

PROGRAMMING

COBOL Intermediate/Advanced

Indexed and relative file organization, copybooks, real JCL batch processing, embedded SQL and DB2/IMS integration, sort/merge and report writing, centralized FILE STATUS error handling, CICS screen handling, object-oriented COBOL, and reading/maintaining legacy code — closing with a complete, worked modernization of a legacy batch system.

10 Chapters

Philip Osztromok

Generated with Claude

CHAPTER 1: INDEXED & RELATIVE FILE ORGANIZATION

COBOL Intermediate/Advanced

Chapter 1 · Indexed & Relative File Organization

Fundamentals' own Chapter 8 covered one real file shape: sequential, read front to back, record after record. Real production COBOL constantly needs something faster — jumping directly to one specific customer, one specific order — without reading everything that comes before it. Two more real organizations make that possible.

Three Real File Organizations, Compared

ORGANIZATION	HOW A RECORD IS GENUINELY LOCATED
SEQUENTIAL	Contiguous, read one after another — Fundamentals' own real shape
INDEXED	Located by a real key value via a genuine index — not by position at all
RELATIVE	Located by its own real ordinal position — record 10 genuinely has key 10

Indexed Files: Real Key-Based Lookup

A real indexed file — implemented in practice by technologies like ISAM and VSAM — carries one or more genuine indexes letting records be found directly by key, and sorted on them. Every real record needs a unique `RECORD KEY`; a real, separate `ALTERNATE RECORD KEY` can also be declared, and — unlike the primary key — it doesn't have to be unique, as long as `WITH DUPLICATES` is stated:

```
SELECT CUSTOMER-FILE ASSIGN TO "CUSTMSTR.DAT"
  ORGANIZATION IS INDEXED
  ACCESS MODE IS DYNAMIC
  RECORD KEY IS CUST-ID
  ALTERNATE RECORD KEY IS CUST-NAME WITH DUPLICATES
  FILE STATUS IS WS-FILE-STATUS.

FD  CUSTOMER-FILE.
01  CUSTOMER-RECORD.
    05  CUST-ID          PIC 9(6).
```

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

Relative Files: The Real Position *Is* the Key

A real relative file works differently again — there's no separate key field stored in each record at all. A record's own ordinal position in the file genuinely **is** its key, tracked in a real **RELATIVE KEY** field:

```
SELECT ORDER-FILE ASSIGN TO "ORDERS.DAT"  
      ORGANIZATION IS RELATIVE  
      ACCESS MODE IS RANDOM  
      RELATIVE KEY IS WS-ORDER-NUM  
      FILE STATUS IS WS-FILE-STATUS.
```

A REAL, EASY-TO-MISS CONSEQUENCE Because position genuinely *is* the key, writing a real record with relative key 5 before records 1 through 4 exist can require the file system to create empty placeholder records to fill that gap — a real structural quirk indexed files simply don't have, since an indexed file's own key values never need to be contiguous.

ACCESS MODE: SEQUENTIAL, RANDOM & DYNAMIC

MODE	WHAT IT GENUINELY ALLOWS
SEQUENTIAL	Records read in real key/positional order, one after another — Fundamentals' own only real option
RANDOM	Any single record located directly by its real key, with no need to read what comes before it
DYNAMIC	Both at once — real sequential reads and real direct-by-key reads, freely mixed in the same program

A real random read against an indexed or relative file uses **INVALID KEY** in place of Fundamentals' own **AT END**, since there's no "end of file" concept for a single direct lookup — only whether that specific real key was found:

```

MOVE 100234 TO CUST-ID.
READ CUSTOMER-FILE
    INVALID KEY
        DISPLAY "CUSTOMER NOT FOUND"
    NOT INVALID KEY
        DISPLAY "FOUND: " CUST-NAME
END-READ.

```

START: Positioning for a Sequential Read

START is a real statement unique to indexed and relative files, in real sequential or dynamic access — it positions the file at a specific real key so a subsequent sequential **READ NEXT** begins from exactly that point, rather than from the very beginning:

```

MOVE 100000 TO CUST-ID.
START CUSTOMER-FILE KEY IS NOT LESS THAN CUST-ID
    INVALID KEY DISPLAY "NO RECORDS FOUND FROM THIS POINT"
END-START.

READ CUSTOMER-FILE NEXT RECORD
    AT END DISPLAY "END OF FILE"
END-READ.

```

Real relational operators beyond **EQUAL TO** — **GREATER THAN**, **NOT LESS THAN**, **GREATER THAN OR EQUAL TO** — let **START** position at the first record satisfying a real range condition, not just an exact match. Leaving out the **KEY** phrase entirely defaults to a real **EQUAL TO** the file's own prime record key.

REWRITE & DELETE: Genuinely New Verbs

Fundamentals' own **WRITE** only ever adds a record. Two further real verbs, valid only against a file opened **I-O**, work on records that already exist:

```

MOVE 100234 TO CUST-ID.
READ CUSTOMER-FILE
    INVALID KEY DISPLAY "NOT FOUND"

```

```
END-READ.
```

```
ADD 500.00 TO CUST-BALANCE.
```

```
REWRITE CUSTOMER-RECORD
```

```
    INVALID KEY DISPLAY "REWRITE FAILED"
```

```
END-REWRITE.
```

```
DELETE CUSTOMER-FILE
```

```
    INVALID KEY DISPLAY "DELETE FAILED"
```

```
END-DELETE.
```

REWRITE REPLACES; IT DOESN'T ADD **REWRITE** genuinely replaces an existing real record in place — it's not supported at all for a line-sequential file, and it always requires a file already opened **I-O**, never **OUTPUT**. **DELETE** is equally real-and-specific: valid only against indexed and relative files, freeing that key's own real storage for later reuse.

Hands-On Exercises

EXERCISE 1

Using this chapter's own real material, explain in your own words why a random READ against CUSTOMER-FILE uses INVALID KEY rather than Fundamentals' own AT END, and why "end of file" genuinely isn't a meaningful concept for that specific kind of read.

 [View solution](#)

EXERCISE 2

Using this chapter's own real material, explain in your own words why writing a relative-file record with RELATIVE KEY 5, before any record with key 1 through 4 exists, can genuinely require the file system to create empty placeholder records — and why an indexed file's own RECORD KEY has no equivalent requirement.

 [View solution](#)

EXERCISE 3

Using this chapter's own real REWRITE example, explain in your own words why the file must genuinely be opened I-O rather than OUTPUT for that statement to work, and why attempting a REWRITE before a successful READ of the same real record would genuinely fail.

 [View solution](#)

CHAPTER 1 QUICK REFERENCE

- **ORGANIZATION IS INDEXED** — real key-based lookup (ISAM/VSAM in practice); **RECORD KEY** unique, **ALTERNATE RECORD KEY ... WITH DUPLICATES** optional and non-unique
- **ORGANIZATION IS RELATIVE** — real ordinal position doubles as the key (**RELATIVE KEY**); writing ahead of existing records can force real empty placeholders
- **ACCESS MODE SEQUENTIAL/RANDOM/DYNAMIC** — real front-to-back reading, real direct-by-key lookup, or both freely mixed
- **INVALID KEY** — the real random-access equivalent of AT END
- **START** — positions for a sequential read from a specific real key; real relational operators beyond EQUAL TO
- **REWRITE** — replaces an existing real record (I-O mode only); **DELETE** — removes one (indexed/relative only, I-O mode only)

CHAPTER 2: COPYBOOKS & MODULAR PROGRAM DESIGN

COBOL Intermediate/Advanced

Chapter 2 · Copybooks & Modular Program Design

Fundamentals' own capstone, TXNBATCH, hand-typed `CUSTOMER-RECORD` directly into its own DATA DIVISION. Any other real program touching the same customer data — a report writer, a validation subprogram, a nightly reconciliation job — would need that identical layout retyped by hand too. A real `COPY` statement makes that unnecessary, and genuinely safer.

The Problem COPY Solves

If ten real programs each hand-type their own copy of `CUSTOMER-RECORD`, and the field `CUST-BALANCE` ever needs to grow from `PIC S9(7)V99` to `PIC S9(9)V99`, all ten need finding and updating — miss one, and it silently reads or writes the wrong number of bytes the next time it runs.

The COPY Statement: Real Compile-Time Inclusion

A real copybook is just a separate text file holding a fragment of COBOL source. `COPY` pulls its content directly into the compiling program, at the exact point the statement appears:

```
FD  CUSTOMER-FILE .  
01  CUSTOMER-RECORD .  
    COPY CUSTREC .
```

`CUSTREC` itself might hold nothing more than:

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

GENUINELY TEXTUAL, NOT A RUNTIME CALL `COPY` does its real work entirely at compile time — the copybook's own text is logically substituted directly into the source before the program is ever compiled. This is a fundamentally different real mechanism from Fundamentals' own `CALL`: there's no separate compiled unit, no runtime linkage, and no positional parameter matching. Once compiled, a program that used `COPY` looks exactly as if the copybook's own lines had been typed there by hand.

REPLACING: One Copybook, Genuinely Different Field Names

The same real layout is often needed under two different real prefixes — once as an `FD` record, once as a subprogram's own `LINKAGE SECTION` item. Real pseudo-text delimiters (`==...==`) inside the copybook mark exactly what `REPLACING` should substitute:

```
01  ==PREFIX== -CUSTOMER-RECORD.
    05 ==PREFIX== -CUST-ID          PIC 9(6).
    05 ==PREFIX== -CUST-NAME        PIC X(30).
    05 ==PREFIX== -CUST-BALANCE     PIC S9(7)V99.
```

```
FD  CUSTOMER-FILE.
    COPY CUSTREC REPLACING ==PREFIX== BY ==FD==.
```

That real inclusion expands to `FD-CUSTOMER-RECORD`, `FD-CUST-ID`, and so on. The identical copybook, included elsewhere with a different real substitution:

```
LINKAGE SECTION.
    COPY CUSTREC REPLACING ==PREFIX== BY ==LS==.
```

genuinely expands to `LS-CUSTOMER-RECORD`, `LS-CUST-ID`, and the rest — the exact same real field shapes, under whatever prefix each including program actually needs.

Where COPY Can Genuinely Appear

COPY is real anywhere a character string or separator can legally appear in the source — not confined to one division. In real practice it's overwhelmingly used in the **DATA DIVISION** for shared record layouts, but a real, shared block of **PROCEDURE DIVISION** logic — a common validation routine, say — can be copied into multiple programs exactly the same way.

A REAL, ENFORCED CONSTRAINT A real copybook can't include itself, even indirectly through a chain of other copybooks, until its own inclusion is fully finished — COBOL genuinely prevents a copybook from being named again partway through its own nested inclusion, closing off any real risk of infinite recursive expansion.

Hands-On Exercises

EXERCISE 1

Using this chapter's own real material, explain in your own words why updating CUST-BALANCE's PICTURE clause inside CUSTREC genuinely fixes every program that used COPY CUSTREC the next time each one is recompiled, without any of those programs' own source files needing to be edited directly.

 [View solution](#)

EXERCISE 2

Using this chapter's own real FD-prefix and LS-prefix examples, explain in your own words why both real inclusions can safely coexist across different programs even though they produce genuinely different field names from the exact same copybook file.

 [View solution](#)

EXERCISE 3

Using this chapter's own real material, explain in your own words the genuine, structural difference between COPY and CALL — specifically, why a change to a copybook requires every including program to be recompiled, while a change to a real subprogram called via CALL does not require the calling program itself to be recompiled at all.

 [View solution](#)

CHAPTER 2 QUICK REFERENCE

- **COPY copybook-name.** — real compile-time textual substitution; the copybook's own text is logically inserted at that exact point
- **REPLACING ==old== BY ==new==** — real pseudo-text substitution, letting one copybook produce genuinely different field names per including program
- **COPY vs. CALL** — COPY is text merged in before compilation, with no runtime linkage at all; CALL is a real, separately compiled unit linked at runtime
- **Legal locations** — anywhere a character string/separator can appear; overwhelmingly used in DATA DIVISION, also valid in PROCEDURE DIVISION
- **Real constraint** — a copybook can't be named again inside its own nested inclusion chain until that inclusion finishes, blocking recursive expansion

CHAPTER 3: BATCH PROCESSING & JCL: HOW COBOL PROGRAMS ACTUALLY RUN

COBOL Intermediate/Advanced

Chapter 3 · Batch Processing & JCL: How COBOL Programs Actually Run

Every program in this course so far has simply "run." On a real mainframe, a compiled COBOL program never runs by itself — it's submitted as one step of a real batch job, described entirely in a separate, genuinely different language: **JCL**, Job Control Language.

What JCL Genuinely Is

JCL emerged real and complete during the **OS/360** project in the 1960s, built specifically to ensure every real resource a batch job needed — programs, datasets, devices — was declared and allocated before that job ever actually ran.

A GENUINELY FAMOUS, REAL COMPLAINT Fred Brooks, who supervised OS/360's own real development, once called JCL "the worst computer programming language ever devised by anybody, anywhere" — a real, documented critique of its designers failing to treat it as a genuine programming language deserving real usability consideration.

Three Real Statement Types

```
//TXNRUN JOB (ACCT123), 'NIGHTLY BATCH', CLASS=A, MSGCLASS=X
//STEP1 EXEC PGM=TXNBATCH
//TXNFILE DD DSN=PROD.TXN.DAILY, DISP=SHR
//SYSOUT DD SYSOUT=*
```

STATEMENT	WHAT IT GENUINELY DECLARES
JOB	Marks the real start of the whole job — billing, run priority, real time/space limits
EXEC PGM=	Names the real compiled program to run for this one step

DD

Connects a real dataset (or device) to the specific name a program's own code refers to internally

Connecting JCL to COBOL: ASSIGN TO a Real DD Name

A mainframe COBOL program's real `SELECT ... ASSIGN TO` clause doesn't name a literal file path the way a PC-based COBOL program might — it names the exact **ddname** the runtime expects the executing JCL to supply:

```
SELECT TRANSACTION-FILE ASSIGN TO TXNFILE
      ORGANIZATION IS SEQUENTIAL
      FILE STATUS IS WS-FILE-STATUS.
```

THE SAME COMPILED PROGRAM, GENUINELY DIFFERENT DATA EVERY RUN `TXNFILE` in the COBOL source and `//TXNFILE DD DSN=PROD.TXN.DAILY,DISP=SHR` in the JCL are the real two halves of the same connection. The compiled program never changes; whoever writes the JCL for a given real run decides which actual dataset `TXNFILE` genuinely points at that time — a different real file, a different real run, with zero recompilation needed.

RETURN-CODE: A Program's Own Real Exit Status

Before ending, a real COBOL program can set its own genuine condition code — a real special register, `RETURN-CODE` — by convention ranging from 0 (normal) up through 16 (a real terminal error):

```
IF WS-FILE-STATUS NOT = "00"
    MOVE 16 TO RETURN-CODE
    STOP RUN
END-IF.

MOVE 0 TO RETURN-CODE.
GOBACK.
```

COND: Real, Genuinely Backward Conditional Logic

A real **COND** parameter on a later **EXEC** statement can skip that step based on an earlier step's own **RETURN-CODE**:

```
//STEP1    EXEC  PGM=TXNBATCH
//STEP2    EXEC  PGM=TXNRPT,COND=(4,LT,STEP1)
```

READ COND BACKWARDS — GENUINELY, NOT A FIGURE OF SPEECH **COND** conditions state when a step is **skipped**, not when it runs. **COND=(4,LT,STEP1)** genuinely means "skip STEP2 if 4 is less than STEP1's real return code" — in other words, STEP2 only actually runs if STEP1 finished with a return code of 4 or lower. Reading a COND clause as "run if" instead of "skip if" gets the real logic backwards every time.

Hands-On Exercises

EXERCISE 1

Using this chapter's own real material, explain in your own words why SELECT TRANSACTION-FILE ASSIGN TO TXNFILE lets the exact same compiled TXNBATCH program process a genuinely different real dataset on two separate nightly runs, with no recompilation between them.

 [View solution](#)

EXERCISE 2

Using this chapter's own real COND example, explain in your own words exactly when STEP2 genuinely runs and when it's genuinely skipped, and why reading COND=(4,LT,STEP1) as "run STEP2 if 4 is less than STEP1's return code" would be a real, backward misreading of it.

 [View solution](#)

EXERCISE 3

Using this chapter's own real RETURN-CODE example, explain in your own words why TXNBATCH setting RETURN-CODE to 16 before a real STOP RUN matters at all to whatever job step comes after it, given that STEP1 has already ended by the time STEP2's own COND is evaluated.

[View solution](#)

CHAPTER 3 QUICK REFERENCE

- **JOB / EXEC PGM = / DD** — real JCL statements marking the job, naming the program, and connecting datasets to internal file names
- **SELECT ... ASSIGN TO ddname** — the real, exact link between a COBOL program's internal file reference and JCL's own DD statement
- **RETURN-CODE** — a real special register; by convention, 0 is normal, 16 is a real terminal error
- **COND=(value,operator,stepname)** — real conditional step logic that states when a step is **skipped**, not when it runs

CHAPTER 4: COBOL & DATABASES: EMBEDDED SQL AND DB2/IMS INTEGRATION

COBOL Intermediate/Advanced

Chapter 4 · COBOL & Databases: Embedded SQL and DB2/IMS Integration

Every real file in this course so far — sequential, indexed, relative — has been COBOL's own native I/O. Real mainframe COBOL just as often talks to a genuinely separate system entirely: a real database, most commonly IBM's own DB2, sometimes the older IMS.

DB2: A Real, Traceable History

DB2's own roots trace to Edgar F. Codd's real 1970 relational-database model, developed further inside IBM's own System R research project starting in 1974 — the very project that produced SQL itself. IBM's real DB2 product launched on the MVS mainframe platform in **1983**.

Embedded SQL: EXEC SQL / END-EXEC

Real SQL statements sit directly inside COBOL source, marked off by `EXEC SQL` and `END-EXEC`:

```
EXEC SQL
    SELECT CUST_NAME, CUST_BALANCE
        INTO :WS-CUST-NAME, :WS-CUST-BALANCE
        FROM CUSTOMER
        WHERE CUST_ID = :WS-CUST-ID
END-EXEC.
```

A REAL PRECOMPILER, NOT A NEW LANGUAGE FEATURE COBOL UNDERSTANDS DIRECTLY A genuine precompiler runs before the real COBOL compiler ever sees the source — it parses every `EXEC SQL` block and replaces it with real, ordinary host-language statements calling into a database code library. Underneath, embedded SQL genuinely resolves down to Chapter 9's own real `CALL` mechanism — the COBOL compiler itself never has any built-in understanding of SQL at all.

Host Variables: The Real Colon Prefix

`:WS-CUST-ID`, `:WS-CUST-NAME`, and `:WS-CUST-BALANCE` above are real **host variables** — ordinary COBOL `WORKING-STORAGE` fields, declared exactly like any other field, referenced inside SQL with a real leading colon so the precompiler can tell a COBOL field apart from a genuine database column name.

SQLCODE: Did the Statement Genuinely Succeed?

After every real embedded SQL statement executes, a genuine result code — `SQLCODE` — reports what happened:

SQLCODE	REAL MEANING
0	Statement genuinely succeeded
100	No real row matched the query
negative	A genuine SQL error occurred

```
IF SQLCODE = 0
    DISPLAY "FOUND: " WS-CUST-NAME
ELSE
    IF SQLCODE = 100
        DISPLAY "NO CUSTOMER WITH THAT ID"
    ELSE
        DISPLAY "SQL ERROR: " SQLCODE
    END-IF
END-IF.
```

Cursors: When One Row Genuinely Isn't Enough

A plain `SELECT ... INTO` can only ever return one real row. Retrieving many rows needs a real **cursor** — a mechanism that genuinely traverses a whole result set, one row at a time, via a real four-step sequence:

```
EXEC SQL
    DECLARE CUST-CURSOR CURSOR FOR
        SELECT CUST_ID, CUST_NAME
```

```

        FROM CUSTOMER
        WHERE CUST_BALANCE < 0
END-EXEC.

EXEC SQL
    OPEN CUST-CURSOR
END-EXEC.

PERFORM UNTIL SQLCODE = 100
    EXEC SQL
        FETCH CUST-CURSOR INTO :WS-CUST-ID, :WS-CUST-NAME
    END-EXEC
    IF SQLCODE = 0
        DISPLAY "OVERDRAWN: " WS-CUST-NAME
    END-IF
END-PERFORM.

EXEC SQL
    CLOSE CUST-CURSOR
END-EXEC.

```

DECLARE names the real query; **OPEN** genuinely executes it and establishes the result set; **FETCH** pulls one real row at a time into host variables; **CLOSE** releases it. This loop's own real shape — checking a result code after every fetch — deliberately echoes Fundamentals' own priming-read pattern for sequential files, applied here to a database result set instead.

IMS: A Genuinely Different, Older Real Model

IMS predates DB2 substantially — real development began in **1966**, to track the bill of materials for the Saturn V rocket, with the system formally installed in 1968. Unlike DB2's real relational model, IMS uses a genuinely **hierarchical** structure, accessed via real **DL/I calls** rather than SQL. Its own real staying power is striking: 2003 — 35 years after release — was IMS's best sales year yet, with over 95% of the Fortune 1000 still using it.

Hands-On Exercises

EXERCISE 1

Using this chapter's own real finding-box, explain in your own words why the real COBOL compiler itself has no built-in understanding of SQL at all, and what actually happens to an EXEC SQL block before the COBOL compiler ever sees the source.

 [View solution](#)

EXERCISE 2

Using this chapter's own real cursor example, explain in your own words why a plain SELECT ... INTO couldn't have been used to find every overdrawn customer, and why the PERFORM UNTIL SQLCODE = 100 loop condition is genuinely the correct one to end the FETCH loop.

 [View solution](#)

EXERCISE 3

Using this chapter's own real material, explain in your own words the genuine structural difference between DB2 and IMS as database systems, and why a program written to use DL/I calls against IMS couldn't simply be pointed at a DB2 database with no other changes.

 [View solution](#)

CHAPTER 4 QUICK REFERENCE

- **EXEC SQL ... END-EXEC** — real embedded SQL, processed by a genuine precompiler before real COBOL compilation
- **:host-variable** — a colon-prefixed real WORKING-STORAGE field, distinguishing it from a genuine database column
- **SQLCODE** — real per-statement result: 0 success, 100 no row found, negative a genuine error
- **DECLARE / OPEN / FETCH / CLOSE** — the real four-step cursor sequence for a multi-row result set
- **DB2** — real relational model, SQL-based, launched 1983; **IMS** — real hierarchical model, DL/I calls, dating to 1966

CHAPTER 5: SORT/MERGE OPERATIONS & REPORT WRITING

COBOL Intermediate/Advanced

Chapter 5 · Sort/Merge Operations & Report Writing

Two genuinely distinct real features have traveled together since COBOL's own early history — real sort/merge facilities and a real, declarative report-writing facility, both introduced together in **COBOL-61 Extended (1963)**. One puts data in order; the other turns ordered data into a formatted, paginated report with almost no manual layout code at all.

SORT: Real Ordering, USING/GIVING

A real **SORT** reorders an entire file by one or more keys — the leftmost is the real major key, each one after it a progressively less significant tiebreaker:

```
SORT SORT-WORK-FILE
  ON ASCENDING KEY TXN-CUSTOMER-ID
  USING RAW-TRANSACTION-FILE
  GIVING SORTED-TRANSACTION-FILE.
```

USING and **GIVING** handle the real file mechanics automatically — opening, reading every input record, and closing, then writing every sorted record to the output and closing that too. Real **INPUT PROCEDURE** / **OUTPUT PROCEDURE** phrases can replace either one when records genuinely need selecting or modifying on the way in or out — but never together with the plain phrase they'd be replacing.

MERGE: Real Combining, Not Sorting

MERGE looks similar but does a genuinely different job — it combines files that are **already sorted** on the same real key into one, without ever sorting anything itself:

```
MERGE COMBINED-WORK-FILE
  ON ASCENDING KEY TXN-CUSTOMER-ID
```

```
USING EAST-TXN-FILE WEST-TXN-FILE
GIVING COMBINED-TXN-FILE.
```

A REAL, LOAD-BEARING PRECONDITION If `EAST-TXN-FILE` and `WEST-TXN-FILE` aren't already genuinely sorted by `TXN-CUSTOMER-ID` before this statement runs, `MERGE` doesn't detect the problem or fix it — it simply produces a genuinely wrong, incorrectly ordered result, since combining already-sorted streams is the entire real algorithm `MERGE` relies on.

Report Writer: A Genuinely Declarative Alternative to Hand-Written Reports

Real Report Writer lets a program declare what a report should look like — the programmer specifies layout and data, not the real logic for page breaks, headings, and formatting:

```
FD  REPORT-FILE.

REPORT SECTION.
RD  CUSTOMER-REPORT
    CONTROL IS WS-BRANCH-CODE
    PAGE LIMIT 60 LINES.

01  TYPE IS DETAIL.
    05 COLUMN 1  PIC X(30) SOURCE CUST-NAME.
    05 COLUMN 35 PIC ZZZ,ZZ9.99 SOURCE CUST-BALANCE.
```

```
INITIATE CUSTOMER-REPORT.
    PERFORM UNTIL WS-EOF-FLAG = "Y"
        GENERATE CUSTOMER-REPORT
        READ CUSTOMER-FILE
            AT END MOVE "Y" TO WS-EOF-FLAG
        END-READ
    END-PERFORM.
TERMINATE CUSTOMER-REPORT.
```

Four real statements drive the whole thing: `INITIATE` starts the report, `GENERATE` produces one real detail line (and automatically fires a real **control break** whenever `WS-BRANCH-CODE`

genuinely changes value), **SUPPRESS** can skip a line that would otherwise print, and **TERMINATE** closes the report out.

A REAL, DOCUMENTED DECLINE Report Writer's own real popularity has always varied sharply by shop — some organizations lean on it heavily, others avoid it entirely, partly over genuine implementation-quality and runtime-memory concerns. It was nearly dropped from COBOL-74 before being reinstated, and by the real COBOL 2014 standard, it had become fully **optional** rather than a guaranteed feature.

Hands-On Exercises

EXERCISE 1

Using this chapter's own real SORT example, explain in your own words what USING and GIVING each genuinely handle automatically, and what would need to change if a program wanted to filter out certain real records before sorting rather than sorting the entire input file as-is.

 [View solution](#)

EXERCISE 2

Using this chapter's own real MERGE example and its warn-box, explain in your own words what would genuinely happen if EAST-TXN-FILE were accidentally left unsorted before this MERGE statement ran, and why MERGE itself wouldn't catch that mistake.

 [View solution](#)

EXERCISE 3

Using this chapter's own real Report Writer example, explain in your own words what would genuinely happen inside CUSTOMER-REPORT's own output the moment WS-BRANCH-CODE changes value from one real customer record to the next, and why the programmer never had to write that logic by hand.

 [View solution](#)

CHAPTER 5 QUICK REFERENCE

- **SORT ... ON ASCENDING/DESCENDING KEY** — real reordering by one or more keys, leftmost most significant; **USING/GIVING** automate file mechanics, **INPUT/OUTPUT PROCEDURE** replace either for real record selection/modification
- **MERGE** — real combining of files *already* sorted the same way; never sorts anything itself, and won't detect unsorted input
- **Report Writer (RD/REPORT SECTION)** — a real declarative report facility, dating to COBOL-61 Extended (1963)
- **INITIATE / GENERATE / SUPPRESS / TERMINATE** — the real four statements driving a Report Writer report; GENERATE triggers real automatic control breaks
- **Real status** — Report Writer became optional as of COBOL 2014, after nearly being dropped from COBOL-74 entirely

CHAPTER 6: ERROR HANDLING: FILE STATUS CODES & EXCEPTION HANDLING

COBOL Intermediate/Advanced

Chapter 6 · Error Handling: FILE STATUS Codes & Exception Handling

Fundamentals' own Chapter 8 covered three real codes — 00, 10, 35 — enough to get a sequential program working. Real production systems distinguish far more real conditions than that, and repeating a check after every single I/O statement gets unwieldy fast. Two things fix that: a fuller real code vocabulary, and a genuinely different place to put the handling logic.

FILE STATUS: The Real Category Structure

Every real two-character `FILE STATUS` value's own first character marks a genuine category of outcome:

FIRST DIGIT	REAL CATEGORY
0	Successful completion
1	At end (real end-of-file conditions)
2	Invalid key (real random/keyed-access failures)
3	Permanent error
4	Logic error — the file used incorrectly by the program itself
9	Implementor-defined

A Fuller Real Code Reference

CODE	REAL MEANING
23	Record not found — a real random READ's key genuinely doesn't exist
22	Duplicate key on a WRITE to an indexed file
35	File not found on OPEN INPUT/I-O — Fundamentals' own real example

41	OPEN attempted on a file already genuinely open
46	Sequential READ attempted with no real current record established
47	READ attempted on a file not genuinely opened for input
48	WRITE attempted on a file not genuinely opened for output
49	REWRITE/DELETE attempted on a file not genuinely opened I-O

A GENUINELY USEFUL PATTERN Codes 4x are real **logic errors** — the program's own code did something the file's current open mode doesn't support (like this chapter's own 47/48/49) — distinct from a real 3x permanent error, which reflects a genuine problem with the file or device itself rather than a mistake in the calling program.

The Real Repetition Problem

Every real file-handling example since Fundamentals Chapter 8 has repeated a check — `AT` `END`, `INVALID KEY`, or a manual `IF WS-FILE-STATUS NOT = "00"` — after every single I/O statement. A real program touching one file dozens of times ends up with that same handling logic copied dozens of times too.

DECLARATIVES: Real, Centralized Error Handling

A real `DECLARATIVES` section, placed before a program's own main `PROCEDURE DIVISION` logic, lets one `USE`-headed section handle every genuine error on a named file automatically — no `INVALID KEY` or `AT END` clause needed on the individual statements at all:

```
PROCEDURE DIVISION.
DECLARATIVES.
CUSTOMER-FILE-ERROR SECTION.
    USE AFTER STANDARD ERROR PROCEDURE ON CUSTOMER-FILE.
CUSTOMER-FILE-ERROR-PARA.
    EVALUATE WS-FILE-STATUS
        WHEN "23"
            DISPLAY "RECORD NOT FOUND"
        WHEN "35"
```

```

        DISPLAY "FILE NOT FOUND"
        MOVE 16 TO RETURN-CODE
        STOP RUN
    WHEN OTHER
        DISPLAY "UNEXPECTED FILE ERROR: " WS-FILE-STATUS
        MOVE 16 TO RETURN-CODE
        STOP RUN
    END-EVALUATE.
END DECLARATIVES.

```

GENUINELY AUTOMATIC, NOT JUST REORGANIZED **USE AFTER STANDARD ERROR PROCEDURE ON CUSTOMER-FILE** genuinely runs this whole paragraph automatically whenever *any* I/O statement against **CUSTOMER-FILE** — a **READ**, a **WRITE**, a **REWRITE**, whatever — produces a non-successful real status. Every future **READ CUSTOMER-FILE** elsewhere in the program can genuinely drop its own **INVALID KEY / AT END** clause entirely, since this one declared section already covers it.

Hands-On Exercises

EXERCISE 1

Using this chapter's own real FILE STATUS category table, explain in your own words the genuine difference between a real code 35 (category 3) and a real code 47 (category 4), and why they represent two structurally different kinds of problem rather than both simply being "errors."

 [View solution](#)

EXERCISE 2

Using this chapter's own real DECLARATIVES example, explain in your own words what genuinely happens the next time a plain READ CUSTOMER-FILE statement (with no INVALID KEY clause of its own) encounters a real code 23, and why that doesn't cause the program to crash or leave the error unhandled.

 [View solution](#)

EXERCISE 3

Using this chapter's own real material, explain in your own words the genuine tradeoff a program takes on by moving all of CUSTOMER-FILE's own error handling into one DECLARATIVES section, compared with handling each individual I/O statement's own INVALID KEY or AT END clause separately, inline, at the point it occurs.

 [View solution](#)

CHAPTER 6 QUICK REFERENCE

- **FILE STATUS category (first digit)** — 0 success, 1 at end, 2 invalid key, 3 permanent error, 4 logic error, 9 implementor-defined
- **Common real codes** — 22 duplicate key, 23 record not found, 35 file not found on OPEN, 41 already open, 46-49 real mode-mismatch logic errors
- **DECLARATIVES ... END DECLARATIVES** — a real, separate PROCEDURE DIVISION section holding centralized error-handling logic
- **USE AFTER STANDARD ERROR PROCEDURE ON file-name** — genuinely, automatically invoked on any non-successful I/O against that file, with no per-statement INVALID KEY/AT END needed

CHAPTER 7: SCREEN HANDLING: CICS & INTERACTIVE COBOL BASICS

COBOL Intermediate/Advanced

Chapter 7 · Screen Handling: CICS & Interactive COBOL Basics

Every real program across both courses so far has been batch — no human involved while it runs. **CICS** is the real online counterpart: a genuine transaction-processing system letting a COBOL program respond directly to someone sitting at a real terminal.

What CICS Genuinely Is

CICS traces to a real 1966 joint project with Michigan Bell, publicly announced in 1968 as "Public Utility Customer Information Control System," and formally released as a general IBM product on **8 July 1969**. Its whole real purpose is online transaction management — high-volume, screen-oriented processing, genuinely distinct from everything batch-related covered so far in this course.

EXEC CICS: The Same Real Embedded-Command Style

Real CICS commands sit inside COBOL source exactly like Chapter 4's own embedded SQL — `EXEC CICS` / `END-EXEC`, processed by a genuine translator before the real COBOL compiler ever sees the source:

```
EXEC CICS
    SEND MAPSET('CUSTSET') MAP('CUSTMAP')
END-EXEC.
```

SEND MAP / RECEIVE MAP: Real Screen I/O

A real CICS screen is defined separately as a **BMS** (Basic Mapping Support) map. `RECEIVE MAP` pulls whatever the user just typed into a COBOL structure; `SEND MAP` pushes new content back out to the real terminal:

```

EXEC CICS
  RECEIVE MAP('CUSTMAP') MAPSET('CUSTSET')
  INTO(CUSTOMER-SCREEN)
END-EXEC.

MOVE CUST-ID-FIELD OF CUSTOMER-SCREEN TO WS-CUST-ID.

EXEC SQL
  SELECT CUST_NAME INTO :WS-CUST-NAME
  FROM CUSTOMER WHERE CUST_ID = :WS-CUST-ID
END-EXEC.

MOVE WS-CUST-NAME TO CUST-NAME-FIELD OF CUSTOMER-SCREEN.

EXEC CICS
  SEND MAP('CUSTMAP') MAPSET('CUSTSET')
  FROM(CUSTOMER-SCREEN)
END-EXEC.

```

Pseudo-Conversational: A Genuinely Distinctive Real Design

A CICS transaction genuinely does **not** sit waiting for the user's next keystroke the way a simple interactive program might. Instead, after sending a screen out, it ends completely:

```

EXEC CICS
  RETURN TRANSID('CUST') COMMAREA(WS-COMM-AREA)
END-EXEC.

```

WHY ENDING COMPLETELY IS THE REAL POINT, NOT A LIMITATION A real **RETURN** releases every resource the transaction was holding — memory, database connections, everything — the instant a screen goes out to the terminal. Whatever real time the human spends reading it and deciding what to type next costs the mainframe genuinely nothing, since no program is sitting idle waiting on them. When the user does respond, CICS starts a genuinely fresh instance of the same transaction from scratch, named by the real **TRANSID** the previous instance specified.

COMMAREA: Carrying State Across Separate Instances

Since each screen is handled by a genuinely new, fresh transaction instance with none of the previous instance's own working storage still around, real state has to be carried forward explicitly. The **COMMAREA** (communication area) is exactly that — a real data area [RETURN](#) hands off, which the next instance receives back automatically, letting a multi-screen real business process continue correctly across what are, underneath, entirely separate program executions.

Hands-On Exercises

EXERCISE 1

Using this chapter's own real material, explain in your own words why CICS's own pseudo-conversational design — ending the transaction completely after every screen — is genuinely more efficient for serving thousands of simultaneous terminal users than a program that simply waits for the next keystroke while remaining active.

[View solution](#)

EXERCISE 2

Using this chapter's own real material, explain in your own words why a value stored only in an ordinary WORKING-STORAGE field during one CICS transaction instance would genuinely be lost by the time the user's next screen response triggers a new instance, and what COMMAREA does differently.

[View solution](#)

EXERCISE 3

Using this chapter's own real customer-lookup example, explain in your own words why EXEC CICS and EXEC SQL are able to sit side by side in the same real PROCEDURE DIVISION code, given what Chapter 4 already established about how embedded SQL is genuinely processed before compilation.

[View solution](#)

CHAPTER 7 QUICK REFERENCE

- **CICS** — real online transaction processing, formally released 1969, genuinely distinct from every batch topic in this course
- **EXEC CICS ... END-EXEC** — real embedded commands, processed by a genuine translator, the same style as Chapter 4's own EXEC SQL
- **SEND MAP / RECEIVE MAP** — real screen output/input against a BMS-defined map
- **Pseudo-conversational** — a transaction genuinely ends via RETURN after every screen, releasing all resources during real human think-time
- **COMMAREA** — the real mechanism carrying state forward from one transaction instance to the genuinely separate next one

CHAPTER 8: OBJECT-ORIENTED COBOL & MODERN EXTENSIONS

COBOL Intermediate/Advanced

Chapter 8 · Object-Oriented COBOL & Modern Extensions

Every real program across both courses so far has been procedural — divisions, paragraphs, `CALL`. A real, genuinely significant turning point arrived decades after COBOL's own 1959 origin: **COBOL 2002** added object-oriented syntax and free-format source together, in the same single revision.

A Real, Traceable Turning Point

Work on object-oriented COBOL began in the early 1990s, with real features drawn directly from **C++** and **Smalltalk**, finally approved and published in late 2002 — driven substantially by a genuine, practical need: real interoperability with newer platforms like .NET and Java.

CLASS-ID & METHOD-ID: A Real Class Definition

A real COBOL class lives in its own source, headed by `CLASS-ID` instead of `PROGRAM-ID`. Reworking Fundamentals' own `BALCHECK` subprogram as a genuine method shows the real shape:

```
IDENTIFICATION DIVISION.
CLASS-ID. CUSTOMER-VALIDATOR.

OBJECT.
    METHOD-ID. IS-VALID-BALANCE.
DATA DIVISION.
LINKAGE SECTION.
01  LS-BALANCE  PIC S9(7)V99.
01  LS-RESULT   PIC X(1).
PROCEDURE DIVISION USING LS-BALANCE LS-RESULT.
    IF LS-BALANCE < 0
        MOVE "N" TO LS-RESULT
    ELSE
        MOVE "Y" TO LS-RESULT
    END-IF.
```

```
END METHOD IS-VALID-BALANCE.
END OBJECT.

END CLASS CUSTOMER-VALIDATOR.
```

The real logic inside `IS-VALID-BALANCE` is identical to Fundamentals' own `BALCHECK` — the genuine change is structural: a method now belongs to a real class, callable on a real object, rather than existing as a standalone subprogram.

INVOKE: The Real OO Counterpart to CALL

`INVOKE` genuinely acts like Chapter 9's own `CALL`, but targets a real method on a real object instead of a standalone program:

```
INVOKE VALIDATOR-OBJ "IS-VALID-BALANCE" USING WS-BALANCE WS-RESULT.
```

A real, genuine shorthand also exists for inline invocation — calling a method directly as part of an ordinary statement, without a separate `INVOKE`:

```
MOVE VALIDATOR-OBJ::"IS-VALID-BALANCE" TO WS-RESULT.
```

INHERITS: Real Class Inheritance

A real subclass genuinely extends an existing class with a plain `INHERITS` clause on its own `CLASS-ID` line:

```
CLASS-ID. PREMIUM-CUSTOMER-VALIDATOR INHERITS CUSTOMER-VALIDATOR.
```

Free-Format Source: A Genuine Departure

The same real 2002 revision that added object orientation also introduced **free-format code** — real COBOL that no longer has to respect the strict, punch-card-era column positions every

earlier chapter's own examples have implicitly followed. Free-format lines can genuinely start in any column at all, matching the visual style of newer languages.

XML & JSON: Real, Later Data-Interchange Extensions

GENUINELY REAL, ADDED LATER STILL Real XML processing capability was addressed through Technical Reports published between 2003 and 2009, extending COBOL well past its own 2002 object-oriented milestone. Real JSON handling arrived as a later addition still, in the same broad spirit — letting a COBOL program genuinely produce and consume the data formats modern web-facing systems actually expect, without abandoning the language entirely.

Hands-On Exercises

EXERCISE 1

Using this chapter's own real CUSTOMER-VALIDATOR example and Fundamentals' own Chapter 9 material, explain in your own words what genuinely changed and what genuinely stayed the same between BALCHECK as a standalone subprogram and IS-VALID-BALANCE as a class method.

 [View solution](#)

EXERCISE 2

Using this chapter's own real material, explain in your own words why PREMIUM-CUSTOMER-VALIDATOR INHERITS CUSTOMER-VALIDATOR genuinely gives the new class access to IS-VALID-BALANCE without that method needing to be rewritten or copied into the new class at all.

 [View solution](#)

EXERCISE 3

Using this chapter's own real material, explain in your own words why object orientation, free-format source, and later XML/JSON support are each described as genuinely separate real additions to COBOL, layered on at different points, rather than one single unified redesign of the language.

 [View solution](#)

CHAPTER 8 QUICK REFERENCE

- **CLASS-ID / METHOD-ID** — real class and method definitions, introduced in COBOL 2002, drawing on C++ and Smalltalk
- **INVOKE object "method" USING ...** — the real OO counterpart to Chapter 9's own CALL
- **object::"method"** — a real, genuine inline shorthand for method invocation
- **INHERITS** — real class inheritance on a CLASS-ID line
- **Free-format source** — real column-position freedom, added in the same 2002 revision
- **XML/JSON** — real, later interchange extensions (XML via Technical Reports 2003-2009) letting COBOL talk to modern web-facing systems

CHAPTER 9: READING, DEBUGGING & MAINTAINING LEGACY CODE

COBOL Intermediate/Advanced

Chapter 9 · Reading, Debugging & Maintaining Legacy Code

Fundamentals' own Chapter 1 established the real scale — some 200 billion lines of COBOL still running, 80% of all business programs. The honest, genuinely dominant reality of real COBOL work today follows directly from that: far more of it is reading and maintaining code someone else wrote decades ago than writing anything new.

GO TO & the Real Structured-Programming Shift

Real code older than the industry's own shift toward structured programming leans heavily on `GO TO`, jumping freely between paragraphs rather than using Chapter 5's own real `PERFORM` / `EVALUATE` structures. That shift traces to a genuinely famous real letter — Edsger Dijkstra's "**Go To Statement Considered Harmful**," published in the March 1968 *Communications of the ACM* (originally titled "A Case Against the Goto Statement," changed by editor Niklaus Wirth). Donald Knuth's own real, wry remark on the resulting backlash: "Dr. Goto cheerfully complained that he was always being eliminated."

WHY THIS GENUINELY MATTERS FOR READING OLD CODE Code written before, or slow to adopt, this real shift can require tracing a real tangle of `GO TO` statements jumping between paragraphs in an order that has nothing to do with how the source is actually laid out on the page — genuinely harder to follow than Chapter 5's own structured `PERFORM`-based style.

ALTER: A Real, Now-Discouraged Trap

Truly old code can hide something genuinely worse than a plain `GO TO` — the real `ALTER` statement, which modifies a `GO TO`'s own target *at runtime*:

```
PROCESS-PARA.
    GO TO STEP-A.

STEP-A.
```

```

DISPLAY "PROCESSING STEP A".
ALTER PROCESS-PARA TO PROCEED TO STEP-B.

```

```

STEP-B.
    DISPLAY "PROCESSING STEP B".

```

A GENUINELY DISORIENTING REAL BEHAVIOR The first time `PROCESS-PARA` runs, its `GO TO` sends control to `STEP-A` — which then genuinely rewrites that same `GO TO` statement so the *next* time `PROCESS-PARA` runs, it goes to `STEP-B` instead. The exact same line of source code does something different the second time, with the real change made from somewhere else in the program entirely.

`ALTER` is real, but genuinely discouraged — it actively encourages unstructured logic, modern compilers can flag it with a real diagnostic, and it's outright disallowed in real recursive programs, methods, or threaded code. The documented fix reuses exactly what Fundamentals' own Chapter 5 already covered:

```

01  WS-CURRENT-STEP  PIC X(1) VALUE "A".

PROCESS-PARA.
    EVALUATE WS-CURRENT-STEP
        WHEN "A"
            DISPLAY "PROCESSING STEP A"
            MOVE "B" TO WS-CURRENT-STEP
        WHEN "B"
            DISPLAY "PROCESSING STEP B"
    END-EVALUATE.

```

Cryptic Names & Undocumented Logic

Real historical field-length limits pushed decades of COBOL toward heavily abbreviated, genuinely cryptic field names — a real practical reading skill for legacy work is inferring a field's actual purpose from how it's used across a program, since the name itself often won't say.

The Real Debugging Line Indicator

A real, literal `D` in a fixed-format line's own indicator column (column 7) marks that line as debugging-only — genuinely present in the source, but otherwise ignored unless the program is compiled with debugging lines specifically enabled. Finding a stray `D`-marked line in old code is a real, direct clue that a previous maintainer left tracing or diagnostic output behind deliberately, not by accident.

A Real, Practical Approach to Modernization

Given the real scale of code involved, wholesale rewriting is rarely the genuinely practical choice. A far more common real approach wraps existing, working COBOL behind a modern interface — Chapter 4's own embedded SQL or Chapter 7's own CICS transaction, for instance — modernizing what's visible at the edges while leaving proven, working internal logic genuinely untouched.

Hands-On Exercises

EXERCISE 1

Using this chapter's own real ALTER example, explain in your own words why reading PROCESS-PARA's own source code once, without knowing STEP-A also contains a real ALTER statement, would give a genuinely misleading picture of what the program actually does the second time it runs.

 [View solution](#)

EXERCISE 2

Using this chapter's own real EVALUATE-based rewrite, explain in your own words why WS-CURRENT-STEP genuinely needs to exist as a real, separate WORKING-STORAGE field for the rewritten version to reproduce the original ALTER version's own real behavior.

 [View solution](#)

EXERCISE 3

Using this chapter's own real material, explain in your own words why wrapping existing, working COBOL behind a modern interface is genuinely often preferred over a full rewrite, given the real scale of COBOL code already established in Fundamentals Chapter 1.

 [View solution](#)

CHAPTER 9 QUICK REFERENCE

- **GO TO-heavy legacy code** — real pre-structured-programming style, predating or slow to adopt Dijkstra's real 1968 critique
- **ALTER** — real, genuinely discouraged; rewrites a GO TO's own target at runtime; documented fix is EVALUATE
- **Cryptic field names** — a real legacy of historical field-length limits; infer purpose from usage, not the name alone
- **Column-7 'D'** — the real debugging line indicator; ignored unless debugging lines are specifically enabled at compile time
- **Wrapping over rewriting** — a real, practical modernization pattern given COBOL's own documented scale

CHAPTER 10: CAPSTONE — MODERNIZING A LEGACY COBOL SYSTEM

COBOL Intermediate/Advanced

Chapter 10 · Capstone: Modernizing a Legacy COBOL System

Fundamentals' own TXNBATCH shipped as a clean, working batch program. A few real years of maintenance later, it has aged into exactly the kind of system Chapter 9 described — real cryptic patches, a lingering `ALTER`, a customer file that's outgrown sequential access, and no real error-handling discipline at all. This capstone modernizes it, one real technique at a time, using everything Course 2 covered.

Step 1 — Fixing the Legacy Control Flow (Ch. 9)

A maintainer's real patch from years ago used `ALTER` to skip validation on a "trusted" batch run. It's replaced with a real, explicit `EVALUATE` flag check instead — the exact fix Chapter 9 documented:

```
EVALUATE WS-RUN-MODE
  WHEN "TRUSTED"
    CONTINUE
  WHEN OTHER
    PERFORM VALIDATE-TRANSACTION
END-EVALUATE.
```

Step 2 — Sharing the Record Layout via a Copybook (Ch. 2)

`CUSTOMER-RECORD`, hand-typed separately in TXNBATCH and in this capstone's own new CICS program (Step 8), moves into one real shared copybook, `CUSTREC`, included with `REPLACING` wherever it's needed.

Step 3 — Converting to a Real Indexed Customer File (Ch. 1)

```

SELECT CUSTOMER-FILE ASSIGN TO CUSTMSTR
      ORGANIZATION IS INDEXED
      ACCESS MODE IS DYNAMIC
      RECORD KEY IS CUST-ID
      FILE STATUS IS WS-FILE-STATUS.

```

TXNBATCH's own original sequential scan for a specific customer is replaced with a real direct-by-key **READ**, dropping the linear search entirely.

Step 4 — Centralizing File Error Handling (Ch. 6)

```

DECLARATIVES.
CUSTOMER-FILE-ERROR SECTION.
      USE AFTER STANDARD ERROR PROCEDURE ON CUSTOMER-FILE.
CUSTOMER-FILE-ERROR-PARA.
      EVALUATE WS-FILE-STATUS
            WHEN "23" DISPLAY "CUSTOMER NOT FOUND"
            WHEN OTHER MOVE 16 TO RETURN-CODE
                        STOP RUN
      END-EVALUATE.
END DECLARATIVES.

```

Every real **READ** against the newly-indexed file drops its own inline **INVALID KEY** clause — this one declared section now covers all of them.

Step 5 — Sorting Transactions Before Processing (Ch. 5)

```

SORT SORT-WORK-FILE
      ON ASCENDING KEY TXN-CUSTOMER-ID
      USING RAW-TRANSACTION-FILE
      GIVING SORTED-TRANSACTION-FILE.

```

Ordering transactions by customer first lets the modernized batch loop group each customer's own transactions together, rather than jumping around the indexed file at random.

Step 6 — Real-Time Balance Verification via Embedded SQL (Ch. 4)

```
EXEC SQL
    SELECT CUST_BALANCE INTO :WS-LIVE-BALANCE
    FROM CUSTOMER WHERE CUST_ID = :WS-CUST-ID
END-EXEC.

IF SQLCODE = 0 AND WS-LIVE-BALANCE NOT = CUST-BALANCE
    DISPLAY "WARNING: FILE/DATABASE BALANCE MISMATCH"
END-IF.
```

A genuine cross-check against DB2's own live balance catches drift between the batch file and the real system of record before it compounds.

Step 7 — Running the Modernized Job (Ch. 3)

```
//TXNRUN2 JOB (ACCT123),'MODERNIZED BATCH',CLASS=A
//STEP1 EXEC PGM=TXNBATCH2
//CUSTMSTR DD DSN=PROD.CUST.INDEXED,DISP=SHR
//TXNFILE DD DSN=PROD.TXN.DAILY,DISP=SHR
//STEP2 EXEC PGM=TXNRPT,COND=(4,LT,STEP1)
```

Step 8 — Wrapping With a Real CICS Front-End (Ch. 7, Ch. 9)

Rather than rewriting the whole system, a new, small CICS transaction — reusing the exact same `CUSTREC` copybook and the same real indexed `CUSTOMER-FILE` — is wrapped around it, letting a real teller look up a live balance without touching TXNBATCH2's own proven batch logic at all:

```
EXEC CICS
    READ FILE('CUSTMSTR') INTO(CUSTOMER-RECORD)
    RIDFLD(WS-CUST-ID)
END-EXEC.

EXEC CICS
    SEND MAP('CUSTMAP') MAPSET('CUSTSET') FROM(CUSTOMER-RECORD)
```

```
END-EXEC.

EXEC CICS
  RETURN TRANSID( 'CUST' ) COMMAREA(WS-COMM-AREA)
END-EXEC.
```

Step 9 — One Shared, Reusable Validation Class (Ch. 8)

Both TXNBATCH2 (Step 1) and the new CICS transaction (Step 8) need the exact same balance validation. Rather than duplicating it, both genuinely `INVOKE` the same real class method:

```
INVOKE VALIDATOR-OBJ "IS-VALID-BALANCE" USING WS-BALANCE WS-RESULT.
```

Chapter-Attribution Table

CHAPTER	WHAT THIS MODERNIZATION STEP DRAWS FROM IT
Ch. 1 — Indexed & Relative Files	Converting CUSTOMER-FILE to real ORGANIZATION IS INDEXED with direct-by-key access
Ch. 2 — Copybooks	One shared CUSTREC layout, included in both the batch program and the new CICS transaction
Ch. 3 — JCL	The real JOB/EXEC/DD job stream running the modernized batch step and its downstream report
Ch. 4 — Embedded SQL	A real DB2 cross-check catching file/database balance drift
Ch. 5 — Sort	Ordering transactions by customer before the main processing loop
Ch. 6 — DECLARATIVES	One centralized error-handling section covering every I/O against the newly-indexed file
Ch. 7 — CICS	The new real-time customer-lookup transaction wrapped around the existing system
Ch. 8 — OO COBOL	One shared, INVOKE-able validation class used by both the batch job and the CICS transaction

Ch. 9 — Legacy Maintenance

Recognizing and replacing the real ALTER-based patch, and choosing to wrap rather than rewrite

Hands-On Exercises

EXERCISE 1

Using this capstone's own real Step 3 and Step 4, explain in your own words why converting CUSTOMER-FILE to an indexed organization made the real DECLARATIVES-based error handling in Step 4 genuinely more valuable than it would have been for the original sequential file.

 [View solution](#)

EXERCISE 2

Using this capstone's own real Step 9, explain in your own words why a shared, INVOKE-able validation class is genuinely a better fit here than either duplicating the validation logic in both programs, or reusing Fundamentals' own plain BALCHECK subprogram unchanged via two separate CALL statements.

 [View solution](#)

EXERCISE 3

Using this capstone's own real Step 8, explain in your own words why wrapping a new CICS transaction around the existing, modernized batch system's own indexed file and copybook is a genuine, direct application of Chapter 9's own "wrap rather than rewrite" recommendation, rather than simply a convenient shortcut.

 [View solution](#)

CHAPTER 10 QUICK REFERENCE — THIS CAPSTONE'S OWN REAL MODERNIZATION STEPS

- **Control flow** — a real ALTER-based patch replaced with EVALUATE
- **Data** — a shared copybook; a sequential file converted to indexed with direct-by-key access

- **Error handling** — centralized via DECLARATIVES, covering the whole indexed file at once
- **Processing order** — SORT grouping transactions by customer before the main loop
- **Data integrity** — a real embedded SQL cross-check against DB2's own live balance
- **Execution** — a real JCL job stream tying the modernized batch step to a downstream report
- **Modernization strategy** — a new CICS front-end wrapped around proven logic, not a rewrite; one shared OO validation class reused by both paths

COURSE COMPLETE COBOL Intermediate/Advanced is now complete, 10/10 chapters — and with it, the full two-course COBOL project (Fundamentals + Intermediate/Advanced) is complete at 20 chapters total.