PROGRAMMING

COBOL Fundamentals

Real history and why COBOL still runs today, the four divisions, PICTURE clauses and data types, MOVE/COMPUTE arithmetic, IF/EVALUATE/PERFORM control flow, tables and the OCCURS clause, real string handling, sequential file I/O, and subprograms via CALL — closing with a complete, working batch processing program tying every chapter together.

10 Chapters | Philip Osztromok | Generated with Claude

CHAPTER 1: WHY COBOL? REAL HISTORY & WHY IT'S STILL RUNNING TODAY

COBOL Fundamentals

Chapter 1 · Why COBOL? Real History & Why It's Still Running Today

COBOL stands for **Common Business-Oriented Language** — a name that describes its entire real design philosophy in three words. It was never meant to be a general-purpose language for every kind of programming; it was built, deliberately, for one specific real purpose: processing business data at scale, in a form readable enough that a manager, not just a programmer, could follow the logic.

The Real 1959 CODASYL Committee

COBOL's design began in **1959**, driven by the **CODASYL** committee (Committee on Data Systems Languages) — a real, deliberate effort to create one standard business language that could run on machines from competing manufacturers, rather than locking a business into one vendor's own proprietary language. A six-person subcommittee did most of the real design work:

William Selden & Gertrude Tierney
IBM

Howard Bromberg & Howard Discount
RCA

Vernon Reeves & Jean E. Sammet
Sylvania Electric Products

Three competing manufacturers, working together on one shared standard — a genuinely unusual real arrangement for the era, and a direct, deliberate answer to the "vendor lock-in" problem business computing faced at the time.

Two Real, Different Influences

COBOL wasn't invented from nothing — it deliberately combined ideas from two existing real languages. **FLOW-MATIC**, on which Grace Hopper served as technical adviser, contributed long, readable variable names, English-word commands, and the real, foundational idea of

separating data descriptions from instructions — exactly the split that would later become COBOL's own DATA DIVISION and PROCEDURE DIVISION. **COMTRAN** contributed real arithmetic formula support, an improved IF statement, and — directly relevant to Chapter 3 of this course — the **PICTURE clause** itself.

> **A REAL, IMPORTANT CORRECTION** Grace Hopper is very often popularly credited as "the creator of COBOL" — but this isn't accurate. She served as a real technical adviser, and FLOW-MATIC's own influence on COBOL was genuine and significant, but she was not part of the six-person committee that actually wrote the specification. Jean E. Sammet — who *was* on that committee — stated this directly herself: Hopper "was not the mother, creator, or developer of Cobol." Hopper's own real, substantial contributions to computing (including popularizing the very idea of a compiler) stand on their own, without needing this specific, inaccurate credit attached to them.

COBOL 60: A Real, Specific Date

The finished specification was approved by the executive committee on **8 January 1960**, then sent to the US Government Printing Office, which published it as *COBOL 60* — the real, original standard every later COBOL revision built on.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO-WORLD.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
    DISPLAY "HELLO, WORLD".
    STOP RUN.
```

Even this simplest possible real COBOL program already shows the shape Chapter 2 covers in full — four named divisions, appearing in a fixed order, each with a distinct real job.

Why This Matters Today, Not Just Historically

COBOL's real, continued relevance is genuinely striking. A **1997** estimate put roughly **200 billion lines** of COBOL in active existence, running an estimated **80%** of all business programs at the time. More recently, a real **2020** statistic found that COBOL processed **95%** of the time a

credit or debit card was swiped. Banking, insurance, and major government agencies — including the IRS, the US Treasury, and the VA — still run real, live COBOL systems today.

WHY THIS SHAPES THE WHOLE COURSE Because of this real, specific history, almost nobody learns COBOL today to write brand-new systems from scratch. The real, practical demand is for people who can **read, understand, and safely maintain** decades-old COBOL already running in production — which is exactly why this course's own companion, COBOL Intermediate/Advanced, closes with a capstone on modernizing legacy code rather than building something new.

Hands-On Exercises

EXERCISE 1

Using this chapter's own real material, explain in your own words why crediting Grace Hopper as "the creator of COBOL" is inaccurate, and what her real, documented role actually was.

View solution

EXERCISE 2

Using this chapter's own real material on FLOW-MATIC and COMTRAN, explain in your own words which specific real COBOL feature traces to which of the two influences, and why.

View solution

EXERCISE 3

A friend asks why anyone would still bother learning a language designed in 1959. Using this chapter's own real, current-relevance statistics, write an accurate answer, in your own words.

View solution

CHAPTER 1 QUICK REFERENCE

- **COBOL** = Common Business-Oriented Language; design began 1959 via the CODASYL committee
- **Six real designers**: Selden & Tierney (IBM), Bromberg & Discount (RCA), Reeves & Sammet (Sylvania)
- **Two real influences**: FLOW-MATIC (readable syntax, data/instruction separation) and COMTRAN (formulas, the PICTURE clause)
- **Real myth correction**: Grace Hopper advised on FLOW-MATIC but, per Jean Sammet's own words, was not COBOL's creator
- **COBOL 60** — approved 8 January 1960
- **Real current relevance**: ~200 billion lines (1997 estimate), 80% of business programs, 95% of card swipes processed by COBOL (2020)

CHAPTER 2: COBOL PROGRAM STRUCTURE: THE FOUR DIVISIONS

COBOL Fundamentals

Chapter 2 · COBOL Program Structure: The Four Divisions

Chapter 1's own "Hello, World" example already showed the shape every COBOL program follows: four named divisions, in a fixed real order. This chapter covers what each one actually does — and the real, distinctive column layout that governs exactly where each line of COBOL is allowed to start.

The Real Column Layout

Traditional COBOL source code is **fixed-format** — a real, direct legacy of the punch-card era, where each physical column on a card had a fixed real meaning.

COLUMNS	REAL PURPOSE
1–6	Sequence number area — originally card/line numbers; ignored by the compiler today
7	Indicator area — `*` or `/` for a comment line, `-` for continuation, `D` for debug lines
8–11	Area A — DIVISION/SECTION headers, procedure (paragraph) headers, and level numbers 01 and 77
12–72	Area B — everything else
73+	Program name area — historically extended to column 80 on physical punch cards

THE REAL, SIMPLE RULE Division headers, section headers, procedure headers, and 01/77-level items must start in Area A. Everything else — including most DATA DIVISION entries and every PROCEDURE DIVISION statement — must start in Area B. This is exactly why COBOL source code has that distinctive, deeply indented look.

NOT THE ONLY REAL OPTION ANYMORE COBOL 2002 introduced a genuine free-format alternative — code can be placed in any column, comments use `*>` instead of column 7's indicator, and a

`>>PAGE` directive replaces the old `/` page-break indicator. Fixed-format remains extremely common in real legacy code, which is why this course teaches it as the default.

IDENTIFICATION DIVISION

The one real, required entry here is **PROGRAM-ID**, naming the program — already used in Chapter 1's own example. Older COBOL also supported several optional documentation-only paragraphs (AUTHOR, INSTALLATION, DATE-WRITTEN among them), historically useful for recording who wrote a program and when; these were later removed from the standard, and PROGRAM-ID is genuinely the only entry a modern program needs here.

ENVIRONMENT DIVISION

This division describes real, machine-dependent details — everything the program needs to know about the system it runs on, split into two real sections:

CONFIGURATION SECTION

Real, variable features like currency signs, locale, and character sets.

INPUT-OUTPUT SECTION

Real, file-related information — the FILE-CONTROL entries that Chapter 8's own file-handling material builds on directly.

DATA DIVISION

This is where every piece of data a program works with gets described — real COBOL splits it into six possible sections, though most programs (including everything in this Fundamentals course) only need the first two:

WORKING-STORAGE SECTION

Static, program-owned variables — the section used constantly throughout this course.

FILE SECTION

Record layouts for files the program reads or writes — Chapter 8's own territory.

LINKAGE SECTION

LOCAL-STORAGE, REPORT & SCREEN

Parameters and return values passed between programs — Chapter 9's own real CALL material.

Real, more specialized sections — automatic (reentrant) variables, structured report layouts (COBOL Intermediate/Advanced Chapter 5), and interactive screen definitions (COBOL Intermediate/Advanced Chapter 7).

PROCEDURE DIVISION

This is where the real, executable logic lives — organized into named **paragraphs** (and, in larger programs, sections grouping paragraphs together), each of which can act as a real label or a simple subroutine. Chapter 5's own real **PERFORM** verb is exactly how one paragraph invokes another, the closest COBOL comes to a function call within a single program.

Putting It Together: A Real, Slightly Expanded Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. GREETING.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-GREETING PIC X(20) VALUE "HELLO, COBOL".
PROCEDURE DIVISION.
    DISPLAY WS-GREETING.
    STOP RUN.
```

