

Welcome To The World of COBOL

- COBOL: COmmon Business Orientated Language
- Who uses COBOL Today?
 - Accounting Systems
 - Banking
 - Financial Services
 - Inventory Control
 - Payroll
 - Etc.



The World of The 1950's

Scientific Programming

FORTRAN

Business Programming

COBOL

Where Did COBOL Come From?

- Grace Brewster Hopper was an American computer scientist, mathematician, and United States Navy rear admiral.
- She was part of the team that developed the UNIVAC I computer. She managed the development of one of the first COBOL compilers.
- She believed that programming should be simplified with an English-based computer programming language.
- Her compiler converted English terms into machine code understood by computers.
- In 1959, she participated in the CODASYL consortium, helping to create a machine-independent programming language called COBOL, which was based on English words.



What Was The Big Deal?

- FORTRAN was too scientific for business applications
- In 1959, programs developed in machine/assembly were not portable.
- It was very expensive to develop programs: \$800,000 on average for a program to be developed (in 1959 dollars!).
- Porting a program to run on another computer would cost \$600,000.
- Goals for creating a programming language for business:
 - Be flexible and work in Government, Business, Healthcare, etc.
 - Be able to handle huge amounts of data on a large scale
 - Syntax that resembles everyday English
- COBOL was designed to be a high capacity language: it can process enormous amounts of data.

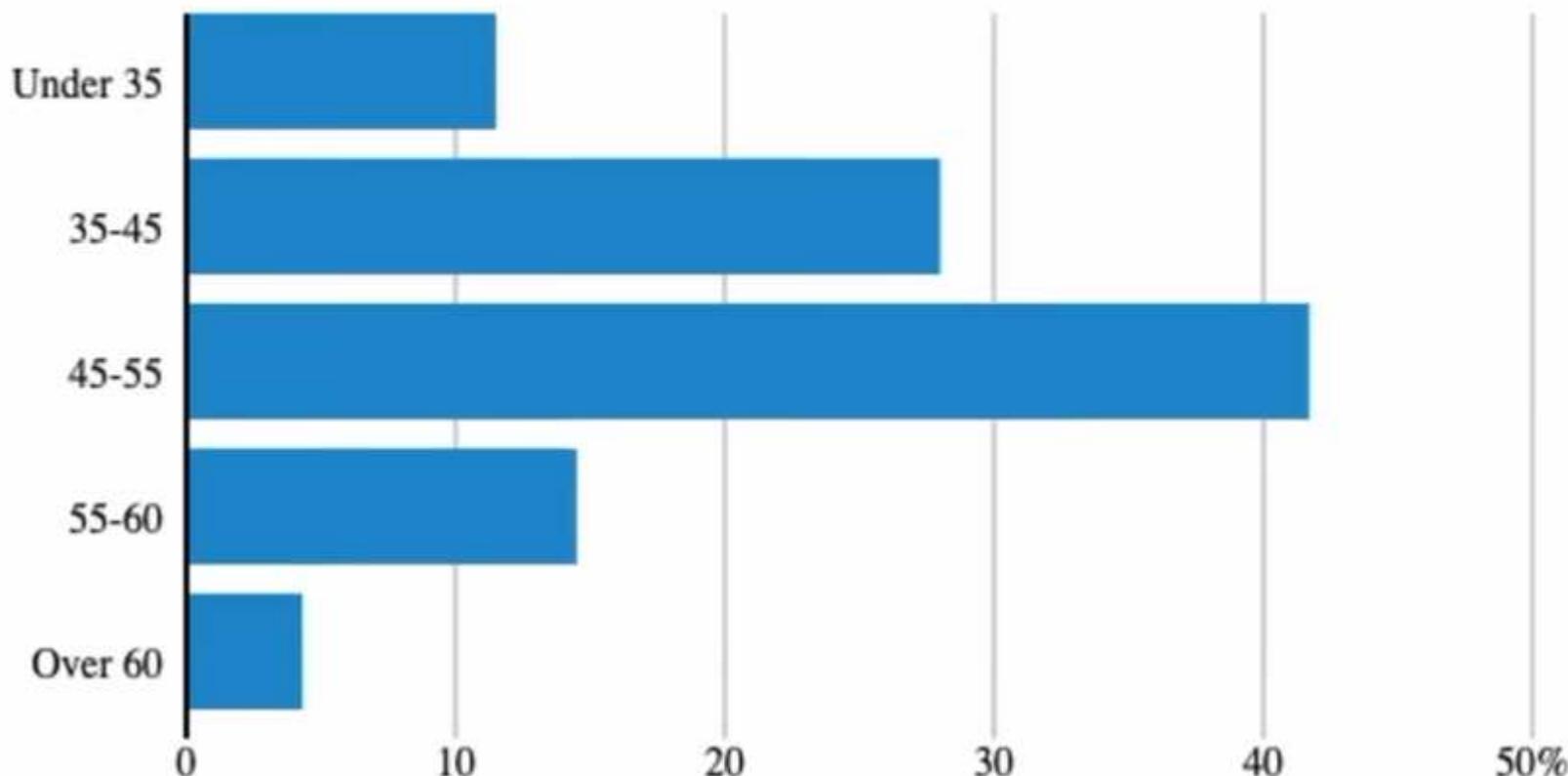
Why Bother To Learn COBOL?

- The reality is that COBOL is the major programming language for business applications even today.
- More than 220 billion lines of COBOL code are still in use which equals ~80% of the world's actively used code.
- 90% of critical business applications use COBOL.
- 95% of ATM transactions are handled by COBOL / 60% healthcare records
- U.S. Government: Social Security, Department of Defense Payment Systems, Internal Revenue Service
- Every day \$3 Trillion worth of transactions are handled by COBOL code.
- No – new applications are generally not developed in COBOL. But all of the existing ones that run the world were developed in COBOL and they have to be enhanced and maintained...

Average Age Of A COBOL Developer

AVERAGE AGE OF DEVELOPERS

On average, COBOL programmers are most likely to be between 45-55 years old.



Caution!

- Just learning COBOL may not be enough to secure a high paying job
- COBOL program often run on Mainframes
- To work in a Mainframe environment you also have to know...
- JCL
 - JCL, or Job Control Language, is a scripting language used primarily on IBM mainframe computers to define and control the execution of batch jobs.
- IMS
 - IMS, or Information Management System, is a premier transaction and hierarchical database management system developed by IBM for mainframes. It's designed for critical online applications and data requiring high availability, performance, and integrity.
- DB2
 - DB2 is a family of database server products developed by IBM. It's a relational database management system (RDBMS) known for its scalability, reliability, and performance, particularly in handling large volumes of data.

How To Install COBOL

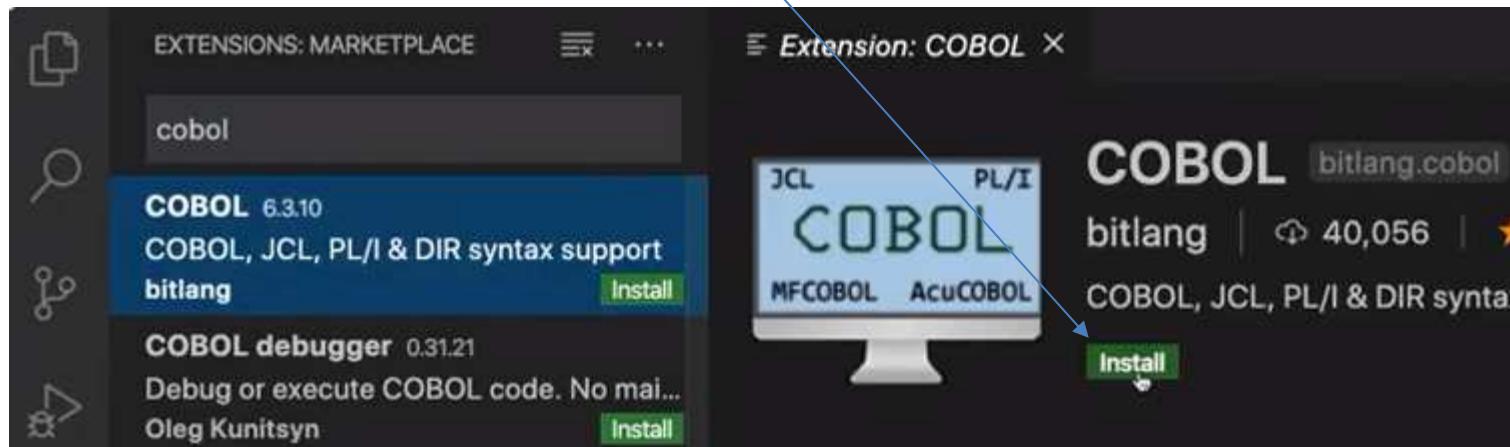


What Types of COBOL Are There?

- Mainframe
 - IBM Enterprise COBOL for z/OS
- PC-based COBOL
 - Designed for smaller applications
 - Micro Focus COBOL: MF-COBOL
 - Open Source: GnuCOBOL
- GnuCOBOL
 - The latest current release is v3.2, which was released on 28 July 2023.
 - GnuCOBOL translates a [COBOL](#) program (source code) into a [C](#) program.

Windows Install

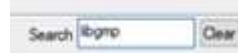
- Install Visual Studio Code (code.visualstudio.com)
- While installing, make sure that you select the "add to path" option.
- Open VSCode
- Click on "Extensions" option on the left hand menu.
- Type in "COBOL" in the search box.
- First option should be: "COBOL" from bitlang.
- Click on "Install" green button.
- You have now installed the VS CODE COBOL extension.

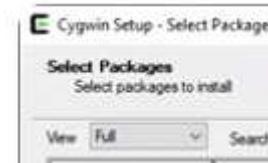


Windows Install

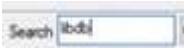
- Use Google to search for GnuCOBOL.
- It is located at **sourceforge.net/project/open-cobol**
- Select "download"
- Unzip file here (go to 7-zip.org to get unzip tool if needed)
- Subdirectory gnucobol-3.0-rc1 will be created.
- Move this subdirectory to the root directory on your laptop.
- Enter into the subdirectory gnucobol-3.0-rc1
- Open README file – it contains the GnuCOBOL installation instructions.

Windows Install

- Now you need to get Cygwin: Linux-like environment on Windows
- Go to Cygwin.com
- Install by running **setup-x86_64.exe**
- Select "Install from Internet" option during the installation process
- During the installation process it will get to the Select Packages screen and stop.
- Click on the View option and select "Full".
- Search for "libgmp" 
- Click in the "new" column so that the libgmp packages will be installed.



Package	Current	New
addlibgmp0	Skip	
libgmp-devel	6.2.0-2	
libgmp-doc	6.2.0-2	
libgmp10	6.2.0-2	
libgmpxx4		6.2.0-2

- Next search for "libdb" 
- Click in new column to install 2 packages lib-devel and libdb5.3

Package	Current	New
libdb-devel	5.3.20-2	
libdb5.3		5.3.20-2

Windows Install

- Search for "ncurses" 
- Click in new column so that packages will be installed

Package	Current	New
gambas3-gb-ncurses	Skip	
libncurses++-10	6.1-1.20190727	
libncurses-devel	6.1-1.20190727	
libncursesw-10	6.1-1.20190727	
mingw64-i686-ncurses	6.0-12.20171125	
mingw64-x86_64-ncurses	6.0-12.20171125	
ncurses	6.1-1.20190727	
ncurses-debuginfo	6.1-1.20190727	
ncurses-demo	6.1-1.20190727	

- Search for "gcc" 
- Click in new column to install the following packages:

Package	Current	New
colorgcc		1.3.2-2
cygwin32-gcc-core		6.4.0-1
cygwin32-gcc-debuginfo		6.4.0-1
cygwin32-gcc-fortran		Skip
cygwin32-gcc-g++		6.4.0-1
dgpp-gcc-ada		Skip
dgpp-gcc-core		Skip
dgpp-gcc-debuginfo		Skip
dgpp-gcc-fortran		Skip
dgpp-gcc-g++		Skip
dgpp-gcc-objc		Skip
dgpp-gcc-objc++		Skip
gcc-core		9.3.0-2
gcc-debuginfo		9.3.0-2
gcc-fortran		Skip
gcc-g++		Skip
gcc-objc		Skip
gcc-objc++		Skip
gcc-tools-epoch1-autocnf		2.59-2
gcc-tools-epoch1-automake		1.9.5-2
gcc-tools-epoch2-autocnf		Skip
gcc-tools-epoch2-automake		Skip
gmakeakedep		Skip
ibacc1		9.3.0-2

Windows Install

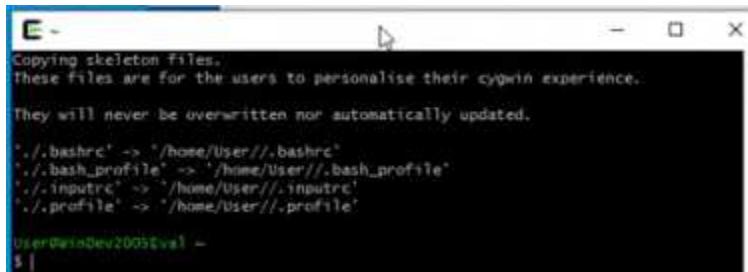
- Search for "make" 
- Select the new column and install the following packages:

Package	Current	New
automake	11.1	
automake1.10	Skip	
automake1.11	Skip	
automake1.12	Skip	
automake1.13	Skip	
automake1.14	Skip	
automake1.15	Skip	
automake1.16	1.16.1-1	
automake1.4	Skip	
automake1.5	Skip	
automake1.6	Skip	
automake1.7	Skip	
automake1.8	Skip	
automake1.9	Skip	
cmake	3.14.5-1	
cmake-debuginfo	3.14.5-1	
cmake-doc	3.14.5-1	
cmake-gui	3.14.5-1	
emacs-cmake	Skip	
extra-cmake-modules	Skip	
gcc-tools-epoch1-automake	1.9.6-2	
gcc-tools-epoch2-automake	Skip	
gccmakeped	1.0.3-1	
make	Skip	
make-debuginfo	Skip	
libWMaker-devel	Skip	
libWMaker1	Skip	
libpagemaker-tools	Skip	
libpagemaker0.0-debuginfo	Skip	
libpagemaker0.0-devel	Skip	
libpagemaker0.0-doc	Skip	
libpagemaker0.0_0	Skip	
make	4.3-1	
make-debuginfo	4.3-1	

- Then click "Next" and complete installation.

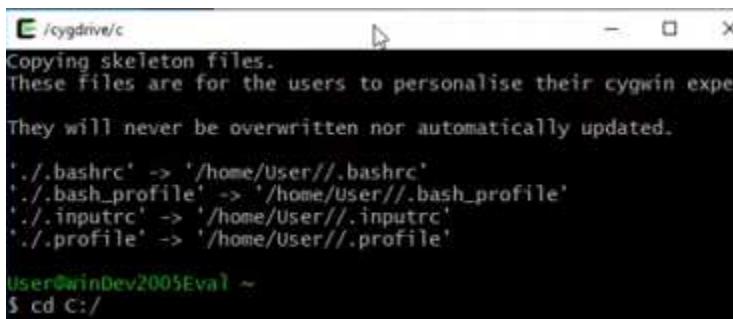
Windows Install

- Open Cygwin application in the gnucobob-3.0-rc1 subdirectory.



A screenshot of a Cygwin terminal window. The title bar says 'E-'. The window content shows the following text:
Copying skeleton files.
These files are for the users to personalise their cygwin experience.
They will never be overwritten nor automatically updated.
./.bashrc' -> '/home/User//.bashrc'
./.bash_profile' -> '/home/User//.bash_profile'
./.inputrc' -> '/home/User//.inputrc'
./.profile' -> '/home/User//.profile'
User@winDev2005Eval ~
\$ |

- Go to subdirectory where GnuCOBOL folder is located



A screenshot of a Cygwin terminal window. The title bar says 'E /cygdrive/c'. The window content shows the following text:
Copying skeleton files.
These files are for the users to personalise their cygwin experience.
They will never be overwritten nor automatically updated.
./.bashrc' -> '/home/User//.bashrc'
./.bash_profile' -> '/home/User//.bash_profile'
./.inputrc' -> '/home/User//.inputrc'
./.profile' -> '/home/User//.profile'
User@winDev2005Eval ~
\$ cd C:/

- Go inside of the folder:



A screenshot of a Cygwin terminal window. The title bar says 'User@winDev2005Eval /cygdrive/c'. The window content shows the following text:
User@winDev2005Eval /cygdrive/c
\$ cd gnucobol-3.0-rc1/

Windows Install

- Display files in folder

```
User@WinDev2005Eval ~ /cygdrive/c/gnucobol-3.0-rc1
$ ls
ABOUT-NLS      cbintr.tex  configure      INSTALL      README
aclocal.m4      cbmem.tex   configure.ac   lib          tarstamp.h
AUTHORS        cbrese.tex  copy          libcob      tests
bin            cbrunt.tex  COPYING       libcob.h   THANKS
build_aux      ChangeLog   COPYING.DOC   m4          TODO
build_windows  cobc       doc          Makefile.am
cbchelp.tex    config     extras       Makefile.in
cbconf.tex     config.h.in gnucobol.spec po
```

- Type ./configure to begin the configuration process

```
User@WinDev2005Eval ~ /cygdrive/c/gnucobol-3.0-rc1
$ ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /usr/bin/mkdir -p
checking for gawk... gawk
```

- Now type make and press enter

```
User@WinDev2005Eval ~ /cygdrive/c/gnucobol-3.0-rc1
$ make
```

- Confirm that make ran correctly by typing "make check" and pressing enter

```
User@WinDev2005Eval ~ /cygdrive/c/gnucobol-3.0-rc1
$ make check
```

Windows Install

- Install GnuCOBOL by typing "make install" and pressing enter.

```
User@WinDev2005Eval ~ /cygdrive/c/gnucobol-3.0-rc1
$ make install
```

- Check installation by typing "cobic -x"

```
User@WinDev2005Eval ~ /cygdrive/c/gnucobol-3.0-rc1
$ cobc -x
```

- Result should be "no input file" error

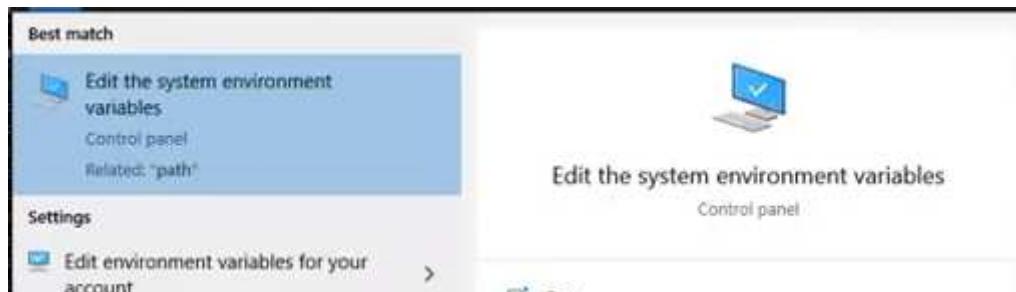
```
User@WinDev2005Eval ~ /cygdrive/c/gnucobol-3.0-rc1
$ cobc -x
cobic: error: no input files
```

- Check location of GnuCOBOL directory

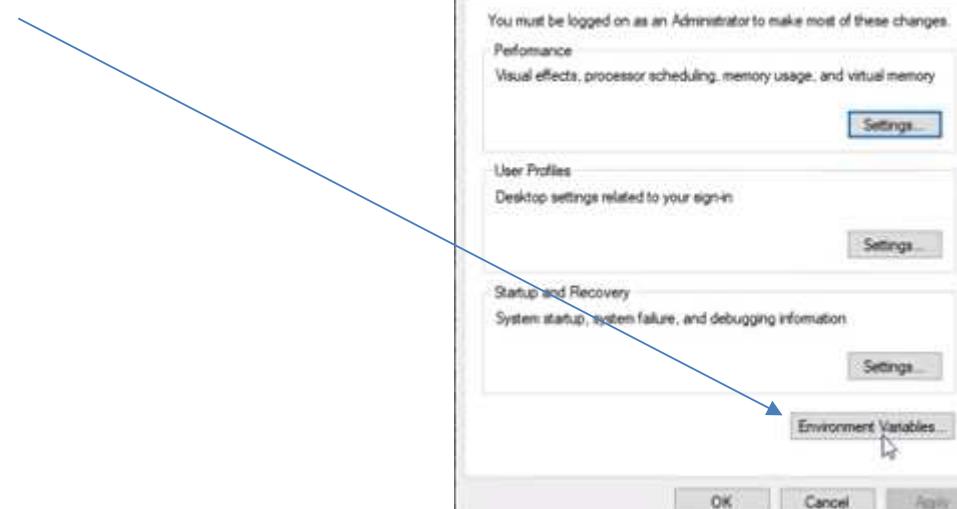
```
User@WinDev2005Eval ~ /cygdrive/c/gnucobol-3.0-rc1
$ whereis cobc
cobc: /usr/local/bin/cobic.exe
```

Windows Install

- To get GnuCOBOL to work on your Windows laptop, you will have to add locations to your path environmental variable.
- Edit environment

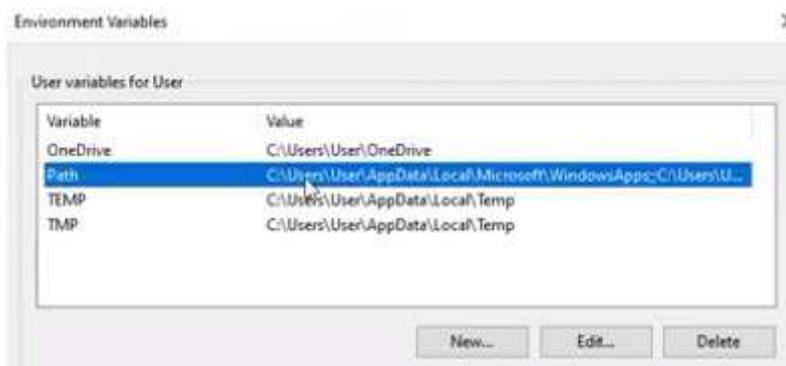


- Select Environment Variables

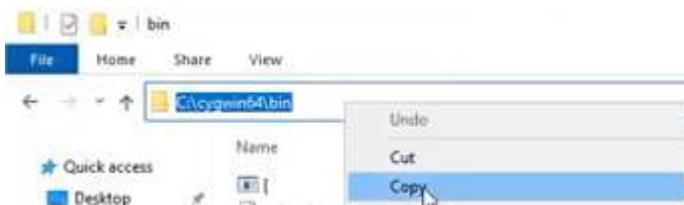


Windows Install

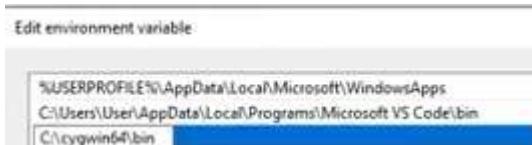
- Edit the path environment variable



- Copy the location of your Cygwin/bin subdirectory



- Select "New" and paste subdirectory into path variable:

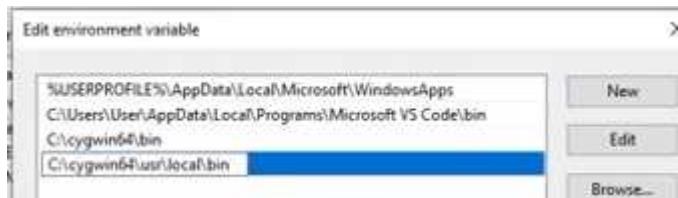


Windows Install

- Now go to Cygwin/user/local/bin and copy that location:



- Once again, press new and paste location into path variable:

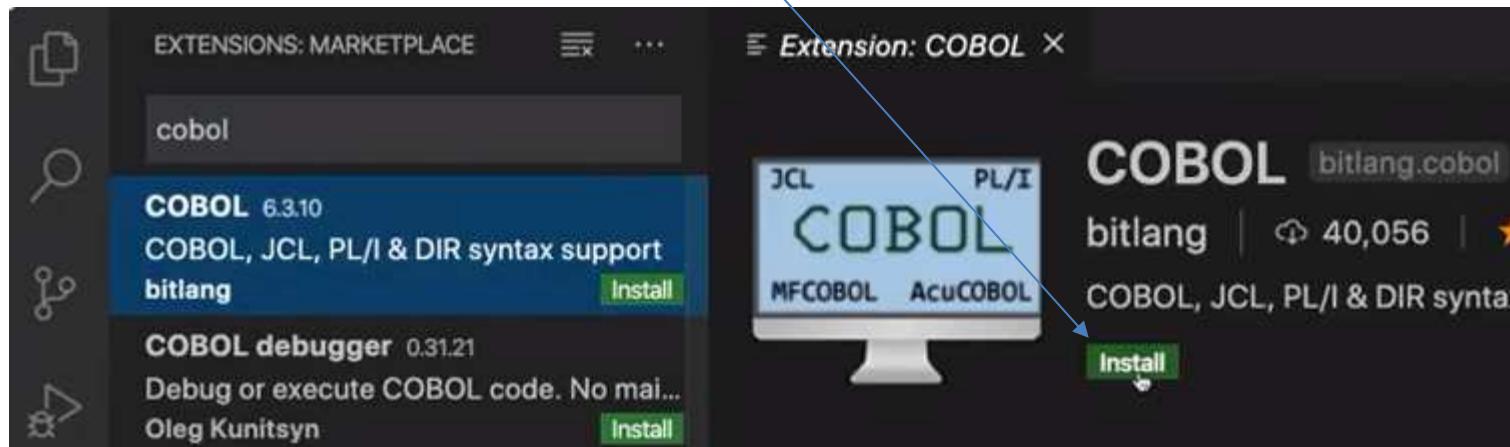


- You are now done editing your environment variable.
- Go into VSCODE and open a terminal and type "cdbc"



Mac Install

- Install Visual Studio Code (code.visualstudio.com)
- While installing, make sure that you select the "add to path" option.
- Open VSCode
- Click on "Extensions" option on the left hand menu.
- Type in "COBOL" in the search box.
- First option should be: "COBOL" from bitlang.
- Click on "Install" green button.
- You have now installed the VS CODE COBOL extension.

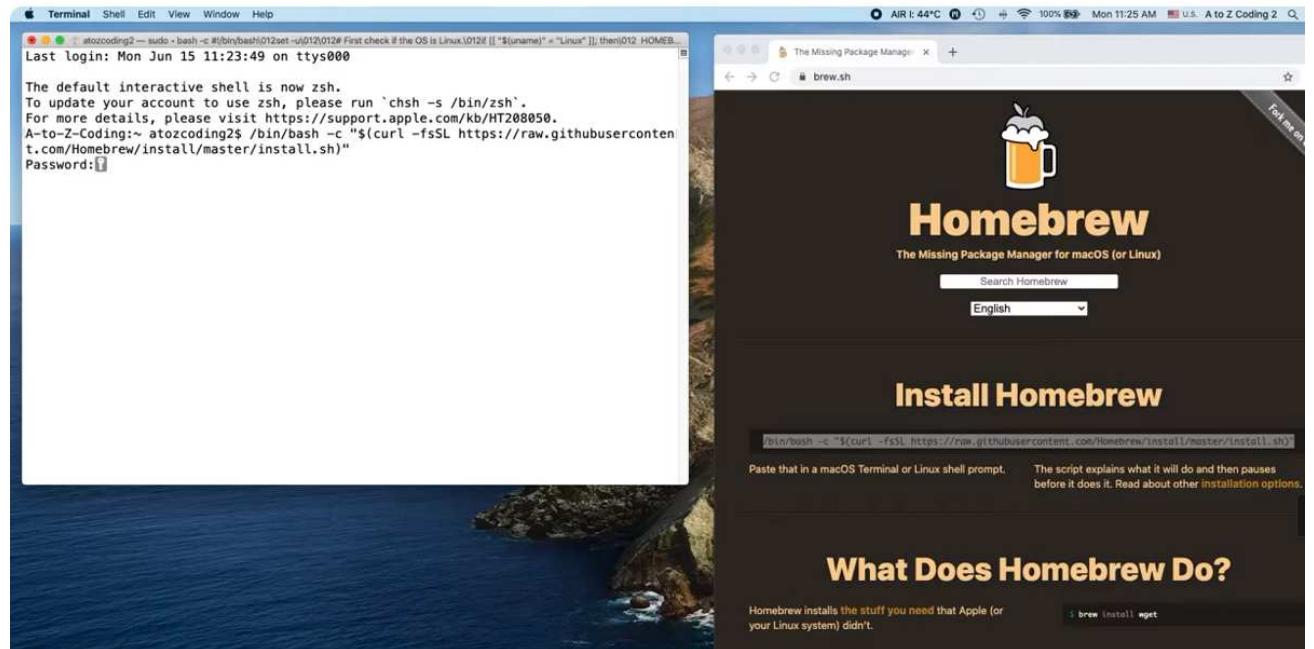


Mac Install

- Go to Homebrew at brew.sh.
- Copy this command:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install.sh)"
```

- Open a terminal and paste the command and execute it.
- Next, type in **brew install gnu-cobol**
- Test installation by typing **cobc -version**



Mac Install

- Go into VS Code and create the following program:

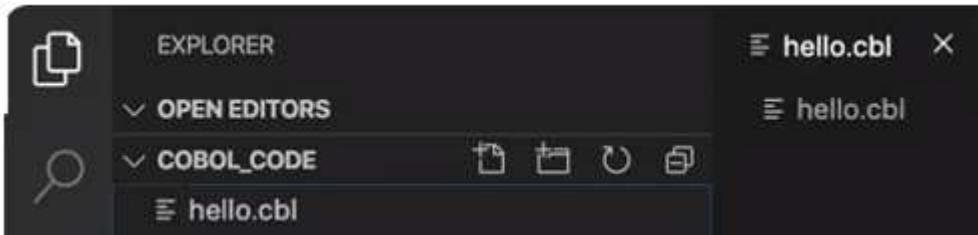
```
1      >>SOURCE FORMAT FREE
2  IDENTIFICATION DIVISION.
3  PROGRAM-ID. hello.
4  PROCEDURE DIVISION.
5  DISPLAY "Hello, world!".
6  STOP RUN.
```

- Save as "hello.cob"
- In terminal, go to subdirectory where you saved hello.cob and type
cobc -x hello.cob
- Run the program by typing "./hello"

```
>> cobc -x hello.cob
>> ./hello
Hello, world!
```

Hello World Program

- Go into VSCode and create a new hello.cbl file



- Create hello world program:

* Hello World Program

IDENTIFICATION DIVISION.

PROGRAM-ID. HELLO-WORLD.

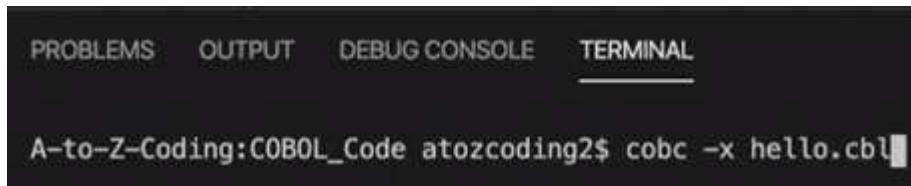
PROCEDURE DIVISION.

DISPLAY "Hello, World!".

STOP RUN.

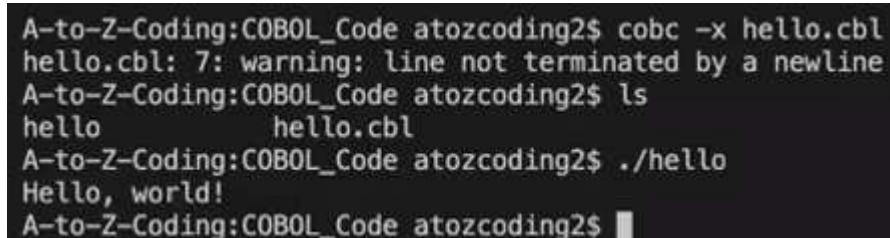
Hello World Program

- Compile program by going to terminal and typing "cobic -x hello.cbl"



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
A-to-Z-Coding:COBOL_Code atozcoding2$ cobic -x hello.cbl
```

- An ".exe" file has been created. Now run it:



```
A-to-Z-Coding:COBOL_Code atozcoding2$ cobic -x hello.cbl
hello.cbl: 7: warning: line not terminated by a newline
A-to-Z-Coding:COBOL_Code atozcoding2$ ls
hello          hello.cbl
A-to-Z-Coding:COBOL_Code atozcoding2$ ./hello
Hello, world!
A-to-Z-Coding:COBOL_Code atozcoding2$
```

Components OF The COBOL Language

Components OF The COBOL Language

- Characters
- Reserved Keywords
- User Defined Words
- Variables, Literals, Structures
- Optional Words
- Constants
- Intrinsic Functions

