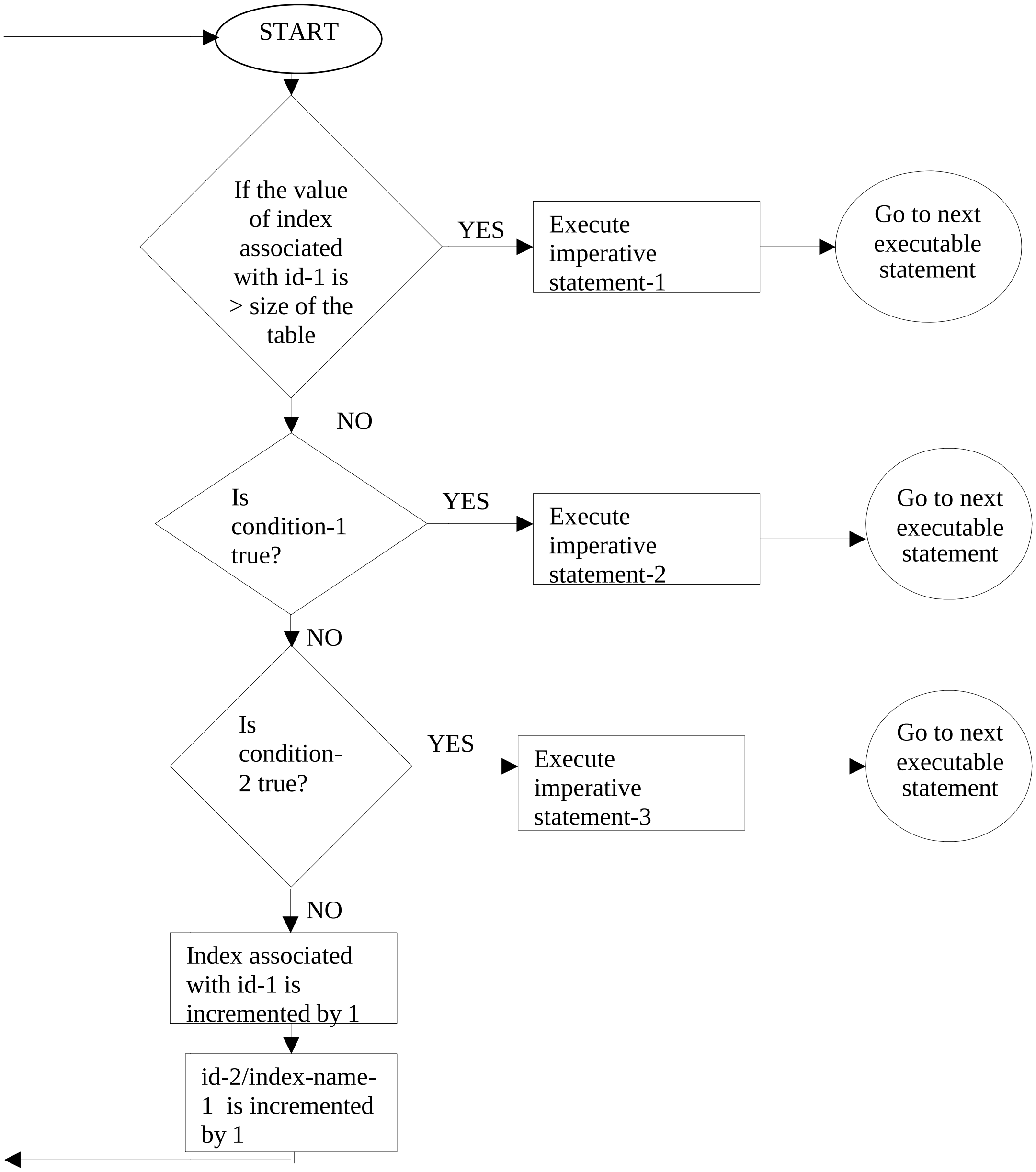
satisfied before the index value reaches the size of the table, the statements following the WHEN clause is executed. If these statements are not present, then execution is transferred to next statement. The value of the index remains set at the point where the condition has been satisfied.

- When the SEARCH terminates without finding the necessary value, the index contains unpredictable values.

Example

```
77 NAME                       PIC X(20).
05 ASSOC-TABLE  OCCURS 350 TIMES
          INDEXED BY I-1.
          10 ASSOC-NAME       PIC X (20).
          10 ASSOC-NUM        PIC 9(4).
SET I-1 TO 1.
SEARCH ASSOC-TABLE
  AT END
     DISPLAY ‘NAME NOT FOUND’
  WHEN NAME = ASSOC-NAME (I-1)
        DISPLAY ‘NAME FOUND’.
```

Flow chart for SEARCH with varying and when option.


START
If the value of index associated with id-1 is > size of the table
YES
Execute imperative statement-1
Go to next executable statement
NO
Is condition-1 true?
YES
Execute imperative statement-2
Go to next executable statement
NO
Is condition-2 true?
YES
Execute imperative statement-3
Go to next executable statement
NO
Index associated with id-1 is incremented by 1
id-2/index-name-1 is incremented by 1

SEARCH ALL

Syntax – Format:

```
SEARCH ALL identifier-3
         [AT END imperative-statement-3]
                                    IS         identifier-4
                   { identifier-  {  EQUAL  } { literal-1      } }
         WHEN      {  3           {  TO     } { arithmetic-    } }
                   {                 IS =   } { expression-1   } }
                     condition-name-1
                                    IS         identifier-6
         [         { identifier-  {  EQUAL  } { literal-2      } }  ]
         [  AND    {  5           {  TO     } { arithmetic-    } }  ]
         [         {                 IS =   } { expression-2   } }  ]
                     condition-name-2
                 { imperative-
                 { statement-4        }}
                 { NEXT SENTENCE
         [END-SEARCH]
```

Equal-clause can be {id-2, lit-1, arithexp-1}:

- This SEARCH all is the Binary search.
- For this search the table needs to be in sorted order using ASCENDING/DESCENDING option in occurs clause
- When the binary search is used, it is assumed that at the time of search the table is arranged in the sorted order depending on the definition of the table.
- In this search, the table is split into 2 halves and determines in which half the item to be searched is present. Then item to be searched is compared with the last item of the first half or first item of the second to determine in which the item to be searched is likely to be present. Then again the half that is selected is split into 2 and the comparison is done to determine in which half the item to be searched is present. Like wise the search continues until the item to be searched is found or the final division is just a single item. Now a single comparison with this item determines whether the item is present or not.
- This type of search minimizes the number of comparison that needs to be done on the table elements.
- Initial setting of index ignored, but if set it must not be greater than the maximum size
- Duplicate index will indicate the first entry it encountered
- Unpredictable results if table is not in sorted order

```
77 NAME                       PIC X(20).
05 ASSOC-TABLE  OCCURS 350 TIMES
        ASCENDING KEY IS A-NAME
      INDEXED BY I-1.
            10 A-NAME   PIC X(20).
            10 A-NUM    PIC 9(4).
SEARCH ALL ASSOC-TABLE
  AT END
     DISPLAY ‘NAME NOT FOUND’
  WHEN  NAME = A-NAME(I-1) DISPLAY ‘NAME FOUND’
```

Summary

- Tables in COBOL are declared using the OCCURS clause and can be of single dimension and multiple dimensions
- Indexes and subscripts are used to access the individual elements of the table
- SEARCH statement is used to extract a particular element from a Table based on a criteria
- SERACH ALL is faster than SERACH but required the table to be sorted as it perform Binary search on the table.

Test your Understanding

1. Write the data division statements for the following table:
   a) The table contains marks of student for the each month for the whole year. The marks will be 3 digits.
   b) The insurance company maintains the premium paid by the policyholders on a periodic basis. The period depends on the policy. It can be a monthly, half-yearly, quarterly or annual premium. The premium is 13 digits with 2 decimal places. Write the table structure with policy number, period and premium as fields. This is a variable length table
2. How many times will the procedure named PROCESS-ROUTINE be executed by the following PERFORM statements?
   a) PERFORM PROCESS-ROUTINE VARYING A FROM 1 BY 1 UNTIL A = 15.
   b) PERFORM PROCESS-ROUTINE VARYING A FROM 1 BY 1 UNITL A > 15.
3. The following are DATA DIVISION entries.

```
77          I                 PIC   99.
77          TOTAL       PIC   99.
TABLE-1.
02    FILLER      PIC   X (9) VALUE IS    “123456789”.
      01          TABLE-2     REDEFINES   TABLE-1.
            02    A     OCCURS      9     TIMES PIC   9.
```

Indicate how many elements of the array A would be added to TOTAL and what will be the final value of TOTAL when the control goes to PARA-2 in the following program segment?

```
MOVE ZERO TO TOTAL.
PERFORM PARA-1 VARYING I FROM 1 BY 2 UNTIL I > 9.
GO TO PARA-2
PARA-1.
      ADD A (I) TO TOTAL.
PARA-2.
```

4. A table contains the monthly sales data for 12 months of the year and for the 4 sales zones where each zone has 8 districts. Write the PROCEDURE DIVISION statements to find the total sale of each district within each zone and then find the total sale of each zone. Store these totals in 2 tables - one district total and the other containing the zone totals. Write the data division entries to define all these tables. The following is a PROCEDURE DIVISION statement.

```
PERFORM PARA-AGAIN VARYING I FROM 1 BY 1 UNTIL I > 4
      AFTER J FROM 0 BY –1 UNTIL J < -2
      AFTER K FROM 1 BY 2 UNTIL K > 8.
```

a) How many times is the procedure named PARA-AGAIN executed? Which data name varied least rapidly and which varied most rapidly.

Session 33: Inter-Programming Communication

Learning Objectives

After completing this session, you will be able to:

- Explain the concepts of calling a sub-program

Call

- Transfers control from one object program to another within the run unit.
- Called program starts executing from:
  - Top of program
  - ENTRY label (not good programming practice)
- Transfer control methods:
  - Call nested program
  - Static call
  - Dynamic call
- Parameters to called program:
  - By reference
  - By content
- Use RETURN-CODE special register to pass return codes
- Return of control depends on the termination stmt issued by the called program:
  - Stop run unit
  - Return to called program

Syntax – Format1:

```
CALL { identifier-1 }
     { literal-1    }

                                   { identifier-2 }
                     BY            { ADDRESS OF   }
                     REFERENCE     { identifier-3 }  ...
                                   { file-name-1  }
                 {                                       }
                 {                                       }  ...
[ USING {                          { [LENGTH OF]  }              } ]
                                   { identifier-2 }
                     BY CONTENT    { ADDRESS OF   }  ...
                                   { identifier-3 }
                                   { literal-2    }

                 { identifier-2 }
                 { literal-2    }  ...

[ON OVERFLOW imperative-statement-1]
```

```
[END-CALL]
```

Syntax – Format2:

```
CALL { identifier-1 }
     { literal-1    }

                                 { identifier-2  }
                BY               { ADDRESS OF    }
                REFERENCE        { identifier-3  } ...
                                 { file-name-1   }
            {                                        } ...
                                 { [LENGTH OF]   }
                                 { identifier-2  }
[ USING {       BY CONTENT       { ADDRESS OF    } ...        } ]
                                 { identifier-3  }
                                 { literal-2     }

            { identifier-
              2
              literal-2 } ...

[ON EXCEPTION imperative-statement-1]
[NOT ON EXCEPTION imperative-statement-2]
[END-CALL]
```

Description: The CALL statement is used to call other programs. While the format of the CALL itself is the same for both formats; the error handling for each is different, each is described in the correspondingly numbered area. For the primary part of the call, each item after the USING represents a parameter which is to be passed to the called subprogram. BY REFERENCE means the actual item will be passed and the item may be modified by the subprogram. BY CONTENT means that only the value of the item will be passed and the item may not be modified by the subprogram. If neither option is specified, the items are passed as by reference parameters.

- The first format is used to check for overflow errors. An overflow occurs when there is not enough memory available to hold the called subprogram. If such an error occurs, the code between ON OVERFLOW and END-CALL will be executed. If no error occurs, control will pass to the statement following the END-CALL.
- The second format is used to check for the occurrence of any error. If such an error occurs, the code between ON EXCEPTION and NOT ON EXCEPTION or END-CALL will be executed. If no error occurs, code between NOT ON EXCEPTION and the END-CALL will be executed.

Tips: he CALL statement is used to execute code outside of your program. To execute code within your program, use the PERFORM statement.

CALL – Examples:

```
CALL ‘SUBPGM1’.
CALL l-SUBPGM-1 USING
        l-PARM1   l-PARM2.
CALL l-SUBPGM1          USING
        BY CONTENT            l-PARM1
                        l-PARM2
        BY REFERENCE    l-PARM3.
```

CALL - Calling program:

```
WORKING-STORAGE SECTION.
       77 l-subpgm-1    pic x(8) value  ‘SUBPGM1’.

      01 l-parm1.
              05 l-name       pic x(30).
              05 l-emp-no     pic 9(4).
      01 l-parm2.
              05 l-salary     pic s9(9)v99  comp-3.
              05 l-hra        pic s9(9)v99  comp-3.
              05 l-leave      pic s9(2)v9    comp-3.
      01 l-parm3.
              05 l-gross      pic s9(9)v99 comp-3.
              05 l-deduct     pic s9(9)v99 comp-3.

PROCEDURE DIVISION.
....
CALL  l-SUBPGM1  USING
      BY CONTENT      l-PARM1  l-PARM2
      BY REFERENCE  l-PARM3.

IDENTIFICATION DIVISION.
PROGRAM-ID. 1-SUBPGM1.
…

LINKAGE SECTION.
      01 L-PARM3.
      05 l-gross        pic s9(9)v99 comp-3.
      05 l-deduct       pic s9(9)v99 comp-3.
      01 L-PARM1.
      05 l-name         pic x(30).
      05 l-emp-no pic 9(4).
      01 L-PARM2.
      05 l-salary       pic s9 (9) v99 comp-3.
      05 l-hra          pic s9(9)v99 comp-3.
      05 l-leave        pic s9(2)v9   comp-3.

PROCEDURE DIVISION USING
L-PARM1  L-PARM2    L-PARM3.
```

CALL - Rules

- The correspondence of identifiers in the **using clause** of called and calling programs is positional
- **File-name** in using is only for QSAM files
- **Address of** option can be used only for Linkage variables with level 01 or 77
- **Exception** or **Overflow** condition occurs when the called program cannot be made available.
- Called program must not execute a CALL statement that directly or indirectly calls the calling program (Recursion not allowed)

Cancel

CANCEL {id-1   lit-1}:

- Ensures that the next time the referenced subprogram is called it will be entered in its initial state.
- All programs contained in the Cancelled program are also cancelled.
- Same as executing an EXIT PROGRAM or GOBACK in the called subprogram if it possesses the INITIAL attrib.

Entry

ENTRY lit-1 [USING id-1 ...]:

- Establishes an alternate entry point
- Execution of the called program begins at the first executable stmt following the ENTRY stmt whose literal corresponds to the CALL stmt literal or identifier
- Not a recommended way of entering a program

Entry - Watch out:

- Static calls to alternate entry point's work without restriction.
- Dynamic calls to alternate entry points require:

  NAME or ALIAS linkage editor control statements.

  NAME (ALIAS) compiler option to generate link-edit ALIAS card for each ENTRY statement.

Exit

EXIT PROGRAM:

- Specifies the end of a called program and returns control to the calling program
- When no CALL statement is active, control passes through the exit point to the next executable stmt

- When there is no next executable stmt in a **called** program, an implicit EXIT PROGRAM stmt is executed
- EXIT PROGRAM stmt in a called program with INITIAL attribute is equivalent to executing a CANCEL
- An EXIT PROGRAM executed in a main program has no effect.

Stop

- STOP {RUN
  lit-1}
- Halts execution of the program:
  - Permanently (RUN option)
  - Temporarily (Lit-1 option)
- Literal communicated to operator and execution suspended
- Program execution is resumed only after operator intervention

STOP RUN statement closes all files defined in any of the programs comprising the run unit.

GOBACK

GOBACK:

- Functions like:
  - EXIT PROGRAM statement when coded in a called program
  - STOP RUN statement when coded in a main program.

END PROGRAM

END PROGRAM program-name:

- An END PROGRAM header terminates a nested program or separates one program from another in a sequence of programs.
- The program-name must be same as the program-name declared in the corresponding PROGRAM-ID paragraph.
- An END PROGRAM is optional for the last program in a sequence only if that program does not contain any nested source programs.