Notice `WS-GREETING` is declared once, in WORKING-STORAGE (DATA DIVISION), then simply referenced by name in PROCEDURE DIVISION — the exact real split FLOW-MATIC first introduced, covered back in Chapter 1. Chapter 3 covers exactly what `PIC X(20)` means and why.

Hands-On Exercises

EXERCISE 1

Using this chapter's own real column-layout table, explain in your own words why `PROGRAM-ID.` must start in Area A, while a line like `DISPLAY WS-GREETING.` must start in Area B.

View solution

EXERCISE 2

A colleague says, "The DATA DIVISION and the PROCEDURE DIVISION are really just two halves of the same thing." Using this chapter's own real material, explain in your own words why they actually serve two genuinely different, separate purposes.

View solution

EXERCISE 3

Using this chapter's own real material on the DATA DIVISION's six sections, explain in your own words which two this Fundamentals course focuses on, and name one real, later chapter each of the remaining sections connects to.

View solution

CHAPTER 2 QUICK REFERENCE

- **Fixed-format columns**: 1–6 sequence, 7 indicator, 8–11 Area A (headers, 01/77 levels), 12–72 Area B (everything else); COBOL 2002 also allows free-format
- **IDENTIFICATION DIVISION** — PROGRAM-ID is the one real required entry
- **ENVIRONMENT DIVISION** — CONFIGURATION SECTION (locale/currency) + INPUT-OUTPUT SECTION (files)
- **DATA DIVISION** — six real sections; this course focuses on WORKING-STORAGE and FILE, with LINKAGE, LOCAL-STORAGE, REPORT, and SCREEN covered or referenced later
- **PROCEDURE DIVISION** — real, executable paragraphs; PERFORM (Chapter 5) invokes one from another

CHAPTER 3: DATA TYPES & PICTURE CLAUSES

COBOL Fundamentals

Chapter 3 · Data Types & PICTURE Clauses

Chapter 2's own `01 WS-GREETING PIC X(20) VALUE "HELLO, COBOL".` line already used three real pieces this chapter covers properly: a level number, a PICTURE clause, and a VALUE clause. Chapter 1 also already traced the PICTURE clause's own real origin to COMTRAN — this chapter shows exactly what it can express, and one real, easy-to-miss detail about how COBOL actually stores the numbers behind it.

Level Numbers: Building Hierarchical Records

COBOL data items are organized hierarchically using real, numbered levels — an item with a higher level number is subordinate to an item with a lower one.

LEVEL	REAL MEANING
01	Begins a record definition — the top of a hierarchy
05, 10, 15…	Nested subordinate items underneath an 01 (or another subordinate level)
77	A standalone field with no subordinate items at all
88	A condition name — a real, named shortcut for testing whether another field holds a specific value, covered fully in Chapter 5

The Real PICTURE Clause Symbols

SYMBOL	REAL MEANING
9	A decimal digit (0–9)
X	Any character — alphanumeric
A	Alphabetic characters only (letters or spaces)

SYMBOL	REAL MEANING
V	The implied decimal point's location — not actually stored
S	Marks the item as signed (positive or negative)

A count like `9(5)` means "five real occurrences of 9" — five digits — the same shorthand Chapter 2's own `X(20)` already used for twenty characters.

A Real, Easy-to-Miss Detail: The Implied Decimal Point

Take `PIC 9(5)V9(2)` — a real, valid field with 5 digits before the decimal and 2 after, 7 digits total. Storing the value **12345.67** in it does *not* store a decimal point character anywhere.

> **WHAT'S ACTUALLY IN MEMORY** Internally, this field genuinely holds just the digit sequence `1234567` — seven raw digits, no decimal point. The `V` only tells COBOL where the decimal point *belongs* when the value is interpreted or displayed; it never occupies real, physical storage space of its own.

> **WHY THIS MATTERS FOR READING LEGACY CODE** This is one of the first real surprises many newcomers hit reading existing COBOL: a field's own raw stored digits will never contain a decimal point, even when the PICTURE clause clearly implies one. Arithmetic on the field works correctly regardless — COBOL applies the implied decimal position automatically — but a raw dump of the underlying storage will never show it.

USAGE Clauses: How Data Is Actually Stored

A field's **USAGE** clause controls its real, physical storage format — separate from what its PICTURE clause describes.

USAGE	REAL STORAGE FORMAT	WHY IT'S USED
DISPLAY (default)	One byte per character/digit, human-readable	Simple, but real, genuinely wasteful for large numeric fields
COMP / COMPUTATIONAL	Often equivalent to native binary storage	Real, faster arithmetic and more compact storage

USAGE	REAL STORAGE FORMAT	WHY IT'S USED
COMP-3 / PACKED-DECIMAL	Packed binary-coded decimal — two digits per byte	Real mainframe storage efficiency, still extremely common in legacy business data

On real mainframes, where storage space directly affected cost, COMP-3 in particular became genuinely standard for business numeric fields — expect to see it constantly once Chapter 8 starts reading real file records.

Putting It Together

```
01 WS-CUSTOMER-RECORD.
   05 WS-CUSTOMER-ID     PIC 9(6).
   05 WS-CUSTOMER-NAME   PIC X(30).
   05 WS-BALANCE         PIC S9(7)V99 COMP-3 VALUE 0.
```

One 01-level record, three real subordinate 05-level fields: a plain numeric ID, a text name, and a signed, two-decimal-place balance stored efficiently as packed decimal — a real, small-scale preview of exactly the kind of record layout Chapter 8's own file-handling material builds on directly.

Hands-On Exercises

EXERCISE 1

Using this chapter's own real material, explain in your own words what the raw stored digits would be for the value 4210.5 in a field defined as `PIC 9(5)V9(1)`, and why a decimal point never appears among those stored digits.

View solution

EXERCISE 2

A colleague says, "COMP-3 and PIC are basically the same kind of thing — they both describe a field's data type." Using this chapter's own real material, explain in your own words why this isn't accurate.

View solution

EXERCISE 3

Using this chapter's own real worked WS-CUSTOMER-RECORD example, explain in your own words the real hierarchical relationship between the 01-level item and each of the three 05-level items beneath it.

View solution

CHAPTER 3 QUICK REFERENCE

- **Level numbers**: 01 (record) → 05/10/15… (subordinate) · 77 (standalone) · 88 (condition name, Chapter 5)
- **PICTURE symbols**: 9 (digit) · X (any character) · A (alphabetic) · V (implied decimal, not stored) · S (signed)
- **Real gotcha**: `PIC 9(5)V9(2)` holding 12345.67 stores just `1234567` — no decimal point character, ever
- **USAGE**: DISPLAY (default, human-readable) · COMP/COMPUTATIONAL (binary) · COMP-3/PACKED-DECIMAL (packed BCD, real mainframe storage efficiency)

CHAPTER 4: WORKING WITH VARIABLES: MOVE, COMPUTE & ARITHMETIC VERBS

COBOL Fundamentals

Chapter 4 · Working with Variables: MOVE, COMPUTE & Arithmetic Verbs

Chapter 3 built the WS-CUSTOMER-RECORD structure but never actually changed any of its values. This chapter covers COBOL's own real verbs for doing that — copying data with MOVE, and every real way COBOL performs arithmetic.