Characters

- The COBOL language is made up of characters.
- The complete set of characters recognized by COBOL is shown in the table:

Character(s)	Meaning
A – Z	Alphabets (Upper case)
a – z	Alphabets (Lower case)
0 – 9	Digits
+	Plus (For addition)
-	Minus (For subtraction) or Hyphen
*	Asterisk (For multiplication)
/	Slant, Stroke or Slash (For division)
=	Equal to sign
\$	Dollar (For currency sign)
,	Comma
;	Semicolon
.	Decimal point or Period
"	Quotation mark
(Left parenthesis
)	Right Parenthesis
>	Greater than
<	Less than
:	Colon
'	Apostrophe

Note: Single quote →

Let's Talk About Periods

- In COBOL, periods (.) serve as statement terminators and logical delimiters—but their usage has some quirks and historical baggage.
- In traditional COBOL, each paragraph, sentence, or section usually ends with a period:
`DISPLAY "HELLO, WORLD".`
`MOVE ZERO TO TOTAL.`
- A period tells the compiler: "This is the end of a complete statement or block."
- When you define a **paragraph** or **section**, COBOL expects a period after the last statement in that block:
`MAIN-PARAGRAPH.`
`DISPLAY "This is the main paragraph".`
`STOP RUN.`
- Periods end all open IFs, EVALUATEs, or PERFORMs, which can lead to logic issues if placed improperly.
- At the end of a **division**, **section**, or program, periods can help denote the end of a unit

Reserved Words In COBOL

1. Keywords
2. Optional Words
3. Figurative Constants
4. Special Character Words
5. Special Object Identifiers
6. Special Registers

Keywords

- Def: Reserved words that are required within a given clause, entry, or statement. Keywords appear in UPPERCASE.
- Example:
 - **ADD** – add two or more variables
 - **DELETE** – delete a record from a file
 - **SEARCH** – search for a value in an area or a table
 - **READ** – read data from a file
 - **WRITE** – write a record to a file
 - **CALL** – transfer control to a subprogram from the main program

```
PROCEDURE DIVISION.

001507*
001508* Read data from the Employee Month File.
001509*
001510 000030-READ-TREMP.
001520      READ EMPRC AT END
001530      MOVE 'Y'      TO EOF-OF-FILE
001540      END-READ
001550      ...
001570      ADD 1          TO TR-INREC
      ...
```

Optional Words

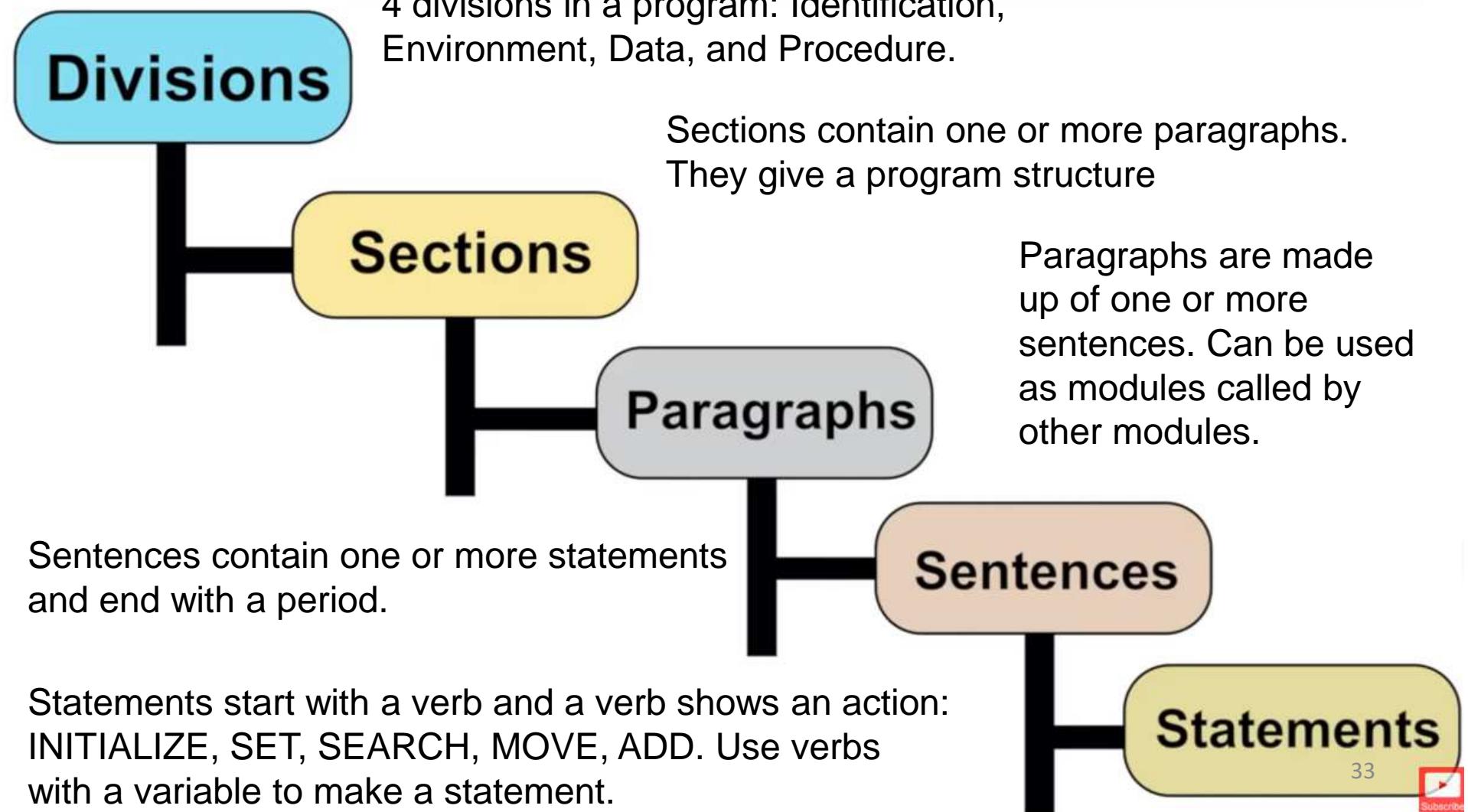
- Def: Reserved words that can be included in the format of a clause, entry, or a statement in order to improve readability.
- They possess no significance and have no effect on the program execution.
- Example: **GIVING**, **ROUNDOFF**, **AFTER**, etc.

```
PROCEDURE DIVISION.

001507*
001508* Read data from the Employee Month File.
001509*
001510  ADD WS-BONUS      TO WS-EMP-SAL GIVING EMP-SAL.
001511*
001512  PERFORM WITH TEST AFTER UNTIL WS-MNTH > 12
001513      SUBTRACT 1 TO WS-MNTH
001514  END-PERFORM.
```

COBOL Program Structure

Hierarchical Structure Of A COBOL Program



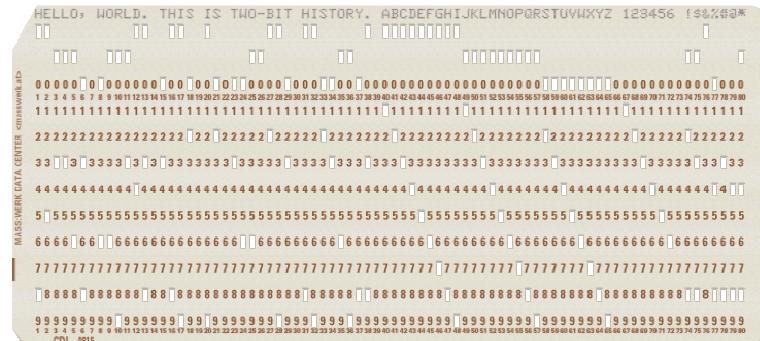
Overall Structure Of A COBOL Program

- A COBOL program is divided into 4 logical divisions.
- Each division is then divided into sections, paragraphs, sentences, and statements.
- **Identification Division (MANDATORY)**
 - Used to identify the program to the operating system
 - Used for documentation purpose
- **Environment Division (Optional)**
 - Specify file name and specific computer equipment that will be used by program
- **Data Division (Optional)**
 - Describes input/output formats to be used by program
 - Define constants and work areas
- **Procedure Division (Optional)**
 - Contains business logic to process input and create output

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. PGMDSP01.  
AUTHOR. Topictrick.  
INSTALLATION. Topictrick.  
DATE-WRITTEN. 08/21/2002.  
DATE-COMPILED. 08/21/2002.  
* PROJ DESC : SAMPLE COBOL PROGRAM TO DISPLAY EMPLOYEE  
* : NAME IN SPOOL.  
*  
ENVIRONMENT DIVISION.  
INPUT-OUTPUT SECTION.  
*  
DATA DIVISION.  
FILE SECTION.  
WORKING-STORAGE SECTION.  
01 WS-EMP-NAME.  
05 WS-FNAME  
05 FILLER  
05 WS-LNAME  
*  
PROCEDURE DIVISION.  
000-MAIN-LOGIC.  
PERFORM 005-PRT-EMP-DATA.
```

COBOL Coding Rules

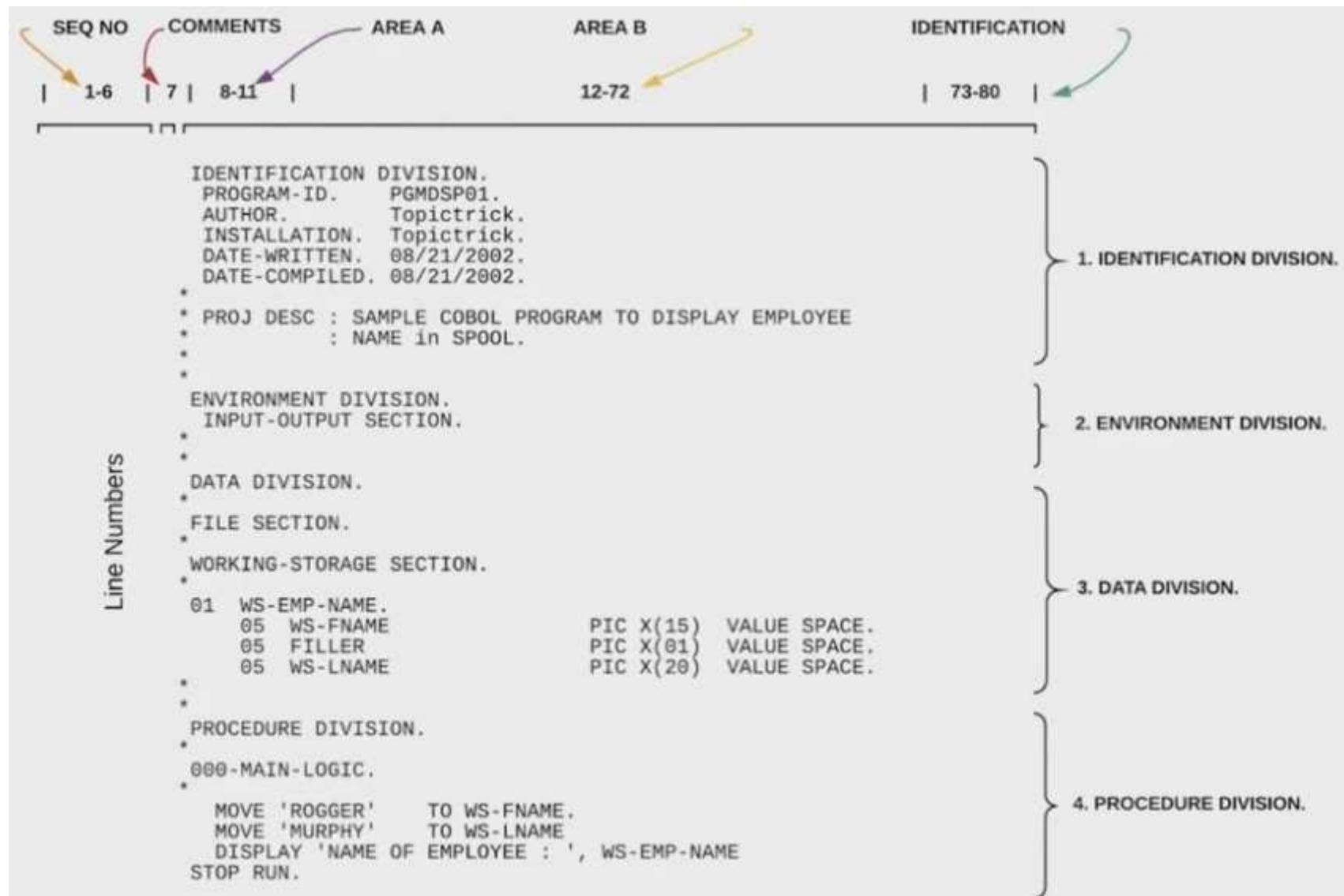
- In the 1960's, computer programs were created on punch cards which were then fed into a computer where the cards were read, the program loaded, and then run.



COBOL Program Formatting

01 - 80				
Sequence No		Section A	Section B	Identification Area
---- 1-6 ----	- 7 -	---- 8-11 ----	---- 12 - 72 ----	---- 73-80 ----
Sequence number is added to each coding line when the program is compiled.	Continuation, Comments, or form feeds.	Used for special entries such as DVISION, SECTION, Paragraph name and some items of the DATA Division.	Used for most COBOL entries, including PROCEDURE DIVISION sentence.	Used in olden days for identification purpose.

Example COBOL Program



Free Format COBOL

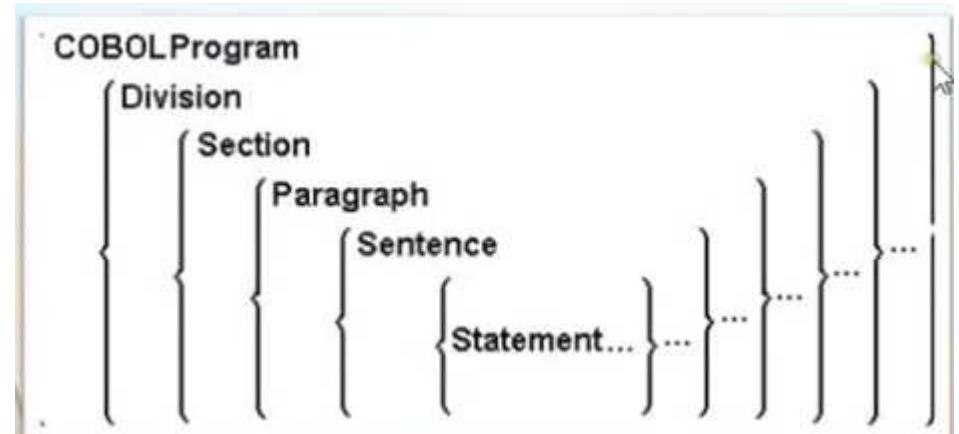
- By default, GnuCOBOL assumes **fixed-format** unless explicitly told otherwise.
- The directive "**>>SOURCE FORMAT FREE**" tells the compiler to **interpret the code using free-format rules**.
- Modern COBOL (especially with COBOL 2002 and later compilers like GnuCOBOL or Micro Focus) supports:
 - No fixed columns
 - No need for sequence numbers or special column positioning
 - Indentation and spacing become stylistic, not required
 - Permits lowercase and modern-style syntax
 - Source files can use .cbl, .cob, or even .cobol



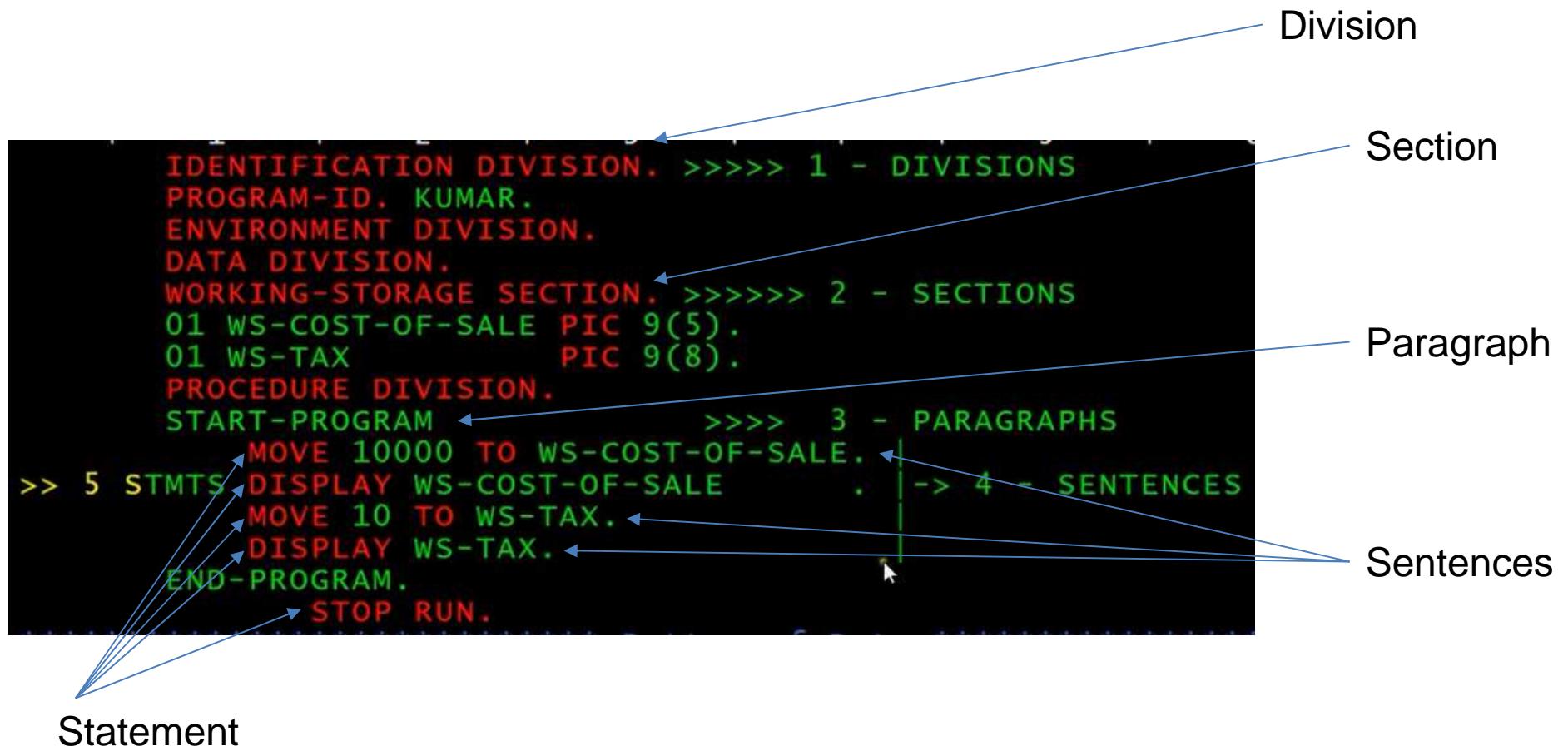
Image Credit: <https://thedataScientist.com/cobols-comeback-can-it-future-proof-your-systems/>

Components Of A COBOL Program

Character	Digits (0-9), Alphabets (A-Z), Space (b), Special Characters (+ - * / () = \$; " > < . ,)
Word	One or more characters- User defined or Reserved
Clause	One or more words. It specifies an attribute for an entry
Statement	One or more valid words and clauses
Sentence	One or more statements terminated by a period
Paragraph	One or more sentences.
Section	One or more paragraphs.
Division	One or more sections or paragraphs
Program	Made up of four divisions



Example Components



Identification Division

- This is the first division in a COBOL program.
- Its purpose is to identify the program.
- **The IDENTIFICATION Division is MANDATORY for a COBOL program.**
- The paragraph "PROGRAMID." followed by a program name is **MANDATORY**.
 - The PROGRAMID parameter is used to specify the name of the COBOL program. It can consist of 1 to 30 characters.
- All other paragraphs are optional and are used for documentation:
 - AUTHOR
 - DATE-WRITTEN
 - DATE-COMPILED
 - SECURITY

```
IDENTIFICATION DIVISION.  
PROGRAM-ID.      PGMDSP01.  
AUTHOR.          Topictrick.  
INSTALLATION.    Topictrick.  
DATE-WRITTEN.    08/21/2002.  
DATE-COMPILED.   08/21/2002.
```

Environment Division

- This section is related to the program's environment and includes such items as computer, hardware, and files that will be used.
- Division includes:
 - Configuration Section – where the program will be compiled [NOT REQUIRED]
 - SOURCE-COMPUTER
 - OBJECT-COMPUTER
 - Input-Output Section – Files used in the program [REQUIRED]
 - FILE-CONTROL
 - I-O CONTROL

Data Division

- This division lists every data item that will be processed by the program
- The variables that will be used in the PROCEDURE Division need to be declared here.
- Data is divided into two separate types: temporary and permanent.
- Temporary Data
 - Available only during the execution of the program
- Permanent
 - File section
 - Working-storage section
 - Linkage section

WORKING STORAGE Section

- Used for declaring user variables or data names.
- Variable naming standards:
 - Must start with an alphabetic character
 - Name should contain between 1-30 alphanumeric characters
 - No space permitted between characters
 - No reserved words: ex. TIME, ADD, COMPUTE
 - Hyphens can be used but cannot be first character and cannot be consecutive ("--").
 - Not case sensitive: TOTAL and ToTaL are the same variable.
 - No special characters: \$, #, etc.

```
DATA DIVISION.  
WORKING-STORAGE SECTION.  
 01 COUNTER      PIC 9(3) VALUE 0.  
 01 CUSTOMER-NAME PIC X(30).  
 01 TOTAL-AMOUNT PIC 9(5)V99 VALUE 0.
```

LOCAL STORAGE Section

- The Local-Storage Section is like the Working-Storage Section’s “short-term memory.”
- Local-Storage is allocated each time the program is called and will be de-allocated when the program stops via an EXIT PROGRAM, GOBACK, or STOP RUN.
- It is defined in the DATA DIVISION after WORKING-STORAGE SECTION
- It’s where you declare variables that:
 - Are freshly allocated every time the program, subprogram, or paragraph is entered.
 - Are reinitialized to their defined VALUE (or spaces/zeros if no VALUE) on each entry.
 - Do not retain values between calls.

```
DATA DIVISION.  
LOCAL-STORAGE SECTION.  
 01 TEMP-VAR PIC 9(3) VALUE 0.  
  
PROCEDURE DIVISION.  
  ADD 1 TO TEMP-VAR  
  DISPLAY "TEMP-VAR is " TEMP-VAR.  
  STOP RUN.
```

LINKAGE SECTION

- The LINKAGE SECTION is the part of the Data Division used in a called program (subprogram) to define variables that are passed in from the calling program.
- Think of it as the "bridge" that lets two COBOL programs share data at runtime.
- When one COBOL program calls another (using the CALL statement), it can pass parameters (values or addresses of data items).
- The called program declares those incoming parameters in its LINKAGE SECTION, so it knows their structure and can work with them.
- The LINKAGE SECTION is:
 - Declared after WORKING-STORAGE SECTION in the Data Division.
 - Used only in subprograms, not in the main program.
 - Variables here are not allocated storage by the subprogram itself — instead, they point to storage owned by the calling program.

LINKAGE SECTION: Main Program (Caller)

IDENTIFICATION DIVISION.

PROGRAM-ID. MAINPROG.

DATA DIVISION.

WORKING-STORAGE SECTION.

01 WS-NAME PIC X(20) VALUE "Alice".

01 WS-AMOUNT PIC 9(5)V99 VALUE 123.45.

PROCEDURE DIVISION.

CALL 'SUBPROG' USING WS-NAME WS-AMOUNT.

STOP RUN.

LINKAGE SECTION: Subprogram (Called Program)

IDENTIFICATION DIVISION.

PROGRAM-ID. SUBPROG.

DATA DIVISION.

LINKAGE SECTION.

01 LK-NAME PIC X(20).

01 LK-AMOUNT PIC 9(5)V99.

PROCEDURE DIVISION USING LK-NAME LK-AMOUNT.

DISPLAY "Customer: " LK-NAME

DISPLAY "Amount: " LK-AMOUNT

EXIT PROGRAM.

LINKAGE SECTION: How It Works

1. MAINPROG calls SUBPROG and passes two variables.
2. SUBPROG declares those in the LINKAGE SECTION.
3. The PROCEDURE DIVISION USING clause ties them together.
4. Any change in SUBPROG to LK-NAME or LK-AMOUNT updates the original variables in MAINPROG (since it's the same memory).

Copybook

- In COBOL, a **copybook** is a reusable file that contains predefined COBOL code, most commonly data declarations.
- It's used to promote code reuse, standardization, and maintainability across programs—especially when the same data structures or procedures need to be shared by multiple COBOL programs.
- What a Copybook Typically Contains
 - Data definitions, such as record layouts for files or screen fields.
 - Sometimes procedural code, but that's less common.
- You include a copybook in a COBOL program using the COPY directive:

```
WORKING-STORAGE SECTION.  
COPY CUSTOMER-RECORD.
```

- This tells the compiler to insert the contents of the file CUSTOMER-RECORD (usually stored as a .cpy file) at that location.

COPY Command

- COBOL supports compiler directives that tell the compiler to insert the contents of an external copybook into a program at compile time.
- Happens at compile time – the preprocessor stage, not runtime.
- A copybook file is just plain COBOL source code.
- Syntax: COPY copybook-name
- Example: **COPY CUSTOMER-RECORD**
- Generally used in either the DATA or PROCEDURE Division

COPY With REPLACING

- The COPY ... REPLACING option is used to modify text from a copybook as it's brought into your program, without having to change the original copybook file.
- Sometimes a copybook contains reusable code, but certain parts—like variable names, constants, or paragraph names—need to be adjusted for a specific program.
- Instead of making a separate copy of the copybook and editing it (which creates maintenance headaches), you can use REPLACING to swap text as it's copied in.
- Syntax
 - **COPY copybook-name REPLACING old-text BY new-text.**
- You can use:
 - ==old== BY ==new== for pseudo-text replacement (most common).
 - identifier-1 BY identifier-2 for replacing specific identifiers.

COPY With REPLACING

- Example

Copybook: EMPDATA.cpy

```
01 EMP-NAME  PIC X(30).  
01 EMP-NUMBER  PIC 9(5).
```

- Main Program:

```
COPY EMPDATA  
      REPLACING ==EMP== BY ==CUST==.
```

- What the compiler sees after replacement:

```
01 CUST-NAME  PIC X(30).  
01 CUST-NUMBER  PIC 9(5).
```

Procedure Division

- The **Procedure Division** is the part of the program where the **actual processing logic** is written — the instructions that tell the program what to do with the data.
- It's where you:
 - Control the flow of execution (using PERFORM, IF, EVALUATE, GO TO, etc.).
 - Perform calculations and data manipulation.
 - Handle input/output operations (read from files, write to files, display output).
 - Implement business rules and error handling (including declaratives).

```
PROCEDURE DIVISION.  
MAIN-PARA.  
  OPEN INPUT CUSTOMER-FILE  
  READ CUSTOMER-FILE AT END  
    DISPLAY "No more records"  
    STOP RUN  
  END-READ  
  PERFORM PROCESS-RECORD  
  CLOSE CUSTOMER-FILE  
  STOP RUN.  
  
PROCESS-RECORD.  
  DISPLAY "Processing record..."
```

DECLARATIVE Section

- Declaratives are special sections in the Procedure Division that automatically execute in response to certain conditions, usually file I/O exceptions or report events.
- Each declarative section starts with a use statement (e.g., USE AFTER ERROR ON file-name) and ends when another section begins or when END DECLARATIVES is reached.
- Inside a declarative section, you can use normal COBOL procedural code — including PERFORM statements that call other paragraphs or sections.
- The only difference is when that code runs — it's automatically triggered by the specified event, not manually.

```
DECLARATIVES.  
FILE-ERROR SECTION.  
  USE AFTER STANDARD ERROR PROCEDURE ON CUSTOMER-FILE.  
FILE-ERROR-PARA.  
  DISPLAY "Error accessing CUSTOMER-FILE".  
  PERFORM LOG-ERROR.  
STOP RUN.  
END DECLARATIVES.  
  
LOG-ERROR.  
  DISPLAY "Logging error details...".
```

Note: Here, LOG-ERROR is outside the declarative section, but it's performed from inside the declarative paragraph. This is perfectly valid.

Paragraphs In COBOL Programs

- A "paragraph" is a named section of code that performs a specific task within the a division or a section of the program.
- It is typically used within the PROCEDURE DIVISION.
- It has a name
- It is followed by one or more COBOL statements
- Ends when the next paragraph starts or a STOP-RUN or EXIT is encountered.
- You can use a PERFORM statement to call/execute a paragraph.
- COBOL uses "gravity driven programming"
- Execution of a program "falls through" the program until a condition or a GOTO redirects the flow to a different part of our file.
- An "open paragraph" is an example of how we "fall through code"
- A "closed paragraph" is executed by name

Paragraphs In COBOL Programs

- In a COBOL program, the PROCEDURE Division will consist mainly of a collection of paragraphs.
- These paragraphs are reusable.
- When you run a COBOL program, the program will be executed paragraph by paragraph in sequential order.
- To EXECUTE a specific paragraph, the PERFORM clause is used.

Paragraph Example

```
11 SubOne.  
12     DISPLAY "In Paragraph 1"  
13     PERFORM SubTwo  
14     DISPLAY "Returned to Paragraph 1"  
15     PERFORM SubFour 2 TIMES.  
16     STOP RUN.  
17     END-PERFORM  
18     STOP RUN.  
19  
20 SubThree.  
21     DISPLAY "In Paragraph 3".  
22  
23 SubTwo.  
24     DISPLAY "In Paragraph 2"  
25     PERFORM SubThree  
26     DISPLAY "Returned to Paragraph 2".  
27  
28 SubFour.  
29     DISPLAY  
30  
31 STOP RUN.
```

In Paragraph 1
In Paragraph 2
In Paragraph 3
Returned to Paragraph
2
Returned to Paragraph
1
Repeat

Strings

COBOL String Manipulation



COBOL String Handling

- STRING Clause: String statements concatenate two or more sending fields into one receiving field.
 - Key words used with the STRING clause:
 - STRING
 - DELIMITED BY
 - WITH POINTER
 - END STRING
- UNSTRING Clause: the UNSTRING statement unstrings a field into the fields that are listed in the INTO clause.
- INSPECT Clause: The INSPECT statement allows you to count characters in a field or replace characters in a field.
- REFERENCE Modification: The REFERENCE modification feature lets you refer to a specific location within a field.

String Example

Means copy the entire variable
- all 20 characters, even blanks

```
WORKING-STORAGE SECTION.  
 01 WS-CUST-NAME      PIC X(40).  
 01 WS-CUST-FIRST-NM  PIC X(20) VALUE IS "VIJAY".  
 01 WS-CUST-LAST-NM  PIC X(20) VALUE IS "KUMAR".  
 01 WS-CUST-POINT    PIC 9(2) VALUE 05.  
PROCEDURE DIVISION.  
  STRING  
    WS-CUST-FIRST-NM , WS-CUST-LAST-NM  
    DELIMITED BY SIZE  
    INTO  
    WS-CUST-NAME  
    WITH POINTER WS-CUST-POINT  
    END-STRING.  
    DISPLAY WS-CUST-NAME.
```

Start placing "Vijay Kumar" into WS-CUST-NAME at location 5.

String POINTER

- The **POINTER** phrase is used with the **STRING** (and sometimes **UNSTRING**) statement. Its purpose is to keep track of the current character position in the receiving data item while concatenating or manipulating strings.
 - It specifies a numeric data item (an integer variable) that holds the current character position within the destination field.
 - COBOL updates this pointer automatically as STRING operations progress.
 - You can initialize it before the STRING statement, and afterward, it tells you where the last character was placed.
- Syntax:

```
STRING
  source-field-1 DELIMITED BY SPACE
  source-field-2 DELIMITED BY SIZE
  INTO destination-field
  WITH POINTER pointer-var
  ON OVERFLOW
    DISPLAY "STRING TOO LONG"
END-STRING.
```

Why it's useful

- You can build strings in multiple steps, resuming at the correct position without overwriting what was already concatenated.
- Lets you check where the STRING ended, which is helpful if you need to append more later.
-  In short: The POINTER in STRING ensures concatenated data goes into the right place and tells you where to continue if you want to add more data.

String POINTER: Example

```
WORKING-STORAGE SECTION.  
01 NAME1          PIC X(10) VALUE "John".  
01 NAME2          PIC X(10) VALUE "Smith".  
01 FULL-NAME      PIC X(25).  
01 PTR            PIC 9(3) VALUE 1.  
  