Usage Clause

The data can be stored in more than one internal form in the computer. In COBOL, a programmer is allowed to specify the internal form of the data item so as to facilitate the use of the data item more efficiently. The USAGE clause specifies the format of a data item in computer storage. These are forms of internal representation

- Computational
- Display
- Pointer
- Packed – decimal

The syntax of the USAGE clause is:

```
USAGE IS   {BINARY
                   COMP       COMPUTATIONAL
                       COMP-1 COMPUTATIONAL-1
                       COMP-2       COMPUTATIONAL-2
                       COMP-3       COMPUTATIONAL-3
                       COMP-4       COMPUTATIONAL-4
                       DISPLAY
INDEX
                       PACKED-DECIMAL
                       POINTER}
```

COMPUTATIONAL: The Computational or COMP phrase has the following 6 formats:

- A computational item is a value used in arithmetic operations. It must be numeric. If the USAGE of a group item is described with any of these items, the elementary items within the group have this usage.
- The maximum length of a computational item is 18 decimal digits.
- The PICTURE of a computational item can contain only:
  - 9: - One or more numeric character positions
  - S :- One operational sign
  - V :- One implied decimal point
  - P :- One or more decimal scaling positions

BINARY:

- Specified for binary data items. Such items have a decimal equivalent consisting of the decimal digits 0 through 9, plus a sign. Negative numbers are represented as the two's complement of the positive number with the same absolute value.
- The amount of storage occupied by a binary item depends on the number of decimal digits defined in its PICTURE clause:

Digits in PICTURE Clause	Storage Occupied
1 through 4	2 bytes (half word)
5 through 9	4 bytes (full word)
10 through 18	8 bytes (double word)

COMPUTATIONAL or COMP: This is the equivalent of BINARY. The COMPUTATIONAL phrase is synonymous with BINARY.

COMPUTATIONAL-1 or COMP-1 (Floating-Point): Specified for internal floating-point items (single precision). COMP-1 items are 4 bytes long. The number is actually represented in Hexadecimal form. Such representation is suitable for Arithmetic operations. The PIC clause cannot be specified for this item

COMPUTATIONAL-2 or COMP-2 (Long Floating-Point): Specified for internal floating-point items (double precision). COMP-2 items are 8 bytes long. The advantage is that this increases the precision of the data, which means more significant digits can be available for the item. The PIC clause cannot be specified

COMPUTATIONAL-3 or COMP-3 (Internal Decimal): This is the equivalent of PACKED-DECIMAL. In this form of internal representation the numeric data is represented in the decimal form, but one digit takes half-a-byte.

The sign is stored separately as rightmost half-a-byte regardless of whether S is specified in the PIC clause or not. The hexadecimal number C or F denotes positive sign and D denotes negative sign.

The number of bytes = n+1/2 Where n is the number of places specified in PIC clause

For example:

```
77 ITEM-1        PIC      S9(7) COMP-3
```

Will occupy 4 bytes:

COMPUTATIONAL-4 or COMP-4 (Binary)

This is the equivalent of BINARY.

DISPLAY: The data item is stored in character form, 1 character for each 8-bit byte. This corresponds to the format used for printed output.

DISPLAY can be explicit or implicit.

USAGE IS DISPLAY is valid for the following types of items:

- Alphabetic
- Alphanumeric
- Alphanumeric-edited
- Numeric-edited
- External floating-point
- External decimal (numeric)

INDEX:

- A data item defined with the INDEX phrase is an index data item.
- An index data item is a 4-byte elementary item (not necessarily connected with any table) that can be used to save index-name values for future reference. Through a SET statement, an index data item can be assigned an index-name value; such a value corresponds to the occurrence number in a table.
- Direct references to an index data item can be made only in a SEARCH statement, a SET statement, a relation condition, the USING phrase of the Procedure Division header, or the USING phrase of the CALL statement.
- An index data item can be part of a group item referred to in a MOVE statement or an input/output statement.

- An index data item saves values that represent table occurrences, yet is not necessarily defined as part of any table. Thus, when it is referred to directly in a SEARCH or SET statement, or indirectly in a MOVE or input/output statement, there is no conversion of values when the statement is executed.
- The USAGE IS INDEX clause can be written at any level. If a group item is described with the USAGE IS INDEX clause, the elementary items within the group are index data items; the group itself is not an index data item, and the group name cannot be used in SEARCH and SET statements or in relation conditions. The USAGE clause of an elementary item cannot contradict the USAGE clause of a group to which the item belongs.
- An index data item cannot be a conditional variable.
- The DATE FORMAT, JUSTIFIED, PICTURE, BLANK WHEN ZERO, SYNCHRONIZED, or VALUE clauses cannot be used to describe group or elementary items described with the USAGE IS INDEX clause.
- SYNCHRONIZED can be used with USAGE IS INDEX to obtain efficient use of the index data item.

POINTER:

A data item defined with USAGE IS POINTER is a pointer data item. A pointer data item is a 4-byte elementary item.

You can use pointer data items to accomplish limited base addressing. Pointer data items can be compared for equality or moved to other pointer items.

A pointer data item can only be used:

- In a SET statement (Format 5 only)
- In a relation condition
- In the USING phrase of a CALL statement, an ENTRY statement, or the Procedure Division header.

The USAGE IS POINTER clause can be written at any level except level 88. If a group item is described with the USAGE IS POINTER clause, the elementary items within the group are pointer data items; the group itself is not a pointer data item and cannot be used in the syntax where a pointer data item is allowed. The USAGE clause of an elementary item cannot contradict the USAGE clause of a group to which the item belongs.

Pointer data items can be part of a group that is referred to in a MOVE statement or an input/output statement. However, if a pointer data item is part of a group, there is no conversion of values when the statement is executed.

- A pointer data item can be the subject or object of a REDEFINES clause.
- SYNCHRONIZED can be used with USAGE IS POINTER to obtain efficient use of the pointer data item.
- A VALUE clause for a pointer data item can contain only NULL or NULLS.
- A pointer data item cannot be a conditional variable.
- A pointer data item does not belong to any class or category.

- The DATE FORMAT, JUSTIFIED, PICTURE, and BLANK WHEN ZERO clauses cannot be used to describe group or elementary items defined with the USAGE IS POINTER clause.
- Pointer data items are ignored in CORRESPONDING operations.
- A pointer data item can be written to a data set, but, upon subsequent reading of the record containing the pointer, the address contained can no longer represent a valid pointer.
- USAGE IS POINTER is implicitly specified for the ADDRESS OF special register.

Difference Between Continue and Next Sentence

CONTINUE:

Syntax:

```
CONTINUE
```

Description: The CONTINUE statement is used as a place holder in conditional code. It has no effect on the execution of the program.

Tips:

- Use the CONTINUE statement in conditional code when no action is required for that particular condition.
- It is not necessary to code a CONTINUE statement in an ELSE when no action is required for that particular condition. If no action is required for an ELSE, do not code the ELSE, simply code the END-IF.

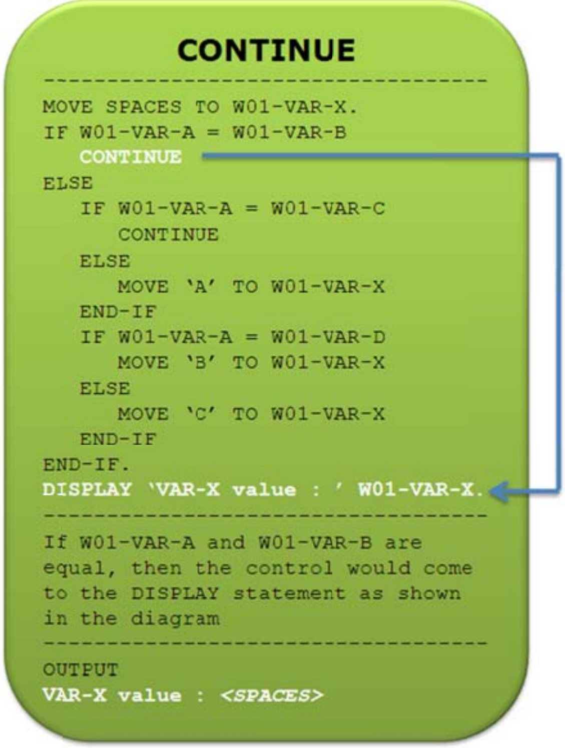
How NEXT SENTENCE Differs from CONTINUE:

- CONTINUE and NEXT SENTENCE are place holder statements.
- If there is logic only in the ELSE part of an IF statement, then this is how it is coded.

```
IF  W01-VAR-A = W01-VAR-B
    CONTINUE (or) NEXT SENTENCE
ELSE
    MOVE W01-VAR-B TO W01-VAR-A
END-IF.
```

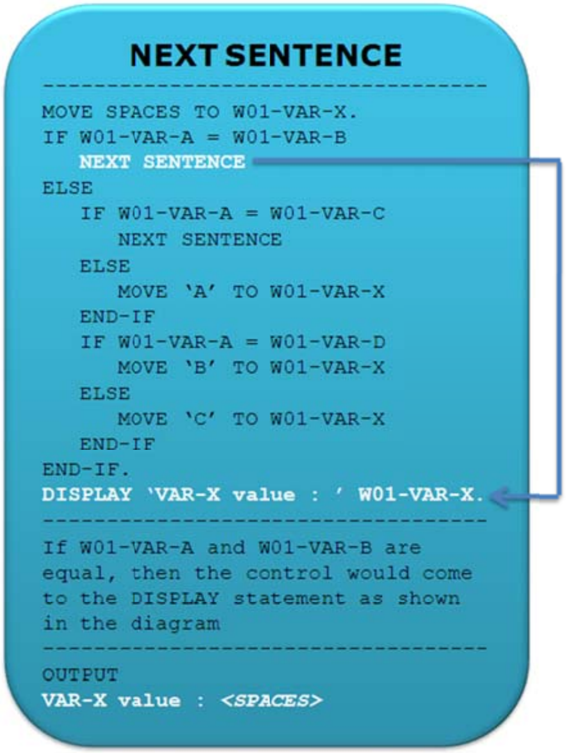
- The CONTINUE statement is used as a place holder in conditional code. It has no effect on the execution of the program.
- NEXT SENTENCE gives control to the verb following the next period. CONTINUE gives control to the next verb after the explicit scope terminator.
- The preceding two slides will explain clearly the difference.

Example 1:



CONTINUE

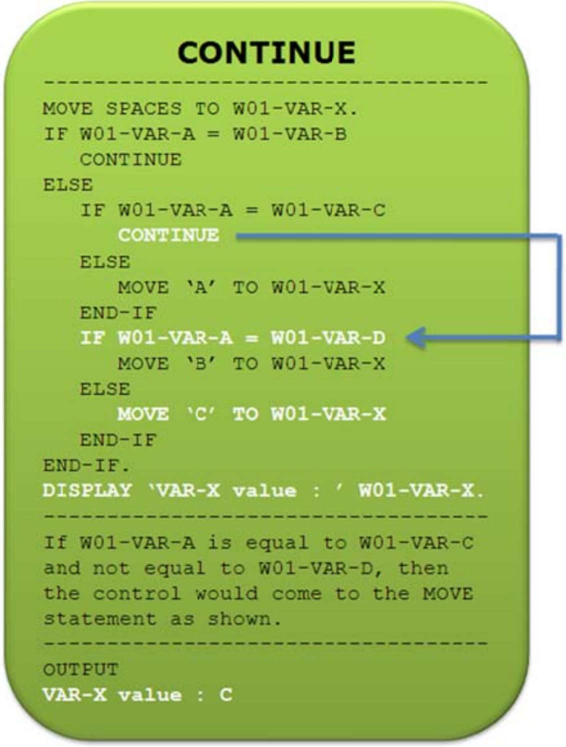
```
-------------------------------------
MOVE SPACES TO W01-VAR-X.
IF W01-VAR-A = W01-VAR-B
   CONTINUE
ELSE
   IF W01-VAR-A = W01-VAR-C
      CONTINUE
   ELSE
      MOVE 'A' TO W01-VAR-X
   END-IF
   IF W01-VAR-A = W01-VAR-D
      MOVE 'B' TO W01-VAR-X
   ELSE
      MOVE 'C' TO W01-VAR-X
   END-IF
END-IF.
DISPLAY 'VAR-X value : ' W01-VAR-X.
-------------------------------------
If W01-VAR-A and W01-VAR-B are
equal, then the control would come
to the DISPLAY statement as shown
in the diagram
-------------------------------------
OUTPUT
VAR-X value : <SPACES>
```



NEXT SENTENCE

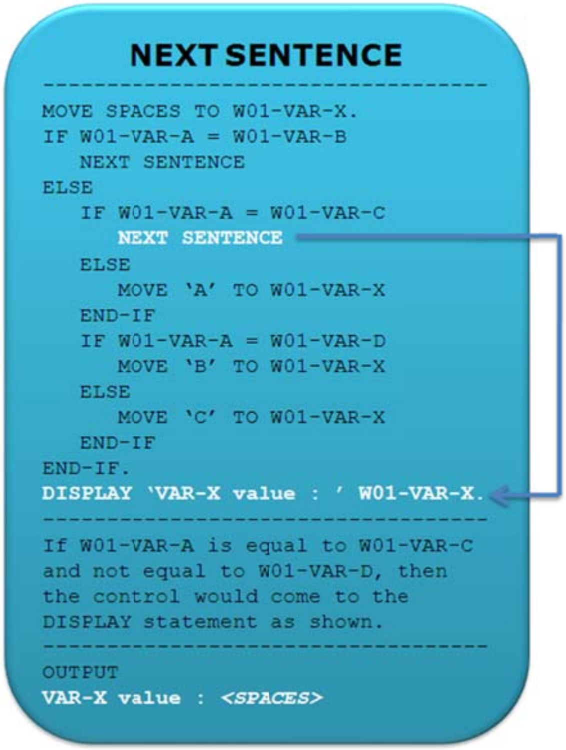
```
-------------------------------------
MOVE SPACES TO W01-VAR-X.
IF W01-VAR-A = W01-VAR-B
   NEXT SENTENCE
ELSE
   IF W01-VAR-A = W01-VAR-C
      NEXT SENTENCE
   ELSE
      MOVE 'A' TO W01-VAR-X
   END-IF
   IF W01-VAR-A = W01-VAR-D
      MOVE 'B' TO W01-VAR-X
   ELSE
      MOVE 'C' TO W01-VAR-X
   END-IF
END-IF.
DISPLAY 'VAR-X value : ' W01-VAR-X.
-------------------------------------
If W01-VAR-A and W01-VAR-B are
equal, then the control would come
to the DISPLAY statement as shown
in the diagram
-------------------------------------
OUTPUT
VAR-X value : <SPACES>
```

Example 2:



CONTINUE

```
-------------------------------------
MOVE SPACES TO W01-VAR-X.
IF W01-VAR-A = W01-VAR-B
   CONTINUE
ELSE
   IF W01-VAR-A = W01-VAR-C
      CONTINUE
   ELSE
      MOVE 'A' TO W01-VAR-X
   END-IF
   IF W01-VAR-A = W01-VAR-D
      MOVE 'B' TO W01-VAR-X
   ELSE
      MOVE 'C' TO W01-VAR-X
   END-IF
END-IF.
DISPLAY 'VAR-X value : ' W01-VAR-X.
-------------------------------------
If W01-VAR-A is equal to W01-VAR-C
and not equal to W01-VAR-D, then
the control would come to the MOVE
statement as shown.
-------------------------------------
OUTPUT
VAR-X value : C
```



NEXT SENTENCE

```
-------------------------------------
MOVE SPACES TO W01-VAR-X.
IF W01-VAR-A = W01-VAR-B
   NEXT SENTENCE
ELSE
   IF W01-VAR-A = W01-VAR-C
      NEXT SENTENCE
   ELSE
      MOVE 'A' TO W01-VAR-X
   END-IF
   IF W01-VAR-A = W01-VAR-D
      MOVE 'B' TO W01-VAR-X
   ELSE
      MOVE 'C' TO W01-VAR-X
   END-IF
END-IF.
DISPLAY 'VAR-X value : ' W01-VAR-X.
-------------------------------------
If W01-VAR-A is equal to W01-VAR-C
and not equal to W01-VAR-D, then
the control would come to the
DISPLAY statement as shown.
-------------------------------------
OUTPUT
VAR-X value : <SPACES>
```

Summary

- **Calling program:** The program that references or calls another subprogram
- **Called program:** The subprogram that is linked and executed within the calling program
- Use of Linkage Section:
  - Data transfer from JCL to program
  - Data transfer across COBOL programs
- Types of calls are static and dynamic
- STOP RUN and GO BACK terminate the job execution, and delete all dynamically called programs in the run unit and all programs link-edited with them.
- The REDEFINES clause allows you to define the same storage area in main memory for different data items whose lengths are not described as variable in an OCCURS clause.

Test Your Understanding

1. What is the compile parameter for having a dynamic call?
2. CONTINUE and NEXT SENTENCE always leads to different results. State true or false.
3. In case of any changes to the called program should the calling program be recompiled?
4. Mention one other use of LINKAGE Section other than using it for passing information across programs.