MOVE: Real Alignment Rules Worth Knowing

`MOVE` copies a value from one field into another — but exactly how it lines the data up depends on the field's own real type, and getting this wrong is a genuinely common source of legacy-code confusion.

FIELD TYPE	REAL ALIGNMENT RULE
Alphanumeric (PIC X)	Left-justified — extra space on the right is padded with blanks; if the destination is shorter, extra characters on the right are truncated
Numeric (PIC 9)	Aligned on the implied decimal point — digits are padded with zeros or truncated at whichever end is needed, never simply left- or right-justified the way text is

WHY THIS GENUINELY SURPRISES NEWCOMERS Moving `"HI"` into a `PIC X(5)` field gives `"HI "` — padded on the right. But moving a value from a `PIC 9(3)V99` field into a shorter `PIC 9(2)V9` field aligns on the decimal point first, then truncates the leftmost integer digit and the rightmost decimal digit — never simply "the first few characters." Text and numbers genuinely follow different real rules.

The Real Arithmetic Verbs

COBOL provides four dedicated real verbs, each with its own genuine syntax variations:

```
ADD A TO B.                         *> B = B + A
ADD A B GIVING C.                   *> C = A + B (A, B unchanged)
SUBTRACT A FROM B.                  *> B = B - A
```

```
SUBTRACT A FROM B GIVING C.          *> C = B - A
MULTIPLY A BY B.                     *> B = A * B
MULTIPLY A BY B GIVING C.            *> C = A * B
DIVIDE A INTO B.                     *> B = B / A
DIVIDE A BY B GIVING C.              *> C = A / B
DIVIDE A BY B GIVING C REMAINDER D.  *> C = quotient, D = remainder
```

Every one of these real forms can add **ROUNDED** and **ON SIZE ERROR** — two genuine safety clauses worth knowing from the start.

ROUNDED

Rounds the result to fit the receiving field's own real PICTURE clause, instead of silently truncating extra decimal places.

ON SIZE ERROR

Runs a real block of statements if the result genuinely doesn't fit the receiving field (overflow, or division by zero) — without it, an overflow can silently produce a wrong, truncated value instead.

COMPUTE: A Real, More Flexible Alternative

Rather than chaining several separate arithmetic verbs together, **COMPUTE** accepts a genuine arithmetic expression directly — real operators `+`, `-`, `*`, `/`, and `**` (exponentiation), combined using standard real operator precedence, with parentheses to override it exactly as in ordinary mathematical notation.

```
COMPUTE WS-TOTAL = (WS-PRICE * WS-QTY) + WS-SHIPPING.
COMPUTE WS-BALANCE ROUNDED = WS-BALANCE + WS-INTEREST
    ON SIZE ERROR
        DISPLAY "BALANCE OVERFLOW"
END-COMPUTE.
```

Both ROUNDED and ON SIZE ERROR work identically here, applying to the entire computed expression's own final result.

Arithmetic Verbs vs. COMPUTE: When to Use Which

	ADD/SUBTRACT/MULTIPLY/DIVIDE	COMPUTE
Best for	A single, simple real operation	Any real, multi-step expression
Readability	Reads close to plain English	Reads like ordinary math notation
Real, common legacy usage	Still extremely common in older code	Preferred for anything beyond one operation

Hands-On Exercises

EXERCISE 1

Using this chapter's own real alignment rules, explain in your own words what value would result from moving the alphanumeric value `"COBOL"` into a field defined as `PIC X(3)`, and why the result differs from what happens with a numeric MOVE of the same length mismatch.

View solution

EXERCISE 2

A colleague says, "ON SIZE ERROR and ROUNDED do basically the same job — they both handle a result that doesn't fit." Using this chapter's own real material, explain in your own words why this isn't accurate.

View solution

EXERCISE 3

Rewrite this chapter's own real `COMPUTE WS-TOTAL = (WS-PRICE * WS-QTY) + WS-SHIPPING.` statement using only the dedicated ADD and MULTIPLY verbs instead. Explain, in your own words, why the COMPUTE version is genuinely easier to read for this specific calculation.

View solution

CHAPTER 4 QUICK REFERENCE

- **MOVE** — alphanumeric: left-justified, right-padded/truncated; numeric: decimal-point-aligned, padded/truncated at either end
- **Real arithmetic verbs**: ADD/SUBTRACT/MULTIPLY/DIVIDE, each with a GIVING form (and DIVIDE's own real REMAINDER)
- **ROUNDED** — rounds instead of silently truncating; **ON SIZE ERROR** — catches genuine overflow/division-by-zero
- **COMPUTE** — real +, -, *, /, ** operators with standard precedence and parentheses, for any multi-step expression

CHAPTER 5: CONTROL FLOW: IF, EVALUATE & PERFORM

COBOL Fundamentals

Chapter 5 · Control Flow: IF, EVALUATE & PERFORM

Chapter 3 introduced 88-level condition names and deferred their real use to this chapter. This chapter delivers on that — real IF logic, COBOL's own switch-equivalent EVALUATE, and every real form PERFORM takes, from a single subroutine call to a full counted loop.

IF Statements

```
IF WS-BALANCE > 0
    DISPLAY "ACCOUNT IN CREDIT"
ELSE
    DISPLAY "ACCOUNT NOT IN CREDIT"
END-IF.
```

Real, ordinary relation conditions (`>` , `<` , `=` , and their combinations) work exactly as expected, closed with the real `END-IF` scope terminator.

Condition Names (88-Levels), Finally Put to Use

Recall Chapter 3's own real 88-level condition names. Given a declaration like:

```
05 WS-ACCOUNT-STATUS PIC X(1).
    88 WS-STATUS-ACTIVE VALUE "A".
    88 WS-STATUS-CLOSED VALUE "C".
```

a real condition name can be tested directly, instead of comparing the underlying field to a literal:

```
IF WS-STATUS-ACTIVE
    DISPLAY "PROCESSING ACTIVE ACCOUNT"
```

```
END-IF.
```

> **WHY THIS IS GENUINELY WORTH USING** `IF WS-STATUS-ACTIVE` and `IF WS-ACCOUNT-STATUS = "A"` do exactly the same real thing — but the condition-name version reads as a genuine business concept ("is this account active?") rather than a raw literal comparison, and if the underlying stored code for "active" ever changed, only the 88-level's own VALUE clause needs updating, not every IF statement that checks it throughout a real program.

EVALUATE: COBOL's Real Switch Equivalent

Introduced in **COBOL-85**, `EVALUATE` is COBOL's real answer to a switch/case statement — and genuinely more flexible than IF/ELSE chains for handling several possible values.

```
EVALUATE WS-ACCOUNT-STATUS
    WHEN "A"
        DISPLAY "ACTIVE"
    WHEN "C"
        DISPLAY "CLOSED"
    WHEN OTHER
        DISPLAY "UNKNOWN STATUS"
END-EVALUATE.
```

`WHEN OTHER` is the real catch-all, matching anything not covered by an earlier WHEN. EVALUATE can also test genuine `TRUE` / `FALSE` conditions rather than only literal values — real, direct condition testing, not just value matching.

