# COBOL

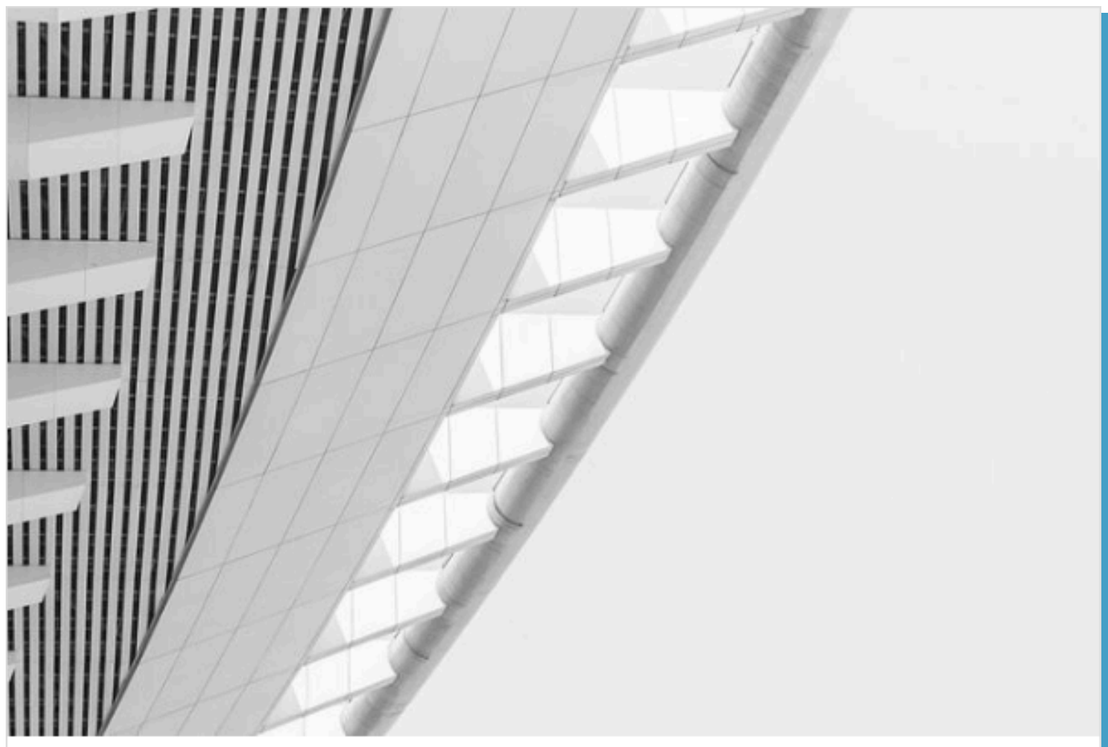
/ˈkoʊbɒl/

Karen Arteaga Mendoza | April, 2025

# COBOL

## *Introducction and Historical Context [1]*

In the late 1950s, as computing technology rapidly evolved, businesses faced a growing challenge: how to efficiently process large volumes of financial and administrative data. While early programming languages were primarily designed for scientific and mathematical applications, there was a pressing need for a language tailored specifically for business operations. In response, a committee of industry experts, government representatives, and academics gathered to develop a solution that would prioritize readability, maintainability, and long-term adaptability. The result was **COBOL (Common Business-Oriented Language)**, a programming language designed to bridge the gap between human communication and machine processing.



**What is COBOL?**

Common business-oriented language (COBOL) is a high-level, English-like, compiled programming language developed specifically for business data processing needs.

ibm.com / Dec 30, 2024

- Designed by CODASYL, Grace Hopper, Charles Phillips, Jean Sammet, and others. [2]
- Structured and modular programming

COBOL is one of the oldest high-level programming languages, designed specifically for business, finance, and administrative systems. Introduced in 1959, COBOL was developed as part of a collaborative effort led by the Conference on Data Systems Languages (CODASYL), with significant contributions from the U.S. Department of Defense, private industry, and academia. Its creation aimed to provide a standardized, English-like programming language that could be easily understood and used by professionals managing large-scale business operations.

At the time of COBOL's creation, computing was transitioning from machine-specific assembly languages to more portable, human-readable languages. The primary objective was to create a language that facilitated data processing in industries such as banking, insurance, and government operations, where handling vast amounts of transactional data was crucial.

HISTORY OF COBOL

# TIMELINE

## Historical Background [1]

COBOL emerged from a need for a universal programming language that could be used across different computer systems. Before its creation, business computing relied on various machine-specific languages, making software development and maintenance highly inefficient. In 1959, the U.S. Department of Defense gathered a committee known as the CODASYL (Conference on Data Systems Languages) to develop a common business-oriented language.

