COP2121

Introduction to COBOL Programming

Lola Haskins

© 2001 Lola Haskins & Melissa Raulston. All rights reserved.

Every attempt has been made to ensure the accuracy of these notes. If you find a mistake or a typo, we want to know! Please send email about these notes to Melissa Raulston, cisco777@alltel.net

TO THE STUDENT:

These notes are as complete as possible with respect to the subject mater of this class, BUT 1. we don't warrant that all things discussed in class and to be tested will be covered here—a course is a fluid animal and 2. you will not find explanations of the programs here. There's really no substitute for being there when it comes to those.

You should also realize that just owning these notes won't guarantee you a high grade. Coming to class and asking questions will make the difference. And, the fact is, it's the instructor's personal opinion that it is most beneficial for a student to take his/her own notes since simply the act of writing information down is a way of learning it. These notes, then, will be useful to see if there's anything missing or unclear afterwards.

A Partial History of Computing as it relates to COBOL…

Many students think that using a non-human device to expand the capabilities of the human mind is something which sets modern society apart from older cultures. Nothing could be further from the truth! The history of the computer begins 5,000 years ago, when the desire for a device that could do large sums quickly gave rise to the abacus. The algorithm (the precise definition of a series of steps taken to produce a particular result), which is the basis of all computer programming, isn't new either. It was first described in a book written by a Tashenkt cleric, Mukhammad ibn Musa Al'Knowarizmi, in the twelfth century. In fact, the slide rule, invented by William Oughtred in 1622, remained a computing requirement for engineering students until the 1970s. Back in the 60's you could always tell an engineering major; he was the one with the slide rule on his belt. Until he started carrying a four-function calculator.

What we've been talking about so far has been pre-history as far as the computer is concerned, because we've been dealing with machines designed to do only certain specific operations, not with general-purpose devices which operate by commands. Even so, the idea of the computer as we know it isn't, as you might think, a twentieth century phenomenon. An English mathematician named Charles Babbage (working on a government grant—sound familiar?) first came up with it in the 1830s. Babbage called his device the analytical engine, and it had all the parts of a modern computer. He just couldn't build it, because the technology to do that wasn't advanced enough at the time.

While we're on the subject of Babbage, it's worth noting that the mathematician daughter of the poet Byron, Ada Augusta, worked with him in the development of the analytical engine, and in the course of that project, she wrote what's been widely acknowledged to be the first programs—sets of instructions for the engine. A hundred and fifty years later, the US Defense Department named a programming language (ADA) in her honor.

The first actual (electric) computer was built at Iowa State University in the 1939, and used Boolean (T/F—on/off) algebra. It was named the ABC, after its developer, a Floridian named Atanasoff and his graduate student, Barry. The first **electronic digital** computer, the ENIAC, was developed by the research team of Mauchley and Eckert, as a result of work begun on a government grant made during World War Two, when the need for fast and accurate calculation in battle situations became apparent.

The Mark I, the first and only electro-mechanical computer, was introduced in 1944. It was an 8-foot by 8-foot concoction of vacuum tubes and relays, and it fascinated a young Naval Ordnance mathematician named Grace Murray Hopper, who became its third-ever programmer. –Ms. Hopper was eventually promoted to Rear Admiral because of her technological triumphs (one of the most important of which, for our purposes, was her involvement in the development of COBOL).
An interesting sideline to Ms. Hopper's story is that in the course of her work on the Mark 1, she is said to have discovered a moth, trapped in relay 7, which led her to coin the term 'debugging a computer'.

U.S. Naval Historical Center Photograph.

Now, the Mark 1 ‘s dimensions hardly describe it. It was truly a behemouth. It had 750,000 parts, 3 million electrical connections, and 500 miles of wire. Not only that, its top speed of 3-5 seconds per calculation turned out to be glacial compared with that of the next comer on the computer scene—the ENIAC, which could do in one minute what it had taken the Mark 1 three hours to accomplish. ENIAC, though, had one big disadvantage. It required physically rewiring thousands of connections in order to create new programs.

The Mark 1 and the ENIAC were pre-generational computers. Generation-one computers, the first of which, the UNIVAC, Grace Hopper was instrumental in the designing of, used huge arrays of vacuum tubes and relays. By 1956, though, transistors (which had been invented in the ‘40s) were the technological basis for supercomputers, both at the Lawrence Livermore Laboratories and at the Naval R&D Center in Washington, D.C. But transistors were not in general use until around1960, which is considered the bridge between the vacuum tube and the transistor eras.

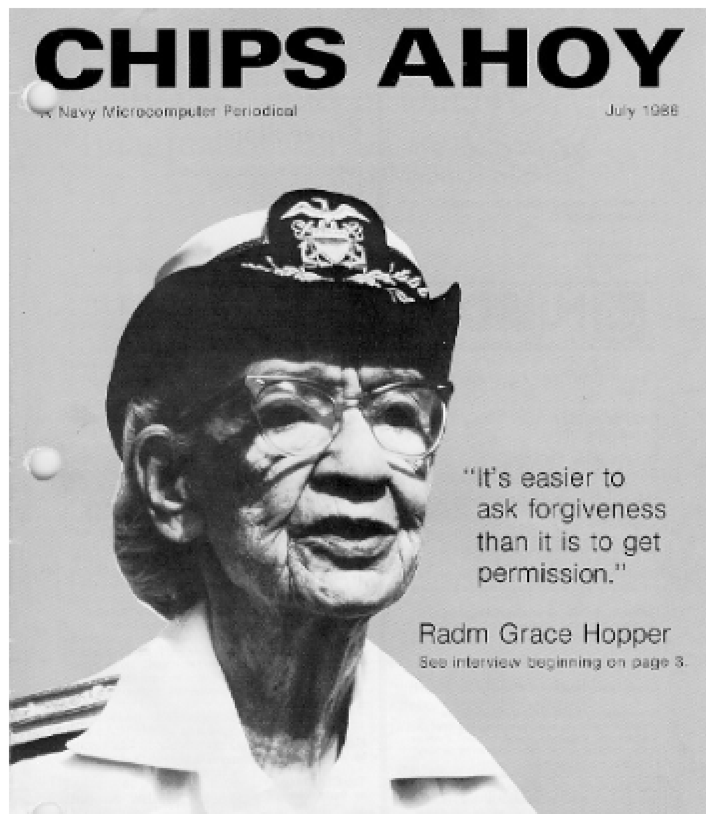
The second (transistor-based) generation of computers was characterized by the invention of Assembly language, which allowed abbreviated program codes to replace some binary instructions. The command you will use to execute your COBOL programs this semester is a holdover from the days of assembler languages – ‘a. out’ is shorthand for ‘assembler output’ and is still used today to execute a compiled program.

Finally, silicon chips using integrated circuits were invented, launching, around 1964, the third generation of machines. This breakthrough enabled the miniaturization that in turn made possible computers as we know them today. The first small computer, a minicomputer, was introduced by Digital Equipment Corporation in 1965. Around 1971, Intel began manufacturing even smaller, affordable microprocessors that for the first time, made it feasible to target computers to home users. The first mass produced personal computer was the Altair, brought out in `975 . It had 256 bytes of memory and no keyboard, monitor or hard drive, and never sold well. Until IBM introduced its version of the personal computer in 1981, though, computers were considered an electronics hobby. The affordability of the IBM PC led to a doubling in the number of users, to over 5 million, between 1981 and 1982. The graphical user interface (GUI) was introduced by Apple in 1984. A few years later, Apple came up with the first commercial mouse. And the rest, as they say, is history. This isn't complete information because this isn't a class about hardware. But a certain understanding of how computers developed will help you appreciate the power of COBOL, then and now.

From here on out we'll be talking about not about computers themselves (ie hardware) but rather about how computers get their instructions (ie software). The earliest, pre-generation one, computers didn't use operating systems. Rather, instructions were made-to-order for each specific task for which the computer was to be used. Each computer had its own binary-coded "machine language" that told it how to operate. This made computers difficult to program since specialized engineering skills were required which had to be relearned every time a programmer changed machines. To combat this obstacle, in 1950, RADM Hopper produced the first compiler, a program that translated symbolic mathematical code into machine code.

The first high-level (ie English-based) language that successfully used a compiler to translate the source code into an object program, was introduced around 1957. FORTRAN, short for Formula Translating System, was designed for the calculation of complex formulas and eventually replaced assembly language in the scientific and military communities.

The second high-level language to be developed was COBOL (Common Business-Oriented Language). In 1959, RADM Hopper and her staff produced the first COBOL specification based on the FLOW-MATIC language, which she had developed in order to teach certain English sentences to UNIVAC 1. An interesting footnote is that in 1966, she was forced into retirement due to her age; however, less than seven months later, after more than 800 unsuccessful attempts to develop a COBOL payroll system, she was called back to help and became the first Naval Reserve woman ever to return to active duty.



Permission requested for reproduction from US Navy publication CHIPS.
Photo available online: http://www.norfolk.navy.mil/chips/grace_hopper/86.gif

In 1968 the first American National Standard for COBOL was published and the federal government issued polices stating that no new computers would be purchased for governmental purposes unless a COBOL compiler conforming to the new standard was included.

It also mandated that all federal business applications were to be written in COBOL. Partly as a result of this policy, there are over 20 billion lines of COBOL code currently in use.

The 1974 COBOL standard introduced the concept of portability; now a COBOL program written on one machine could be recompiled on another with the same results on either machine. A third COBOL standard was introduced in 1985.

In 1986, at eighty years of age, RADM Hopper retired (again!) from the Navy. She remained active as a senior consultant to Digital Equipment Corporation until her death in 1992. Her legacy, COBOL is now in its fourth generation. COBOL 97 includes many object-oriented features; however, since the vast majority of COBOL in the business world is procedural and likely to stay that way, you will be using a unix-based version of the 1985 COBOL standard in this class.

Why learn COBOL at all??

COBOL is considered by many programmers to be a relic of the mainframe days and not very important in today's PC-oriented environment. Part of this prejudice can be traced to the early days of PC's, when mainframe environments tended to move forward at a snail's pace. There was also a snobbishness among the usually highly-experienced mainframe people in those days, that tended to push PC people into a renegade community, which took pride in its independence. This in turn caused PC programmers to ignore the advice and experience of the mainframe programmers, no matter how relevant.

PC programming gained a lot of strength between COBOL-74 and COBOL-85 because the evolution of COBOL had become as stagnant as the IS departments that used it. In this environment COBOL floundered, while PC languages sprouted like fungi. Some languages, like C, became very strong while others, such as Modula II or SmallTalk faded away.

A word now about C versus COBOL: Although the popularity of UNIX has made C the language of choice for many applications programmers, C lacks the standardization and portability that COBOL has developed in its 30+-year history. This limits C's ability to become the dominant programming language for large-scale business applications. Although COBOL was designed to address many of these issues, PC programmers have shown some unwillingness to learn from their mainframe brethren who have already traveled this path. The computing community is slowly beginning to realize that each side has something of value to offer the other, especially as environments become more integrated and applications are written to run on both mainframes and microcomputers.

The COBOL community learned from the controversy surrounding the process to adopt the COBOL-85 standard that it must move quickly if it were to address the current needs of the programming community. To that end, the International COBOL Committee created an addendum process that allowed new features to be added to the language every few years without causing incompatibilities with older COBOL programs. In 1989, new intrinsic functions such as ANNUITY, MIN(MAX), SUM, CURRENT-DATE (and many others) were added in order to speed up programming productivity. The COBOL-97 standard includes object-oriented features. Today, there are many COBOL compilers and tools available for the both the PC and mainframe environments.

The C community is learning that one of the most important attributes a program can have is maintainability. Although C can be concise and clever, it can also be very ambiguous and hard to understand. This is acknowledged within the C community with a yearly contest for the most obfuscated code [1]. This lack of clarity makes C programs expensive to maintain over a long life involving many maintenance programmers. It is important to note that one of RADM Hopper's primary goals when developing COBOL was to make computers and programming accessible to everyone, not just computer science majors. To that end, the verbosity of COBOL, which many C programmers criticize, makes it easy to read and understand, especially for the less technically inclined. This clarity is the single most important attribute of COBOL.

C does have its strengths; it has good run-time performance and uses less overhead than many high-level languages. It allows the programmer to have access to bit and byte level operations, addressing functions and dynamic memory assignments. C is useful for programs that require 'closeness' to the system, such as utilities and small applications that will not require much maintenance over their lifetimes. However, for long-lived applications that need to process large amounts of data, COBOL has superior file handling capabilities and is easier to maintain. C can only read byte stream files, whereas COBOL can read and process sequential, indexed and random files and relational databases. Also, because COBOL was developed to solve business problems, it has many built-in business functions such as PRESENT-VALUE that would require separate libraries in C.

[1] Here are some excerpts from the contest on a site maintained by Professor Manuel Enrique Bermudez: http://www.cise.ufl.edu/~manuel/obfuscate/obfuscate.html

The bottom line here is that COBOL generally has a lower cost-per-line of code than C, has greater portability and operating system independence and has a huge amount of support tools available - from CASE code generators and integrated debuggers to code analyzers. The fact is that whatever the academic opinion of the language, business likes COBOL and has been crying out for more and better-trained COBOL programmers. According to a recent ITWorld.com article, 75% to 80% of the world's business data is written in COBOL. Most companies are not willing to take the risk of moving their mission-critical data out of the COBOL environment and into a C/C++ or Java environment. This dependence on COBOL creates a steady demand for programmers that is expected to increase as many of the first-generation mainframe programmers reach retirement age. According to Paul Halpern, director of traditional development solutions at a Web-enabling training company in Mountain View, Calif., "If all the COBOL programs stopped working, the US economy would collapse."

Bibliography

"COBOL programmers back in the game." ITworld.com. 3/19/01. Available Online: http://www.itworld.com/Career/1944/ITW0319weinstein/ [Accessed 5/29/01].

MacDonald, Laurie. "Cobol Still The Major Language For Business Applications Programmers." JOURNAL OF INFORMATION SYSTEMS EDUCATION 3/89
Volume 1, Number 3 . Reprinted in Journal of IS Education Online. Available online: http://gise.org/JISE/Vol1-5/COBOL.htm [Accessed 5/29/01].

LaMorte, Christopher & John Lily. "Computers: History and Development." Jones Telecommunications & Multimedia Encyclopedia. Available Online: http://www.digitalcentury.com/encyclo/update/comp_hd.html. [Accessed 5/17/01].

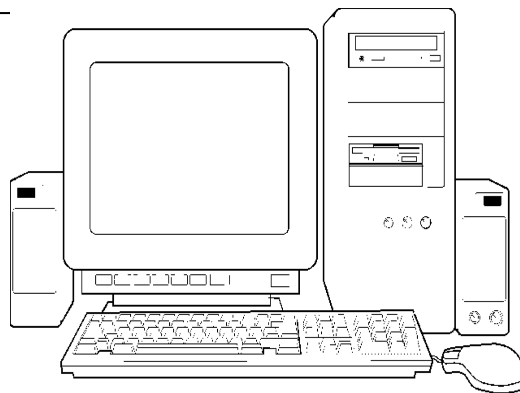
Norman, Rebecca. "Grace Murray Hopper." Available Online: http://www.agnesscott.edu/lriddle/women/hopper.htm. [Accessed 5/17/01].

"Hindu Arabic Numbers." Social And Historical Aspects Of Mathematics, Module Mm2217. Available Online: http://www.scit.wlv.ac.uk/university/scit/modules/mm2217/han.htm. [Accessed 5/17/01].

Lee, J.A.N., Stanley Winkler, Merlin Smith. "Key Events in the History of Computing." Prepared for the IEEE Computer Society For Distribution in 1996
As Part of the 50th Anniversary Activities. Available Online: http://ei.cs.vt.edu/~history/50th/30.minute.show.html [Accessed 5/16/01]

Danis, Sharron Ann. "Rear Admiral Grace Murray Hopper." Updated 2/16/97. Available Online: http://ei.cs.vt.edu/~history/Hopper.Danis.html. [Accessed 5/17/01].

Source Computer: Compilation is the job of the source computer. The compiler is a program that checks for *syntax and grammar errors* in a source program. The compiler reads from position 7 to position 72, and writes the program together with any errors it finds to a file stored using the program name and the 'lis' file type, e.g. 'example.lis'



Source Program: A program written in a high-level language that is the original submission to the compiler on the source computer. For the unix COBOL compiler, the source program name is a user-chosen word followed by a file type. 'cob' is the file type of the source program, e.g. example.cob.

When all of the errors in a source program have been fixed, it is translated into an **object program**, which is in machine language (binary) and ready for the **Object computer** whose job is to execute the program (following all its commands). This is when you find out if there are *logic errors*. 'data' is the file type of the output of an executed program on the object computer, e.g. example.data.