PERFORM: Simple Invocation & THRU

The simplest real form of `PERFORM` invokes one named paragraph and returns:

```
PERFORM PARA-VALIDATE.
```

A real `THRU` form executes a whole sequence of adjacent paragraphs, in order:

```
PERFORM PARA-A THRU PARA-C.
```

PERFORM UNTIL & Inline PERFORM

`PERFORM ... UNTIL` is COBOL's real conditional loop, and — also introduced in COBOL-85 — an **inline PERFORM** lets the loop body sit directly inside the statement itself, closed with `END-PERFORM`, with no separate paragraph needed:

```
PERFORM UNTIL WS-EOF-FLAG = "Y"
    DISPLAY "PROCESSING..."
    MOVE "Y" TO WS-EOF-FLAG
END-PERFORM.
```

PERFORM VARYING & PERFORM TIMES

`PERFORM VARYING` is COBOL's real counted loop — the closest equivalent to a for-loop in more modern languages — automatically increasing or decreasing an index each pass:

```
PERFORM VARYING WS-I FROM 1 BY 1 UNTIL WS-I > 10
    DISPLAY WS-I
END-PERFORM.
```

`PERFORM ... TIMES` repeats a fixed real number of times instead of testing a condition:

```
PERFORM 5 TIMES
    DISPLAY "HELLO"
END-PERFORM.
```

> **A REAL, DOCUMENTED LIMIT** The TIMES phrase can genuinely repeat up to **999,999,999** times — a real, documented ceiling, not an arbitrary round number, worth knowing exists even though almost no real program ever approaches it.

Hands-On Exercises

EXERCISE 1

Using this chapter's own real 88-level example, explain in your own words why `IF WS-STATUS-ACTIVE` is genuinely preferable to `IF WS-ACCOUNT-STATUS = "A"`, beyond simply being shorter to type.

View solution

EXERCISE 2

A colleague says, "PERFORM UNTIL and PERFORM VARYING do the same job — they're both loops." Using this chapter's own real material, explain in your own words the specific real difference between them, and when you'd genuinely choose one over the other.

View solution

EXERCISE 3

Rewrite this chapter's own real EVALUATE example (testing WS-ACCOUNT-STATUS for "A" and "C") as an equivalent IF/ELSE IF chain. Explain, in your own words, one genuine real advantage EVALUATE has over the IF/ELSE IF version once a third or fourth status code needs handling.

View solution

CHAPTER 5 QUICK REFERENCE

- **IF/ELSE/END-IF** — real relation conditions; 88-level condition names can be tested directly (e.g. `IF WS-STATUS-ACTIVE`)
- **EVALUATE** (COBOL-85) — real switch equivalent; `WHEN OTHER` catch-all; can test TRUE/FALSE conditions, not just values
- **PERFORM paragraph** / **PERFORM a THRU b** — real subroutine-style invocation
- **PERFORM UNTIL** / inline PERFORM (COBOL-85) — real conditional loop

- **PERFORM VARYING … FROM … BY … UNTIL** — real counted loop; **PERFORM n TIMES** — real fixed repetition, up to 999,999,999

CHAPTER 6: WORKING WITH TABLES (ARRAYS) & THE OCCURS CLAUSE

COBOL Fundamentals

Chapter 6 · Working with Tables (Arrays) & the OCCURS Clause

Every field so far has held exactly one value. Real COBOL programs constantly need many values of the same shape — twelve months of sales figures, a hundred customer records held in memory at once — and `OCCURS` is the real clause that makes a single data item repeat into a table.

OCCURS: Defining a Table

Reusing Chapter 5's own real counted-loop variable, `WS-I`, here's a real one-dimensional table of twelve monthly sales figures:

```
01 WS-MONTHLY-SALES.
    05 WS-SALES-AMT PIC 9(6)V99 OCCURS 12 TIMES.
```

`OCCURS 12 TIMES` tells COBOL to reserve twelve real copies of `WS-SALES-AMT`, one after another in storage. A single element is referenced by writing a subscript directly in parentheses after the item's name:

```
PERFORM VARYING WS-I FROM 1 BY 1 UNTIL WS-I > 12
    ADD WS-SALES-AMT(WS-I) TO WS-YEAR-TOTAL
END-PERFORM.
```

Subscripts Start at 1, Not 0

> **A GENUINE, EASY TRAP COMING FROM MOST OTHER LANGUAGES** COBOL tables are real **1-based**, not 0-based. `WS-SALES-AMT(1)` is January, the genuine first element — not January minus one. Chapter 5's own `PERFORM VARYING WS-I FROM 1 BY 1 UNTIL WS-I > 12` example above already assumes this directly: it starts `WS-I` at 1 and stops once it exceeds 12, the real count of elements — no adjustment needed, because COBOL never had a zeroth element to begin with.

INDEXED BY: A Real, More Efficient Alternative to a Subscript

A plain numeric field like `WS-I` works as a subscript, but COBOL also offers a genuinely different mechanism: an **index**, declared with `INDEXED BY` directly on the table itself. Extending Chapter 3's own real `WS-CUSTOMER-RECORD` into a table of a hundred customers:

```
01 WS-CUSTOMER-TABLE.
    05 WS-CUSTOMER-ENTRY OCCURS 100 TIMES
        INDEXED BY WS-CUST-IDX.
        10 WS-CUST-ID    PIC 9(6).
        10 WS-CUST-NAME  PIC X(30).
```

> **WHY AN INDEX IS GENUINELY FASTER, NOT JUST A STYLE CHOICE** A plain subscript like `WS-I` holds an ordinary occurrence number (1, 2, 3, ...) that the runtime has to multiply by the element's own size every single time it's used to find the real storage address. A real index, by contrast, is stored directly as a **displacement** — it moves by the element's own size each step, not by a plain count of 1 — so the runtime can use it straight away with no multiplication needed. On a table accessed millions of times in a real batch job, that's a genuine, measurable difference, not a cosmetic one.

An indexed table is set and stepped with real `SET` statements rather than ordinary arithmetic:

```
SET WS-CUST-IDX TO 1.
SET WS-CUST-IDX UP BY 1.
```

SEARCH: Real Sequential Table Lookup

`SEARCH` walks an indexed table one element at a time, starting from wherever the index currently points — which is exactly why a real `SET ... TO 1` normally comes first, to guarantee the search starts at the beginning rather than partway through:

```
SET WS-CUST-IDX TO 1
SEARCH WS-CUSTOMER-ENTRY
    AT END
        DISPLAY "CUSTOMER NOT FOUND"
    WHEN WS-CUST-ID(WS-CUST-IDX) = 402118
```

```
        DISPLAY "FOUND: " WS-CUST-NAME(WS-CUST-IDX)
END-SEARCH.
```

`AT END` is the real clause that fires only if the whole table is walked with no match found. `SEARCH` genuinely requires the table to have been declared with `INDEXED BY` in the first place — it's the index itself that `SEARCH` advances automatically on every failed comparison.

SEARCH ALL: Real Binary Search

`SEARCH ALL` is a genuinely different algorithm — a real binary search, jumping to the midpoint of the remaining range each time rather than checking one element after another. That real speed comes with a real requirement: the table's own definition must declare which field it's sorted by, using `ASCENDING KEY` or `DESCENDING KEY`:

```
01 WS-CUSTOMER-TABLE.
    05 WS-CUSTOMER-ENTRY OCCURS 100 TIMES
        ASCENDING KEY IS WS-CUST-ID
        INDEXED BY WS-CUST-IDX.
        10 WS-CUST-ID    PIC 9(6).
        10 WS-CUST-NAME  PIC X(30).
```

```
SEARCH ALL WS-CUSTOMER-ENTRY
    AT END
        DISPLAY "CUSTOMER NOT FOUND"
    WHEN WS-CUST-ID(WS-CUST-IDX) = 402118
        DISPLAY "FOUND: " WS-CUST-NAME(WS-CUST-IDX)
END-SEARCH.
```

A REAL, SILENT FAILURE MODE Declaring `ASCENDING KEY IS WS-CUST-ID` is a genuine promise to the compiler, not an instruction that sorts anything. If the real data actually loaded into the table isn't genuinely in ascending order by `WS-CUST-ID`, `SEARCH ALL` doesn't detect the mismatch — it just follows the binary-search algorithm anyway, and a real entry that's genuinely present in the table can come back as "not found," or the wrong entry can be returned, with no error raised anywhere.