## Overview

The first COBOL specifications were completed in 1960, and the language was quickly adopted by both government agencies and businesses. Over the years, COBOL has undergone several updates, including COBOL-68, COBOL-74, COBOL-85, and the more recent COBOL 2002 and COBOL 2014 standards, which introduced features like object-oriented programming and improved data handling.

**1959** **Birth of COBOL** - The CODASYL is formed by the U.S. Department of Defense to create a standardized business programming language. COBOL specification is developed, with key contributions from Grace Hopper and others.

**1960s** **Early adoption and standarization** - The first official COBOL-60 program runs successfully in 1960. In 1961, COBOL is adopted by the U.S. Department of Defense, ensuring widespread government and industry use. In 1968, **COBOL-68**, the first official ANSI standard, is released, improving consistency across different systems.

**1970s** **Enhancements** - In 1974, **COBOL-74** is introduced, refining language features and improving standardization. In the late 1970s, COBOL becomes the dominant language for business data processing worldwide.

**1980s** **Structured programming** - COBOL-85 introduces structured programming features, making code more maintainable. In the late 1980s, COBOL continues to power major financial, government, and business systems.

**1990s** **Y2K Crisis** - The Y2K problem raises concerns due to date-handling limitations in old COBOL programs. COBOL programmers are in high demand to fix Y2K issues, reinforcing the language's continued importance.

**2000s** **Modernization** - COBOL 2002 introduces object-oriented programming (OOP) features, aligning with modern software development practices. COBOL 2014 standard is released, adding JSON support, dynamic memory allocation, and better interoperability with modern systems.

### *Syntax and structure [6]*

COBOL follows a hierarchical structure, where programs are divided into well-defined components. This structure ensures clarity, maintainability, and organization, which is critical for large-scale business applications. Unlike modern programming languages that use symbols and compact expressions, COBOL emphasizes an English-like syntax that improves code readability.

## Basic Program Structure

```
IDENTIFICATION DIVISION.
  PARAGRAPHS
    ENTRIES
      CLAUSES
ENVIRONMENT DIVISION.
  SECTIONS
    PARAGRAPHS
      ENTRIES
        CLAUSES
          PHRASES
DATA DIVISION.
  SECTIONS
    ENTRIES
      CLAUSES
        PHRASES
PROCEDURE DIVISION.
  SECTIONS
    PARAGRAPHS
      SENTENCES
        STATEMENTS
          PHRASES
```

```
* This is a comment line.

COMPUTE WS-A = WS-B * WS-C.
COMPUTE WS-A = WS-B + WS-C.
COMPUTE WS-A = WS-B - WS-C.
COMPUTE WS-A = WS-B / WS-C.

IF WS-A = 10.
MOVE 10 TO WS-A, WS-B.

01 WS-VAR PIC X(5) VALUE “HELLO”.
01 WS-VAR PIC X(5) VALUE ‘HELLO’.
```

A COBOL program is divided into four main divisions, each serving a specific purpose.

1. **Identification division:** Provides metadata about the program.
2. **Environment division:** Defines system specific configurations, such as file handling.
3. **Data division:** Declares variables, records and file structures.
4. **Procedure division:** Contains the executable logic of the program.

## Characters and Words

The most basic and indivisible unit of the COBOL language is the character. The basic character set includes the letters of the Latin alphabet, digits, and special characters.

COBOL recognizes the following character types:

- Alphabetic (A-Z, a-z)
- Numeric (0-9)
- Special characters ( +-*=, etc)

Individual characters are joined to form character-strings and separators, which then are used to form the words, literals, phrases, clauses, statements, and sentences that form the language.

A COBOL word consists of one or more characters and can be reserved words, user-defined words or literals. User-defined words include names for variables, files, procedures, etc. Literals refer to the fixed values enclosed in quotes commonly known as strings.

## Entries, Sentences and Statements

Sentences are subdivided into statements, and statements are subdivided into phrases. Entries are subdivided into clauses.

- **Entries**: An entry is a series of clauses that ends with a separator period. Entries are constructed in the identification, environment, and data divisions.
- **Clauses**: A clause is an ordered set of consecutive COBOL character-strings that specifies an attribute of an entry. Clauses are constructed in the identification, environment, and data divisions.
- **Sentences**: A sentence is a sequence of one or more statements that ends with a separator period. Sentences are constructed in the Procedure Division.
- **Statements**: A statement specifies an action to be taken by the program. Statements are constructed in the Procedure Division.
- **Phrases**: Each clause or statement in a program can be subdivided into smaller units called phrases.

```
DISPLAY “Enter your name:”. ----------> Statement ---|
ACCEPT EMP-NAME. ------> Statement                    |---> Sentence
DISPLAY “Welcome, ” EMP-NAME. ------> Statement   ---|
```