HOW COMPUTERS READ YOUR PROGRAM (IN A NUTSHELL)

In other words, there are two Steps:

1. Compilation: Source Program → Source Computer
(The outcome of the compilation step is the object program)

2. Execution: Object Program → Object Computer
(The outcome of the execution step is an output report, file, etc)

Writing Programs In A Business Setting:

1. The job spec

Overview: describes input, processing, and output:

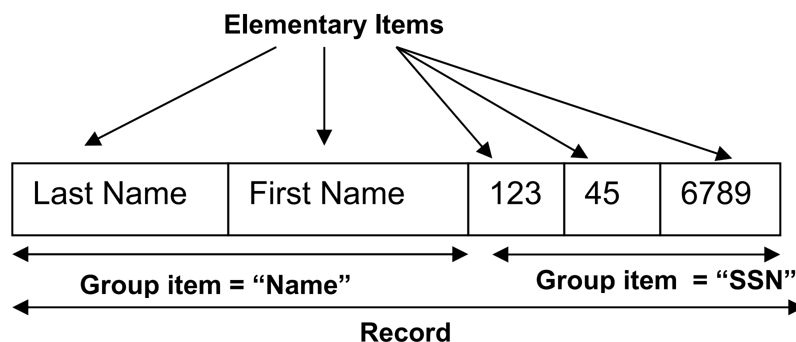
✓ **Input** – Information that comes from outside of the program. Input can be read from a file or provided by the user with an input device, like a keyboard. Each item within a file must be described precisely so the computer knows how many positions to allow for that item. If the items are in files, you will need to know:

- » The input file names
- » How the items are organized in the input files
- » Where the input files are located

Terms Used for Input:

- » **Byte** = 1 position
- » **Item** = definable string of bytes (also called **field**)
  (items/fields are of fixed widths)
  Group items have subdivisions
  Elementary items are not subdivided
- » **Record** = group of formatted items
- » **File** = group of formatted records

Elementary versus Group items:



✓ **Processing –** Performs one or more algorithms on the input data in order to generate the output. Processing, in the case of structured programs, consists of some combination of sequencing, selection and/or repetition

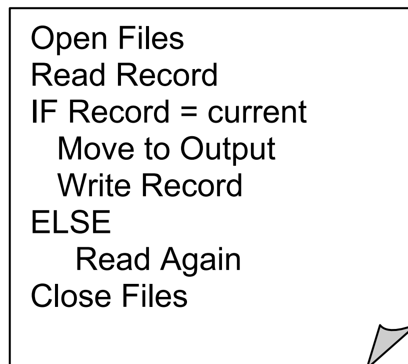
✓ **Output –** The final result of the data manipulations, often presented in a human-friendly format. It is not always necessary to use every piece of input data when generating the output. You will need to know what output is needed and where it is going to be stored. Output is represented using print charts , record layout forms, or screen diagrams.

2. Finding a Solution to the Problem

Step 1: Come up with a schematic solution and "walk" data through it

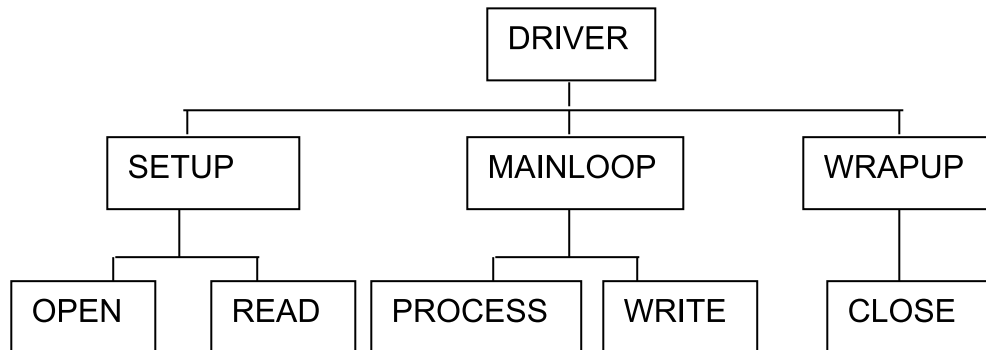
This is done using various diagramming and planning tools including:

✓ **Pseudo-code:** Describing the program in English without using the formal syntax of a particular language:



✓ **Flowcharts:** A chart that uses different shapes to symbolize the logic of the program. Order, frequency and logic are shown. The symbols are shown on page 35.

✓ **Hierarchy charts:** A chart that divides the program into functional components and then divides each component into individual tasks. The chart does not show the order or frequency of the functions or the logic of the processing.



Step 2: Code the program in whatever high-level language you're using

3. Testing the Program (on the computer)

Run the coded program using phony **test data** and checking the resulting output against the known solution. If the results are not what was expected, something is wrong with the logic of the program. Once the program runs correctly with the original test data (which is error-free), errors are intentionally put into it and the program is run again. Using test data with mistakes in it allows programmers to be sure they have planned correctly for any data entry or manipulation errors that might occur. Checking for division by zero is a common example of **error checking.**

Words of wisdom: NEVER assume a program runs correctly just because it compiles.

4. Documentation

This is CRITICAL! Programmers should include **comments** that describe how the program works and any assumptions made about the data. Whenever the program is changed or updated, the programmer should include additional comments with the date and a description of the modification. Comments make it easier for others to understand your program and your logic.

The COBOL Language

- ✓ CO**mmon** B**usiness** O**riented** L**anguage**
- ✓ Devised specifically for business and is completely standardized
- ✓ Deals with large amounts of data in file format and simple math:
  → addition, subtraction, multiplication, division, exponentiation
- ✓ Cannot write an operating system in COBOL

The COBOL character set:

- ✓ 51 characters, which consist of:
  - » Letters
  - » Digits
  - » Punctuation
    - ✓ The comma is a 'noise' word that the compiler ignores.
    - ✓ The period means STOP!
- ✓ Special characters include math and logical operators, and characters for editing output

The Elements of COBOL

COBOL was designed to be as English-like as possible. Programs are organized into sections, paragraphs, sentences, statements, and clauses. COBOL is a verbose language that can be read like a manual. The elements consist of:

1. Reserved words: Words that have a special meaning to COBOL. Reserved words can only be used in specific ways. Correct spelling and hyphenation is critical – 'end read' and 'end-read' are NOT the same! Reserved words include:

- » The four division names followed by a period
- » The section names for the first three divisions
- » **Verbs** – SELECT, ASSIGN, etc….
- » **Figurative constants** such as SPACE or ZEROS

2. Programmer-Supplied Names (PSNs): A name that the programmer makes up. The names of items, records, files and paragraphs in the PROCEDURE division are programmer-supplied. In other languages, these are called variable names.

- » PSN's may only contain letters, digits, and hyphens
- » Program ID may NOT contain hyphens, only letters and digits
- » 30 character limit
- » Cannot start or end word with hyphen
- » All PSN's must contain at least one letter EXCEPT paragraph titles in the Procedure division. Paragraphs can have numeric titles
- » NO reserved words or spaces

A style note for making up PSNs: Names given to data items in WORKING-STORAGE must EXACTLY match names used the PROCEDURE division in both spelling and hyphenation. For example, CR-TOTAL and CR-TOTALS would not be considered interchangeable or equivalent. Choose PSN's that describe the function of the data element you are naming - try to avoid ambiguous or generic data names. Names like HEADING1 or TOTAL are not very meaningful or descriptive – names like DATE-HEADING or CUSTOMER-TOTAL give a lot more information about the function of the data.

3. **Symbols:** Punctuation, editing characters, math and logical operators

4. **Literals:** An item with a specified value. Also called a constant.

- **Non-numeric Literal:** Any character data, including spaces, enclosed within single (‘) quotes. Called a String Literal in other languages. Anything is legal in a literal EXCEPT the single quote
  - ✓ Up to 120 characters
  - ✓ Signaled by the keyword VALUE
  - ✓ Cannot be used for math
    - ADD 5 to X - YES!
    - ADD ‘5’ to X – NO!

- **Numeric Literal:**
  - ✓ 18 digit limit
  - ✓ Signs represented by S
  - ✓ Decimal represented by V
  - ✓ Digits represented by 9
  - ✓ MUST be used for math

5. **Level numbers:** Defines the order of items and its relationship to other items

- ✓ Every item description in the data division MUST have a level number
- ✓ Hierarchal level number are 01 to 49
- ✓ 01 can be an elementary item or can be the highest level in a data group (record)
- ✓ Use 01,05,10,15,20 etc – this leaves room to insert additional levels in between when needed
- ✓ 66, 77, and 88 are special level numbers that are used for different types of data definitions
  - Level 66 is used with the verb RENAMES to create group items from one or more fields in a record.
  - Level 88 is used to create programmer-defined conditions, such as switches which can be set to Y or N

Example of Hierarchal Level Numbers:

01 (group)
    05 (group)
        10 (elementary)
        10 (group)
            15 (elementary)
            15 (elementary)
            15 (elementary)
    05 (elementary)

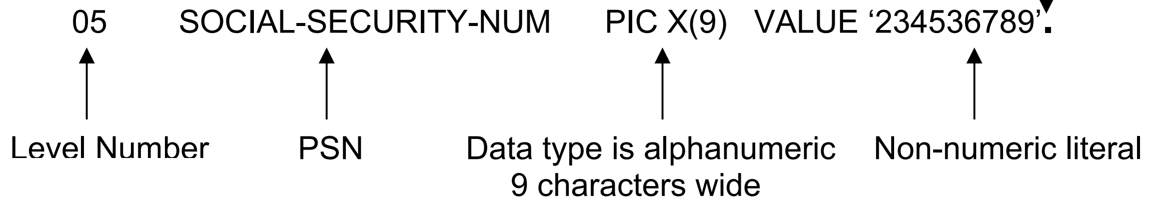
6. **Picture Clauses:** Describes the type and size for each data element. Use the reserved word **PIC** to define each data element. Only elementary items have PIC clauses – group items **may not** have a PIC clause.

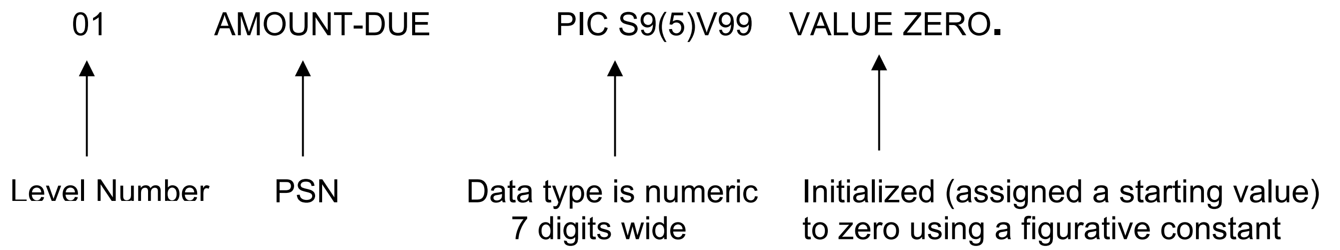
Item Type	PIC	Data
Alphanumeric	X	Any data type
Alphabetic	A	Letters and spaces
Numeric	S9V	9= Digits (ONLY) S = read a sign V = read a decimal point

Avoid data-type errors by declaring all data as alphanumeric, unless there is some over-riding reason for not doing so. Phone numbers, zips, SSN should all be set up as PIC X. Declare numbers as signed to avoid errors. Any data you will use in math operations MUST be declared as the numeric type.

Don't forget the period!

Data Description examples:





HOW A COBOL PROGRAM IS ENTERED INTO THE COMPUTER (ANS)

The columnar format defined by the **A**merican **N**ational **S**tandards **I**nstitute

American National Standard Format:

» Columns 1-6 are not used.

» Column 7 is the **continuation column** or **indicator area** – used ONLY for comments or to continue a non-numeric literal. Only the hyphen (-) or asterisk (*) may appear in column 7.

✓ Columns 8-11 are called **Area A**. Items that go ONLY in Area A :

- » Division names
- » Section names
- » Paragraph names
- » 01
- » FD

✓ Columns 12-72 are **Area B** - everything else goes here. Area B is read by punctuation - for example a period always means 'STOP'. Style makes the program easier to read for humans - the compiler does not care.

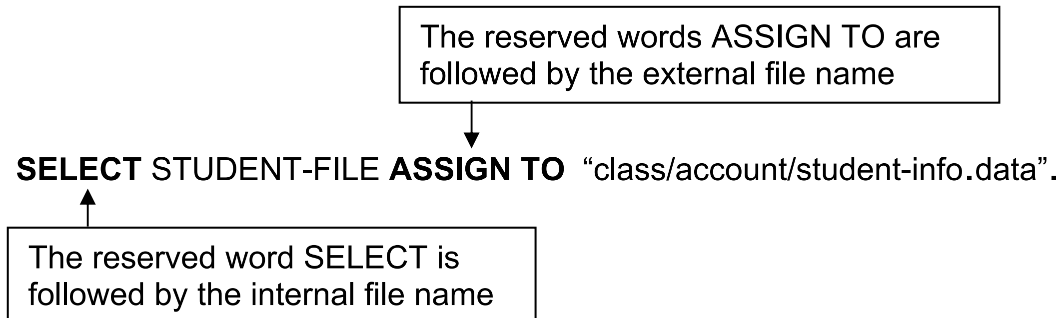
✓ The compiler ignores anything beyond column 72.

Divisions of a COBOL program

Every COBOL program consists of four divisions, which must appear in order.

1. **IDENTIFICATION DIVISION:** Must contain the reserved words **PROGRAM-ID** and the system name of the program. All other entries in this division are optional and treated as comments by the compiler.

2. **ENVIRONMENT DIVISION:** Contains the **INPUT-OUTPUT SECTION**, which describes all of the files the program will use. This section is the interface between COBOL and the operating system. In this section, under the heading **FILE-CONTROL**, a file is given two names – an internal name that you will use to refer to the file in the program and an external name that the operating system will use to locate and manipulate the file. Here is an example of how this works, followed by an explanation of why it is necessary



When you want to use this file in the program, you will refer to it as STUDENT-FILE. If you want to look for it in your class account directory, you will look for student-info.data. The reason for this two-name scheme is to provide for error-free portability among different types of operating systems. Different operating systems have different rules for naming files and this two-name scheme reduces the chance of errors if the external file name needs to be changed for some reason. For instance, notice that the name student-info.data is 12 characters long. Most operating systems do not allow file names to be longer than eight characters. To run this program on one of these systems, you would only need to change the name once, after the ASSIGN TO clause, rather than in every place you refer to the file in the program. The new assignment clause might look like:

```
SELECT STUDENT-FILE ASSIGN TO “std-inf.data”.
```

Notice that only the external file name is changed in this example, from student-info.data to std-inf.data. This is the only change that is made to the program. The file is still called STUDENT-FILE within the program.

The order in which internal file names are used in a program:

SELECT → **FD** → **OPEN** → **READ** → **CLOSE**

If you are using a UNIX operating system, ORGANIZATION IS LINE SEQUENTIAL is required for output files.

```
SELECT PRINT-FILE ASSIGN TO “std-inf.data”
ORGANIZATION IS LINE SEQUENTIAL.
```

Don’t forget the period!

3. **DATA DIVISION:** Describes in detail all of the files, records, and items the program will use. It has two sections:

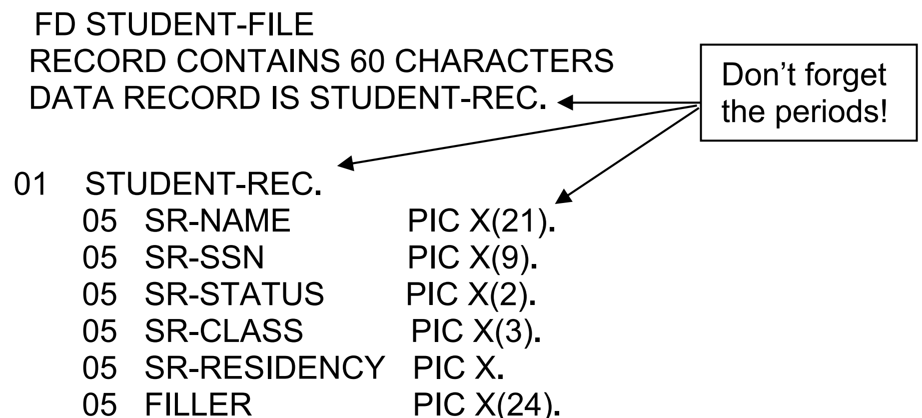
   a. **FILE SECTION** describes all of the input and output files that were defined in the INPUT-OUTPUT SECTION (in the SELECT statement). Each file must have an FD (**F**ile **D**escription) followed by a **record layout** that defines each item in the record using PIC clauses and level numbers.

   b. **WORKING-STORAGE** describes all of the data that are used in the program, but are not part of the input or output files, such as the end-of-file (EOF) condition, report headings and other temporary items, such as those used to calculate totals.

FDs and record layouts: The internal filename should be followed by the **RECORD CONTAINS** clause, which gives the total number of characters in the record. This clause isn't required but it's useful because it causes COBOL to add the PIC clauses in the record layout and generate an error if they do not sum to the number of characters declared in the RECORD CONTAINS clause.

An FD MUST be followed by an 01 level identifier that describes the contents and layout of the file. Both input and output files must be described this way. If an output record is to be printed, the first position in its record must be left blank to allow for the carriage return, otherwise the first character of each printed line will be cut off.