Multi-Dimensional Tables

An `OCCURS` item can itself contain another `OCCURS` item, producing a genuine multi-dimensional table — here, quarterly sales figures across four real regions:

```
01 WS-REGIONAL-SALES.
    05 WS-REGION OCCURS 4 TIMES.
        10 WS-QUARTER-SALES PIC 9(7)V99 OCCURS 4 TIMES.
```

Referencing one specific element needs both real subscripts at once, separated by a space inside one set of parentheses — the region subscript first, then the quarter:

```
ADD WS-QUARTER-SALES(WS-REGION-IDX WS-QTR-IDX) TO WS-GRAND-TOTAL.
```

OCCURS DEPENDING ON: Real Variable-Length Tables

Every table so far has had a fixed, unchanging size. `OCCURS DEPENDING ON` lets a table's real size vary at runtime instead, between a stated minimum and maximum, driven by the current value of another field:

```
01 WS-ORDER-LINES.
    05 WS-LINE-COUNT     PIC 9(3).
    05 WS-ORDER-LINE OCCURS 1 TO 50 TIMES
        DEPENDING ON WS-LINE-COUNT
        PIC X(20).
```

Here, an order can genuinely hold anywhere from 1 to 50 line items, and the true, current number in use is tracked in `WS-LINE-COUNT` — a real space-saving mechanism for records where the count of repeating items is itself unknown until the program actually runs.

Hands-On Exercises

EXERCISE 1

Using this chapter's own real `WS-MONTHLY-SALES` example, explain in your own words why `WS-SALES-AMT(1)`, not `WS-SALES-AMT(0)`, is the correct reference for January's real figure, and why this genuinely matters for someone coming from a language where arrays start at 0.

View solution

EXERCISE 2

Using this chapter's own real material, explain in your own words the specific, genuine reason an `INDEXED BY` index is faster than an ordinary numeric subscript like `WS-I`, and why `SEARCH` specifically requires a table to be declared with `INDEXED BY` in the first place.

View solution

EXERCISE 3

A colleague declares `WS-CUSTOMER-ENTRY OCCURS 100 TIMES ASCENDING KEY IS WS-CUST-ID`, then loads real customer records into the table in the exact order they arrive from an unsorted file, without ever sorting them first. Using this chapter's own real material, explain in your own words what could genuinely go wrong the next time the program runs `SEARCH ALL` against that table.

View solution

CHAPTER 6 QUICK REFERENCE

- **OCCURS n TIMES** — defines a real table with n repeating elements; subscripts are **1-based**, never 0
- **Subscript** — a plain numeric field (e.g. `WS-I`) in parentheses after the item name
- **INDEXED BY** — a real, genuinely faster displacement-based alternative to a subscript; set/advanced with `SET`, not ordinary arithmetic
- **SEARCH** — real sequential search from the index's current position; requires `INDEXED BY`; `AT END` for no match
- **SEARCH ALL** — real binary search; requires `ASCENDING KEY`/`DESCENDING KEY` declared *and* data genuinely sorted that way, or results are silently wrong
- **Multi-dimensional tables** — `OCCURS` nested inside `OCCURS`; multiple subscripts are space-separated in one set of parentheses

- **OCCURS m TO n TIMES DEPENDING ON** — a real variable-length table, sized at runtime by another field's current value

CHAPTER 7: STRING HANDLING: STRING, UNSTRING & REFERENCE MODIFICATION

COBOL Fundamentals

Chapter 7 · String Handling: STRING, UNSTRING & Reference Modification

MOVE copies a whole field. Real COBOL programs constantly need something finer-grained — building one field out of several pieces, splitting one field apart into several, or reaching into the middle of a field without touching the rest. Four real verbs cover that: STRING, UNSTRING, reference modification, and INSPECT.

STRING: Real Concatenation

Extending Chapter 3's own real `WS-CUSTOMER-RECORD`, `STRING` builds one output field out of several source pieces at once — genuinely more direct than a chain of separate MOVE statements:

```
01 WS-REPORT-LINE PIC X(50).

STRING "CUSTOMER: " DELIMITED BY SIZE
       WS-CUSTOMER-NAME DELIMITED BY SPACE
       " (ID: " DELIMITED BY SIZE
       WS-CUSTOMER-ID DELIMITED BY SIZE
       ")" DELIMITED BY SIZE
    INTO WS-REPORT-LINE
END-STRING.
```

Each source piece gets its own real `DELIMITED BY` clause, and the two forms genuinely differ:

- `DELIMITED BY SIZE` — transfers the entire sending field, exactly as declared, with no stopping point
- `DELIMITED BY SPACE` (or any other literal/field) — stops the moment that real delimiter is reached, so a name field padded with trailing spaces doesn't drag its own padding into the result

A real optional `POINTER` phrase can control exactly where in the receiving field `STRING` starts writing:

```
STRING "MORE TEXT" DELIMITED BY SIZE
    INTO WS-REPORT-LINE
    WITH POINTER WS-POS
END-STRING.
```

A REAL, DOCUMENTED REQUIREMENT EASY TO MISS A `POINTER` field genuinely must be initialized to a real value before `STRING` ever runs — 1, to start writing from the very beginning of the receiving field. Skip that initialization and the pointer holds whatever value was already sitting in storage, so `STRING` genuinely starts writing from wherever that leftover value happens to point, silently overwriting the middle of the field instead of building it from the start.

UNSTRING: Real Splitting

`UNSTRING` does the reverse — one real source field split apart into several destination fields, wherever a chosen delimiter appears:

```
01 WS-FULL-NAME    PIC X(30) VALUE "JANE MARY DOE".
01 WS-FIRST-NAME   PIC X(15).
01 WS-MIDDLE-NAME  PIC X(15).
01 WS-LAST-NAME    PIC X(15).

UNSTRING WS-FULL-NAME DELIMITED BY SPACE
    INTO WS-FIRST-NAME
         WS-MIDDLE-NAME
         WS-LAST-NAME
END-UNSTRING.
```

Real `DELIMITED BY` treats each occurrence of the given delimiter as one real split point, checked left to right through the source field. Two further real phrases capture extra detail about each split as it happens:

- `DELIMITER IN` — stores which real delimiter actually triggered each individual split (genuinely useful when more than one possible delimiter is allowed)
- `COUNT IN` — stores the real number of characters that were examined for each individual destination field

Like `STRING`, `UNSTRING` also accepts a real `WITH POINTER` phrase to control where scanning of the source field begins.