PROCEDURE DIVISION.  
  STRING  
    NAME1 DELIMITED BY SPACE  
    " "    DELIMITED BY SIZE  
    NAME2 DELIMITED BY SPACE  
    INTO FULL-NAME  
    WITH POINTER PTR  
  END-STRING  
  
  DISPLAY "Full Name: " FULL-NAME  
  DISPLAY "Pointer now at: " PTR.
```

Output:

```
Full Name: John Smith  
Pointer now at: 11
```

Unstring

- The UNSTRING command is used to de-concatenate a string into multiple variables.
- The UNSTRING command should have one sending field and multiple receiving fields.
- Key words used with UNSTRING are:
 - UNSTRING
 - DELIMITED BY
 - INTO
 - WITH POINTER
 - END UNSTRING

UNSTRING Example

```
WORKING-STORAGE SECTION.  
 01 WS-CUST-NAME      PIC X(40) VALUE IS "VIJAY,KUMAR".  
 01 WS-CUST-FIRST-NM  PIC X(20).  
 01 WS-CUST-LAST-NM  PIC X(20).  
PROCEDURE DIVISION.  
  UNSTRING  
    WS-CUST-NAME  
  DELIMITED BY ","  
  INTO  
    WS-CUST-FIRST-NM , WS-CUST-LAST-NM  
  END-UNSTRING.  
  DISPLAY WS-CUST-FIRST-NM.  
  DISPLAY WS-CUST-LAST-NM.  
  STOP RUN.
```

Editing Strings

```
8 WORKING-STORAGE SECTION.  
9 01 StartNum PIC 9(8)V99 VALUE 00001123.55.  
10 01 NoZero PIC ZZZZZZZ9.99.  
11 01 NoZPlusC PIC ZZ,ZZZ,ZZ9.99.  
12 01 Dollar PIC $$,$$$,$$9.99.  
13 01 BDay PIC 9(8) VALUE 12211974.  
14 01 ADate PIC 99/99/9999.  
15 PROCEDURE DIVISION.  
16 MOVE StartNum TO NoZero  
17 DISPLAY NoZero  
18 MOVE StartNum TO NoZPlusC  
19 DISPLAY NoZPlusC  
20 MOVE StartNum TO Dollar  
21 DISPLAY Dollar  
22 MOVE BDay TO ADate  
23 DISPLAY ADate  
24 STOP RUN.
```

1123.55
1,123.55
\$1,123.55
12/21/1974

Editing Strings: Starting Variables

```
8 WORKING-STORAGE SECTION.  
9 01 SampStr      PIC X(18) VALUE 'eerie beef sneezed'.  
10 01 NumChars     PIC 99 VALUE 0.  
11 01 NumEs       PIC 99 VALUE 0.  
12 01 FName        PIC X(6) VALUE 'Martin'.  
13 01 MName        PIC X(11) VALUE 'Luther King'.  
14 01 LName        PIC X(4) VALUE 'King'.  
15 01 FLName       PIC X(11).  
16 01 FMLName      PIC X(18).  
17 01 SStr1        PIC X(7) VALUE "The egg".  
18 01 SStr2        PIC X(9) VALUE "is #1 and".  
19 01 Dest         PIC X(33) VALUE "is the big chicken".  
20 01 Ptr          PIC 9 VALUE 1.  
21 01 SStr3        PIC X(3).  
22 01 SStr4        PIC X(3).
```

String Processing

```
26 INSPECT SampStr TALLYING NumEs FOR ALL 'e'.
27 DISPLAY "Number of Es : " NumEs.
28 DISPLAY FUNCTION UPPER-CASE(SampStr)
29 DISPLAY FUNCTION LOWER-CASE(SampStr)
30
31 STRING FName DELIMITED BY SIZE
32 SPACE
33 LName DELIMITED BY SIZE
34 INTO FLName.
35 DISPLAY FLName.
36 STOP RUN.
```

```
Number of Characters
: 18
Number of Es : 08
EERIE BEEF SNEEZED
eerie beef sneezed
Martin King
```

Means "get the whole thing"

String Processing

```
37 STRING FLName DELIMITED BY SPACES
38 SPACE
39 MName DELIMITED BY SIZE
40 SPACE
41 LName DELIMITED BY SIZE
42 INTO FMLName
43 ON OVERFLOW DISPLAY 'Overflowed'.
44 DISPLAY FMLName.
```

Overflowed
Martin Luther King

String Processing

```
46 STRING SStr1 DELIMITED BY SIZE
47 SPACE
48 SStr2 DELIMITED BY "#"
49 INTO Dest
50 WITH POINTER Ptr
51 ON OVERFLOW DISPLAY 'Overflowed'.
52 DISPLAY Dest.
```

The egg is chicken

String Processing

```
54 UNSTRING SStr1 DELIMITED BY SPACE
55 INTO SStr3, SStr4
56 END-UNSTRING.
57 DISPLAY SStr4.
```

egg

Numbers In COBOL

Number Accuracy In COBOL

- Most computer languages used floating point calculations.
- Problem: this can introduce errors due to how computers represent real numbers. Example: $01.+0.2 = 0.3000000000000004$
- COBOL uses fixed point decimal arithmetic.
- Fixed point decimal arithmetic is a method of representing numbers that have a fixed number of digits after the decimal point.
- Fixed point keeps the decimal point in a constant position.
- Fixed point is precise and quantifiable especially for money and quantities.
- COBOL allows you to specify how you want to round values

Financial Calculations

```
8 WORKING-STORAGE SECTION.  
9 01 Price PIC 9(4)V99.  
10 01 TaxRate PIC V999 VALUE .075.  
11 01 FullPrice PIC 9(4)V99.  
12 PROCEDURE DIVISION.  
13 DISPLAY "Enter the Price : " WITH NO ADVANCING  
14 ACCEPT Price  
15 COMPUTE FullPrice ROUNDED = Price + (Price *  
    TaxRate)  
16 DISPLAY "Price + Tax : " FullPrice.  
17 STOP RUN.
```

```
Enter the Price : 4567  
.98  
Price + Tax : 4910.49
```

Value Clause

- The VALUE clause is an optional clause that is used to specify the initial value for the data item.
- The initial value can be a numeric literal, non-numeric literal, or a figurative constant.
- This initial value can be changed in the program's PROCEDURE DIVISION based on user requirements.
- The initial value of a data-item/variable can be changed or modified under the following scenarios:
 - Due to calculations
 - Data movement

Value Clause Example

WORKING-STORAGE SECTION.

01 WS-TEMP-VAL.

05 WS-EMP-REC-IN	PIC 9(06)	VALUE ZEROES.
05 WS-ERR-MSG	PIC X(12)	VALUE SPACES.
05 WS-TOT-CR-AMT	PIC S9(08)v99.	VALUE +0.
05 WS-TOT-DR-AMT	PIC S9(08)v99.	
05 WS-EMP-DPT	PIC A(30).	
01 WS-EMPL-SALARY	PIC Z,ZZZ,ZZZ.99.	

Initial values

WORKING-STORAGE SECTION.

01 WS-EMP-REC.

05 WS-EMP-NO	PIC 9(06).	VALUE SPACE.
05 WS-EMP-FNAME	PIC X(12)	VALUE SPACES.
05 WS-EMP-LNAME	PIC X(14)	VALUE 'MED'
05 WS-EMP-DEPT	PIC A(03)	
05 WS-EMP-SAL	PIC S9(10)v9(02).	
77 TOT-REJ-REC-CNT	PIC 9(05).	
01 TOT-REC-CNT	PIC 9(05)	VALUE ZEROES.

Initial values

Note single quotes

Value Clause Notes

- Alphanumeric items: unused spaces to the right are set to spaces
- Numeric items: unused spaces to the left are set to "0"
- If you try to initialize a variable to a value that is larger than it has been configured for, this will cause a compile error.
- Numeric Edited items generally typically receive a value as the result of a MOVE statement, therefore these items are usually not defined with a VALUE clause.

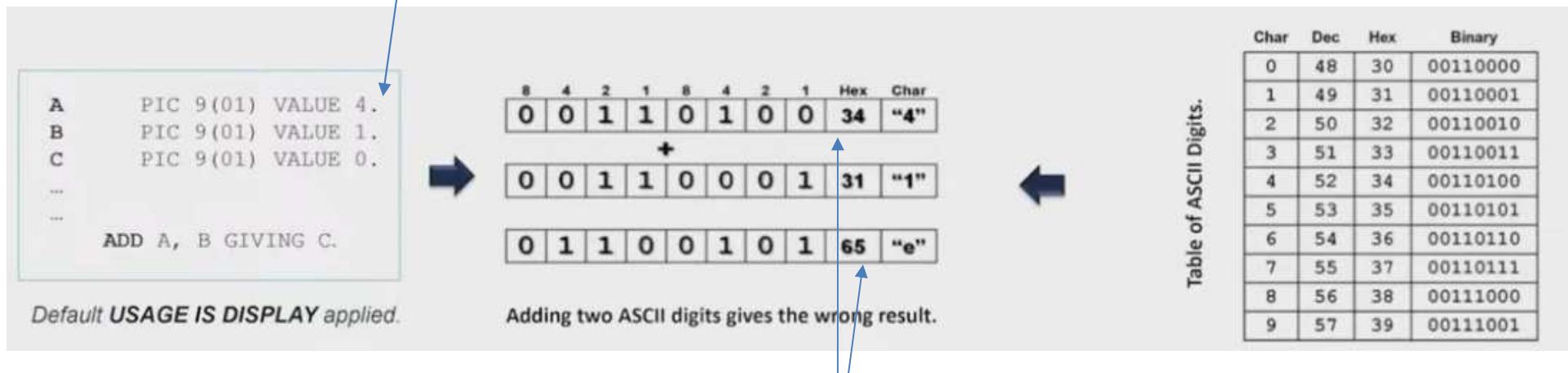
USAGE Clause (Data Format)

- Computers store data in the form of binary digits.
- Except for positive integers, all other data uses some type of formatting convention.
- Example: text data is stored using ASCII.
- COBOL provides a programmer with a great deal of control over how a program's data will be stored.
- The USAGE clause is used to specify how a data item is to be stored in the computer's memory.
- Every data item declared in a COBOL program has a USAGE clause even if you don't specify it.
- The default USAGE clause is DISPLAY.
- Data that is heavily used in computation is declared using a usage optimized for computation such as USAGE IS COMPUTATIONAL.

SYSTEM	CHAR	HEX	DEC	8	4	2	1	8	4	2	1
ASCII	"A"	41	65	0	1	0	0	0	0	1	

USAGE Clause Example

No USAGE clause. This means that the numbers will be stored as ASCII characters.



The addition of A and B will not give 5 – instead the ASCII values of "4" and "1" will be added.

This error will require the ASCII values to be converted to binary, added, and then converted back into ASCII – this takes a great deal of time.

USAGE Clause Notes

- The default USAGE representation: USAGE IS DISPLAY used by COBOL for numeric data items can negatively impact the speed of computations.
- The USAGE clause may be used with any data description entry except those with level numbers of 66 or 88.
- When the usage clause is declared for a group item, the usage specified is applied to every item in the group. The group item itself is still treated as an alphanumeric data item.
- The USAGE clause of an elementary item cannot override the USAGE clause of the group to which it is subordinate.
- USAGE IS COMPUTATIONAL and COMP and BINARY are synonyms of each other.
- The USAGE IS INDEX clause is used to provide an optimized table subscript
- The PICTURE clause used for COMP and PACKED-DECIMAL items must be numeric and the picture string can contain only the symbols 9, S, V and/or P

USAGE Example

Syntax – Usage Clause.

USAGE IS
 BINARY
 COMPUTATIONAL
 COMP
 INDEX
 PACKED-DECIMAL
 DISPLAY

Default: DISPLAY

Example – Usage Clause.

WORKING-STORAGE SECTION.

```
*  
01 WS-EMP-NAME      PIC X(20)      VALUE SPACES.  
01 WS-EMP-SAL       PIC S9(05) COMP-3  VALUE ZEROES.  
01 WS-EMP-TAX        PIC 9(02) COMP  VALUE ZEROES.  
01 WS-EMP-NAME      PIC X(30)  USAGE IS DISPLAY.
```

compressed binary decimal form

computational

COMP In COBOL

- COMP, COMP-1, and COMP-2 are USAGE clauses that tell the compiler how to store a numeric value in memory — they control the internal representation and how much precision you get.
- 1. COMP (Computational)
 - Purpose: Store numeric data in binary (integer) format for faster arithmetic.
 - Storage size: Depends on PIC (picture) clause length and compiler/system (often 2, 4, or 8 bytes).
 - Use cases: Integer math, counters, indexes.
 - Example: **01 WS-COUNT PIC S9(4) COMP.**
 - Stores up to ± 9999 in binary form.

COMP SYNC

- COMP SYNC (or COMP, SYNCHRONIZED) is a USAGE clause option that affects how binary numeric data is aligned in memory.
- Breaking it down
 - COMP → means the item is stored in binary (integer) form for faster arithmetic.
 - SYNCHRONIZED (often shortened to SYNC) → tells the compiler to align the variable in memory to a natural boundary (like 2-, 4-, or 8-byte boundaries) so the CPU can access it faster.
- Why alignment matters
 - Some systems (especially older mainframes or certain hardware architectures) access binary data more quickly — or only correctly — if the data is aligned to a specific memory boundary.

```
01 EMPLOYEE-RECORD.  
      05 EMP-ID          PIC 59(4) COMP SYNC.  
      05 EMP-NAME        PIC X(20).
```

COMP In COBOL

- 2. COMP-1
 - Purpose: Store single-precision floating-point numbers.
 - Storage size: Always 4 bytes (IEEE single precision or equivalent).
 - Use cases: Real numbers that need fractional precision but not extremely high precision.
 - Notes: Precision is about 6–7 decimal digits; exact decimal storage is not guaranteed — it's binary floating point.
 - Example: **01 WS-SINGLE USAGE COMP-1.**
- 3. COMP-2
 - Purpose: Store double-precision floating-point numbers.
 - Storage size: Always 8 bytes (IEEE double precision or equivalent).
 - Use cases: Scientific or engineering calculations needing more precision than COMP-1.
 - Notes: Precision is about 15–16 decimal digits.
 - Example: **01 WS-DOUBLE USAGE COMP-2.**

COMP In COBOL

Quick comparison table

USAGE	Type	Bytes	Precision	Typical use
COMP	Binary integer	varies (2–8)	Exact (integer)	Counters, indexes
COMP-1	Single-precision FP	4	~6–7 decimal digits	Medium-precision math
COMP-2	Double-precision FP	8	~15–16 decimal digits	High-precision math

Note: If you need **exact decimal precision** (e.g., money), use **COMP-3** (packed decimal), not **COMP-1/COMP-2** — because floating point can introduce rounding errors.

COBOL Variable Type Checking

- COBOL does not perform strict type checking in the way modern strongly-typed languages like Java or C++ do.
- Instead, COBOL uses a concept called category matching and level number compatibility to determine whether variables are compatible during operations like MOVE, arithmetic operations, or procedure calls.
- Category Matching COBOL groups data items into categories such as: Alphabetic (PIC A), Alphanumeric (PIC X), Numeric (PIC 9), Decimal or Floating-point.
 - When using MOVE, COMPUTE, or arithmetic verbs, COBOL checks if the source and target categories are compatible, not whether their types match exactly.
- Level Numbers: COBOL uses level numbers (01, 05, 77, etc.) to define structure.
 - Operations are valid if level structure allows (e.g., moving a group item to another of compatible structure).

What COBOL Doesn't Strictly Enforce

- You can often move an alphanumeric variable to a numeric one, and the compiler will convert or truncate the data at runtime.
- You can pass variables with different PIC clauses to a subprogram and the compiler won't complain unless CALL is explicitly ... USING BY VALUE or BY CONTENT.
- Example:

```
01 WS-NUMBER  PIC 9(4).
01 WS-TEXT    PIC X(4).
MOVE WS-TEXT TO WS-NUMBER.
```
- This will compile, but may cause runtime issues if WS-TEXT doesn't contain digits.

Arithmetic Operations

Arithmetic Operators			
Binary	Meaning	Unary	Meaning
+	Addition	+	Multiplication by +1
-	Subtract	-	Multiplication by -1
*	Multiplication		
/	Division		
**	Exponentiation		

Safer: COMPUTE RESULT = FUNCTION POW(A,B)

```
15 COMPUTE FullPrice ROUNDED = Price + (Price *  
TaxRate)
```

ADD 1 TO Ind.

Arithmetic Program

```
22 01 Num1 PIC 9 VALUE 5.  
23 01 Num2 PIC 9 VALUE 4.
```

```
50 ADD Num1 TO Num2 GIVING Ans  
51 DISPLAY Ans  
52 SUBTRACT Num1 FROM Num2 GIVING Ans  
53 DISPLAY Ans  
54 MULTIPLY Num1 BY Num2 GIVING Ans  
55 DISPLAY Ans  
56 DIVIDE Num1 INTO Num2 GIVING Ans  
57 DISPLAY Ans  
58 DIVIDE Num1 INTO Num2 GIVING Ans REMAINDER  
    Rem  
59 DISPLAY "Remainder " Rem
```

+09.00
-01.00
+20.00
+00.80
Remainder 0.00

Handle errors:

```
ADD WS-A, WS-B, WS-ITEM GIVING WS-TOT  
ON SIZE ERROR  
DISPLAY "NO MEMORY".
```

Another Arithmetic Program

50	ADD	Num1, Num2	TO	Num3	GIVING	Ans		
51	ADD	Num1, Num2,	Num3	GIVING	Ans			
52	DISPLAY	Ans					+12.00	
53	COMPUTE	Ans	=	Num1	+	Num2		
54	COMPUTE	Ans	=	Num1	-	Num2		
55	COMPUTE	Ans	=	Num1	*	Num2		
56	COMPUTE	Ans	=	Num1	/	Num2		
57	DISPLAY	Ans					+01.25	
58	COMPUTE	Ans	=	Num1	**	2		
59	DISPLAY	Ans					+25.00	
60	COMPUTE	Ans	=	(3	+	5)	*	5
61	DISPLAY	Ans					+40.00	
62	COMPUTE	Ans	=	3	+	5	*	5
63	DISPLAY	Ans					+28.00	
64	COMPUTE	Ans	ROUNDED	=	3.0	+	2.005	
65	DISPLAY	Ans					+05.01	

Error Trapping

- The **ON SIZE ERROR** phrase is used with arithmetic statements (like ADD, SUBTRACT, MULTIPLY, DIVIDE, COMPUTE) to trap arithmetic overflow or truncation errors.
- It's executed if the result of the arithmetic operation:
 - Does not fit in the receiving field because the numeric value is too large (overflow).
 - Has too many decimal places to fit in the receiving field's scale, causing truncation.
 - Would produce an invalid numeric (such as dividing by zero in some COBOL implementations).
- Important notes
 - If ON SIZE ERROR is not specified, COBOL will normally just truncate or wrap the value without warning—dangerous in financial or safety-critical code.
 - Using ON SIZE ERROR is good practice when you need to ensure calculations are valid and within range.

Error Trapping

- Example — Overflow

```
01 NUM-A    PIC 9(3) VALUE 999.
```

```
01 NUM-B    PIC 9(3) VALUE 2.
```

```
ADD NUM-B TO NUM-A
```

```
ON SIZE ERROR
```

```
    DISPLAY "Overflow happened!"
```

```
END-ADD.
```

- $999 + 2 = 1001$ won't fit in PIC 9(3) (max 999), so the ON SIZE ERROR block will execute.

- Example — Truncation

```
01 RESULT  PIC 9(3)V99.
```

```
COMPUTE RESULT = 999.99 + 0.02
```

```
ON SIZE ERROR
```

```
    DISPLAY "Value too big for RESULT."
```

```
END-COMPUTE.
```

- This triggers because the sum (1000.01) can't fit in PIC 9(3)V99.

Variables

Constants

- Def: a constant is a data item that has only one value.
- Example:

```
19 01 PIValue CONSTANT AS 3.14.
```

Figurative Constants

- Def: Figurative constants are reserved words that refer to specific constant values that have been defined by the compiler.
- List of figurative constants:

- ZERO
- ZEROS
- SPACE
- SPACES
- HIGH-VALUE
- LOW-VALUE
- QUOTE
- NULL
- ALL

Functionally identical.
The difference is purely stylistic or contextual—
not semantic.

```
WORKING-STORAGE SECTION.  
  . . .  
  01 WS-DPT-NME          PIC X(30)          VALUE SPACES.  
  . . .  
  01 WS-MORT-INT-RTE    PIC 9(01)v9(04)    VALUE ZEROES.  
  . . .  
  01 WS-EMPNO           PIC 9(06)          VALUE ZERO.
```

COBOL Data Types

- Data types in COBOL:
 - String
 - Character
 - Integer
 - Floating
 - Boolean
- As in Java, C, etc. variables have to have their data type declared prior to being used.

DAVID MURPHY 30 EDU DEPT 2,000 € 10.01

COBOL Data Types

- **Alphabetic** – made up of uppercase/lowercase letters [DAVE MURRY]
Maximum size is 35,535 characters.
- **Numeric** – made up of digits [30].
Maximum number of digits is 18.
- **Alphanumeric** – made up of letters and digits [\$2000].
Maximum size is 35,535 characters.

Edited Data Types

- **Edited numeric** – digits and special characters [\$1,200]
 - An edited numeric variable is declared in the DATA DIVISION using a PICTURE clause that contains both numeric (9) symbols and editing symbols (like Z, , , +, -, CR, DB, *, etc.).
 - Meaning:
 - \$ → Dollar sign literal
 - Z → Suppresses leading zeros (blank instead)
 - , → Comma separator
 - 9 → Required numeric digit (shows zeros)
 - .99 → Decimal point with two digits
- **Edited Alphanumeric** – letters, digits, and special characters [11/12/2025]
 - An Edited Alphanumeric item is declared using an alphanumeric PICTURE clause with editing symbols such as B, 0, /, or any literal characters embedded in the picture string.
 - Base type: Always X (alphanumeric), not 9 (numeric).
 - Editing characters:
 - B = blank insertion
 - / = slash insertion
 - Any literal character in quotes in the PIC will appear in the output.
 - You can mix data and literals to produce formatted display output.

```
01 WS-SALARY      PIC $ZZZ,ZZ9.99.
```

```
01 edited-alphanumeric  PIC X(10)BXX/XX.
```

Variables

- Variables are "Data-Name" or "Data Item"
- Variable names can contain letters, digits, and "-"
- Maximum variable name size: 30 characters
- Four pieces of information required to define a variable:
 - Level Number
 - Variable name
 - PICTURE Clause
 - Data type

Variable Level Number

- In COBOL, level numbers specify the hierarchy of data within a record and identify special-purpose data entries.
- A level number begins a data description entry and has value taken from the set of integers between 1 and 49, or from one of the special-level numbers: 66, 77, or 88.
- The following table shows the significances of the various level numbers that can be used with data items:

LEVEL NUMBERS	
Level Number	Meaning
01	For record description
02-49	For fields with records
01 / 77	For independent items
66	For RENAMES clause
88	For condition names

77 Level Numbers

- 77 is only used for independent data names.
- An independent data name refers to a standalone data item.
 - A variable that is not part of any group structure.
 - It is not subordinate to another field.
 - Does not contain subordinate fields.
 - Has a PIC clause.
 - Holds an actual value.
- Level 77 is valid but rarely used in modern COBOL.
- Most developers just define all data items under 01 level and use sublevels as needed for grouping.
- Note that ANY level is the same as another level assuming that they don't have any subordinate items.

01 EMP-NAME PIC X(35) .

77 EMP-ADD PIC X(40) .

Level 66 – COBOL Renames Statement

- The RENAMES statement is used to define an alternate name or alias for data elements or a group of data items.
- The RENAMES clause is used with the special level number 66
- RENAME clause syntax:

```
66 data-name-1
      RENAMEs data-name-2
      THRU    data-name-3.
```

- Example:

```
01 CUSTOMER-DETAIL.
  05 CUSTOMER-ID    PIC X(5).
  05 FIRST-NAME     PIC X(10).
  05 LAST-NAME      PIC X(15).
  05 PHONE-NUMBER   PIC X(10).

66 NAME-ONLY RENAMES FIRST-NAME THRU LAST-NAME.
```



MOVE SPACES TO NAME-ONLY

This would clear both FIRST-NAME and LAST-NAME at once.

COBOL Renames Rules

- All RENAME entries associated with one logical record must immediately follow that record's last data description entry.
- A level 66 entry cannot rename:
 - A level 01 entry
 - A level 77 entry
 - A level 88 entry'
 - Another level 66 entry
- No data item between data-name-2 and data-name-3 can contain an OCCURS clause.
- The words THRU and THROUGHOUT are equivalent.
- You can write any number of RENAMES entries for a logical record.

COBOL Level 88 Conditions

- A condition name is a name that refers to a condition.
- To define a condition name, you use an 88 level in the Data Division.
- Once defined, the condition name can be used as the condition in an IF, PERFORM UNTIL, or EVALUATE statement.
- Condition names are frequently used with switches and flags.
- A condition name is always coded on the 88 level and only has a VALUE clause associated with it.
- Since a condition name is NOT a name of a field, it will not contain a PICTURE clause.
- The condition name must be unique and its VALUE must be a literal constant with the data type of the field proceeding it.

COBOL Level 88 Conditions Example

Syntax – Condition Names Syntax.

```
88 condition-name  VALUE IS {literal-1 [THRU] literal-2}
```

```
* WORKING-STORAGE SECTION.  
05 MARITAL-STATUS PIC X(01)  
  88 SINGLE VALUE 'S'.  
  88 MARRIED VALUE 'M'.  
  88 DIVORCED VALUE 'D'.  
...  
* PROCEDURE DIVISION.  
*  
A001-MAIN-LOGIC.  
...  
...  
  SET DIVORCED TO TRUE  
  IF DIVORCED THEN  
    DISPLAY 'DIVORCED'  
  END-IF  
  
  IF MARITAL-STATUS = 'S' THEN  
    DISPLAY 'SINGLE'  
  END-IF
```

Variable can take on one of three different condition names.

Variable can have a value of a single character: S/M/D.

88 means that these are condition names – not elements

COBOL Level 88 Conditions Example

```
WORKING-STORAGE SECTION.  
77 WS-MARITAL-STATUS PIC 9.  
88 WS-SINGLE VALUE 0.  
88 WS-MARRIED VALUE 1.  
88 WS-DIVORCE VALUE 2.  
88 WS-WIDOWED VALUE 3.  
77 WS-AMOUNT PIC 9(3) VALUE 100.  
PROCEDURE DIVISION.  
EMPLOYEE-TAX-CAL.  
    ACCEPT WS-MARITAL-STATUS.  
    IF WS-SINGLE SUBTRACT 10 FROM WS-AMOUNT.  
    IF WS-MARRIED SUBTRACT 100 FROM WS-AMOUNT.  
    IF WS-DIVORCED SUBTRACT 20 FROM WS-AMOUNT.  
    DISPLAY WS-AMOUNT.  
    STOP RUN.
```

Group & Elementary Variables

- Group Data Names
 - A group data name refers to a logical grouping of related fields (elementary items) under a single name.
 - Similar to a STRUCT or RECORD in other languages.
 - These variables are declared without a PIC class.
 - The first group level is always 01.
 - Can contain other group data names
 - Can be moved as a unit
- Elementary Data names
 - They should always use data levels that are greater than 01
 - Example:
01 WS-DATE
 - 02 WS-YYYY PIC 9(4)
 - 02 WS-MM PIC 9(2)
 - 02 WS-DD PIC 9(2)

COBOL Group Items

- In COBOL, a Group Items consists of one or more elementary items (data names). A group item is described by:
 - Level number
 - Data name
 - Value clause
- The level numbers must be between 01-49. Typically you will start with 01, and then use multiples of 5: 5, 10, 15, etc.
- Level 01 items must be in the A margin. Other level numbers can begin in the A or B margins.
- You can't code a Picture clause for a group item. Instead, you should code a Picture clause for an elementary item.
- A group item will always be treated as an alphanumeric item, no matter how the elementary items beneath it are defined.

COBOL Group Items Examples

WORKING-STORAGE SECTION.

01 WS-TEMP-VAL.			Group variable
05 WS-REC-LN	PIC 99.		
05 WS-EMP-REC-IN	PIC 9(06).		
05 WS-ERR-MSG	PIC X(12).		
05 WS-TOT-CR-AMT	PIC S9(08)v99.		
05 WS-TOT-DR-AMT	PIC S9(08)v99.		
05 WS-EMP-DPT	PIC A(30).		
01 WS-EMPL-SALARY	PIC Z,ZZZ,ZZZ.99.		Elementary Item

WORKING-STORAGE SECTION.

01 WS-EMP-REC.			Group variable
05 WS-EMP-NO	PIC 9(06).		
05 WS-EMP-NAME.			Group variable
10 WS-INITL	PIC XX	VALUE SPACES.	
10 WS-EMP-FNAME	PIC X(12)	VALUE SPACES.	
10 WS-EMP-LNAME	PIC X(14)	VALUE SPACES.	
05 WS-EMP-DEPT	PIC A(13).		
05 WS-EMP-SAL	PIC S9(10)v9(02).		
77 TOT-REJ-REC-CNT	PIC 9(05).		
01 TOT-REC-CNT	PIC 9(05)	VALUE ZEROES.	

Note: To make the structure of the group variables easy to read, you should align the levels as shown in these examples.

COBOL Initialize Statement

- You can reset the values of all subordinate items in a group by applying the INITIALIZE statement to the group item.
- If an INITIALIZE statement is used to initialize a group item:
 - all numeric and numeric edited fields are set to zero,
 - all alphabetic, alphanumeric, and alphanumeric edited fields are set to spaces.
- **Elementary Item:** If an INITIALIZE statement without a Replacing clause is used to initialize an elementary item, it is set to spaces or zeros depending on how it has been defined.
- The INITIALIZE clause always ignores fields that are defined as Filler, fields that are redefined and index data items.
- However, it is inefficient to initialize an entire group unless you really need all of the items in the group initialized.

COBOL Initialize Statement Example

○ Syntax – Initialize Statement.

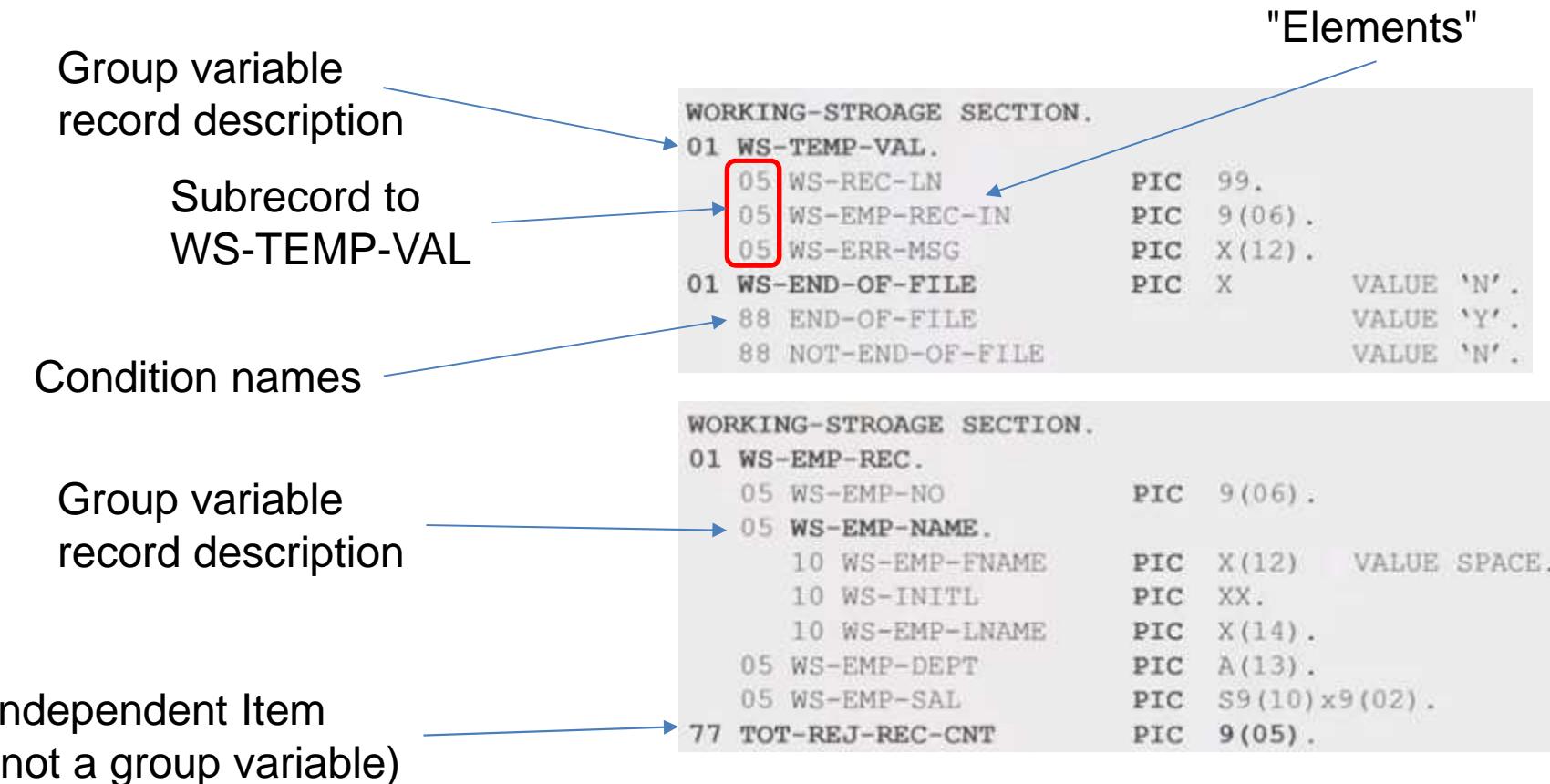
```
INITIALIZE identifier-1
  [REPLACING
    [ALPHABETIC
     ALPHANUMERIC
     NUMERIC
     ALPHANUMERIC-EDITED
     NUMERIC-EDITED]
    ] DATA BY identifier-2
          literal-1
```