***RECORD CONTAINS EXAMPLE*:**



When a record is read from the STUDENT-FILE, it is placed into STUDENT-REC. To use the record, you must MOVE it from STUDENT-REC **NOT** from STUDENT-FILE. Each READ off the STUDENT-FILE will overwrite the previous contents of STUDENT-REC.

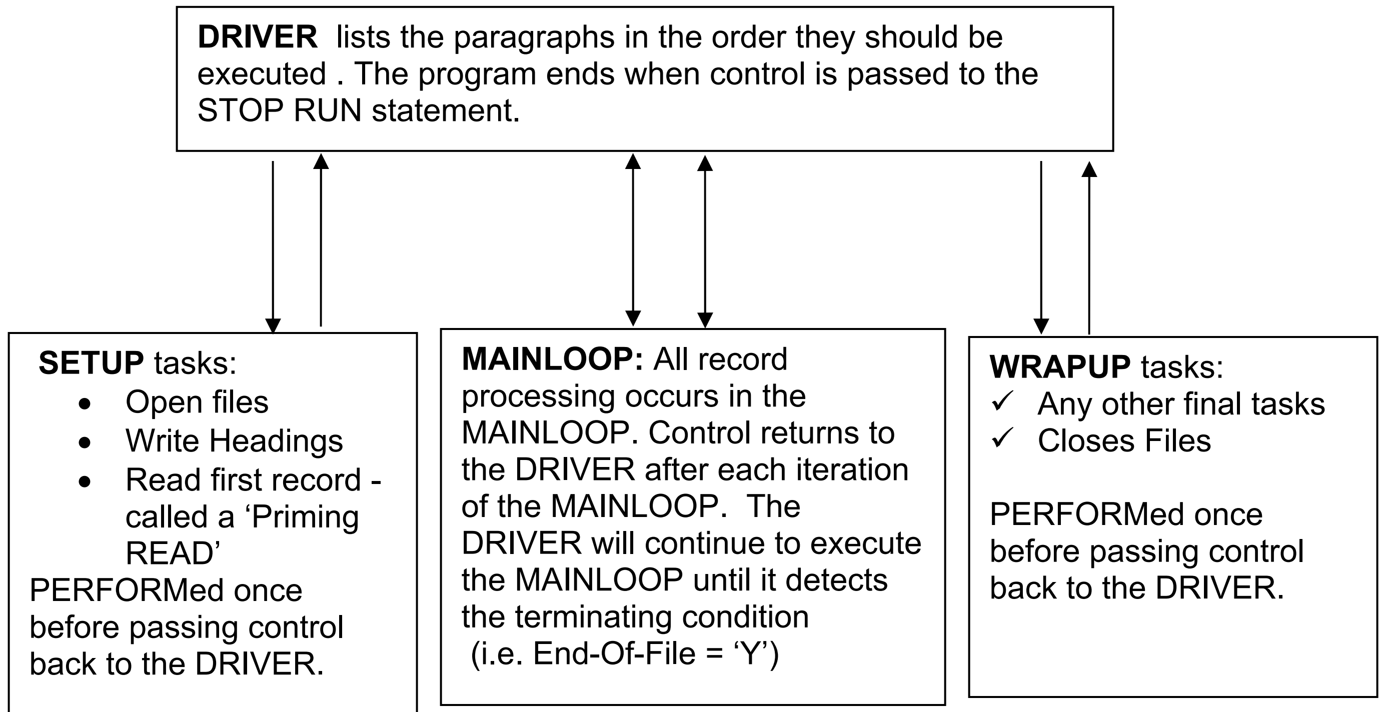
The optional reserved word FILLER is a placeholder used with data that do not need to be assigned PSN's for some reason. It doesn't mean the area is empty, just that it does not need to be referred to during the program. Many FILLERs, for instance, have non-numeric literals for content. The word FILLER may be omitted for COBOL-85 or later compilers.

The END-OF-FILE condition: For files to be read sequentially**,** an End-Of-File item, or EOF is usually specified in WORKING-STORAGE. The EOF condition is set to false in WORKING-STORAGE . Later in the program, when the file runs out of records, it will be set to true.

4. PROCEDURE DIVISION: Contains the logic that controls the program and produces the output. The PROCEDURE DIVISION will contain paragraphs for each action (or 'function') the program will perform. This is the only division where the programmer supplies all of the paragraph names.

Coding the Procedure Division

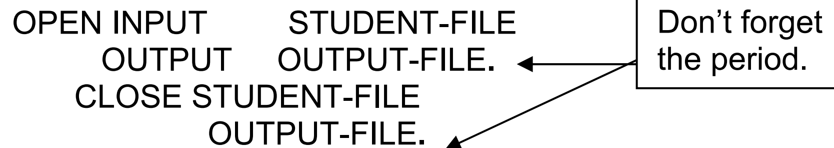
Using a top-down structure: The Driver paragraph: The DRIVER paragraph delegates the program's jobs among the paragraphs that follow it. The DRIVER paragraph is always in control; every paragraph returns to this DRIVER when it has completed its task.



The Verbs used in the Example Program

OPEN – CLOSE: The verb OPEN tells COBOL the file exists, and matches its FD specs. The program will abort if the input file does not exist. The program will create output files if they do not already exist, and will overwrite any existing output file with the same name. Input files MUST match their FDs exactly or program will not run. The OPEN verb must be followed by either INPUT or OUTPUT and the internal filename. The CLOSE verb requires only the internal file name.

OPEN - CLOSE Syntax Examples:



READ – WRITE:

READ retrieves data from an input file finding the data by looking at the internal file name back in the FD. A file to be READ must first have been OPENed INPUT.

The reserved words AT END are required for sequential processing and tell COBOL what to do when the End-Of-File (EOF) condition is detected. This condition is encountered only after COBOL has read the last record in the file and is trying to read again.The use of END-READ is optional unless the NOT AT END condition (which says what you want to do if COBOL *does* find a record) is present; then, END-READ is required.

There are two READ methods:

1. **Sequential Read:** Reads each record in order;cannot go back to a previous record.

2. **Direct Access Read**: Reads records by using a record key and index to identify the location of each record. Reads only the specified record on each pass and can read records in any order. This is not covered in this class.

Sequential READ Syntax Example:

```
READ STUDENT-FILE
    AT END
        MOVE 'Y' TO EOF
END-READ.
```


The Priming Read

A **priming read** is used to read the file once (usually from the SETUP paragraph) before the MAINLOOP is PERFORMed. This means that when you enter the MAINLOOP, a record has already been placed in the input buffer and is ready for processing, so MAINLOOP starts in immediately with that assumption.

WRITE sends one line of data to a file which has been OPENed as OUTPUT. If the output is to be printed, the data will be moved to the output record from a line layout described in WORKING-STORAGE. If the line is not completely defined with FILLERs, then data must be MOVEd into it before the line can be moved to the output record for writing.

An example; suppose there is a line of data in the STUDENT-FILE that you want to dump into the output file. *(Dumping* a record means to print it out without adding any formatting, such as hyphens between the parts of the social security number).

Assume that STUDENT-FILE has already been READ and that you have created a record layout format in WORKING-STORAGE named DUMPED-REC-SLOT that can hold 60 alphanumeric characters – i.e. PIC X(60). To write a line, you will first move the data from STUDENT-REC to DUMPED-REC-SLOT:

Example of Getting Ready to Write:

```
MOVE STUDENT-REC TO DUMPED-REC-SLOT.
```


Then, MOVE the record layout DUMPED-RED-SLOT to the output record and WRITE the record:

```
MOVE DUMPED-REC-SLOT TO OUTPUT-REC.
WRITE OUTPUT-REC.
```

MOVE:

- ✓ MOVE actually copies data (COPY is a reserved word used for fetching definitions such as FD's from a library and bringing them into the program)
- ✓ Can MOVE multiple items with one move statement, as follows:
  - » MOVE A TO B,C,D
- ✓ If A and B are elementary items, the move is an elementary move statement
- ✓ If A or B is a group item, the move is a group move statement
- ✓ How A is moved into B depends on B's (the receiving item) PIC clause.
- ✓ Data are moved left to right if B is PIC X or A. Extra positions in B are padded with blanks. If B is too small for A's PIC clause, extra positions are simply left off.
- ✓ If B is PIC 9, data are truncated or padded with zeros at the "outside edges" which avoids changing the value of the number. The padding moves away from the decimal.

.

MOVE Examples:

SENDING		RECEIVING	
PIC	VALUE	PIC	VALUE
X(3)	123	X(5)	123bb
X(3)	123	S9(5)	00123
S9(3)V99	12345	S9(5)	00123

PERFORM - GO TO

PERFORM

PERFORM is the only iteration structure in COBOL. The syntax of a simple PERFORM is:

```
PERFORM paragraph-title.
```

Which causes the program to execute the instructions in the paragraph (ONCE) whose name follows the **PERFORM** verb, then return to the sentence following the PERFORM statement to see what it should do next.

If you want the paragraph to be PERFORMed multiple times, you can use UNTIL:

```
PERFORM paragraph-title UNTIL condition.
```

In this case, COBOL checks the condition before it performs the paragraph. If the condition is true already, then it won't do the paragraph at all but just go to the next sentence.

GO TO

The syntax is similar to PERFORM:

```
GO TO paragraph-title.
```

Except that when the program is sent somewhere by the **GO TO** verb, it does NOT return to the statement following the GO TO, but continues the from where ever it was sent.

In other words, PERFORM returns, but GO TO does not. GO TO is generally considered a dangerous programming construct and should be used only in certain specific circumstances - such as to move around within performed procedures.

PERFORM-GO TO Syntax Examples:

```
PERFORM 500-WRITE-HEADINGS.
PERFORM 400-MAINLOOP UNTIL EOF = 'Y' .
```

Don't forget the periods!

In this example, the PERFORM verb will cause the program to follow each instruction in the 500-WRITE-HEADINGS paragraph, stopping when it encounters a new paragraph title in Area A. The 500-WRITE-HEADINGS sentence will be executed only once because it does not contain the reserved word UNTIL followed by a condition. The program sees that it is finished with 500-WRITE-HEADINGS because it has encountered a new paragraph and proceeds to the next sentence, which is PERFORM 400-MAINLOOP. The condition following UNTIL will be checked, and if it is false, every statement in 400-MAINLOOP will be executed until a new paragraph title is encountered in Area A. The program will then return to the sentence to check the EOF condition again.
As long as EOF = 'Y' continues to be false, 400-MAINLOOP will be executed, and the condition will be re-checked. PERFORM UNTIL is equivalent to using WHILE in other languages, and should be used when you do not know ahead of time how many times to repeat an action.

Be careful! When you control execution using UNTIL, you must provide some way for the condition to become true or you will end up in an infinite loop and your program will never stop running!! If this happens to you in the lab, press the control [CTRL] key and the 'c' key at the same time and the program will quit. Compare the example above with this example using GO TO:

```
PERFORM 500-WRITE-HEADINGS.
GO TO 400-MAINLOOP UNTIL EOF = 'Y' .
```

As above, the program will follow all of the instructions in 500-WRITE-HEADINGS and return to the next sentence. However, because GO TO is used to send the program to 400-MAINLOOP, the program will never return to the sentence to check if EOF = 'Y'. The verb GO TO will cause the program to follow all of the instructions in 400-MAINLOOP once and then to continue program execution with whatever paragraph it encounters next.

Normally, when UNTIL is used with PERFORM, the condition is checked before any statements are executed. If the condition, such as EOF, is found to be true, the statements will never be PERFORMed.

You can change this. If you use WITH TEST AFTER with UNTIL the statement(s) will be executed *before* the condition is checked. This is often used with a decrementing counter. In this example, the counter will be decremented inside the 600-CALCULATE-TOTAL paragraph.

WITH TEST AFTER Example:

```
MOVE 20 TO COUNTER.
PERFORM  600-CALCULATE-TOTAL WITH TEST AFTER
                                        UNTIL COUNTER = 0.
```

If you know the exact number of iterations a loop requires, you can use TIMES with PERFORM. Either a whole number numeric literal or a numeric PSN may follow TIMES. If the value of the PSN is zero or negative, the statements following PERFORM will not be executed. If you use a simple PERFORM (no UNTIL), the value following TIMES is checked only once, at the beginning of the loop, and never checked again so you can't change it midstream.

TIMES Example:

```
PERFORM 600-CALCULATE-TOTAL 20 TIMES.
```

PERFORM can be used to execute statements within the same paragraph, rather than sending the program to a different paragraph. This is called an *in-line* PERFORM and requires the use of the END-PERFORM phrase. TIMES, UNTIL and WITH TEST AFTER may be used with in-lline PERFORMs.

In-line PERFORM Example:

```
PERFORM
      DISPLAY   'Press Any Key To Continue'  3 TIMES
END-PERFORM.
```

PERFORM THRU allows control of the program to be passed to a block of code that can contain several paragraphs or an entire section. The program will not return to the statement following the PERFORM statement until every line of code in the specified block has been executed.

PERFORM THRU Example:

```
PERFORM 600-CALCULATE-TOTAL THRU 900-DAILY-REPORT.
```

STOP

STOP RUN ends the execution of the program ends and passes control back to the operating system.

Data Description (Files and Items)

Files

More about FD's**:** Several types of information about the records in an input file and how they are to be read can be placed in the File Description. These statements are optional, but if used, must follow the FD with a period after the last statement. The DATA RECORD statement must be last and followed by a description of the record layout at the 01 level. FD information can include:

- ✓ **RECORDING MODE IS F** is used with files that have sequential records and refers to record's lengths relative to each other. **F** means fixed-length records; **V** means variable length records.

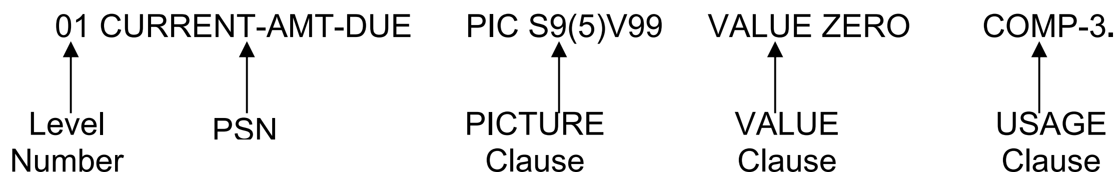
- ✓ **LABEL RECORDS ARE STANDARD** is an IBM clause used to check the label records at the beginning and end of a file. The label record contains information about the file and ensures the proper file is being processed. **STANDARD** means the label is IBM compatible, **OMITTED** means no labels or non-standard labels.

- ✓ **RECORD CONTAINS ___ TO ___ CHARACTERS** allows for a variable number of characters to be present in the record without generating compiler errors.

- ✓ **BLOCK ___ RECORDS** allows a specified number of records to be fetched every time the file is read. The default, when there is no blocking clause, is one record per READ. Reading is an expensive operation - meaning it is time-and-processor intensive. Blocking records reduces the number of times the READ must be executed on a file. If a file contains 10 records and blocks of 5 are specified, the file will only be read twice. This may not seem important when you are working with small files, however, business and government programs may require that thousands of records be read. Records cannot be blocked in this way for writing since only one record can be written at a time. In other words, the blocksize for output records is always one, which is also the default so it's usually not specified.

Items

A Full Item Description



Value clauses:

The format of the **VALUE** clause is**:**

… VALUE [ALL] [literal]
[Figurative constant]

Figurative Constants are a subclass of reserved words. Singular and plural forms are interchangeable – i.e. ZERO and ZEROS are equivalent. Below are the main figurative constants in COBOL:

- ✓ **SPACES**
- ✓ **ZEROS** is only figurative constant that can be used with numeric data
  - » VALUE ZEROS is used to give an initial value (or 'initialize') data
    elements that will be used for math
  - » May not be used in FILE SECTION, only WORKING-STORAGE
  - » VALUE and PIC must be compatible
  - » PIC X(5) VALUE ZEROS is a legal statement – X can hold anything
  - » PIC 9(5) VALUE SPACES is an illegal statement – 9 can only hold numeric data

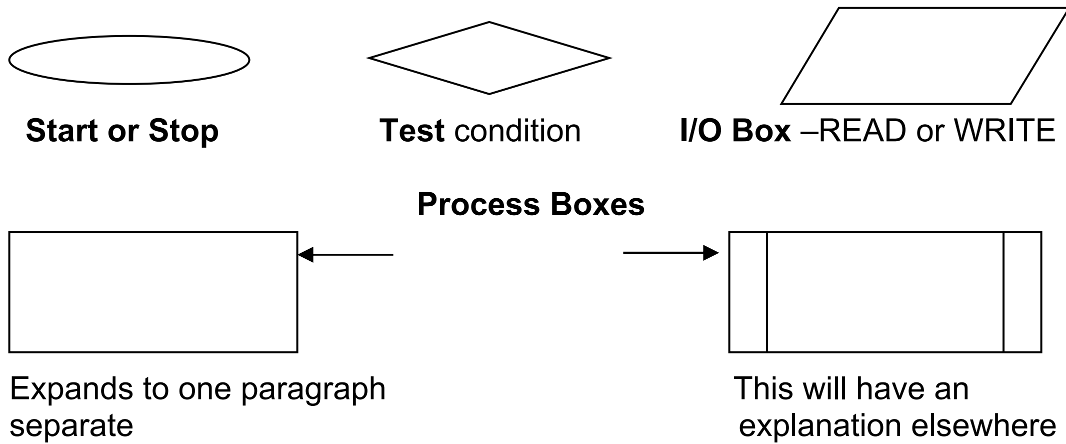
- ✓ **HIGH-VALUES** is the highest valued alphanumeric character in the character set – no other value can be higher. Used for comparisons.
- ✓ **LOW-VALUES** is the lowest valued alphanumeric character in the character set – no other value can be lower. Used for comparisons.