Reference Modification: Real Substrings

Introduced in **COBOL-85** — the same real standard that added Chapter 5's own `EVALUATE` and inline `PERFORM` — reference modification reaches directly into the middle of a field with no separate STRING or UNSTRING statement needed at all, using a real colon-separated syntax directly after the field's own name:

```
MOVE WS-CUSTOMER-NAME(1:5) TO WS-NAME-PREFIX.
```

`(1:5)` means "start at position 1, take 5 characters" — and, matching Chapter 6's own real 1-based subscripting, position 1 is genuinely the field's first real character, never a zeroth one. The length is real but optional — leaving it off extends the real reference all the way to the end of the field:

```
MOVE WS-CUSTOMER-NAME(6:) TO WS-NAME-REMAINDER.
```

INSPECT: Counting & Replacing

`INSPECT` examines a field's own real content without moving it anywhere — counting how often something appears (`TALLYING`) or swapping characters in place (`REPLACING`):

```
INSPECT WS-REPORT-LINE TALLYING WS-SPACE-COUNT FOR ALL SPACES.
```

A real `REPLACING` phrase substitutes matching characters, one for one, directly inside the existing field:

```
INSPECT WS-PHONE-NUMBER REPLACING ALL "-" BY SPACE.
```

REAL COUNTING/REPLACING MODES Both `TALLYING` and `REPLACING` accept real `ALL` (every occurrence), `LEADING` (only contiguous occurrences starting from the very beginning of the field), and — for `REPLACING` specifically — `FIRST` (only the single leftmost occurrence). `TALLYING ... CHARACTERS` counts every real character in the field regardless of content.

Hands-On Exercises

EXERCISE 1

Using this chapter's own real STRING example, explain in your own words the genuine difference between `DELIMITED BY SIZE` and `DELIMITED BY SPACE`, and why `WS-CUSTOMER-NAME` specifically uses the SPACE form rather than SIZE.

View solution

EXERCISE 2

Using this chapter's own real warn-box, explain in your own words what genuinely happens if a program uses `STRING ... WITH POINTER WS-POS` but never initializes `WS-POS` to 1 beforehand, and why the resulting bug is a real, silent one rather than something COBOL would catch on its own.

View solution

EXERCISE 3

Using this chapter's own real UNSTRING example, explain in your own words what would genuinely happen if `WS-FULL-NAME` held only two words instead of three (e.g. "JANE DOE"), given that UNSTRING has three destination fields waiting to receive values.

View solution

CHAPTER 7 QUICK REFERENCE

- **STRING … INTO** — real concatenation; each source needs its own `DELIMITED BY SIZE` (whole field) or `DELIMITED BY` a stopping value
- **UNSTRING … INTO** — real splitting on a delimiter; `DELIMITER IN` / `COUNT IN` capture per-split detail
- **WITH POINTER** (STRING/UNSTRING) — must be genuinely initialized (typically to 1) before use, or the starting position is whatever leftover value was already in storage
- **Reference modification** (COBOL-85) — `field(start:length)`; 1-based; length is real but optional, defaulting to the rest of the field
- **INSPECT … TALLYING** — real counting; **INSPECT … REPLACING** — real in-place character substitution; both support ALL/LEADING/CHARACTERS

CHAPTER 8: BASIC FILE HANDLING: SEQUENTIAL FILES

COBOL Fundamentals

Chapter 8 · Basic File Handling: Sequential Files

Every real record processed so far has lived only in `WORKING-STORAGE`, gone the moment the program ends. Real COBOL programs exist to read and write actual files — and a sequential file is COBOL's own most basic, most common shape: records genuinely contiguous, read one after another from the front, "similarly to a linked list."

Three Places a File Touches a Program

A real COBOL file spans three separate divisions, each doing a genuinely different job:

DIVISION	WHAT IT DECLARES
ENVIRONMENT DIVISION	Which real external file this program talks to (`SELECT` / `ASSIGN`)
DATA DIVISION	What one real record inside that file looks like (the `FD` and its record layout)
PROCEDURE DIVISION	The real verbs that actually move data in and out (`OPEN`, `READ`, `WRITE`, `CLOSE`)

SELECT & FD: Declaring the File

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT CUSTOMER-FILE ASSIGN TO "CUSTFILE.DAT"
        ORGANIZATION IS SEQUENTIAL
        FILE STATUS IS WS-FILE-STATUS.

DATA DIVISION.
FILE SECTION.
FD  CUSTOMER-FILE
    LABEL RECORDS ARE STANDARD.
01  CUSTOMER-RECORD.
```

```
    05 CUST-ID       PIC 9(6).
    05 CUST-NAME     PIC X(30).
    05 CUST-BALANCE  PIC S9(7)V99.
```

This chapter's real `CUSTOMER-RECORD` deliberately mirrors Chapter 3's own `WS-CUSTOMER-RECORD` — a real record layout doesn't change shape just because it's now coming from a file instead of being built by hand in `WORKING-STORAGE`.

OPEN: Real Access Modes

MODE	WHAT IT GENUINELY ALLOWS
INPUT	Reading only, from a file that already exists
OUTPUT	Writing only; a real, existing file's contents are replaced entirely
I-O	Both reading and writing on the same open file (not valid for a real line-sequential file)
EXTEND	Writing new records appended after whatever a real file already contains

```
OPEN INPUT CUSTOMER-FILE.
```

READ ... AT END & the Real Priming-Read Pattern

A real `READ` pulls the next record in, and its `AT END` phrase fires the moment a read genuinely goes past the last record in the file:

```
READ CUSTOMER-FILE
    AT END MOVE "Y" TO WS-EOF-FLAG
END-READ.
```

WHY A SINGLE READ INSIDE A LOOP'S CONDITION DOESN'T WORK `READ` is a real statement, not an expression a loop condition can call directly. Chapter 5's own `PERFORM UNTIL` can't say "loop until the next READ hits end-of-file" in one step — it needs a genuine value to test. The standard real fix is the **priming read**: read once before the loop starts, then read again as the very last thing the loop

body does, so `WS-EOF-FLAG` always reflects the most recent real read by the time the loop condition is checked again.

```
READ CUSTOMER-FILE
    AT END MOVE "Y" TO WS-EOF-FLAG
END-READ.

PERFORM UNTIL WS-EOF-FLAG = "Y"
    DISPLAY CUST-NAME
    READ CUSTOMER-FILE
        AT END MOVE "Y" TO WS-EOF-FLAG
    END-READ
END-PERFORM.
```

WRITE: Real Records, Not Files

```
OPEN OUTPUT CUSTOMER-FILE.
MOVE 100234 TO CUST-ID.
MOVE "SMITH, JOHN" TO CUST-NAME.
MOVE 500.00 TO CUST-BALANCE.
WRITE CUSTOMER-RECORD.
```

A GENUINELY EASY MISTAKE `WRITE` always names the real 01-level record — `CUSTOMER-RECORD` — never the file itself. Writing `WRITE CUSTOMER-FILE` is a genuine compile error: COBOL writes whatever record is currently sitting in that record's own storage, and the file name never appears as the object being written.

CLOSE

```
CLOSE CUSTOMER-FILE.
```

A real, plain `CLOSE` releases the file — every file genuinely opened in a program needs a matching close before the program ends.

FILE STATUS: Real Result Codes

I/O status codes were a real COBOL-85 addition — the same standard that added Chapter 5's `EVALUATE`, Chapter 7's reference modification, and inline PERFORM. A real two-character `FILE STATUS` field, declared in `WORKING-STORAGE` and named in the file's own `SELECT` clause, is set after every single file operation:

CODE	REAL MEANING
00	Operation genuinely successful
10	End of file reached — the real code behind every `AT END` trigger
23	Record not found (indexed/relative files)
35	OPEN INPUT/I-O attempted against a file that genuinely doesn't exist