## Sections and Paragraphs

Sections and paragraphs define a program. Sections and paragraphs are subdivided into sentences, statements, and entries. A paragraph is a named block of sentences, it is defined in the Procedure Division and provides logical grouping. A section is a collection of paragraphs, mainly used in Data Division and Procedure Division to organize code.

```
PROCEDURE DIVISION.
MAIN-PROCESS.
  DISPLAY “Processing Data...”
  STOP RUN.
```

```
DATA DIVISION.
  WORKING-STORAGE SECTION.
```

## Identification Division

The first division in a COBOL program is the identification division. This division is mandatory and it provides essential details about the program, such as name, author, date written and a brief description. Within this division, COBOL programs need a PROGRAM-ID paragraph to function.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. YourProgramName.
AUTHOR. Your name.
DATE-WRITTEN. YYYY-MM-DD.
COMMENT. "Description of the program".
```

# Environment Division

The Environment Division defines the computer system and external files used by the program. It contains an optional **Configuration Section** that defines the system-specific settings and an **Input-Output Section** for file handling.

```
ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  SOURCE-COMPUTER. IBM-Z-SERIES.
  OBJECT-COMPUTER. IBM-Z-SERIES.
  INPUT-OUTPUT SECTION.
  FILE-CONTROL.
     SELECT EMP-FILE ASSIGN TO 'EMPLOYEE.DAT'.
```

# Data Division

The Data Division of a COBOL program declares all variables, constants and file structures. It describes all the data to be processed by the program, as well as the relationship between records. This division includes four main sections:

- **File Section:** externally stored data.
- **Working-Storage Section:** internal data.
- **Local-Storage Section:** internal data located on a per-invocation basis.
- **Linkage Section:** data available in another program.

```
DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 EMPLOYEE-DETAILS.
    05 EMP-ID       PIC 9(5) .
    05 EMP-NAME     PIC X(30).
    05 EMP-SALARY   PIC 9(7)V99.
```

```
DATA DIVISION.
   FILE SECTION.
   FD FILEN
   01 NAME PIC A(25)
```

Data is described using level numbers (01-49, 66, 77, 88) and PIC (Picture) clauses that define data types and sizes.

# Procedure Division

The Procedure Division contains the executable code (declaratives and procedures) that processes data. The structure of the Procedure Division is made up of paragraphs , sections, sentences and statements. Each paragraph or section can be given a name for modular control flow. Statements perform actions like DISPLAY, ACCEPT, IF, PERFORM, READ, ect.

The execution begins with the first statement in the Procedure Division, excluding declaratives. Statements are executed in order of compilation. The execution ends with STOP RUN, GO BACK od EXIT statements.

```
PROCEDURE DIVISION.
  A000-FIRST-PARA.
  DISPLAY ‘Hello World!’.
  MOVE ‘Tutorials’ TO WS-NAME.
  DISPLAY “My name is: “WS-NAME
  DISPLAY “My ID is: “WS-ID.
STOP RUN.
```

# Level Numbers

Level numbers indicate the hierarchy or nesting level of data items in the DATA DIVISION.
They show how data items relate to one another — whether a field is a top-level variable, a group of fields, or a component of a larger structure.

**Commonly Used Level Numbers**

| | |
|---|---|
| **01** | Top-level data item (often called a "record" or "group") |
| **05, 10, 15** | Subfields of the group; show nesting inside the 01-level item |
| **88** | Condition names (used to define named values for testing fields) |

# PIC Clause

One of the most important parts of COBOL programming is how you define data in the DATA DIVISION. COBOL uses something called the PIC (Picture) clause to describe what kind of data a variable will hold and how much space it needs. Learning how to read and write PIC clauses is essential for correctly managing input, output, file records, and calculations.

A PIC clause has two main parts:

- **Type Character(s):** A letter that defines what kind of data the field holds.
- **Size (inside parentheses):** A number that tells how many characters or digits the field can store.

The basic form is:

```
PIC <Type-Character>(<Size>)
```

For example, PIC X(30) means a field that can hold 30 alphanumeric characters, while PIC 9(5) means a field that can hold 5 numeric digits.

**Common Type Characters**

| | |
|---|---|
| **X** | Alphanumeric (letters, numbers, symbols) |
| **9** | Numeric digits only (0-9) |
| **A** | Alphabetic characters (A-Z) |
| **S** | Signed numeric (positive/negative) |
| **V** | Implied decimal point (only in numbers-9 |

### *A Complete COBOL Program*

This COBOL program, called EmployeeManager, prompts the user to enter an employee's ID, name, and salary, stores the inputs in working storage variables, and then displays the entered information back to the screen. Although it defines a file (EMPLOYEE.DAT) and a record structure (EMP-RECORD) for future file operations, in this version, it only performs user interaction and does not read from or write to the file. The program serves as a basic example of input, storage, and output in COBOL.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EmployeeManager.

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT EMP-FILE ASSIGN TO 'EMPLOYEE.DAT'.

DATA DIVISION.
FILE SECTION.
FD EMP-FILE.
01 EMP-RECORD.
   05 EMP-ID       PIC 9(5).
   05 EMP-NAME     PIC X(30).
   05 EMP-SALARY   PIC 9(7)V99.

WORKING-STORAGE SECTION.
01 WS-EMP-ID       PIC 9(5).
01 WS-EMP-NAME     PIC X(30).
01 WS-EMP-SALARY   PIC 9(7)V99.

PROCEDURE DIVISION.
MAIN-PROCESS.
    DISPLAY "Enter Employee ID:".
    ACCEPT WS-EMP-ID.
    DISPLAY "Enter Employee Name:".
    ACCEPT WS-EMP-NAME.
    DISPLAY "Enter Salary:".
    ACCEPT WS-EMP-SALARY.

    DISPLAY "Employee Details:".
    DISPLAY "ID: " WS-EMP-ID.
    DISPLAY "Name: " WS-EMP-NAME.
    DISPLAY "Salary: $" WS-EMP-SALARY.

    STOP RUN.
```

### *Other COBOL Features and Aspects*

Traditionally, COBOL programs have often been monolithic. Early COBOL programas were typically huge, monolithic blocks of code — long files where all the logic, data handling, and file processing were tightly packed together. Everything was procedural. Subroutines and modularity existed, but it was still often one giant program managing full business process from start to finish. Over time, Developers started using "modular" techniques inside COBOL — breaking programs into sections, paragraphs, and using PERFORM statements to reuse logic. Today COBOL can be fully modular and even object-oriented.

## Loops and Repetitions

Loops in COBOL are handled using a few structured programming constructs. While COBOL doesn't have traditional for or while loops like some other languages, it achieves repetition using keywords like PERFORM, UNTIL, and VARYING.

```
PERFORM <Paragraph-Name> UNTIL <Condition>.
```

This is the most common looping pattern in COBOL. You repeat a paragraph or block of code until a condition becomes true.

**Example.** Printing the value of COUNTER from 1 to 5.

```
WORKING-STORAGE SECTION.
01 COUNTER      PIC 9(2) VALUE 1.

PROCEDURE DIVISION.
    PERFORM DISPLAY-NUMBER UNTIL COUNTER > 5.
    STOP RUN.

DISPLAY-NUMBER.
    DISPLAY "Counter = " COUNTER.
    ADD 1 TO COUNTER.
```

If you want to execute something a fixed number of times, use the PERFORM <n> TIMES format.

```
PERFORM <Paragraph-Name> <n> TIMES.
```

# Error Detection

Error handling in COBOL is primarily done through conditional checks and special status codes provided by file operations, I/O operations, and external calls. Unlike modern languages that use try-catch blocks or exceptions, COBOL relies on built-in structures like FILE STATUS, AT END, INVALID KEY, and RETURN-CODE to manage and respond to errors.

## 1. File Handling

When working with files (sequential, indexed, or relative), you can define a FILE STATUS variable to monitor the result of each file operation.

**Declaration Example.** After each file operation (e.g., OPEN, READ, WRITE), WS-EMP-STATUS will contain a 2-character code that indicates success or a specific error.

```
FILE-CONTROL.
    SELECT EMP-FILE ASSIGN TO 'EMP.DAT'
        FILE STATUS IS WS-EMP-STATUS.
```

```
WORKING-STORAGE SECTION.
01 WS-EMP-STATUS     PIC XX.
```

**Example**

```
READ EMP-FILE
    AT END
        DISPLAY "End of file reached."
    NOT AT END
        DISPLAY "Record read.".
IF WS-EMP-STATUS NOT = "00"
    DISPLAY "Error: " WS-EMP-STATUS.
```

This small program fragment attempts to read one record from the file EMP-FILE.

- If it reaches the end of the file, it says so.
- If it successfully reads a record, it acknowledges it.
- If there’s any kind of file error (like file not found, bad read, etc.), it prints the error code for debugging or logging purposes.

**Common File Status Codes**

| | |
|---|---|
| **“00”** | Successful operation |
| **“10”** | End of file |
| **“23”** | Record not found (indexed) |
| **“35”** | File not found |
| **“39”** | File mismatch (access mode) |
| **“91”** | File already open or locked |

You can refer to your COBOL compiler documentation for the full list of codes.

## 2. Key Access Errors

When using indexed or relative files with key access (like READ or WRITE with keys), you can use the INVALID KEY clause.