```
PROCEDURE DIVISION.
*
* Initialize statements to reset the value of the record.
*
A000-MAIN-LOGIC.
*
  INITIALIZE WS-CUST-REC.
  INITIALIZE WS-CUST-REC
    REPLACING ALPHANUMERIC DATA BY SPACE
    NUMERIC DATA BY ZERO.
```

Simplest form:
initialize everything

Specify what to replace items
with: alpha spaces, numbers zero

Variable Level Number



Note:

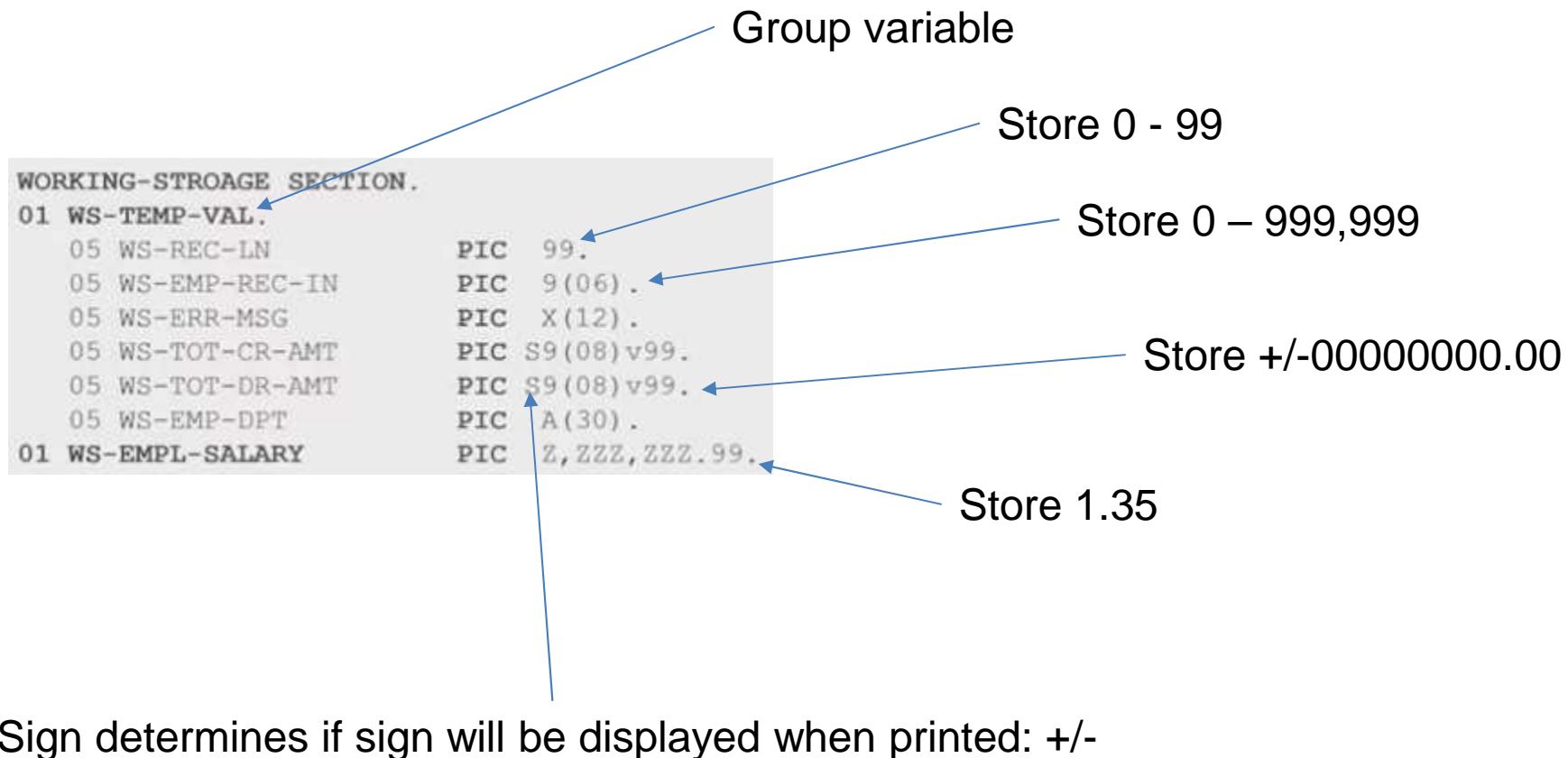
- Level 01 & 77 must begin in area A
- Levels 02 – 49 can begin in Areas A or B
- Level 66 & 88 can begin in Areas A or B

Picture Clause

- The Picture clause specifies the data type and the amount of storage that is required for a data item. It is denoted by PICTURE, often abbreviated as PIC.
- A PICTURE clause is specified only for elementary data items and consists of picture characters. Each picture character denotes storage to be reserved for a character of that type.
- The following are the general picture characters and their meaning:

PICTURE Clause		
Item Type	Characters	Meaning
Alphabetic	A	Alphabetic
Alphanumeric	X	Any character
Numeric	9	Digits
	S	Sign
	V	Assumed decimal point
Numeric Edited	9	Digit
	Z	Zero subpress digit
	,	Inserted comma
	.	Inserted decimal
	-	Minus sign if negative

Picture Clause Example



Picture Clause Example

```
WORKING-STORAGE SECTION.  
01 WS-EMP-REC.  
    05 WS-EMP-NO          PIC 9(06).  
    05 WS-EMP-FNAME       PIC X(12)  VALUE SPACE.  
    05 WS-INITL           PIC XX.  
    05 WS-EMP-LNAME       PIC X(14).  
    05 WS-EMP-DEPT        PIC A(13).  
    05 WS-EMP-SAL         PIC S9(10)v9(02).  
77 TOT-REJ-REC-CNT      PIC 9(05).  
01 TOT-REC-CNT          PIC 9(05)  VALUE ZEROES.
```

Initialized to space

Initialized to zero

Note:

- Alphanumeric items: unused spaces to the right are set to spaces
- Numeric items: unused spaces to the left are set to "0"

Picture Clause Examples

```
01 W■-EDITING-CHAR.  
* ZERO SUPRESING ZERO REPLACED WITH SPACE-> IN(0123) OUT( 123)  
 02 WS-Z999      PIC 999.  
 02 WS-ZZ999     PIC ZZ999.  
 02 WS-ZZZ999    PIC ZZZ999.  
* USING ** SYMBOLS -> IN (00123) OUT (**123)  
 02 WS-ASTERIC   PIC **999.  
* USING $ SYMBOLS -> IN (123.25 ) OUT ( $123.25 )  
 02 WS-DOLLAR    PIC $999.99.  
* USING MINUS SYMOBOL -> IN(1234) OUT (-1234/1234-)  
 02 WS-MINUS-L   PIC -9999. ←  
 02 WS-MINUS-R   PIC 9999-. ←  
* USING PLUS SYMOBOL -> IN(1234) OUT(+1234/1234+)  
 02 WS-PLUS-L    PIC +9999. ←  
 02 WS-PLUS-R    PIC 9999+. ←
```

Note: unlike using S, the +/- will always be printed

```
USING CREDIT & DEBIT SYMBOL -> IN(-1234) OUT(1234CR/1234DR)  
 02 WS-CR        PIC 999CR.  
 02 WS-DR        PIC 999DB.  
USING DOT IN(123.44) OUT(123.44)  
 02 WS-DOT       PIC 9(3).9(2). ←  
USING , IN(12345) OUT(12,345)  
 02 WS-CAMA      PIC 999,99.  
USING BLANK IN(12345) OUT(12 345)  
 02 WS-BLANK     PIC 99B999.  
USING ZERO IN(123) OUT(12300)  
 02 WS-ZERO      PIC 99900.  
 02 WS-ZERO-F    PIC 00999.  
USING / SLASH IN(07072020) OUT (07/07/2020)  
 02 WS-SLASH     PIC 99/99/9999.
```

Note: not supported by all COBOL compilers. V is a better choice

The Difference Between . And v

- PIC 9.99
 - Has an explicit decimal point in the data item.
 - The decimal point is stored as part of the field in memory.
 - Size: 4 characters (1 digit, 1 decimal point, 2 fractional digits).
 - Type: Treated as alphanumeric-numeric (since the decimal is a literal character, not just implied).
 - Usage: Typically when you want to display or accept data with a visible decimal point.
- Example:
01 PRICE PIC 9.99.
- If PRICE = 2.50, the memory actually contains "2.50" (with the dot stored).

The Difference Between . And v

- PIC 9V99
 - The V (assumed decimal point) is not physically stored in memory.
 - The decimal point is implied; COBOL assumes its position.
 - Size: 3 digits (1 before the V, 2 after).
 - Type: Purely numeric (better for arithmetic).
 - Usage: Best when you need to do calculations, since the decimal point is implicit.
- Example:
01 AMOUNT PIC 9V99.
- If AMOUNT = 250, this means 2.50 logically, but the memory holds only 250 (the V is not stored).

👉 Rule of thumb:

- Use **v** (implied decimal) for computations.
- Use **.** (stored decimal) for display or edited fields.

Global Variables

- Definition:
 - A variable that is declared in the main program and can be accessed anywhere within that program, including all paragraphs and sections.
- Scope:
 - Limited to the program in which it is declared.
- Visibility:
 - Not visible to other programs or subprograms unless explicitly passed.
- Declaration Example:

WORKING-STORAGE SECTION.

01 WS-COUNTER PIC 9(3) VALUE 0. *> This is a global variable
- Usage:
 - Can be read or updated by any part of the main program.

External Variable

- Definition:
 - A variable that is declared with the EXTERNAL keyword in a program and can be shared across multiple programs (main program and subprograms).
- Scope:
 - Program-wide but across multiple programs. Any program that declares the same variable as EXTERNAL accesses the same memory location.
- Visibility:
 - Visible to all programs that declare it as EXTERNAL.
- Declaration Example:

WORKING-STORAGE SECTION.
01 SHARED-COUNT PIC 9(3) VALUE 0 EXTERNAL. *> This is an external variable
- Usage:
 - Multiple programs can read or update SHARED-COUNT, and changes are reflected across programs.

Global / External Differences

Key Differences at a Glance

Feature	Global Variable	External Variable
Scope	Within the program only	Across multiple programs
Visibility	Only within the program	Shared across programs
Memory	Local to program	Single memory location shared across programs
Declaration	Standard WORKING-STORAGE entry	<code>EXTERNAL</code> keyword required
Use Case	Temporary or program-local data	Data shared between main and subprograms

Loops

While / For Loop

```
7 FILE SECTION.  
8 WORKING-STORAGE SECTION.  
9 01 Ind PIC 9(1) VALUE 0.  
10 PROCEDURE DIVISION.  
11 PERFORM OutputData WITH TEST AFTER UNTIL Ind > 5  
12     GO TO ForLoop.  
13  
14 OutputData.  
15     DISPLAY Ind.  
16     ADD 1 TO Ind.  
17  
18 ForLoop.  
19     PERFORM OutputData2 VARYING Ind FROM 1 BY  
20         1 UNTIL Ind=5  
21         STOP RUN.  
22     I  
23 OutputData2.  
24     DISPLAY Ind.
```

Paragraph

0
1
2
3
4
5
1
2
3
4

PERFORM PARAGRAPH

- PERFORM is used to transfer control to a paragraph or section, execute the paragraph or section, and then have control returned to the PERFORM clause so that execution can continue from that point.
- Example of PERFORM calling a paragraph:

```
PROCEDURE DIVISION.  
    PERFORM PARA-1.  
    PERFORM PARA-2.  
    STOP RUN.  
PARA-1.  
    COMPUTE C = A + B.  
    DISPLAY C.  
    MOVE C TO D.  
    DISPLAY D.  
PARA-2.  
    COMPUTE C = A - B.  
    DISPLAY C.
```

PERFORM SECTION

- If you PERFORM a section, COBOL executes all the paragraphs in that section from top to bottom, then returns control to the statement after the PERFORM.
- Section names end with the word SECTION. in your code.
- Sections can contain one or more paragraphs.
- PERFORM paragraph-name → runs that paragraph only (unless it contains GO TO or PERFORMs itself).
- PERFORM section-name → runs every paragraph in that section until it hits the next section or the end of the Procedure Division.

```
PROCEDURE DIVISION.  
MAIN-PARA.  
  PERFORM INITIALIZATION-SECTION  
  PERFORM PROCESSING-SECTION  
  STOP RUN.  
  
INITIALIZATION-SECTION.  
  INIT-PARA.  
    DISPLAY "Initializing...".  
  
PROCESSING-SECTION.  
  PROC-PARA-1.  
    DISPLAY "Step 1".  
  PROC-PARA-2.  
    DISPLAY "Step 2".
```

Note: When PERFORM PROCESSING-SECTION is executed, COBOL runs PROC-PARA-1 and then PROC-PARA-2 in order.

PERFORM Types

- There are two types of PERFORM commands:
 - Inline
 - Inline PERFORM is a form of the PERFORM statement that executes a block of code written directly within the PERFORM statement itself, rather than calling a separately defined paragraph or section.
 - The logic is written directly between PERFORM and END-PERFORM.
 - It's typically used for short, self-contained logic that doesn't need to be reused.
 - It improves readability by keeping the code localized instead of jumping to another paragraph.
 - When using this format, no period is permitted between the PERFORM and END-PERFORM

```
PERFORM
  DISPLAY "Enter your name:"
  ACCEPT USER-NAME
  DISPLAY "Hello, " USER-NAME
END-PERFORM.
```

```
PERFORM VARYING I FROM 1 BY 1 UNTIL I > 5
  DISPLAY "Iteration number: " I
END-PERFORM.
```

Perform Types

- PERFORM Paragraph / Out-Of-Line
 - Executes a named paragraph or section defined elsewhere.
 - This is the most common format in most COBOL programming.
- Example:

```
PERFORM CALCULATE-TOTAL.
```

PERFORM Formats

- SIMPLE
- TIMES
- UNTIL
- VARYING

```
|| INLINE PERFORM - SIMPLE
  PERFORM
  DISPLAY 'SIMPLE IN-LINE PERFORM'
  END-PERFORM.
* INLINE PERFORM - TIMES
  PERFORM 3 TIMES
  DISPLAY 'INLINE TIMES PERFORM'
  END-PERFORM.
* INLINE PERFORM - UNTIL
  PERFORM UNTIL I = 5
  COMPUTE I = I + 1
  DISPLAY 'INLINE USING UNTIL PERFORM'
  END-PERFORM.
* INLINE PERFORM - UNTIL VARYING
  PERFORM VARYING I FROM 1 BY 1 UNTIL I = 5
  DISPLAY 'INLINE USING VARYING UNTIL PERFROM'
  END-PERFORM.      ↵
```

PERFORM ... WITH TEST AFTER and PERFORM ... WITH TEST BEFORE

- PERFORM ... WITH TEST BEFORE and PERFORM ... WITH TEST AFTER control when the loop condition is tested in a PERFORM UNTIL or PERFORM VARYING statement.
- TEST BEFORE
 - The condition is evaluated before each iteration.
 - If the condition is already true at the start, the loop body will not execute at all.
 - This is like a “while” loop in other languages.
- Example:

```
PERFORM WITH TEST BEFORE UNTIL counter > 5
  DISPLAY "Counter: " counter
  ADD 1 TO counter
END-PERFORM
```
- If counter starts at 6, the body never runs.
- TEST BEFORE is the default statement and the condition is checked before executing the instructions under Perform.

PERFORM ... WITH TEST AFTER and PERFORM ... WITH TEST BEFORE

- TEST AFTER
 - The condition is evaluated after the loop body runs.
 - The body always runs at least once.
 - This is like a “do-while” loop in other languages.

- Example:

```
PERFORM WITH TEST AFTER UNTIL counter > 5
  DISPLAY "Counter: " counter
  ADD 1 TO counter
END-PERFORM
```

- Even if counter starts at 6, the loop body runs once before the condition is checked.

Feature	TEST BEFORE	TEST AFTER
Condition checked before loop body	✓	✗
Condition checked after loop body	✗	✓
Can skip loop entirely	✓	✗ (runs at least once)
Similar to	while	do...while

CONTINUE

- Meaning: Do nothing and move to the next statement in sequence.
 - Think of it as a “no-op” placeholder — it’s often used in conditional blocks where you want nothing to happen if the condition is true.
 - Structured: Yes — works cleanly with END-IF, END-PERFORM, etc.

- Example:

```
IF A > 100
    CONTINUE
ELSE
    DISPLAY "Value is too small"
END-IF
DISPLAY "Done".
```

- If $A > 100$, the program just skips to the next statement (DISPLAY "Done").

NEXT SENTENCE

- Meaning: Transfer control to the first statement after the next period (.) in the source code.
 - This is period-sensitive — it ignores END-IF, END-PERFORM, etc., and jumps out to the next sentence boundary.
 - Structured: No — it's considered obsolete and risky because if you move or remove a period, control flow can break in unexpected ways.
- Example:

```
IF A > 100
  NEXT SENTENCE
  ELSE
    DISPLAY "Value is too small"
  END-IF
  DISPLAY "Done".
```
- If $A > 100$, the program skips everything until the statement after the next period — it may skip "Done" completely depending on where the period is.

Key Differences

Feature	CONTINUE	NEXT SENTENCE
Action	Do nothing, go to next statement	Jump to after next period
Sensitive to <code>.</code>	No	Yes
Structured?	Yes	No (obsolete)
Safety	Safe	Can cause unintended jumps
Common usage	Placeholders in IF/ELSE blocks	Rarely used in modern code; mostly legacy

Best practice: Use CONTINUE in modern COBOL. Avoid NEXT SENTENCE unless you're maintaining old code that depends on period-based control flow.

EXIT

- EXIT is a very simple statement — it does nothing at runtime.
- It's mainly used as a placeholder to mark the end of a paragraph, section, or inline code block where no executable statements are needed.
- Key points about EXIT:
 - Purpose: Provide a clear end point in code, often for readability or as a target for PERFORM ... THRU.
 - Runtime behavior: No operation — control simply continues to the next statement.
- Allowed forms:
 - EXIT. — plain no-op.
 - EXIT PROGRAM — ends the current program and returns control to the calling program (or operating system if it's the main program).
 - EXIT PARAGRAPH — exits the current paragraph immediately (optional in structured code).
 - EXIT PERFORM — exits a named or inline PERFORM loop early (not supported by all compilers).
 - EXIT SECTION — exits the current section immediately (optional in structured code).

Exit Examples

Plain EXIT as placeholder

```
END-PROCESS.  
  EXIT.
```

EXIT program

```
IF ERROR-FLAG = 'Y'  
  DISPLAY "Exiting..."  
  EXIT PROGRAM  
END-IF
```

EXIT in PERFORM THRU

```
PERFORM STEP-1 THRU STEP-3  
  
STEP-1.  
  DISPLAY "Step 1".  
STEP-2.  
  DISPLAY "Step 2".  
STEP-3.  
  EXIT.
```

Best practice:

- Use EXIT for readability and maintenance clarity — not for logic.
- Use EXIT PROGRAM intentionally to end the program early.
- Avoid using EXIT as a “jump” — it doesn’t change control flow unless it’s one of the special forms like EXIT PROGRAM.

How To Use Storage Space Effectively



Using Storage Space Effectively

- Two ways to make use of COBOL storage space effectively
 - Reuse existing storage space for another purpose
 - REDEFINES
 - Compress the storage space

COBOL Redefines Statement

- The COBOL REDEFINES statement is used to define the same field storage into two more different ways.
- The definition and redefinition of the field refer to the same bytes of storage.
- REDEFINE Level Number: it cannot have a VALUE clause and it cannot be used at level 01 in the FILE Section or REPORT Section.
- Syntax: 01-49 data-name-1 REDEFINES data-name-2
- Example:

```
01 EMP-REC.  
  05 EMP-REC-LY          PIC X(30).  
  05 EMP-NAME-DTL  REDEFINES EMP-REC-LY.  
    10 EMP-FNAME          PIC X(15).  
    10 EMP-LNAME          PIC X(15).  
  05 EMP-DPT-NME  REDEFINES EMP-REC-LY.  
    10 EMP-DPT            PIC X(10).  
    10 EMP-ADD             PIC X(15).  
  
```

Note: total size of variable is 30 bytes

Reuse 30 bytes

Reuse 25 bytes

COBOL Redefines Statement

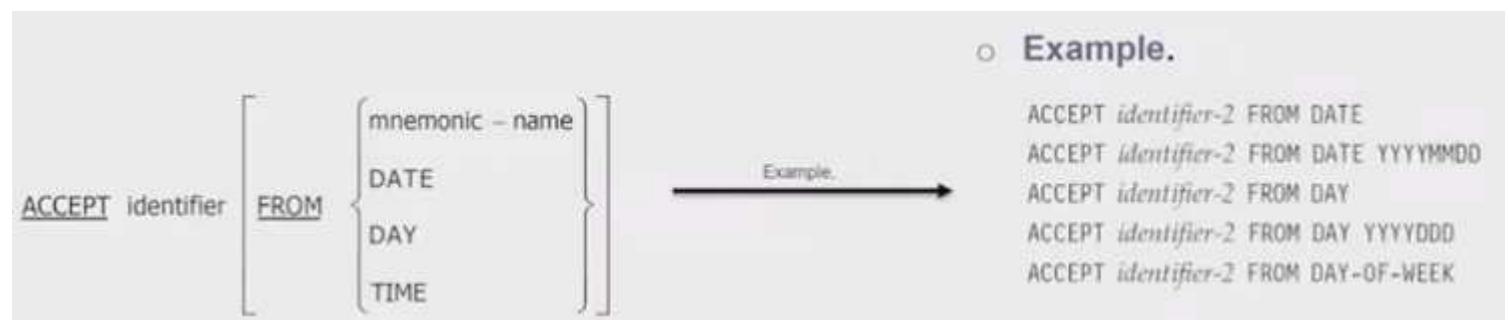
- The REDEFINES clause must follow immediately after data-name-2
- Also, data-name-1 cannot have a VALUE clause.
- The level number of data-name-1 and data-name-2 must be identical and must not be 66 or 88.
- Data-name-1 size should NEVER be larger than data-name-2. However, it can be smaller.
- The data-name-2 item cannot contain an OCCURS clause.

Input / Output



COBOL ACCEPT Statement

- The ACCEPT statement is used to transfer data or system related information into the program during program execution.
- The data is transferred with the help of an identifier.
- No editing or error checking of the incoming data.
- When the ACCEPT statement is executed, the computer will stop and wait for the user to enter data via the keyboard.
- Entered data is stored in a variable and the cursor moves to the next line on the screen.
- Syntax:



COBOL ACCEPT Statement

- Example:

```
PROCEDURE DIVISION.  
*  
* Get Today's Date.  
*  
ACCEPT WS-TDY-DATE  FROM DATE
```

- Note: If the FROM option is omitted, the data will be stored in the identifier using left justification.

Getting A Date

- Using **ACCEPT ... FROM DATE**
 - Returns the date in YYMMDD format (2-digit year).
- Example:

```
WORKING-STORAGE SECTION.  
01 WS-DATE  PIC 9(6).  
PROCEDURE DIVISION.  
    ACCEPT WS-DATE FROM DATE.  
    DISPLAY "Current date (YYMMDD): " WS-DATE.
```

- Limitation: Year is 2 digits, so no century.

Getting A Date w/ Century

- Using **ACCEPT ... FROM CURRENT-DATE**

- Returns a full 21-character string including century, time, and more.
 - Format: YYYYMMDDHHMMSS...
 - You can extract the date with century from it.
 - Example:

```
WORKING-STORAGE SECTION.  
01 WS-CURR-DATE    PIC X(21).  
01 WS-FULL-DATE    PIC 9(8).  
PROCEDURE DIVISION.  
    ACCEPT WS-CURR-DATE FROM CURRENT-DATE.  
  
    * Extract YYYYMMDD (first 8 characters)  
    MOVE WS-CURR-DATE(1:8) TO WS-FULL-DATE.  
  
    DISPLAY "Current date with century: " WS-FULL-DATE.
```

- Output: 20250817 (for August 17, 2025)
- You can also extract time if needed: **WS-CURR-DATE(9:6)** gives HHMMSS.

Convert Into Separate Numeric Fields

IDENTIFICATION DIVISION.

PROGRAM-ID. CURRENT-DATE-DEMO.

DATA DIVISION.

WORKING-STORAGE SECTION.

*> Raw 21-character current date string

01 WS-CURR-DATE PIC X(21).

*> Separate numeric fields

01 WS-YEAR PIC 9(4).

01 WS-MONTH PIC 9(2).

01 WS-DAY PIC 9(2).

PROCEDURE DIVISION.

MAIN-PARA.

*> Get the current date and time

ACCEPT WS-CURR-DATE FROM CURRENT-DATE.

*> Extract numeric fields

MOVE WS-CURR-DATE(1:4) TO WS-YEAR

MOVE WS-CURR-DATE(5:2) TO WS-MONTH

MOVE WS-CURR-DATE(7:2) TO WS-DAY

DISPLAY "Year : " WS-YEAR

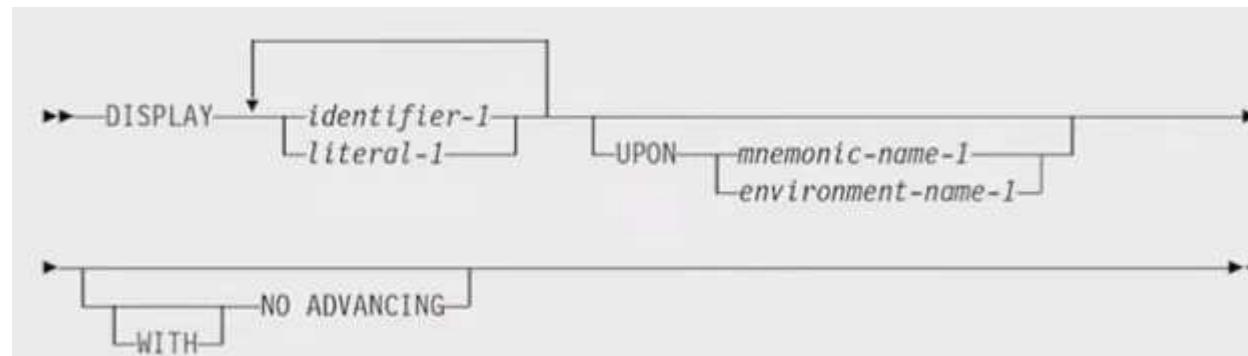
DISPLAY "Month: " WS-MONTH

DISPLAY "Day : " WS-DAY

STOP RUN.

COBOL DISPLAY Statement

- The DISPLAY statement will display a literal or a variable on the monitor or output device.
- Syntax:



- Example:

```
PROCEDURE DIVISION.  
*  
* Display Message in Spool/output.  
*  
    DISPLAY "HELLO WORLD!"
```

- In COBOL, numeric data is right justified and alphabetic / alphanumeric data is left justified.

"Hello World" Program

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. HELLO01.  
AUTHOR. TOPICTRICK.  
DATE-WRITTEN. MAR 31, 2022.  
*  
* DESC: THE HELLO WORLD PROGRAM.  
*  
PROCEDURE DIVISION.  
A001-MAIN.  
    DISPLAY "HELLO WORLD!"  
    STOP RUN.  
  
OUTPUT:  
    HELLO WORLD!
```

2 Divisions

Paragraph

This is how every COBOL
program ends

Name & SS Number Program

```
11 01 UserName PIC X(30) VALUE "You".
12 01 Num1    PIC 9 VALUE ZEROS.
13 01 Num2    PIC 9 VALUE ZEROS.
14 01 Total   PIC 99 VALUE 0.
15 01 SSNum.
16      02 SSArea  PIC 999.
17      02 SSGroup PIC 99.
18      02 SSSerial PIC 9999.
26 MOVE ZERO TO UserName
27 DISPLAY UserName
28 DISPLAY "Enter 2 values to sum "
29 ACCEPT Num1
30 ACCEPT Num2
31 COMPUTE Total = Num1 + Num2
32 DISPLAY Num1 " + " Num2 " = " Total
33 DISPLAY "Enter your social security number "
34 ACCEPT SSNum
35 DISPLAY "Area " SSArea
```

Note leading "0" on 2 digit number.

```
What is your name Derek
Hello Derek
00000000000000000000000000000000
Enter 2 values to sum
4
5
4 + 5 = 09
Enter your social security number
123456789
Area 123
>> █
```

Note this command causes "0"s to be placed in all 30 locations

Note: load full SS number and then only print out part of it.

Program Data

```
10 01 SampleData PIC X(10) VALUE "Stuff".
11 01 JustLetters PIC AAA VALUE "ABC".
12 01 JustNums PIC 9(4) VALUE 1234.
13 01 SignedInt PIC S9(4) VALUE -1234.
14 01 PayCheck PIC 9(4)V99 VALUE ZEROS.
15 01 Customer.
16      02 Ident    PIC 9(3).
17      02 CustName PIC X(20).
18      02 DateOfBirth.
19          03 MOB PIC 99.
20          03 DOB PIC 99.
21          03 YOB PIC 9(4).
22 01 Num1 PIC 9 VALUE 5.
23 01 Num2 PIC 9 VALUE 4.
24 01 Num3 PIC 9 VALUE 3.
25 01 Ans  PIC S99V99 VALUE 0.
26 01 Rem  PIC 9V99.    I
```

Program Execution

```
28 PROCEDURE DIVISION.  
29 MOVE "More Stuff" TO SampleData  
30 MOVE "123" TO SampleData  
31 MOVE 123 TO SampleData  
32 DISPLAY SampleData  
33 DISPLAY PayCheck  
34 MOVE "123Bob Smith" 12211974" TO  
      Customer  
35 DISPLAY CustName  
36 DISPLAY MOB "/" DOB "/" YOB
```

20 characters

Note that information for all fields in a customer record are being entered fully.

123
0000.00
Bob Smith
12/21/1974

Program Execution

```
37 MOVE ZERO TO SampleData
38 DISPLAY SampleData
39 MOVE SPACE TO SampleData
40 DISPLAY SampleData
41 MOVE HIGH-VALUE TO SampleData
42 DISPLAY SampleData
43 MOVE LOW-VALUE TO SampleData
44 DISPLAY SampleData
45 MOVE QUOTE TO SampleData
46 DISPLAY SampleData
47 MOVE ALL "2" TO SampleData
48 DISPLAY SampleData
```

0000000000	??????????	"??????????"	2222222222
------------	------------	--------------	------------

Get User Name Program

Cursor will stay on last line printed – will NOT advance

```
21 PROCEDURE DIVISION.  
22 DISPLAY "What is your name " WITH NO ADVANCING  
23 ACCEPT UserName  
24 DISPLAY "Hello " UserName
```

MOVE Command

- COBOL command to move data from one variable to another.
- If you are moving data from an Elementary Variable, it is called an Elementary Move or simple move.
- If you are moving data from a Group Variable, the it is called a Group Move.
- MOVE Rules:
 - Sending fields can be either constant or variable.
 - Receiving fields must be variable (group / elementary).
- Note: it is possible to move a single variable into multiple variables:

```
MOVE 74      TO WS-MSG-LENGTH, WS-TEST.
```

- When performing a Group MOVE, both sender and receiver should be of type Group.

MOVE CORRESPONDING

- If you are trying to move one group into another group and their elementary components are not identical, you can use **CORESSPONDING** or **CORR** to specify which components you want to move.
- It copies values from fields in group1 to fields with the exact same names in group2. It ignores fields that don't match
- Given:

```
01 WS-SALARY.  
  02 WS-GROSS  PIC 9(5) VALUE 50000.  
  02 WS-TAX    PIC 9(3) VALUE 100.  
01 WS-A-SALARY.  
  02 WS-GROSS  PIC 9(5).  
  02 WS-TAX    PIC 9(3).  
  02 WS-BONUS  PIC 9(3) VALUE 200.  
01 WS-B-SALARY.  
  02 WS-GROSS  PIC 9(5).  
  02 WS-TAX    PIC 9(3).  
  02 WS-BONUS  PIC 9(3).  
  