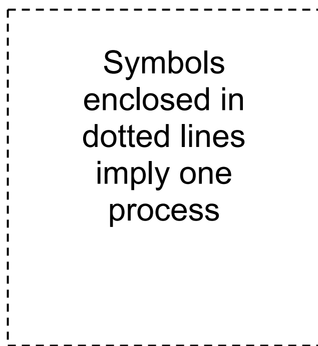
Some more options available with VALUE clauses:

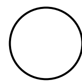
- ✓ **QUOTES** specifies the single quote character in a non-numeric literal – 'QUOTE Hello QUOTE' would print as 'Hello'
- ✓ **ALL** repeats the character as many times as specified in the PIC clause – PIC X(5) VALUE ALL '*" will print *****.

Usage Clauses: The USAGE clause dictates the way the PSN value is stored in memory. If not stated, USAGE reverts to the default, DISPLAY, which stores data as the character data type using 1 character per byte. The default storage for a clause declared as PIC S9(3)V99 would be 5 bytes of storage. The clause **USAGE IS COMP-3** forces the binary storage of digits. Each COMP-3 digit uses ½ byte and ½ byte is allowed for the sign. COMP-3 is used only with numeric picture clauses that will participate in math operations. COBOL will automatically convert data elements in a math expression to COMP-3 before performing calculations; declaring the item as COMP-3 in WORKING-STORAGE eliminates the need for COBOL to perform the conversion every time it works with the data, thereby saving processing time.

Flow Charting Symbols (needed to read assignments)



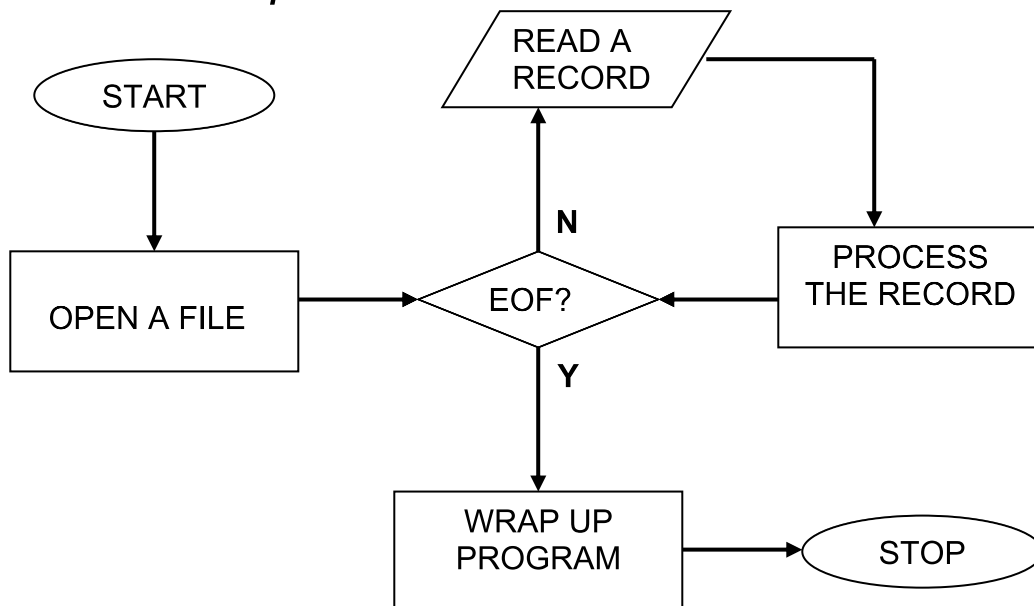




Collector circle - indicates where to follow a branch that is on more than one page. There will be a letter or number inside the circle at the point where the branch leaves one page and at the point where it continues on another.

.

***Flow Chart Example*:**



Testing in COBOL: IF - ELSE and EVALUATE - WHEN

The IF statement: Evaluates the truth of the expression that follows it. If the statement is true, the program performs one or more specified actions. If the statement is false, the program moves to the sentence that follows the IF statement.

- ✓ **Simple vs. Nested Ifs:** A **simple IF** has only one IF before the period; more than one IF before the period is a **nested IF**. Pair the **ELSE** with closest unpaired IF. Read from bottom of loop to top in order to correctly match IF and ELSE.

Nested IF Examples:

```
IF X IS NUMERIC
   IF X IS POSITVE
      PERFORM X-PLUS
ELSE
   IF X IS NEGATIVE
           PERFORM X-MINUS
ELSE
           PERFORM INVALID-DATA.
```

If you did not read this carefully and match the IF - ELSE from the bottom up, you might think that if X is not numeric, it is an invalid data. The first IF has no match – if X is not numeric, the program will not examine any other statement in the IF block. Instead, it will continue the program with the sentence that follows this IF statement.

```
IF X IS NUMERIC
   IF X  IS POSITIVE
           PERFORM X-PLUS
    ELSE
         IF X IS NEGATIVE
             PERFORM X-MINUS
        ELSE
             PERFORM INVALID-DATA.
             PERFORM GET-NEW-X.
```

Indentation does not matter to the compiler, but it does matter to the reader of your program. IF blocks should always be indented so that the IF-ELSE conditions that belong together are lined up. The actions associated with a true condition should also be indented so that it is clear to the reader what happens when a condition is true.

The poor indentation of this sentence does not make it clear that it is PERFORMed on every pass through the loop.

Four Types of Tests used with the IF statement:

1.The Relation Test: Words and symbols that describe relationships may be used interchangeably; **EQUAL TO** is equivalent to **=**, **LESS THAN** is equivalent to **<** and **GREATER THAN** is equivalent to **>**. **NOT** does not have a symbolic equivalent in COBOL. Items are compared from left to right, one position at a time.

IF PSN [IS] [NOT]

EQUAL [TO]	=
GREATER THAN	>
LESS THAN	<

These are the rules COBOL follows when comparing A to B:

- ✓ If both A and B are numeric (9), the decimal points will be aligned and they will be compared numerically. Unsigned numbers are treated as positive.

- ✓ If either or both are non-numeric (X or A), they will be compared using the **collating sequence**, which is the ranking of characters within the COBOL character set. If one data element is X and the other is A, both will be converted to X before comparison. If A and B are not the same size, the smaller item will be padded on the right with blanks to make it the same size as the larger item before the comparison is executed.

EBCDIC Collating Sequence

Highest ←———————————————— **Lowest**

9 0 Z A z a " = ' @ # : ? > _ % < / - ¬ ;) * $! & | + (< > ¢ space

The collating sequence determines the order in which character values are sorted and compared. IBM mainframes use EBCDIC character set while PC's use the ASCII character set. PC's and mainframes will sort and compare differently.

This process can lead to strange results or How 572 can be greater than 1238!

Compare: PIC X(3) VALUE ‘572’ to PIC 9(4) VALUE 1238

1. First, the program pads the smaller item (spaces for X or zeros for 9)
2. Next, 562blank is compared to 1238, position by position from left to right
3. The program stops the comparison when it finds one character larger then another. This happens as soon as it compares the 1 to the 5….the result? 562 > 1238!!

If both items were declared as PIC 9, then the padding would be done, not on the right, but away from the decimal point, so comparison would be: 0562 to 1238, which would produce the correct outcome.

2.The class test: Used to determine if data is **NUMERIC** or **ALPHABETIC**. The numeric test must be used to check any data element that will participate in a math operation to ensure that it contains only digits. If the program attempts to do math with non-numeric data, the program will quit prematurely and without warning. Failing to check data for validity before executing math operations is the main reason for abnormal program endings (ABEND).

```
IF PSN [IS] [NOT]  | NUMERIC    |
                   | ALPHABETIC |
```

The **NUMERIC** test is legal for X or 9, not A. Will evaluate to true if: digits, sign, decimal point

The **ALPHABETIC** test is legal for X or A, not 9. Will evaluate to true if: letters or spaces but not characters

3.The sign test: Checks for **POSITIVE**, **NEGATIVE** or **ZERO** value data. If data tested for the sign condition of positive has a value of zero, the expression will be true. Used to check numeric (9) data, especially for division by zero.

```
IF  PSN [IS][NOT]    POSITIVE
                     NEGATIVE
                     ZERO
```

4.Compound relation testing using AND-OR: These are Boolean logical operators that examine the truth-value of the expressions that follow them. When using AND**,** both conditions must be true for a true result. When using **OR**, both conditions must be false for a false result. Relational NOT is used with the relational operators LESS THAN, GREATER THAN or EQUAL TO (or their symbolic representations). OR and NOT can never be used in same statement because the statement will never become false. For example, if you tested for IF NOT FROGS OR TOADS, you would never reach a terminating condition because anything that is NOT FROGS is TOADS and conversely, anything that is TOADS is NOT FROGS - there is no way for this condition to become false. In a program, this would cause an infinite loop because the program would not be able to escape from this condition.

How expressions are evaluated: Compound conditions are evaluated in a hierarchal order from left to right. The hierarchy is:

1. Math expressions
2. Relational operators
3. NOT
4. AND
5. OR

Statements may be compressed, but clarity can be lost as a result. Write your statements so that your intention is clear, even if it requires a few more keystrokes! COBOL sentences can be diagrammed into subject, verb and object(s) to determine how the program will interpret a compressed expression.

Identifying the parts of a COBOL sentence:

IF X = Y AND Z

The SUBJECT of this expression is X: the VERB is EQUAL TO and the OBJECTS are Y and Z. These are some of the rules COBOL uses to evaluate compressed compound expressions:

1. If the subject is missing, COBOL matches it to the last subject it saw – in the statement above, IF X = Y AND IF X = Z is the result. Since this is an AND, both conditions must be true in order for the statement to evaluate as true.
2. Abbreviations are terminated if a parenthesis is encountered. In other words, an expression like: IF A = B OR C AND (< D OR > E) is not allowed
3. Abbreviations are terminated when the next simple condition is encountered: For example, IF A = B OR C OR D = E OR F expands to:
   IF A = B OR IF A = C OR IF D = E OR IF D = F

It is good programming practice to use parentheses to remove ambiguity from expressions, but remember than any expression in parentheses is evaluated first. The outcome of IF A > B OR (A = C AND A < D) is not logically the same as
IF (A > B OR A = C) OR A < D.

Condition-name test: allows the programmer to test using a programmer-supplied name (called a condition-name) for a specific value of an item (as opposed to using a relation test asking if the item EQUALS that value). The form is: IF PSN = CONDITION-NAME (where condition-name is a word defined at a special level number—88). The 88 will be found under an elementary item somewhere in the Data Division. 88s, entered in Area B, list condition-names and the values that go with those names. 88s do NOT create group items.No matter how many 88's are described in within a record layout, the item is only as long as its original (and only) picture clause.

88's make maintenance of programs easier, since when correspondence between, say item numbers and the items that go with those numbers, changes a programmer needs only change one set of 88s instead of having to search out every place an item number is referenced in the Procedure Division. Here is an example of one use of 88's:

Condition-name Example:

The WORKING-STORAGE layout:

```
01  EOF   PIC X      VALUE  'N'.
    88    NO-MORE-DATA       VALUE  'Y'.
    88    DATA-REMAINS       VALUE  'N'.
```

The EOF condition uses one byte of storage in this example and will store one of two values – either 'Y' or 'N'. The initial value is 'N'. In the PROCEDURE DIVISION, the 88's can be used to check for the EOF condition:

```
IF NO-MORE-DATA
    PERFORM 500-WRAPUP
ELSE
    IF DATA-REMAINS
       PERFORM 400-MAINLOOP.
```

The program will interpret IF NO-MORE-DATA as "IF EOF = 'Y'" and IF DATA-REMAINS as " EOF = 'N'". This is an example of how 88 tests are, as explained above, really just relation tests in disguise.

Other phrases used with IF statements:

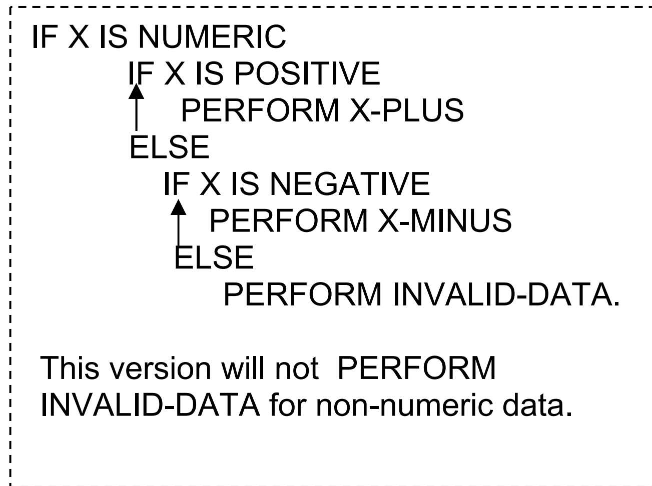
- ✓ **END-IF:** END-IF will terminate an IF structure without an ELSE being required. Use of END-IF is optional and can be redundant.. An IF will terminate either when it reaches a period or the END-IF phrase.

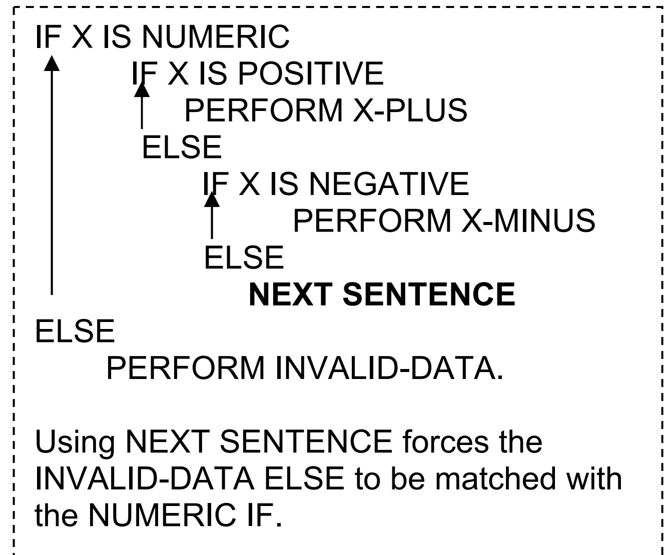
Example of the use of END-IF:

```
IF X = 4
   PERFORM PROCESS-RECORD
END-IF.
```

- ✓ **NEXT SENTENCE:** Used as a placeholder, mostly in nested IFs, when there is no statement following an ELSE. COBOL interprets next sentence to be the sentence that follows the period that ends the IF statement.

NEXT SENTENCE Example:





The EVALUATE Statement: EVALUATE is the COBOL structure for decisions that involve three or more possible values. A test in this form is called a "case structure". EVALUATE statements are useful because IFs that are nested more than three levels deep are hard to understand and may create subtle errors that are difficult to detect.

Since there must be a single entry and a single exit from the EVALUATE structure, GOTO may not be used with EVALUATE. Numeric criteria should be listed in ascending order.

Multiple conditions may be created by using ALSO in the first clause of an EVALUATE statement (these conditions in the tests that follow are dealt with in the order of the first clause). WHEN OTHER sets up a default condition in case none of the previous tests is satisfied.

EVALUATE Structure Example:

```
EVALUATE      DESTINATION-CODE
ALSO          SHIPPING-WEIGHT
ALSO          TRUE
ALSO          SHIPPING-WEIGHT * SHIPPING-RATE

        WHEN        “A” THRU “C”
        ALSO        1.0 THRU 1.5
        ALSO        DESTINATION-CODE = SOURCE-CODE
        ALSO        MAX-SHIPPING-CHARGE
            PERFORM  SHIPPING-PROCEDURE-1


        WHEN        “D”
        ALSO        1.0 THRU 1.5
        ALSO        DESTINATION-CODE  NOT = SOURCE-CODE
        ALSO        MIN-SHIPPING-CHARGE
            PERFORM  SHIPPING-PROCEDURE-2


        WHEN        “A” THRU “C”
        ALSO        5.0 THRU 9.0
        ALSO        DESTINATION-CODE =  “B”
        ALSO        AVG-SHIPPING-CHARGE
            PERFORM  SHIPPING-PROCEDURE-3

WHEN OTHER
            PERFORM GET-OPERATOR-HELP
END-EVALUATE.
```

In this example, DESTINATION-CODE is checked for 3 possible values and if a match is not found, the condition following WHEN OTHER is PERFORMed.

WHEN clauses are evaluated in the order they are coded. If a match for DESTINATION-CODE is found, then the ALSO conditions are examined. Expressions can be evaluated for equality with conditions and vice versa. DESTINATION-CODE = SOURCE-CODE, DESTINATION-CODE NOT = SOURCE-CODE and DESTINATION-CODE = “B” are compared to the logical constant TRUE and SHIPPING-WEIGHT * SHIPPING-RATE is compared for equality with MAX-SHIPPING-CHARGE, MIN-SHIPPING-CHARGE and AVG-SHIPPING-CHARGE.