**Example.** This prevents your program from crashing when a key operation fails.

```
READ EMP-FILE KEY IS EMP-ID
    INVALID KEY DISPLAY "Record not found".
```

```
WRITE EMP-RECORD
    INVALID KEY DISPLAY "Duplicate key — record already exists".
```

## 3. Loop Termination or Read Failures

AT END is used during sequential file reads to detect the end of file.

**Example.**

```
READ INPUT-FILE
    AT END
        SET EOF-FLAG TO TRUE
    NOT AT END
        PERFORM PROCESS-RECORD.
```

## 4. Program-Level Return Codes

You can set a value in the RETURN-CODE special register before exiting your program to indicate success or failure to the calling environment (e.g., a JCL job or shell script).

**Example.** On most systems, RETURN-CODE = 0 means success.

```
IF WS-EMP-STATUS NOT = "00"
    MOVE 1 TO RETURN-CODE
ELSE
    MOVE 0 TO RETURN-CODE.
```

# Object Oriented Programming [3]

Modern COBOL (starting officially from the COBOL 2002 standard) added OOP features like classes, objects, methods, inheritance, encapsulation and polymorphism.  This allows COBOL to behave similarly to Java, C++, and other OOP languages — although adoption in the real world has been relatively limited compared to procedural COBOL.

## 1. Classes

A class in COBOL defines the structure and behavior of objects. It’s similar to a "factory" or "blueprint" for creating objects.

**Declared using:**

```
CLASS-ID. ClassName.
```

**Example.**

```
CLASS-ID. EMPLOYEE.

DATA DIVISION.
WORKING-STORAGE SECTION.
01 EMP-ID      PIC 9(5).
01 EMP-NAME    PIC X(30).

PROCEDURE DIVISION.
METHOD-ID. Set-Details.
    ACCEPT EMP-ID.
    ACCEPT EMP-NAME.
END METHOD Set-Details.

METHOD-ID. Display-Details.
    DISPLAY "Employee ID: " EMP-ID.
    DISPLAY "Employee Name: " EMP-NAME.
END METHOD Display-Details.

END CLASS EMPLOYEE.
```

In this example, a class called EMPLOYEE is defined. This class introduces two data items: EMP-ID (to store an employee’s identification number) and EMP-NAME (to store the employee’s name). It contains two methods: Set-Details and Display-Details. The Set-Details method prompts the user to input an employee ID and name using the ACCEPTstatement, storing the inputs into the corresponding variables. At this stage, no actual execution occurs — the class is simply being defined like a blueprint, ready to be used by a separate program.

## 2. Objects

An object is an instance of a class.

**Declared and instantiated using:**

```
OBJECT REFERENCE OF ClassName
```

**Created dynamically with:**

```
INVOKE ClassName "NEW" RETURNING object-name
```

**Example.**

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MainProgram.

DATA DIVISION.
WORKING-STORAGE SECTION.
01 EMPLOYEE-OBJ OBJECT REFERENCE EMPLOYEE.

PROCEDURE DIVISION.
    INVOKE EMPLOYEE "NEW" RETURNING EMPLOYEE-OBJ
    INVOKE EMPLOYEE-OBJ "Set-Details"
    INVOKE EMPLOYEE-OBJ "Display-Details"
    STOP RUN.
```

In this example, the program MainProgram actually creates and uses an EMPLOYEE object. In the WORKING-STORAGE section, an object reference EMPLOYEE-OBJ is declared. The program uses the INVOKE statement to create a new instance of the EMPLOYEE class, assigning it to EMPLOYEE-OBJ. It then calls the Set-Details method, which asks the user to enter the employee ID and name, followed by calling Display-Details, which prints those values back to the screen. Essentially, this program brings the EMPLOYEE class to life by making a real object from it and interacting with it.

## 3. Methods

Methods define actions or behaviors inside a class. You can have multiple methods inside a class, and they can access and modify the class's data.

**Declared using:**

```
CLASS-ID. ClassName.
```

## 4. Inheritance

A class can inherit properties and behaviors from another class using the INHERITS clause.

**Example.** MANAGER inherits from EMPLOYEE, and adds a new property (DEPARTMENT-NAME).

```
CLASS-ID. MANAGER INHERITS EMPLOYEE.

DATA DIVISION.
WORKING-STORAGE SECTION.
01 DEPARTMENT-NAME PIC X(20).

PROCEDURE DIVISION.
METHOD-ID. Set-Department.
    ACCEPT DEPARTMENT-NAME.
END METHOD Set-Department.