```

- The MOVE would look like this:

Note that these
are comments!

Actual command

```
PROCEDURE DIVISION.  
EMPLOYEE-SAL-CAL.  
*      MOVE WS-GROSS OF WS-SALARY TO WS-GROSS OF WS-A-SALARY  
*      MOVE WS-GROSS OF WS-SALARY TO WS-GROSS OF WS-B-SALARY  
MOVE CORRESPONDING WS-SALARY TO WS-A-SALARY.  
  
```

Reference Modification

- In COBOL, a reference modification lets you access substrings of an alphanumeric field by specifying a starting position and length — similar to substring operations in other languages.
- Syntax: identifier(start:length)
 - start: The 1-based position (i.e., first character is position 1)
 - length: Number of characters to operate on
 - You can use it in a MOVE, IF, STRING, INSPECT, etc.
- Example:

```
DATA DIVISION.  
WORKING-STORAGE SECTION.  
01 FULL-NAME  PIC X(20) VALUE SPACES.  
01 FIRST-NAME  PIC X(10) VALUE "JIM".  
01 LAST-NAME  PIC X(10) VALUE "ANDERSON".
```

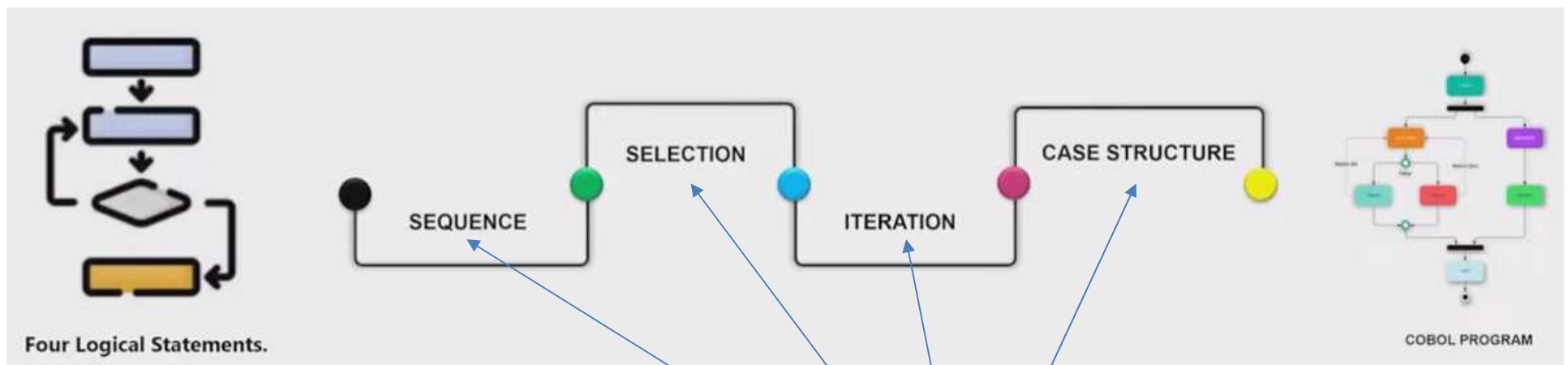
```
PROCEDURE DIVISION.  
MOVE FIRST-NAME TO FULL-NAME(1:10).  
MOVE LAST-NAME TO FULL-NAME(11:10).  
  
DISPLAY "Full Name: [" FULL-NAME "]".
```

Output:

```
Full Name: [JIM ANDERSON ]
```

Conditional Statements

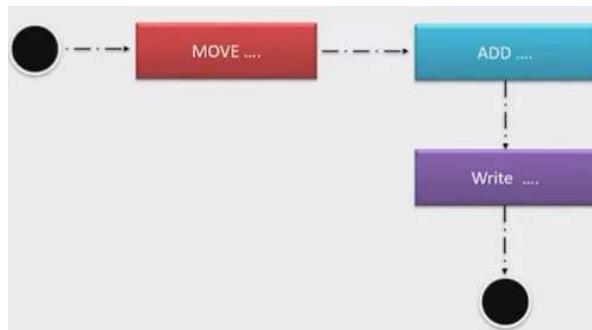
COBOL Logical Control Structures



4 logical structures in COBOL

Sequence

- Instructions are to be processed step-by-step in a specific sequence.
- For example, the following example would be executed from top to bottom:

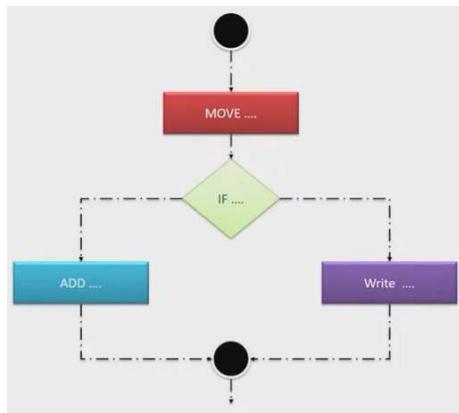


- Example:

```
PROCEDURE DIVISION.  
*  
*  
A001-CALC-MNTH-SAL.  
    MOVE EMP-MNTH-SAL      TO WS-EMP-SAL  
    MOVE EMP-MNTH-ARR      TO WS-EMP-ARR  
    MOVE EMP-BONUS         TO WS-EMP-BONUS  
    COMPUTE TOT-MNTH-SAL = WS-EMP-SAL + WS-EMP-ARR +  
                           WS-EMP-BONUS.  
    DISPLAY 'EMP MNTH SAL :' TOT-MNTH-SAL.
```

Selection

- Selection is a logical control construct that executes instructions depending on the existence of a condition.
- It is sometimes called an IF-THEN-ELSE control structure.



```
PROCEDURE DIVISION.

A001-CALC-MNTH-SAL.
  MOVE EMP-MNTH-SAL      TO WS-EMP-SAL.
  MOVE EMP-MNTH-ARR      TO WS-EMP-ARR.
  MOVE ZEROES             TO WS-EMP-BONUS.
  IF EMP-GRADE = '03' THEN
    MOVE EMP-BONUS        TO WS-EMP-BONUS
  END-IF.
  COMPUTE TOT-MNTH-SAL = WS-EMP-SAL + WS-EMP-ARR +
                        WS-EMP-BONUS.
  DISPLAY 'EMP MNTH SAL :' TOT-MNTH-SAL.
```

- Example:

Conditional Statements In COBOL

- Conditional statements are instructions that let your program make decisions and branch depending on whether a condition is true or false.
- They're used to control flow of execution — deciding which code runs under which circumstances.
- There are four main types of conditional statements in COBOL
 - IF – ELSE – END IF
 - EVALUATE – END EVALUATE
 - SEARCH / SEARCH ALL
 - PERFORM UNTIL / PERFORM VARYING
- *Scope terminators* are used to mark the end of a verb. Examples are: EVALUATE END-EVALUATE, IF END-IF
- A condition is an expression that evaluates to true or false.
- COBOL doesn't require parentheses for complex conditions, but using them makes the logic clearer — especially with AND and OR.

IF – ELSE – END IF

- Checks if a condition is true, then executes specific statements.

```
IF CUSTOMER-BALANCE > 0
    DISPLAY "Customer owes money"
ELSE
    DISPLAY "Customer is paid up"
END-IF.
```

EVALUATE

- Lets you test multiple conditions or values in a structured way.

EVALUATE TRUE

WHEN AGE < 18

DISPLAY "Minor"

WHEN AGE >= 18 AND AGE < 65

DISPLAY "Adult"

WHEN OTHER

DISPLAY "Senior"

END-EVALUATE.

SEARCH / SEARCH ALL

- Used with tables (arrays) to find an element meeting a condition.

```
SEARCH STUDENT-TABLE
  AT END DISPLAY "Not found"
  WHEN STUDENT-NAME = "ALICE"
    DISPLAY "Found Alice"
  END-SEARCH.
```

PERFORM UNTIL / PERFORM VARYING

- A conditional can control loop execution.

```
PERFORM UNTIL EOF-FLAG = 'Y'  
      READ CUSTOMER-FILE  
      AT END MOVE 'Y' TO EOF-FLAG  
      END-READ  
END-PERFORM.
```

```
WORKING-STORAGE SECTION.  
01 COUNTER    PIC 9(2) VALUE 0.  
  
PROCEDURE DIVISION.  
      PERFORM DISPLAY-NUMBER  
      VARYING COUNTER FROM 1 BY 1  
      UNTIL COUNTER > 5  
      STOP RUN.  
  
      DISPLAY-NUMBER.  
      DISPLAY "COUNTER = " COUNTER.
```

The PERFORM VARYING statement is a looping construct — it lets you repeat a block of code while automatically changing (varying) the value of a control variable each time through the loop.

Conditional Statements

- < or LESS THAN
- <= or LESS THAN OR EQUAL
- > or GREATER THAN
- >= or GREATER THAN OR EQUAL
- = or EQUAL TO
- <> or NOT EQUAL TO

COBOL Conditionals

```
7 CONFIGURATION SECTION.  
8 SPECIAL-NAMES.  
9      CLASS PassingScore IS "A" THRU "C",  
      "D".
```

"Classification"

```
12 WORKING-STORAGE SECTION.  
13 01 Age PIC 99 VALUE 0.  
14 01 Grade PIC 99 VALUE 0.  
15 01 Score PIC X(1) VALUE "B".  
16 01 CanVoteFlag PIC 9 VALUE 0.  
17      88 CanVote VALUE 1.  
18      88 CantVote VALUE 0.  
19 01 TestNumber PIC X.  
20      88 IsPrime VALUE "1", "3", "5", "7".  
21      88 IsOdd VALUE "1", "3", "5", "7",  
      "9".  
22      88 IsEven VALUE "2", "4", "6", "8".  
23      88 LessThan5 VALUE "1" THRU "4".  
24      88 ANumber VALUE "0" THRU "9".
```

Conditional Program

```
59 IF Age > 18 THEN
60     SET CanVote TO TRUE
61 ELSE
62     SET CantVote TO TRUE
63 END-IF
35 IF Age LESS THAN 5 THEN
36     DISPLAY "Stay Home"
37 END-IF
38 IF Age = 5 THEN
39     DISPLAY "Go to Kindergarten"
40 END-IF
41 IF Age > 5 AND Age < 18 THEN
42     COMPUTE Grade = Age - 5
43     DISPLAY "Go to Grade " Grade
44 END-IF
45 IF Age GREATER THAN OR EQUAL TO 18
46     DISPLAY "Go to College"
47 END-IF
```

```
Enter Age : 19
You can vote
Go to College
```

Using Classification

```
49 IF Score IS PassingScore THEN
50     DISPLAY " You Passed"
51 ELSE
52     DISPLAY "You Failed"
53 END-IF
54 *> NUMERIC ALPHABETIC ALPHABETIC-UPPER
55 IF Score IS NOT NUMERIC THEN
56     DISPLAY "Not a Number"
57 END-IF
58
59 IF Age > 18 THEN
60     SET CanVote TO TRUE
61 ELSE
62     SET CantVote TO TRUE
63 END-IF
64 DISPLAY "Vote " CanVoteFlag
```

```
7 CONFIGURATION SECTION.
8 SPECIAL-NAMES.
9     CLASS PassingScore IS "A" THRU "C",
"D".
```

Built in classifications

```
16 01 CanVoteFlag PIC 9 VALUE 0.
17             88 CanVote VALUE 1.
18             88 CantVote VALUE 0.
```

```
Enter Age : 19
You can vote
Go to College
You Passed
Not a Number
Vote 1
```

Notes On The IF Statement

- The IF statement is a conditional statement that is used to check a condition and then either bypass or execute a set of instructions.
- The IF statement is terminated by either a "." or an END-IF
- No periods are allowed between the IF statement and an END-IF.
- It is permitted to have nested IF statements.
- Conditions that can be used by an IF statement:
 - IS EQUAL TO or =
 - IS LESS THAN or <
 - IS GREATER THAN or >

```
MOVE 3500 TO WS-SALARY.  
IF WS-SALARY > 3000  
MOVE 40 TO WS-TAX  
SUBTRACT WS-TAX FROM WS-SALARY GIVING WS-SAL-TAX  
ELSE  
MOVE 00 TO WS-TAX  
END-IF.  
DISPLAY 'SALARY + TAX = ' WS-SAL-TAX.  
DISPLAY 'SALARY = ' WS-SALARY.  
DISPLAY 'TAX = ' WS-TAX.
```

IF Compound Conditions

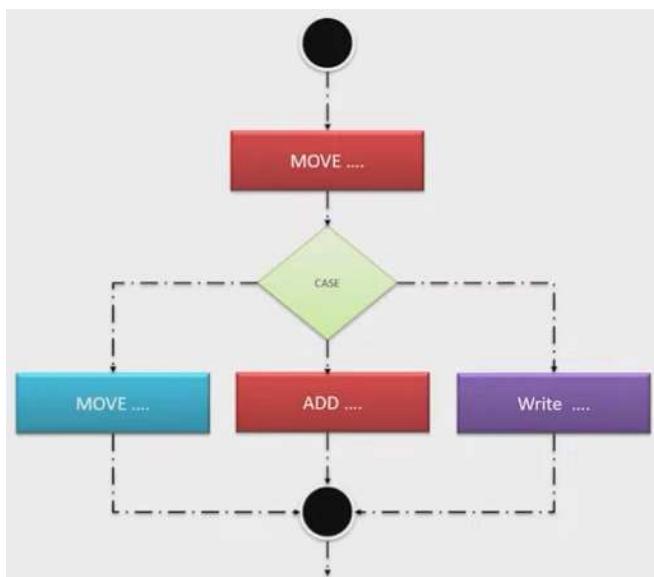
- With an IF statement you can use:
 - AND
 - OR

Note use of
ELSE stmt

```
MOVE 3500 TO WS-SALARY.  
IF WS-SALARY > 3000  
  IF WS-SALARY > 3000 OR WS-SALARY < 3200  
    IF WS-SALARY > 3000 AND WS-SALARY < 3200  
      MOVE 40 TO WS-TAX  
      SUBTRACT WS-TAX FROM WS-SALARY GIVING WS-SAL-TAX  
    ELSE  
      MOVE 00 TO WS-TAX  
    END-IF.  
    DISPLAY 'SALARY + TAX = ' WS-SAL-TAX.  
    DISPLAY 'SALARY'      = ' WS-SALARY.  
    DISPLAY 'TAX'        = ' WS-TAX.  
STOP RUN.
```

Case

- The case structure is a special logical control structure used when there are numerous paths to be followed depending on the contents of a given field.



```
PROCEDURE DIVISION.
...
A001-CALC-MNTH-SAL.
    MOVE EMP-MNTH-SAL      TO WS-EMP-SAL.
    MOVE EMP-MNTH-ARR      TO WS-EMP-ARR.

    EVALUATE TRUE
        WHEN EMP-GRADE = '02'
        WHEN EMP-GRADE = '03'
        WHEN EMP-GRADE = '04'
        WHEN OTHER
    END-EVALUATE
    DISPLAY 'EMP MNTH SAL : ' TOT-MNTH-SAL.
```

- Example:

What is a Scope Terminator?

- A scope terminator is a reserved phrase (like END-IF, END-PERFORM, END-READ, etc.) that explicitly marks the end of a statement's scope (the range of code it controls).
- When is a Scope Terminator Mandatory?
 - Scope terminators are mandatory in COBOL in these cases:
 - When one conditional or loop is nested inside another.
- Example: Nested IF statements. Without END-IF, the compiler wouldn't know which IF pairs with which ELSE.

```
IF A = B
  IF X = Y
    DISPLAY "Match"
  ELSE
    DISPLAY "Mismatch"
  END-IF
END-IF
```

What is a Scope Terminator?

- Example: When an ELSE is used. If you use ELSE, you must close the IF with END-IF.

```
IF A = B
  DISPLAY "Equal"
ELSE
  DISPLAY "Not Equal"
END-IF
```

- Example: When writing in a structured (modern) COBOL style.
 - Certain compilers (especially with EVALUATE, PERFORM ... UNTIL, READ, SEARCH, etc.) require explicit scope terminators.
 - Example with EVALUATE:

```
EVALUATE TRUE
  WHEN A = 1
    DISPLAY "One"
  WHEN OTHER
    DISPLAY "Other"
END-EVALUATE
```

What is a Scope Terminator?

- When multiple statements follow a conditional without a period to end the sentence.
 - Older COBOL used a period (.) to end the scope, but in structured COBOL the period isn't needed (and not recommended), so you must use the proper END-xxx.

Common Scope Terminators:

1. END-IF
2. END-PERFORM
3. END-EVALUATE
4. END-READ
5. END-SEARCH
6. END-STRING
7. END-UNSTRING
8. END-RETURN

EVALUATE

- EVALUATE is another conditional statement that can be used in place of an IF statement if the condition being analyzed is complex.
- It is coded with a WHEN condition and an END-EVALUATE scope terminator.
- Between the EVALUATE and the END-EVALUATE clauses, multiple WHEN conditions are applied.
- The goal is to avoid having to write a collection of IF/ELSE statements.

```
PROCEDURE DIVISION.  
MONTH-PARA.  
  EVALUATE WS-MONTH  
    WHEN 1 MOVE 'JAN' TO WS-MONTH-NAME.  
    WHEN 2 MOVE 'FEB' TO WS-MONTH-NAME.  
    WHEN 3 MOVE 'MAR' TO WS-MONTH-NAME.  
    WHEN 4 MOVE 'APR' TO WS-MONTH-NAME.  
    WHEN 5 MOVE 'MAY' TO WS-MONTH-NAME.  
    WHEN 6 MOVE 'JUN' TO WS-MONTH-NAME.  
    WHEN 7 MOVE 'JUL' TO WS-MONTH-NAME.  
    WHEN 8 MOVE 'AUG' TO WS-MONTH-NAME.  
    WHEN 9 MOVE 'SEP' TO WS-MONTH-NAME.  
    WHEN 10 MOVE 'OCT' TO WS-MONTH-NAME.  
    WHEN 11 MOVE 'NOV' TO WS-MONTH-NAME.  
    WHEN 12 MOVE 'DEC' TO WS-MONTH-NAME.  
  END-EVALUATE.  
  DISPLAY 'MONTH IS : ' WS-MONTH-NAME.
```

```
EVALUATE TRUE  
  WHEN WS-MONTH = 4 OR 6 OR 9 OR 11  
    MOVE 30 TO WS-DAYS  
  WHEN WS-MONTH = 2  
    MOVE 28 TO WS-DAYS  
  WHEN OTHER  
    MOVE 31 TO WS-DAYS.
```

Note use of "other" for all additional conditions

Using EVALUATE

```
19 01 TestNumber PIC X.  
20      88 IsPrime  VALUE "1", "3", "5", "7".  
21      88 IsOdd   VALUE "1", "3", "5", "7",  
         "9".  
22      88 IsEven  VALUE "2", "4", "6", "8".  
23      88 LessThan5  VALUE "1" THRU "4".  
24      88 ANumber  VALUE "0" THRU "9".
```

```
68 PERFORM UNTIL NOT ANumber  
69      EVALUATE TRUE  
70          WHEN IsPrime DISPLAY "Prime"  
71          WHEN IsOdd  DISPLAY "Odd"  
72          WHEN IsEven DISPLAY "Even"  
73          WHEN LessThan5 DISPLAY "Less than  
9"          5"  
74          WHEN OTHER  DISPLAY "Default Action"  
75      END-EVALUATE  
76      ACCEPT TestNumber  
77 END-PERFORM
```

```
Enter Single Number or X to Exit :  
1  
Prime  
2  
Even  
3  
Prime  
9  
Odd  
X
```

EVALUATE TRUE

- EVALUATE TRUE means "Evaluate each WHEN condition as a Boolean expression. If it's TRUE, execute that block."
- Example: Replacing IF/ELSE IF with EVALUATE TRUE

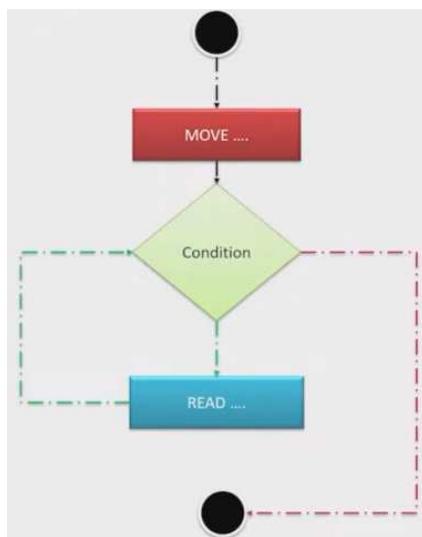
```
IF score >= 90
    DISPLAY "A"
ELSE IF score >= 80
    DISPLAY "B"
ELSE IF score >= 70
    DISPLAY "C"
ELSE
    DISPLAY "F".
```

```
EVALUATE TRUE
WHEN score >= 90
    DISPLAY "A"
WHEN score >= 80
    DISPLAY "B"
WHEN score >= 70
    DISPLAY "C"
WHEN OTHER
    DISPLAY "F"
END-EVALUATE.
```

- Same logic, but EVALUATE TRUE makes it easier to read, especially with many conditions.

Iteration

- Iteration is a logical control structure used for specifying the repeated execution of a series of statements.

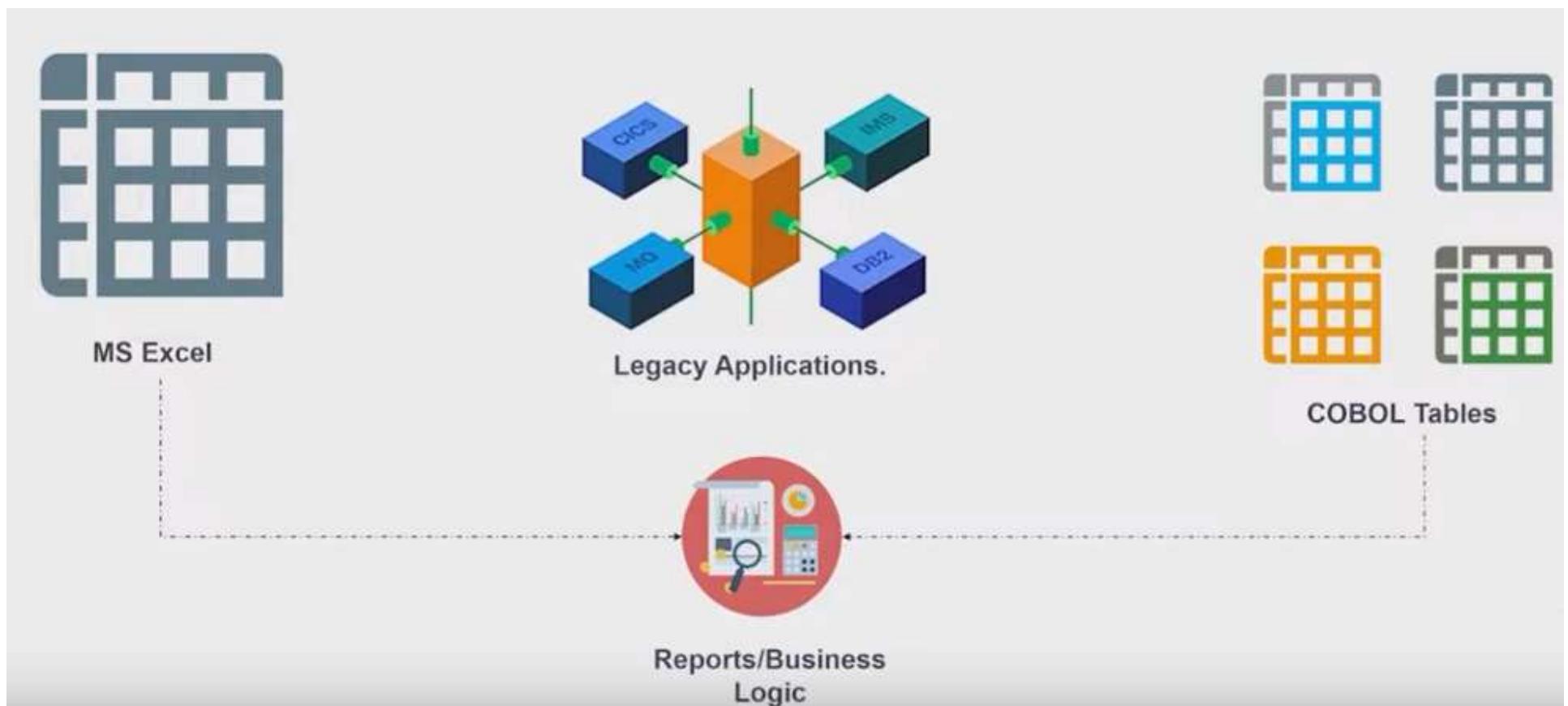


Example:

```
PROCEDURE DIVISION.  
...  
MOVE EMP-MNTH-SAL           TO WS-EMP-MNTH-SAL.  
...  
A001-READ-EMP-FLE.  
    PERFORM UNTIL EOF-OF-FILE  
        READ EMP-REC INTO WS-EMP-REC  
        AT END  
            MOVE 'Y'           TO EOF-OF-FILE  
            NOT AT END  
        ...  
    END-READ  
END-PERFORM.
```

Tables / Arrays

COBOL Table (Array)



COBOL Table (Array)

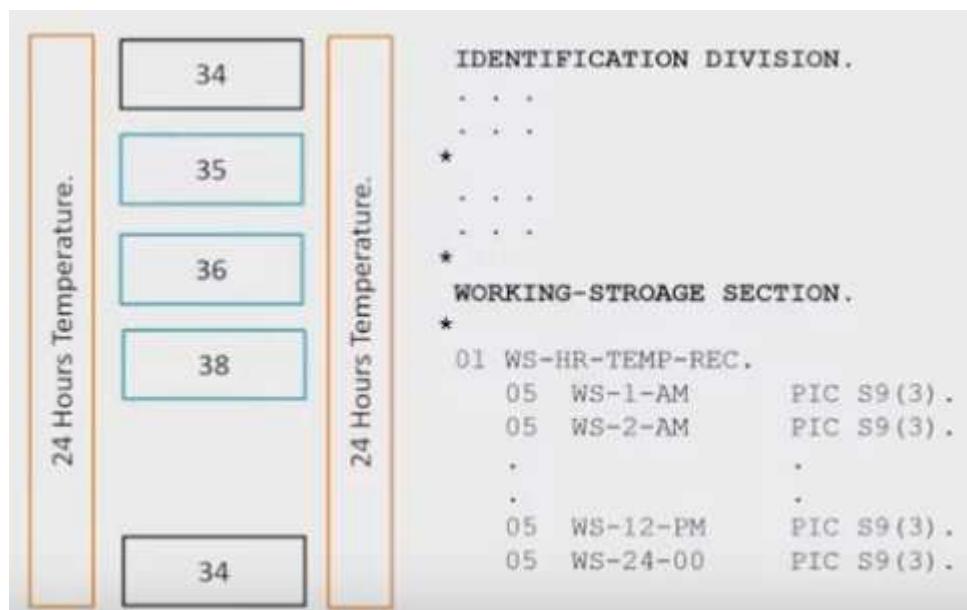
- Note that in COBOL, arrays are indexed using indexes which are called subscripts and unlike other computer languages, indexing starts at 1 (not 0).
- In COBOL, an array/table is a linear data structure that is used to store, access, and process data easily and efficiently.
- An array is a collection of homogenous data items. These data items posses the same size and structure.
- An array can be referred to by a single name.
- The data items contained in a COBOL table are called its **elements** and can be used in arithmetic and logical operations.
- The elements of a table are stored contiguously in memory and it can be either elementary items or group items.
- A table can be of fixed length or variable length.
- In COBOL, a table can have up to 7 dimensions.

OCCURS Clause

- The OCCURS clause is used to define the array/table. It actually indicates the repeated occurrence of fields with the same format.
- An array/table is defined in the working-storage section of the program.
- Code that uses an array is defined in the procedure division.
- Note that the OCCURS clause can only be used with level numbers 02-49.
- It cannot be used with level 01 because it must be used to define fields and not records.

OCCURS Example: Storing 24 Hours Of Temperatures

Without An Array



With An Array

*
WORKING-STORAGE SECTION.
*
01 WS-HR-TEMP-REC.
 05 WS-HR-TEMP OCCURS 24 TIMES PIC S9(3).

OCCURS Examples: 1D, 2D

1D ARRAY:

C	O	D	I	N	G	E	E	K
1	2	3	4	5	6	7	8	9

← single row of elements

Example – Define Array (One Dimensional).

```
*  
WORKING-STORAGE SECTION.  
*  
01 WS-YR-MNTHS VALUE "JANFEBMARAPRJUNJULAUGSEPOCTNOVDEC".  
 05 WS-MNTH      OCCURS 12 TIMES    PIC X(3) .
```

Will create an array where each of the 12 elements is initialized with three letters: JAN, FEB, MAR, APR, etc.