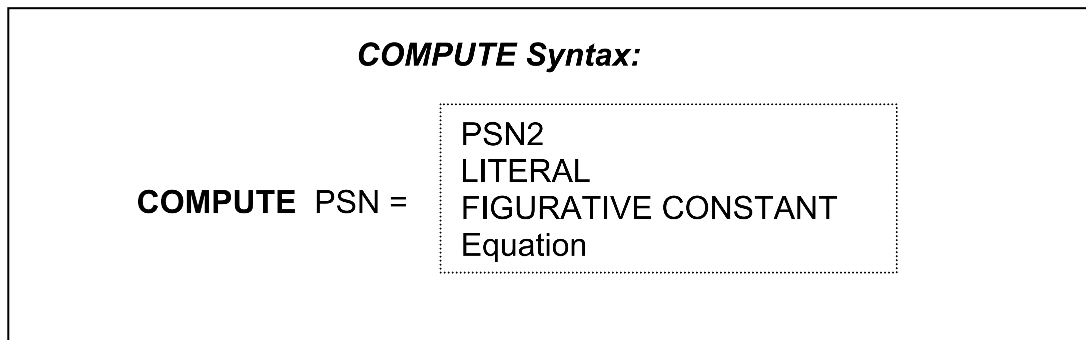
Math Operations

Math can be done two ways:

1. Equations using COMPUTE
2. Individual statements using ADD, SUBTRACT, MULTIPLY, DIVIDE

Multiple items in a statement may or may not be separated by commas – COBOL does not care. Commas are considered noise and are treated like spaces, so consistency is not required when using them.

The COMPUTE statement



There will usually be an equation on the right - using PSN2, LITERAL, or a FIGURATIVE CONSTANT on the right is a holdover from the earliest version of COBOL.

Symbolic Operators: The four math symbols are + - * /. Square roots can be taken as follows: PSN ** **.**5

The Order of Operations:

1. Inner parentheses to outer parentheses
2. Exponentiation and negation
3. Multiplication and division
4. Addition and subtraction

COBOL makes a separate pass for each level of operations, from left to right.

COMPUTE Example:

COMPUTE ANSWER = (5*(2*4)) + (-5) *2 + 3

The computer will make four passes to solve this equation, as follows:

1. ANSWER = 5*8 + (-5) *2 + 3
2. ANSWER = 5*8 – 5*2 + 3
3. ANSWER = 40 – 10 + 3
4. ANSWER = 33

Individual Math Statements

These have 2 formats – with or without the verb GIVING. All items used to do arithmetic in these statements must be numerically defined. If literals are used, they must be numeric, rather than nonnumeric, literals.

Math Statement Without GIVING Example:

MULTIPLY PRICE BY QUANTITY.
(The result is stored in QUANTITY)

ADD SALE-AMT TO SALE-TOTAL.
(The sum is accumulated in SALE-TOTAL)

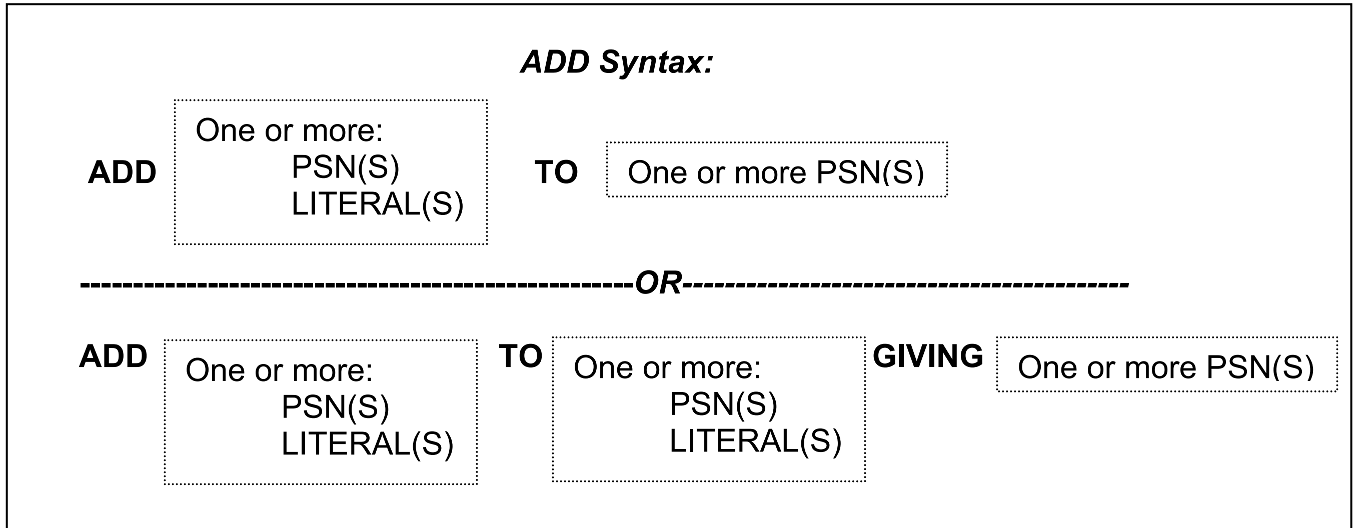
Each time the MULTIPLY statement is executed, PRICE retains its original value, but the amount in QUANTITY is overwritten. Every time the ADD statement is executed, SALE-AMT retains its original value, but SALE-TOTAL is overwritten.

Math Statement With GIVING Example:

MULTIPLY PRICE BY QUANTITY GIVING SALE-AMT.

PRICE and QUANTITY retain their original values and the result is stored in SALE-AMT, overwriting any previously existing value in SALE-AMT.

The ADD Statement (for examples, see after SUBTRACT)



ADD Examples:

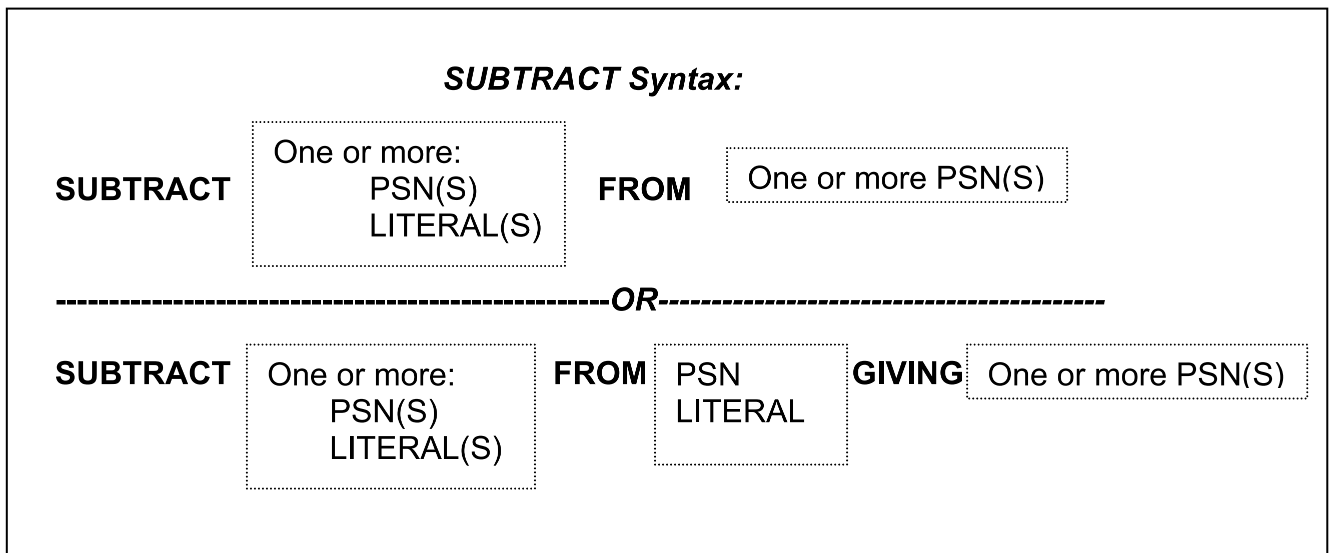
ADD X, Y GIVING Z.

X and Y must both be numeric (S,9,V) since they are involved in the math. Z may be numeric or numeric-edited because it is a passive recipient. Z will be overwritten.

ADD X, Y GIVING Y.

Y will be overwritten.

The SUBTRACT statement



Some ADD and SUBTRACT Statement Examples:

NOTE:C is overwritten

	Value Before				Value After			
Statement	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
ADD A TO B	3	6			3	9		
ADD A TO B GIVING C	3	6	28		3	6	9	
ADD A B TO C D	8	5	10	11	8	5	23	24
ADD A B TO B	1	2			1	5		
SUBTRACT A FROM B	3	1			3	-2		
SUBTRACT A B FROM C, D	3	2	5	6	3	2	0	1
SUBTRACT A B FROM 11, C	**LITERAL ON RIGHT NOT LEGAL**							

NOTE:
Equivalent to:
(A + B) + C = C
(A + B) + D = D

NOTE:
Equivalent to:
C – (A + B)
D – (A + B)

The MULTIPLY Statement

More than two operands are not allowed in a MULTIPLY statement. For example, MULTIPLY A B BY B. is not legal. This is due to the amount of temporary storage required for multiplication.

MULTIPLY Syntax:

MULTIPLY [PSN / LITERAL] **BY PSN2.**

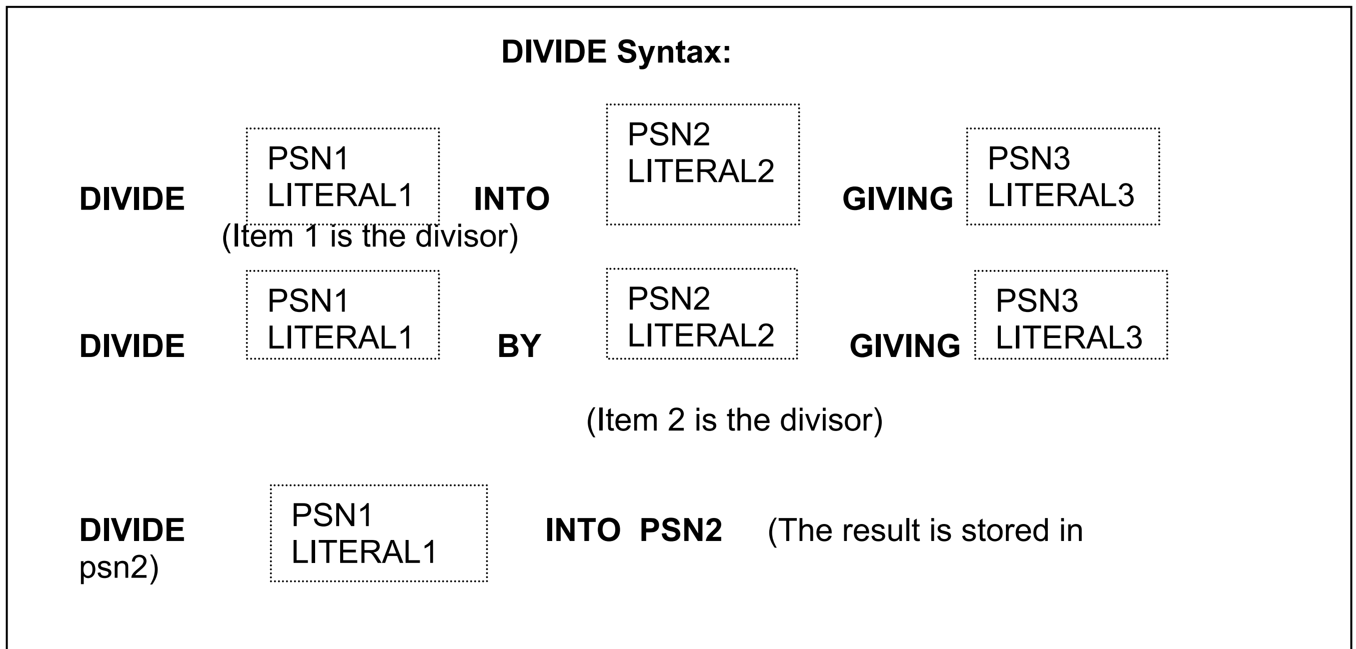
(In this statement, the product is placed in PSN2)

MULTIPLY [PSN / LITERAL] **BY** [PSN2 / LITERAL2] **GIVING PSN3.**

(In this statement, the product is placed in PSN3)

Exponentiation using individual statements: There is no special word or symbol to perform exponential math operations. Simply multiply the number by itself as many times as necessary, as follows: MULTIPLY PSN1 BY PSN1 PSN1 is equivalent to PSN1 cubed.

The DIVIDE statement



The use of the reserved word **REMAINDER** is optional. The remainder PSN follows the result field—it will be after GIVING if GIVING is used-- as follows:

DIVIDE X INTO Y GIVING Z REMAINDER Q.

Optional Phrases Used with Math Statements
(Individual or COMPUTE):

ROUNDED: Used to handle overflow to the right side of the decimal point. Rounds the result to the size of the picture clause. The word **ROUNDED** must follow the receiving field.

ROUNDED Example:

```
MULTIPLY A BY B  GIVING C ROUNDED, D, E ROUNDED
```

D will not be rounded in this statement, but C and E will be rounded. The ROUNDED function adds 5 to the intermediate result, then truncates. This is the accepted mechanism for accounting and finance.

```
COMPUTE X ROUNDED = (A + B) / C.
```

Because COMPUTE statements are not allowed multiple result fields, X will be rounded and there is no potential for rounding other items. **ON SIZE ERROR**: Used to identify situations where data has overflowed its allocated space on the LEFT (integer) side of the decimal. When a SIZE ERROR occurs, the receiving math field is NOT modified by the math statement, but the statement(s) following ON SIZE ERROR are executed instead. SIZE ERROR clauses go at the end of the math statements to which they apply. Size errors will occur:

1. When the absolute value of a result exceeds the largest value that can be stored in that field
2. When division by zero is attempted
3. When zero is raised to the zero or negative power, or a negative number raised to a fractional power

If the ROUNDED phrase is used with ON SIZE ERROR, rounding takes place before size error checking. When a size error occurs, what the program does depends on whether or not the ON SIZE ERROR phrase is specified. If the ON SIZE ERROR phrase is not specified and a SIZE ERROR condition occurs, truncation rules apply and a value is computed in the math field.

ON SIZE ERROR EXAMPLES:

```
MULTIPLY PRICE BY QUANTITY GIVING TOTAL-COST
          ON SIZE ERROR
                 PERFORM WRITE-ERROR-LIST.

COMPUTE TOTAL-COST = PRICE * QUANTITY
          ON SIZE ERROR
                 PERFORM WRITE-ERROR-LIST.
```

SIZE ERROR clauses are important in practical terms because over-large results from a math operation need to be looked at. It would be possible to avoid SIZE errors ever occurring by simply increasing the picture clause size to hold every possible result, but that would allow errors to slip through so it's not an intelligent approach. Leaving the picture clause alone, without using size error, allows for the possibility that an item might have its LEFTMOST digit truncated, which can lead to bizarre-looking sums.

.

EDITING STATEMENTS

EDITING IS DONE WITH Elementary MOVEs.

MOVE basics:

Three kinds of MOVES:

1. **Elementary** (one elementary item to another; replaces entire receiving group):

- ✓ The receiving item is completely replaced.
- ✓ How the move is done depends on picture of the receiving item.
- ✓ If nonnumeric, left to right, pad on the right with blanks.
- ✓ If numeric, moves away from the decimal point, pads with zeros.

2. **Group** (either or both is a group; also replaces entire receiving group)

Treats the receiving item as one alphanumeric (X) item. COBOL doesn't check receiving items to see what type they are supposed to be. So, if you group-move items that you intend to use in math operations, you may have a problem.

3**. MOVE CORRESPONDING** (replaces only those fields in the receiving group which correspond to fields from the sending group)

Correspondence means

identical names and same relative positions within the group hierarchy (in other words belong to the same subgroups)

but does not require

same picture clauses, same level numbers or same order in the two groups

MOVE CORRESPONDING Example:

Only the 01 level PSN's will be different – everything below 01 level must have identical names and group memberships. In this example, TR-GROUP-ID will not be moved, because the PSN's are not identical. TR-SSN will be moved because the PSN's and PIC clauses match and they are at the same depth within their respective groups; it does not matter that the level numbers are different.