Session 38: Performance Tuning

Learning Objectives

After completing this session, you will be able to:

- ❑ Explain the Performance Considerations for CALLS
- ❑ Comprehend the Performance Considerations for QSAM files
- ❑ Explain the Performance Considerations for Datatypes
- ❑ Comprehend the Performance Considerations for Constant and Variables
- ❑ Identify the Performance Considerations for Tables
- ❑ Identify the Performance Considerations for Indexes and Subscripts
- ❑ Identify the Performance Considerations for Compiler Options

Performance Considerations for CALLS

- CALLS:
  - » Call literal(Static call) faster than dynamic call literal.
    - **Example** :
      - **CALL ‘*PROGRAMA*’ - Static call**

        is faster than
      - 01 WS-PGM PIC X(08).
        Move ‘PROGRAMA’ to WS-PGM
        CALL WS-PGM - Dynamic Call
  - » Nested calls are faster than static calls.
    - Because When a COBOL program calls a nested program, the CALL is resolved by the compiler without any system intervention.
  - » IS INITIAL on PROGRAM-ID can be very penalizing in terms of time.

Performance Considerations for QSAM files

- QSAM files:
  - » To allow the OS/390 operating system to optimize your block size code
    - BLOCK CONTAINS 0 RECORDS on FD section of COBOL program
    - and BLKSIZE = 0 on DCB in JCL

Performance Considerations for Datatypes

- PACKED-DECIMAL (COMP-3): Use 15 or few digits in the PIC clause to avoid the use of library routines.
- Always code an odd number of digits!
- While performing arithmetic, always use the signed numeric fields.
  - » COBOL performs faster with signed fields than unsigned fields.
- Specify SYNC for BINARY items.
- Use signed data items with 8 or fewer digits – [S9(8)BINARY SYNC]
  - » 9 or more digits is slower
  - » 18 digits is slowest
- Avoid USAGE IS DISPLAY for numeric fields [PIC S99].

Performance Considerations for Constant and Variables

- Constants and Variables
  - Use PIC S9(8) BINARY fields for loop control variables.
  - Initialize constants with a value clause and do not modify them or pass them by reference (Compile will optimize the constants).

Performance Considerations for Tables

- Tables
  - Using variable-length tables is 5% slower than using a fixed-length table.
  - Using a variable-length table that references the first complex ODO(Occurs Depending On)element is 7% slower than using a fixed-length table.

Performance Considerations for Indexes and Subscripts

- Indexes and Subscripts:
  - Indexes are faster than subscripts.
  - If you select subscripts, code S9(8) BINARY SYNC other Usage clauses like COMP,COMP-3 and DISPLAY results in lower performance
  - When referring to tables sequentially, having the leftmost subscript vary the most often can be 50% slower than having the rightmost subscript vary the most often.
  - Binary search on table(using SEARCH ALL) is faster than Linear search(SEARCH) .

Performance Considerations for Compiler Options

- **Compiler Options:**
  - **AWO:** Can result in performance savings, because this option results in fewer calls to data management services to handle I/O.
  - **OPTIMIZE:** Results in more efficient run-time code.
  - **Data(31):** Allocates the buffers in unrestricted storage avoids virtual storage constraint problems.
  - **DYNAM:** Subprograms are easier to maintain, because the application does not have to be link-edited again if a subprogram is changed.
  - **FASTSRT:** Eliminates the overhead of returning to COBOL for OS/390 & VM after each record is processed.
  - **RMODE(ANY):** RMODE(ANY) with NORENT lets the program where the WORKING-STORAGE resides. With RMODE(24), WORKING-STORAGE will be below the 16-MB line. With RMODE(31), the WORKING-STORAGE is above the 16-MB line.

Summary

- ❑ This chapter dealt with the following concepts:
  - o Performance Issues in COBOL programs.
  - o Good COBOL Programming Practices for developing an efficient code.

Test Your Understanding

1. Using binary data items (COMP) to address a table is 30% __________ than using indexes:
   a. slower
   b. Faster
2. Specify _________ Usage for Binary items:
   a. comp
   b. Comp-1
   c. display
   d. Synch
3. Using variable-length tables is 5% ________ than using a fixed-length table:
   a. slower
   b. Faster
4. What is the advantage of using AWO compiler option?
5. Nested calls are faster than static calls. (True/False)

Session 39 and 40: Advanced COBOL

Learning Objectives

After completing this session, you will be able to:

- ❑ Explain the overview of programming for DB2 environment
- ❑ Apply SQL include
- ❑ Test the Return Code of SQL statements
- ❑ Compile Cobol code involving SQL statements using DB2 coprocessor

Overview of Programming for DB2 Environment

The coding of COBOL program will not change irrespective of the need to access a DB2 database.

To retrieve, update, insert, and delete DB2 data and apply other DB2 services, you must implement SQL statements.

To communicate with DB2, the following are needed:

- Delimit SQL statements with EXEC SQL and END-EXEC statements
- Declare a communications area (SQLCA) in the WORKING-STORAGE SECTION
- Declare all host variables used in SQL statements in the WORKING-STORAGE or LINKAGE SECTIONs
- Code any SQL statements needed
- Check the return code from DB2 in the SQLCA to handle exceptional conditions that are indicated

Application of SQL Include: A SQL INCLUDE statement is treated identically to a native COBOL COPY statement.

Example:

```
EXEC SQL INCLUDE name
COPY name
```

You can also use DCLGENs for layouts.

DCLGEN stands for Declarations Generator. It is a facility to generate DB2 SQL data structures in COBOL or PL/I programs.

The library search order for SQL INCLUDE statements is the same SYSLIB concatenation as the compiler uses to resolve COBOL COPY statements that do not specify a library name.

Testing the Return Code of SQL Statements:

- When DB2 finishes executing a SQL statement, DB2 sends a return code in the SQLCA to indicate whether the operation succeeded or failed.
- The program should handle the validation of the return code and take appropriate action.
- DB2 passes the return code back in the SQLCA and not in Register 15. Therefore, the COBOL RETURN-CODE special register might contain an invalid value.
- The COBOL program uses the SQLCA return- code to validate SQL statements execution and COBOL return-code for other cases.

DB2 Coprocessor: When you use the DB2 coprocessor (called SQL statement coprocessor by DB2), the compiler handles your source program containing embedded SQL statements without your having to use a separate precompiled step. When the compiler encounters SQL statements at significant points in the source program, it interfaces with the DB2 coprocessor. This coprocessor takes appropriate actions on the SQL statements and indicates to the compiler what native COBOL statements to generate for them.

Although the use of a separate precompiled step continues to be supported, use of the coprocessor is recommended. Interactive debugging with Debug Tool is enhanced when you use the coprocessor because you only see the SQL statements in the listing (and not the generated COBOL source). However, you must have DB2 for OS/390 Version 7 or later. The DB2 coprocessor is not supported on VM.

Compiling with the DB2 coprocessor generates a DB2 database request module (DBRM) along with the usual COBOL compiler outputs such as object module and listing. The DBRM writes to the data set that you specified on the DBRMLIBB DD statement in the JCL for the COBOL compile step. As input to the DB2 bind process, the DBRM data set contains information about the SQL statements and host variables in the program.

The COBOL compiler listing includes the error diagnostics (such as syntax errors in the SQL statements) that the DB2 coprocessor generates. Certain restrictions on the use of COBOL language that apply when you use the precompiled step do not apply when you use the DB2 coprocessor:

- You can use SQL statements in any nested program. (With the pre-compiler, SQL statements are restricted to the outermost program.)
- You can use SQL statements in copy books.
- REPLACE statements work on SQL statements.

Overview of Programming for CICS Environment

CICS COBOL application programs must be written using the CICS command level interface.

CICS commands are statements you include in the PROCEDURE DIVISION of your application program. They have the following basic format:

```
EXEC CICS command name and command options
END-EXEC
```

CICS Coding Considerations:

Input/ Output:

- CICS handles all input/output between the application program and devices (including terminals).
- Therefore, instead of applying COBOL input/output statements to perform input/output, implement CICS commands.
- While applying the CICS commands, the COBOL statements of OPEN, CLOSE, READ, START, REWRITE, WRITE, DELETE, ACCEPT, or DISPLAY must not be implemented.
- Instead, CICS commands must be used to retrieve, update, insert, and delete data.

Compiler options:

- When coding for CICS, certain compiler options are required while others are recommended or ignored.
- To prepare COBOL program to run under CICS one must use the CICS translator to convert the CICS commands to COBOL statement and compile and link the program to create the executable module.
- When coding for CICS, the required compiler options are:
  - RENT
  - NODYNAM (if the program is translated by the CICS translator)
  - LIB (if the program has a COPY or BASIS statement in it)

Reserved word table:

- The CICS reserved word table supplied by IBM can be used during compilation to flag certain COBOL language elements not supported under CICS.
- COBOL provides an alternate reserved word table (IGYCCICS) specifically for CICS application programs.
- If you use the compiler option WORD(CICS), then COBOL words not supported under CICS are flagged by the compiler with an error message.

Using CICS HANDLE:

- The CICS HANDLE command is used to handle conditions, aids, and abends caused by a COBOL subprogram.
- COBOL does not support the use of the CICS HANDLE commands with the LABEL option to handle conditions, aids, and abends in a program that were caused by another program invoked using the COBOL CALL statement.
- Attempts to perform cross-program branching by using the CICS HANDLE command with the LABEL option will result in a transaction abend.

Coding restrictions:

- Do not apply EXEC, CICS, or END-EXEC for variable names.
- Do not implement the FILE-CONTROL entry in the ENVIRONMENT DIVISION, unless the FILE-CONTROL entry is being used for a SORT statement.
- Do not apply the FILE SECTION of the DATA DIVISION, unless the FILE SECTION is being used for a SORT statement.
- Do not implement parameters specified by user to the main program.

- Do not apply USE declaratives (except USE FOR DEBUGGING).
- Do not implement these COBOL language statements like ACCEPT, CLOSE, DELETE.1111.

Translating CICS commands:

- The CICS translator interprets CICS commands and generates COBOL code.
- The CICS translator takes the source program and converts the EXEC CICS commands to COBOL code.
- The translator replaces each EXEC CICS command with one or more COBOL statements, one of which is a CALL statement.

Compiling CICS code:

- The COBOL compiler compiles the code generated by the CICS translator.
- COBOL programs that use CICS commands must be link-edited with the CICS stub.

Calls under CICS:

- Certain CALL restrictions and requirements must be observed if the program is to run under CICS.
- The NODYNAM compiler option must be used if the COBOL program has been translated by the CICS translator.
- CALL identifier can be used with the NODYNAM compiler option to dynamically call a program. Called programs can contain any function supported by CICS for the language. Dynamically called programs have to be defined in the CICS Program Processing Table (PPT).
- If you are calling a COBOL program that has been translated, then you must pass DFHEIBLK and DFHCOMMAREA as the first two parameters in the CALL statement.

Overview of Object Oriented Programming in COBOL

Advantages of Object-Oriented Programming:

- Object-oriented programming (OOP) is a concept that has the potential for significantly enhancing the quality of all programs.
- This concept is considered by many to be the key to improved productivity in future programs.
- ANSI formed an OO COBOL Task Group to define the OO extensions to the COBOL language as a part of the COBOL 2000+ standard.
- Some programming languages that already implement the OO approach are:
  - SMALLTALK
  - C++
- Keep in mind that the concept of object-oriented programming will be added to the COBOL standard as an extension.
- Thus any new compiler incorporating object oriented techniques will be compatible with previous standards.

Overview of Object-Oriented Programming:

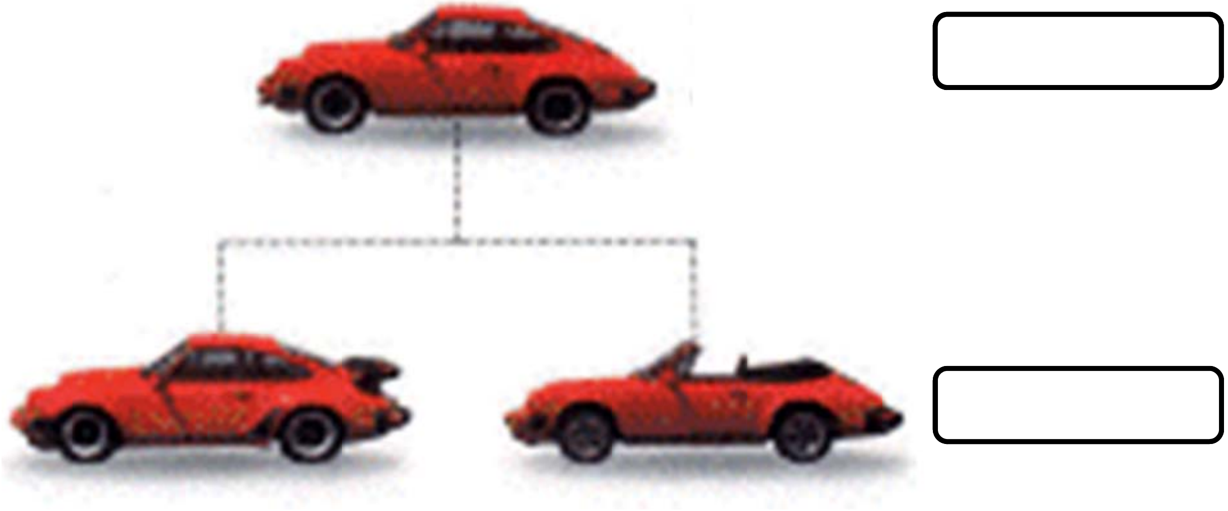
- Data and procedures are combined and stored in a program as units referred to as objects.
- Action occurs when an object receives a message from the user.
- Services or actions are performed by the program as responses to user messages
- Objects can be written so that they share attributes--data and methods--with other objects.
- Objects can have any number of other attributes unique to them.
- Data and procedure components need only be written once and copied to all objects with same attributes.

Overview:

- A class defines a template for a series of similar objects.
- The class can be used to define and create objects that are the same except for the value of their data.
- By defining classes and objects within classes the following takes place:
  - Complex applications will be easier to develop.
  - Programs will become more standardized.
  - Reusable code stored in libraries will reduce duplication of effort, programming and maintenance costs, and errors.
  - Improve programmer productivity
  - Objects can be acted on in any program by responding to the messages provided by the user.

OOP is a different way of writing programs:

- With traditional code, actions are accomplished by procedures that act on data.
- With OOP, objects that include data and procedures are prewritten and acted upon in a user program by messages sent to the objects.
- Objects are ENCAPSULATED:
- This means their data and procedures are hidden behind an INTERFACE.
- One goal of OOP is to develop libraries of objects that can be shared and called into user programs as needed
- CALL and COPY are verbs that can achieve some of the objectives of OOP.
- A CLASS is a group of objects that share attributes which are methods for operating on the data.
- A METHOD is an action achieved by issuing a message to an object.
- A method is really an object’s way of responding to a message.
- Defining classes means placing reusable code in a central location or library.
- An object is called an INSTANCE of a class.
- Objects INHERIT attributes--data and procedures- from their class.
- A class can include a group of objects and may be included in another object called a MEMBER object.



Example-1:

- A Bank-Account may be defined as a class.
- It may have checking-account and savings-account as subclasses, each of which shares data attributes (e.g., account number and balance) and procedures (e.g., calculating interest) defined as part of Bank-Account.

Example-2:

- A Method that can be applied to these classes includes deposit and withdrawal.
- All objects within bank-account can share deposit and withdrawal services.
- That means that the mechanisms used for withdrawing or depositing money will be the same for all objects within the class.
- Each object can have additional methods not inherited from the class but unique to that object.
- That is, process-check-fee can be a method applied to checking-account but not to savings-account.

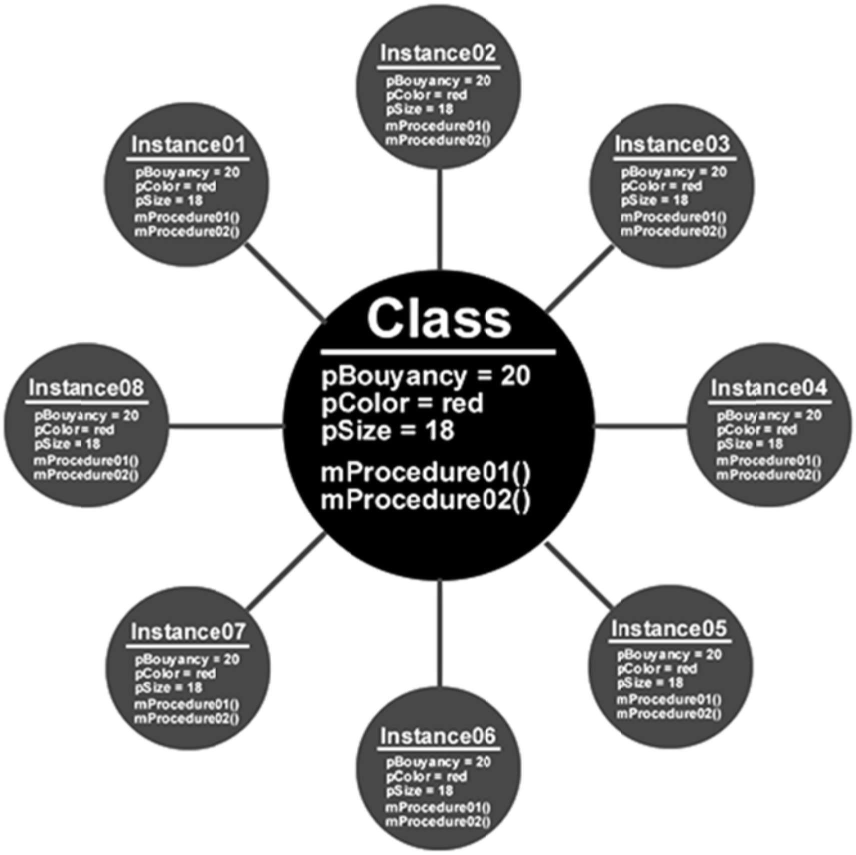
Example-3:

- Data such as account-number can be shared by objects within the class as well.
- This may be made available to users for processing but may also be protected so that users can enter and retrieve them, but not be able to change them.