Example – Define Array (Two Dimensional).

```
*  
WORKING-STORAGE SECTION.  
*  
01 WS-TEMP-REC.  
 05 WS-DAYS      OCCURS 07 TIMES.  
    10 WS-HOURS    OCCURS 24 TIMES.  
      15 WS-TEMP      PIC S9(3) .
```

2D ARRAY:

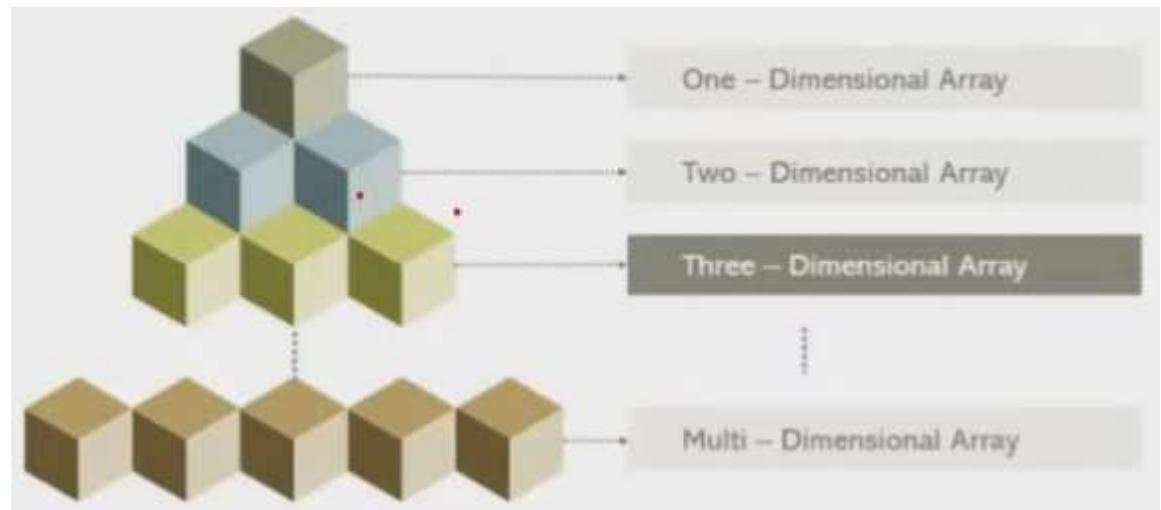
		col 0	col 1	col 2
		i	j	
row 0	1	A	A	A
	2	B	B	B
	3	C	C	C

↑
rows

← columns

← array elements

OCCURS Examples: 3D



Example – Define Array (Three Dimensional).

```
*  
WORKING-STORAGE SECTION.  
*  
01 WS-TEMP-REC.  
    05 WS-MNTH      OCCURS 12 TIMES.  
        10 WS-DAYS    OCCURS 31 TIMES.  
            15 WS-HOURS   OCCURS 24 TIMES.  
                20 WS-TEMP      PIC S9(3).  
                
```

OCCURS Examples: Group Data Items

Example – Define Array (Group data-items).

```
*  
WORKING-STORAGE SECTION.  
*  
01 WS-BOOKS.  
    05 WS-BK-DTLS      OCCURS 100 TIMES.  
        10 WS-AUTHOR.  
            15 WS-FIRST-NAME          PIC A(15).  
            15 WS-MID-NAME          PIC A(10).  
            15 WS-LAST-NAME          PIC A(10).  
        10 WS-TITLE              PIC A(60).  
    
```

Create / Use Table

```
3 PROGRAM-ID. coboltut.  
4  
5 DATA DIVISION.  
6  
7 WORKING-STORAGE SECTION.  
8 01 Table1.  
9     02 Friend PIC X(15) OCCURS 4 TIMES.  
10 PROCEDURE DIVISION.  
11 MOVE 'Joy' TO Friend(1).  
12 MOVE 'Willow' TO Friend(2).  
13 MOVE 'Ivy' TO Friend(3).  
14 DISPLAY Friend(1).  
15 DISPLAY Table1.  
16  
17 STOP RUN.
```

Joy	Joy	Willo
Joy		Ivy
W		

Create / Use Multidimensional Table

```
11 01 CustTable.  
12      02 CustName OCCURS 5 TIMES.  
13          03 FName PIC X(15).  
14          03 LName PIC X(15).  
15 PROCEDURE DIVISION.  
16 MOVE 'Paul' TO FName(1).  
17 MOVE 'Smith' TO LName(1).  
18 MOVE 'Sally' TO FName(2).  
19 MOVE 'Smith' TO LName(2).  
20 DISPLAY CustName(1).  
21 DISPLAY CustTable.  
22  
23 STOP RUN.
```

Paul	Smith
Paul	Smith
	Sally
	Smith

Table w/ Subscripts Inside Of Them

```
16 01 OrderTable.  
17      02 Product OCCURS 2 TIMES INDEXED BY I.  
18          03 ProdName PIC X(10).  
19          03 ProdSize OCCURS 3 TIMES INDEXED BY J.  
20              04 SizeType PIC A.  
21  
22 PROCEDURE DIVISION.  
23 SET I J TO 1.  
24 MOVE 'Blue Shirt' TO Product(I).  
25 MOVE 'S' TO ProdSize(I,J).  
26 SET J UP BY 1.  
27 MOVE 'M' TO ProdSize(I,J).  
28 SET J DOWN BY 1.  
29 MOVE 'Blue ShirtSMLRed Shirt SML' TO OrderTable.  
30 PERFORM GetProduct VARYING I FROM 1 BY 1 UNTIL I>2.  
31 GO TO LookUp.  
32  
33 GetProduct.  
34     DISPLAY PRODUCT(I).  
35     PERFORM GetSizes VARYING J FROM 1 BY 1 UNTIL  
36 J>3.  
37 GetSizes.  
38     DISPLAY ProdSize(I,J).  
39 LookUp.  
40     SET I TO 1.  
41     SEARCH Product  
42         AT END DISPLAY 'Product Not Found'  
43         WHEN ProdName(I) = 'Red Shirt'  
44             DISPLAY 'Red Shirt Found'  
45         END-SEARCH.  
46  
47 STOP RUN.
```

Blue ShirtSML
S
M
L
Red Shirt SML
S
M
L
Red Shirt Found

Pre-Fill A Table

```
7 WORKING-STORAGE SECTION.  
8 01 ProdTable.  
9     02 ProdData.  
10        03 FILLER PIC X(8) VALUE "Red SML".  
11        03 FILLER PIC X(8) VALUE "Blue SML".  
12        03 FILLER PIC X(8) VALUE "GreenSML".  
13    02 FILLER REDEFINES ProdData.  
14        03 Shirt OCCURS 3 TIMES.  
15            04 ProdName PIC X(5).  
16            04 ProdSizes PIC A OCCURS 3 TIMES.  
17 01 ChangeMe.  
18     02 TextNum PIC X(6).  
19     02 FloatNum REDEFINES TextNum PIC 9(4)V99.  
20  
21 PROCEDURE DIVISION.  
22 DISPLAY Shirt(1).  
23 MOVE '123456' TO TextNum.  
24 DISPLAY FloatNum.
```

Red SML
1234.56

Searching An Array

- When you search an array, it will be a linear or sequential search $[O(n)]$.
- AT END is used to identify the end of an array
- SEARCH is coded with the WHEN clause to specify a condition Value.
- **The table must use OCCURS with INDEXED BY.**
- You must set the index to an initial position before the search.
- Syntax:

```
SEARCH table-name
      AT END
          imperative-statements-when-not-found
      WHEN condition-1
          imperative-statements-1
      WHEN condition-2
          imperative-statements-2
END-SEARCH.
```

Search Example

```
WORKING-STORAGE SECTION.  
77 NAME PIC X(10).  
01 J PIC 9(1) VALUE 0.  
01 BANK-ACCT.  
    02 ACC-TABLE OCCURS 3 TIMES INDEXED BY I.  
        03 ACC-NO PIC 9(10).  
        03 ACC-NM PIC X(10).  
        03 ACC-AMT PIC 9(10).  
PROCEDURE DIVISION.  
    ACCEPT NAME.  
    DISPLAY "U ENTERED THE NAME " , NAME.  
    MOVE 1 TO J.  
    PERFORM 3 TIMES  
    ACCEPT ACC-NO(J)  
    ACCEPT ACC-NM(J)  
    ACCEPT ACC-AMT(J)  
    ADD 1 TO J
```

```
SET I TO 1.  
SEARCH ACC-TABLE  
AT END DISPLAY "NO RECORD FOUND"  
WHEN ACC-NM(I) = NAME  
DISPLAY "THE DETAIL ARE"  
DISPLAY "ACC-NO IS " ACC-NO(I)  
DISPLAY "NAME OF ACCOUNT HOLDER " ACC-NM(I)  
DISPLAY "AMOUNT IS " ACC-AMT(I)  
SET I UP BY 1  
END-SEARCH.  
STOP RUN.
```

Search Example

```
01 EMPLOYEE-TABLE.  
 05 EMPLOYEE-ENTRY OCCURS 10 TIMES  
    INDEXED BY EMP-IDX.  
    10 EMP-ID      PIC X(5).  
    10 EMP-NAME    PIC X(20).  
  
WORKING-STORAGE SECTION.  
01 SEARCH-ID      PIC X(5).  
01 FOUND-FLAG    PIC X VALUE 'N'.  
  
PROCEDURE DIVISION.  
  MOVE 'E102' TO SEARCH-ID  
  SET EMP-IDX TO 1  
  
  SEARCH EMPLOYEE-ENTRY  
    AT END  
      DISPLAY "EMPLOYEE NOT FOUND"  
    WHEN EMP-ID(EMP-IDX) = SEARCH-ID  
      DISPLAY "FOUND EMPLOYEE: " EMP-NAME(EMP-IDX)  
      MOVE 'Y' TO FOUND-FLAG  
  END-SEARCH.
```

Searching Using Indexes

- When you have a table (OCCURS clause) with two or more indexes defined, the SEARCH statement will use only the index specified in the SEARCH statement itself.
- How it works: A COBOL table (array) can have multiple indexes defined with the OCCURS clause, for example:

```
01 STUDENT-TABLE.  
 05 STUDENT-ENTRY OCCURS 100 TIMES  
    INDEXED BY IDX1 IDX2.  
    10 STUDENT-ID    PIC 9(5).  
    10 STUDENT-NAME  PIC X(20).
```

- Here, the table **STUDENT-ENTRY** has two indexes: **IDX1** and **IDX2**. Both can be used to traverse or reference the same array independently.

SEARCH with Index

- When you code a SEARCH statement, you must explicitly specify which index to use:

```
SEARCH STUDENT-ENTRY
  AT END DISPLAY "Not found"
  WHEN STUDENT-ID(IDX1) = 12345
    DISPLAY "Found student: " STUDENT-NAME(IDX1)
  END-SEARCH
```

- In this case, **IDX1** is the controlling index. If you instead coded **STUDENT-ID(IDX2)**, then **IDX2** would be used.
- Important Rules:
 - Only the index mentioned in the SEARCH (e.g., IDX1 or IDX2) is advanced and controlled.
 - Multiple indexes give you flexibility — you could be searching with IDX1 while IDX2 is being used elsewhere (e.g., nested searches, or keeping a "bookmark").
 - If you don't specify the index in reference, the compiler will throw an error — because SEARCH needs to know which index it is operating on.

SEARCH ALL – Binary Search

- Faster [$O(\log n)$] but requires the table to be **sorted** in ascending or descending order based on the search key.
- Requires ASCENDING KEY IS clause in the table definition.
- You must use USING to specify the search key.
- Only supports 1 WHEN condition
- Syntax:

```
SEARCH ALL table-name
    AT END
        imperative-statements-when-not-found
    WHEN search-key = table-key
        imperative-statements
    END-SEARCH.
```

SEARCH ALL Example

```
WORKING-STORAGE SECTION.  
01 NUMBER1.  
    02 NO1 PIC 9(2) OCCURS 5 TIMES  
        . . . ASCENDING KEY IS NO1 INDEXED BY I.  
PROCEDURE DIVISION.  
    PERFORM PARA1 THRU PARA1-END.  
PARA1.  
    PERFORM VARYING I FROM 1 BY 1 UNTIL I > 5  
    ACCEPT NO1(I)  
    DISPLAY NO1(I)  
    END-PERFORM.  
    SEARCH ALL NO1  
    AT END DISPLAY "NOT FOUND"  
    WHEN NO1(I) = 99  
        DISPLAY "FOUND".  
PARA1-END.
```

SEARCH vs SEARCH ALL

- SEARCH and SEARCH ALL are two different ways to retrieve data from an array in COBOL.
- SEARCH is faster than PERFORM
- SEARCH ALL is faster than SEARCH
- Both SEARCH and SEARCH ALL will use INDEX – it's a displacement position of an array data name.
- INDEX is declared by using INDEX BY clause along with the OCCURS clause.

SUBSCRIPT & INDEX

- In COBOL, subscripts and indexes are both used to access elements of tables (arrays), but they are very different in how they work, how they're defined, and how efficient they are.

Feature	Subscript	Index
What it is	A numeric data item (e.g., <code>WS-I</code>)	A special internal pointer
How it's defined	Regular <code>PIC 9(n)</code> variable	Defined via <code>INDEXED BY</code> clause
How it's used	<code>ARRAY(SUB)</code>	<code>ARRAY(INDEX)</code>
Speed	Slower (COBOL must compute offset)	Faster (already an offset)
Required for	Manual access, loops	<code>SEARCH</code> , <code>SEARCH ALL</code>
Can you move it?	Yes, with <code>MOVE</code>	No, must use <code>SET</code> to change value
Declaration example	<code>01 WS-I PIC 9(2).</code>	<code>Occurs 10 Times INDEXED BY IDX-NAME.</code>

Subscript – Example

- A **subscript** is just a number that refers to a position in the table.

```
01  MONTH-NAMES.  
    05  MONTH OCCURS 12 TIMES.  
        10  MONTH-NAME PIC X(9).  
  
01  I          PIC 99.  
  
PROCEDURE DIVISION.  
    PERFORM VARYING I FROM 1 BY 1 UNTIL I > 12  
        DISPLAY MONTH-NAME(I)  
    END-PERFORM.
```

- Here, I is a subscript, and COBOL calculates the position each time.

Index – Example

- An **index** is an efficient internal counter COBOL uses to access table elements.

```
01  MONTH-NAMES.  
    05  MONTH OCCURS 12 TIMES  
        INDEXED BY M-INDEX.  
    10  MONTH-NAME PIC X(9).  
  
PROCEDURE DIVISION.  
    SET M-INDEX TO 1  
    PERFORM UNTIL M-INDEX > 12  
        DISPLAY MONTH-NAME(M-INDEX)  
        SET M-INDEX UP BY 1  
    END-PERFORM.
```

- Here, M-INDEX is an index, and SET is used to manipulate it.

Subscript / Index: When to Use What?

- Use subscripts for:
 - Simple loops
 - When index efficiency isn't important
 - Easy debugging
- Use indexes for:
 - SEARCH / SEARCH ALL (required)
 - Better performance (especially with large tables)

Subscript / Index Pitfalls

DISPLAY MONTH-NAME(1)	*> Subscript 
DISPLAY MONTH-NAME(M-INDEX)	*> Index 
DISPLAY MONTH-NAME(WS-I)	*> Subscript 
DISPLAY MONTH-NAME(I-NDX)	*> Index 
MOVE I TO M-INDEX	*>  Invalid!
SET M-INDEX TO I	*>  Invalid!
SET M-INDEX TO 1	*>  Correct!

You must use SET with indexes and MOVE with subscripts.

Array Bounds Caution

- COBOL does not do array bounds checking.
- This means that you can reference an item that is beyond the end of your array and the compiler will not flag it.
- Example:

```
01 MY-ARRAY
  05 ITEM PIC 9(2) OCCURS 5 TIMES
```

```
MOVE 99 to ITEM(6)
```

- The compiler will compile it and at runtime you will overwrite memory past the array.

Convert String To Float

```
17 01 ChangeMe.  
18      02 TextNum PIC X(6).  
19      02 FloatNum REDEFINES TextNum PIC 9(4)V99.  
20 01 StrNum PIC X(7).  
21 01 SplitNum.  
22      02 WNum PIC 9(4) VALUE ZERO.  
23      02 FNum PIC 99 VALUE ZERO.  
24 01 FlNum REDEFINES SplitNum PIC 9999V99.  
25 01 DollarNum PIC $$,$$9.99.
```

```
Red SML  
1234.56  
Enter a Float : 1234  
.78  
$1,234.07
```

```
27 PROCEDURE DIVISION.  
28 DISPLAY Shirt(1).  
29 MOVE '123456' TO TextNum.  
30 DISPLAY FloatNum.  
31 DISPLAY "Enter a Float : " WITH NO ADVANCING  
32 ACCEPT StrNum  
33 UNSTRING StrNum  
34      DELIMITED BY "." OR ALL SPACES  
35      INTO WNum, FNum  
36 MOVE FlNum TO DollarNum  
37 DISPLAY DollarNum.
```

MOVE Statement

- The MOVE statement transfers data from one area of storage to one or more other storage areas.
- The original data is retained in the sending field.
- If necessary, the data that is moved is converted to the usage of the receiving field.
- Source and destination identifiers can be either a group or an elementary data item.
- Three types of moves:
 - Alphabetic
 - Alphanumeric
 - Numeric
- Moving Alphabetic or Numeric data into an Alphanumeric edited or Numeric edited field makes it become edited. Moving a Alphanumeric edited or Numeric edited field into an Alphabetic or Numeric field makes it become deedited.
- If the receiving field is bigger than the sending field then spaces or 0's will be added.

MOVE Statement

Syntax – MOVE Statement.

```
MOVE  Identifier-1 /Literal-1  TO  Identifier-2
      |
      +----- Verb -----+
      |
```

Example – MOVE Data items.

```
PROCEDURE DIVISION.
*
...
MOVE EMP-NO          TO WS-EMP-NO.
MOVE EMP-FNAME        TO WS-EMP-FNAME.
MOVE EMP-LNAME        TO WS-EMP-LNAME.
MOVE 1233              TO WS-EMP-SAL.
MOVE 'COMP'            TO WS-EMP-DEPT.
MOVE ZEROES            TO WS-EMP-INT-RT.
```

Subprograms

CALL Statement

- A **run unit** is the active session or execution instance of a COBOL program, including:
 - The main program
 - Any called subprograms (statically or dynamically)
 - The data and working storage areas used by those programs
- Once you start running a COBOL program, you've created a run unit. It ends when the program exits completely (e.g., with STOP RUN).
- First program in a run unit is referred to as being the main program.
- All other COBOL programs in the run unit are sub program.
- Subprograms can also call other subprograms

Subprograms

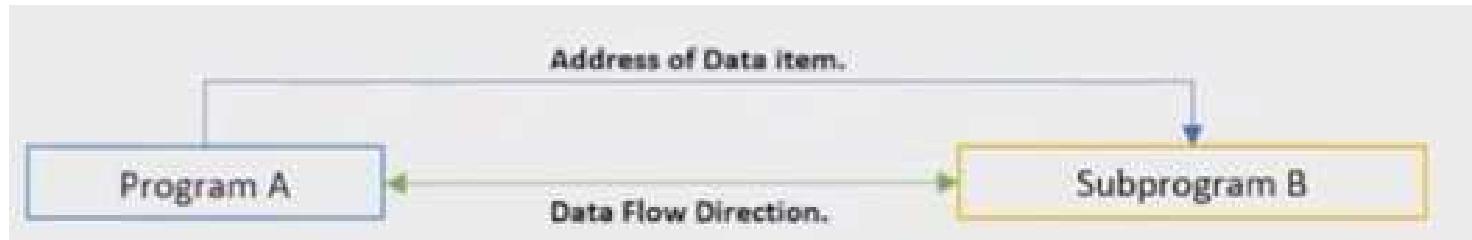
- A subprogram is terminated by a **EXIT PROGRAM** or **GOBACK** clause.
- MAIN program variables are called as actual parameters.
- Subprogram variables are called as formal parameters.
- Subprograms are called using **PROCEDURE <procName> USING <parameters>**
- A subprogram is terminated with an **EXIT PROGRAM** clause.

Subprogram BY REFERENCE Parameters

- BY REFERENCE – Actual Parameters will be changed when formal parameters are changed
 - the subprogram works with the **same memory location** as the main program.
 - Any change you make to that variable inside the subprogram will be **visible** in the main program after the subprogram finishes.
 - **CALL 'SUBPROG' USING MY-VAR.**
 - **CALL 'SUBPROG' USING BY REFERENCE MY-VAR.**
 - Changes **will** be reflected in the main program.
 - If you don't specify anything, most COBOL compilers default to BY REFERENCE — meaning changes in the subprogram will carry over to the main program.

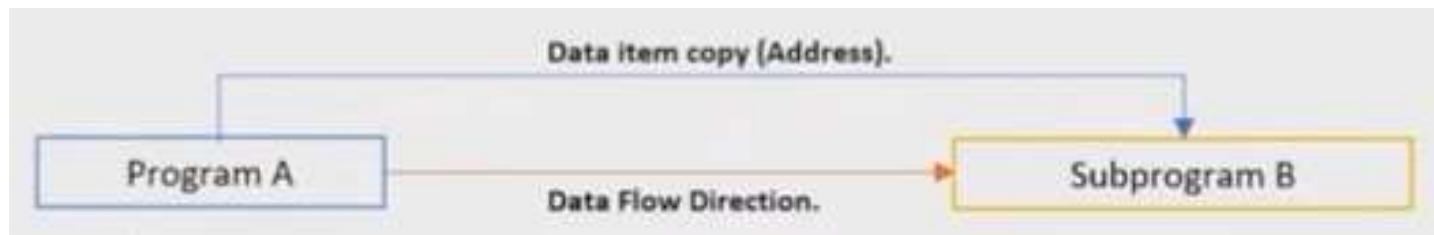
Call By REFERENCE

- The subprogram refers to and processes the data items in storage of the calling program rather than working on a copy of the data.
- Any change made to the data item in the subprogram will also be made to the data item in the main program.



Call By CONTENT

- The calling program passes only the contents of the literal or identifier.
- Any change made to the copy of the data item in the subprogram will only affect the copy of the data item.



Subprogram BY CONTENT Parameters

- BY VALUE - Actual Parameters are not changed when formal parameters are changed
 - If you pass **BY CONTENT**, COBOL sends a **copy** of the variable's value to the subprogram.
 - The subprogram can change its own copy, but the changes **will not** affect the original in the main program.
 - **CALL 'SUBPROG' USING BY CONTENT MY-VAR.**
 - Changes **will not** be reflected in the main program.

Call By VALUE

- The calling program passes the value of the data items instead of a reference to the sending data item.
- The called program can change the parameter, but the changes will not affect the argument in the calling program.

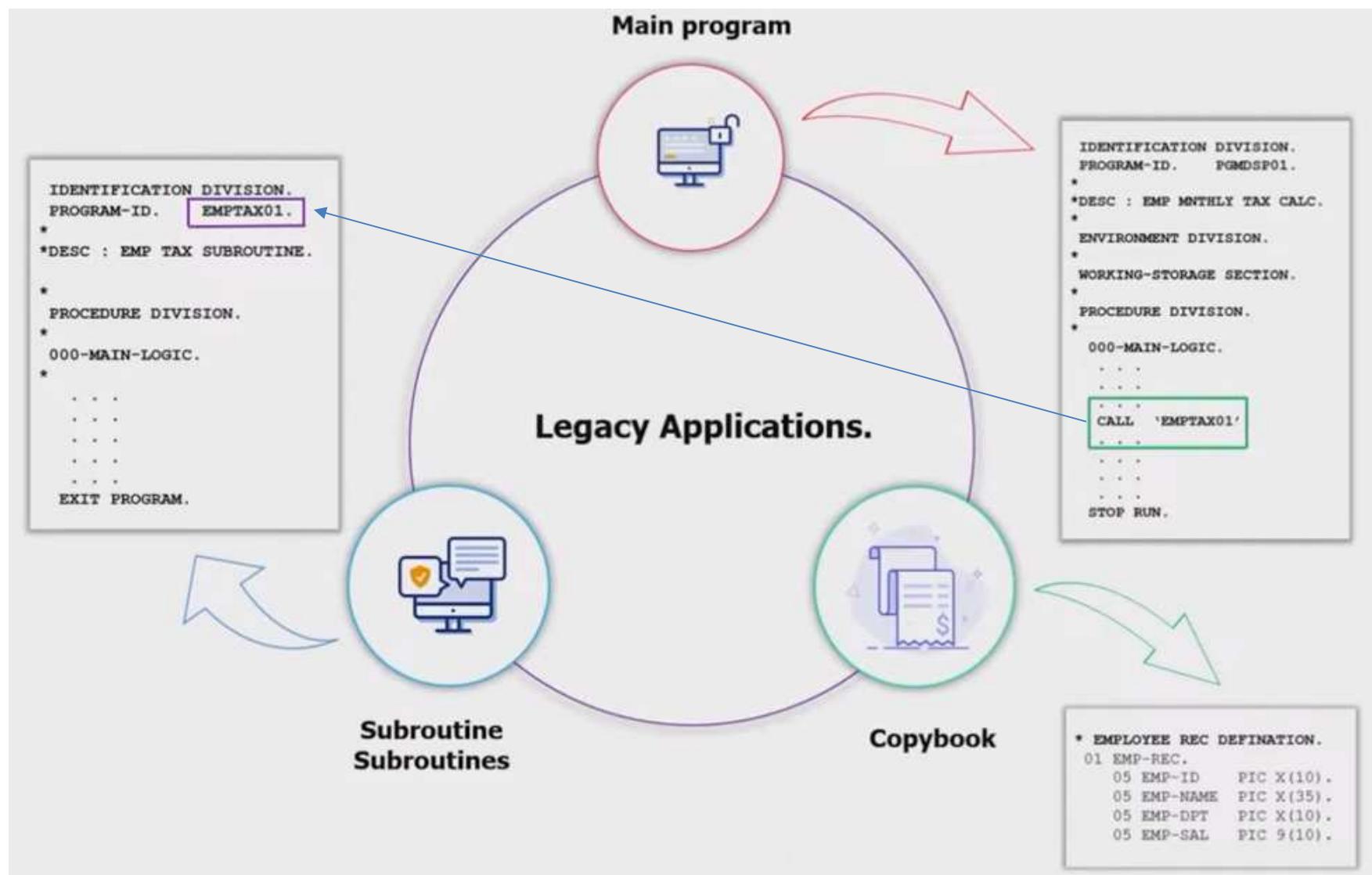
Subprogram BY VALUE Parameters

- Less common
- Similar to BY CONTENT, but specifically passes the value as an immediate literal/constant.
- Changes never affect the main program's variable.

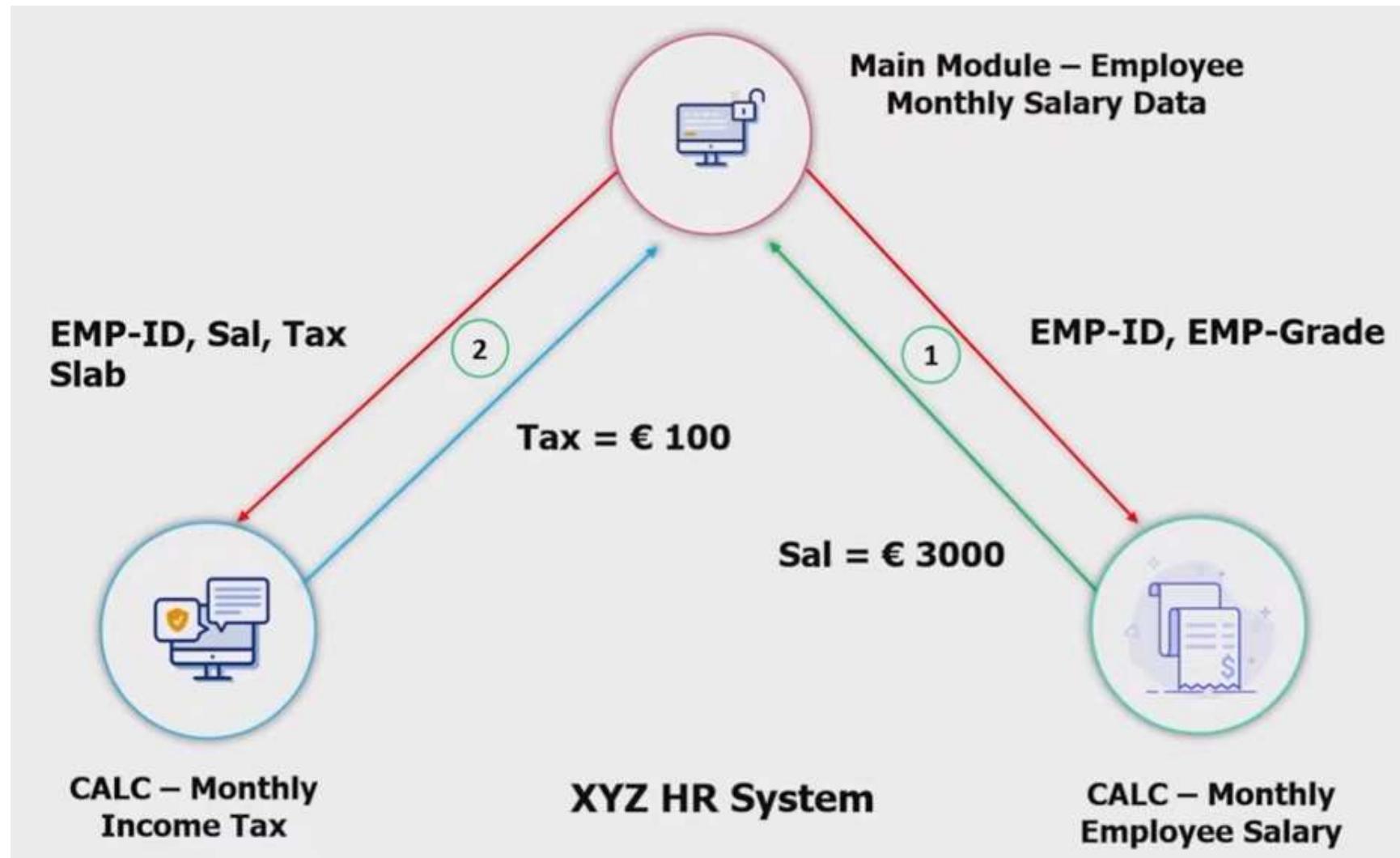
CALL Statement Example

IDENTIFICATION DIVISION.
PROGRAM-ID. PGMDSP01.
*
*DESC : EMP MONTHLY TAX CALC.
*
ENVIRONMENT DIVISION.
*
WORKING-STORAGE SECTION.
*
PROCEDURE DIVISION.
*
000-MAIN-LOGIC.
* * *
* * *
* * *
* * *
CALL 'EMPTAX01'.
* * *
* * *
* * *
* * *
* * *
STOP RUN.

Program Example



Program Example: Employee Payroll Tax

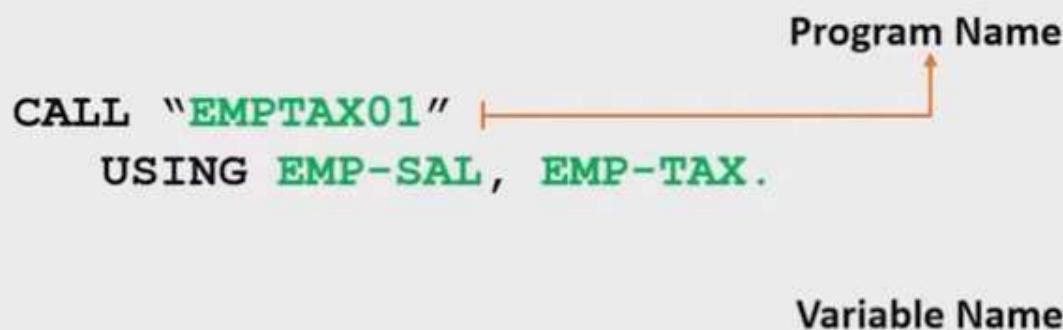


Passing Parameters



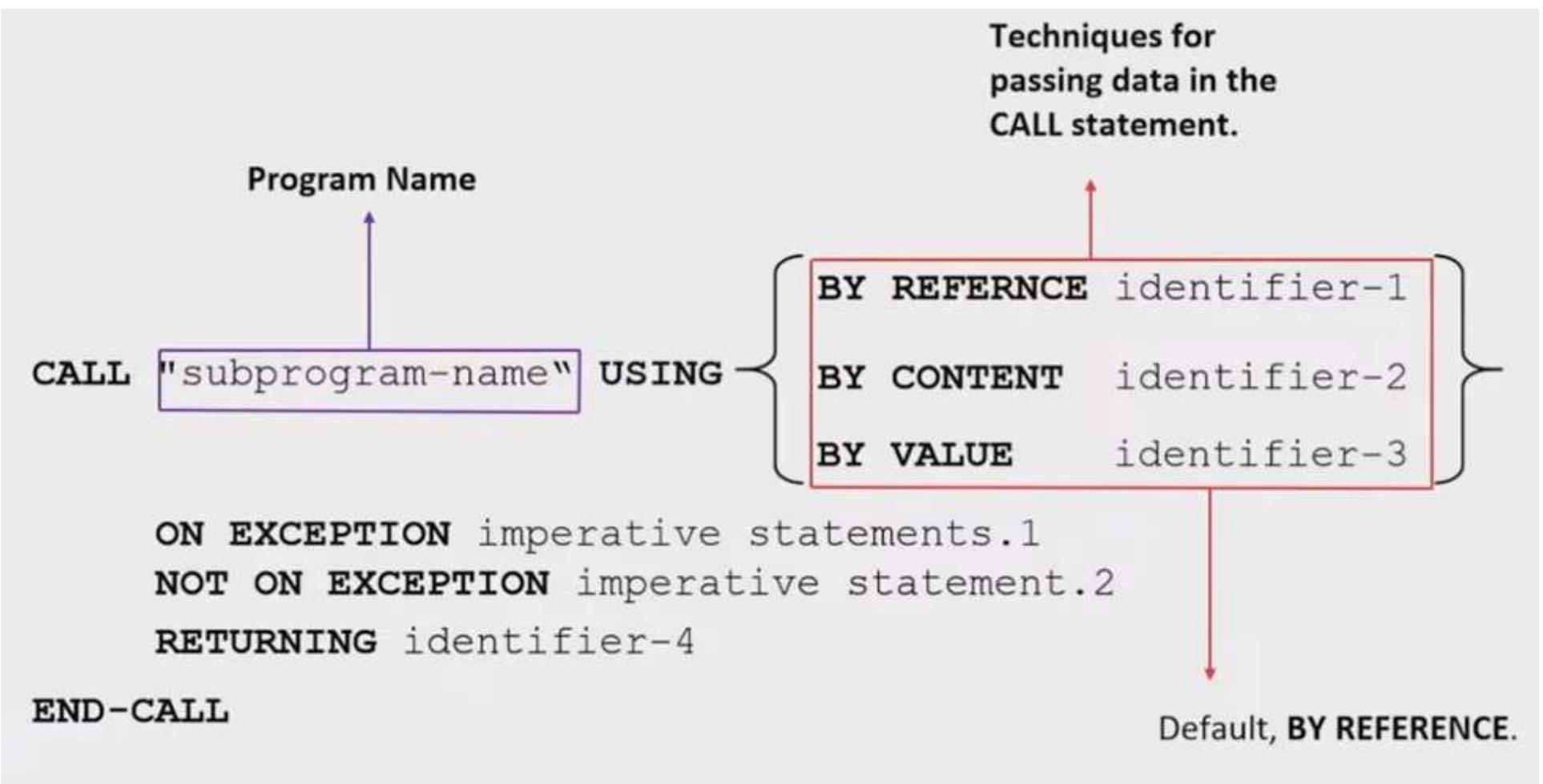
IBM Mainframe Call Statement Example.

MICROFOCUS Call Statement Example.



```
CALL "C:\EMPSAL\EMPTAX01"
      USING EM-SAL, EM-TAX
```

Control Parameters When Passing



Static & Dynamic Subprogram Calls

- In COBOL, static and dynamic calls refer to how one program calls another
 - and specifically, how the called subprogram is linked and loaded.
- Static Call
 - What it is: The called program is linked into the main program at compile or link time.
 - It becomes part of the final executable.
 - Syntax:
CALL 'SUBPROG'.
 - Characteristics:
 - The subprogram must be available at compile/link time.
 - The subprogram is loaded once and stays in memory.
 - Faster execution (no loading at runtime).
 - Any changes to the subprogram require recompiling/relinking the main program.
 - Use case:
 - When the subprogram rarely changes.
 - When performance is critical.