END CLASS MANAGER.
```

This example demonstrates inheritance by creating a new class called MANAGER that inherits from the EMPLOYEE class. This means MANAGER automatically has access to everything defined in EMPLOYEE — both its data items (EMP-ID, EMP-NAME) and its methods (Set-Details, Display-Details). In addition to inheriting these features, MANAGER defines a new data item DEPARTMENT-NAME, and adds a new method Set-Department, which prompts the user to input a department name. In this way, MANAGER extends the functionality of EMPLOYEE by including department-related information.

## 5. Encapsulation and Polymorphism

Variables (data items) declared inside a class are typically accessed and modified through methods, not directly from outside the class. This protects the internal state of the object, which is a key principle of OOP. COBOL classes can specify PRIVATE or PUBLIC access to data and method.
COBOL allows method overriding in subclasses — meaning a subclass can change the behavior of a method inherited from a superclass. (However, polymorphism in COBOL is more limited compared to Java or Python.)

| Aspect | Procedural COBOL | Object Oriented COBOL |
|---|---|---|
| **Code organization** | Divisions,sections and paragraphs | Classes, methods and objects |
| **Reuse** | COPY Books, PERFORM sections | Inheritance and method overriding |
| **Control flow** | PERFORM and CALL paragraphs | INVOKE methods on objects |

### *Basic Full Program (Calculator Class)*

In this example, a simple Calculator class is defined, along with a main program that uses it. The Calculator class contains a single data item, RESULT, and a method Add that initializes RESULT to zero, adds 5 to it, and displays the final result. In the CalcProgram, an object reference CALC-OBJ is declared, and a new Calculator object is created. The program then calls the Add method, resulting in a display of “Result: 00005”. This illustrates the full cycle of defining a class, creating an object, and invoking methods on that object to perform an operation.

```
CLASS-ID. Calculator.

DATA DIVISION.
WORKING-STORAGE SECTION.
01 RESULT PIC 9(5).

PROCEDURE DIVISION.

METHOD-ID. Add.
    MOVE 0 TO RESULT.
    ADD 5 TO RESULT.
    DISPLAY "Result: " RESULT.
END METHOD Add.

END CLASS Calculator.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CalcProgram.

DATA DIVISION.
WORKING-STORAGE SECTION.
01 CALC-OBJ OBJECT REFERENCE Calculator.

PROCEDURE DIVISION.
    INVOKE Calculator "NEW" RETURNING CALC-OBJ
    INVOKE CALC-OBJ "Add"
    STOP RUN.
```

SUMMARY

# PROGRAMMING GUIDE

| | |
|---|---|
| **Basic units** | Characters, words and literals |
| **Statements and sentences** | Instructions for execution |
| **Paragraphs and sections** | Logical grouping of instructions |
| **Divisions** | The four major divisions that form a COBOL program |
| **Complete program** | Combining all the components to create a functional COBOL program |

## User-Defined Words

Names of variables, data items, paragraph labels, file names, etc.

- Can’t start with a number
- Can’t contain special characters

## System-Names

Predefined words that refer to system-specific resources, configurations or logical file names.

**Examples:**

- SOURCE-COMPUTER
- ASSIGN TO
- PRINTER

## Reserved Words

Keywords that are fixed parts of COBOL syntax. Can’t be used as variable names or identifiers. They define the structure and behavior of the language.

**Examples:**

- IF
- PERFORM
- MOVE
- STOP RUN
- ACCEPT
- DISPLAY

## Function-Names

Names of intrinsic functions built into COBOL. These functions perform operations like math, date handling and string manipulation.

**Examples:**

- FUNCTION NUMVAL
- FUNCTION LENGTH
- FUNCTION RANDOM

## Context-Sensitive Words

Words that have a meaning only in certain contexts. The same word might act differently depending on where or how it’s used in a statement.

**Examples:**

- THRU
- IS, OF, IN

# PROGRAMMING GUIDE

COBOL programs are organized into four major divisions.

| | |
|---|---|
| **Identification** | Program metadata |
| **Environment** | Defines hardware setup and file assignments |
| **Data** | Declares all data items, files and structures |
| **Procedure** | Contains the executable instructions |

Sentences end with a period (.), and English-like keywords make COBOL highly readable.

| **Concept** | **Example** |
|---|---|
| **MOVE** | MOVE 100 TO SALARY. |
| **DISPLAY** | DISPLAY “Hello World!”. |
| **ACCEPT** | ACCEPT EMP-ID. |
| **IF (condition)** | IF SALARY > 50000 DISPLAY “High Salary”. |
| **PERFORM** | PERFORM UNTIL COUNTER > 5. |

## Data Declarations

Data items are declared with a hierarchical level number and a PIC (Picture) clause describing their type and size

```
01 EMPLOYEE-DETAILS.
   05 EMP-ID     PIC 9(5).
   05 EMP-NAME   PIC X(30).
   05 EMP-SALARY PIC 9(7)V99.
```

- 9 means numeric.
- X means alphanumeric (text).
- V indicates an implied decimal point.

# PROGRAMMING GUIDE

## Error Checking

Files are handled via the ENVIRONMENT and DATA DIVISION with status tracking.