Classes and their objects consist of data and procedures:

- There are two types of data in a class:
  - INSTANCE VARIABLE-- object data and procedures that are unique for each object in the class.
  - FACTORY--data and procedures that are data shared by all objects in the class. Class is a COBOL reserved word so we use FACTORY. Factory data must be set to its initial value using the INITIALIZE verb or a VALUE clause (interest rate).
- METHODS can be two types:
  - Object methods unique to each object in the class.

- Factory methods shared by all objects in the class.
- Objects are identified by unique names called OBJECT HANDLES



POLYMORPHISM means a method can be implemented differently depending on the object. Withdrawal, for example, may be implemented:

- One way for objects like checking-account within the bank-account class.
- A different way for objects in a credit-card class.

Although the term "polymorphism" itself may be new to you, it has relevance to the current COBOL language.

The READ statement, for example is polymorphic in that it results in different actions depending on whether we are reading from a sequential, indexed, or relative file.

The Interface that links the Method to the object may be different but the service provided will be similar.

An **INTERFACE** is the entire set of messages to which an object can respond along with the parameters required by each message.

The concept of **DATA ABSTRACTION** in OOP encourages programmers to think of data and methods in abstract terms, as fixed objects;

Common objects--data and procedures--should be factored out and located in ancestor classes, sometimes referred to as abstract classes.

ENCAPSULATION--the ability to hide internal details of data and procedures while providing a public interface through a user-defined message

BASE CLASS--a new class from an existing class through INHERITANCE.

Conceptually the implementation of OO in COBOL begins using standard COBOL:

- ❑ The COPY statement enables the copying of class definitions and objects into a program.
- ❑ The CALL statement enables the sending of message to objects to get results.
- ❑ The INVOKE statement may be used also.

A COBOL example:

```
INVOKE ASAVINGSACCOUNT  ‘Withdraw’ USING CUSTOMER-ACCT TRANS-AMT
      RETURNING ACCOUNT
ASAVINGSACCOUNT may be an instance of some class object such as ACCOUNT.
‘Withdraw’  is a method.
USING indicate the parameters to pass to the object.
RETURNING indicate the parameters passed back to the user program.
Often you begin establishing a new instance of an object from a class of
objects.
MYSAVINGSACCOUNT may be an instance  of ASAVINGSACCOUNT.
Defining an instance of an object is called INSTANTIATION.
This may be accomplished by code such as:
    INVOKE ASAVINGSACCOUNT 'New' RETURNING MYSTAVINGSACCOUNT.
```

PERSISTENCE is the ability for changes to be retained after the program is terminated and when the program executes it begins just as it ended.

This is analogous to a "Save on EXIT" menu item in some programs.

Compiler options

- ❑ Helps to control the compilation
- ❑ Specify these options in:
  - o PARM field of the JCL
  - o CBL or PROCESS statement in program
- ❑ Most of the options come in pairs like XREF/NOXREF
- ❑ Some options have sub-parameters like LINECOUNT (44)
- ❑ Watch out that there are default options when compiler is installed
- ❑ Precedence of Compiler Options:
  - o **Level 1:** (Highest precedence) Installation defaults, fixed by your installation

  - **Level 2:** Those on PROCESS (or CBL) stmt
  - **Level 3:** Those on the JCL PARM
  - **Level 4:** (Lowest precedence) Installation defaults, but not fixed.
- **Object code generation:** COMPILE, CMPR2, DECK, NAME, OBJECT
- **Use of virtual storage:** BUFSIZE, SIZE
- Object code execution:
  - ADV, ADW, DATA, DYNAM, FASTSRT, OPTIMIZE, OUTDD, NUMPROC
  - RENT, RESIDENT, SSRANGE, TRUNC, ZWB
- Maps, listing, and diagnostics:
  - DUMP, FLAG, FLAGMIG, FLAGSAA, FLAGSTD, LANGUAGE, LINECOUNT, LIST, MAP, NUMBER, OFFSET, SEQUENCE, SOURCE, SPACE, TERMINAL, VBREF, XREF
- **Debugging:** FDUMP, TEST
- **Literal delimiters:** APOST, QUOTE, DBCS
- **Reserved word list:** WORD
- **Processing of COPY ETC:** LIB
- **Provision of User Exit:** EXIT
- ADV / NOADV:
  - Default is: ADV
  - ADV has meaning only if you use WRITE . . . ADVANCING in your source code.
  - With ADV in effect, the compiler adds 1 byte to the record length to account for the printer control character. Use NOADV if you have already adjusted your record length to include 1 byte for the printer control character
- CMPR2 / NOCMPR2:
  - Default is: NOCMPR2
  - Use CMPR2 when you want the compiler to generate code that is compatible with code generated by VS COBOL II Release 2. Valid COBOL source programs that compiled successfully under VS COBOL II Release 2 will also compile successfully under IBM COBOL for MVS & VM with CMPR2 in effect and will provide compatible results.
- COMPILE / NOCOMPILE (W/E/S):
  - Default is: NOCOMPILE (S)
  - Abbreviations are: C / NOC
  - Use the COMPILE option only if you want to force full compilation even in the presence of serious errors. All diagnostics and object code will be generated. Do not try to run the object code generated if the compilation resulted in serious errors--the results could be unpredictable or an abnormal termination could occur.
- DATA (24/31):
  - Default is: DATA(31)
  - Language Environment provides services that control the storage used at run time. IBM COBOL for MVS & VM uses these services for all storage requests.
  - For reentrant programs, the DATA (24|31) compiler option and the HEAP run-time option control whether storage for dynamic data areas (such as WORKING-STORAGE section and FD record areas) is obtained from below the 16-megabyte line or from unrestricted storage.

- When you specify the run-time option HEAP (BELOW), the DATA (24|31) compiler option has no effect; the storage for all dynamic data areas is allocated from below the 16-megabyte line. However, with HEAP (ANYWHERE) as the run-time option, storage for dynamic data areas is allocated from below the line if you compiled the program with the DATA (24) compiler option or from unrestricted storage if you compiled with the DATA (31) compiler option.
- Specify the DATA (24) compiler option for programs running in 31-bit addressing mode that is passing data parameters to programs in 24-bit addressing mode. This ensures that the data will be addressable by the called program.- With NORENT compiler option, DATA(31) has no effect

- DECK / NODECK:
  - Default is: NODECK
  - Abbreviations are: D / NOD
  - Use DECK to produce object code in the form of 80-column card images. If you use the DECK option, be certain that SYSPUNCH is defined in your JCL for compilation.
  - SYSPUNCH ddname must for DECK opt
- DYNAM / NODYNAM:
  - Default is: NODYNAM
  - Abbreviations are: DYN / NODYN
  - Use DYNAM to cause separately compiled programs invoked through the CALL literal statement to be loaded dynamically at run time. DYNAM causes dynamic loads (for CALL) and deletes (for CANCEL) of separately compiled programs at object time. Any CALL identifier statements that cannot be resolved in your program are also handled as dynamic calls.
  - DYNAM forced RESIDENT option
- FLAG (x,y) / NOFLAG:
  - Default is: FLAG (I)
  - Abbreviations are: F / NOF

    x: I, W, E, S, or U

    y: I, W, E, S, or U
  - Use FLAG (x) to produce diagnostic messages for errors of a severity level x or above at the end of the source listing. Use FLAG (x, y) to produce diagnostic messages for errors of severity level x or above at the end of the source listing, with error messages of severity y and above to be embedded directly in the source listing. The severity coded for y must not be lower than the severity coded for x. To use FLAG (x, y), you must also specify the SOURCE compiler option.
  - Error messages in the source listing are set off by embedding the statement number in an arrow that points to the message code. The message code is then followed by the message text
  - Use NOFLAG to suppress error flagging. NOFLAG will not suppress error messages for compiler options.
  - Control listing of error messages
- LIB / NOLIB:
  - Default is: NOLIB
  - Abbreviation is: None

- If your program uses COPY, BASIS, or REPLACE statements, you need to specify the LIB compiler option. In addition, for COPY and BASIS statements, include in your JCL DD statements for the library or libraries from which the compiler can take the copied code, and also include a JCL DD statement to allocate SYSUT5.- Allocate - SYSLIB and SYSUT5 ddnames

- ❑ LIST / NOLIST:
  - Default is: NOLIST
  - Use LIST to produce a listing of the assembler-language expansion of your source code. You will also get these in your output listing:

    Global tables

    Literal pools

    Information about Working-Storage

    Size of the program's Working-Storage and its location in the object code if the program is compiled with the NORENT option
  - LIST and OFFSET are mutually exclusive and if both given LIST ignored
- ❑ MAP / NOMAP:
  - Default is: NOMAP
  - Use MAP to produce a listing of the items you defined in the DATA DIVISION. Map output includes:

    DATA DIVISION map

    Global tables

    Literal pools

    Nested program structure map, and program attributes

    Size of the program's Working-Storage and its location in the object code if the program is compiled with the NORENT option

    By selecting the MAP option, you can also print an embedded MAP report in the source code listing. The condensed MAP information is printed to the right of data-name definitions in the FILE SECTION, WORKING-STORAGE SECTION, and LINKAGE SECTION of the DATA DIVISION.
- ❑ NUMBER / NONUMBER:
  - Default is: NONUMBER
  - Abbreviations are: NUM / NONUM
  - Use NUMBER if you have line numbers in your source code and want those numbers to be used in error messages and MAP, LIST, and XREF listings.
  - If you request NUMBER, columns 1 through 6 are checked to make sure that they contain only numbers, and the sequence is checked according to numeric collating sequence. When a line number is found to be out of sequence, the compiler assigns to it a line number with a value one number higher than the line number of the preceding statement. Sequence checking continues with the next statement, based on the newly assigned value of the previous line.
  - If you use COPY statements and NUMBER is in effect, be sure that your source program line numbers and the COPY member line numbers are coordinated. If you are doing a batch compilation and LIB and NUMBER are in effect, all programs in the batch compile will be treated as a single input file. The sequence numbers of the entire input file must be in ascending order.
  - Use NONUMBER if you do not have line numbers in your source code, or if you want the compiler to ignore the line numbers you do have in your source code. With NONUMBER in effect, the compiler generates line numbers for your source statements and uses those numbers as references in listings.

- NUMPROC (PFD or NOPFD or MIG):
  - Default is: NUMPROC (NOPFD)
  - Use NUMPROC (NOPFD) if you want the compiler to perform invalid sign processing. This option is not as efficient as NUMPROC (PFD); object code size will be increased, and there could be an increase in run-time overhead to validate all signed data.
  - NUMPROC (PFD) is a performance option that can be used to bypass invalid sign processing. Using NUMPROC (PFD) can affect class tests for numeric data
  - Use NUMPROC (MIG) to aid in-migrating OS/VS COBOL programs to IBM COBOL for MVS & VM. When NUMPROC (MIG) is in effect, the following processing occurs:

    Preferred signs are created only on the output of MOVE statements and arithmetic operations.

    No explicit sign repair is done on input.

    Some implicit sign repair might occur during conversion.

    Numeric comparisons are performed by a decimal compare, not a logical compare.
- OBJECT / NOOBJECT:
  - Default is: OBJECT
  - Abbreviations are: OBJ / NOOBJ
  - Use OBJECT to place the generated object code on disk or tape to be later used as input for the linkage editor.
  - If you specify OBJECT, include a SYSLIN DD statement in your JCL for compilation.
  - The only difference between DECK and OBJECT is in the routing of the data sets:

    DECK output goes to the data set associated with SYSPUNCH ddname.

    OBJECT output goes to the data set associated with SYSLIN ddname.
  - NOOBJECT & TEST - mutually exclusive
- OFFSET / NOOFFSET:
  - Default is: NOOFFSET
  - Abbreviations are: OFF / NOOFF
  - Use OFFSET to produce a condensed PROCEDURE DIVISION listing. With OFFSET, the condensed PROCEDURE DIVISION listing will contain line numbers, statement references, and the location of the first instruction generated for each statement. In addition, the following are produced:

    Global tables

    Literal pools

    Size of the program's Working Storage, and its location in the object code if the program is compiled with the NORENT option.
  - OFFSET & LIST are mutually exclusive.
- OPTIMIZE / NOOPTIMIZE:
  - The COBOL optimizer is activated when you use the OPTIMIZE compiler option. The purpose of the OPTIMIZE compiler option is to do the following:
  - Eliminate unnecessary transfers of control or simplify inefficient branches, including those generated by the compiler that is not evident from looking at the source program.

- Simplify the compiled code for both a PERFORM statement and a CALL statement to a contained (nested) program. Where possible, the optimizer places the statement inline, eliminating the need for linkage code.
- This optimization, known as procedure integration, is further discussed in "PERFORM Procedure Integration" If procedure integration cannot be done, the optimizer uses the simplest linkage possible (perhaps as few as two instructions) to get to and from the called program.
- Eliminate duplicate computations (such as subscript computations and repeated statements) that have no effect on the results of the program.
- Eliminate constant computations by performing them when the program is compiled.
- Eliminate constant conditional expressions.
- Aggregate moves of contiguous items (such as those that often occur with the use of MOVE CORRESPONDING) into a single move. Both the source and target must be contiguous for the moves to be aggregated.
- Delete from the program, and identify with a warning message, code that can never be performed (unreachable code elimination).
- The FULL suboption requests that the compilers discard any unreferenced data items from the DATA DIVISION, and suppress generation of code to initialize these data items to their VALUE clauses.
- For unit testing your programs, you might find it easier to debug code that has not been optimized. But when the program is ready for final test, specify OPTIMIZE, so that the tested code and the production code are identical. You might also want to use the option during development, if a program is used frequently without recompilation. However, the overhead for OPTIMIZE might outweigh its benefits if you re-compile frequently, unless you are using the assembler language expansion (LIST option) to fine tune your program.
- OPTIMIZE & TEST - mutually exclusive

❑ **OUTDD (ddname):** To get run-time DISPLAY output on a data set other than SYSOUT

❑ QUOTE / APOST:

- Default is: QUOTE
- Abbreviations are: Q / APOST
- Use QUOTE if you want the figurative constant [ALL] QUOTE or [ALL] QUOTES to represent one or more quotation mark (") characters.
- Use APOST if you want the figurative constant [ALL] QUOTE or [ALL] QUOTES to represent one or more apostrophe (') characters.
- **Note:** Either quotes or apostrophes can be used as literal delimiters, regardless of whether the APOST or QUOTE option is in effect. The delimiter character used as the opening delimiter for a literal must be used as the closing delimiter for that literal.

❑ RENT / NORENT:

- Object code is to be re-entrant
- Have RESIDENT option for all programs in a unit unless the run-time option MIXRES has been specified.

❑ SOURCE / NOSOURCE:

- Default is: SOURCE
- Abbreviations are: S / NOS
- Use SOURCE to get a listing of your source program. This listing will include any statements embedded by PROCESS or COPY statements.