Static & Dynamic Subprogram Calls

- Dynamic Call
 - What it is: The subprogram is not linked into the main program.
 - It's loaded at runtime from disk (e.g., .so, .dll, or .gnt file).
 - Syntax:
CALL WS-PROGRAM-NAME. *> Name is in a variable
 - Characteristics:
 - Subprogram name is stored in a data item, allowing runtime flexibility.
 - You can replace the subprogram file without recompiling the caller.
 - Slightly slower because it loads at runtime.
 - Needs correct environment setup (e.g., file paths).
 - Use case:
 - When you need to switch subprograms dynamically (e.g., plugin-like behavior).
 - For modular systems where you update parts independently.

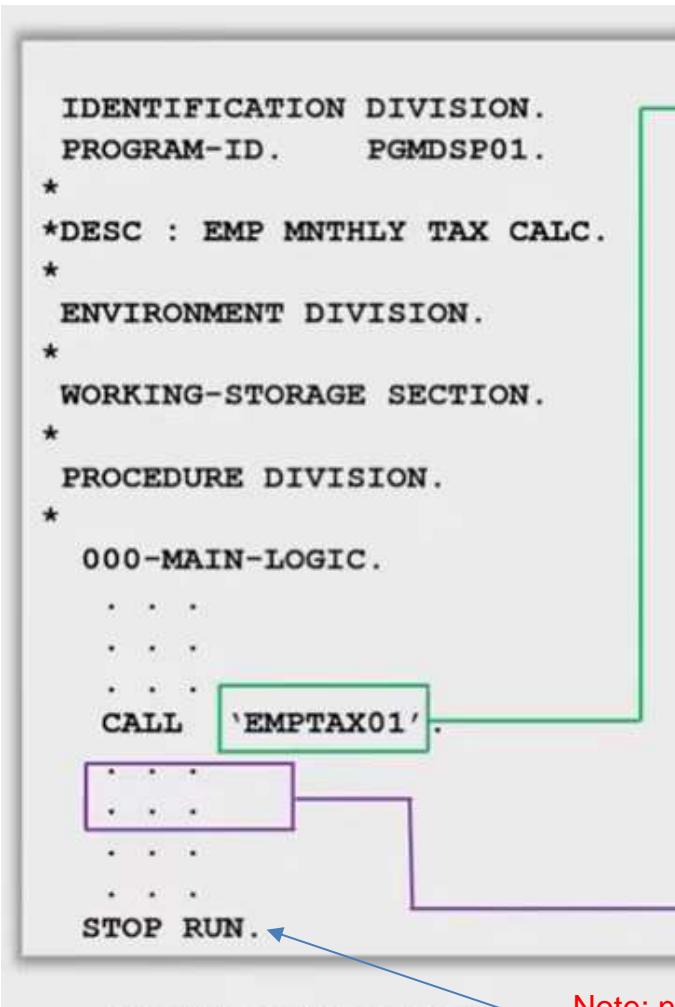
Static & Dynamic Subprogram Calls

Feature	Static Call	Dynamic Call
Program name	Literal	Variable
Linking	Compile/link time	Runtime
Performance	Faster	Slightly slower
Flexibility	Less flexible	Highly flexible
Recompile needed	Yes (if subprogram changes)	No (as long as interface matches)
Memory use	Stays in memory	Can be reloaded or replaced

Notes:

- Static calls use more memory overall because all code is loaded upfront.
- Dynamic calls are powerful, but must be handled carefully to avoid runtime errors (e.g., file not found, interface mismatch).
- Some COBOL compilers (like Micro Focus) allow you to override this behavior globally using compiler/linker directives like or NODYNAM.

Calling A Subprogram

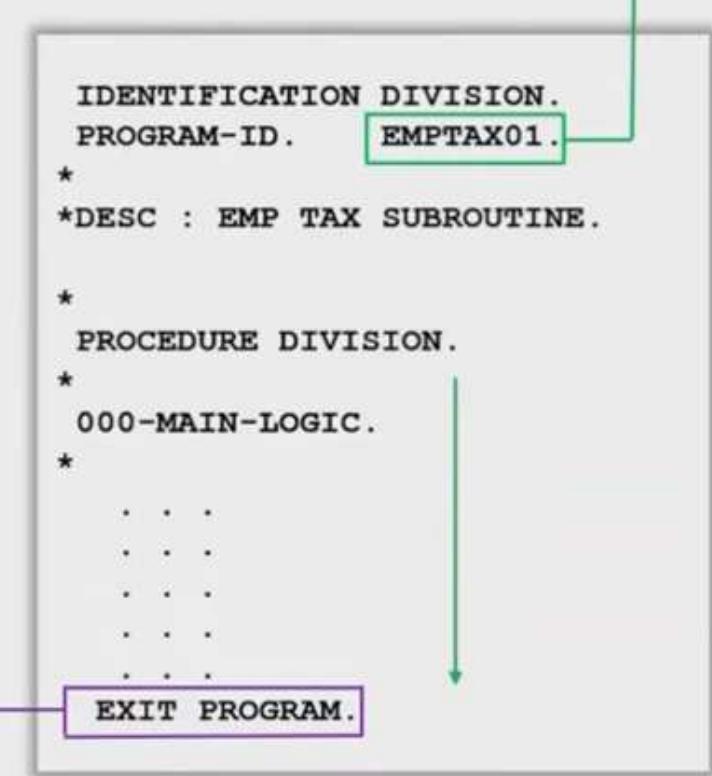


Main-program
CALL
subprogram
program.

Control
return to
calling
program.

Calling Program.

Note: never use "STOP RUN" in a subprogram –
it will stop everything.



Called Program.

Calling A Subprogram With Parameters

```
IDENTIFICATION DIVISION.
PROGRAM-ID.      PGMDSP01.

*DESC : EMP MNTHLY TAX CALC.

*ENVIRONMENT DIVISION.

*WORKING-STORAGE SECTION.
01 EM-SAL      PIC 9(04).
01 EM-TAX      PIC 9(06).

PROCEDURE DIVISION.
000-MAIN-LOGIC.

.
.
.

CALL      'EMPTAX01'
        USING EM-SAL, EM-TAX

.
.
.

STOP RUN.
```

Note: 1st two data items match parameters

Main-program
CALL
subprogram
program.

Control
return to
calling
program.

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. EMPTAX01.  
*  
*DESC : EMP TAX SUBROUTINE.  
*  
LINKAGE SECTION.  
01 WS-SAL      PIC 9(04).  
01 WS-TAX      PIC 9(06).  
PROCEDURE DIVISION USING  
      WS-SAL, WS-TAX.
```

EXIT PROGRAM.

Calling program.

Called program

Subprogram Notes

- A subprogram must be compiled before the main program.
- A subprogram should end with an EXIT PROGRAM or a GOBACK statement.
- A static call is faster than a dynamic call because the main program and called programs are part of the same load modules.
- The sequence of fields in a call statement and the linkage section of the called program should be the same.

Calling a Subprogram

coboltut.cob

```
2 IDENTIFICATION DIVISION.  
3 PROGRAM-ID. coboltut.  
4 DATA DIVISION.  
5 WORKING-STORAGE SECTION.  
6     01 Num1    PIC 9 VALUE 5.  
7     01 Num2    PIC 9 VALUE 4.  
8     01 Sum1    PIC 99.  
9 PROCEDURE DIVISION.  
10 CALL 'GETSUM' USING Num1, Num2, Sum1.  
11 DISPLAY Num1 " + " Num2 " = " Sum1.  
12  
13 STOP RUN.
```

GETSUM.cob

```
2 IDENTIFICATION DIVISION.  
3 PROGRAM-ID. GETSUM.  
4 DATA DIVISION.  
5 LINKAGE SECTION.  
6     01> LNum1    PIC 9 VALUE 5.  
7     01> LNum2    PIC 9 VALUE 4.  
8     01> LSum1    PIC 99.  
9 PROCEDURE DIVISION USING LNum1, LNum2, LSum1.  
10     COMPUTE LSum1 = LNum1 + LNum2.  
11 EXIT PROGRAM.
```

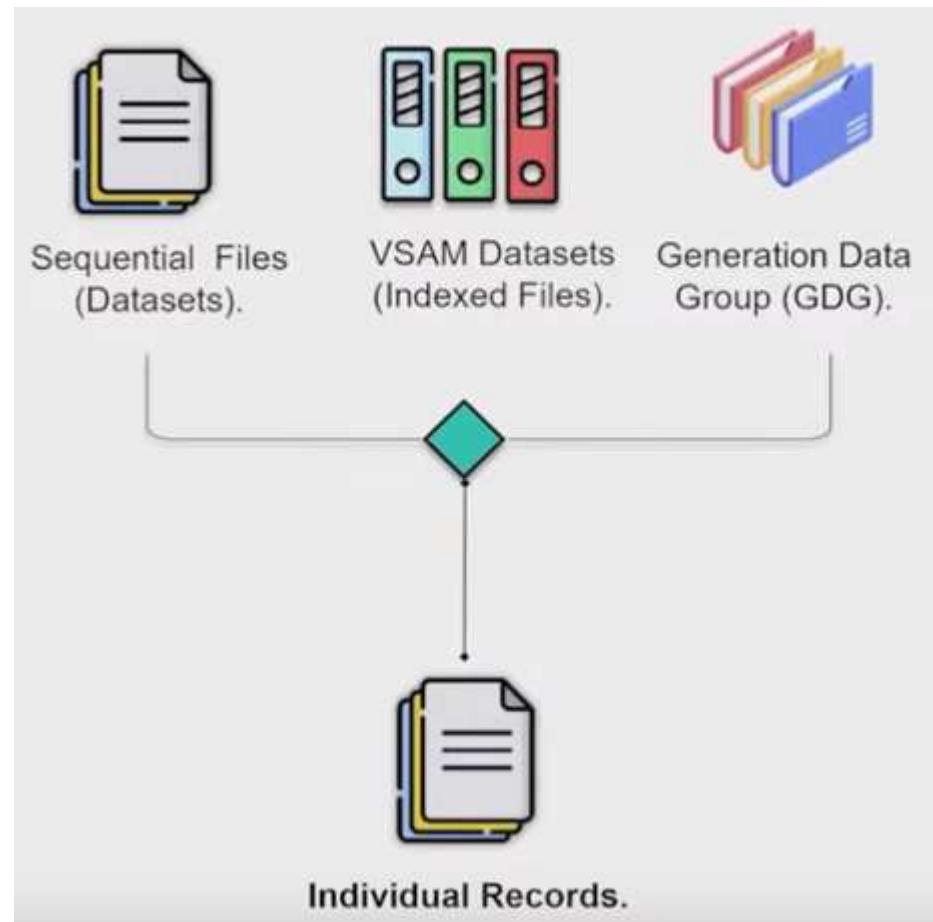
```
>> cobc -m GETSUM.cob  
>> cobc -x coboltut.cob  
>> ./coboltut  
5 + 4 = 09
```

Changes to LSum in the subprogram will be reflected in Sum1 in the calling program

Compile subprogram with -m
Compile main program with -x

Files

Files In The World Of COBOL



In COBOL, What Is A File?

- A FILE is a collection of information where data can be stored permanently.
- Information in a file is divided in the following way:
 - FILE
 - RECORD
 - FIELD
 - CHARACTERS
- A FILE is identified by a file name.
- A RECORD is identified by a record name.
- A FIELD is identified by a field name.
- All names are user defined names and must be between 1-30 alphanumeric characters not starting with a number.

Life Cycle Of A File

- Two steps:
 - Declare a file
 - Process a file

```
ENVIRONMENT DIVISION.  
INPUT-OUTPUT SECTION.  
FILE-CONTROL.  
  SELECT EMPFILE ASSIGN TO EMPDD  
  ORGANIZATION IS SEQUENTIAL  
  ACCESS MODE IS SEQUENTIAL  
  FILE STATUS IS WS-STAT.
```

- Declaring a file
 - Both the file name and the device that it is stored on will be declared.
 - This will be coded in the ENVIRONMENT Division in the INPUT-OUTPUT Section in the FILE CONTROL paragraph.

Holds the two digit result of an I/O operation

Define the file

Define the record in the file

```
DATA DIVISION.  
FILE SECTION.  
FD EMPFILE  
  RECORDING MODE IS F  
  RECORD CONTAINS 36 CHARACTERS  
  BLOCK CONTAINS 0 RECORDS.  
01 EMPREC.  
  05 EMP-ID PIC X(5).  
  05 FILLER PIC X(2).  
  05 EMP-NAME PIC X(10).  
  05 FILLER PIC X(2).  
  05 EMP-LOC PIC X(10).  
  05 FILLER PIC X(2).
```

Opening A File

- Every file that will be used in a COBOL program will need to be opened before it can be used.
- There are different modes that a file can be opened in:
 - OPEN INPUT – will be used for read operations
 - OPEN OUTPUT – will be used for write operations
 - OPEN INPUT-OUTPUT – will be used for both read and write operations
 - OPEN EXTEND – open to extend a file
- If you want to update data in a file, then it should be opened in INPUT-OUTPUT mode. Make sure that you read the record from the file before updating the record.

FILE STATUS Values When Opening For INPUT

File Status	Meaning
00	Successful open.
05	File opened successfully, but reached end-of-file immediately (common with empty sequential files).
35	File not found (non-OPTIONAL file) — does not exist.
37	File exists but wrong open mode (e.g., opening an output-only file in INPUT mode).
39	Conflict in file attributes (record length, key definition mismatch).
92	Logic error — often file not open when I/O attempted.
93	Record lock error or file in use by another process (system-dependent).

FILE STATUS Values When Opening For INPUT

When opening for OUTPUT

File Status	Meaning
00	Successful open. File created or overwritten depending on environment.
37	Wrong open mode — file is restricted (e.g., read-only system file).
39	File attributes mismatch with program definition.
41	File already open (same program).
93	Record lock or sharing violation.

FILE STATUS Values When Opening For INPUT-OUTPUT

When opening for I-O

File Status	Meaning
00	Successful open.
35	File not found (non-OPTIONAL file).
37	Wrong mode.
39	Attribute mismatch.
92/93	Logic or sharing errors.

Using FILE STATUS

```
SELECT CUSTOMER-FILE
      ASSIGN TO "customer.dat"
      ORGANIZATION IS SEQUENTIAL
      FILE STATUS IS WS-FILE-STATUS.

      WORKING-STORAGE SECTION.
      01 WS-FILE-STATUS PIC XX.

      PROCEDURE DIVISION.
      OPEN INPUT CUSTOMER-FILE
      IF WS-FILE-STATUS NOT = "00"
          DISPLAY "Error opening file: " WS-FILE-STATUS
          STOP RUN
      END-IF.
```

Types Of Files IN COBOL

- Sequential Files
 - Records are stored one after another, in order.
 - Duplicate records are allowed
 - You can only read (or write) sequentially, starting at the beginning.
- Indexed Files
 - Records are stored with one or more keys (primary key + alternate keys).
 - Duplicated records are NOT allowed
 - Access:
 - Random access (direct by key).
 - Sequential access (in key order).
 - Dynamic access (mix of both).
- Relative File
 - Records are stored by relative record number (RRN), like array positions.
 - Access:
 - Direct (random) access: retrieve a record by its number.
 - Sequential access: read records in ascending order of their RRNs.

```
FILE-CONTROL.  
  SELECT EMPFILE ASSIGN TO EMPDD  
  ORGANIZATION IS SEQUENTIAL  
          INDEXED  
          RELATIVE  
  ACCESS MODE IS SEQUENTIAL  
          INDEXED  
          RELATIVE  
  RECORD KEY IS ACT-NO  
  RELATIVE KEY IS RNDM-NO.
```

Sequential Files

- To open a file for sequential access, you use the OPEN statement with the file name you defined in the FILE-CONTROL paragraph (in the ENVIRONMENT DIVISION).
1. Define the file in ENVIRONMENT DIVISION

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
 SELECT INFILE ASSIGN TO 'input.dat'
 ORGANIZATION IS SEQUENTIAL.
 2. Describe the record in DATA DIVISION

DATA DIVISION.
FILE SECTION.
FD INFILE.
01 IN-REC.
 05 IN-NAME PIC X(20).
 05 IN-AMOUNT PIC 9(5)V99.
 3. OPEN the file in PROCEDURE DIVISION

PROCEDURE DIVISION.
OPEN INPUT INFILE
PERFORM UNTIL EOF
 READ INFILE
 AT END MOVE "Y" TO EOF
 NOT AT END DISPLAY IN-REC
 END-READ
END-PERFORM
CLOSE INFILE
STOP RUN.

Sequential Files

- OPEN modes for sequential access
 - OPEN INPUT file-name → For reading an existing sequential file.
 - OPEN OUTPUT file-name → For writing a new sequential file (any existing file is usually replaced).
 - OPEN EXTEND file-name → For adding records to the end of an existing sequential file.
 - OPEN I-O file-name → For reading and updating (rare with sequential, more common with indexed/relative).
- Example — Writing to a sequential file

OPEN OUTPUT OUTFILE
WRITE OUT-REC
WRITE OUT-REC
CLOSE OUTFILE

- Example — Reading a sequential file

OPEN INPUT INFILE
READ INFILE
CLOSE INFILE

Indexed Files

- Indexed files are the most common in real-world COBOL applications.
- To open a file for indexed access, you use the OPEN statement with the appropriate mode (INPUT, OUTPUT, I-O, or EXTEND), after defining the file as ORGANIZATION IS INDEXED in the ENVIRONMENT DIVISION.

Indexed Files: Define The File

- Define the file (ENVIRONMENT DIVISION)

ENVIRONMENT DIVISION.

INPUT-OUTPUT SECTION.

FILE-CONTROL.

SELECT CUST-FILE ASSIGN TO "customer.dat"

ORGANIZATION IS INDEXED

ACCESS MODE IS DYNAMIC

RECORD KEY IS CUST-ID

FILE STATUS IS WS-FILE-STATUS.

- Describe the record (DATA DIVISION)

DATA DIVISION.

FILE SECTION.

FD CUST-FILE.

01 CUST-REC.

 05 CUST-ID PIC 9(5).

 05 CUST-NAME PIC X(30).

 05 CUST-BAL PIC 9(7)V99.

WORKING-STORAGE SECTION.

01 WS-FILE-STATUS PIC XX.

Indexed Files: Open The File

- Open the file (PROCEDURE DIVISION)
- You have several options depending on what you want to do:
 - OPEN INPUT → Read existing records only.
 - OPEN OUTPUT → Create a new indexed file (existing file is usually replaced).
 - OPEN I-O → Read and update existing records.
 - OPEN EXTEND → Add new records to the end of the file.
- Example: Reading existing indexed file

PROCEDURE DIVISION.

```
OPEN INPUT CUST-FILE
IF WS-FILE-STATUS NOT = "00"
  DISPLAY "Error opening file: " WS-FILE-STATUS
  STOP RUN
END-IF
```

Indexed File: Examples

- Example: Creating new indexed file
 - OPEN OUTPUT CUST-FILE
- Example: Reading & updating
 - OPEN I-O CUST-FILE

Indexed File: Typical Workflow

* Open file for update

OPEN I-O CUST-FILE

* Read by key

MOVE 12345 TO CUST-ID

READ CUST-FILE

KEY IS CUST-ID

INVALID KEY

DISPLAY "Customer not found"

NOT INVALID KEY

DISPLAY "Customer found: " CUST-NAME

END-READ

* Update the record

ADD 100.00 TO CUST-BAL

REWRITE CUST-REC

* Close file

CLOSE CUST-FILE

STOP RUN.

Relative Files

- Relative files in COBOL are less common today than sequential or indexed, but they're still part of the standard.
- A relative file stores records by Relative Record Number (RRN) — like slots in an array. Each record is accessed using its position number.

Relative Files: Define File

- Define the file (ENVIRONMENT DIVISION)

ENVIRONMENT DIVISION.

INPUT-OUTPUT SECTION.

FILE-CONTROL.

SELECT REL-FILE ASSIGN TO "relfile.dat"

ORGANIZATION IS RELATIVE

ACCESS MODE IS DYNAMIC

RELATIVE KEY IS REL-KEY

FILE STATUS IS WS-FILE-STATUS.

- ORGANIZATION IS RELATIVE → tells COBOL this is a relative file.
- RELATIVE KEY IS REL-KEY → variable in your program that holds the record number (RRN).
- ACCESS MODE can be:
 - SEQUENTIAL → process records in order.
 - RANDOM → jump directly to a specific record number.
 - DYNAMIC → mix both.

Relative Files: Describe Record

DATA DIVISION.

FILE SECTION.

FD REL-FILE.

01 REL-REC.

 05 REL-ID PIC 9(3).

 05 REL-NAME PIC X(20).

WORKING-STORAGE SECTION.

01 REL-KEY PIC 9(3).

01 WS-FILE-STATUS PIC XX.

Relative Files: Open the file

- Like other files, you use OPEN with the mode you need:
 - OPEN INPUT REL-FILE → read existing records.
 - OPEN OUTPUT REL-FILE → create a new relative file (existing one usually erased).
 - OPEN I-O REL-FILE → read and update existing records.
 - OPEN EXTEND REL-FILE → add new records after the last one.

Relative Files Example: Create and write records

PROCEDURE DIVISION.

OPEN OUTPUT REL-FILE

MOVE 1 TO REL-KEY

MOVE 001 TO REL-ID

MOVE "ALICE" TO REL-NAME

WRITE REL-REC INVALID KEY

DISPLAY "Write failed for record 1".

MOVE 2 TO REL-KEY

MOVE 002 TO REL-ID

MOVE "BOB" TO REL-NAME

WRITE REL-REC

CLOSE REL-FILE

STOP RUN.

Relative Files Example: Read a specific record

OPEN I-O REL-FILE

MOVE 2 TO REL-KEY

READ REL-FILE

INVALID KEY DISPLAY "No record at position 2"

NOT INVALID KEY DISPLAY "Found: " REL-NAME

END-READ

CLOSE REL-FILE

STOP RUN.

START Statement

- The START statement is used to position an indexed or relative file at a particular logical record, based on the value of the record key.
- The access modes of the START statement are:
 - KEY =
 - Positions the file at the first record whose key exactly matches the specified value.
 - KEY >
 - Positions the file at the first record whose key is greater than the specified value.
 - KEY >=
 - Positions the file at the first record whose key is greater than or equal to the specified value.
 - KEY < (not always supported in all compilers)
 - Positions the file at the last record whose key is less than the specified value.
 - KEY <= (not always supported in all compilers)
 - Positions the file at the last record whose key is less than or equal to the specified value.
-  Key Notes:
 - The START statement requires that the file be opened in INPUT, I-O, or EXTEND.
 - It is only valid for indexed or relative files (not sequential).
 - After a START, you normally follow with a READ NEXT or READ PREVIOUS (if supported)¹⁵³ to actually retrieve records.

START Statement Example

```
* Position file at first record with EMP-ID >= WS-SEARCH-ID
MOVE "A100" TO WS-SEARCH-ID
START EMP-FILE KEY >= WS-SEARCH-ID
  INVALID KEY
    DISPLAY "NO RECORD FOUND STARTING AT " WS-SEARCH-ID
    GO TO CLOSE-PARA
  END-START

* Now read sequentially from that position
PERFORM UNTIL WS-FILE-STATUS NOT = "00"
  READ EMP-FILE NEXT RECORD
  AT END
    EXIT PERFORM
  END-READ

  DISPLAY "ID=" EMP-ID "  NAME=" EMP-NAME "  SAL=" EMP-SALARY
END-PERFORM.
```

Relative vs. Indexed Files in COBOL

- Relative File
 - Records are stored in slots, identified by a Relative Record Number (RRN).
 - Think of it like an array:
 - Record 1 → Slot 1
 - Record 2 → Slot 2
 - ...
 - To access a record, you must know its number.
 - Strengths:
 - Very fast if you know the record's number (direct slot lookup).
 - Simple to set up.
 - Weaknesses:
 - Wastes space if many records are missing (holes).
 - If you don't know the record's number, searching is awkward.
 - No flexibility in choosing keys (only the RRN).
- Analogy: Like a parking lot where each car has a numbered spot. To find a car, you must know its spot number.

Relative vs. Indexed Files in COBOL

- Indexed File
 - Records are stored with keys (like customer ID, employee number, etc.).
 - COBOL maintains an index behind the scenes (like a database B-tree).
 - You can access records by:
 - Primary key (the main identifier).
 - Alternate keys (optional, like secondary indexes).
 - Supports sequential and random access.
 - Strengths:
 - Much more flexible → you can look up by meaningful keys (not just record numbers).
 - Efficient searching thanks to index structure.
 - Allows alternate keys (multiple ways to find a record).
 - Weaknesses:
 - More overhead (index maintenance).
 - Slightly slower than relative for pure "record number" lookups.
 - File structure more complex.
- Analogy: Like a phone book — you can find someone by their name, and you don't need to know their position in the list.

Relative vs. Indexed Files: Use Cases

- Relative File → used when:
 - You always know the record's position (like accessing student #1234 in a fixed roster).
 - Example: A system logging sensor data every second into slot t.
- Indexed File → used when:
 - You need flexible searching.
 - Example: Payroll, banking, inventory → you want to find "Employee 12345" or "Customer by Name".

File Example

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. EMP0002.  
AUTHOR. TOPICTRICK.  
* * *  
ENVIRONMENT DIVISION.  
CONFIGURATION SECTION.  
* * *  
INPUT-OUTPUT SECTION.  
FILE-CONTROL.  
  SELECT EMPRC ASSIGN TO EMPMNTH  
  ORGANIZATION IS SEQUENTIAL  
  ACCESS MODE IS SEQUENTIAL  
  FILE STATUS IS TREMP-ST.  
* * *  
DATA DIVISION.  
FILE SECTION.  
  FD EMPRC.  
  01 EMPLOYEE-REC.  
    05 EMPNO          PIC X(06).  
    05 FIRSTNAME     PIC X(12).  
* * *  
WORKING-STORAGE SECTION.  
  01 WS-SWITCH.  
    05 EOF-FILE      PIC X VALUE 'N'.  
* * *  
PROCEDURE DIVISION.  
A000-CORE-PROCESSING.  
  OPEN INPUT EMPRC  
* * *  
  READ EMPRC INTO ...  
* * *  
  END-READ  
  CLOSE EMPRC  
* * *  
  STOP RUN.
```

File to be using

Layout of file

Create Data File

Mac

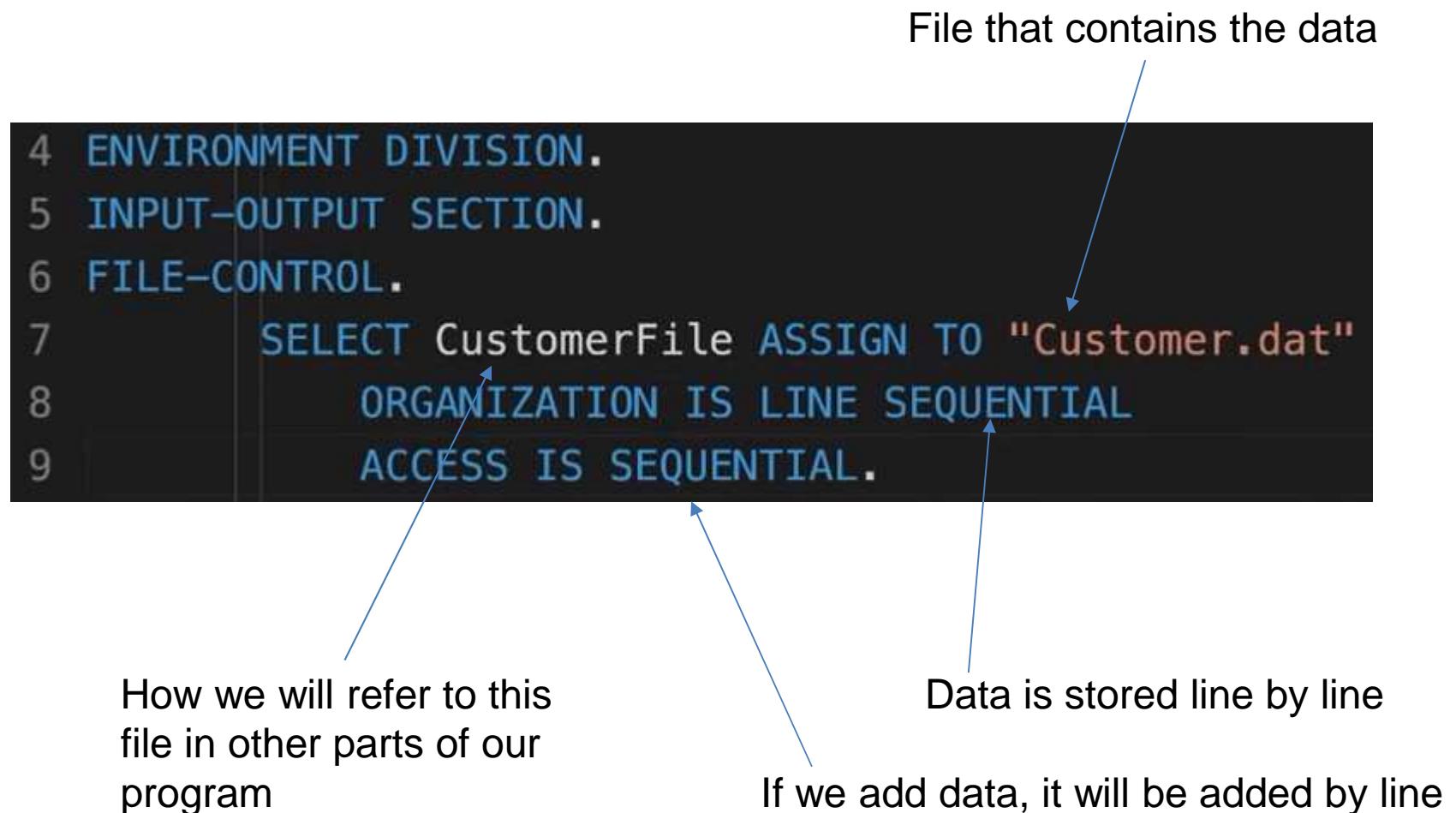
```
touch Customer.dat
```

Windows (admin mode)

```
fsutil file createnew Customer.dat 2000
```

2000 bytes

Assign Name To File



File Description

FD =
File
Description

```
12 FILE SECTION.  
13 FD CustomerFile.  
14 01 CustomerData.  
15      02 IDNum      PIC 9(5).  
16      02 CustName.  
17          03 FirstName PIC X(15).  
18          03 FirstName PIC X(15).
```

Customer record layout

Working Storage

```
20 WORKING-STORAGE SECTION.  
21 01 WSCustomer.  
22      02 WSIDNum    PIC 9(5).  
23      02 WSCustName.  
24          03 WSFirstName PIC X(15).  
25          03 WSLastName PIC X(15).
```

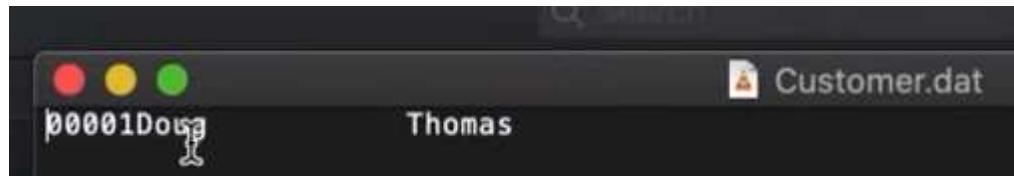
Note: Copied from file description and variable names were changed.

Write Record To File

Sends data
to file

```
27 PROCEDURE DIVISION.  
28 OPEN OUTPUT CustomerFile.  
29      MOVE 00001 TO IDNum.  
30      MOVE 'Doug' TO FirstName.  
31      MOVE 'Thomas' TO LastName.  
32      WRITE CustomerData  
33      END-WRITE.  
34 CLOSE CustomerFile.
```

Note: to run program, CONFIGURATION SECTION must be deleted.