```
OPEN INPUT CUSTOMER-FILE.
IF WS-FILE-STATUS NOT = "00"
    DISPLAY "ERROR OPENING FILE: " WS-FILE-STATUS
    STOP RUN
END-IF.
```

Hands-On Exercises

EXERCISE 1

Using this chapter's own real material, explain in your own words why COBOL needs a priming read before a PERFORM UNTIL loop rather than simply writing PERFORM UNTIL the next READ reaches end-of-file directly in the loop's own condition.

View solution

EXERCISE 2

Using this chapter's own real material, explain in your own words what would genuinely happen if a program opened CUSTOMER-FILE with OPEN OUTPUT when the goal was actually to read the file's existing records.

View solution

EXERCISE 3

Using this chapter's own real material, explain in your own words why WS-FILE-STATUS being "00" after a READ statement genuinely can't be relied on forever within a real priming-read loop, and what real status value eventually replaces it.

View solution

CHAPTER 8 QUICK REFERENCE

- **SELECT ... ASSIGN TO** (ENVIRONMENT DIVISION) — names the real external file; **FD** (DATA DIVISION) — describes one real record's layout
- **OPEN INPUT/OUTPUT/I-O/EXTEND** — real access modes; OUTPUT genuinely replaces existing content
- **READ ... AT END** — fires once a read genuinely passes the last record; needs a real **priming read** before a PERFORM UNTIL loop
- **WRITE** — always names the real 01-level record, never the file itself
- **CLOSE** — every genuinely opened file needs a matching close
- **FILE STATUS** (COBOL-85) — real 2-character result code after every file operation; **00** success, **10** end of file, **35** file not found on OPEN

CHAPTER 9: SUBPROGRAMS & THE CALL STATEMENT

COBOL Fundamentals

Chapter 9 · Subprograms & the CALL Statement

Every real program written so far has been one self-contained unit. Real COBOL systems are routinely built from many separately compiled programs calling one another — `CALL` is the real statement that hands control from one to the next, and back again.

CALL: Static vs. Dynamic

A real `CALL` comes in two genuinely different forms:

```
CALL "BALCHECK" USING WS-BALANCE WS-VALID-FLAG.
```

A literal name in quotes, like `"BALCHECK"` above, is a real **static call** — the specific subprogram is fixed at compile time. A real **dynamic call** uses a data item instead of a literal, so which program actually runs is only decided while the program is genuinely running:

```
MOVE "BALCHECK" TO WS-PROGRAM-NAME.
CALL WS-PROGRAM-NAME USING WS-BALANCE WS-VALID-FLAG.
```

Passing Parameters: BY REFERENCE, BY CONTENT & BY VALUE

MODE	WHAT GENUINELY HAPPENS
BY REFERENCE (real default)	The called program's item occupies the exact same real storage as the caller's — a change made inside the subprogram genuinely changes the caller's own data
BY CONTENT	The called program receives a real copy; nothing it does can change the caller's original data
BY VALUE	Passes the real value itself rather than any storage reference at all, working on a genuine temporary copy

A GENUINELY EASY ASSUMPTION TO GET WRONG Because `BY REFERENCE` is the real default whenever no keyword is written at all, a subprogram that only reads a parameter to make a decision can still accidentally overwrite the caller's genuine data with one careless MOVE — there's no automatic protection unless `BY CONTENT` or `BY VALUE` is deliberately specified.

The LINKAGE SECTION: Receiving Parameters

A called subprogram doesn't declare its incoming parameters in `WORKING-STORAGE`. It declares them in a real, separate `LINKAGE SECTION` — real storage "for parameters and the return value," genuinely matched by position, not by name, against the caller's own `CALL ... USING` list:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. BALCHECK.

DATA DIVISION.
LINKAGE SECTION.
01 LS-BALANCE     PIC S9(7)V99.
01 LS-VALID-FLAG  PIC X(1).

PROCEDURE DIVISION USING LS-BALANCE LS-VALID-FLAG.
    IF LS-BALANCE < 0
        MOVE "N" TO LS-VALID-FLAG
    ELSE
        MOVE "Y" TO LS-VALID-FLAG
    END-IF.
    GOBACK.
```

NAMES DON'T HAVE TO MATCH — POSITION DOES The caller's field is `WS-BALANCE`; the subprogram's own matching field is `LS-BALANCE` — genuinely different names. What has to match is the real **order** and **number** of items: the first item in `PROCEDURE DIVISION USING` receives whatever the first item in `CALL ... USING` was, regardless of what either one is actually called.

Returning Control: GOBACK's Real Two Faces

`GOBACK` genuinely behaves differently depending on where it's used — it doesn't always mean the same thing:

WHERE GOBACK APPEARS	WHAT IT GENUINELY DOES
In a called subprogram	Behaves like `EXIT PROGRAM` — returns control to the caller, right after the CALL statement
In a main program	Behaves like `STOP RUN` — ends the whole application

`STOP RUN` itself is never appropriate inside a real subprogram meant to return — it would end the entire application rather than hand control back to whatever called it.

Hands-On Exercises

EXERCISE 1

Using this chapter's own real BALCHECK example, explain in your own words why WS-VALID-FLAG in the calling program genuinely ends up holding "Y" or "N" after the CALL, even though the subprogram itself only ever assigns a value to LS-VALID-FLAG.

View solution

EXERCISE 2

A colleague writes CALL "BALCHECK" USING WS-VALID-FLAG WS-BALANCE, reversing the real order from this chapter's own example, without changing BALCHECK's own LINKAGE SECTION or PROCEDURE DIVISION USING at all. Using this chapter's own real material, explain in your own words what genuinely goes wrong.

View solution

EXERCISE 3

Using this chapter's own real material, explain in your own words the genuine difference between using GOBACK inside BALCHECK versus accidentally writing STOP RUN there instead.

View solution

CHAPTER 9 QUICK REFERENCE

- **CALL "literal"** — real static call, fixed at compile time; **CALL identifier** — real dynamic call, decided at runtime
- **BY REFERENCE** (real default) — shared real storage, changes propagate back; **BY CONTENT** — a real copy, caller's data untouched; **BY VALUE** — a real value copy, no storage reference at all
- **LINKAGE SECTION** — where a subprogram declares its incoming parameters, matched by real position (not name) to `CALL ... USING`
- **PROCEDURE DIVISION USING** — names the LINKAGE SECTION items a called program actually receives, in order
- **GOBACK** — real context-dependent: acts like `EXIT PROGRAM` in a subprogram, like `STOP RUN` in a main program

CHAPTER 10: CAPSTONE — WRITING A COMPLETE BATCH PROCESSING PROGRAM

COBOL Fundamentals

Chapter 10 · Capstone: Writing a Complete Batch Processing Program

Nine chapters built individual real pieces — divisions, PICTURE clauses, arithmetic, control flow, tables, string handling, file I/O, subprograms. Real COBOL work is almost always exactly this shape combined into one program: a nightly batch job that reads a transaction file, updates an in-memory customer table, and reports what happened. This capstone builds one, **TXNBATCH**, using every real piece from Chapters 1 through 9.

The Scenario

TXNBATCH reads a real sequential transaction file — one record per deposit or withdrawal, each carrying a customer ID, a transaction code, an amount, and a composite reference/branch string — validates each amount, applies it to an in-memory table of customer balances, and prints a real, running report line for every transaction processed.

File & Data Declarations

```
IDENTIFICATION DIVISION.
PROGRAM-ID. TXNBATCH.

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT TRANSACTION-FILE ASSIGN TO "TXNFILE.DAT"
        ORGANIZATION IS SEQUENTIAL
        FILE STATUS IS WS-FILE-STATUS.

DATA DIVISION.
FILE SECTION.
FD  TRANSACTION-FILE
    LABEL RECORDS ARE STANDARD.
01  TRANSACTION-RECORD.
    05 TXN-CUSTOMER-ID  PIC 9(6).
    05 TXN-CODE         PIC X(1).
```

```
    05 TXN-AMOUNT       PIC S9(7)V99.
    05 TXN-DETAIL       PIC X(20).

WORKING-STORAGE SECTION.
01  WS-FILE-STATUS   PIC X(2).
01  WS-EOF-FLAG      PIC X(1) VALUE "N".
01  WS-VALID-FLAG    PIC X(1).
01  WS-REF-NUMBER    PIC X(10).
01  WS-BRANCH-CODE   PIC X(10).
01  WS-REPORT-LINE   PIC X(60).
01  WS-GRAND-TOTAL   PIC S9(9)V99 VALUE 0.
01  WS-DEPOSIT-COUNT PIC 9(5) VALUE 0.
01  WS-WITHDRAW-COUNT PIC 9(5) VALUE 0.

01  WS-CUSTOMER-TABLE.
    05 WS-CUSTOMER-ENTRY OCCURS 100 TIMES
        ASCENDING KEY IS WS-CUST-ID
        INDEXED BY WS-CUST-IDX.
        10 WS-CUST-ID       PIC 9(6).
        10 WS-CUST-NAME     PIC X(30).
        10 WS-CUST-BALANCE  PIC S9(7)V99.
```

`WS-CUSTOMER-TABLE` is Chapter 6's own real `OCCURS` / `INDEXED BY` / `ASCENDING KEY` table, assumed loaded from a customer master file before TXNBATCH's own transaction loop begins — the real focus here is what happens once it's populated.