  - SOURCE must be specified if you want embedded messages in the source listing.
  - Use NOSOURCE to suppress the source code from the compiler output listing.
- **SPACE (1/2/3):** To select single, double, or triple spacing in your source code listing.
- TEST / NOTEST:
  - Default is: NOTEST
  - Use TEST to produce object code that enables Debug Tool to perform batch and interactive debugging. The amount of debugging support available depends on which TEST suboptions you use. The TEST option also allows you to request that symbolic variable be included in the formatted dump produced by Language Environment.
  - Use NOTEST if you do not want to generate object code with debugging information and do not want the formatted dump to include symbolic variables.
  - TEST has two sub options; you can specify both, just one of the sub options, or neither of the sub options:
  - **Hook:** The hook-location sub option controls where compiled-in hooks will be generated to provide information to the debugger.
  - **NONE:** No hooks will be generated.
  - **BLOCK:** Hooks will be generated at all program entry and exit points.
- **PATH:** Hooks will be generated at all program entry and exit points and at all path points. A path point is anywhere in a program where the logic flow is not necessarily sequential or can change. Some examples of path points are IF-THEN-ELSE constructs, PERFORM loops, ON SIZE ERROR phrases, and CALL statements.
- **STMT:** Hooks will be generated at every statement and label, as well as at all program entry and exit point. In addition, if the DATEPROC option is in effect, hooks will be generated at all date processing statements.
- **ALL:** Hooks will be generated at all statements, all path points, and at all program entry and exit points. In addition, if the DATEPROC option is in effect, hooks will be generated at all date processing statements.
- Symbol: The symbol-table sub option controls whether dictionary tables will be generated.
  - **SYM**: Dictionary and calculation tables will be generated.
  - **NOSYM**: Dictionary and calculation tables will not be generated.
- SSRANGE / NOSSRANGE:
  - Default is: NOSSRANGE
  - Abbreviations are: SSR / NOSSR
  - Use SSRANGE to generate code that checks if subscripts (including ALL subscripts) or indexes try to reference an area outside the region of the table. Each subscript or index is not individually checked for validity; rather, the effective address is checked to ensure that it does not cause a reference outside the region of the table. Variable-length items will also be checked to ensure that the reference is within their maximum defined length.
  - Reference modification expressions will be checked to ensure that:

    The reference modification starting position is greater than or equal to 1.

    The reference modification starting position is not greater than the current length of the subject data item.

    The reference modification length value (if specified) is greater than or equal to 1.

The reference modification starting position and length value (if specified) do not reference an area beyond the end of the subject data item.

Use it while testing

- TRUNC (OPT or STD or BIN):
  - Default is: TRUNC (STD)
  - **TRUNC (STD):** Use TRUNC (STD) to control the way arithmetic fields are truncated during MOVE and arithmetic operations. TRUNC (STD) applies only to USAGE BINARY receiving fields in MOVE statements and arithmetic expressions. When TRUNC (STD) is in effect, the final result of an arithmetic expression, or the sending field in the MOVE statement, is truncated to the number of digits in the PICTURE clause of the BINARY receiving field.
- **TRUNC (OPT):** TRUNC (OPT) is a performance option. When TRUNC (OPT) is specified, the compiler assumes that the data conforms to PICTURE and USAGE specifications of the USAGE BINARY receiving fields in MOVE statements and arithmetic expressions. The results are manipulated in the most optimal way, either truncating to the number of digits in the PICTURE clause, or to the size of the binary field in storage (halfword, fullword, or double word).
- **TRUNC (BIN):** The TRUNC (BIN) option applies to all COBOL language that processes USAGE BINARY data. When TRUNC (BIN) is in effect:
  - BINARY receiving fields are truncated only at half word, fullword, or doubleword boundaries.
  - BINARY sending fields are handled as halfwords, fullwords, or doublewords when the receiver is numeric; TRUNC (BIN) has no effect when the receiver is not numeric.
  - The full binary content of the field is significant.
  - DISPLAY will convert the entire content of the binary field, with no truncation.
- VBREF / NOVBREF:
  - Default is: NOVBREF
  - Use VBREF to get a cross-reference among all verb types used in the source program and the line numbers in which they are used. VBREF also produces a summary of how many times each verb was used in the program.
  - Use NOVBREF for more efficient compilation.
- WORD (xxxx) / NOWORD:
  - Default is: NOWORD
  - Abbreviations are: WD / NOWD
  - xxxx: Are the ending characters of the name of the reserved word table (IGYCxxxx) to be used in your compilation. IGYC are the first 4 standard characters of the name, and xxxx can be 1 to 4 characters in length.
  - Use WORD(xxxx) to specify that an alternate reserved word table is to be used during compilation.
  - Alternate reserved word tables provide changes to the IBM-supplied default reserved word table.
- XREF(FULL or SHORT) / NOXREF:
  - Default is: NOXREF
  - Abbreviations are: X / NOX
  - You can choose XREF, XREF (FULL), or XREF (SHORT).

  - Use XREF to get a sorted cross-reference listing. EBCDIC data-names and procedure-names are listed in alphanumeric order. DBCS data-names and procedure-names are listed based on their physical order in the program, and appear before the EBCDIC data-names and procedure-names, unless the DBCSXREF installation option is selected with a DBCS ordering program. In this case, DBCS data-names and procedure-names are ordered as specified by the DBCS ordering program.
  - Also included is a section listing all the program names that are referenced in your program and the line number where they are defined. External program names are identified as such.
  - If you use XREF and SOURCE, cross-reference information will also be printed on the same line as the original source in the listing. Line number references or other information, will appear on the right hand side of the listing page. On the right of source lines that reference intrinsic functions, the letters 'IFN' will appear with the line numbers of the location where the function's arguments are defined. Information included in the embedded references lets you know if an identifier is undefined or defined more than once. (UND or DUP will be printed); if an item is implicitly defined (IMP), such as special registers or figurative constants; and if a program name is external (EXT).
  - If you use XREF and NOSOURCE, you'll get only the sorted cross-reference listing.
  - XREF (SHORT) will print only the explicitly referenced variables in the cross-reference listing. XREF (SHORT) applies to DBCS data names and procedure-names as well as EBCDIC names.
  - NOXREF suppresses this listing.

- ZWB / NOZWB:
  - Default is: ZWB
  - With ZWB, the compiler removes the sign from a signed external decimal (DISPLAY) field when comparing this field to an alphanumeric elementary field during execution.
  - If the external decimal item is a scaled item (contains the symbol 'P' in its PICTURE character-string), its use in comparisons is not affected by ZWB. Such items always have their sign removed before the comparison is made to the alphanumeric field.
  - ZWB affects how the program runs; the same COBOL source program can give different results, depending on the option setting.
  - Use NOZWB if you want to test input numeric fields for SPACES.

Compiler option - Coding Example

```
//COB   EXEC    PGM=IGYCRCTL,
     // PARM='LIST, NOCOMPILE (S), OBJECT'
CBL LIST,NOCOMPILE(S),OBJECT
      IDENTIFICATION DIVISION.
      PROGRAM-ID. myprog.
PROCESS LIST,NOCOMPILE(S),OBJECT
      IDENTIFICATION DIVISION.
      PROGRAM-ID. myprog
```

Summary

Overview of Programming for DB2 Environment:

- ❑ The DB2 tables can be accessing from Cobol programs by using SQL statements
- ❑ The pre-defined layouts of the DB2 tables may also be plugged into the Cobol code
- ❑ DB2 Bind is process that builds "access paths" to DB2 tables.
- ❑ A DBRM is a DB2 component created by the DB2 pre-compiler containing the SQL source statements extracted from the application program. DBRMs are input to the bind process.
- ❑ A bind uses the Database Request Modules(s) (DBRM(s)) from the DB2 pre-compile step as input and produces an application plan.

Overview of Programming for CICS Environment:

- ❑ Customer Information Control System (CICS) is an on-line teleprocessing system developed by IBM.
- ❑ CICS allows:
  - o Data to be transmitted from the terminal to the host computer, have the data processed, access files/databases, and
  - o then have data to be transmitted from the terminal to the host computer, have the data processed, access files/databases, and then have data transmitted back to the terminal
- ❑ Some of the new functionality includes:
  - o Expanded features for the system programmer
  - o Improved above the line storage utilization
  - o New options for many CICS commands
  - o Improved cross-platform communication facilities

Overview of Object Oriented Programming in COBOL:

- ❑ **Structured Programming:** Structured programming is a Logical way of programming with the functionalities divided into modules and is coded logically.
- ❑ **Object Oriented Programming (OOP):** OOP is a natural way of programming by identifying the objects first, and then writing functions, procedures around the objects.
- ❑ The following OO Concepts play a crucial role in code optimization and increase in productivity:
  - o Polymorphism
  - o Encapsulation
  - o Inheritance

Significance of Compiler options in COBOL:

- ❑ Compilation can be controlled by using compiler options or compiler-directing statements (compile directives)
- ❑ Compiler options can affect the following aspects of the program:
  - o Source language
  - o Debugging and diagnostics
  - o Date processing
  - o Maps and listings

  - o Object deck generation
  - o Object code control
  - o Virtual storage usage
- ❑ The default options that were set up when the compiler was installed are in effect unless they are overridden with other options.
- ❑ The DYNAM, FASTSRT, NUMPROC, OPTIMIZE, RENT, SSRANGE, TEST, and TRUNC compiler options affect run-time performance.

Test Your Understanding

1. ______ is a DB2 component created by the DB2 pre-compiler containing the SQL source statements
2. ____________ handles your source program containing embedded SQL statements without your having to use a separate pre-compile step
3. Dynamically called programs have to be defined in the CICS _________ Table.
4. _____________ means a method can be implemented differently depending on the object.
5. The concept of ___________ in OOP encourages programmers to think of data and methods in abstract terms, as fixed objects.
6. ________ is used if you want the figurative constant __________ to represent one or more quotation mark (") characters.

Appendix 1

File Status Codes and Meaning

- 2 char code
  - Status key 1
  - Status key 2
- Status key 1 tells the type of error
- Status key 2 gives the detail
- **Status key 1**
  - 0 Successful Completion
  - 1 At end condition
  - 2 Invalid key condition
  - 3 Permanent error condition
  - 4 Logic error
  - 9 Implementor defined (VSAM)
- Successful Completion-0
  - 0 No further info
  - 1 Duplicate - Read, (Re) Write
  - 2 Length error for Fixed files
  - 5 Optional file not present
- At end condition-1
  - 0 End of file
  - 4 Relative key > max possible
- Invalid Key-2
  - 1 Ascending key violation/Rewrite change primary Key
  - 2 Duplicate key when not allowed (Primary, Alternate or relative key)
  - 3 Record not found
  - 4 Write beyond boundary
- Permanent error-3
  - 0 no further info
  - 4 Due to boundary violation
  - 5 File not available (ddname not found/defined)
  - 7 Invalid open mode on open
  - 8 Open of file closed with lock
  - 9 DCB conflict during open
- Logic error-4
  - 1 Open of file in open status
  - 2 Close of file not in open stat
  - 3 Rewrite or Delete after unsuccessful Read
  - 4 Rewrite changed record len or Write record len > max
  - 5 No position for Read next
  - 7 Read on file not opened for Input or I-O

  - 8 Write on file not opened for Output or I-O or Extend
  - 9 Rewrite on file not opened for I-O
- Implementor def- 9(VSAM)
  - 0 No further info
  - 1 Password failure
  - 2 Logic error
  - 3 Resource not available
  - 4 No file position for cmpr2 opt
  - 5 Invalid/Incomplete file info
  - 6 No DDname found
  - 7 Open successful; File verified

Glossary

A

Abbreviated combined relation condition: The combined condition that results from the explicit omission of a common subject or a common subject and common relational operator in a consecutive sequence of relation conditions.

Abend: Abnormal termination of program.

Access mode: The manner in which records are to be operated upon within a file.

Actual decimal point: The physical representation, using the decimal point characters period (.) or comma (,), of the decimal point position in a data item.

Alphabet-name: A user-defined word, in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION that assigns a name to a specific character set and/or collating sequence.

Alphabetic character: A letter or a space character.

Alphanumeric character: Any character in the computer's character set.

Alphanumeric-edited character: A character within an alphanumeric character-string that contains at least one B, 0 (zero), or / (slash).

Alphanumeric function: A function whose value is composed of a string of one or more characters from the computer's character set.

Alternate record key: A key, other than the prime record key, whose contents identify a record within an indexed file.

ANSI (American National Standards Institute): An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States.

Argument: An identifier, a literal, an arithmetic expression, or a function-identifier that specifies a value to be used in the evaluation of a function.

Arithmetic expression: An identifier of a numeric elementary item, a numeric literal, such identifiers and literals separated by arithmetic operators, two arithmetic expressions separated by an arithmetic operator, or an arithmetic expression enclosed in parentheses.

Arithmetic operation: The process caused by the execution of an arithmetic statement, or the evaluation of an arithmetic expression, that results in a mathematically correct solution to the arguments presented.

Arithmetic operator: A single character, or a fixed two-character combination that belongs to the following set:

Character	**Meaning**
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation

Arithmetic statement: A statement that causes an arithmetic operation to be executed. The arithmetic statements are the ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT statements.

Array: In Language Environment, an aggregate consisting of data objects, each of which may be uniquely referenced by subscripting. Roughly analogous to a COBOL table.

Ascending key: A key upon the values of which data is ordered, starting with the lowest value of the key up to the highest value of the key, in accordance with the rules for comparing data items.

ASCII: American National Standard Code for Information Interchange. The standard code, using a coded character set consisting of 7-bit coded characters (8 bits including parity check), used for information interchange between data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters.

Assignment-name: A name that identifies the organization of a COBOL file and the name by which it is known to the system.

Assumed decimal point: A decimal point position that does not involve the existence of an actual character in a data item. The assumed decimal point has logical meaning with no physical representation.

AT END condition: A condition caused:

During the execution of a READ statement for a sequentially accessed file, when no next logical record exists in the file, or when the number of significant digits in the relative record number is larger than the size of the relative key data item, or when an optional input file is not present.

During the execution of a RETURN statement, when no next logical record exists for the associated sort or merge file.

During the execution of a SEARCH statement, when the search operation terminates without satisfying the condition specified in any of the associated WHEN phrases.

B

Big-endian: Default format used by the mainframe and the AIX workstation to store binary data. In this format, the least significant digit is on the highest address. Compare with "little-endian."

Binary item: A numeric data item represented in binary notation (on the base 2 numbering system). Binary items have a decimal equivalent consisting of the decimal digits 0 through 9, plus an operational sign. The leftmost bit of the item is the operational sign.

Binary search: A dichotomizing search in which, at each step of the search, the set of data elements is divided by two; some appropriate action is taken in the case of an odd number.

Block: A physical unit of data that is normally composed of one or more logical records. For mass storage files, a block may contain a portion of a logical record. The size of a block has no direct relationship to the size of the file within which the block is contained or to the size of the logical record(s) that are either contained within the block or that overlap the block. The term is synonymous with physical record.

Breakpoint: A place in a computer program, usually specified by an instruction, where its execution may be interrupted by external intervention or by a monitor program.

Btrieve: A key-indexed record management system that allows applications to manage records by key value, sequential access method, or random access method. IBM COBOL supports COBOL sequential and indexed file I-O language through Btrieve.

Buffer: A portion of storage used to hold input or output data temporarily.

Built-in function: See "intrinsic function."

Byte: A string consisting of a certain number of bits, usually eight, treated as a unit, and representing a character.

C

Callable services: In Language Environment, a set of services that can be invoked by a COBOL program using the conventional Language Environment-defined call interface, and usable by all programs sharing the Language Environment conventions.

Called program: A program that is the object of a CALL statement.

Calling program: A program that executes a CALL to another program.

Case structure: A program processing logic in which a series of conditions is tested in order to make a choice between a numbers of resulting actions.

Cataloged procedure: A set of job control statements placed in a partitioned data set called the procedure library (SYS1.PROCLIB). You can use cataloged procedures to save time and reduce errors coding JCL.

Century window: A century window is a 100-year interval within which any 2-digit year is unique. There are several types of century window available to COBOL programmers:

For windowed date fields, the YEARWINDOW compiler option

For windowing intrinsic functions DATE-TO-YYYYMMDD, DAY-TO-YYYYDDD, and YEAR-TO-YYYY, it is specified by argument-2

For Language Environment date and time callable services, it is specified in CEESCEN

Character: The basic indivisible unit of the language.

Character position: The amount of physical storage required to store a single standard data format character described as USAGE IS DISPLAY.

Character set: All the valid characters for a programming language or a computer system.

Character-string: A sequence of contiguous characters that form a COBOL word, a literal, a PICTURE character-string, or a comment-entry. Must be delimited by separators.

Checkpoint: A point at which information about the status of a job and the system can be recorded so that the job step can be later restarted.

Class: The entity that defines common behavior and implementation for zero, one, or more objects. The objects that share the same implementation are considered to be objects of the same class.

Class condition: The proposition, for which a truth value can be determined, that the content of an item is wholly alphabetic, is wholly numeric, or consists exclusively of those characters listed in the definition of a class-name.

Class Definition: The COBOL source unit that defines a class.

Class identification entry: An entry in the CLASS-ID paragraph of the IDENTIFICATION DIVISION which contains clauses that specify the class-name and assign selected attributes to the class definition.

Class-name: A user-defined word defined in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION that assigns a name to the proposition for which a truth value can be defined, that the content of a data item consists exclusively of those characters listed in the definition of the class-name.

Class object: The run-time object representing a SOM class.

Clause: An ordered set of consecutive COBOL character-strings whose purpose is to specify an attribute of an entry.