Enter Customer Information

```
26 PROCEDURE DIVISION.  
27 OPEN EXTEND CustomerFile.  
28     DISPLAY "Customer ID " WITH NO ADVANCING  
29     ACCEPT IDNum  
30     DISPLAY "Customer First Name " WITH NO  
31     ADVANCING  
32     ACCEPT FirstName  
33     DISPLAY "Customer Last Name " WITH NO  
34     ADVANCING  
35     ACCEPT LastName  
36     WRITE CustomerData  
37     END-WRITE.  
38 CLOSE CustomerFile.
```

"append" data to file

```
Customer ID 00002  
Customer First Name  
Sally  
Customer Last Name S  
mith
```



Reading Data From A File

```
19 WORKING-STORAGE SECTION.  
20 01 WSCustomer.  
21      02 WSIDNum      PIC 9(5).  
22      02 WSCustName.  
23          03 WSFirstName PIC X(15).  
24          03 WSLastName  PIC X(15).  
25 01 WSEOF  PIC A(1).
```

New Item (EOF)

```
27 PROCEDURE DIVISION.  
28 OPEN INPUT CustomerFile.  
29     PERFORM UNTIL WSEOF='Y'  
30         READ CustomerFile INTO WSCustomer  
31             AT END MOVE 'Y' TO WSEOF  
32             NOT AT END DISPLAY WSCustomer  
33         END-READ  
34     END-PERFORM  
35 CLOSE CustomerFile.
```

Open for input

00001	Doug
Thomas	
00002	Sally
Smith	

Optional Files

- In COBOL, when you define a file as OPTIONAL in the SELECT statement (in the Environment Division → Input-Output Section → File-Control Paragraph), it means:
 - The program is allowed to run even if that file does not physically exist when the program starts.
- Why it's used
 - Normally, if you try to open a non-existent file in COBOL, the runtime will throw an error and stop.
 - If the file is declared OPTIONAL, COBOL will treat the missing file as if it were empty (for input) or will create it when opened for output.
- Syntax example

```
SELECT OPTIONAL CUSTOMER-FILE  
ASSIGN TO "customer.dat"  
ORGANIZATION IS LINE SEQUENTIAL.
```

Optional Files

- Behavior based on open mode:
- INPUT mode:
 - If the file exists → it's opened normally.
 - If it doesn't exist → the file is treated as empty (EOF is immediately returned).
- OUTPUT mode:
 - If the file doesn't exist → it is created.
 - If it exists → it is replaced or truncated depending on environment settings.
- I-O mode:
 - If the file doesn't exist → you'll usually get an error unless your system allows creation in I-O mode.

Reports



Reports

```
6 FILE-CONTROL.  
7     SELECT CustomerReport ASSIGN TO "CustReport.  
rpt"  
8         ORGANIZATION IS LINE SEQUENTIAL.  
9     SELECT CustomerFile ASSIGN TO "Customer.dat"  
10        ORGANIZATION IS LINE SEQUENTIAL.
```

```
12 FILE SECTION.  
13 FD CustomerReport.  
14 01 PrintLine PIC X(44).  
15  
16 FD CustomerFile.  
17 01 CustomerData.  
18     02 IDNum    PIC 9(5).  
19     02 CustName.  
20         03 FirstName PIC X(15).  
21         03 LastName PIC X(15).  
22     88 WSEOF VALUE HIGH-VALUE.
```

Creating The Report

- Note that the report will be made up of multiple parts: header, footer, columns, etc.

Use "FILLER" when
you don't need to
identify labels

```
24 WORKING-STORAGE SECTION.  
25 01 PageHeading.  
26      02 FILLER PIC X(13) VALUE "Customer List".  
27 01 PageFootning.  
28      02 FILLER PIC X(15) VALUE SPACE.  
29      02 FILLER PIC X(7) VALUE "Page : ".  
30      02 PrnPageNum PIC Z9.  
31 01 Heads PIC X(36) VALUE "IDNum      FirstName  
      LastName".  
32 01 CustomerDetailLine.  
33      02 FILLER PIC X VALUE SPACE.  
34      02 PrnCustID PIC 9(5).  
35      02 FILLER PIC X(4) VALUE SPACE.  
36      02 PrnFirstName PIC X(15).  
37      02 FILLER PIC XX VALUE SPACE.  
38      02 PrnLastName PIC X(15).  
39 01 ReportFootning PIC X(13) VALUE "END OF REPORT".  
40 01 LineCount PIC 99 VALUE ZERO.  
41      88 NewPageRequired VALUE 40 THRU 99.  
42 01 PageCount PIC 99 VALUE ZERO.
```

FILLER

- In COBOL, FILLER is a special keyword used in data definitions to define fields that do not need a name and will not be referenced directly in the program. These are essentially "placeholder" fields, often used to:
 - Skip unused parts of a data record:
 - Sometimes, input or output records contain fields you don't care about. You use FILLER to reserve space for them without needing to assign a name.
 - Align data fields:
 - When dealing with fixed-format records, FILLER helps maintain proper spacing and alignment.
 - Reserve space for future use:
 - You can define fields with FILLER now and later rename or redefine them if needed.
 - Output formatting:
 - In reports, FILLER can be used to create spaces or static formatting characters (like dashes or blank areas).

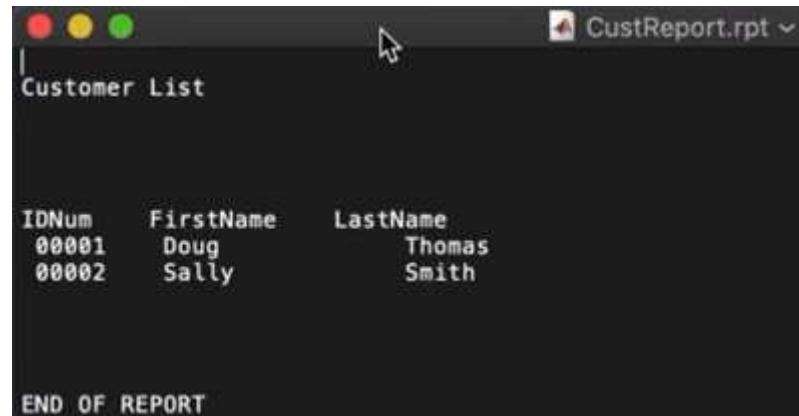
```
01 CUSTOMER-RECORD.  
 05 CUSTOMER-ID      PIC X(10).  
 05 FILLER          PIC X(5).      *> Unused space  
 05 CUSTOMER-NAME   PIC X(20).  
 05 FILLER          PIC X(10) VALUE SPACES. *> Formatting or spacing
```

Report Control Code

```
44 PROCEDURE DIVISION.  
45 OPEN INPUT CustomerFile.  
46 OPEN OUTPUT CustomerReport  
47 PERFORM PrintPageHeading  
48 READ CustomerFile  
      AT END SET WSEOF TO TRUE  
50 END-READ  
51 PERFORM PrintReportBody UNTIL WSEOF  
52 WRITE PrintLine FROM ReportFooting AFTER ADVANCING 5  
      LINES  
53 CLOSE CustomerFile, CustomerReport.  
54 STOP RUN.
```

Report Routines

```
56 PrintPageHeading.  
57 WRITE PrintLine FROM PageHeading AFTER ADVANCING Page  
58 WRITE PrintLine FROM Heads AFTER ADVANCING 5 LINES  
59 MOVE 3 TO LineCount  
60 ADD 1 TO PageCount.  
61  
62 PrintReportBody.  
63 IF NewPageRequired  
64     MOVE PageCount TO PrnPageNum  
65     WRITE PrintLine FROM PageFooting AFTER  
          ADVANCING 5 LINES  
66     PERFORM PrintPageHeading  
67 END-IF  
68 MOVE IDNum TO PrnCustID  
69 MOVE FirstName to PrnFirstName  
70 MOVE LastName to PrnLastName  
71 WRITE PrintLine FROM CustomerDetailLine AFTER  
          ADVANCING 1 LINE  
72 ADD 1 TO LineCount  
73 READ CustomerFile  
74     AT END SET WSEOF TO TRUE  
75 END-READ.
```



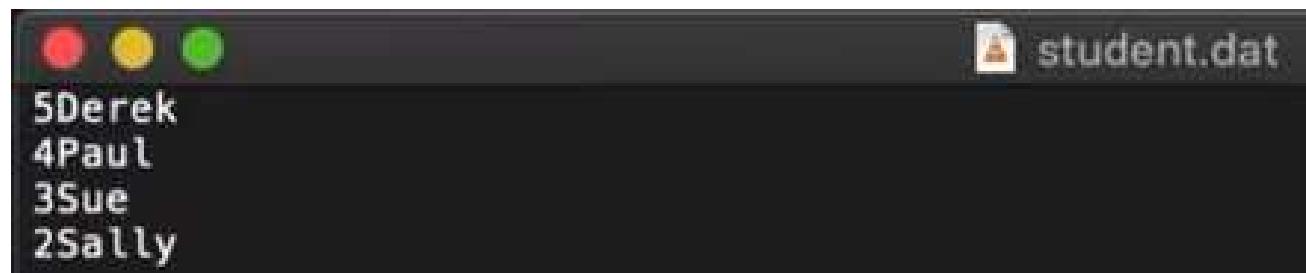
IDNum	FirstName	LastName
00001	Doug	Thomas
00002	Sally	Smith

END OF REPORT

Sort Files

Create Data File

- File name is student.dat
- Preloaded with data.
- Note that the key values (1-5) are in backwards order.
- We will be sorting this data.



INPUT PROCEDURE and OUTPUT PROCEDURE

- INPUT PROCEDURE and OUTPUT PROCEDURE are special options you can use in SORT and MERGE statements to control how records are read into and written out of the sort/merge process.
- They let you do processing before and after the actual sort/merge happens.

INPUT PROCEDURE

- Runs before the SORT or MERGE begins reading records.
- You can:
 - Read records from one or more input files.
 - Filter out unwanted records.
 - Modify data before it's given to the sort.
- In the INPUT PROCEDURE, you don't write to an output file—you instead use RELEASE to pass records to the sort work file.
- Example:

```
SORT SORT-FILE
  ON ASCENDING KEY CUST-NAME
  INPUT PROCEDURE IS READ-CUSTOMERS
  OUTPUT PROCEDURE IS WRITE-CUSTOMERS.

  READ-CUSTOMERS.
  READ INFILE
    AT END GO TO RC-END.
  MOVE IN-NAME TO CUST-NAME
  MOVE IN-AMOUNT TO CUST-AMOUNT
  RELEASE SORT-REC
  GO TO READ-CUSTOMERS.

RC-END.
  EXIT.
```

In the INPUT PROCEDURE, the input file is opened, records are read and edited and then are released to the sorting operation. Finally the file will be closed;

OUTPUT PROCEDURE

- Runs after the SORT or MERGE has completed.
- You can:
 - Post-process the sorted/merged records.
 - Format data for reports.
 - Write the sorted records to one or more files.
- In the OUTPUT PROCEDURE, you use RETURN to get the sorted records from the sort work file.
- Example (continuing from previous):

```
WRITE-CUSTOMERS.  
  RETURN SORT-FILE  
    AT END GO TO WC-END.  
  WRITE OUT-REC FROM SORT-REC  
  GO TO WRITE-CUSTOMERS.  
  
WC-END.  
  EXIT.
```

In the OUTPUT PROCEDURE, output file is opened, the sorted record is returned to the Output record and then the record will be written. Finally the file will be closed.

Creating A Program



Working With Customer Data

- Sequential files allow us to access data in them sequentially.
- We will be using a file with keys.
- To do that, the file has to be an indexed file as opposed to non-indexed.
- This will allow us to randomly access data as opposed to sequentially access data.

```
SELECT CustomerFile ASSIGN TO "customer.txt"
      ORGANIZATION IS INDEXED
      ACCESS MODE IS RANDOM
      RECORD KEY IS IDNum.
```

- Program that we will create will have a menu system that allows you to add, update and delete customer data

Setting Up File And What Data Looks Like

```
6 FILE-CONTROL.  
7   SELECT CustomerFile ASSIGN TO "customer.txt"  
8     ORGANIZATION IS INDEXED  
9     ACCESS MODE IS RANDOM  
10    RECORD KEY IS IDNum.  
11 DATA DIVISION.  
12 FILE SECTION.  
13 FD CustomerFile.  
14 01 CustomerData.  
15    02 IDNum    PIC 99.  
16    02 FirstName PIC X(15).  
17    02 LastName PIC X(15).
```

Define Variables

```
19 WORKING-STORAGE SECTION.  
20 01 Choice PIC 9.  
21 01 StayOpen PIC X VALUE 'Y'.  
22 01 CustExists PIC X.
```

Code

```
24 PROCEDURE DIVISION.
25 StartPara.
26     OPEN I-O CustomerFile.
27     PERFORM UNTIL StayOpen='N'
28         DISPLAY " "
29         DISPLAY "CUSTOMER RECORDS"
30         DISPLAY "1 : Add Customer"
31         DISPLAY "2 : Del Customer"
32         DISPLAY "3 : Update Customer"
33         DISPLAY "4 : Get Customer"
34         DISPLAY "0 : Quit"
35         DISPLAY ":" WITH NO ADVANCING
36         ACCEPT Choice
37         EVALUATE Choice
38             WHEN 1 PERFORM AddCust
39             WHEN 2 PERFORM DeleteCust
40             WHEN 3 PERFORM UpdateCust
41             WHEN 4 PERFORM GetCust
42             WHEN OTHER move 'N' TO StayOpen
43         END-EVALUATE
44     END-PERFORM.
45     CLOSE CustomerFile
46     STOP RUN.
47
48 CLOSE CustomerFile.
49 STOP RUN.
```

AddCust Routine

```
48 AddCust.  
49     DISPLAY ""  
50     DISPLAY "Enter ID : " WITH NO ADVANCING.  
51     ACCEPT IDNum.  
52     DISPLAY "Enter First Name : " WITH NO  
53         ADVANCING.  
54     ACCEPT FirstName.  
55     DISPLAY "Enter Last Name : " WITH NO  
56         ADVANCING.  
57     ACCEPT LastName. I  
58     DISPLAY ""  
59     WRITE CustomerData  
          INVALID KEY DISPLAY "ID Taken"  
      END-WRITE.
```

DeleteCust Routine

```
61 DeleteCust.  
62      DISPLAY " "  
63      DISPLAY "Enter Customer ID to Delete : " WITH  
64      NO ADVANCING.  
64      ACCEPT IDNum.  
65      DELETE CustomerFile  
66      INVALID KEY DISPLAY "Key Doesn't Exist"  
67      END-DELETE.
```

UpdateCust Routine

```
69 UpdateCust.  
70     MOVE 'Y' TO CustExists.  
71     DISPLAY ""  
72     DISPLAY "Enter ID to Update : " WITH NO  
ADVANCING.  
73     ACCEPT IDNum.  
74     READ CustomerFile  
        INVALID KEY MOVE 'N' TO CustExists  
    END-READ  
77     IF CustExists='N'  
        DISPLAY "Customer Doesn't Exist"  
    ELSE  
        DISPLAY "Enter the New First Name : "  
        WITH NO ADVANCING  
        ACCEPT FirstName  
        DISPLAY "Enter the New Last Name : " WITH  
        NO ADVANCING  
        ACCEPT LastName  
    END-IF.  
85     REWRITE CustomerData  
        INVALID KEY DISPLAY "Customer Not Updated"  
87     END-REWRITE.
```

REWRITE

- The REWRITE statement is used to update (modify) an existing record in a file that has already been read into the program's record area.
- Here's the flow:
 1. The program reads a record from the file.
 2. It changes the record in memory.
 3. REWRITE writes the modified record back in the same position.
- Key points:
 - You can only use REWRITE on files opened in I-O mode.
 - It replaces the current record in the file with the contents of the program's record area.
 - You must read the record first (with READ) before rewriting it — this sets the file's current record pointer.
 - The record length after REWRITE must match the length defined for that file.
- Gotchas:
 - If you try to REWRITE without a successful READ first, you'll get an error.
 - On sequential files, the rewritten record must be the same size as the original.
 - On indexed/relative files, the key field cannot be changed in a REWRITE — you must delete and re-add the record if you need to change the key.

```
REWRITE EMP-REC FROM WS-EMP-REC  
END-REWRITE .
```

Ending A Random Read

- Should an AT END clause be used when you are issuing a random read?
- No, you cannot use the AT END clause because you are reading records directly via a key.
- You'll never have an end-of-file condition in such a scenario.
- However, you should code a INVALID KEY clause in order to handle "record not found" - file status 23.
- Note that in our example this was not a problem because the reading stopped when the user quit.

```
READ EMP-REC INTO WS-EMP-REC
  INVALID KEY
    DISPLAY "EMP REC NOT FND"
    DISPLAY "FLE ST CDE:" WS-STAT-CDE
  END-READ
```

Run Program

1

```
CUSTOMER RECORDS
1 : Add Customer
2 : Delete Customer
3 : Update Customer
4 : Get Customer
0 : Quit
: 1
```

```
Enter ID : 01
Enter First Name : Derek
Enter Last Name : Banas
```

```
CUSTOMER RECORDS
1 : Add Customer
2 : Delete Customer
3 : Update Customer
4 : Get Customer
0 : Quit
: 4
```

```
Enter Customer ID to Find : 01
ID : 01
First Name : Derek
Last Name : Banas
```

2

```
CUSTOMER RECORDS
1 : Add Customer
2 : Delete Customer
3 : Update Customer
4 : Get Customer
0 : Quit
: 3
```

```
Enter ID to Update : 01
Enter the New First Name : Paul
Enter the New Last Name : Banas
```

```
CUSTOMER RECORDS
1 : Add Customer
2 : Delete Customer
3 : Update Customer
4 : Get Customer
0 : Quit
: 4
```

```
Enter Customer ID to Find : 01
ID : 01
First Name : Paul
Last Name : Banas
```

3

```
CUSTOMER RECORDS
1 : Add Customer
2 : Delete Customer
3 : Update Customer
4 : Get Customer
0 : Quit
: 2
```

```
Enter Customer ID to Delete : 01
```

```
CUSTOMER RECORDS
1 : Add Customer
2 : Delete Customer
3 : Update Customer
4 : Get Customer
0 : Quit
: 4
```

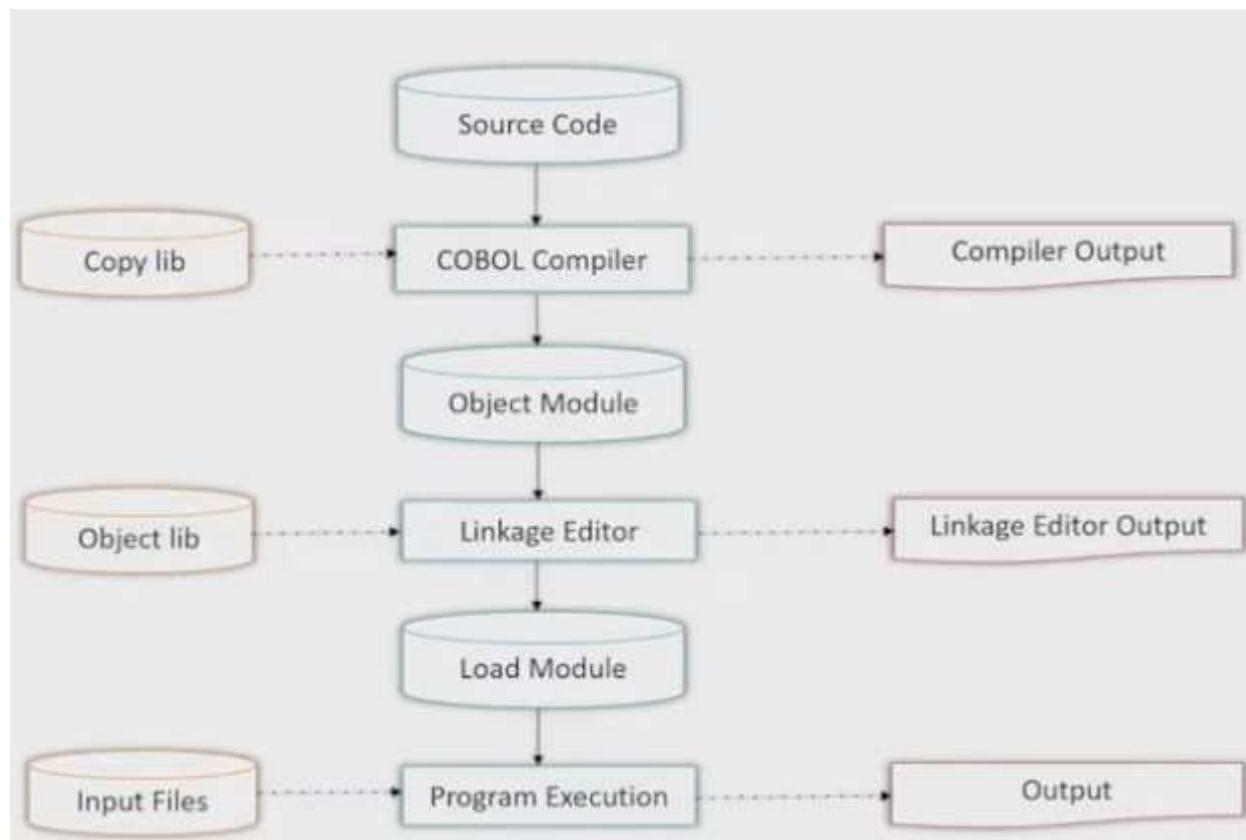
```
Enter Customer ID to Find : 01
Customer Doesn't Exist
```

```
CUSTOMER RECORDS
1 : Add Customer
2 : Delete Customer
3 : Update Customer
4 : Get Customer
0 : Quit
: 0
>> █
```

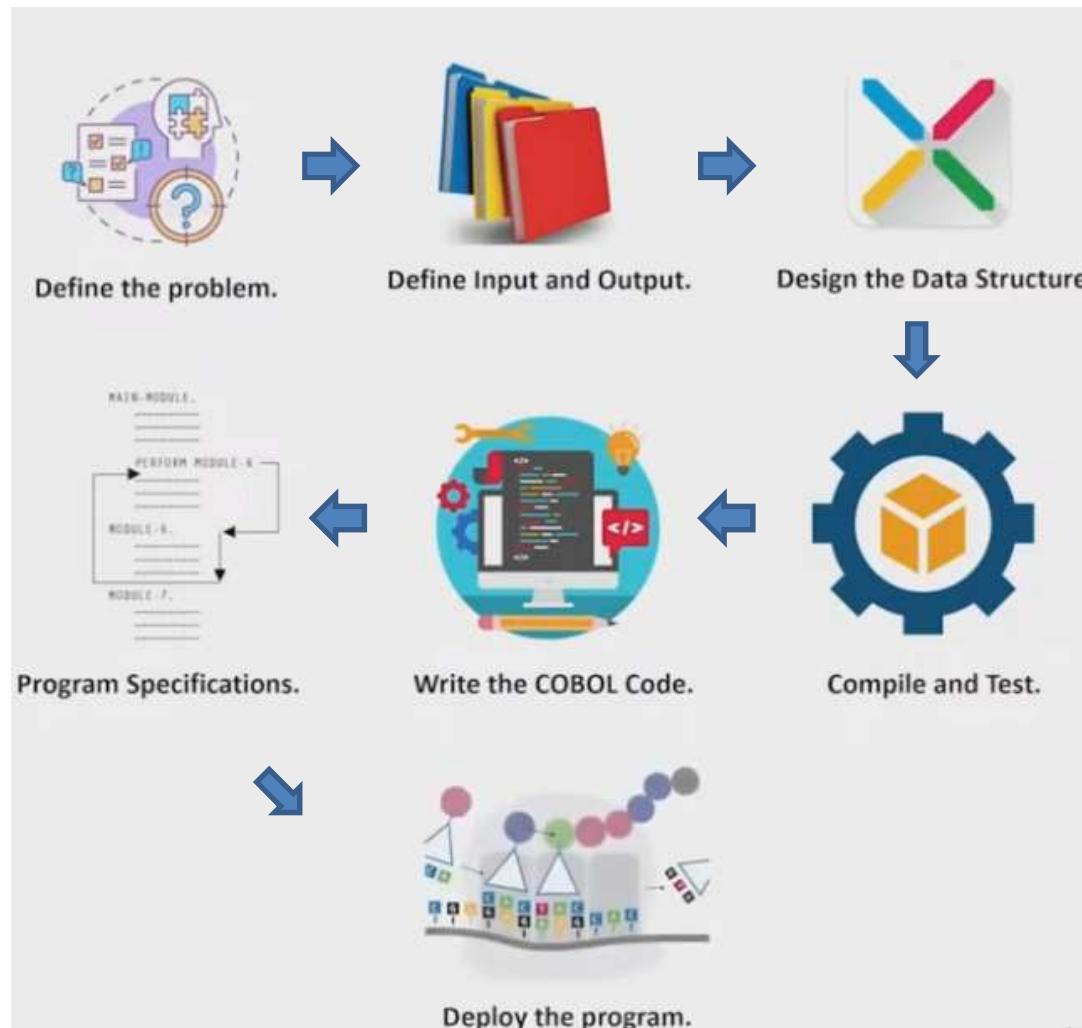
GetCust Routine

```
89 GetCust.  
90     MOVE 'Y' TO CustExists.  
91     DISPLAY ""  
92     DISPLAY "Enter Customer ID to Find : " WITH  
NO ADVANCING.  
93     ACCEPT IDNum.  
94     READ CustomerFile  
95         INVALID KEY MOVE 'N' TO CustExists  
96     END-READ  
97     IF CustExists='N'  
98         DISPLAY "Customer Doesn't Exist"  
99     ELSE  
100         DISPLAY "ID : " IDNum  
101         DISPLAY "First Name : " FirstName  
102         DISPLAY "Last Name : " LastName  
103     END-IF.
```

Program Compilation Process



How To Design A COBOL Program



Environment Division

```
1      >>SOURCE FORMAT FREE
2  IDENTIFICATION DIVISION.
3  PROGRAM-ID. coboltut.
4  ENVIRONMENT DIVISION.
5  INPUT-OUTPUT SECTION.
6  FILE-CONTROL.
7      SELECT WorkFile ASSIGN TO 'work.tmp'.
8      SELECT OrgFile ASSIGN TO 'student.dat'
9          ORGANIZATION IS LINE SEQUENTIAL.
10     SELECT SortedFile ASSIGN TO 'student2.dat'
11         ORGANIZATION IS LINE SEQUENTIAL.
```

Data Division

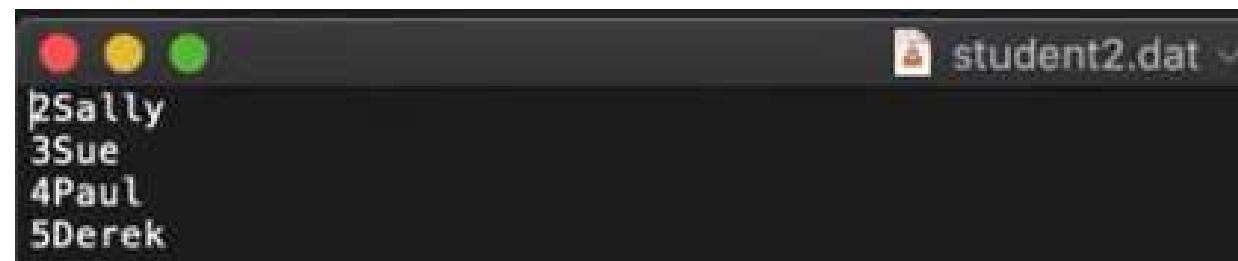
- In COBOL, within the Data Division, FD and SD are used to define files, but they serve different purposes:
- **FD (File Description):** FD stands for **File Description**.
 - It is used for regular (sequential, indexed, or relative) files.
 - Appears in the **File Section** of the Data Division.
 - Describes the structure of a file that your program will read from or write to.
- **SD (Sort Description):** SD stands for **Sort Description**.
 - It is used **only with the SORT verb**, defining a **work file** that COBOL uses internally for sorting.
 - Also appears in the **File Section** of the Data Division.
 - Does **not** refer to an actual physical file on disk—it's a temporary, system-managed file.

Data Division

```
13  DATA DIVISION.  
14  FILE SECTION.  
15  FD OrgFile.  
16  01 StudData.  
17      02 IDNum PIC 9.  
18      02 StudName PIC X(10).  
19  SD WorkFile.  
20  01 WStudData.  
21      02 WIDNum PIC 9.  
22      02 WStudName PIC X(10).  
23  FD SortedFile.  
24  01 SStudData.  
25      02 SIDNum PIC 9.  
26      02 SStudName PIC X(10).
```

Procedure Division

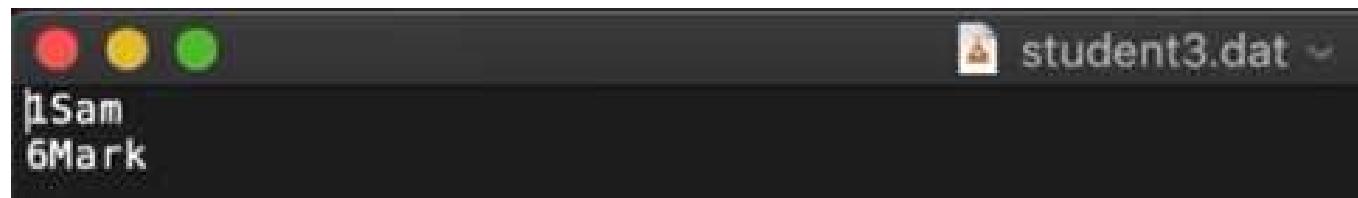
```
29 PROCEDURE DIVISION.  
30  SORT WorkFile ON ASCENDING KEY SIDNum  
31          USING OrgFile  
32          GIVING SortedFile.  
33  
34 STOP RUN.
```



```
student2.dat  
2Sally  
3Sue  
4Paul  
5Derek
```

Merging Files

Create New Data File



Environment Division

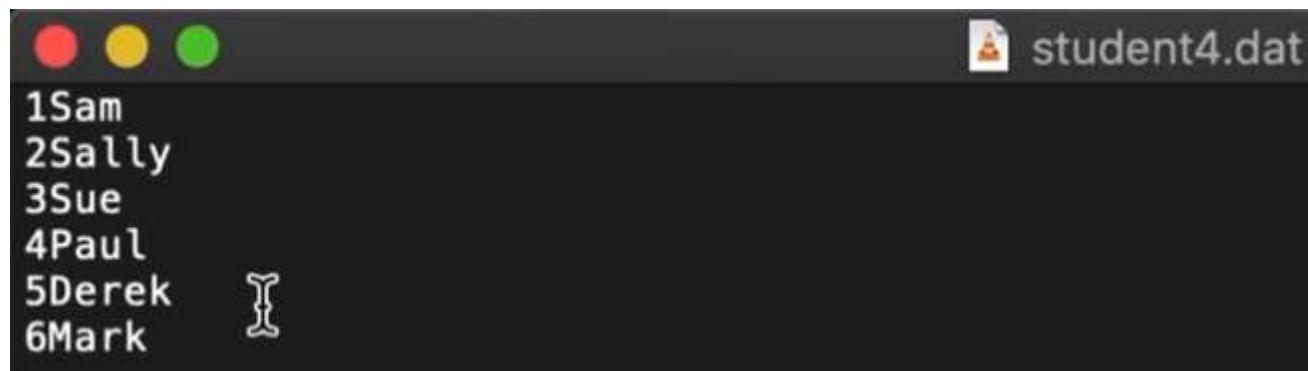
```
4 ENVIRONMENT DIVISION.  
5 INPUT-OUTPUT SECTION.  
6 FILE-CONTROL.  
7     SELECT WorkFile ASSIGN TO 'work.tmp'.  
8     SELECT File1 ASSIGN TO 'student.dat'  
9         ORGANIZATION IS LINE SEQUENTIAL.  
10    SELECT File2 ASSIGN TO 'student3.dat'  
11        ORGANIZATION IS LINE SEQUENTIAL.  
12    SELECT NewFile ASSIGN TO 'student4.dat'  
13        ORGANIZATION IS LINE SEQUENTIAL.
```

Data Division

```
15 DATA DIVISION.  
16 FILE SECTION.  
17 FD File1.  
18 01 StudData.  
19     02 IDNum PIC 9.  
20     02 StudName PIC X(10).  
21 FD File2.  
22 01 StudData.  
23     02 IDNum PIC 9.  
24     02 StudName PIC X(10).  
25 SD WorkFile.  
26 01 WStudData.  
27     02 WIDNum PIC 9.  
28     02 WStudName PIC X(10).  
29 FD NewFile.  
30 01 NStudData.  
31     02 NIDNum PIC 9.  
32     02 NStudName PIC X(10).
```

Procedure Division

```
35 PROCEDURE DIVISION.  
36 MERGE WorkFile ON ASCENDING KEY NIDNum  
37      USING File1, File2  
38      GIVING NewFile.  
39  
40 STOP RUN.
```



```
1Sam  
2Sally  
3Sue  
4Paul  
5Derek  
6Mark
```

YouTube COBOL References

- COBOL 101
<https://www.youtube.com/watch?v=cnz9y9k2jvs>
- COBOL Tutorial : Learn COBOL in One Video
<https://www.youtube.com/watch?v=TBs7HXI76yU>
- Complete COBOL Refresher in 1 Hour #COBOL
<https://www.youtube.com/watch?v=38pNOuGiSmw>
- COBOL Lesson 1 – Introduction
<https://www.youtube.com/watch?v=u9M52sAnrOs>