MOVE CORRESPONDING TRANSACTION-REC TO DATA-LINE**.**

```
01 TRANSACTION-REC.
     05 TR-INSURED-ID
             10 TR-GROUP-ID    PIC XX.
             10 TR-SSN.
                  15 TR-SSN1   PIC X(3).
                  15 TR-SSN2   PIC XX.
                  15 TR-SSN3  PIC X(4).
```

```
01 DATE-LINE.
    05  TR-INSURED-ID.
          10   TR-GROUP     PIC XX.
          10 TR-SSN.
               20 TR-SSN1   PIC X(3).
               20 TR-SSN2   PIC XX.
               20  TR-SSN3  PIC X(4).
```

A complete Data Item Typology:

NOT NUMBERS	Legal in PIC	Legal data
Alphabetic	A, B	letters and spaces
Alphanumeric	X, XA, XB	anything
Alphanumeric-edited	X, B, 0	anything
Numeric-edited	See explanation below	digits, sign
NUMBERS	**Legal PIC**	**Legal data**
Numeric int	S, 9	digits, sign
Numeric non-int	S, 9, V	digits, sign

Editing Options: Only elementary items may be edited. It is not a good idea to use the alphabetic (A) type for names due to possible presence of punctuation (i.e. O'Hare).

0 and **B** are **fixed insertion** characters and will always print in a fixed position. B inserts blanks and thus eliminates the need for filler. Multiple spaces can be designated by PIC B(number of blanks you want).

Note that when using Bs, you have to provide for ALL your sending positions. If XX is moved to XB, only the first X is moved – the second X is replaced by a blank.

B Example:

MOVE SR-NAME TO DL-NAME

```
05 SR-Name.
          10 SR-First   PIC X (10).
          10 SR-MI      PIC X.
          10 SR-Last    PIC X (10).
```

```
05  DL-Name PICX(10)BXB(10).
```

Use B instead of FILLER and save a few lines in the record layout!

How to print separators (hyphens, slashes, etc.) without using FILLER:

INSPECT… REPLACING:

```
10 TR-SSN.                          10 DL-SSN   PIC X(3)BXXBX(4).
        15 SSN1   PIC X (3).
        15 SSN2   PIC XX.
        15 SSN3   PIC X(4).

                    MOVE TR-SSN TO DL-SSN.
                    INSPECT DL-SSN REPLACING ALL SPACES BY  '/'.
```

The **0** (Zero) editing character is used to expand the width of an alphanumeric field. This is practical for when you want to expand the width of a field in an existing file without having to manually reenter all the existing data.

0 Example:

If the current input file contains:

```
        05 CUST-ACCT-NUM        PIC X(6).
```

To increase the field to 9 spaces, rewrite the layout using:

```
        05 WS-CUST-ACCT-NUM    PIC 0(3)X(6).
```

Then, set up the new data line for the nine-space number:

```
        05 DL-CUST-ACCT-NUM     PIC X(9).
```

Numeric Editing

Numeric-edited picture clauses are used to make numbers on reports easier to read. The symbols used in these clauses are 9, decimal point, comma, $ + - CR DB Z and *. Note that math operations CANNOT be done using edited clauses. A clause such as PIC 999.99, for instance is considered edited due to the presence of the decimal point so math with this clause would not be allowed.

How numeric data are represented on the printed page if editing is not used: If data are signed, right-most digit will not be printed as a number. The sign and the last digit are combined into one representation as follows:

Digit	Sign Positive	Sign Negative
0	{	}
1	A	J
2	B	K
3	C	L
4	D	M
5	E	N
6	F	O
7	L	P
8	H	Q
9	I	R

The S, 9, V characters form the numeric clause. These clauses MUST be used for math operations, but are not very human-friendly when printed.

Example of How a Numeric Clause will appears on a printed page:

PIC S9(4)V99   VALUE 56.70 will print as 00567{

The use of 9 in numeric-edited pictures

Programmers use 9s in numeric-edited pictures to force a digit. This is usually in cases where, if zero were sent to that picture, the result would otherwise be blank. For instance, $ZZ,ZZZ.ZZ or $$$,$$$.$$ would be more likely rendered $ZZ,ZZZ.99 and $$$,$$$.99.

Decimal point and Comma:

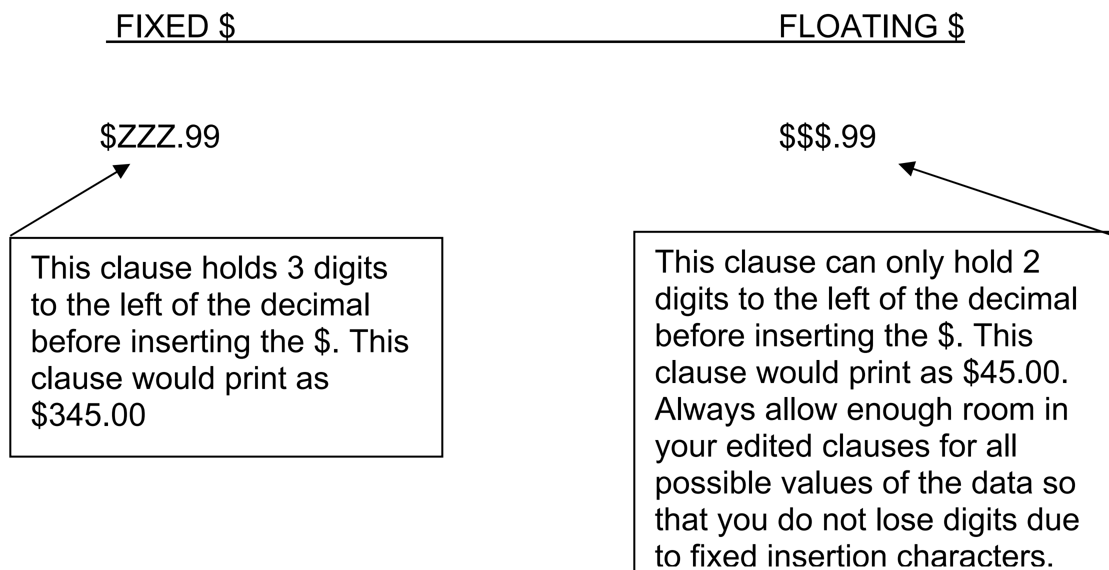
These are **fixed insertion** characters and will appear in the result in the position that is specified in the PIC clause. If there is no number to the left of the comma, it's replaced by a blank. If the overall picture clause wouldn't print anything, then the decimal and comma aren't printed either.

$

This may be used as a **fixed insertion** character, but it may also be used for **floating insertion**. If it's fixed, it will print in the position it's put in the picture (always preceding the leftmost digit of the number). If it's floating (more than one in the picture), it prints (only once!) in front of the first non-zero digit COBOL encounters. In other words, a floating $ sign suppresses zeros. If you're using floating dollar signs, there must always be one more to the left of the decimal than there are potential digits to be put there. If you send all zeros to a picture which is, essentially, all dollar signs, then nothing will be printed.

Example of Fixed $ versus Floating $:

Given a value of 345.00 and these PIC clauses:



Forcing a sign indicator to print

As we said earlier, having a sign on a number won't print a sign in a humanly readable format. So, there are two options for sign-printing.

+ and **–**

These will print actual, physical signs on your output.

+ always prints a sign. + for positive or zero, - for negative.
- prints a sign only for negative numbers

Pluses and minuses can be fixed or floating in exactly the same way as $ signs.
If they're fixed, they can print on the right or the left, depending on where the programmer puts them in the picture. If they float, they'll print on the left.

CR and DB

Both of these are intended for negative numbers. They are always fixed, always on the right, and they print themselves.

Z

Z stands for **Zero-suppression.** It replaces any leading (left-hand) zeros with blanks—note that $, + - ,CR, and DB also do this zero replacement, but they also insert-- and, like those other symbols Z suppresses unnecessary commas. As with other numeric pictures with a potential value of blanks, it's a good idea to avoid the appearance of error by using 9s as well.

Example of a zero-suppressed PIC clause:

PIC $ZZZZ.ZZ VALUE 56.70 will print $ 56.70.

Example of using 9 to force a character to print for a data value of zero:

Sending zero to PIC ZZZZ.ZZ will print
Sending zero to PIC ZZZ9.99 will print 0.00

*

The * is intended for check-protection. It prints itself where a Z would print spaces. In doing this it will also replace commas but not the decimal point.

Example of the check-protection character:

PIC $**,***.99 VALUE 500.00 will print $***500.00

More examples of numeric-edited clauses:

Will cut off 1 because $ must print

Sending		Recieving	
PIC	**VALUE**	**PIC**	**VALUE**
PIC S9(5)V99	100256Ō	+$$,$$$.99	-$0,025.60
PIC S9(5)V99	002005	$--,---	$ 20.
PIC S9(5)V99	00000.08	$++,+++.++	$ -.08
PIC s9(5)V99	1234567(neg)	$$,$$$.99	$2,345.67

Will insert the decimal in the sending clause, but will come into receiving clause as '0020' since receiving clause does not allow for positions to the right of the decimal.

$ followed by 5 blanks. If you see zeros to the left of the decimal, you can assume that there was a digit replaced by the $ and that your PIC clause is not large enough.

There is no provision for the sign in the receiving clause….

EXHIBIT NAMED and **READY TRACE** are verbs available in IBM-compatible environments. Their only purpose is to find errors.

EXHIBIT NAMED

displays the value held in a PSN while the program is running. You can use **DISPLAY** to accomplish the same function on COBOL versions that do not support EXHIBIT NAMED (this includes Grove) or if you want to display information besides than the value.

Example Demonstrating How To Use EXHIBIT NAMED:

To check the values of CUR-DEPT and HOLD-DEPT in this code snippet:

```
IF HOLD-DEPT = CUR-DEPT
       PERFORM ITEM-BREAK
        PERFORM DEPT-BREAK
ELSE…..
```

```
EXHIBIT-NAMED HOLD-DEPT
                         CUR-DEPT.

IF HOLD-DEPT = CUR-DEPT
       PERFORM ITEM-BREAK
        PERFORM DEPT-BREAK
ELSE…..
```

Insert EXHIBIT NAMED before the logic begins. The values of HOLD-DEPT and CUR-DEPT will be written on the output listing in the form of 'PSN = value'. If of HOLD-DEPT and CUR-DEPT were declared as PIC X(2) VALUE ZERO, then the print out for the first pass through this loop would be 'HOLD-DEPT = 00, CUR-DEPT = 00'.

DISPLAY

is a general-purpose 'on-screen' verb available in all versions of COBOL. We will see much more of it in Program 4.

An example of the same function using DISPLAY:

```
DISPLAY 'HOLD-DEPT = ', HOLD-DEPT AT LINE 6 COLUMN 10.
DISPLAY 'CUR-DEPT =', CUR-DEPT AT LINE 6 COLUMN 13.
```

Display, unless printing is specified, will display on the screen.

READY TRACE

This lists every paragraph that COBOL has executed and will report the information in the output list in the form of: '200-SETUP, 500-WRITE, 300-MAINLOOP, 400-READ…..'. Sometimes READY TRACE uses line numbers instead of paragraph titles.

Example using READY TRACE in the DRIVER paragraph:

```
100-DRIVER
        READY TRACE.
        PERFORM 200-SETUP.
        PERFORM 300-MAINLOOP…
```

You cannot use an IF statement with a trace – COBOL will ignore it. Use ON to detect conditions that will trigger or stop the trace. STOP cannot be used with READY TRACE - use RESET.

Example using READY TRACE to trace the third record:

```
ON 3
   READY TRACE.
ON 4
   RESET TRACE.
```

Some common mistakes:

The most common reason COBOL programs stop running is that the program is trying to do math with something that in the data is not a number. The cause for this abend isn't simple.

Following, for instance, are some reasons why a program may stop running the first time math is tried:

- ✓ The first data record has data-entry error which made data non-numeric
- ✓ The input record description may be incorrect
- ✓ An accidental 'return' (enter key) in data entry introduced a line of spaces into data record. When program tries to do math with spaces, it quits.
- ✓ The math group was not initialized math group to zero and program is trying to do math with spaces

Here are some reasons why the program might run for a while, and then unexpectedly quit:

- ✓ If it quits after N records, any of the above errors could be the culprit except initialization error. The problem will be in the record that follows the last record in the output.
- ✓ If it quits before the final totals, there is probably a line of spaces in the output. There cannot be any read errors, since the reading is already finished.

Logic problems often come from using wrong words or wrong picture clauses on items being tested. Misplacement of periods in IF statements is also, as we've seen all semester, a famous pitfall.

Direct Access (ISAM) Files:

There are several file types that allow direct access to specific records. The one we will use in this class, ISAM (Indexed Sequential Access Method) is a file management system developed at IBM that allows records to be accessed sequentially (in order), randomly (directly), or dynamically (a combination of random and sequential access).

Sequential files are easy to store and efficient to process and excellent in a master update situation when most or all of the records must be updated in a single process. However, files of this type are inefficient when only a few records will be updated because the whole file still has to be read, processed and rewritten. If that's the case, then indexed files are a better choice because they allow any record to be accessed and updated individually, without need to read all of the preceding or subsequent records.

The physical difference between ISAM files and sequential files is that ISAM files have index areas (where the record keys are stored) and overflow areas. When records are added to an ISAM file, they aren't inserted directly but are placed in the overflow area. When records are deleted from these files, they aren't deleted either but just removed from the index area so they can't be accessed any more. A system utility has to be run every once in awhile to clean this situation up—physically insert added records and physically remove deleted ones.

ISAM files may use one of several indices to associate a logical record with its physical location on disk. The key for each record that is stored in the index is called a **primary key**. It consists of one or more fields of data from within the record. The primary key must be unique to each record. 'Intelligent keys' that depend on things phone numbers or zip codes are not good choices for primary keys because that information can and does change frequently and can also be duplicated. Intelligent key information is, however, suited for use as an **alternate key**, which can be used to speed access to data by providing additional information about the record to the index.

ISAM files are declared as such in the ORGANIZATION clause of the SELECT statement. The SELECT must also specify which access type you plan to use for this program. Keys, both primary and alternate, must be specified here as well. Though the primary key must be unique, alternate keys may have duplicates, if so, that's declared in the SELECT statement by using WITH DUPLICATES.

Here is a typical ISAM SELECT statement:

```
SELECT        STUDENT-FILE
ASSIGN TO    "class/account/student-info.data"
ORGANIZATION IS INDEXED
ACCESS IS SEQUENTIAL [RANDOM, DYNAMIC]
RECORD KEY IS STUDENT-RECORD-KEY
ALTERNATE RECORD KEY IS
              STUDENT-NAME WITH DUPLICATES.
```

ACCESS IS SEQUENTIAL is the default access mode; use of this clause is optional in that case. RANDOM and DYNAMIC access modes must be stated. The primary key is, of course, required. There is no limit to the number of alternate keys a file may have but these should be used sparingly, because while they can reduce access time to records, they also create additional overhead in storing and maintaining index files.

You are already familiar with two of the four OPEN modes - INPUT and OUTPUT. There are two other modes that can be used to update files, **I/O** and **EXTEND**. I/O can be used with any access method (sequential, random or dynamic) to READ, REWRITE and DELETE. EXTEND can only be used to add records to the end of a file that is accessed sequentially. The primary keys of the records being added must be greater than the last primary key in the existing file. This chart shows the verbs that may be used with each access method and OPEN mode:

	OPEN INPUT	OPEN OUTPUT	OPEN I/O	OPEN EXTEND
Sequential	READ START	WRITE	READ REWRITE START DELETE	WRITE
Random	READ	WRITE	READ WRITE REWRITE DELETE	**INVALID**
Dynamic	READ START	WRITE	READ WRITE REWRITE START DELETE	**INVALID**

Because sequential files are read in order, you must specify actions for the program to take when it reaches the EOF condition using AT END. You are already familiar with this format for a READ statement. This method will not work with random or dynamic access methods because the last record in the physical file can be read at any point in the process, not just at the end. Instead, the primary key must be read from the record and compared to a valid primary key which has already been MOVEd into the RECORD KEY that is designated in the SELECT statement. The phrase INVALID KEY specifies what action should be taken if the keys do not match. INVALID KEY may be used only with random and dynamic access, never with sequential access.

Example of READing a Random Access File:

```
MOVE '999888777' TO STUDENT-RECORD-KEY.

READ STUDENT-INFO-FILE INTO WS-STUDENT-RECORD
      INVALID KEY
            MOVE 'No such student' TO ERROR-MSG
      NOT INVALID KEY
            PERFORM  NEW-STUDENT-RECORD
END-READ.
```

The verb START is used with sequential or dynamic files to position the file to a specific record. The record is not read or modified, so START must be followed by a READ statement. As above, a value must be moved to RECORD KEY, than a comparison is made to the record keys in the file using relational operators (<>=).A file can be STARTed multiple times within one program using the same or different keys each time.

Example using START:

```
MOVE '999888777' TO STUDENT-RECORD-KEY.

START STUDENT-INFO-FILE
        KEY IS GREATER THAN OR EQUAL TO '999888777'
        INVALID KEY
             MOVE 'No records match your criteria' TO ERROR-MSG
END-START.
```

With dynamic file access, once the file pointer has been positioned using START, reading can proceed sequentially using the reserved words NEXT RECORD. Note the NEXT RECORD clause of the READ. This is required when you read sequentially and access is dynamic.