CMS (Conversational Monitor System): A virtual machine operating system that provides general interactive, time-sharing, problem solving, and program development capabilities, and that operates only under the control of the VM/SP control program.

COBOL character set: The complete COBOL character set consists of the characters listed below:

Character	Meaning
0,1...,9	digit
A,B,...,Z	uppercase letter
a,b,...,z	lowercase letter
º	space
+	plus sign

-	minus sign (hyphen)
*	asterisk
/	slant (virgule, slash)
=	equal sign
$	default currency sign
,	comma (decimal point)
;	semicolon
.	period (decimal point, full stop)
"	quotation mark
(	left parenthesis
)	right parenthesis
>	greater than symbol
<	less than symbol
:	colon

COBOL DLL program: See "DLL program."

COBOL word: See "word."

Code page: An assignment of graphic characters and control function meanings to all code points; for example, assignment of characters and meanings to 256 code points for 8-bit code, assignment of characters and meanings to 128 code points for 7-bit code.

Collating sequence: The sequence in which the characters that are acceptable to a computer are ordered for purposes of sorting, merging, comparing, and for processing indexed files sequentially.

Column: A character position within a print line. The columns are numbered from 1, by 1, starting at the leftmost character position of the print line and extending to the rightmost position of the print line.

Combined condition: A condition that is the result of connecting two or more conditions with the AND or the OR logical operator.

Comment-entry: An entry in the IDENTIFICATION DIVISION that may be any combination of characters from the computer's character set.

Comment line: A source program line represented by an asterisk (*) in the indicator area of the line and any characters from the computer's character set in area A and area B of that line. The comment line serves only for documentation in a program. A special form of comment line represented by a slant (/) in the indicator area of the line and any characters from the computer's character set in area A and area B of that line causes page ejection prior to printing the comment.

Common program: A program which, despite being directly contained within another program, may be called from any program directly or indirectly contained in that other program.

Compatible date field: The meaning of the term "compatible," when applied to date fields, depends on the COBOL division in which the usage occurs.

Compile: (1) To translate a program expressed in a high-level language into a program expressed in an intermediate language, assembly language, or a computer language. (2) To prepare a machine language program from a computer program written in another programming language by making use of the overall logic structure of the program, or generating more than one computer instruction for each symbolic statement, or both, as well as performing the function of an assembler.

Compile time: The time at which a COBOL source program is translated, by a COBOL compiler, to a COBOL object program.

Compiler: A program that translates a program written in a higher level language into a machine language object program.

Compiler directing statement: A statement that specifies actions to be taken by the compiler during processing of a COBOL source program. Compiler directives are contained in the COBOL source program. Thus, you can specify different sub-options of the directive within the source program by using multiple compiler directive statements in the program.

Complex condition: A condition in which one or more logical operators act upon one or more conditions. (See also "negated simple condition," "combined condition," and "negated combined condition.")

Computer-name: A system-name that identifies the computer upon which the program is to be compiled or run.

Condition: An exception that has been enabled, or recognized, by Language Environment and thus is eligible to activate user and language condition handlers. Any alteration to the normal programmed flow of an application. Conditions can be detected by the hardware/operating system and results in an interrupt. They can also be detected by language-specific generated code or language library code.

Condition: A status of a program at run time for which a truth value can be determined. Where the term 'condition' (condition-1, condition-2,...) appears in these language specifications in or in reference to 'condition' (condition-1, condition-2,...) of a general format, it is a conditional expression consisting of either a simple condition optionally parenthesized, or a combined condition consisting of the syntactically correct combination of simple conditions, logical operators, and parentheses, for which a truth value can be determined.

Conditional expression: A simple condition or a complex condition specified in an EVALUATE, IF, PERFORM, or SEARCH statement. (See also "simple condition" and "complex condition.")

Conditional phrase: A conditional phrase specifies the action to be taken upon determination of the truth value of a condition resulting from the execution of a conditional statement.

Conditional statement: A statement specifying that the truth value of a condition is to be determined and that the subsequent action of the object program is dependent on this truth value.

Conditional variable: A data item one or more values of which has a condition-name assigned to it.

Condition-name: A user-defined word that assigns a name to a subset of values that a conditional variable may assume; or a user-defined word assigned to a status of an implementor defined switch or device. When 'condition-name' is used in the general formats, it represents a unique data item reference consisting of a syntactically correct combination of a 'condition-name', together with qualifiers and subscripts, as required for uniqueness of reference.

Condition-name condition: The proposition, for which a truth value can be determined, that the value of a conditional variable is a member of the set of values attributed to a condition-name associated with the conditional variable.

CONFIGURATION SECTION: A section of the ENVIRONMENT DIVISION that describes overall specifications of source and object programs and class definitions.

CONSOLE: A COBOL environment-name associated with the operator console.

Contiguous items: Items that are described by consecutive entries in the Data Division, and that bear a definite hierarchic relationship to each other.

Copybook: A file or library member containing a sequence of code that is included in the source program at compile time using the COPY statement. The file can be created by the user, supplied by COBOL, or supplied by another product.

CORBA: The Common Object Request Broker Architecture established by the Object Management Group. IBM's Interface Definition Language used to describe the interface for SOM classes is fully compliant with CORBA standards.

Counter: A data item used for storing numbers or number representations in a manner that permits these numbers to be increased or decreased by the value of another number, or to be changed or reset to zero or to an arbitrary positive or negative value.
cross-reference listing. The portion of the compiler listing that contains information on where files, fields, and indicators are defined, referenced, and modified in a program.

Currency sign value: A character-string that identifies the monetary units stored in a numeric-edited item. Typical examples are '$', 'USD', and 'EUR'. A currency sign value can be defined by either the CURRENCY compiler option or the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph of the Environment Division. If the CURRENCY SIGN clause is not specified and the NOCURRENCY compiler option is in effect, the dollar sign ($) is used as the default currency sign value. See also "currency symbol."

Currency symbol: A character used in a PICTURE clause to indicate the position of a currency sign value in a numeric-edited item. A currency symbol can be defined by either the CURRENCY compiler option or by the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph of the Environment Division. If the CURRENCY SIGN clause is not specified and the NOCURRENCY compiler option is in effect, the dollar sign ($) is used as the default currency sign value and currency symbol. Multiple currency symbols and currency sign values can be defined. See also "currency sign value."

Current record: In file processing, the record that is available in the record area associated with a file.

Current volume pointer: A conceptual entity that points to the current volume of a sequential file.

D

Data clause: A clause, appearing in a data description entry in the DATA DIVISION of a COBOL program that provides information describing a particular attribute of a data item.

Data description entry: An entry in the DATA DIVISION of a COBOL program that is composed of a level-number followed by a data-name, if required, and then followed by a set of data clauses, as required.

DATA DIVISION: One of the four main components of a COBOL program, class definition, or method definition. The DATA DIVISION describes the data to be processed by the object program, class, or method: files to be used and the records contained within them; internal working-storage records that will be needed; data to be made available in more than one program in the COBOL run unit. (Note, the Class DATA DIVISION contains only the WORKING-STORAGE SECTION.)

Data item: A unit of data (excluding literals) defined by a COBOL program or by the rules for function evaluation.

Data-name: A user-defined word that names a data item described in a data description entry. When used in the general formats, 'data-name' represents a word that must not be reference-modified, subscripted or qualified unless specifically permitted by the rules for the format.

DBCS (Double-Byte Character Set): See "Double-Byte Character Set (DBCS)."

Debugging line: A debugging line is any line with a 'D' in the indicator area of the line.

Debugging section: A section that contains a USE FOR DEBUGGING statement.

Declarative sentence: A compiler directing sentence consisting of a single USE statement terminated by the separator period.

Declaratives: A set of one or more special purpose sections, written at the beginning of the Procedure Division, the first of which is preceded by the key word DECLARATIVES and the last of which is followed by the key words END DECLARATIVES. A declarative is composed of a section header, followed by a USE compiler directing sentence, and followed by a set of zero, one, or more associated paragraphs.

De-edit: The logical removal of all editing characters from a numeric edited data item in order to determine that item's unedited numeric value.

Delimited scope statement: Any statement that includes its explicit scope terminator.

Delimiter: A character or a sequence of contiguous characters that identify the end of a string of characters and separate that string of characters from the following string of characters. A delimiter is not part of the string of characters that it delimits.

Descending key: A key upon the values of which data is ordered starting with the highest value of key down to the lowest value of key, in accordance with the rules for comparing data items.

Digit: Any of the numerals from 0 through 9. In COBOL, the term is not used in reference to any other symbol.

Digit position: The amount of physical storage required to store a single digit. This amount may vary depending on the usage specified in the data description entry that defines the data item.

Direct access: The facility to obtain data from storage devices or to enter data into a storage device in such a way that the process depends only on the location of that data and not on a reference to data previously accessed.

Division: A collection of zero, one or more sections or paragraphs, called the division body that are formed and combined in accordance with a specific set of rules. Each division consists of the division header and the related division body. There are four (4) divisions in a COBOL program: Identification, Environment, Data, and Procedure.

Division header: A combination of words followed by a separator period that indicates the beginning of a division. The division headers are:

IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.

DLL: See "dynamic link library."

Do construction: In structured programming, a DO statement is used to group a number of statements in a procedure. In COBOL, an in-line PERFORM statement functions in the same way.

Do-until: In structured programming, a do-until loop will be executed at least once, and until a given condition is true. In COBOL, a TEST AFTER phrase used with the PERFORM statement functions in the same way.

Do-while: In structured programming, a do-while loop will be executed if, and while, a given condition is true. In COBOL, a TEST BEFORE phrase used with the PERFORM statement functions in the same way.

Double-Byte Character Set (DBCS): A set of characters in which each character is represented by two bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require Double-Byte Character Sets. Because each character requires two bytes, entering, displaying, and printing DBCS characters requires hardware and supporting software that are DBCS-capable.

Dynamic access: An access mode in which specific logical records can be obtained from or placed into a mass storage file in a nonsequential manner and obtained from a file in a sequential manner during the scope of the same OPEN statement.

Dynamic link library: A file containing executable code and data bound to a program at load time or run time, rather than during linking. The code and data in a dynamic link library can be shared by several applications simultaneously.

Dynamic Storage Area (DSA): Dynamically acquired storage composed of a register save area and an area available for dynamic storage allocation (such as program variables). DSAs are generally allocated within STACK segments managed by Language Environment.

E

EBCDIC (Extended Binary-Coded Decimal Interchange Code): A coded character set consisting of 8-bit coded characters.

EBCDIC character: Any one of the symbols included in the 8-bit EBCDIC (Extended Binary-Coded-Decimal Interchange Code) set.

Edited data item: A data item that has been modified by suppressing zeroes and/or inserting editing characters.

Element (text element): One logical unit of a string of text, such as the description of a single data item or verb, preceded by a unique code identifying the element type.

Elementary item: A data item that is described as not being further logically subdivided.

Enclave: When running under the Language Environment product, an enclave is analogous to a run unit. An enclave can create other enclaves on OS/390 and CMS by a LINK, on CMS by CMSCALL, and the use of the system () function of C.

End class header: A combination of words, followed by a separator period, that indicates the end of a COBOL class definition. The end class header is:

END CLASS class-name.

End method header: A combination of words, followed by a separator period that indicates the end of a COBOL method definition. The end method header is:

END METHOD method-name.

End of Procedure Division: The physical position of a COBOL source program after which no further procedures appear.

End program header: A combination of words, followed by a separator period that indicates the end of a COBOL source program. The end program header is:

END PROGRAM program-name.

Entry: Any descriptive set of consecutive clauses terminated by a separator period and written in the IDENTIFICATION DIVISION, ENVIRONMENT DIVISION, or DATA DIVISION of a COBOL program.

Environment clause: A clause that appears as part of an ENVIRONMENT DIVISION entry.

ENVIRONMENT DIVISION: One of the four main component parts of a COBOL program, class definition, or method definition. The ENVIRONMENT DIVISION describes the computers upon which the source program is compiled and those on which the object program is executed, and provides a linkage between the logical concept of files and their records, and the physical aspects of the devices on which files are stored.

Environment-name: A name, specified by IBM, that identifies system logical units, printer and card punch control characters, report codes, and/or program switches. When an environment-name is associated with a mnemonic-name in the ENVIRONMENT DIVISION, the mnemonic-name may then be substituted in any format in which such substitution is valid.

Environment variable: Any of a number of variables that describe the way an operating system is going to run and the devices it is going to recognize.

Execution time: See "run time."

Execution-time environment: See "run-time environment."

Expanded date field: A date field containing an expanded (4-digit) year. See also "date field" and "expanded year."

Expanded year: Four digits representing a year, including the century (for example, 1998). Appears in expanded date fields. Compare with "windowed year."

Explicit scope terminator: A reserved word that terminates the scope of a particular Procedure Division statement.

Exponent: A number, indicating the power to which another number (the base) is to be raised. Positive exponents denote multiplication, negative exponents denote division, and fractional exponents denote a root of a quantity. In COBOL, an exponential expression is indicated with the symbol '**' followed by the exponent.

Expression: An arithmetic or conditional expression.

Extend mode: The state of a file after execution of an OPEN statement, with the EXTEND phrase specified for that file, and before the execution of a CLOSE statement, without the REEL or UNIT phrase for that file.

Extensions: Certain COBOL syntax and semantics supported by IBM compilers in addition to those described in ANSI Standard.

External data: The data described in a program as external data items and external file connectors.

External data item: A data item which is described as part of an external record in one or more programs of a run unit and which itself may be referenced from any program in which it is described.

External data record: A logical record which is described in one or more programs of a run unit and whose constituent data items may be referenced from any program in which they are described.

External decimal item: A format for representing numbers in which the digit is contained in bits 4 through 7 and the sign is contained in bits 0 through 3 of the rightmost byte. Bits 0 through 3 of all other bytes contain 1's (hex F). For example, the decimal value of +123 is represented as 1111 0001 1111 0010 1111 0011. (Also known as "zoned decimal item.")

External file connector: A file connector which is accessible to one or more object programs in the run unit.

External floating-point item: A format for representing numbers in which a real number is represented by a pair of distinct numerals. In a floating-point representation, the real number is the product of the fixed-point part (the first numeral), and a value obtained by raising the implicit floating-point base to a power denoted by the exponent (the second numeral).

For example, a floating-point representation of the number 0.0001234 is: 0.1234 -3, where 0.1234 is the mantissa and -3 is the exponent.

External program: The outermost program. A program that is not nested.

External switch: A hardware or software device defined and named by the implementor, which is used to indicate that one of two alternate states exists.

F

Figurative constant: A compiler-generated value referenced through the use of certain reserved words.

File: A collection of logical records.

File attribute conflict condition: An unsuccessful attempt has been made to execute an input-output operation on a file and the file attributes, as specified for that file in the program, do not match the fixed attributes for that file.

File clause: A clause that appears as part of any of the following DATA DIVISION entries: file description entry (FD entry) and sort-merge file description entry (SD entry).

File connector: A storage area which contains information about a file and is used as the linkage between a file-name and a physical file and between a file-name and its associated record area.

File-Control: The name of an ENVIRONMENT DIVISION paragraph in which the data files for a given source program are declared.

File control entry: A SELECT clause and all its subordinate clauses which declare the relevant physical attributes of a file.

File description entry: An entry in the File Section of the DATA DIVISION that is composed of the level indicator FD, followed by a file-name, and then followed by a set of file clauses as required.

File-name: A user-defined word that names a file connector described in a file description entry or a sort-merge file description entry within the File Section of the DATA DIVISION.

File organization: The permanent logical file structure established at the time that a file is created.

File position indicator: A conceptual entity that contains the value of the current key within the key of reference for an indexed file, or the record number of the current record for a sequential file, or the relative record number of the current record for a relative file, or indicates that no next logical record exists, or that an optional input file is not present, or that the at end condition already exists, or that no valid next record has been established.

File Section: The section of the DATA DIVISION that contains file description entries and sort-merge file description entries together with their associated record descriptions.

File system: The collection of files and file management structures on a physical or logical mass storage device, such as a diskette or minidisk.

Fixed file attributes: Information about a file which is established when a file is created and cannot subsequently be changed during the existence of the file. These attributes include the organization of the file (sequential, relative, or indexed), the prime record key, the alternate record keys, the code set, the minimum and maximum record size, the record type (fixed or variable), the collating sequence of the keys for indexed files, the blocking factor, the padding character, and the record delimiter.

Fixed length record: A record associated with a file whose file description or sort-merge description entry requires that all records contain the same number of character positions.

Fixed-point number: A numeric data item defined with a PICTURE clause that specifies the location of an optional sign, the number of digits it contains, and the location of an optional decimal point. The format may be either binary, packed decimal, or external decimal.

Floating-point number: A numeric data item containing a fraction and an exponent. Its value is obtained by multiplying the fraction by the base of the numeric data item raised to the power specified by the exponent.

Format: A specific arrangement of a set of data.

Function: A temporary data item whose value is determined at the time the function is referenced during the execution of a statement.