```
FILE-CONTROL.
SELECT EMP-FILE ASSIGN TO 'EMPLOYEE.DAT'
    FILE STATUS IS WS-EMP-STATUS.
```

- AT END checks if a file read hits end-of-file.
- INVALID KEY handles file write/read errors.
- FILE STATUS stores operation results ("00" = success).

## Object Oriented Extensions

COBOL 2002 and beyond support OOP features:

- Define classes with CLASS-ID.
- Create objects with INVOKE ClassName "NEW".
- Encapsulate data and methods inside objects.
- Support for inheritance and method overriding.

```
CLASS-ID. EMPLOYEE.
METHOD-ID. Display-Info.
    DISPLAY "Employee Info".
END METHOD Display-Info.
END CLASS EMPLOYEE.
```

## Conditional

```
IF A > B
    DISPLAY "A is greater."
ELSE
    DISPLAY "B is greater."
END-IF.
```

## Looping

```
PERFORM UNTIL COUNTER > 10
    DISPLAY COUNTER
    ADD 1 TO COUNTER
END-PERFORM.
```

## Simple File Read

```
READ EMP-FILE
    AT END
        SET EOF-FLAG TO TRUE
    NOT AT END
        PERFORM PROCESS-RECORD.
```

## Basic Flowchart

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
    [Executable Logic Here]
    STOP RUN.
```

## *COBOL Installation and Setup (with GnuCOBOL)*

COBOL is a compiled programming language. The COBOL program is written in a .cob or .cbl file. Then, a COBOL compiler converts the source code into object code. Several COBOL compilers exist, including:

- GnuCOBOL (Open-source)
- IBM Enterprise COBOL (Used on mainframes)
- Micro Focus COBOL (Widely used in businesses)
- Fujitsu COBOL (Popular in enterprise applications)

Because COBOL is compiled, it runs efficiently, especially for high-volume transaction processing in business environments.

To begin practicing COBOL on your own computer, you'll need a compiler. GnuCOBOL is a free, open-source COBOL compiler that allows you to write and run COBOL programs easily on Windows, Linux, or MacOS.

This section guides you step-by-step through installing and setting up GnuCOBOL.

# 1. What is GnuCOBOL?

GnuCOBOL (formerly OpenCOBOL) translates COBOL source code into C code, which is then compiled into machine language by a C compiler like GCC. It closely follows COBOL standards and is ideal for learning, development, and even production use.

GnuCOBOL supports:

- COBOL 85 syntax
- Most COBOL 2002 features
- Some COBOL 2014 extensions
- File handling, screen I/O, and even basic object-oriented features

# 2. Installing GnuCOBOL on Linux (Ubuntu/Debian)

On most Linux systems, installation is very simple:

```
sudo apt update
sudo apt install gnucobol
```

You can verify installation by typing:

```
cobc -v
```

# 3. Installing GnuCOBOL on MacOS (Homebrew)

You can use Homebrew to install GnuCOBOL easily:

```
brew install gnucobol
```

You can verify installation by typing:

```
cobc -v
```

# 4. Writing and Running Your First COBOL Program

After installing GnuCOBOL, you can immediately start writing and compiling COBOL programs!

**Step 1: Create a simple program**
Create a file called hello.cob with this content:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HelloWorld.

PROCEDURE DIVISION.
    DISPLAY "Hello, COBOL!".
    STOP RUN.
```

**Step 2: Compile the program**
Use the cobc compiler:

```
cobc -x hello.cob
```

the -x flag tells cobc to create an executable.
This will generate a file called hello (or hello.exe on Windows).

**Step 3: Run the program**
Run the executable:

```
./hello
```

You should see:

```
Hello, COBOL!
```

Learning COBOL isn't just about mastering a programming language — it's about connecting to one of the most important and enduring parts of computing history. To end this chapter, here are some fascinating facts, statistics, and useful tips about COBOL programming today.

## Fun Facts About COBOL

- **COBOL is older than the moon landing!**

COBOL was created in 1959. That's a full decade before humans landed on the moon in 1969.

- **Billions of lines still in use**

It is estimated that there are over 200 to 250 billion lines of COBOL code still running in production systems worldwide today.

- **COBOL runs the world's money**

According to Reuters and IBM reports, over 70% of global business transactions [1] — including banking, insurance, and government records — are processed using COBOL.

- **A huge COBOL workforce still exists**

It is estimated that around 2 million COBOL programmers are actively working today, although many are nearing retirement age. (This creates high demand for new COBOL-trained developers!)

- **Mainframes and COBOL are still modern**

Modern IBM mainframes can process billions of transactions per day, many of them running COBOL programs under the hood — connected to web apps, APIs, and mobile banking systems.

- **COBOL applications are immortal**

Some banking systems running today are based on COBOL programs written in the 1970s or 1980s, still operating reliably with only minor updates.

## Tips for Learning Programming in COBOL [5]

- **Think like a business analyst, not Just a programmer**

COBOL is about managing real-world business processes. Understand the "why" behind a program — payments, records, calculations — and your coding will be better.

- **Practice good structure**

COBOL loves clear organization: Divisions → Sections → Paragraphs → Sentences. Keep your programs neat and easy to follow.

- **Master the PERFORM statement**

Loops and modular control flows often rely on PERFORM. Learn PERFORM UNTIL, PERFORM VARYING, and PERFORM THRUearly — it's the heart of procedural COBOL.

- **Always check your file status**

When dealing with files, checking FILE STATUS after every I/O operation saves you from mysterious bugs and crashes.

- **Comment generously**

Even though COBOL is English-like, large programs can get complex fast. Use * comments often to explain what each paragraph or section does.

- **Leverage modern tools**

Use modern editors like Visual Studio Code (with a COBOL extension) or GnuCOBOL compilers to practice more efficiently on today's computers.

- **Stay curious about mainframes**

Even if you start on GnuCOBOL, understanding IBM mainframes, CICS (Customer Information Control System), or DB2 databases can greatly expand your career opportunities.

COBOL is not just an old language — it is a living, breathing foundation of modern business systems.
By learning COBOL, you're joining a lineage of developers who keep essential industries running safely, reliably, and efficiently. And because fewer new programmers are learning COBOL, your skills could be in higher demand than ever.

Welcome to the world of COBOL — where your code might quietly run behind millions of daily transactions, decades into the future!

## COBOL Fun Challenge: Build Your Own Mini Payroll System!

Now that you know the basics of COBOL, it's time to put your skills into action with a real-world inspired project. This mini-challenge will help you practice working with data input, calculations, conditionals, and structured output — just like a real business application.

**Challenge Objective**

Create a COBOL program that calculates employee paychecks based on hours worked and hourly wage.

Your program should:

- Accept employee name, hourly wage, and hours worked.
- Calculate total pay (wage × hours).
- Apply overtime rules (extra pay if more than 40 hours are worked).
- Display a formatted "pay stub" summary for each employee.

**Requirements**

1. Ask for Input:
    - Employee Name (text)
    - Hours Worked (number)
    - Hourly Wage (number)

2. Overtime Calculation:
    - If Hours Worked > 40:
        - Overtime hours are paid at 1.5 times the hourly wage.
    - Otherwise, regular pay applies.

3. Display Output:
    - Show something like:

```
Employee Name: John Doe
Regular Hours: 40
Overtime Hours: 5
Total Pay: $725.00
```

4. Optional Extra Credit:
    - Allow multiple employees to be processed (using a loop).
    - Write the pay stubs to a simple text file instead of just displaying them.

**Tips to Help You**

- Use WORKING-STORAGE to define your variables.
- Use IF and COMPUTE statements to handle the overtime logic.
- Use DISPLAY to format the output nicely.
- Try organizing your code into paragraphs (e.g., INPUT-PARAGRAPH, CALCULATION-PARAGRAPH, OUTPUT-PARAGRAPH).
- Add small comments to explain each section.

## *Bibliography*


[1] D. Martin, "How COBOL Still Powers the Global Economy at 60 Years Old," Texas Public Radio, 2019. [Online]. Available: https://web.archive.org/web/20190524035248/https:/www.tpr.org/post/how-cobol-still-powers-global-economy-60-years-old. [Accessed: 27-Apr-2025].

[2] IBM, "COBOL," IBM Think, [Online]. Available: https://www.ibm.com/think/topics/cobol. [Accessed: 27-Apr-2025].

[3] COBOL Standards Committee, "Welcome to COBOL Standards Information," COBOL Standard Info, 2017. [Online]. Available: https://web.archive.org/web/20170110012927/http:/www.cobolstandard.info/. [Accessed: 27-Apr-2025].

[4] IBM, "Character Sets and Code Pages," IBM Documentation for COBOL for z/OS 6.3, [Online]. Available: https://www.ibm.com/docs/en/cobol-zos/6.3.0?topic=structure-character-sets-code-pages. [Accessed: 27-Apr-2025].

[5] D. McCracken, A Simplified Guide to Structured COBOL Programming, Wiley, 1988. [Online]. Available: https://archive.org/details/simplifiedguidet0000mccr. [Accessed: 27-Apr-2025].

[6] TutorialsPoint, "COBOL - Program Structure," TutorialsPoint, [Online]. Available: https://www.tutorialspoint.com/cobol/cobol_program_structure.htm. [Accessed: 27-Apr-2025].