Example of dynamic access READ using NEXT RECORD:

```
READ STUDENT-INFO-NAME NEXT RECORD
                                          INTO WS-STUDENT-RECORD
AT END
      MOVE 'Y' TO EOF
NOT AT END
      PERFORM PROCESS-STUDENT-RECORD
END-READ.
```

Any access method can write files using the WRITE syntax you already know. You can also use the optional phrase INVALID KEY to check for non-unique primary keys or, in the case of sequential files, keys that are not in order. REWRITE is used with non-sequential access to replace existing records when the file has been opened in I/O mode. The record that is modified depends on which access method is used - for sequential files, the record rewritten is the one in the input buffer from the previous read; for random and dynamic access the record that is rewritten is the one that matches the primary and/or alternate key(s). The INVALID KEY condition may not be used with sequential files.

Example of REWRITE For a Random or Dynamic Access file:

```
REWRITE STUDENT-RECORD FROM WS-STUDENT-RECORD
    INVALID KEY
        PERFORM WRITE-ERROR-FILE
  END-REWRITE.
```

In order to delete a record from a sequentially accessed file, you must first read it into the input buffer. Randomly accessed records can be selected for deletion by their primary key and do not need to be read first. As always, you may not use the KEY conditions with sequential files.

DELETE Examples:

```
MOVE ‘223224444’ TO STUDENT-SSN.
DELETE STUDENT-INFO-FILE RECORD
      INVALID KEY
          DISPLAY ‘No such student’ AT LINE 1 COILUMN 10
END-DELETE.

DELETE STUDENT-INFO-FILE RECORD
     INVALID KEY
          PERFORM WRITE-ERROR-FILE
  END-DELETE.
```

Working with Tables (Arrays):

Tables are lists of data items with a collective name.

Because all items in the table have the SAME name, you have to be clear which one you want when you're using the name. There are two types of "table locators in COBOL—subscripts and indexes. Either a subscript or an index can be used to point to a specific entry. They look the same syntactically—each is a word in parentheses following the PSN-- but they aren't. The difference will be explained later. Here are two examples of table locators.

```
DEPARTMENT (5)

DEPARTMENT (ORDER-DEPARTMENT)
```

The first of these will pick up the fifth department off a list of departments. The second will look at the current value of ORDER-DEPARTMENT -- which has to be an integer—and use that to see which department the programmer wants.

Here is an example of how you might use a table—first a program without it, then the same program with it.

A Payroll Input Record Without a Table:

….

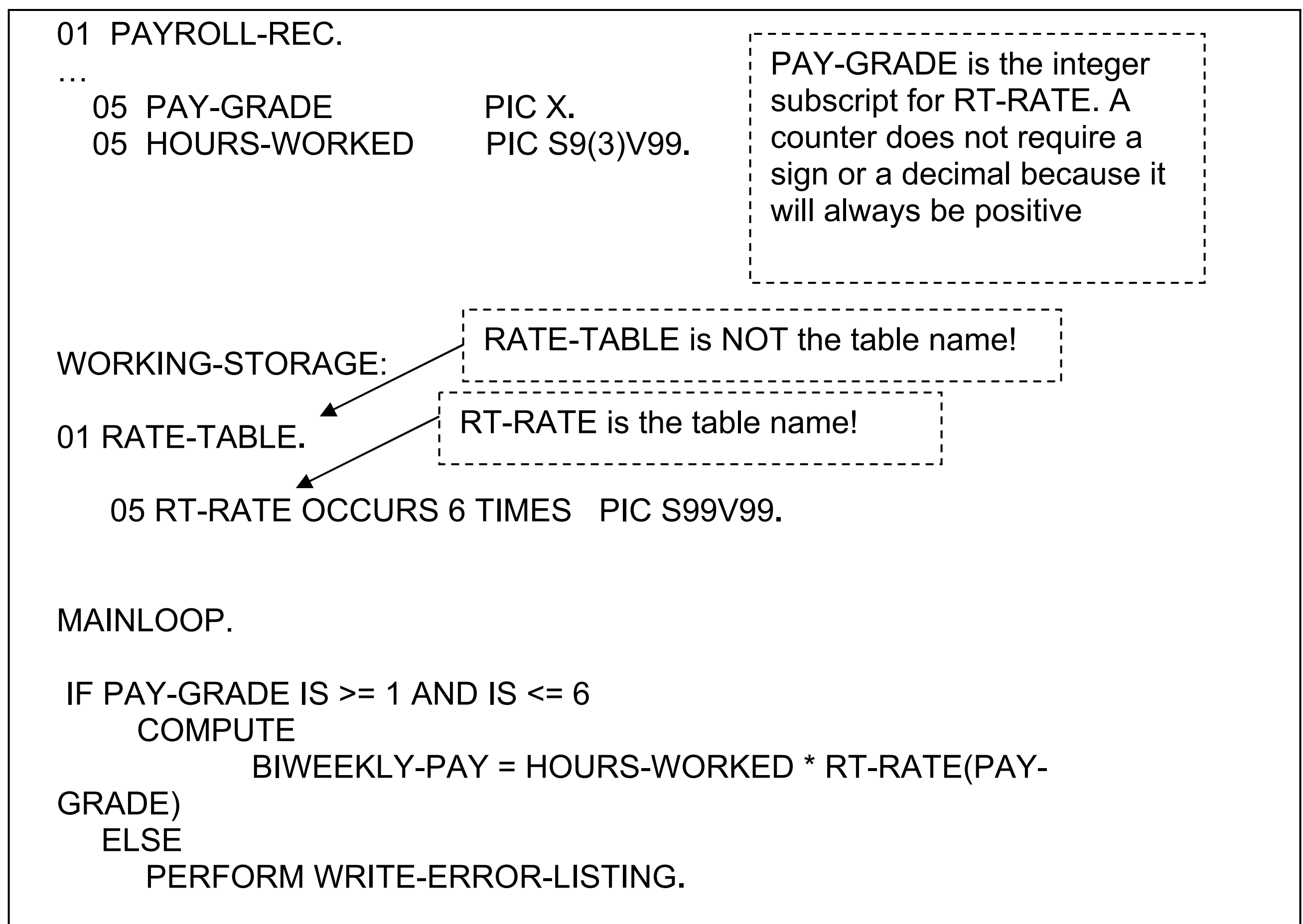
```
01  PAYROLL-REC.
…
   05  PAY-GRADE              PIC X.
   05  HOURS-WORKED           PIC S9(3)V99.

WORKING-STORAGE SECTION.

01 RATE-LIST.
    05    RATE-1     PIC S99V99     VALUE   0515.
    05    RATE-2     PIC S99 V99    VALUE   0575.
              |
              v
   05     RATE-6     PIC S99V99      VALUE  2500.

PROCEDURE DIVISION.
DRIVER.
SETUP
(opens, reads first record)
MAINLOOP.
  IF PAY-GRADE = '1'
     COMPUTE BIWEEKLY-PAY = HOURS-WORKED * RATE-1
   ELSE
         IF PAY-GRADE = '2'
                 COMPUTE BIWEEKLY-PAY = HOURS-WORKED * RATE-2
           ELSE
               IF PAY GRADE = '3'…
```

Of course, EVALUATE would be a better way to do this loop, but a table is even more efficient………..

To change this example to use a table:

1. Change the PIC of PAY-GRADE to 9's because a table can be accessed only numerically
2. Change RATE-LIST to a table

A Payroll Input Record With a Table:



Now! Here is what we learn from the above.

OCCURS

A table is established when you use **OCCURS**.

OCCURS:

1. CANNOT be used at 01
2. Is not allowed to have a VALUE clause

The word TABLE isn't reserved. Notice what the example tells you—that the name of the table is wherever the OCCURS clause is, never at 01 where it's defined.

The name RT-RATE can’t be used by itself because that would be ambiguous since there are 6 occurrences of RT-RATE.in the example. In fact, if you hadn’t specified which occurrence you wanted in the MAINLOOP above, the program wouldn’t even have compiled.

The phrase OCCURS DEPENDING ON is used to define a table that contains variable-length records. The amount of memory allocated to each record is controlled by the value of the PSN that follows OCCURS DEPENDING ON. This can result in a significant amount of savings in storage overhead, especially in the case of very large files. Consider a file that contains a record for every course a student has completed along with their grade and the instructor name. For a freshman, the record would be quite small, but for a graduate student the record would be very long, (especially if they changed their mind about their major a few times along the way!). Consider the following:

```
01  STUDENT-COURSE-LISTING.
            05   SCL-TABLE OCCURS 0 TO 100 TIMES
                     DEPENDING ON NUM-COURSES-COMPLETED.
```

Table Range

The range of any table is the spread from one to the number of occurrences in a table, and so it represents the possible values a subscript (index) for that table can have. The range for our original (nonvarying) example is 1-6, and if the subscript is, for instance 7, the computer won’t be able to find that entry and it will stop your program. To avoid this problem you should always test your subscript or index—as we did above-- to make sure it is within range before you use it.

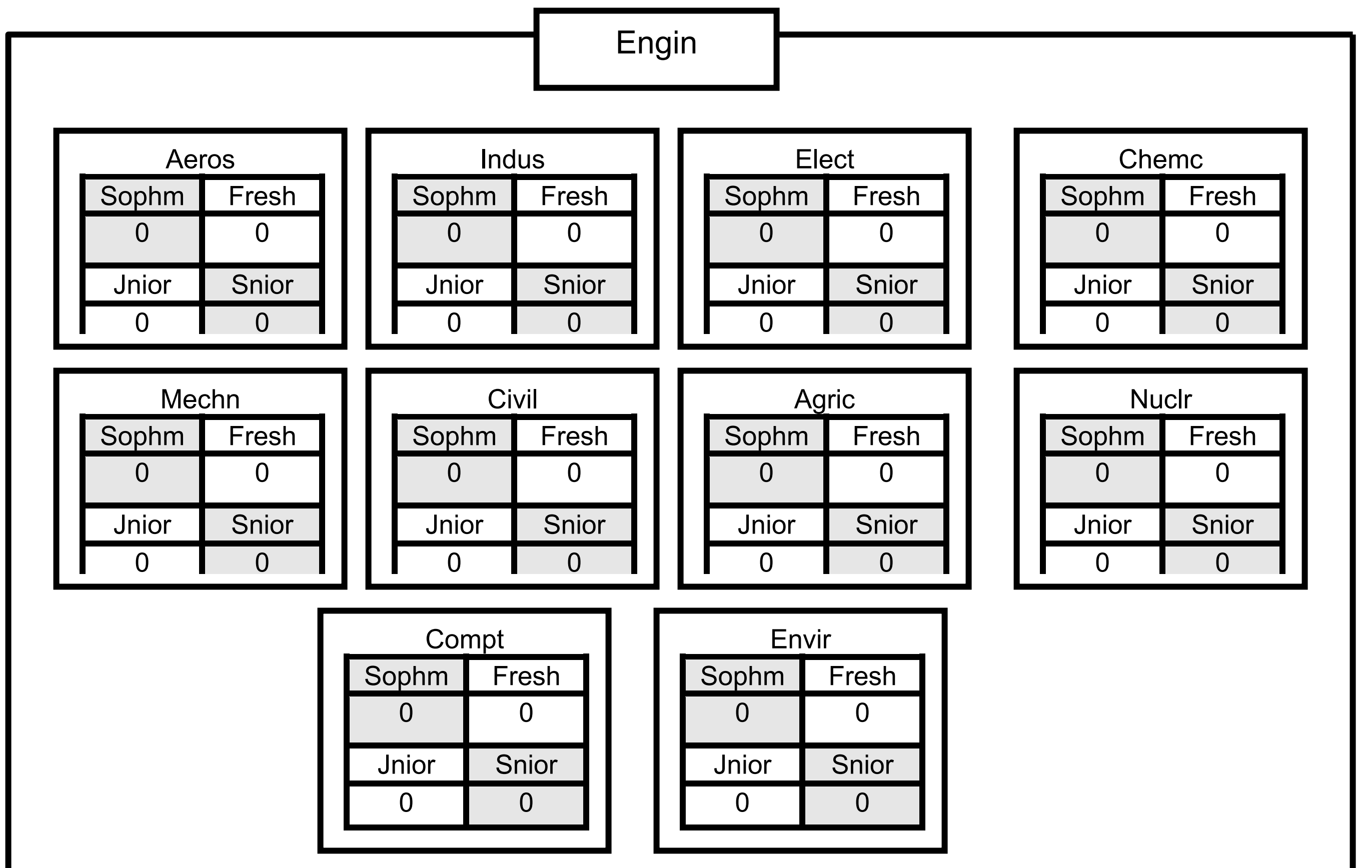
Multi-dimensional Tables

Multi-dimensional tables are created by nesting OCCURs clauses inside each other. Although COBOL allows nesting up to 7 levels deep, tables more than 3 levels deep are unwieldy and difficult to work with. The following table counts the number of freshman, sophomore, junior and senior students enrolled in various majors within each college.

Example of a Multi-dimensional Table:

```
01  COLLEGE-ENROLLMENT-TABLE.
    05    COLLEGE OCCURS  26 TIMES.
          10     MAJOR  OCCURS 10 TIMES.
                 15    CLASS-STANDING OCCURS 4 TIMES.
                       20  NUMBER-OF-STUDENTS    PIC  SX(5).
```

The memory allocation for this table is the result of multiplying all of the elements by the size of the PIC clause. This table would require (26 X 10 X 4) X 4 = 4160 bytes of storage. The illustration below shows one occurrence of COLLEGE - there are 25 more occurrences laid out like this. Note that this is supposed to be read horizontally; the paper is just not wide enough.



How values get into tables

Since one of the rules for the OCCURS verb is that it can't use a VALUE clause, there must be some way to put values into a table or the whole situation would be ridiculous. Well, it's not (ridiculous). Here are the ways:

1. READ INTO
This will load the records directly into the table.

Example 1 (only one record to be loaded):

Always use sub-paragraph to load table.

```
SETUP.
   (open)
   PERFORM 500-LOAD-TABLE'.
   CLOSE INPUT-FILE.

LOAD-TABLE.
   READ INPUT-FILE INTO RATE-TABLE.
   AT END
        MOVE 'Y' TO EOF.
```

Example 2 (table requires loading multiple records):

```
SETUP.
   (open)
   MOVE 1 TO COUNTER. (COUNTER from W-S PIC 9s—as many as
                                  needed)
   PERFORM 500-LOAD-TABLE UNTIL EOF = 'Y'.
   CLOSE INPUT-FILE.

500-LOAD-TABLE.

    READ INPUT-FILE
      AT END
         MOVE 'Y' TO EOF
     NOT AT END
```

```
        MOVE INPUT-REC TO TABLE-RECORD (COUNTER)
        ADD 1 TO COUNTER.
```

The above could also be rendered:

```
        READ INPUT-FILE
             AT END
                 MOVE 'Y' TO EOF.
        IF EOF NOT = 'Y'
              ADD 1 TO COUNTER.
```

2. **MOVE**

   Used to initialize table, usually with blanks or spaces.

   Example: MOVE ZEROS TO EOJ-STATS-TABLE.

3. **CALCULATION**.

   Perform a computation and place the results in the table.

This example reads 25 records on the input file computes the sales totals for the items on the records, then places them in a table. A counter is used to move through the table.

Example of loading a table using a computation:

```
…
FILE SECTION.
…
01   SALES-INFORMATION-FILE.
          05      ITEM-NUMBER                  PIC X(6).
          05      QUANTITY                     PIC 9(4).
          05       PRICE                       PIC 9(3)V99.

WORKING-STORAGE-SECTION.

01   SALES-TOTAL-TABLE.
     05    STT-TABLE OCCURS 25 TIMES          PIC S9(5)V99.

01   ITEM-COUNTER                              PIC 99          VALUE 1.
```

```
PROCEDURE DIVISION.
…
PERFORM 900-READ-SALES-FILE.
COMPUTE SST-TABLE (ITEM-COUNTER) =
      QUANTITY   *    PRICE.
ADD 1 TO ITEM-COUNTER.
```

A report that gives the item number and the sales total can be written by using the counter and MOVE statements.

```
SET ITEM-COUNTER TO 1.

IF ITEM-COUNTER <= 25
        PERFORM 900-READ-SALES-FILE
        PERFORM 800-PRINT-LINE
ELSE…

 800-PRINT-LINE.
        MOVE ITEM-NUMBER TO OUTPUT-FORMAT.
        MOVE STT-TABLE (ITEM-COUNTER) TO OUTPUT-FORMAT.
        MOVE OUTPUT-FORMAT TO OUTPUT RECORD.
        PERFORM WRITE-OUTPUT.
        ADD 1 TO ITEM-COUNTER.
```

4. **Hard-coding**.

This means the values in the table can be read directly by looking at the program. You use a new verb for this-- REDEFINES.

REDEFINES

This verb gets around the prohibition of not using value clauses with OCCURS by setting up a second area after the one with the value clauses, and putting the OCCURS clause in that. The most standard use for this approach is the following table that stores the names of the months of the year:

The Month table:

```
01 MONTH-LIST.
   05      PIC X(9)    VALUE  ‘  January’.
   05      PIC X(9)    VALUE  ‘ February’.
   05      PIC X(9)    VALUE  ‘    March’.
                      .
                      .
   05    PIC X(9)      VALUE  ‘ December’.

01 MONTH-TABLE REDEFINES MONTH-LIST.
         05 MT-MONTH OCCURS 12 TIMES      PIC X(9).
```

Notice that each month name is right justified using spaces…

This is what the month table looks like internally. To access it, you’d just use either the number of the month, like MT-MONTH (5), which would get you May, or a word whose value would contain a number between one and twelve, which would give you flexibility.

1	January
2	February
3	March
4	April
5	May
6	June
7	July
8	August
8	September
10	October
11	November
12	December

Notice that we right-justified the names of the months by padding on the left with spaces. This forces the name of the month to line up next to the day when, later in the program, it is MOVEd to the heading. If you do not right justify, May and other months with less than 9 characters in their names will print like this:

May   5,2001   *instead of*   May 5,2001

The rules for REDEFINES:

1.This verb isn't allowed at the 01 level in the file section.
2.Redefinitions must follow the word being redefined.
3. They must be exactly the same length.
4.The must be at the same level numbers.
5.You can redefine things as many times as you want, but all redefinitions have to use the original PSN.

Note that redefining can let you use an item in more than one way, and that's the reason you do it. It's NOT illegal to redefine Xs as 9s with decimal points, for instance.

Other Uses for REDEFINES:

REDEFINES is an important verb. It's not used only for tables. It can be used anywhere in the data division to reconfigure elementary or group data items. There are a couple of reasons you might want to do this. First, you may have a record, which you want to use a part of in more than one way. In that case, you can redefine the whole area. Another reason for using REDEFINES is to check for errors, such as the presence of blanks in numeric data. This cannot be done directly, since the reserved word SPACES cannot be used with numeric data. Suppose you have a pay rate stored as:

```
05     PAY-RATE        PIC S99V99.
```

In order to check for blanks, add a second entry following PAY-RATE to redefine the same data as alphanumeric:

```
01 PAY-RATE-2    REDEFINES PAY-RATE PIC X(4).
```

In the PROCEDURE division, check the redefined data for blanks:

```
IF PAY-RATE-2 = SPACES
    PERFORM ERROR-HANDLING-ROUTINE….
```

Using the SEARCH verb

All the items in tables we've looked at so far have been directly located using a subscript or index. Often, though we want to search through the table to match some value—for instance, looking up an account number for a customer.

This can be accomplished using verbs we already know, but it is easiest to use the verb COBOL offers us—namely, the SEARCH verb. The format of this verb is:

```
SEARCH table-name
       AT END
         command(s)
       WHEN (*matches)
         command(s).
```

*this is expressed in the form of psn-you-want = psn-in-the-table(index)

The commands following AT END are what you want to do if COBOL looks through the whole table and doesn't find a match. The commands following WHEN are what you want to do if the match is found.

To use a SEARCH verb, you must define your table as INDEXED. (SEARCH verbs won't work with a subscript). This is easy. You say

```
01 TABLE.
     05  ENTRY OCCURS n TIMES  PIC whatever
                INDEXED BY index-psn.
```

Here's another, more specific, example:

```
01    ACCOUNT-TABLE.
           05   AT-REC OCCURS 5 TIMES INDEXED BY AT-INDEX.
                    10 AT-ACCOUNT-NUMBER        PIC X(10).
                    10 AT-CUSTOMER-NAME.
                       15    AT-CN-FIRST              PIC X(10).
                       15    AT-CN-MIDDLE             PIC X.
                       15    AT-CN-LAST               PIC X(15).
```

In other words, all you have to do is claim your table's indexed and COBOL will get your index ready for you to use. The index-psn is NOT defined anywhere else, just here.

Before you search, you must put a value in your index by using a special verb for indexes—SET (you can't use MOVE):

SET index-psn TO numeric-literal.

You usually set the index to 1, because usually you want to start looking through the table beginning with the first entry. While the SEARCH is being done, COBOL will automatically increment the index until it either finds a match or finishes looking through the table entries. If you don't reset the index every time, the search will begin at the index number where it left off previously. For example, if the previous match was found at the third entry and the index was not reset, the next search would begin at the fourth record. Records 1-3 would not be searched again.

The difference between indexes and subscripts

When COBOL looks something up in a table, it performs a little calculation:

Initial address (where the table starts in memory) + displacement (how far from the start of the table the item is).

When you use an index, the displacement for each item in the table is computed and stored, so COBOL doesn't have to compute it every time. In its simplest form, the formula for computing displacement is:

(Occurrence # - 1) * picture-size.

In other words, the first item on any table will have 0 in the index, since it's right at the start of the table; the second will store the picture size, and so on.

Now, while the index stores displacement, the subscript stores only the occurrence number—which entry this is—a number from one to the size of the table. This means that when COBOL is using a subscript to locate something, it must calculate the displacement freshly every time. Because of this, indexes are more efficient than subscripts.

FYI, a couple of points: math operations are not allowed on an index PSN in COBOL, because the program does not store the actual hexadecimal addresses. Second, an index is unique to the table it refers to and may not be used with any other tables in the program, but a subscript may be used in more than one table (though it usually isn't).

Binary SEARCHing

The SEARCH verb as it's been explained so far deals with searching a table entry by entry in sequence until a match is found or COBOL falls off the end of the table. Besides searching sequentially, COBOL is capable of doing binary searching. Binary searching means it cuts the table in half each time until it finds a match or not. The format for a binary SEARCH verb is only one word different from that for a sequential SEARCH. It's:

```
SEARCH ALL table-name
        AT END
          command(s)
        WHEN (*matches)
           command(s).
```

Binary searches also require the table be indexed, but with a little difference. You have to declare a key for the table when you define it and tell COBOL if the records are in ascending or descending order. You may conclude from this that if you're searching binarily, the table must be sorted. Right! And when you're doing a binary SEARCH, it makes sense you don't set the index first but let COBOL go where it needs to go.

Here is a sample declaration for a table you want to use a binary SEARCH on:

```
01 TABLE.
      05  ENTRY OCCURS n TIMES
                    INDEXED BY index-name
                    [ASCENDING] KEY IS FIRST-ITEM
                    [DESCENDING]
           10 FIRST-ITEM       PIC whatever
           10  SECOND-ITEM  PIC whatever.
```

Some Extra Stuff

PERFORM VARYING is used to move through a table by incrementing a counter a specified number of times. In this example, the number of new entries has already been counted and a value has been moved into NUM-ENTRIES. NEW-ENTRY-COUNTER will start at one and increment forward by 1 until it becomes equal to NUM-ENTRIES.

PERFORM VARYING EXAMPLE:

```
PERFORM  ADD-NEW-ENTRIES VARYING NEW-ENTRY-COUNTER
        FROM 1 BY 1 UNTIL NEW-ENTRY-COUNTER = NUM-ENTRIES
```

Level 66 RENAMES:

- ✓ Level 66 items must follow the last item in the record layout
- ✓ Level 66 items may overlap each other
- ✓ There is no limit to the number of 66's a record may have

Level 66 RENAMES Example:

```
01   PATIENT-RECORD.
     05   PATIENT-NAME.
          10   PN-LAST                        PIC X(15).
          10   PN-FIRST                       PIC X(10).
     05   PATIENT-ADDRESS.
          10   PA-STREET                      PIC X(15).
          10   PA-CITY                        PIC X(10).
          10   PA-STATE                       PIC X(2).
          10   PA-ZIP                         PIC X(5).
     05   PATIENT-ACCOUNT-NUM.
          10   PAN-PHYSICIAN-CODE             PIC X(4).
          10   PAN-INSURER-CODE               PIC X(4).
     05   OFFICE-LOCATION-NUMBER              PIC  X(2).
     66   MAILING-LABEL RENAMES PATIENT-NAME
                                     THRU PATIENT-ADDRESS.
     66   BILLING-ID RENAMES PA-ZIP
                                 THRU OFFICE-LOCATION-NUMBER.
```

Notice that PA-ZIP is included in both RENAMES clauses - this is acceptable. The group items MAILING-LABEL and BILLING-ID do not have their own PIC clauses - they are limited to the PIC clauses already defined for the included items. For example, BILLING-ID is limited to 15 characters.

COBOL Sub-programs

You do not have to write every function that you use in a program; in fact, many useful functions are already contained in files called ‘libraries’ that are stored elsewhere on the operating system. These functions, or subprograms, are brought into the main program only when they are needed and save the programmer the trouble of ‘reinventing the wheel’ to solve common programming problems. Using sub-programs also allows a large program to be parceled out to many different programmers in smaller, more manageable chunks that call each other from within the main program. Subprograms in libraries can be brought into the program in two ways - the entire routine can be placed in the main program using the verb COPY or the subprogram can be used from its current location by being ‘called’ by the main program.

COPY

COPY can be used to bring in other types of data from programs or libraries besides functions, such as record layouts. Record layouts can be described once and then reused over and over again by COPYing the description from one program to another. This saves a lot of tedious typing for the programmer. If the data description ever needs to be changed, it only has to be changed in the program referred to by the COPY statement, not in every program where it appears. The only restrictions on COPY are that the text being copied cannot contain another COPY statement and you cannot COPY into the IDENTIFICATION division. This example copies the entire contents of STUDENT-INFO-PROGRAM from a library and uses the verb REPLACING to change the name of STUDENT-ID with STUDENT-IDENTIFICATION-NUM. STUDENT-ID is NOT changed in the original program, only in the program which contained the COPY.

COPY with REPLACING Example:

```
COPY  STUDENT-INFO-PROGRAM IN STUDENT-PROGRAM-LIBRARY
          REPLACING STUDENT-ID BY STUDENT-IDENTIFICATION-NUM.
```

CALL

The CALL statement transfers control of the main program to a subprogram, which may in turn CALL other subprograms. The subprogram you are calling must already be compiled. Data is exchanged between the calling program and the called program using a parameter list (or 'arguments'). The program being called must have a LINKAGE SECTION in the DATA DIVISION to store its arguments. The LINKAGE SECTION may not contain VALUE clauses, except for level 88 condition-names, because values are given to it ('passed') by the calling program. The calling program defines its arguments in WORKING-STORAGE. The arguments in both programs must have identical PIC clauses (both size and type), but the names do not need to be the same. The EXIT PROGRAM statement returns control to the calling program.

In this example, the main program processes tuition bills by calling a subprogram, CALC-TUTION, to calculate the totals.

The main program's WORKING-STORAGE:

```
01  CURRENT-SEMESTER-DATA.
          05  NUMBER-OF-CREDIT-HOURS        PIC  99.
          05   RESIDENCY-STATUS             PIC X.
          05  CURRENT-TUTITION-DUE          PIC S9(5)V99
                                            VALUE ZERO.
```

The LINKAGE SECTION in CALC-TUITION:

```
01    BILLING-DATA
      05    HOURS-BILLED                     PIC 99.
      05    RESIDENCY-STATUS                 PIC X.
      05    TUTITION-DUE                     PIC S9(5)V99.
```

Notice that the items in both descriptions have identical PIC clauses…..

When the main program is ready to process a bill, it passes arguments to the subprogram with the verbs CALL and USING. NUMBER-OF-CREDIT-HOURS and RESIDENCY-STATUS got their values somewhere else in the main program, either by reading a file or by accepting user input.

CURRENT_TUTION_DUE will be passed back to the calling program by the subprogram when the total has been calculated. Error checking should be done on all values that are used as arguments before they are passed to the subprogram.

The subprogram is called by placing its PROGRAM-ID within single-quotes in the CALL statement:

```
CALL  'CALC-TUITION' USING   RESIDENCY-STATUS
              CURRENT-TUITION-DUE    NUMBER-OF-CREDIT-HOURS.
```

.

The PROCEDURE division in the subprogram lists the subprogram's arguments with the verb USING, as follows:

```
PROCEDURE DIVISION USING RESIDENCY-STATUS
                         TUITION-DUE    HOURS-BILLED.
```

Notice that the values do not have to be passed in the same order they are listed in WORKING-STORAGE, however, it is critical that the arguments are listed in the same order in both programs after the verb USING or the values that are passed will be placed in the wrong PSN's. For example, if you listed RESIDENCY-STATUS first in the calling program and then listed TUITION-DUE first in the subprogram, an alphanumeric value would be placed in TUITION-DUE and the subprogram would crash when it tried to do math with it.

Control passes to CALC-TUITION once the CALL statement is executed and it becomes the object program until it reaches the EXIT PROGRAM statement. In COBOL-74, EXIT PROGRAM must be the only statement in an EXIT paragraph, but in COBOL-85, it only needs to be the last statement in the procedure division. Do not use STOP RUN in a subprogram or everything (including the calling program) will come to a screeching halt!

The PROCEDURE division of the subprogram might look like:

```
IF RESIDENCY-STATUS = “I”
      MULTIPLY HOURS-BILLED BY INSTATE-FEE
                                                GIVING TUITION-DUE
ELSE
        IF RESIDENCY-STATUS = “O”
                  MULTIPLY HOURS=BILLED BY OUTSTATE-FEE
                                                  GIVING TUITION-DUE.
EXIT PROGRAM.
```

The value of TUTION-DUE is available to the calling program as CURRENT-TUITION-DUE as soon as control is passed back to it.

Structured Programming:

Once computing got going, hardware improved rapidly—computers got smaller and faster. No one thought much about programming because that was the cheap part. In the 50s and early 60s, 80% of the cost of maintaining a data processing operation was tied up in machines. But this situation was fast reversing. Now, the cost of the machines has become minimal compared to the cost of paying people to program them,

In the 50s what mattered to software was machine efficiency. Now, what matters is instead, "people efficiency", which means programs which are

1. reliable and
2. easy to maintain (and add to)

Poor reliability has hidden costs. Here is a list:

1.rerun the program

2.possibly recreate the data b/t the program has scrambled it

3. fires (fixes under pressure) create new bugs (badly fixed/not documented)

4. you're losing forward momentum by diverting staff time

5. you're wasting salary money (see above)

6. if you have unreliable systems you'll lose business

By the 60s software was in a mess. IBM had an operating system brought out with lots of fanfare that documented 100K bugs in the first year.

People had thought there could be no theory of programming b/c all languages are different. Then:

1. 1966, Corrado Bohm and Guiseppe Jacopini published a paper containing a mathematical proof that all computer programs could be written using three structures - - sequence, selection (alternation, or decision) and repetition (iteration, or looping). Each of these structures has one entry point and one exit point. This idea was critical to the development of SP.

2. 1968, Edsger Dijkstra published a paper "GO TO considered Harmful".

IBM used those papers among other things to experiment:

1. the superprogrammer experiment—Harlan Mills

2. New York Times Project—1969-71

85K plus lines, over half the programs correct the first run;max number of runs to fix those that weren't—3.

NY Times used s.p. theory. Also used a new management approach called Chief Programmer team.

Characteristics of traditional dp shop: headed by a non-programmer; large salary discrepancy between head and programmers; access to internal (on computer) and external (off) libraries not restricted; communications problems rampant; programs tested bottom-up (whole program)

Characteristics of CPT: head the best programmer there; head codes (not a pun); program librarian job created to deal with job submission (not necessary now) and retrieval; librarian controlled access to internal program library and kept on file the latest run of every program for every project; programs tested top-down (higher levels coded first, with lower levels left as stubs (paragraphs with minimal code) until high levels worked.

CPT also stressed documentation and used structured walkthroughs to clear code before it was entered into programs but this was general soon in all DP shops and doesn't specifically distinguish the 2 approaches.

3. Skylab—1973 100K lines of code, similar results

So what is Structured Programming?

"Top-down Segmentation"

Program is divided into segments/modules, each of which is limited in size—about 30 lines.

Modules are stacked in a hierarchy at the top of which is a driving segment/module which ultimately controls the execution of all the other modules.

Another important idea in structured programming is that segment has a single entry point and a single exit point. The importance of this point was emphasized by Bohm and Jacopini.

Interactive Programming

These days, programs are often run from screens. A screen is 24 positions deep by 80 positions wide. To define a screen you use the same approach you did when using a print chart; just write in the boxes what you want to appear.

There are two main verbs specific to interactive programming. These are ACCEPT and DISPLAY.

ACCEPT

This is used to collect input from the system, in the case of getting date and time, and also from the user.

The date/time format for ACCEPT using unix is:

```
ACCEPT psn FROM [DATE]
                [TIME].
```

The date is stored YYMMDD in the system, the time HHMMSS. You need to set up Working-Storage psns configured like that to use this format.

```
ACCEPT user-psn FROM LINE whatever COLUMN whatever.
```

You will need to create a user-psn in WORKING-STORAGE to hold the user input. The line and column numbers are wherever they fall on the layout you made before you started writing the program.

DISPLAY

This is used to put information on a screen. Its format will look rather familiar:

```
DISPLAY [psn] AT LINE whatever COLUMN whatever.
        [literal]
```

When you're programming interactively, you don't necessarily need Working-Storage line layouts. You could say

```
DISPLAY HEAD-1 AT LINE 1 COLUMN 1
```

But you could just as easily say

DISPLAY ***'Hi, welcome to my system!'*** AT LINE 1 COLUMN 1.
If you did that, COBOL would start in the leftmost position at the top of the screen and just keep going until the end of the literal was reached.

ACCEPT INPUT-ACCOUNT-NUM PROTECTED FROM LINE ____ COL ____ .

There are a number of enhancements to the ACCEPT and DISPLAY verbs provided with Program 4. You are responsible for those.