Function-identifier: A syntactically correct combination of character-strings and separators that references a function. The data item represented by a function is uniquely identified by a function-name with its arguments, if any. A function-identifier may include a reference-modifier. A function-identifier that references an alphanumeric function may be specified anywhere in the general formats that an identifier may be specified, subject to certain restrictions. A function-identifier that references an integer or numeric function may be referenced anywhere in the general formats that an arithmetic expression may be specified.

Function-name: A word that names the mechanism whose invocation, along with required arguments, determines the value of a function.

G

Global name: A name which is declared in only one program but which may be referenced from that program and from any program contained within that program. Condition-names, data-names, file-names, record-names, report-names, and some special registers may be global names.

Group item: A data item that is composed of subordinate data items.

H

Header label: (1) A file label or data set label that precedes the data records on a unit of recording media. (2) Synonym for beginning-of-file label.

High order end: The leftmost character of a string of characters.

I

IBM COBOL extension: Certain COBOL syntax and semantics supported by IBM compilers in addition to those described in ANSI Standard.

IDENTIFICATION DIVISION: One of the four main component parts of a COBOL program, class definition, or method definition. The IDENTIFICATION DIVISION identifies the program name, class name, or method name. The IDENTIFICATION DIVISION may include the following documentation: author name, installation, or date.

Identifier: A syntactically correct combination of character-strings and separators that names a data item. When referencing a data item that is not a function, an identifier consists of a data-name, together with its qualifiers, subscripts, and reference-modifier, as required for uniqueness of reference. When referencing a data item which is a function, a function-identifier is used.

IGZCBSN: The COBOL/370 Release 1 bootstrap routine. It must be link-edited with any module that contains a COBOL/370 Release 1 program.

IGZCBSO: The COBOL for MVS & VM Release 2 and IBM COBOL for OS/390 & VM bootstrap routine. It must be link-edited with any module that contains a COBOL for MVS & VM Release 2 or IBM COBOL for OS/390 & VM program.

Imperative statement: A statement that either begins with an imperative verb and specifies an unconditional action to be taken or is a conditional statement that is delimited by its explicit scope terminator (delimited scope statement). An imperative statement may consist of a sequence of imperative statements.

Implicit scope terminator: A separator period which terminates the scope of any preceding unterminated statement or a phrase of a statement which by its occurrence indicates the end of the scope of any statement contained within the preceding phrase.

Index: A computer storage area or register, the content of which represents the identification of a particular element in a table.

Index data item: A data item in which the values associated with an index-name can be stored in a form specified by the implementer.

Indexed data-name: An identifier that is composed of a data-name, followed by one or more index-names enclosed in parentheses.

Indexed file: A file with indexed organization.

Indexed organization: The permanent logical file structure in which each record is identified by the value of one or more keys within that record.

Indexing: Synonymous with subscripting using index-names.

Index-name: A user-defined word that names an index associated with a specific table.

Inheritance (for classes): A mechanism for using the implementation of one or more classes as the basis for another class. A sub-class inherits from one or more super-classes. By definition the inheriting class conforms to the inherited classes.

Initial program: A program that is placed into an initial state every time the program is called in a run unit.

Initial state: The state of a program when it is first called in a run unit.

Inline: In a program, instructions that are executed sequentially, without branching to routines, subroutines, or other programs.

Input file: A file that is opened in the INPUT mode.

Input mode: The state of a file after execution of an OPEN statement, with the INPUT phrase specified, for that file and before the execution of a CLOSE statement, without the REEL or UNIT phrase for that file.

Input-output file: A file that is opened in the I-O mode.

INPUT-OUTPUT SECTION: The section of the ENVIRONMENT DIVISION that names the files and the external media required by an object program or method and that provides information required for transmission and handling of data during execution of the object program or method definition.

Input-Output statement: A statement that causes files to be processed by performing operations upon individual records or upon the file as a unit. The input-output statements are: ACCEPT (with the identifier phrase), CLOSE, DELETE, DISPLAY, OPEN, READ, REWRITE, SET (with the TO ON or TO OFF phrase), START, and WRITE.

Input procedure: A set of statements, to which control is given during the execution of a SORT statement, for the purpose of controlling the release of specified records to be sorted.

Instance data: Data defining the state of an object. The instance data introduced by a class is defined in the WORKING-STORAGE SECTION of the DATA DIVISION of the class definition. The state of an object also includes the state of the instance variables introduced by base classes that are inherited by the current class. A separate copy of the instance data is created for each object instance.

Integer: (1) A numeric literal that does not include any digit positions to the right of the decimal point. (2) A numeric data item defined in the DATA DIVISION that does not include any digit positions to the right of the decimal point. (3) A numeric function whose definition provides that all digits to the right of the decimal point are zero in the returned value for any possible evaluation of the function.

Integer function: A function whose category is numeric and whose definition does not include any digit positions to the right of the decimal point.

Interface: The information that a client must know to use a class--the names of its attributes and the signatures of its methods. With direct-to-SOM compilers such as COBOL, the interface to a class may be defined by native language syntax for class definitions. Classes implemented in other languages might have their interfaces defined directly in SOM Interface Definition Language (IDL). The COBOL compiler has a compiler option, IDLGEN, to automatically generate IDL for a COBOL class.

Interface Definition Language (IDL): The formal language (independent of any programming language) by which the interface for a class of objects is defined in a IDL file, which the SOM compiler then interprets to create an implementation template file and binding files. SOM's Interface Definition Language is fully compliant with standards established by the Object Management Group's Common Object Request Broker Architecture (CORBA).

Interlingua communication (ILC): The ability of routines written in different programming languages to communicate. ILC support allows the application writer to readily build applications from component routines written in a variety of languages.

Intermediate result: An intermediate field containing the results of a succession of arithmetic operations.

Internal data: The data described in a program excluding all external data items and external file connectors. Items described in the LINKAGE SECTION of a program are treated as internal data.

Internal data item: A data item which is described in one program in a run unit. An internal data item may have a global name.

Internal decimal item: A format in which each byte in a field except the rightmost byte represents two numeric digits. The rightmost byte contains one digit and the sign. For example, the decimal value +123 is represented as 0001 0010 0011 1111. (Also known as packed decimal.)

Internal file connector: A file connector which is accessible to only one object program in the run unit.

Intra-record data structure: The entire collection of groups and elementary data items from a logical record which is defined by a contiguous subset of the data description entries which describe that record. These data description entries include all entries whose level-number is greater than the level-number of the first data description entry describing the intra-record data structure.

Intrinsic function: A pre-defined function, such as a commonly used arithmetic function, called by a built-in function reference.

Invalid key condition: A condition, at object time, caused when a specific value of the key associated with an indexed or relative file is determined to be invalid.

I-O-CONTROL: The name of an ENVIRONMENT DIVISION paragraph in which object program requirements for rerun points, sharing of same areas by several data files, and multiple file storage on a single input-output device are specified.

I-O-CONTROL entry: An entry in the I-O-CONTROL paragraph of the ENVIRONMENT DIVISION which contains clauses that provide information required for the transmission and handling of data on named files during the execution of a program.

I-O-Mode: The state of a file after execution of an OPEN statement, with the I-O phrase specified, for that file and before the execution of a CLOSE statement without the REEL or UNIT phase for that file.

I-O status: A conceptual entity which contains the two-character value indicating the resulting status of an input-output operation. This value is made available to the program through the use of the FILE STATUS clause in the file control entry for the file.

ISPF. Interactive System Productivity Facility: An IBM software product that provides a menu-driven interface to the TSO or VM user. Includes library utilities, a powerful editor, and dialog management.

Iteration structure: A program processing logic in which a series of statements is repeated while a condition is true or until a condition is true.

K

K: When referring to storage capacity, two to the tenth power; 1024 in decimal notation.

Key: A data item that identifies the location of a record or a set of data items which serve to identify the ordering of data.

Key of reference: The key, either prime or alternate, currently being used to access records within an indexed file.

Key word: A reserved word or function-name whose presence is required when the format in which the word appears is used in a source program.

Kilobyte (KB): One kilobyte equals 1024 bytes.

L

Language-name: A system-name that specifies a particular programming language.

Language Environment-conforming: A characteristic of compiler products (COBOL for OS/390 & VM, COBOL for MVS & VM, COBOL/370, AD/Cycle® C/370(TM), C/C++ for MVS and VM, PL/I for MVS and VM) that produce object code conforming to the Language Environment format.

Last-used state: A program is in last-used state if its internal values remain the same as when the program was exited (are not reset to their initial values).

Letter: A character belonging to one of the following two sets:
Uppercase letters: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z
Lowercase letters: a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z

Level indicator: Two alphabetic characters that identify a specific type of file or a position in a hierarchy. The level indicators in the DATA DIVISION are: CD, FD, and SD.

Level-number: A user-defined word, expressed as a two digit number, which indicates the hierarchical position of a data item or the special properties of a data description entry. Level-numbers in the range from 1 through 49 indicate the position of a data item in the hierarchical structure of a logical record. Level-numbers in the range 1 through 9 may be written either as a single digit or as a zero followed by a significant digit. Level-numbers 66, 77 and 88 identify special properties of a data description entry.

Library-name: A user-defined word that names a COBOL library that is to be used by the compiler for a given source program compilation.

Library text: A sequence of text words, comment lines, the separator space, or the separator pseudo-text delimiter in a COBOL library.

LINAGE-COUNTER: A special register whose value points to the current position within the page body.

LINKAGE SECTION: The section in the DATA DIVISION of the called program that describes data items available from the calling program. These data items may be referred to by both the calling and called program.

Literal: A character string whose value is specified either by the ordered set of characters comprising the string, or by the use of a figurative constant.

Local: A set of attributes for a program execution environment indicating culturally sensitive considerations, such as: character code page, collating sequence, date/time format, monetary value representation, numeric value representation, or language.

LOCAL-STORAGE SECTION: The section of the DATA DIVISION that defines storage that is allocated and freed on a per-invocation basis, depending on the value assigned in their VALUE clauses.

Logical operator: One of the reserved words AND, OR, or NOT. In the formation of a condition, either AND, or OR, or both can be used as logical connectives. NOT can be used for logical negation.

Logical record: The most inclusive data item. The level-number for a record is 01. A record may be either an elementary item or a group of items. The term is synonymous with record.

Low order end: The rightmost character of a string of characters.

M

Main program: In a hierarchy of programs and subroutines, the first program to receive control when the programs are run.

Mass storage: A storage medium in which data may be organized and maintained in both a sequential and non-sequential manner.

Mass storage device: A device having a large storage capacity; for example, magnetic disk, magnetic drums.

Mass storage file: A collection of records that is assigned to a mass storage medium.

Megabyte (M): One megabyte equals 1,048,576 bytes.

Merge file: A collection of records to be merged by a MERGE **statement. The merge file is created and can be used only by the merge function.**

Metaclass: A SOM class whose instances are SOM class-objects. The methods defined in metaclasses are executed without requiring any object instances of the class to exist, and are frequently used to create instances of the class.

Method: Procedural code that defines one of the operations supported by an object, and that is executed by an INVOKE statement on that object.

Method Definition: The COBOL source unit that defines a method.

Method identification entry: An entry in the METHOD-ID paragraph of the IDENTIFICATION DIVISION which contains clauses that specify the method-name and assign selected attributes to the method definition.

Method-name: A user-defined word that identifies a method.

Mnemonic-name: A user-defined word that is associated in the ENVIRONMENT DIVISION with a specified implementor-name.

MLE: See "millennium language extensions."

Multitasking: Mode of operation that provides for the concurrent, or interleaved, execution of two or more tasks. When running under the Language Environment product, multitasking is synonymous with multithreading.

N

Name: A word composed of not more than 30 characters that defines a COBOL operand.

Native character set: The implementor-defined character set associated with the computer specified in the OBJECT-COMPUTER paragraph.

Native collating sequence: The implementor-defined collating sequence associated with the computer specified in the OBJECT-COMPUTER paragraph.

Negated combined condition: The 'NOT' logical operator immediately followed by a parenthesized combined condition.

Negated simple condition: The 'NOT' logical operator immediately followed by a simple condition.

Nested program: A program that is directly contained within another program.

Next executable sentence: The next sentence to which control will be transferred after execution of the current statement is complete.

Next executable statement: The next statement to which control will be transferred after execution of the current statement is complete.

Next record: The record that logically follows the current record of a file.

Noncontiguous items: Elementary data items in the WORKING-STORAGE and LINKAGE SECTIONs that bear no hierarchic relationship to other data items.

Non-numeric item: A data item whose description permits its content to be composed of any combination of characters taken from the computer's character set. Certain categories of non-numeric items may be formed from more restricted character sets.

Non-numeric literal: A literal bounded by quotation marks. The string of characters may include any character in the computer's character set.

Null: Figurative constant used to assign the value of an invalid address to pointer data items. NULLS can be used wherever NULL can be used.

Numeric character: A character that belongs to the following set of digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Numeric-edited item: A numeric item that is in such a form that it may be used in printed output. It may consist of external decimal digits from 0 through 9, the decimal point, commas, the dollar sign, editing sign control symbols, plus other editing symbols.

Numeric function: A function whose class and category are numeric but which for some possible evaluation does not satisfy the requirements of integer functions.

Numeric item: A data item whose description restricts its content to a value represented by characters chosen from the digits from '0' through '9'; if signed, the item may also contain a '+', '-', or other representation of an operational sign.

Numeric literal: A literal composed of one or more numeric characters that may contain either a decimal point, or an algebraic sign, or both. The decimal point must not be the rightmost character. The algebraic sign, if present, must be the leftmost character.

O

Object: An entity that has state (its data values) and operations (its methods). An object is a way to encapsulate state and behavior.

Object code: Output from a compiler or assembler that is itself executable machine code or is suitable for processing to produce executable machine code.

OBJECT-COMPUTER: The name of an ENVIRONMENT DIVISION paragraph in which the computer environment, within which the object program is executed, is described.

Object computer entry: An entry in the OBJECT-COMPUTER paragraph of the ENVIRONMENT DIVISION which contains clauses that describe the computer environment in which the object program is to be executed.

Object deck: A portion of an object program suitable as input to a linkage editor. Synonymous with object module and text deck.

Object module: Synonym for object deck or text deck.

Object of entry: A set of operands and reserved words, within a DATA DIVISION entry of a COBOL program that immediately follows the subject of the entry.

Object program: A set or group of executable machine language instructions and other material designed to interact with data to provide problem solutions. In this context, an object program is generally the machine language result of the operation of a COBOL compiler on a source program. Where there is no danger of ambiguity, the word 'program' alone may be used in place of the phrase 'object program.'

Object time: The time at which an object program is executed. The term is synonymous with execution time.

Obsolete element: A COBOL language element in Standard COBOL that is to be deleted from the next revision of Standard COBOL.

ODBC: Open Database Connectivity that provides you access to data from a variety of databases and file systems.

Open mode: The state of a file after execution of an OPEN statement for that file and before the execution of a CLOSE statement without the REEL or UNIT phrase for that file. The particular open mode is specified in the OPEN statement as either INPUT, OUTPUT, I-O or EXTEND.

Operand: Whereas the general definition of operand is "that component which is operated upon," for the purpose of this document, any lowercase word (or words) that appears in a statement or entry format may be considered to be an operand and, as such, is an implied reference to the data indicated by the operand.

Operational sign: An algebraic sign, associated with a numeric data item or a numeric literal, to indicate whether its value is positive or negative.

Optional file: A file which is declared as being not necessarily present each time the object program is executed. The object program causes an interrogation for the presence or absence of the file.

Optional word: A reserved word that is included in a specific format only to improve the readability of the language and whose presence is optional to the user when the format in which the word appears is used in a source program.

OS/2 (Operating System/2*): A multi-tasking operating system for the IBM Personal Computer family that allows you to run both DOS mode and OS/2 mode programs.

Output file: A file that is opened in either the OUTPUT mode or EXTEND mode.

Output mode: The state of a file after execution of an OPEN statement, with the OUTPUT or EXTEND phrase specified, for that file and before the execution of a CLOSE statement without the REEL or UNIT phrase for that file.

Output procedure: A set of statements to which control is given during execution of a SORT statement after the sort function is completed, or during execution of a MERGE statement after the merge function reaches a point at which it can select the next record in merged order when requested.

Overflow condition: A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage.

P

Packed decimal item: See "internal decimal item."

Padding character: An alphanumeric character used to fill the unused character positions in a physical record**.**

Page: A vertical division of output data representing a physical separation of such data, the separation being based on internal logical requirements and/or external characteristics of the output medium.

Page body: That part of the logical page in which lines can be written and/or spaced.

Paragraph: In the Procedure Division, a paragraph-name followed by a separator period and by zero, one, or more sentences. In the IDENTIFICATION and ENVIRONMENT DIVISIONs, a paragraph header followed by zero, one, or more entries.