The Main Processing Loop

```
PROCEDURE DIVISION.
    OPEN INPUT TRANSACTION-FILE.
    IF WS-FILE-STATUS NOT = "00"
        DISPLAY "ERROR OPENING TRANSACTION FILE: " WS-FILE-STATUS
        STOP RUN
    END-IF.

    READ TRANSACTION-FILE
        AT END MOVE "Y" TO WS-EOF-FLAG
    END-READ.

    PERFORM UNTIL WS-EOF-FLAG = "Y"
```

```
        UNSTRING TXN-DETAIL DELIMITED BY "-"
            INTO WS-REF-NUMBER WS-BRANCH-CODE
        END-UNSTRING

        SEARCH ALL WS-CUSTOMER-ENTRY
            AT END
                DISPLAY "UNKNOWN CUSTOMER: " TXN-CUSTOMER-ID
            WHEN WS-CUST-ID(WS-CUST-IDX) = TXN-CUSTOMER-ID

                CALL "BALCHECK" USING TXN-AMOUNT WS-VALID-FLAG

                IF WS-VALID-FLAG = "Y"
                    EVALUATE TXN-CODE
                        WHEN "D"
                            ADD TXN-AMOUNT
                                TO WS-CUST-BALANCE(WS-CUST-IDX)
                            ADD 1 TO WS-DEPOSIT-COUNT
                        WHEN "W"
                            SUBTRACT TXN-AMOUNT
                                FROM WS-CUST-BALANCE(WS-CUST-IDX)
                            ADD 1 TO WS-WITHDRAW-COUNT
                        WHEN OTHER
                            DISPLAY "UNKNOWN TRANSACTION CODE: " TXN-CODE
                    END-EVALUATE

                    COMPUTE WS-GRAND-TOTAL =
                        WS-GRAND-TOTAL + TXN-AMOUNT

                    STRING WS-CUST-NAME(WS-CUST-IDX) DELIMITED BY SPACE
                           " [" DELIMITED BY SIZE
                           TXN-CODE DELIMITED BY SIZE
                           "] BRANCH " DELIMITED BY SIZE
                           WS-BRANCH-CODE(1:2) DELIMITED BY SIZE
                        INTO WS-REPORT-LINE
                    END-STRING

                    DISPLAY WS-REPORT-LINE
                ELSE
                    DISPLAY "REJECTED - NEGATIVE AMOUNT: "
                            TXN-CUSTOMER-ID
                END-IF
        END-SEARCH

        READ TRANSACTION-FILE
            AT END MOVE "Y" TO WS-EOF-FLAG
```

```
            END-READ

        END-PERFORM.

        CLOSE TRANSACTION-FILE.

        DISPLAY "BATCH COMPLETE".
        DISPLAY "DEPOSITS PROCESSED:    " WS-DEPOSIT-COUNT.
        DISPLAY "WITHDRAWALS PROCESSED: " WS-WITHDRAW-COUNT.
        DISPLAY "GRAND TOTAL MOVED:     " WS-GRAND-TOTAL.

        GOBACK.
```

Chapter-Attribution Table

CHAPTER	WHAT TXNBATCH DRAWS FROM IT
Ch. 2 — Four Divisions	The genuine ENVIRONMENT/DATA/PROCEDURE division structure the whole program is built on
Ch. 3 — Data Types & PICTURE	Every field's real PICTURE clause, and level numbers grouping `TRANSACTION-RECORD` and `WS-CUSTOMER-ENTRY`
Ch. 4 — MOVE, COMPUTE & Arithmetic	The real `ADD` / `SUBTRACT` / `COMPUTE` statements applying each transaction and tracking `WS-GRAND-TOTAL`
Ch. 5 — IF, EVALUATE & PERFORM	The real `EVALUATE TXN-CODE` dispatch, and `PERFORM UNTIL` driving the whole batch loop
Ch. 6 — OCCURS & Tables	`WS-CUSTOMER-TABLE` itself, its real `INDEXED BY`, and the real `SEARCH ALL` binary-search lookup
Ch. 7 — STRING, UNSTRING & Reference Modification	Splitting `TXN-DETAIL` with real `UNSTRING`, building `WS-REPORT-LINE` with real `STRING`, and `WS-BRANCH-CODE(1:2)` as a real substring
Ch. 8 — Sequential Files	The real `SELECT` / `FD`, `OPEN`, the priming-read `PERFORM UNTIL` pattern, `WRITE`-free reading, and `CLOSE`
Ch. 9 — Subprograms & CALL	The real `CALL "BALCHECK"`, reused unchanged from Chapter 9, validating each amount before it's applied

Hands-On Exercises

EXERCISE 1

Using this chapter's own real TXNBATCH program, explain in your own words why SEARCH ALL, rather than a plain sequential SEARCH, is the genuinely correct choice for finding a customer by TXN-CUSTOMER-ID in WS-CUSTOMER-TABLE — and what real, silent risk this chapter's own Chapter 6 warning would apply here if the customer table were ever loaded out of order.

View solution

EXERCISE 2

Using this chapter's own real TXNBATCH program, explain in your own words what genuinely happens to a real transaction record whose TXN-CUSTOMER-ID doesn't match any entry in WS-CUSTOMER-TABLE, and separately, what happens to one whose TXN-CODE is neither "D" nor "W".

View solution

EXERCISE 3

Using this chapter's own real TXNBATCH program, explain in your own words why CALL "BALCHECK" USING TXN-AMOUNT WS-VALID-FLAG works correctly here even though BALCHECK's own LINKAGE SECTION, back in Chapter 9, was written using the names LS-BALANCE and LS-VALID-FLAG, and was originally called with WS-BALANCE rather than TXN-AMOUNT.

View solution

CHAPTER 10 QUICK REFERENCE — TXNBATCH'S OWN REAL PIECES

- **File I/O** — real SELECT/FD, OPEN INPUT, priming-read PERFORM UNTIL, CLOSE
- **Table lookup** — WS-CUSTOMER-TABLE with real INDEXED BY and SEARCH ALL
- **Subprogram validation** — CALL "BALCHECK" reused unchanged from Chapter 9, matched purely by real position
- **Control flow** — real EVALUATE dispatch on TXN-CODE, nested IF for validation results

- **String handling** — real UNSTRING splitting TXN-DETAIL, STRING building the report line, reference modification for a real 2-character branch substring
- **Arithmetic** — real ADD/SUBTRACT applying each transaction, COMPUTE tracking the running grand total

COURSE COMPLETE COBOL Fundamentals is now complete, 10/10 chapters. Real COBOL work rarely stops at fixed-format sequential files, though — **COBOL Intermediate/Advanced** picks up directly from here with indexed and relative file organization, copybooks, JCL and batch scheduling, embedded SQL against a real database, sort/merge operations, and reading and maintaining genuine legacy code.