Paragraph header: A reserved word, followed by the separator period that indicates the beginning of a paragraph in the IDENTIFICATION and ENVIRONMENT DIVISIONs.

Paragraph-name: A user-defined word that identifies and begins a paragraph in the Procedure Division.

Parameter: Parameters are used to pass data values between calling and called programs.

Password: A unique string of characters that a program, computer operator, or user must supply to meet security requirements before gaining access to data.

Phrase: A phrase is an ordered set of one or more consecutive COBOL character-strings that form a portion of a COBOL procedural statement or of a COBOL clause.

Physical record: See "block."

Pointer data item: A data item in which address values can be stored. Data items are explicitly defined as pointers with the USAGE IS POINTER clause. ADDRESS OF special registers are implicitly defined as pointer data items. Pointer data items can be compared for equality or moved to other pointer data items.

Portability: The ability to transfer an application program from one application platform to another with relatively few changes to the source program.

Preloaded: In COBOL this refers to COBOL programs that remain resident in storage under IMS instead of being loaded each time they are called.

Prime record key: A key whose contents uniquely identify a record within an indexed file.

Priority-number: A user-defined word which classifies sections in the Procedure Division for purposes of segmentation. Segment-numbers may contain only the characters '0','1', ... , '9'. A segment-number may be expressed either as a one- or two-digit number.

Procedure: A paragraph or group of logically successive paragraphs, or a section or group of logically successive sections, within the Procedure Division.

Procedure branching statement: A statement that causes the explicit transfer of control to a statement other than the next executable statement in the sequence in which the statements are written in the source program. The procedure branching statements are: ALTER, CALL, EXIT, EXIT PROGRAM, GO TO, MERGE, (with the OUTPUT PROCEDURE phrase), PERFORM and SORT (with the INPUT PROCEDURE or OUTPUT PROCEDURE phrase).

Procedure Division: One of the four main component parts of a COBOL program, class definition, or method definition. The Procedure Division contains instructions for solving a problem. The Program and Method Procedure Divisions may contain imperative statements, conditional statements, and compiler directing statements, paragraphs, procedures, and sections. The Class Procedure Division contains only method definitions.

Procedure integration: One of the functions of the COBOL optimizer is to simplify calls to performed procedures or contained programs. PERFORM procedure integration is the process whereby a PERFORM statement is replaced by its performed procedures. Contained program procedure integration is the process where a CALL to a contained program is replaced by the program code.

Procedure-name: A user-defined word that is used to name a paragraph or section in the Procedure Division. It consists of a paragraph-name (which may be qualified) or a section-name.

Procedure-pointer data item: A data item in which a pointer to an entry point can be stored. A data item defined with the USAGE IS PROCEDURE-POINTER clause contains the address of a procedure entry point.

Program identification entry: An entry in the PROGRAM-ID paragraph of the IDENTIFICATION DIVISION which contains clauses that specify the program-name and assign selected program attributes to the program.

Program-name: In the IDENTIFICATION DIVISION and the end program header, a user-defined word that identifies a COBOL source program.

Pseudo-text: A sequence of text words, comment lines, or the separator space in a source program or COBOL library bounded by, but not including, pseudo-text delimiters.

Pseudo-text delimiter: Two contiguous equal sign characters (==) used to delimit pseudo-text.

Q

QSAM (Queued Sequential Access Method): An extended version of the basic sequential access method (BSAM). When this method is used, a queue is formed of input data blocks that are waiting processing or of output data blocks that have been processed and are awaiting transfer to auxiliary storage or to an output device.

Qualified data-name: An identifier that is composed of a data-name followed by one or more sets of either of the connectives OF and IN followed by a data-name qualifier.

Qualifier: A data-name or a name associated with a level indicator which is used in a reference either together with another data-name which is the name of an item that is subordinate to the qualifier or together with a condition-name.

A section-name that is used in a reference together with a paragraph-name specified in that section.

A library-name that is used in a reference together with a text-name associated with that library.

R

Random access: An access mode in which the program-specified value of a key data item identifies the logical record that is obtained from, deleted from, or placed into a relative or indexed file.

Record: See "logical record."

Record area: A storage area allocated for the purpose of processing the record described in a record description entry in the File Section of the DATA DIVISION. In the File Section, the current number of character positions in the record area is determined by the explicit or implicit RECORD clause.

Record description: See "record description entry."

Record description entry: The total set of data description entries associated with a particular record. The term is synonymous with record description.

Recording mode: The format of the logical records in a file. Recording mode can be F (fixed-length), V (variable-length), S (spanned), or U (undefined).

Record key: A key whose contents identify a record within an indexed file.

Record-name: A user-defined word that names a record described in a record description entry in the DATA DIVISION of a COBOL program.

Record number: The ordinal number of a record in the file whose organization is sequential.

Recursion: A program calling itself or being directly or indirectly called by a one of its called programs.

Recursively capable: A program is recursively capable (can be called recursively) if the RECURSIVE attribute is on the PROGRAM-ID statement.

Reel: A discrete portion of a storage medium, the dimensions of which are determined by each implementor that contains part of a file, all of a file, or any number of files. The term is synonymous with unit and volume.

Reentrant: The attribute of a program or routine that allows more than one user to share a single copy of a load module.

Reference format: A format that provides a standard method for describing COBOL source programs.

Reference modification: A method of defining a new alphanumeric data item by specifying the leftmost character and length relative to the leftmost character of another alphanumeric data item.

Reference-modifier: A syntactically correct combination of character-strings and separators that defines a unique data item. It includes a delimiting left parenthesis separator, the leftmost character position, a colon separator, optionally a length, and a delimiting right parenthesis separator.

Relation: Refer "relational operator" or "relation condition."

Relational operator: A reserved word, a relation character, a group of consecutive reserved words, or a group of consecutive reserved words and relation characters used in the construction of a relation condition.

Relation condition: The proposition, for which a truth value can be determined, that the value of an arithmetic expression, data item, non-numeric literal, or index-name has a specific relationship to the value of another arithmetic expression, data item, non-numeric literal, or index name. (See also "relational operator.")

Relative file: A file with relative organization.

Relative key: A key whose contents identify a logical record in a relative file.

Relative organization: The permanent logical file structure in which each record is uniquely identified by an integer value greater than zero, which specifies the record's logical ordinal position in the file.

Relative record number: The ordinal number of a record in a file whose organization is relative. This number is treated as a numeric literal which is an integer.

Reserved word: A COBOL word specified in the list of words that may be used in a COBOL source program, but that must not appear in the program as user-defined words or system-names.

Resource: A facility or service, controlled by the operating system that can be used by an executing program.

Resultant identifier: A user-defined data item that is to contain the result of an arithmetic operation.

Reusable environment: A reusable environment is when you establish an assembler program as the main program by using either ILBOSTP0 programs, IGZERRE programs, or the RTEREUS run-time option.

Routine: A set of statements in a COBOL program that causes the computer to perform an operation or series of related operations. In Language Environment, refers to a procedure, function, or subroutine.

Routine-name: A user-defined word that identifies a procedure written in a language other than COBOL.

Run time: The time at which an object program is executed. The term is synonymous with object time.

Run-time environment: The environment in which a COBOL program executes.

Run unit: A stand-alone object program, or several object programs, that interact via COBOL CALL statements, which function at run time as an entity.

S

SBCS (Single Byte Character Set): See "Single Byte Character Set (SBCS)."

Scope terminator: A COBOL reserved word that marks the end of certain Procedure Division statements. It may be either explicit (END-ADD, for example) or implicit (separator period).

Section: A set of zero, one or more paragraphs or entities, called a section body, the first of which is preceded by a section header. Each section consists of the section header and the related section body.

Section header: A combination of words followed by a separator period that indicates the beginning of a section in the Environment, Data, and Procedure Divisions. In the ENVIRONMENT and DATA DIVISIONs, a section header is composed of reserved words followed by a separator period.

Section-name: A user-defined word that names a section in the Procedure Division.

Selection structure: A program processing logic in which one or another series of statements is executed, depending on whether a condition is true or false.

Sentence: A sequence of one or more statements, the last of which is terminated by a separator period.

Separator: A character or two contiguous characters used to delimit character-strings.

Separator comma: A comma (,) followed by a space used to delimit character-strings.

Separator period: A period (.) followed by a space used to delimit character-strings.

Separator semicolon: A semicolon (;) followed by a space used to delimit character-strings.

Sequential access: An access mode in which logical records are obtained from or placed into a file in a consecutive predecessor-to-successor logical record sequence determined by the order of records in the file.

Sequential file: A file with sequential organization.

Sequential organization: The permanent logical file structure in which a record is identified by a predecessor-successor relationship established when the record is placed into the file.

Serial search: A search in which the members of a set are consecutively examined, beginning with the first member and ending with the last.

*** 77-level-description-entry:** A data description entry that describes a noncontiguous data item with the level-number 77.

*** sign condition:** The proposition, for which a truth value can be determined, that the algebraic value of a data item or an arithmetic expression is either less than, greater than, or equal to zero.

*** simple condition:** Any single condition chosen from the set:
Relation condition
Class condition
Condition-name condition
Switch-status condition
Sign condition

Single Byte Character Set (SBCS): A set of characters in which each character is represented by a single byte. See also "EBCDIC (Extended Binary-Coded Decimal Interchange Code)."

Slack bytes: Bytes inserted between data items or records to ensure correct alignment of some numeric items. Slack bytes contain no meaningful data. In some cases, they are inserted by the compiler; in others, it is the responsibility of the programmer to insert them. The SYNCHRONIZED clause instructs the compiler to insert slack bytes when they are needed for proper alignment. Slack bytes between records are inserted by the programmer.

SOM: See "System Object Model"

Sort file: A collection of records to be sorted by a SORT statement. The sort file is created and can be used by the sort function only.

Sorbetière file description entry: An entry in the File Section of the DATA DIVISION that is composed of the level indicator SD, followed by a file-name, and then followed by a set of file clauses as required.

SOURCE-COMPUTER: The name of an ENVIRONMENT DIVISION paragraph in which the computer environment, within which the source program is compiled, is described.

Source program: Although it is recognized that a source program may be represented by other forms and symbols, in this document it always refers to a syntactically correct set of COBOL statements. A COBOL source program commences with the IDENTIFICATION DIVISION or a COPY statement. A COBOL source program is terminated by the end program header, if specified, or by the absence of additional source program lines.

Special-character word: A reserved word that is an arithmetic operator or a relation character.

SPECIAL-NAMES: The name of an ENVIRONMENT DIVISION paragraph in which environment-names are related to user-specified mnemonic-names.

Special names entry: An entry in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION which provides means for specifying the currency sign values and currency symbols; choosing the decimal point; specifying symbolic characters; relating implementor-names to user-specified mnemonic-names; relating alphabet-names to character sets or collating sequences; and relating class-names to sets of characters.

Special registers: Certain compiler generated storage areas whose primary use is to store information produced in conjunction with the use of a specific COBOL feature.

Standard data format: The concept used in describing the characteristics of data in a COBOL DATA DIVISION under which the characteristics or properties of the data are expressed in a form oriented to the appearance of the data on a printed page of infinite length and breadth, rather than a form oriented to the manner in which the data is stored internally in the computer, or on a particular external medium.

Statement: A syntactically valid combination of words, literals, and separators, beginning with a verb, written in a COBOL source program.

STL: Standard Language file system: native workstation and PC file system for COBOL and PL/I. Supports sequential, relative, and indexed files, including the full ANSI 85 COBOL standard I/O language and all of the extensions described in IBM COBOL Language Reference, unless exceptions are explicitly noted.

Structured programming: A technique for organizing and coding a computer program in which the program comprises a hierarchy of segments, each segment having a single entry point and a single exit point. Control is passed downward through the structure without unconditional branches to higher levels of the hierarchy.

Sub-class: A class that inherits from another class. When two classes in an inheritance relationship are considered together, the sub-class is the inheritor or inheriting class; the super-class is the inheritee or inherited class.

Subprogram: See "called program."

Subscript: An occurrence number represented by either an integer, a data-name optionally followed by an integer with the operator + or -, or an index-name optionally followed by an integer with the operator + or -, that identifies a particular element in a table. A subscript may be the word ALL when the subscripted identifier is used as a function argument for a function allowing a variable number of arguments.

Symbolic-character: A user-defined word that specifies a user-defined figurative constant.

Syntax: (1) The relationship among characters or groups of characters, independent of their meanings or the manner of their interpretation and use. (2) The structure of expressions in a language. (3) The rules governing the structure of a language. (4) The relationship among symbols. (5) The rules for the construction of a statement.

System-name: A COBOL word that is used to communicate with the operating environment.

System Object Model (SOM): IBM's object-oriented programming technology for building, packaging, and manipulating class libraries. SOM conforms to the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) standards.

T

Table: A set of logically consecutive items of data that are defined in the DATA DIVISION by means of the OCCURS clause.

Table element: A data item that belongs to the set of repeated items comprising a table.

Text deck: Synonym for object deck or object module.

Text-name: A user-defined word that identifies library text.

Trailer-label: (1) A file or data set label that follows the data records on a unit of recording medium. (2) Synonym for end-of-file label.

Truth value: The representation of the result of the evaluation of a condition in terms of one of two values: true or false.

U

Unary operator: A plus (+) or a minus (-) sign, that precedes a variable or a left parenthesis in an arithmetic expression and that has the effect of multiplying the expression by +1 or -1, respectively.

Unit: A module of direct access, the dimensions of which are determined by IBM.

Universal object reference: A data-name that can refer to an object of any class.

Unsuccessful execution: The attempted execution of a statement that does not result in the execution of all the operations specified by that statement. The unsuccessful execution of a statement does not affect any data referenced by that statement, but may affect status indicators.

UPSI switch: A program switch that performs the functions of a hardware switch. Eight are provided: UPSI-0 through UPSI-7.

User-defined word: A COBOL word that must be supplied by the user to satisfy the format of a clause or statement.

V

Variable: A data item whose value may be changed by execution of the object program. A variable used in an arithmetic expression must be a numeric elementary item.

Variable length record: A record associated with a file whose file description or sort-merge description entry permits records to contain a varying number of character positions.

Variable occurrence data item: A variable occurrence data item is a table element which is repeated a variable number of times. Such an item must contain an OCCURS DEPENDING ON clause in its data description entry, or be subordinate to such an item.

Variably located group: A group item following, and not subordinate to, a variable-length table in the same level-01 record.

Variably located item: A data item following, and not subordinate to, a variable-length table in the same level-01 record.

Verb: A word that expresses an action to be taken by a COBOL compiler or object program.

VM/SP (Virtual Machine/System Product): An IBM-licensed program that manages the resources of a single computer so that multiple computing systems appear to exist. Each virtual machine is the functional equivalent of a "real" machine.

Volume: A module of external storage. For tape devices it is a reel; for direct-access devices it is a unit.

Volume switch procedures: System specific procedures executed automatically when the end of a unit or reel has been reached before end-of-file has been reached.

VSAM (Virtual Storage Access Method): A high-performance mass storage access method. Three types of data organization are available to COBOL programs: entry sequenced data sets (ESDS), key sequenced data sets (KSDS), and relative record data sets (RRDS). Their COBOL equivalents are, respectively: sequential, indexed, and relative organizations. A fourth type of organization, linear data sets, is only available from Assembler.

W

Windowed date field: A date field containing a windowed (2-digit) year. See also "date field" and "windowed year."

Windowed year: Two digits representing a year within a century window (for example, 98). Appears in windowed date fields.

Word: A character-string of not more than 30 characters which forms a user-defined word, a system-name, a reserved word, or a function-name.

WORKING-STORAGE SECTION: The section of the DATA DIVISION that describes working storage data items, composed either of noncontiguous items or working storage records or of both.

Y

Year 2000 problem: The Year 2000 problem refers to the limitation of 2-digit year date fields that were used to save storage in the 1960s and 1970s. For example, it is not possible to compute the age of someone who is older than 100 years with 2-digit year date fields, and on 1/1/2000, the current date will not be greater than the previous day's date. Because so many applications and data have only 2-digit year dates, they must all be changed before the year 2000 to avoid failure.

Z

Zoned decimal item: See "external decimal item."

References

Websites

- http://homepage.ntlworld.com/zingmatter/zingcobol/index.html
- http://www.techtutorials.info/cobol.html
- http://www.techiwarehouse.com/Cobol/Cobol_Tutorial.html
- http://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/library
- http://www.csis.ul.ie/cobol/Course/COBOLIntro.htm
- http://docs.hp.com/cgi-bin/doc3k/B3150090013.11820/10

Books

Stern and Stern Structured COBOL Programming	- Nancy stern
Structured COBOL	- Philippakis and Kazmier
COBOL Programming	- Roy & Dastidar
How to design & develop COBOL programs	- Paul N.
VS COBOL II - guide for programmers 2 ed.	- Anne prince
Various COBOL IBM Manuals	
Online Training Materials in Web	

STUDENT NOTES: