IBM Solution & Service Company (China)

COBOL Programming
Fundamental

ISSC SH
Walker JIA
Version 1.0

2004/11

Training Schedule

	Day 1	Day 2	Day 3	Day 4
Moring	**Introduction to COBOL** **COBOL Basics 1**	**Introduction to Sequential Files** **Processing Sequential Files**	**Simple iteration with the PERFORM verb** **Arithmetic and Edited Pictures**	**Conditions** **Tables and the PERFORM .. VARYING**
After noon	**Exercise 1** **COBOL Basics 2**	**Exercise 2**	**Exercise 3**	**Exercise 3 (Cont.)** **Designging Programs**

Table of contents

- Introduction to COBOL
- COBOL Basics 1
- COBOL Basics 2
- Introduction to Sequential Files
- Processing Sequential Files
- Simple iteration with the PERFORM verb
- Arithmetic and Edited Pictures
- Conditions
- Tables and the PERFORM ... VARYING
- Designing Programs

Introduction to COBOL

Overview

- COBOL design goals.
- Structure of COBOL programs.
- The four divisions.
- IDENTIFICATION DIVISION, DATA DIVISION, PROCEDURE DIVISION.
- Sections, paragraphs, sentences and statements.
- Example COBOL programs.

Introduction to COBOL

COBOL

- COBOL is an acronym which stands for

  Common Business Oriented Language.

- The name indicates the target area of COBOL applications.
  - COBOL is used for developing business, typically file-oriented, applications.
  - It is not designed for writing systems programs. You would not develop an operating system or a compiler using COBOL.

- COBOL is one of the oldest computer languages in use (it was developed around the end of the 1950s). As a result it has some idiosyncracies which programmers may find irritating.

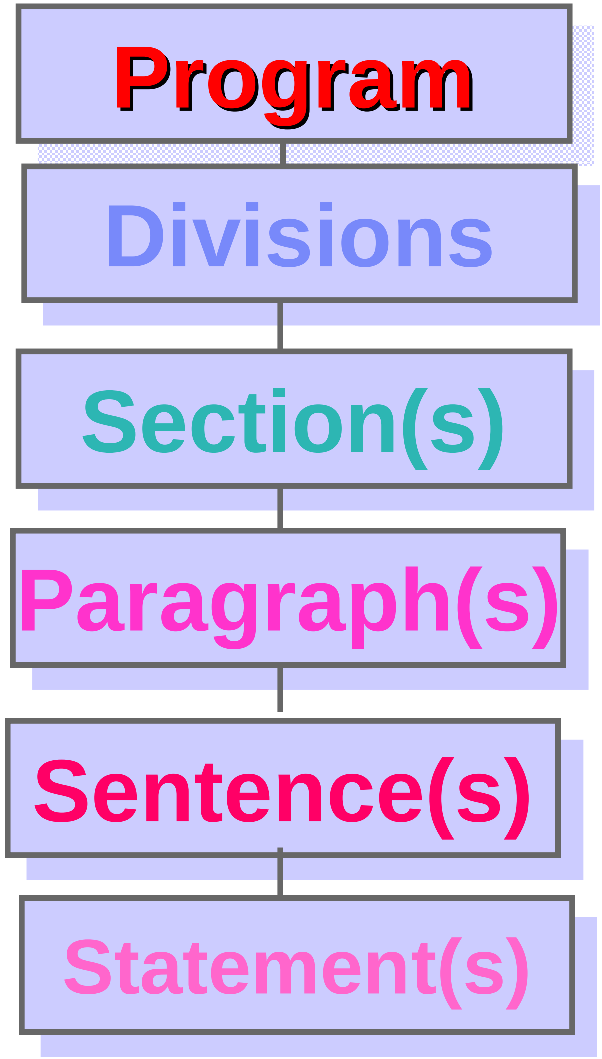
Introduction to COBOL

COBOL idiosyncracies

- One of the design goals was to make the language as English-like as possible. As a consequence
  - the COBOL reserved word list is quite extensive and contains hundreds of entries.
  - COBOL uses structural concepts normally associated with English prose such as section, paragraph, sentence and so on.
    As a result COBOL programs tend to be verbose.
- Some implementations require the program text to adhere to certain, archaic, formatting restrictions.
- Although modern COBOL has introduced many of the constructs required to write well structured programs it also still retains elements which, if used, make it difficult, and in some cases impossible, to write good programs.

Introduction to COBOL

Structure of COBOL programs



Program

Divisions

Section(s)

Paragraph(s)

Sentence(s)

Statement(s)

Introduction to COBOL

The Four Divisions

- DIVISIONS are used to identify the principal components of the program text. There are four DIVISIONS in all.

  - IDENTIFICATION DIVISION.
  - ENVIRONMENT DIVISION.
  - DATA DIVISION.
  - PROCEDURE DIVISION.

- Although some of the divisions may be omitted the sequence in which the DIVISIONS are specified is fixed and must follow the pattern shown above.

Introduction to COBOL

Functions of the four divisions

- The IDENTIFICATION DIVISION is used to supply information about the program to the programmer and to the compiler.
- The ENVIRONMENT DIVISION describes to the compiler the environment in which the program will run.
- As the name suggests, the DATA DIVISION is used to provide the descriptions of most of the data to be processed by the program.
- The PROCEDURE DIVISION contains the description of the algorithm which will manipulate the data previously described. Like other languages COBOL provides a means for specifying sequence, selection and iteration constructs.

Introduction to COBOL

COBOL Program Text Structure

IDENTIFICATION DIVISION.

Program Details

DATA DIVISION.

Data Descriptions

PROCEDURE DIVISION.

Algorithm Description

NOTE

The keyword DIVISION and a '**full-stop**' is used in every case.

Introduction to COBOL

IDENTIFICATION DIVISION

- The purpose of the IDENTIFICATION DIVISION is to provide information about the program to the programmer and to the compiler.
- Most of the entries in the IDENTIFICATION DIVISION are directed at the programmer and are treated by the compiler as comments.
- An exception to this is the PROGRAM-ID clause. Every COBOL program must have a PROGRAM-ID. It is used to enable the compiler to identify the program.
- There are several other informational paragraphs in the IDENTIFICATION DIVISION but we will ignore them for the moment.

Introduction to COBOL

The IDENTIFICATION DIVISION Syntax

- The IDENTIFICATION DIVISION has the following structure

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ProgName.
[AUTHOR. YourName.]
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. BMJA01.
AUTHOR. Michael Coughlan.
```

- The keywords IDENTIFICATION DIVISION represent the division header and signal the commencement of the program text.
- The paragraph name PROGRAM-ID is a keyword. It must be specified immediately after the division header.
- The program name can be up to 8 characters long.

Introduction to COBOL

The DATA DIVISION

- The DATA DIVISION is used to describe most of the data that a program processes.
- The DATA DIVISION is divided into two main sections;
  - FILE SECTION.
  - WORKING-STORAGE SECTION.
- The FILE SECTION is used to describe most of the data that is sent to, or comes from, the computer's peripherals.
- The WORKING-STORAGE SECTION is used to describe the general variables used in the program.

Introduction to COBOL

DATA DIVISION Syntax

- The DATA DIVISION has the following structure

```
DATA  DIVISION .
[FILE  SECTION .                       ]
[    File   Section    entries.        ]
[WORKING - STORAGE    SECTION         .]
[    WS  entries.                      ]
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  Sequence-Program.
AUTHOR.  Michael Coughlan.

DATA DIVISION.
WORKING-STORAGE SECTION.
01  Num1          PIC 9  VALUE ZEROS.
01  Num2          PIC 9  VALUE ZEROS.
01  Result        PIC 99 VALUE ZEROS.
```

Introduction to COBOL

The PROCEDURE DIVISION

- The PROCEDURE DIVISION is where all the data described in the DATA DIVISION is processed and produced. It is here that the programmer describes his algorithm.

- The PROCEDURE DIVISION is hierarchical in structure and consists of Sections, Paragraphs, Sentences and Statements.

- Only the Section is optional. There must be at least one paragraph, sentence and statement in the PROCEDURE DIVISION.

- In the PROCEDURE DIVISION paragraph and section names are chosen by the programmer. The names used should reflect the processing being done in the paragraph or section.

Introduction to COBOL

Sections

- A section is a block of code made up of one or more paragraphs.
- A section begins with the section name and ends where the next section name is encountered or where the program text ends.
- A section name consists of a name devised by the programmer or defined by the language followed by the word SECTION followed by a full stop.

```
FILE SECTION.
```

Introduction to COBOL

Paragraphs

- Each section consists of one or more paragraphs.
- A paragraph is a block of code made up of one or more sentences.
- A paragraph begins with the paragraph name and ends with the next paragraph or section name or the end of the program text.
- The paragraph name consists of a name devised by the programmer or defined by the language followed by a full stop.

  PrintFinalTotals.

  PROGRAM-ID.

Introduction to COBOL

Sentences and Statements

- A paragraph consists of one or more sentences.
- A sentence consists of one or more statements and is terminated by a full stop.

```
MOVE .21 TO VatRate
COMPUTE VatAmount = ProductCost * VatRate.


DISPLAY "Enter name " WITH NO ADVANCING
ACCEPT  StudentName
DISPLAY "Name entered was " StudentName.
```

- A statement consists of a COBOL verb and an operand or operands.

```
SUBTRACT Tax FROM GrossPay GIVING NetPay
READ StudentFile
          AT END SET EndOfFile TO TRUE
END-READ.
```

Introduction to COBOL

A Full COBOL program

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  SAMPLE1.
AUTHOR.  Michael Coughlan.

DATA DIVISION.
WORKING-STORAGE SECTION.
01  Num1            PIC 9  VALUE ZEROS.
01  Num2            PIC 9  VALUE ZEROS.
01  Result          PIC 99 VALUE ZEROS.

PROCEDURE DIVISION.
CalculateResult.
   ACCEPT Num1.
   ACCEPT Num2.
   MULTIPLY Num1 BY Num2 GIVING Result.
   DISPLAY "Result is = ", Result.
   STOP RUN.
```

Introduction to COBOL

The minimum COBOL program

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  SAMPLE2.

PROCEDURE DIVISION.
DisplayPrompt.
    DISPLAY "I did it".
    STOP RUN.
```

Table of contents

Introduction to COBOL

➜ COBOL Basics 1

COBOL Basics 2

Introduction to Sequential Files

Processing Sequential Files

Simple iteration with the PERFORM verb

Arithmetic and Edited Pictures

Conditions

Tables and the PERFORM ... VARYING

Designing Programs

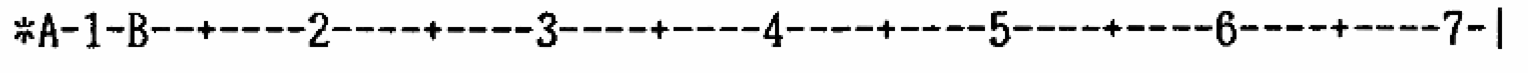
COBOL Basics 1

Overview

- The COBOL coding rules.
- Name construction.
- Describing Data.
- Data names/variables.
- Cobol Data Types and data description.
- The PICTURE clause.
- The VALUE clause.
- Literals and Figurative Constants.
- Editing, compiling, linking and running COBOL programs

COBOL Basics 1

COBOL coding rules



```
*A-1-B--+----2----+----3----+----4----+----5----+----6----+----7-|
```

***：Identification Area（7th byte）**

A：AreaA（8th ～11th byte）

B：Area B（12th ～72th byte）

- Almost all COBOL compilers treat a line of COBOL code as if it contained two distinct areas. These are known as;
  Area A and Area B
- When a COBOL compiler recognizes these two areas, all division, section, paragraph names, FD entries and 01 level numbers must start in Area A. All other sentences must start in Area B.
- Area A is four characters wide and is followed by Area B.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  Program.
* This is a comment. It starts
* with an asterisk in column 1
```

COBOL Basics 1

Name Construction

- All user defined names, such as data names, paragraph names, section names and mnemonic names, must adhere to the following rules;
  - They must contain at least one character and not more than 30 characters.
  - They must contain at least one alphabetic character and they must not begin or end with a hyphen.
  - They must be contructed from the characters A to Z, the number 0 to 9 and the hyphen.   e.g. TotalPay, Gross-Pay, PrintReportHeadings, Customer10-Rec

- All data-names should describe the data they contain.
- All paragraph and section names should describe the function of the paragraph or section.

COBOL Basics 1

Describing DATA

There are basically three kinds of data used in COBOL programs;

1. Variables.
2. Literals.
3. Figurative Constants.

Unlike other programming languages, COBOL does not support user defined constants.

COBOL Basics 1

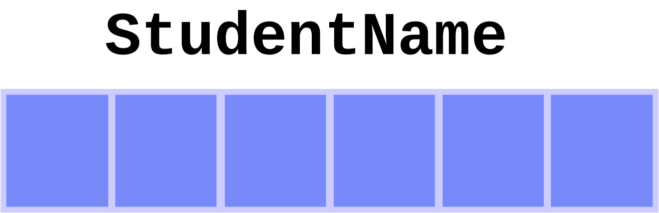
Data-Names / Variables

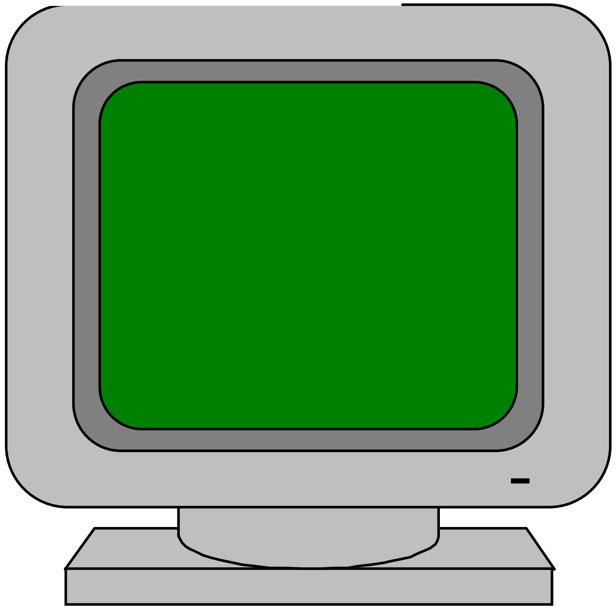
- A variable is a named location in memory into which a program can put data and from which it can retrieve data.
- A data-name or identifier is the name used to identify the area of memory reserved for the variable.
- Variables must be described in terms of their type and size.
- Every variable used in a COBOL program must have a description in the DATA DIVISION.

COBOL Basics 1

Using Variables

```
01 StudentName   PIC X(6) VALUE SPACES.

MOVE "JOHN" TO StudentName.
DISPLAY "My name is ", StudentName.
```





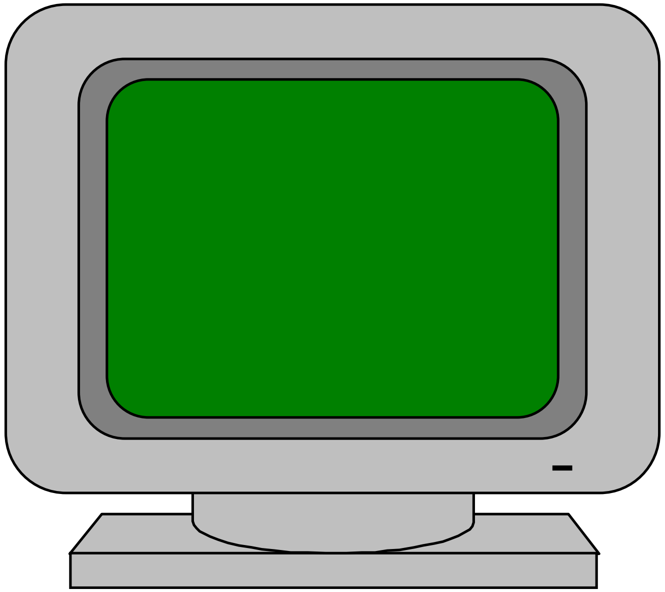
COBOL Basics 1

Using Variables

```
01 StudentName   PIC X(6) VALUE SPACES.

MOVE "JOHN" TO StudentName.
DISPLAY "My name is ", StudentName.
```

StudentName

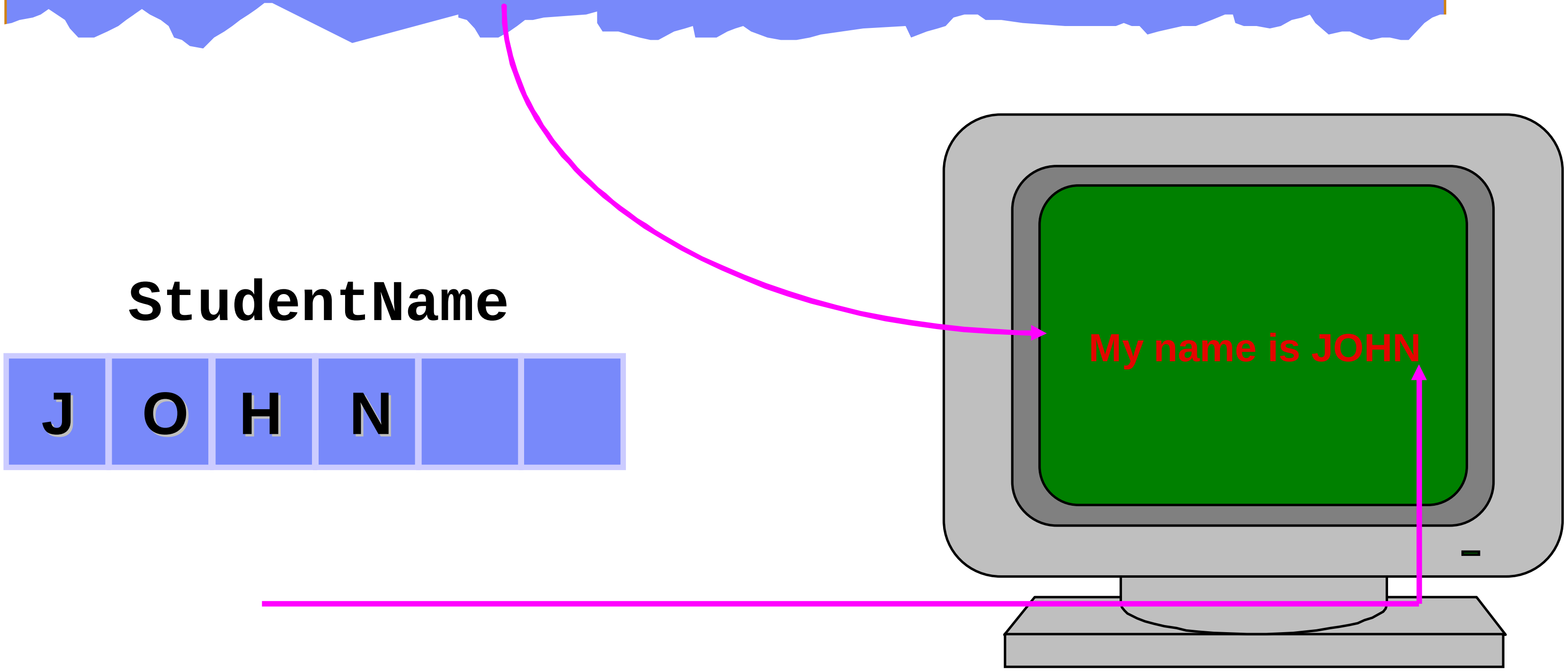
J	O	H	N		



COBOL Basics 1

Using Variables

```
01 StudentName   PIC X(6) VALUE SPACES.

MOVE "JOHN" TO StudentName.
DISPLAY "My name is ", StudentName.
```



COBOL Basics 1

COBOL Data Types

- COBOL is not a “typed” language and the distinction between some of the data types available in the language is a little blurred.
- For the time being we will focus on just two data types,
  - numeric
  - text or string
- Data type is important because it determines the operations which are valid on the type.
- COBOL is not as rigorous in the application of typing rules as other languages.

  For example, some COBOL “numeric” data items may, from time to time, have values which are not “numeric”!

COBOL Basics 1

Quick Review of "Data Typing"

- In "typed" languages simply specifying the type of a data item provides quite a lot of information about it.
- The type usually determines the range of values the data item can store.

  For instance a CARDINAL item can store values between 0..65,535 and an INTEGER between -32,768..32,767
- From the type of the item the compiler can establish how much memory to set aside for storing its values.
- If the type is "REAL" the number of decimal places is allowed to vary dynamically with each calculation but the amount of the memory used to store a real number is fixed.

COBOL Basics 1

COBOL data description

- Because COBOL is not typed it employs a different mechanism for describing the characteristics of the data items in the program.
- COBOL uses what could be described as a “declaration by example” strategy.
- In effect, the programmer provides the system with an example, or template, or PICTURE of what the data item looks like.
- From the “picture” the system derives the information necessary to allocate it.

COBOL Basics 1

COBOL ‘PICTURE’ Clause symbols

- To create the required ‘picture’ the programmer uses a set of symbols.
- The following symbols are used frequently in picture clauses;

  9 (the digit nine) is used to indicate the occurrence of a digit at the corresponding position in the picture.

  X (the character X) is used to indicate the occurrence of any character from the character set at the corresponding position in the picture

  V (the character V) is used to indicate position of the decimal point in a numeric value! It is often referred to as the “assumed decimal point” character.

  S (the character S) indicates the presence of a sign and can only appear at the beginning of a picture.

COBOL Basics 1

COBOL 'PICTURE' Clauses

- Some examples

PICTURE 999	a three digit (+ive only) integer
PICTURE S999	a three digit (+ive/-ive) integer
PICTURE XXXX	a four character text item or string
PICTURE 99V99	a +ive 'real' in the range 0 to 99.99
PICTURE S9V9	a +ive/-ive 'real' in the range ?

- If you wish you can use the abbreviation PIC.
- Numeric values can have a maximum of 18 (eighteen) digits (i.e. 9's).
- The limit on string values is usually system-dependent.

COBOL Basics 1

Abbreviating recurring symbols

- Recurring symbols can be specified using a ‘repeat’ factor inside round brackets

PIC 9(6) is equivalent to PICTURE 999999

PIC 9(6)V99 is equivalent to PIC 999999V99

PICTURE X(10) is equivalent to PIC XXXXXXXXXX

PIC S9(4)V9(4) is equivalent to PIC S9999V9999

PIC 9(18) is equivalent to PIC 999999999999999999

COBOL Basics 1

Declaring DATA in COBOL

- In COBOL a variable declaration consists of a line containing the following items;

  1. **A level number.**
  2. **A data-name or identifier.**
  3. **A PICTURE clause.**

- We can give a starting value to variables by means of an extension to the picture clause called the value clause.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01  Num1           PIC 999    VALUE ZEROS.
01  VatRate        PIC V99    VALUE .18.
01  StudentName    PIC X(10)  VALUE SPACES.
```

DATA

Num1	VatRate	StudentName
000	.18	

COBOL Basics 1

COBOL Literals

- String/Alphanumeric literals are enclosed in quotes and may consists of alphanumeric characters

  e.g. "Michael Ryan",   "-123",   "123.45"

- Numeric literals may consist of numerals, the decimal point and the plus or minus sign. Numeric literals are not enclosed in quotes.

  e.g.   123,   123.45, -256,   +2987

COBOL Basics 1

Figurative Constants

- COBOL provides its own, special constants called Figurative Constants.

SPACE or SPACES	=	..
ZERO or ZEROS or ZEROS	=	**0**
QUOTE or QUOTES	=	**"**
HIGH-VALUE or HIGH-VALUES	=	Max Value
LOW-VALUE or LOW-VALUES	=	Min Value
ALL *literal*	=	Fill With Literal

COBOL Basics 1

Figurative Constants - Examples

```
01   GrossPay    PIC 9(5)V99 VALUE 13.5.

     ZERO
MOVE ZEROS    TO GrossPay.
     ZEROES
```

GrossPay

0	0	0	1	3	5	0

ñ
I

```
StudentName   PIC X(10) VALUE "MIKE".

MOVE ALL "-" TO StudentName.
```

StudentName

M	I	K	E	..	..	.. ..	..	..	

COBOL Basics 1

Figurative Constants - Examples

```
01  GrossPay    PIC 9(5)V99 VALUE 13.5.

     ZERO
MOVE ZEROS     TO GrossPay.
     ZEROES
```

GrossPay

0	0	0	0	0	0	0

ñ

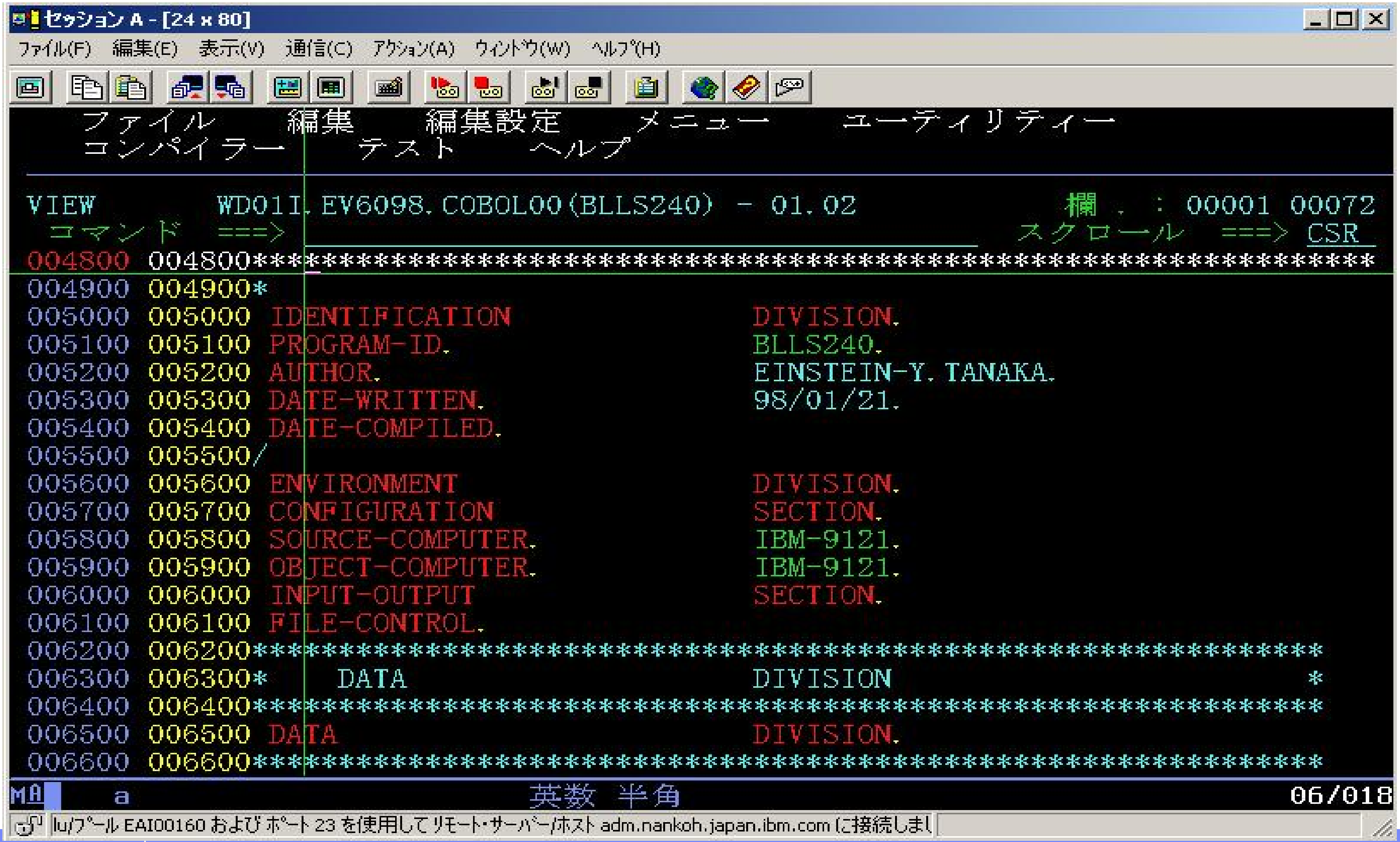
```
01  StudentName    PIC X(10) VALUE "MIKE".

MOVE ALL "-" TO StudentName.
```

StudentName

-	-	-	-	-	-	-	-	-	-

COBOL Basics 1

Editing, Compiling, Running


セッション A - [24 x 80]
ファイル(F) 編集(E) 表示(V) 通信(C) アクション(A) ウィンドウ(W) ヘルプ(H)

```
   ファイル    編集   編集設定   メニュー    ユーティリティー
   コンパイラー    テスト    ヘルプ
────────────────────────────────────────────────────────────────────────────
VIEW       WD01I.EV6098.COBOL00(BLLS240) - 01.02            欄 . : 00001 00072
 コマンド ===>                                               スクロール ===> CSR
004800 004800*****************************************************************
004900 004900*
005000 005000 IDENTIFICATION                      DIVISION.
005100 005100 PROGRAM-ID.                         BLLS240.
005200 005200 AUTHOR.                             EINSTEIN-Y.TANAKA.
005300 005300 DATE-WRITTEN.                       98/01/21.
005400 005400 DATE-COMPILED.
005500 005500/
005600 005600 ENVIRONMENT                         DIVISION.
005700 005700 CONFIGURATION                       SECTION.
005800 005800 SOURCE-COMPUTER.                    IBM-9121.
005900 005900 OBJECT-COMPUTER.                    IBM-9121.
006000 006000 INPUT-OUTPUT                        SECTION.
006100 006100 FILE-CONTROL.
006200 006200****************************************************************
006300 006300*    DATA                            DIVISION                  *
006400 006400****************************************************************
006500 006500 DATA                                DIVISION.
006600 006600****************************************************************
MA    a                      英数 半角                                   06/018
```

lu/プール EAI00160 および ポート 23 を使用して リモート・サーバー/ホスト adm.nankoh.japan.ibm.com に接続しま

COBOL Basics 1

Editing, Compiling, Running

```
//EV6098A  JOB (F9500B,WD01X),'EV6098',NOTIFY=EV6098,
//             MSGLEVEL=(1,1),
//             CLASS=M,MSGCLASS=R,USER=WD01UJ1,PASSWORD=MON10JUN
//*********************************************************************
//*  UIBMCL: COMPILE AND LINKEDIT A COBOL PROGRAM
//*
//UIBMCL  PROC WSPC=500,NAME=TEMPNAME
//*
//*            COMPILE THE COBOL PROGRAM
//*
//COB     EXEC PGM=IGYCRCTL,
//         PARM='APOST,LIB,NOSEQ,RENT,TRUNC(BIN),LANG(UE)‘
//STEPLIB  DD  DSN=SYS1.IGY.SIGYCOMP,DISP=SHR
//SYSIN    DD  DSN=WD01I.DS.COBOL&SRC(&NAME),DISP=SHR
//SYSLIB   DD  DSN=WD01I.DS.COPY&COPY,DISP=SHR <=== BLK 3120
//         DD  DSN=MQM.SCSQCOBC,DISP=SHR
//SYSLIN   DD  DSN=WD01I.DS.UT.OBJ&SRC(&NAME),DISP=SHR
//OUTDEF OUTPUT PRMODE=SOSI2,CHARS=(KN10,KNJE)
//SYSPRINT DD  SYSOUT=*,OUTPUT=*.OUTDEF
//SYSUDUMP DD  SYSOUT=*
//SYSUT1   DD  SPACE=(800,(&WSPC,&WSPC),,,ROUND),UNIT=3390
//SYSUT2   DD  SPACE=(800,(&WSPC,&WSPC),,,ROUND),UNIT=3390
```

COBOL Basics 1

Editing, Compiling, Running

```
//*
//*            LINKEDIT IF THE  COMPILE
//*            RETURN CODES ARE 4 OR LESS
//*
//LKED    EXEC PGM=HEWL,PARM='XREF',COND=(4,LT,COB)
//SYSLIB   DD  DSN=SYS1.SCEELKED,DISP=SHR
//         DD  DSN=DSNCFD.SDSNEXIT,DISP=SHR
//         DD  DSN=DSNCFD.DSNLOAD,DISP=SHR
//OBJECT   DD  DSN=WD01I.DS.UT.OBJ&SRC,DISP=SHR
//CSQSTUB  DD  DSN=MQM.SCSQLOAD,DISP=SHR
//CEEUOPT  DD  DSN=WD01I.DS.LOAD00,DISP=SHR
//SYSLMOD  DD  DSN=WD01I.DS.UT.LOAD&SRC(&NAME),DISP=SHR
//SYSLIN   DD  DSN=WD01I.DS.UT.OBJ&SRC(&NAME),DISP=SHR
//         DD  DSN=WD01I.CSL1.PARMLIB(DSNELI),DISP=SHR
//         DD  DSN=WD01I.DS.PARAM00(CEEUOPT),DISP=SHR
//OUTDEF OUTPUT PRMODE=SOSI2,CHARS=(KN10,KNJE)
//SYSPRINT DD  SYSOUT=*,OUTPUT=*.OUTDEF
//SYSUDUMP DD  SYSOUT=*
//SYSUT1   DD  SPACE=(4096,(500,500)),UNIT=3390
//      PEND
//*
//COMP   EXEC UIBMCL,SRC=00,COPY=00,NAME=BUAC25
//COB.SYSIN  DD  DSN=WD01I.EV6098.COBOL00(BUAC25)
```

COBOL Basics 1

Editing, Compiling, Running

```
//EV6098G2 JOB (F9500B,WD01X),CFD,TIME=1440,
//             REGION=8M,CLASS=M,MSGCLASS=R,MSGLEVEL=(1,1),
//       NOTIFY=EV6098,USER=WD01UJ1,PASSWORD=MON10JUN
//JOBLIB   DD  DSN=WD01I.DS.UT.LOAD00,DISP=SHR
//         DD  DSN=DSNCFD.DSNLOAD,DISP=SHR
//********************************************************************
//SCR     EXEC DSNDCR
  DSN=WD01I.DS.PCDERR.CHK.REPORT
//*-------------------------------------------------------------------
//*     BUAC25 DUW25 CREATE                                        ***
//*-------------------------------------------------------------------
//STEP160  EXEC  PGM=BUAC25,COND=(4,LT)
//IDUW13   DD  DSN=&&DUW13T,DISP=(OLD,DELETE)
//UAC250   DD  DSN=WD01I.DS.PCDERR.CHK.REPORT,DISP=(,CATLG),
//             UNIT=3390,VOL=SER=EGF001,SPACE=(CYL,(15,15),RLSE),
//             DCB=(RECFM=FBA,LRECL=133,BLKSIZE=0)
//OFSW16   DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//SYSUDUMP DD  SYSOUT=*
//SYSABOUT DD  SYSOUT=*
//SYSOUT   DD  SYSOUT=*
/*
```

EXERCISE 1

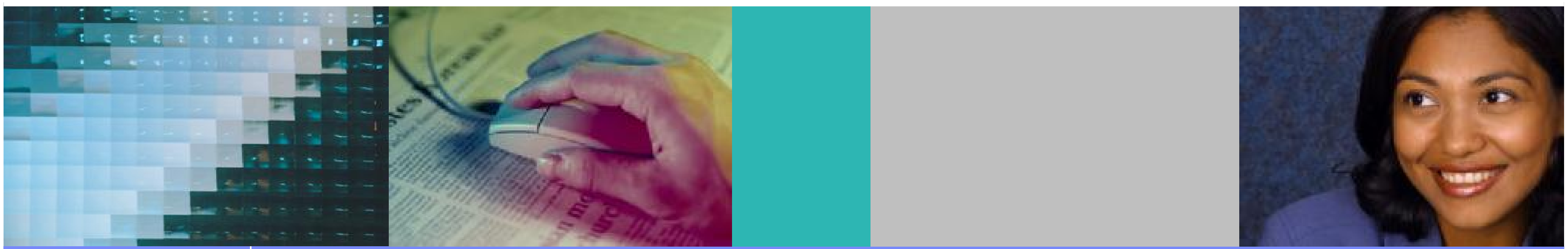


Table of contents

COBOL Basics 2

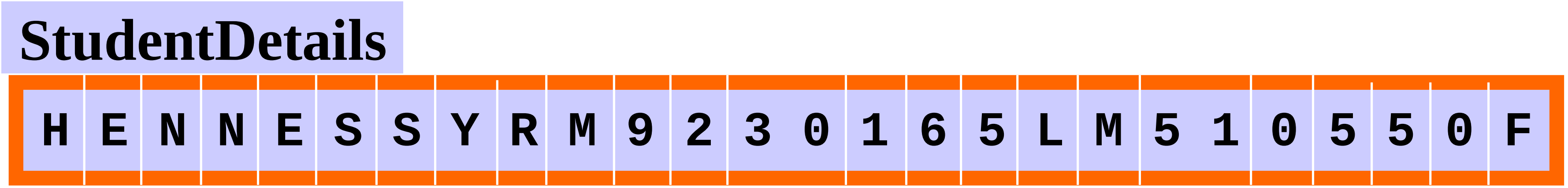
Overview

- **Level Numbers.**
- **Group and elementary data items.**
- **Group item PICTURE clauses.**
- **The MOVE. MOVEing numeric items.**
- **DISPLAY and ACCEPT.**

COBOL Basics 2

Group Items/Records

```
WORKING-STORAGE SECTION.
01  StudentDetails       PIC X(26).
```



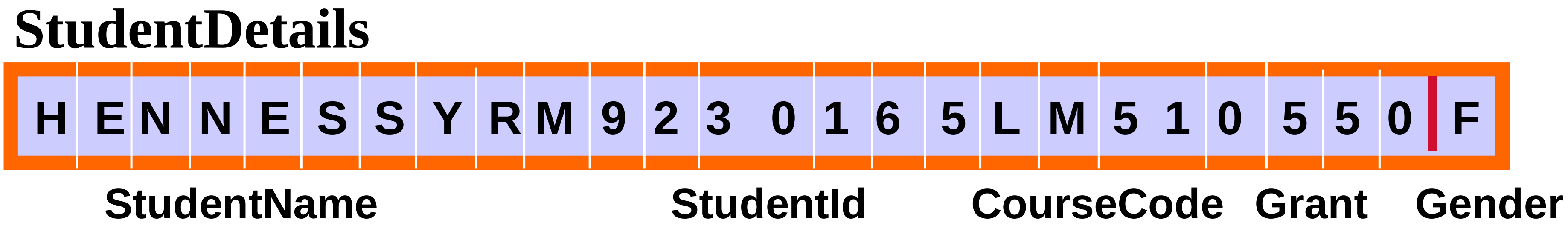
StudentDetails

H E N N E S S Y R M 9 2 3 0 1 6 5 L M 5 1 0 5 5 0 F

COBOL Basics 2

Group Items/Records

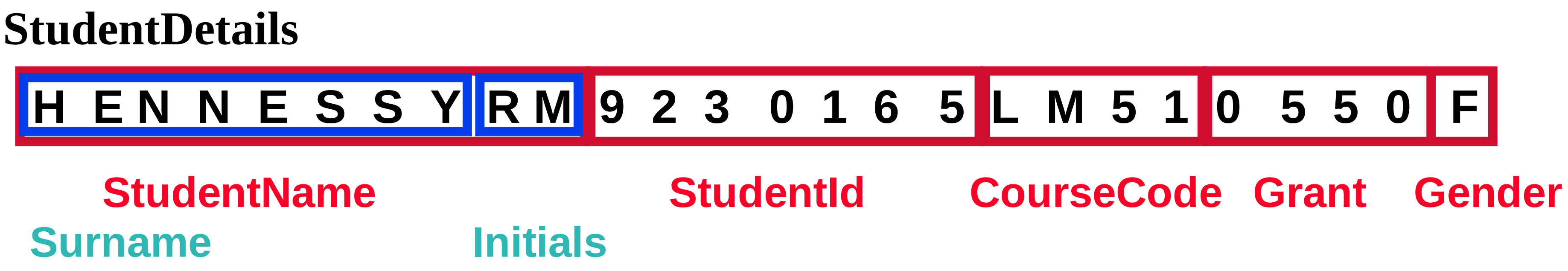
```
WORKING-STORAGE SECTION.
01  StudentDetails.
    02  StudentName        PIC X(10).
    02  StudentId          PIC 9(7).
    02  CourseCode         PIC X(4).
    02  Grant              PIC 9(4).
    02  Gender             PIC X.
```



COBOL Basics 2

Group Items/Records

```
WORKING-STORAGE SECTION.
01  StudentDetails.
    02  StudentName.
        03 Surname          PIC X(8).
        03 Initials         PIC XX.
    02  StudentId           PIC 9(7).
    02  CourseCode          PIC X(4).
    02  Grant               PIC 9(4).
    02  Gender              PIC X.
```



StudentDetails

H E N N E S S Y R M 9 2 3 0 1 6 5 L M 5 1 0 5 5 0 F

StudentName

Surname

Initials

StudentId

CourseCode

Grant

Gender

COBOL Basics 2

LEVEL Numbers express DATA hierarchy

- In COBOL, level numbers are used to decompose a structure into it’s constituent parts.
- In this hierarchical structure the higher the level number, the lower the item is in the hierarchy. At the lowest level the data is completely atomic.
- The level numbers 01 through 49 are general level numbers but there are also special level numbers such as 66, 77 and 88.
- In a hierarchical data description what is important is the relationship of the level numbers to one another, not the actual level numbers used.

```
01 StudentDetails.
   02 StudentName.
      03 Surname      PIC X(8).
      03 Initials     PIC XX.
   02 StudentId       PIC 9(7).
   02 CourseCode      PIC X(4).
   02 Grant           PIC 9(4).
   02 Gender          PIC X.
```

=

```
01 StudentDetails.
   05 StudentName.
      10 Surname      PIC X(8).
      10 Initials     PIC XX.
   05 StudentId       PIC 9(7).
   05 CourseCode      PIC X(4).
   05 Grant           PIC 9(4).
   05 Gender          PIC X.
```

COBOL Basics 2

Group and elementary items

- In COBOL the term "group item" is used to describe a data item which has been further subdivided.
  - A Group item is declared using a level number and a data name. It cannot have a picture clause.
  - Where a group item is the highest item in a data hierarchy it is referred to as a record and uses the level number 01.
- The term "elementary item" is used to describe data items which are atomic; that is, not further subdivided.
- An elementary item declaration consists of;

  1. a level number,
  2. a data name
  3. picture clause.

  An elementary item must have a picture clause.
- Every group or elementary item declaration must be followed by a full stop.

COBOL Basics 2

PICTUREs for Group Items

- Picture clauses are NOT specified for ‘group’ data items because the size a group item is the sum of the sizes of its subordinate, elementary items and its type is always assumed to be PIC X.
- The type of a group items is always assumed to be PIC X because group items may have several different data items and types subordinate to them.
- An X picture is the only one which could support such collections.

COBOL Basics 2

Assignment in COBOL

- In “strongly typed” languages like Modula-2, Pascal or ADA the assignment operation is simple because assignment is only allowed between data items with compatible types.
- The simplicity of assignment in these languages is achieved at the “cost” of having a large number of data types.
- In COBOL there are basically only three data types,

  Alphabetic (PIC A)

  Alphanumeric (PIC X)

  Numeric (PIC 9)
- But this simplicity is achieved only at the cost of having a very complex assignment statement.
- In COBOL assignment is achieved using the MOVE verb.

COBOL Basics 2

The MOVE Verb

$$\underline{\text{MOVE}}\begin{Bmatrix}\mathit{Identifier}\\ \mathit{Literal}\end{Bmatrix}\underline{\text{TO}}\{\mathit{Identifier}\}\ldots$$

- The MOVE copies data from the source identifier or literal to one or more destination identifiers.
- The source and destination identifiers can be group or elementary data items.
- When the destination item is alphanumeric or alphabetic (PIC X or A) data is copied into the destination area from left to right with space filling or truncation on the right.
- When data is MOVEd into an item the contents of the item are completely replaced. If the source data is too small to fill the destination item entirely the remaining area is zero or space filled.

COBOL Basics 2

MOVEing Data

```
MOVE "RYAN" TO Surname.
MOVE "FITZPATRICK" TO Surname.
```

```
01 Surname   PIC X(8).
```

C	O	U	G	H	L	A	N

COBOL Basics 2

MOVEing Data

```
MOVE “RYAN” TO Surname.
MOVE “FITZPATRICK” TO Surname.
```

```
01 Surname   PIC X(8).
```

R	Y	A	N				

COBOL Basics 2

MOVEing Data

```
MOVE "RYAN" TO Surname.
MOVE "FITZPATRICK" TO Surname.
```

```
01 Surname   PIC X(8).
```

F	I	T	Z	P	A	T	R	I	C	K

COBOL Basics 2

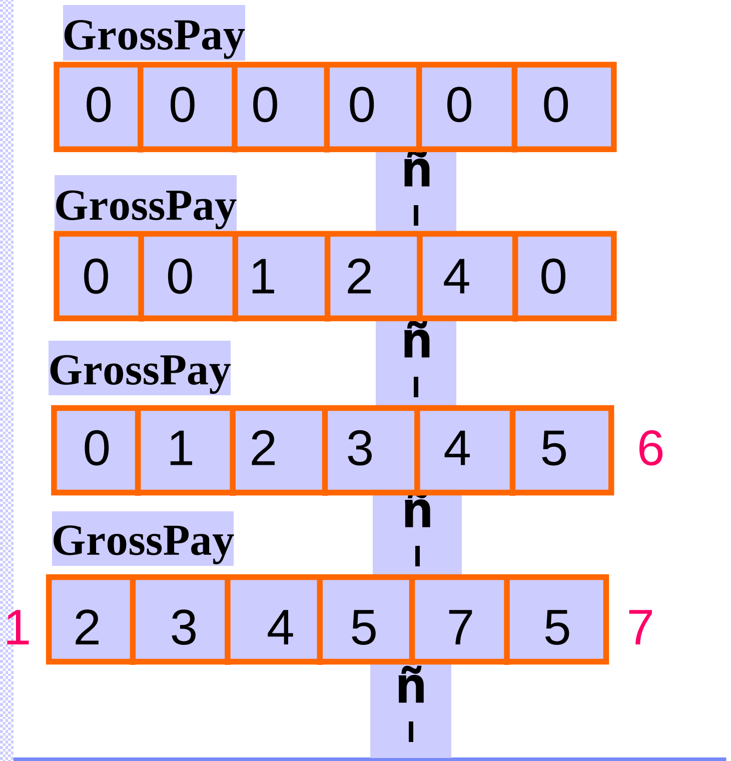
MOVEing to a numeric item

- When the destination item is numeric, or edited numeric, then data is aligned along the decimal point with zero filling or truncation as necessary.
- When the decimal point is not explicitly specified in either the source or destination items, the item is treated as if it had an assumed decimal point immediately after its rightmost character.

COBOL Basics 2

MOVEing to a numeric item

```
01 GrossPay                  PIC 9(4)V99.

MOVE ZEROS TO GrossPay.

MOVE 12.4 TO GrossPay.

MOVE 123.456 TO GrossPay.

MOVE 12345.757 TO GrossPay.
```



GrossPay

0	0	0	0	0	0

ñ ı

GrossPay

0	0	1	2	4	0

ñ ı

GrossPay

0	1	2	3	4	5	6

ñ ı

GrossPay

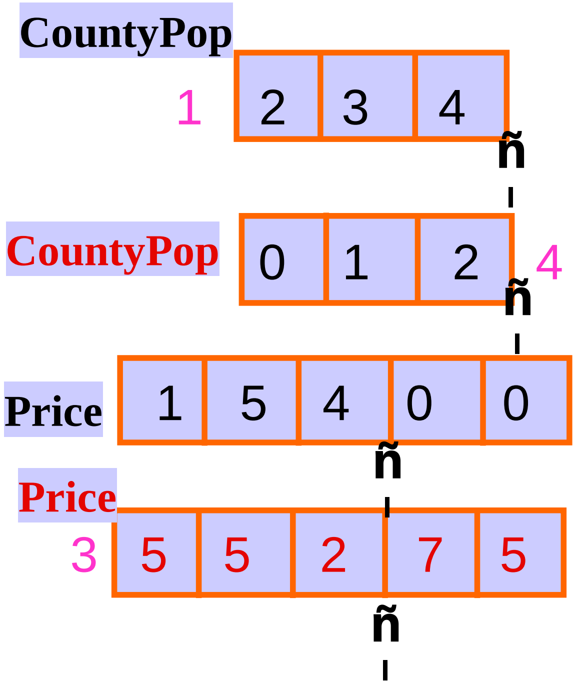
1	2	3	4	5	7	5	7

ñ ı

COBOL Basics 2

MOVEing to a numeric item

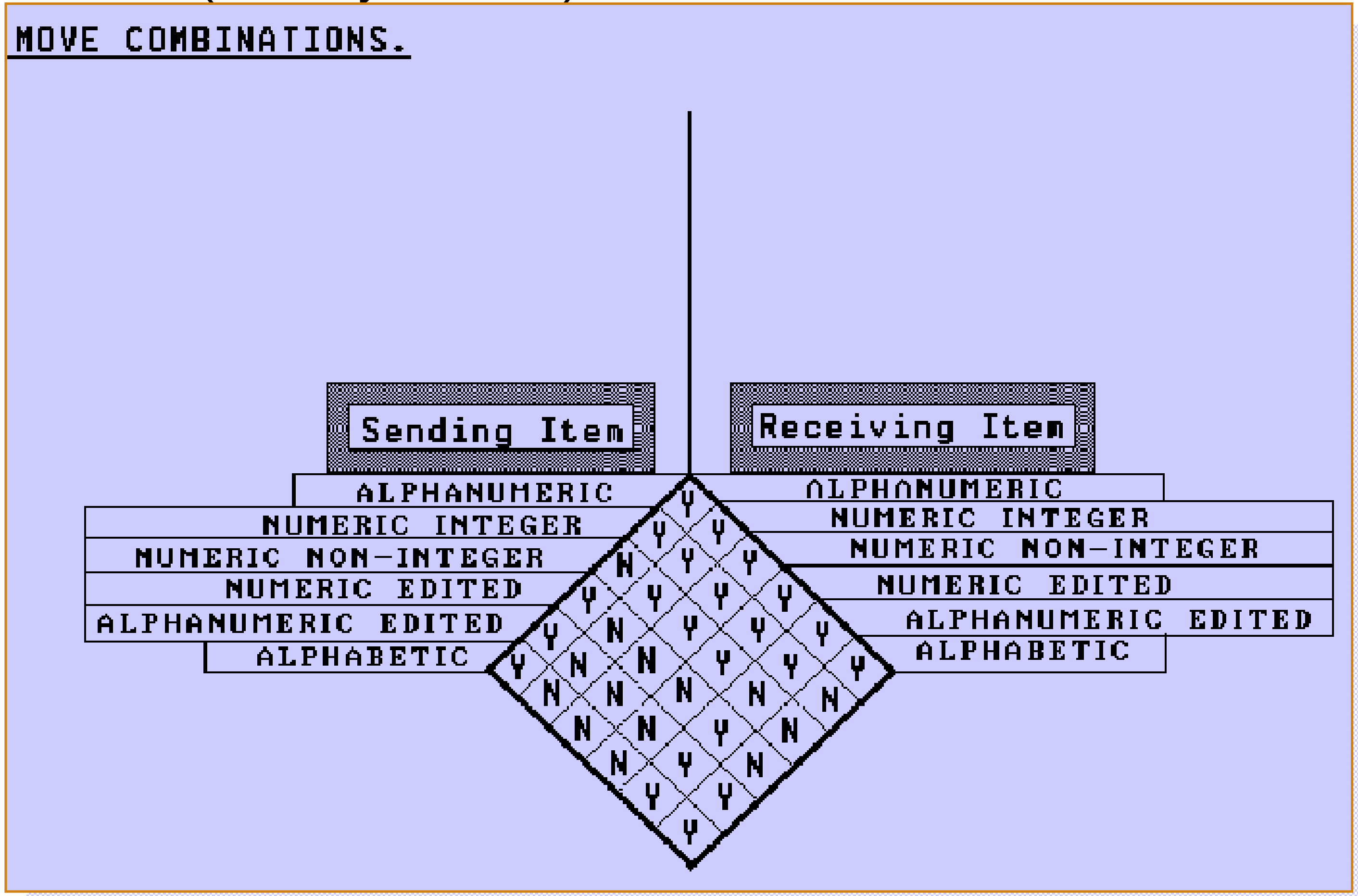
```
  01 CountyPop                PIC 999.
  01 Price                    PIC 999V99.


MOVE 1234 TO CountyPop.

MOVE 12.4 TO CountyPop.

MOVE 154 TO Price.

MOVE 3552.75 TO Price.
```



COBOL Basics 2

Legal MOVEs

Certain combinations of sending and receiving data types are not permitted (even by COBOL).



MOVE COMBINATIONS.

Receiving Item

Sending Item	ALPHANUMERIC	NUMERIC INTEGER	NUMERIC NON-INTEGER	NUMERIC EDITED	ALPHANUMERIC EDITED	ALPHABETIC
ALPHANUMERIC	Y	Y	Y	Y	Y	Y
NUMERIC INTEGER	Y	Y	Y	Y	Y	N
NUMERIC NON-INTEGER	N	Y	Y	Y	N	N
NUMERIC EDITED	Y	N	N	N	Y	N
ALPHANUMERIC EDITED	Y	N	N	N	Y	Y
ALPHABETIC	Y	N	N	N	Y	Y

COBOL Basics 2

The DISPLAY Verb

$$\underline{\mathit{DISPLAY}} \begin{Bmatrix} \mathit{Identifier} \\ \mathit{Literal} \end{Bmatrix} \left[\begin{Bmatrix} \mathit{Identifier} \\ \mathit{Literal} \end{Bmatrix} \right] \ldots$$

$$\left[\underline{\mathit{UPON}}\ \text{Mnemonic - Name}\right]\left[\mathit{WITH}\ \ \underline{\mathit{NO}}\ \underline{\mathit{ADVANCING}}\right]$$

- From time to time it may be useful to display messages and data values on the screen.
- A simple DISPLAY statement can be used to achieve this.
- A single DISPLAY can be used to display several data items or literals or any combination of these.
- The WITH NO ADVANCING clause suppresses the carriage return/line feed.

COBOL Basics 2

The ACCEPT verb

$$\text{Format 1. } \underline{\text{ACCEPT}}\ \text{Identifier}\ \left[\underline{\text{FROM}}\ \text{Mnemonic - name}\right]$$

$$\text{Format 2. } \underline{\text{ACCEPT}}\ \text{Identifier}\ \underline{\text{FROM}}\ \left\{\begin{array}{l}\underline{\text{DATE}}\\ \underline{\text{DAY}}\\ \underline{\text{DAY - OF - WEEK}}\\ \underline{\text{TIME}}\end{array}\right\}$$

```
01 CurrentDate              PIC 9(6).
*  YYMMDD

01 DayOfYear                PIC 9(5).
*  YYDDD

01 DayOfWeek                PIC 9.
*  D (1=Monday)

01 CurrentTime              PIC 9(8).
*  HHMMSSss    s = S/100
```

COBOL Basics 2

Run of Accept and Display program

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  AcceptAndDisplay.
AUTHOR.  Michael Coughlan.

DATA DIVISION.
WORKING-STORAGE SECTION.
01 StudentDetails.
   02  StudentName.
       03 Surname        PIC X(8).
       03 Initials       PIC XX.
   02  StudentId         PIC 9(7).
   02  CourseCode        PIC X(4).
   02  Grant             PIC 9(4).
   02  Gender            PIC X.

01 CurrentDate.
   02  CurrentYear       PIC 99.
   02  CurrentMonth      PIC 99.
   02  CurrentDay        PIC 99.

01 DayOfYear.
   02  FILLER            PIC 99.
   02  YearDay           PIC 9(3).

01 CurrentTime.
   02  CurrentHour       PIC 99.
   02  CurrentMinute     PIC 99.
   02  FILLER            PIC 9(4).
```

```
PROCEDURE DIVISION.
Begin.
    DISPLAY "Enter student details using template below".
    DISPLAY "NNNNNNNNNNSSSSSSSCCCCGGGGS                 ".
    ACCEPT  StudentDetails.
    ACCEPT  CurrentDate FROM DATE.
    ACCEPT  DayOfYear FROM DAY.
    ACCEPT  CurrentTime FROM TIME.
    DISPLAY "Name is ", Initials SPACE Surname.
    DISPLAY "Date is " CurrentDay SPACE CurrentMonth SPACE CurrentYear.
    DISPLAY "Today is day " YearDay " of the year".
    DISPLAY "The time is " CurrentHour ":" CurrentMinute.
    STOP RUN.
```

```
Enter student details using template below
NNNNNNNNNNSSSSSSSCCCCGGGGS
COUGHLANMS9476532LM511245M
Name is MS COUGHLAN
Date is 24 01 94
Today is day 024 of the year
The time is 22:23
```

Table of contents

Introduction to COBOL
COBOL Basics 1
COBOL Basics 2
➜ Introduction to Sequential Files
Processing Sequential Files
Simple iteration with the PERFORM verb
Arithmetic and Edited Pictures
Conditions
Tables and the PERFORM ... VARYING
Designing Programs

Introduction to Sequential Files

Overview

- Files, records, fields.
- The record buffer concept.
- The SELECT and ASSIGN clause.
- OPEN, CLOSE, READ and WRITE verbs.

Introduction to Sequential Files

COBOL's forte

- COBOL is generally used in situations where the volume of data to be processed is large.
- These systems are sometimes referred to as “*data intensive*” systems.
- Generally, large volumes of data arise not because the data is inherently voluminous but because the same items of information have been recorded about a great many instances of the same object.

Introduction to Sequential Files

Files, Records, Fields

- We use the term FIELD to describe an item of information we are recording about an object

  (e.g. StudentName, DateOfBirth, CourseCode).

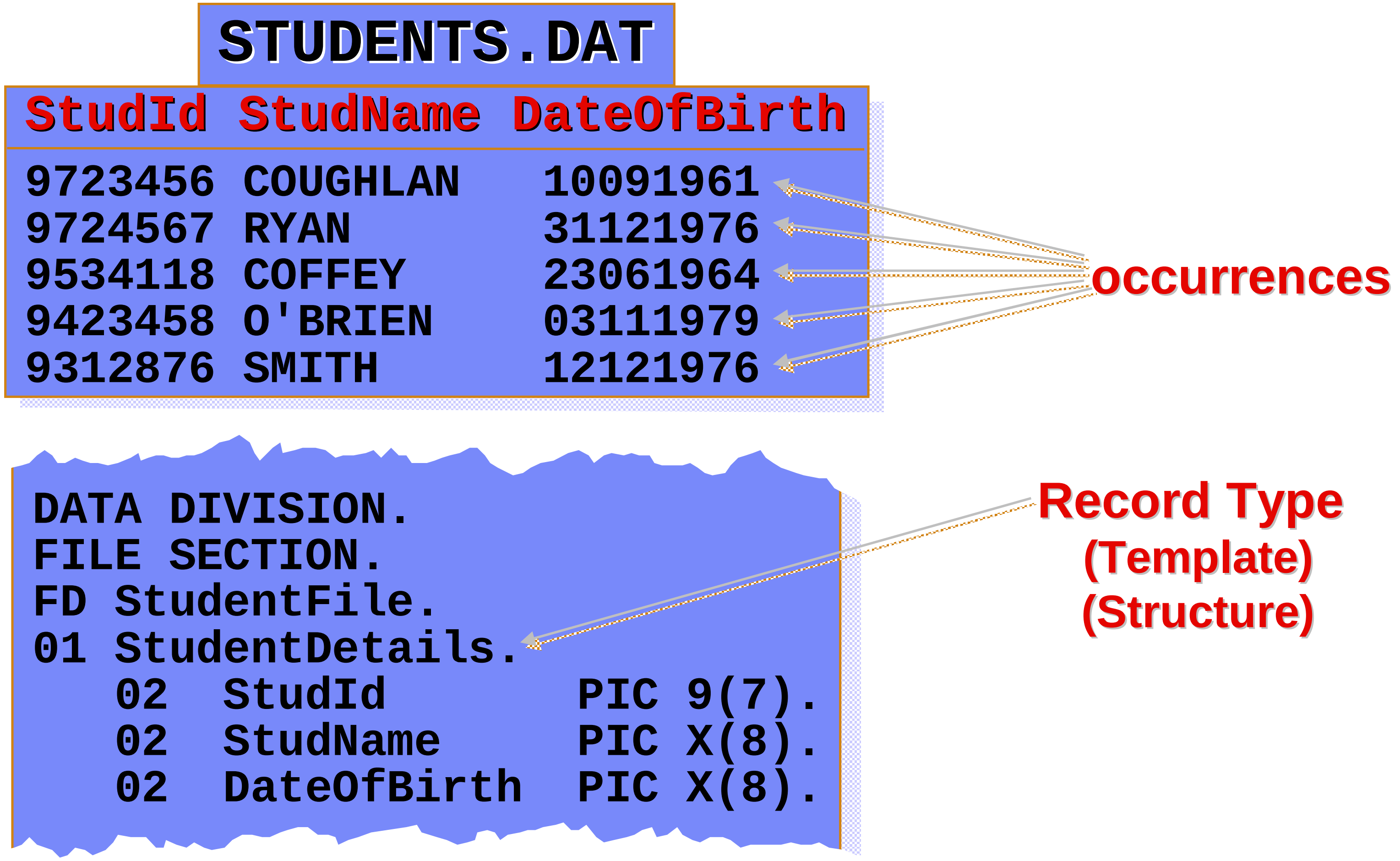
- We use the term RECORD to describe the collection of fields which record information about an object

  (e.g. a StudentRecord is a collection of fields recording information about a student).

- We use the term FILE to describe a collection of one or more occurrences (instances) of a record type (template).
- It is important to distinguish between the record occurrence (i.e. the values of a record) and the record type (i.e. the structure of the record). Every record in a file has a different value but the same structure.

Introduction to Sequential Files

Files, Records, Fields



STUDENTS.DAT

StudId	StudName	DateOfBirth
9723456	COUGHLAN	10091961
9724567	RYAN	31121976
9534118	COFFEY	23061964
9423458	O'BRIEN	03111979
9312876	SMITH	12121976

occurrences

```
DATA DIVISION.
FILE SECTION.
FD StudentFile.
01 StudentDetails.
   02  StudId        PIC 9(7).
   02  StudName      PIC X(8).
   02  DateOfBirth   PIC X(8).
```

Record Type
(Template)
(Structure)

Introduction to Sequential Files

How files are processed

- Files are repositories of data that reside on backing storage (hard disk or magnetic tape).
- A file may consist of hundreds of thousands or even millions of records.
- Suppose we want to keep information about all the TV license holders in the country. Suppose each record is about 150 characters/bytes long. If we estimate the number of licenses at 1 million this gives us a size for the file of 150 X 1,000,000 = 150 megabytes.
- If we want to process a file of this size we cannot do it by loading the whole file into the computer's memory at once.
- Files are processed by reading them into the computer's memory one record at a time.

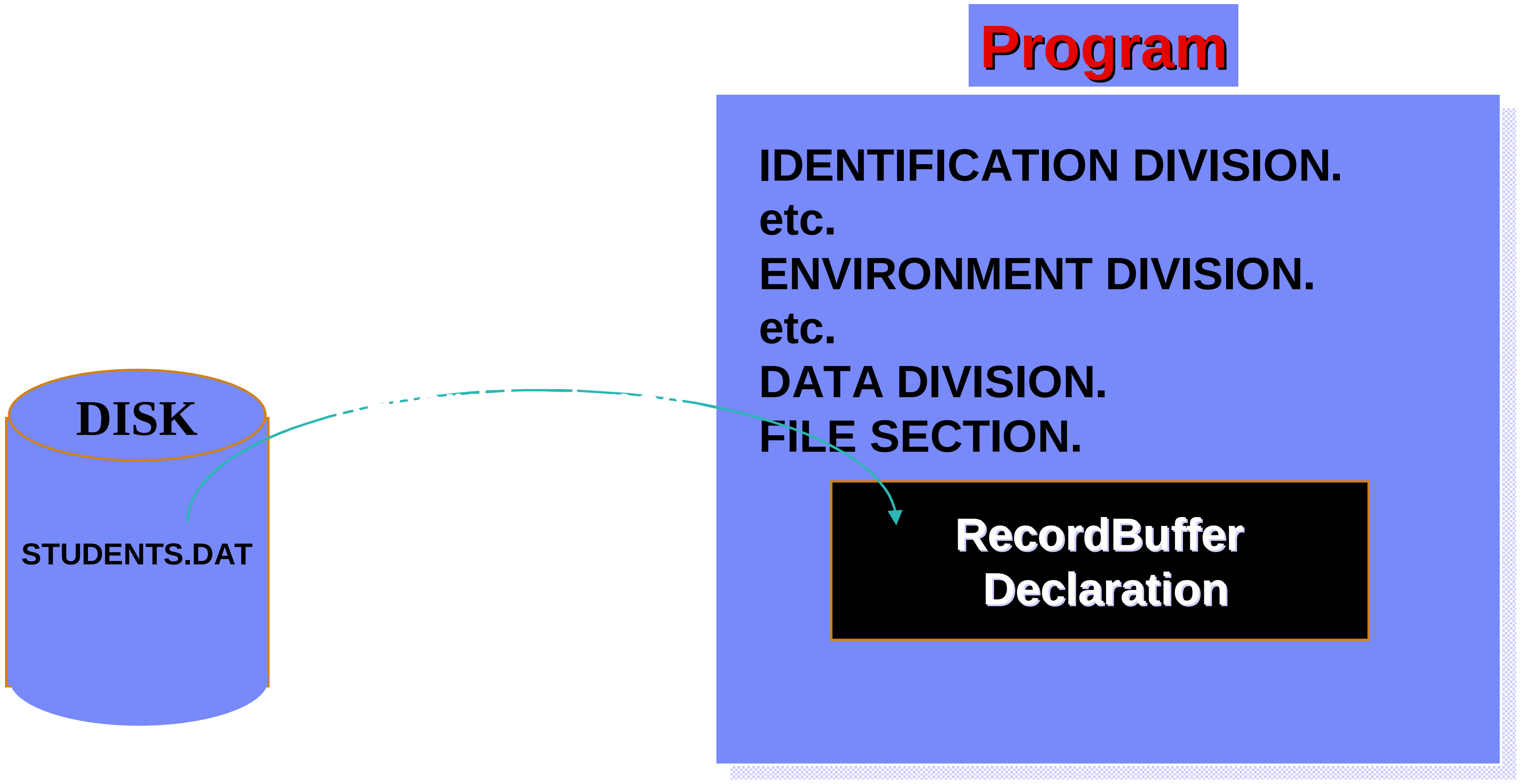
Introduction to Sequential Files

Record Buffers

- To process a file records are read from the file into the computer’s memory one record at a time.
- The computer uses the programmers description of the record (i.e. the record template) to set aside sufficient memory to store one instance of the record.
- Memory allocated for storing a record is usually called a “record buffer”
- The record buffer is the only connection between the program and the records in the file.

Introduction to Sequential Files

Record Buffers



Introduction to Sequential Files

Implications of 'Buffers'

- If your program processes more than one file you will have to describe a record buffer for each file.
- To process all the records in an INPUT file each record instance must be copied (read) from the file into the record buffer when required.
- To create an OUTPUT file containing data records each record must be placed in the record buffer and then transferred (written) to the file.
- To transfer a record from an input file to an output file we will have to

  read the record into the input record buffer

  transfer it to the output record buffer

  write the data to the output file from the output record buffer

Introduction to Sequential Files

Creating a Student Record

Student Details.

	`01 StudentDetails.`
§ Student Id.	`02 StudentId PIC 9(7).`
§ Student Name.	`02 StudentName.`
Surname	`03 Surname PIC X(8).`
Initials	`03 Initials PIC XX.`
§ Date of Birth	`02 DateOfBirth.`
Year of Birth	`03 YOBirth PIC 99.`
Month of Birth	`03 MOBirth PIC 99.`
Day of Birth	`03 DOBirth PIC 99.`
§ Course Code	`02 CourseCode PIC X(4).`
§ Value of grant	`02 Grant PIC 9(4).`
§ Gender	`02 Gender PIC X.`

Introduction to Sequential Files

Describing the record buffer in COBOL

```
DATA DIVISION.
FILE SECTION.
FD StudentFile.
01 StudentDetails.
   02  StudentId          PIC 9(7).
   02  StudentName.
       03 Surname         PIC X(8).
       03 Initials        PIC XX.
   02  DateOfBirth.
       03 YOBirth         PIC 9(2).
       03 MOBirth         PIC 9(2).
       03 DOBirth         PIC 9(2).
   02  CourseCode         PIC X(4).
   02  Grant              PIC 9(4).
   02  Gender             PIC X.
```

- The record type/template/buffer of every file used in a program must be described in the FILE SECTION by means of an FD (file description) entry.

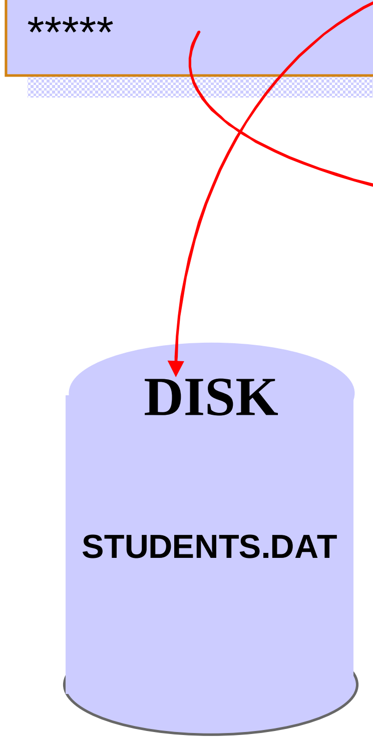
- The FD entry consists of the letters FD and an internal file name.

Introduction to Sequential Files

The Select and Assign Clause

```
//STEP160  EXEC  PGM=BUAC25,COND=(
//STUDENTS  DD  DSN= STUDENTS.DAT,
*****
```

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT StudentFile
        ASSIGN TO “STUDENTS”.

DATA DIVISION.
FILE SECTION.
FD StudentFile.
01 StudentDetails.
   02  StudentId          PIC 9(7).
   02  StudentName.
       03 Surname         PIC X(8).
       03 Initials        PIC XX.
   02  DateOfBirth.
       03 YOBirth         PIC 9(2).
       03 MOBirth         PIC 9(2).
       03 DOBirth         PIC 9(2).
   *********
```



- The internal file name used in the FD entry is connected to an external file (on disk or tape) by means of the Select and Assign clause.

Introduction to Sequential Files

Select and Assign Syntax

$$\underline{\text{SELECT}}\ \ \text{FileName}\,\underline{\text{ASSIGN}}\ \text{TO ExternalFileReference}$$

$$[\underline{\text{ORGANIZATION}}\ \text{IS}\begin{Bmatrix}\underline{\text{LINE}}\\ \underline{\text{RECORD}}\end{Bmatrix}\underline{\text{SEQUENTIAL}}].$$

- LINE SEQUENTIAL means each record is followed by the carriage return and line feed characters.

- RECORD SEQUENTIAL means that the file consists of a stream of bytes. Only the fact that we know the size of each record allows us to retrieve them.

Introduction to Sequential Files

COBOL file handling Verbs

- OPEN
  Before your program can access the data in an input file or place data in an output file you must make the file available to the program by OPENing it.

- READ
  The READ copies a record occurrence/instance from the file and places it in the record buffer.

- WRITE
  The WRITE copies the record it finds in the record buffer to the file.

- CLOSE
  You must ensure that (before terminating) your program closes all the files it has opened. Failure to do so may result in data not being written to the file or users being prevented from accessing the file.

Introduction to Sequential Files

OPEN and CLOSE verb syntax

$$\underline{\text{OPEN}} \left\{ \left\{ \begin{array}{l} \underline{\textit{INPUT}} \\ \underline{\textit{OUTPUT}} \\ \underline{\textit{EXTEND}} \end{array} \right\} \text{InternalFileName} \right\} \ldots$$

- When you open a file you have to indicate to the system what how you want to use it (e.g. INPUT, OUTPUT, EXTEND) so that the system can manage the file correctly.

- Opening a file does not transfer any data to the record buffer, it simply provides access.

Introduction to Sequential Files

The READ verb

- Once the system has opened a file and made it available to the program it is the programmers responsibility to process it correctly.
- Remember, the file record buffer is our only connection with the file and it is only able to store a single record at a time.
- To process all the records in the file we have to transfer them, one record at a time, from the file to the buffer.
- COBOL provides the READ verb for this purpose.

Introduction to Sequential Files

READ verb syntax

```
READ InternalFilename [NEXT] RECORD
          [INTO Identifier]
           AT END StatementBlock
END-READ
```

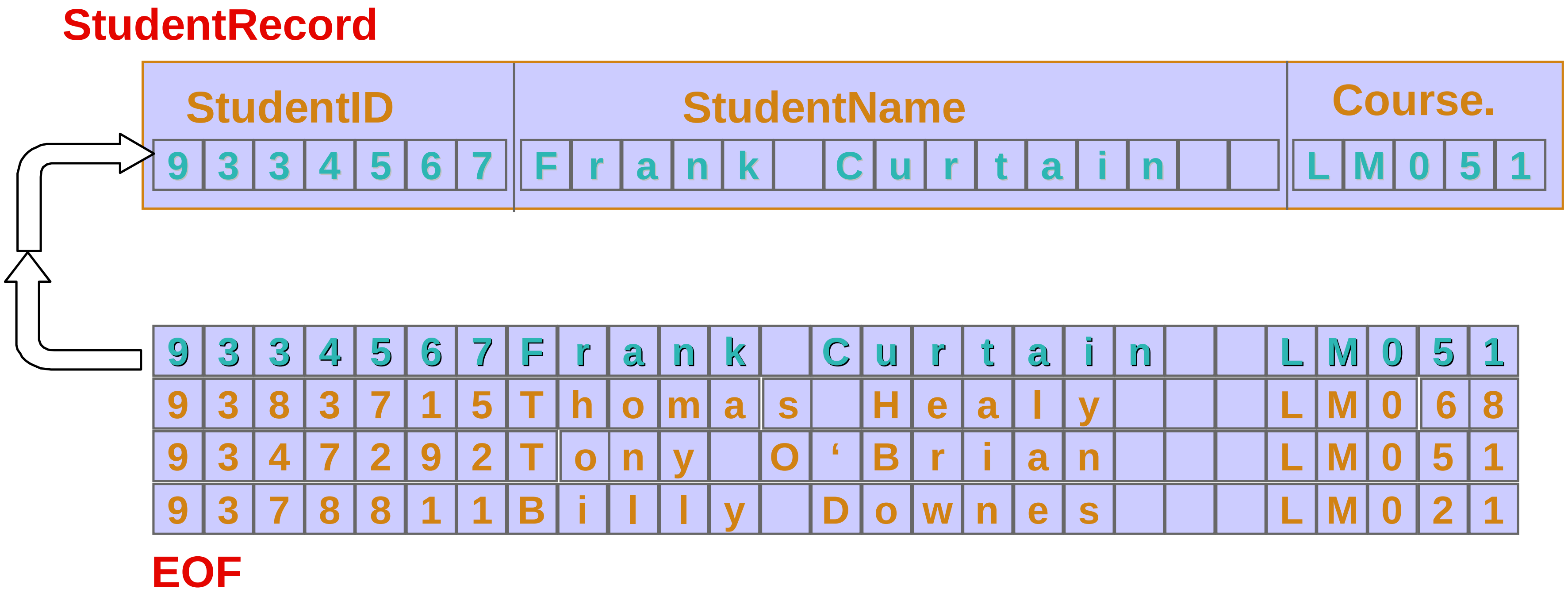
- The InternalFilename specified must be a file that has been OPENed for INPUT.

- The NEXT RECORD clause is optional and generally not used.

- Using INTO Identifier clause causes the data to be read into the record buffer and then copied from there to the specified Identifier in one operation.
  - When this option is used there will be two copies of the data. It is the equivalent of a READ followed by a MOVE.

Introduction to Sequential Files

How the READ works

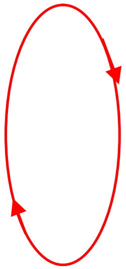


StudentRecord

StudentID	StudentName	Course.
9 3 3 4 5 6 7	F r a n k C u r t a i n	L M 0 5 1

9	3	3	4	5	6	7	F	r	a	n	k		C	u	r	t	a	i	n			L	M	0	5	1
9	3	8	3	7	1	5	T	h	o	m	a	s		H	e	a	l	y				L	M	0	6	8
9	3	4	7	2	9	2	T	o	n	y		O	‘	B	r	i	a	n				L	M	0	5	1
9	3	7	8	8	1	1	B	i	l	l	y		D	o	w	n	e	s				L	M	0	2	1

EOF

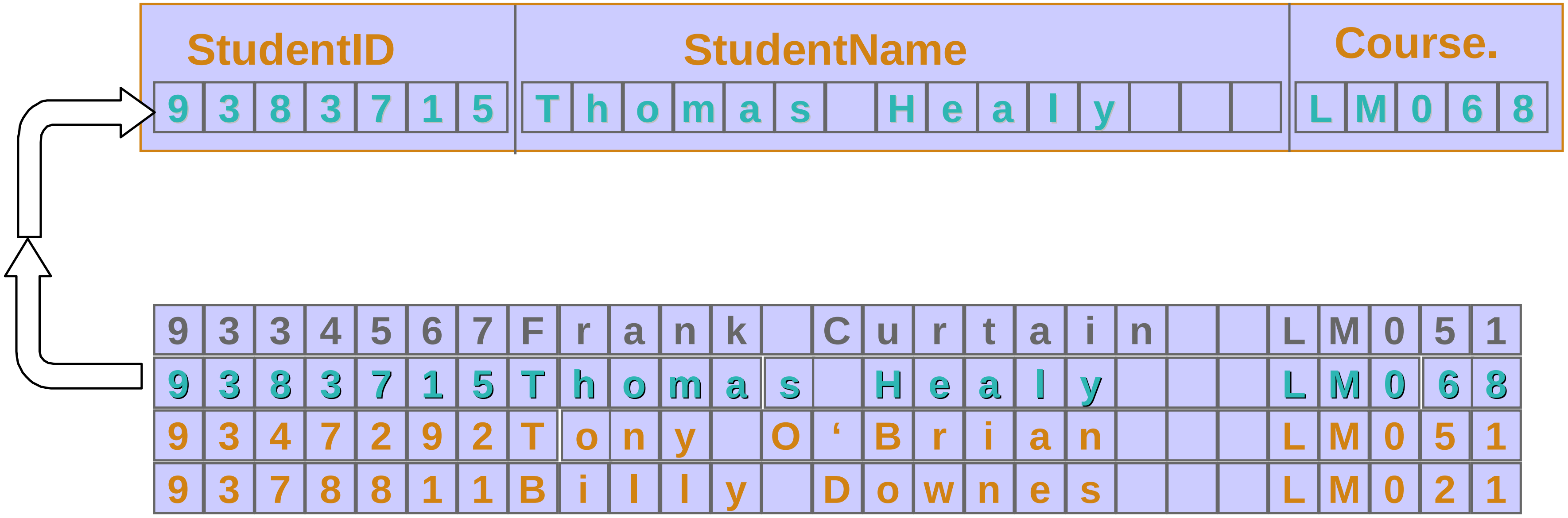


```
PERFORM UNTIL StudentRecord = HIGH-VALUES
    READ StudentRecords
        AT END MOVE HIGH-VALUES TO StudentRecord
    END-READ
END-PERFORM.
```

Introduction to Sequential Files

How the READ works

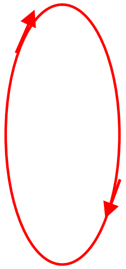
StudentRecord



StudentID	StudentName	Course.
9383715	Thomas Healy	LM068

9	3	3	4	5	6	7	F	r	a	n	k		C	u	r	t	a	i	n			L	M	0	5	1
9	3	8	3	7	1	5	T	h	o	m	a	s		H	e	a	l	y				L	M	0	6	8
9	3	4	7	2	9	2	T	o	n	y		O	‘	B	r	i	a	n				L	M	0	5	1
9	3	7	8	8	1	1	B	i	l	l	y		D	o	w	n	e	s				L	M	0	2	1

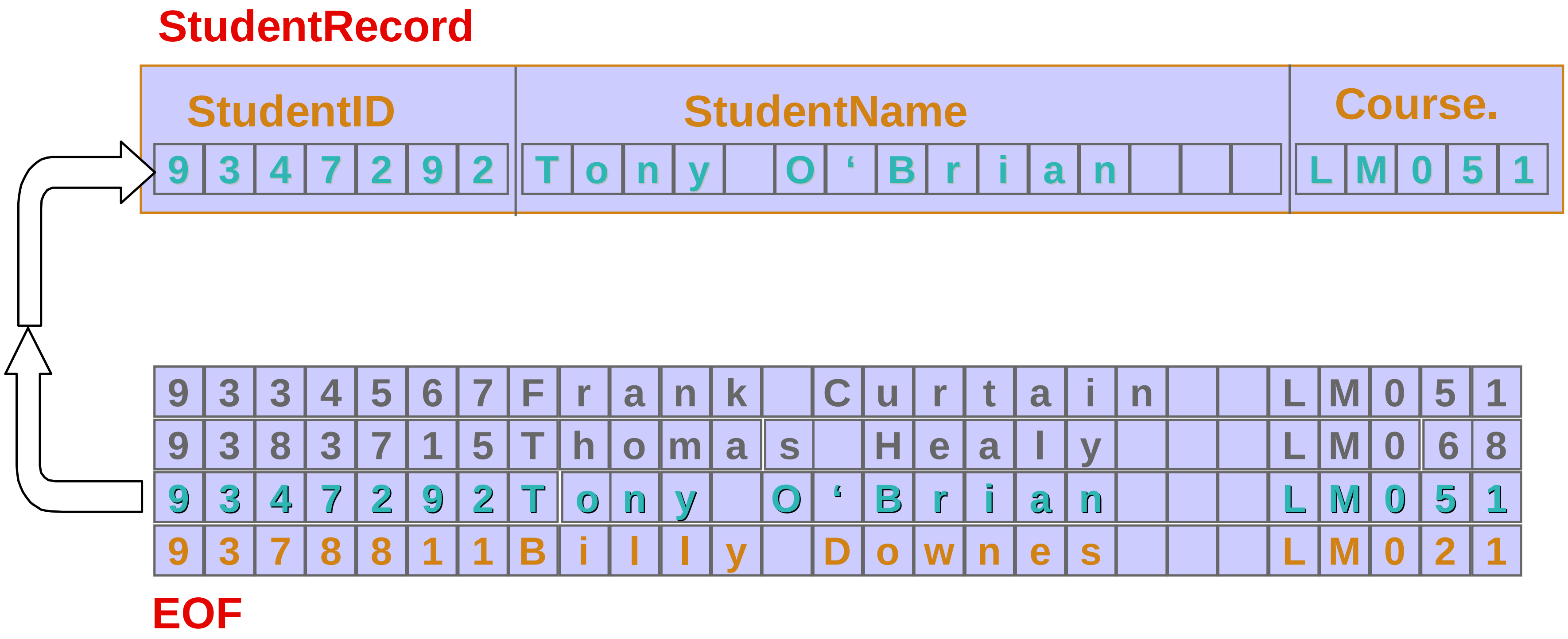
EOF



```
PERFORM UNTIL StudentRecord = HIGH-VALUES
    READ StudentRecords
        AT END MOVE HIGH-VALUES TO StudentRecord
    END-READ
END-PERFORM.
```

Introduction to Sequential Files

How the READ works



StudentRecord

StudentID	StudentName	Course.
9347292	Tony O‘Brian	LM051

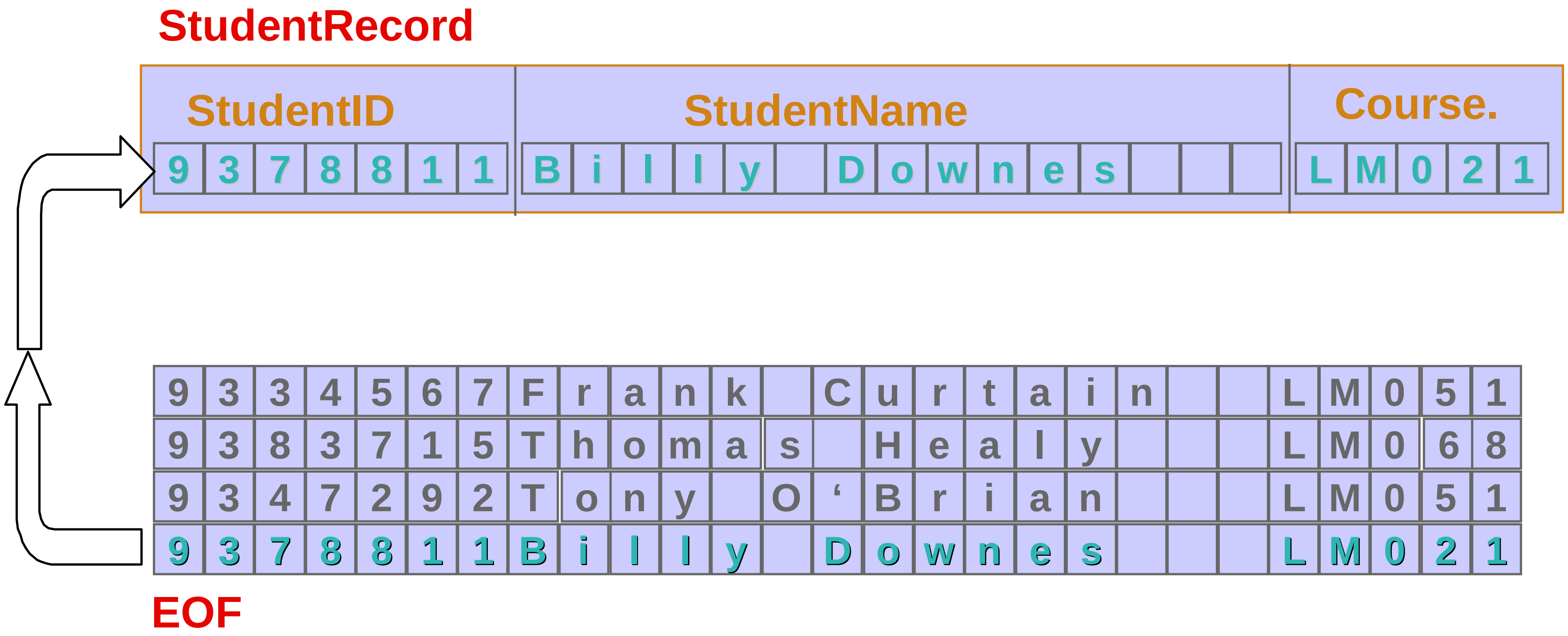
9334567	Frank Curtain	LM051
9383715	Thomas Healy	LM068
9347292	Tony O‘Brian	LM051
9378811	Billy Downes	LM021

EOF

```
PERFORM UNTIL StudentRecord = HIGH-VALUES
    READ StudentRecords
        AT END MOVE HIGH-VALUES TO StudentRecord
    END-READ
END-PERFORM.
```

Introduction to Sequential Files

How the READ works



StudentRecord

StudentID	StudentName	Course.
9378811	Billy Downes	LM021

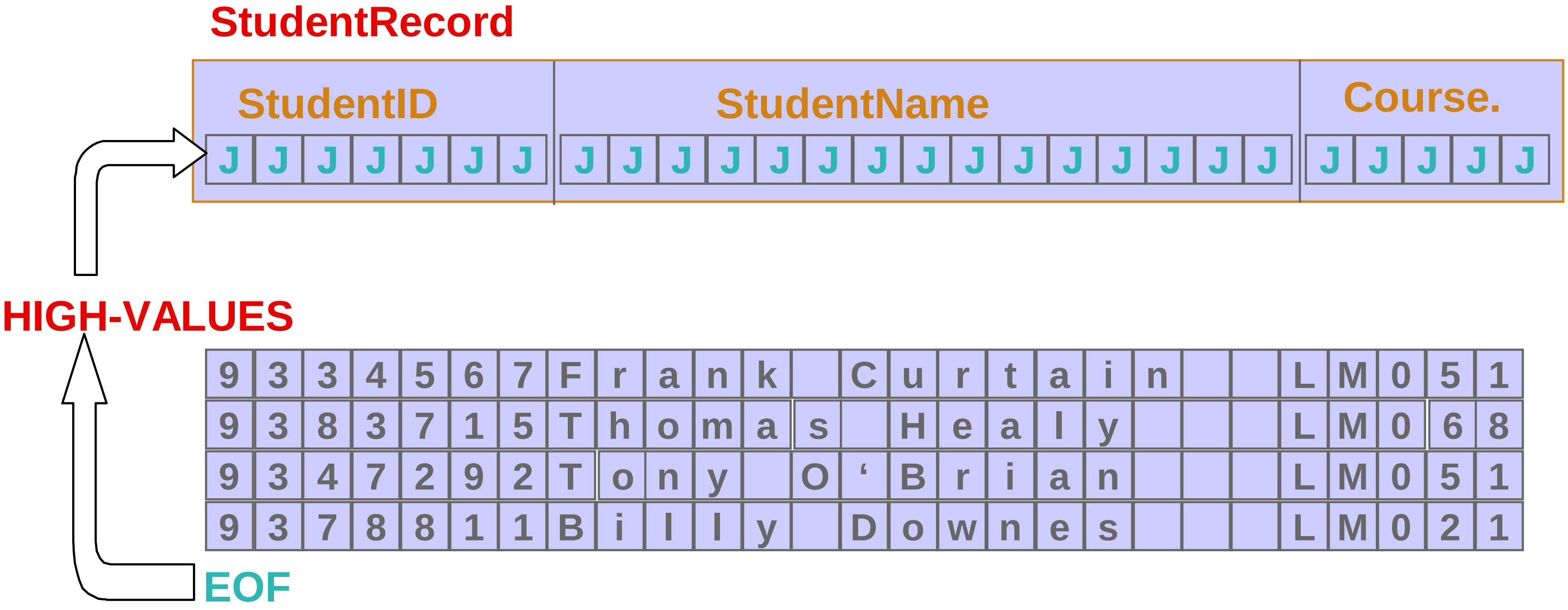
9	3	3	4	5	6	7	F	r	a	n	k		C	u	r	t	a	i	n			L	M	0	5	1
9	3	8	3	7	1	5	T	h	o	m	a	s		H	e	a	l	y				L	M	0	6	8
9	3	4	7	2	9	2	T	o	n	y		O	‘	B	r	i	a	n				L	M	0	5	1
9	3	7	8	8	1	1	B	i	l	l	y		D	o	w	n	e	s				L	M	0	2	1

EOF

```
PERFORM UNTIL StudentRecord = HIGH-VALUES
    READ StudentRecords
        AT END MOVE HIGH-VALUES TO StudentRecord
    END-READ
END-PERFORM.
```

Introduction to Sequential Files

How the READ works



StudentRecord

StudentID							StudentName														Course.				
J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J	J

HIGH-VALUES

9	3	3	4	5	6	7	F	r	a	n	k		C	u	r	t	a	i	n			L	M	0	5	1
9	3	8	3	7	1	5	T	h	o	m	a	s		H	e	a	l	y				L	M	0	6	8
9	3	4	7	2	9	2	T	o	n	y		O	‘	B	r	i	a	n				L	M	0	5	1
9	3	7	8	8	1	1	B	i	l	l	y		D	o	w	n	e	s				L	M	0	2	1

EOF

```
PERFORM UNTIL StudentRecord = HIGH-VALUES
    READ StudentRecords
        AT END MOVE HIGH-VALUES TO StudentRecord
    END-READ
END-PERFORM.
```

Introduction to Sequential Files

WRITE Syntax

$$\underline{\textit{WRITE}}\ \ \text{RecordName}\ \ \left[\underline{\text{FROM}}\ \ \text{Identifier}\right]$$

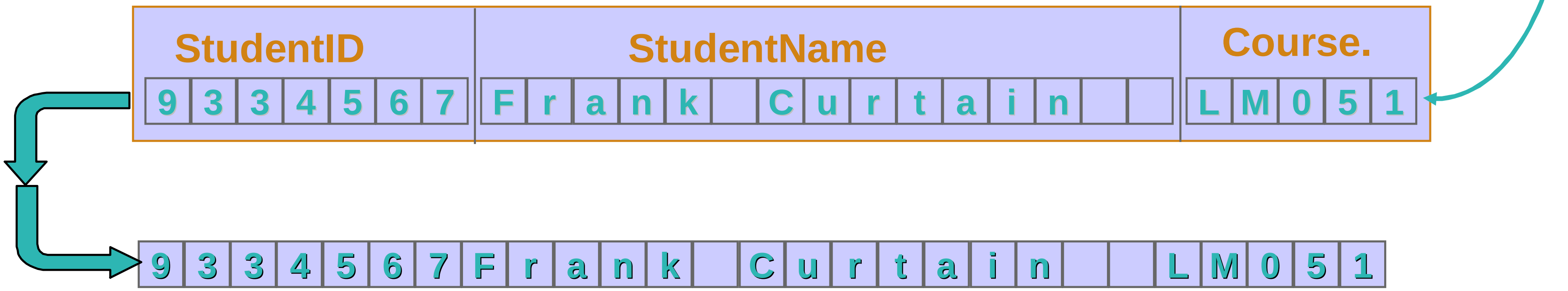
$$\left[\left\{\begin{array}{l}\underline{\text{BEFORE}}\\ \underline{\text{AFTER}}\end{array}\right\}\textit{ADVANCING}\ \left\{\begin{array}{l}\textit{AdvanceNum}\ \left[\begin{array}{l}\underline{\text{LINE}}\\ \underline{\text{LINES}}\end{array}\right]\\ \textit{MnemonicNa}\ \ \textit{me}\\ \underline{\textit{PAGE}}\end{array}\right\}\right]$$

- To WRITE data to a file move the data to the record buffer (declared in the FD entry) and then WRITE the contents of record buffer to the file.

Introduction to Sequential Files

How the WRITE works

```
OPEN OUTPUT StudentFile.
MOVE "9334567Frank Curtain  LM051" TO StudentDetails.
WRITE StudentDetails.
MOVE "9383715Thomas Healy   LM068" TO StudentDetails.
WRITE StudentDetails.
CLOSE StudentFile.
STOP RUN.
```



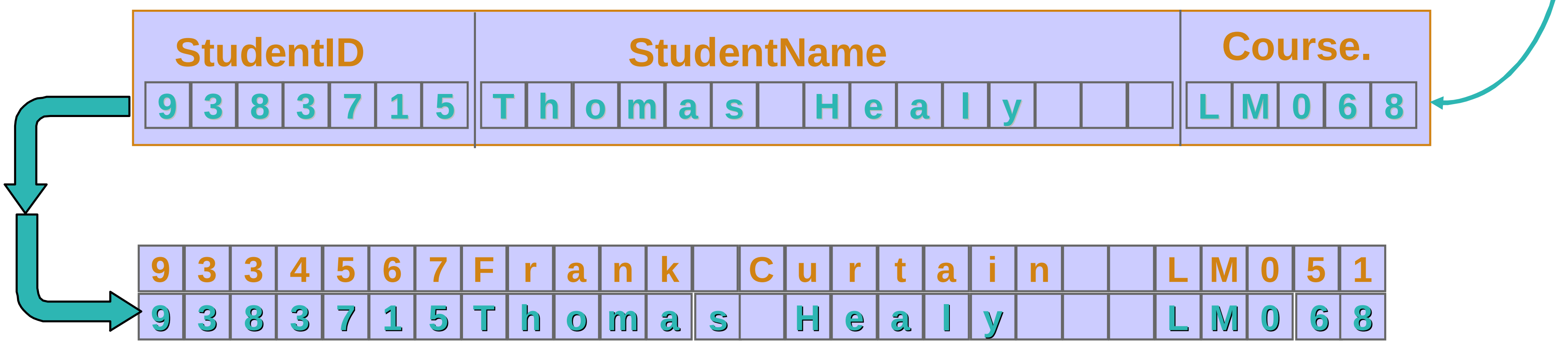
StudentID							StudentName														Course.				
9	3	3	4	5	6	7	F	r	a	n	k		C	u	r	t	a	i	n			L	M	0	5	1

9	3	3	4	5	6	7	F	r	a	n	k		C	u	r	t	a	i	n			L	M	0	5	1

Introduction to Sequential Files

How the WRITE works

```
OPEN OUTPUT StudentFile.
MOVE "9334567Frank Curtain  LM051" TO StudentDetails.
WRITE StudentDetails.
MOVE "9383715Thomas Healy   LM068" TO StudentDetails.
WRITE StudentDetails.
CLOSE StudentFile.
STOP RUN.
```



StudentID	StudentName	Course.
9383715	Thomas Healy	LM068

```
9334567Frank Curtain  LM051
9383715Thomas Healy   LM068
```

Introduction to Sequential Files

Sample Code

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  SeqWrite.
AUTHOR.  Michael Coughlan.

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT Student ASSIGN TO STUDENTS
        ORGANIZATION IS LINE SEQUENTIAL.

DATA DIVISION.
FILE SECTION.
FD Student.
01 StudentDetails.
   02  StudentId       PIC 9(7).
   02  StudentName.
       03 Surname      PIC X(8).
       03 Initials     PIC XX.
   02  DateOfBirth.
       03 YOBirth      PIC 9(2).
       03 MOBirth      PIC 9(2).
       03 DOBirth      PIC 9(2).
   02  CourseCode      PIC X(4).
   02  Grant           PIC 9(4).
   02  Gender          PIC X.

PROCEDURE DIVISION.
Begin.
   OPEN OUTPUT Student.
   DISPLAY "Enter student details using template below.  Enter no data to end.".
   PERFORM GetStudentDetails.
   PERFORM UNTIL StudentDetails = SPACES
      WRITE StudentDetails
      PERFORM GetStudentDetails
   END-PERFORM.
   CLOSE Student.
   STOP RUN.

GetStudentDetails.
   DISPLAY "NNNNNNNSSSSSSSSIIYYMMDDCCCCGGGGS".
   ACCEPT  StudentDetails.
```

Introduction to Sequential Files

Sample Code

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  SeqRead.
AUTHOR.  Michael Coughlan.

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT Student ASSIGN TO STUDENTS
        ORGANIZATION IS LINE SEQUENTIAL.

DATA DIVISION.
FILE SECTION.
FD Student.
01 StudentDetails.
   02  StudentId       PIC 9(7).
   02  StudentName.
       03 Surname      PIC X(8).
       03 Initials     PIC XX.
   02  DateOfBirth.
       03 YOBirth      PIC 9(2).
       03 MOBirth      PIC 9(2).
       03 DOBirth      PIC 9(2).
   02  CourseCode      PIC X(4).
   02  Grant           PIC 9(4).
   02  Gender          PIC X.

PROCEDURE DIVISION.
Begin.
   OPEN INPUT Student
   READ Student
      AT END MOVE HIGH-VALUES TO StudentDetails
   END-READ
   PERFORM UNTIL StudentDetails = HIGH-VALUES
   DISPLAY StudentId SPACE StudentName SPACE CourseCode
   READ Student
      AT END MOVE HIGH-VALUES TO StudentDetails
   END-READ
   END-PERFORM
   CLOSE Student
   STOP RUN.
```

Table of contents

Introduction to COBOL
COBOL Basics 1
COBOL Basics 2
Introduction to Sequential Files
Processing Sequential Files
Simple iteration with the PERFORM verb
Arithmetic and Edited Pictures
Conditions
Tables and the PERFORM ... VARYING
Designing Programs

Processing Sequential Files

Overview

- File organization and access methods.
- Ordered and unordered Sequential Files.
- Processing unordered files.
- Processing ordered files.

Processing Sequential Files

Run of SeqWrite

```
Enter student details using template below.
NNNNNNNSSSSSSSSIIYYMMDDCCCCGGGGS
9456789COUGHLANMS580812LM510598M
NNNNNNNSSSSSSSSIIYYMMDDCCCCGGGGS
9367892RYAN    TG521210LM601222F
NNNNNNNSSSSSSSSIIYYMMDDCCCCGGGGS
9368934WILSON  HR520323LM610786M
NNNNNNNSSSSSSSSIIYYMMDDCCCCGGGGS
CarriageReturn
```

```
      $ SET SOURCEFORMAT"FREE"
IDENTIFICATION DIVISION.
PROGRAM-ID.  SeqWrite.
AUTHOR.  Michael Coughlan.

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT StudentFile ASSIGN TO STUDENTS
        ORGANIZATION IS LINE SEQUENTIAL.

DATA DIVISION.
FILE SECTION.
FD StudentFile.
01 StudentDetails.
   02  StudentId        PIC 9(7).
   02  StudentName.
       03 Surname       PIC X(8).
       03 Initials      PIC XX.
   02  DateOfBirth.
       03 YOBirth       PIC 9(2).
       03 MOBirth       PIC 9(2).
       03 DOBirth       PIC 9(2).
   02  CourseCode       PIC X(4).
   02  Grant            PIC 9(4).
   02  Gender           PIC X.
```

```
PROCEDURE DIVISION.
Begin.
    OPEN OUTPUT StudentFile
    DISPLAY "Enter student details using template below.  Press CR to end.".
    PERFORM GetStudentDetails
    PERFORM UNTIL StudentDetails = SPACES
        WRITE StudentDetails
        PERFORM GetStudentDetails
    END-PERFORM
    CLOSE StudentFile
    STOP RUN.

GetStudentDetails.
    DISPLAY "NNNNNNNSSSSSSSSIIYYMMDDCCCCGGGGS".
    ACCEPT  StudentDetails.
```

Processing Sequential Files

RUN OF SeqRead

```
      $ SET SOURCEFORMAT"FREE"
IDENTIFICATION DIVISION.
PROGRAM-ID.  SeqRead.
AUTHOR.  Michael Coughlan.

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT StudentFile ASSIGN TO STUDENTS
        ORGANIZATION IS LINE SEQUENTIAL.

DATA DIVISION.
FILE SECTION.
FD StudentFile.
01 StudentDetails.
   02  StudentId       PIC 9(7).
   02  StudentName.
       03 Surname      PIC X(8).
       03 Initials     PIC XX.
   02  DateOfBirth.
       03 YOBirth      PIC 9(2).
       03 MOBirth      PIC 9(2).
       03 DOBirth      PIC 9(2).
   02  CourseCode      PIC X(4).
   02  Grant           PIC 9(4).
   02  Gender          PIC X.
```

```
9456789 COUGHLANMS LM51

9367892 RYAN     TG LM60

9368934 WILSON   HR LM61
```

```
PROCEDURE DIVISION.
Begin.
   OPEN INPUT StudentFile
   READ StudentFile
      AT END MOVE HIGH-VALUES TO StudentDetails
   END-READ
   PERFORM UNTIL StudentDetails = HIGH-VALUES
      DISPLAY StudentId SPACE StudentName SPACE CourseCode
      READ StudentFile
         AT END MOVE HIGH-VALUES TO StudentDetails
      END-READ
   END-PERFORM
   CLOSE StudentFile
   STOP RUN.
```

Processing Sequential Files

Organization and Access

- Two important characteristics of files are
  - **DATA ORGANIZATION**
  - **METHOD OF ACCESS**

- Data organization refers to the way the records of the file are organized on the backing storage device.
  COBOL recognizes three main file organizations;

Sequential	- Records organized serially.
Relative	- Relative record number based organization.
Indexed	- Index based organization.

- The method of access refers to the way in which records are accessed.
  - A file with an organization of Indexed or Relative may still have its records accessed sequentially.
  - But records in a file with an organization of Sequential can not be accessed directly.

Processing Sequential Files

Sequential Organization

- The simplest COBOL file organization is Sequential.
- In a Sequential file the records are arranged serially, one after another, like cards in a dealing shoe.
- In a Sequential file the only way to access any particular record is to;

  Start at the first record and read all the succeeding records until you find the one you want or reach the end of the file.

- Sequential files may be

  Ordered

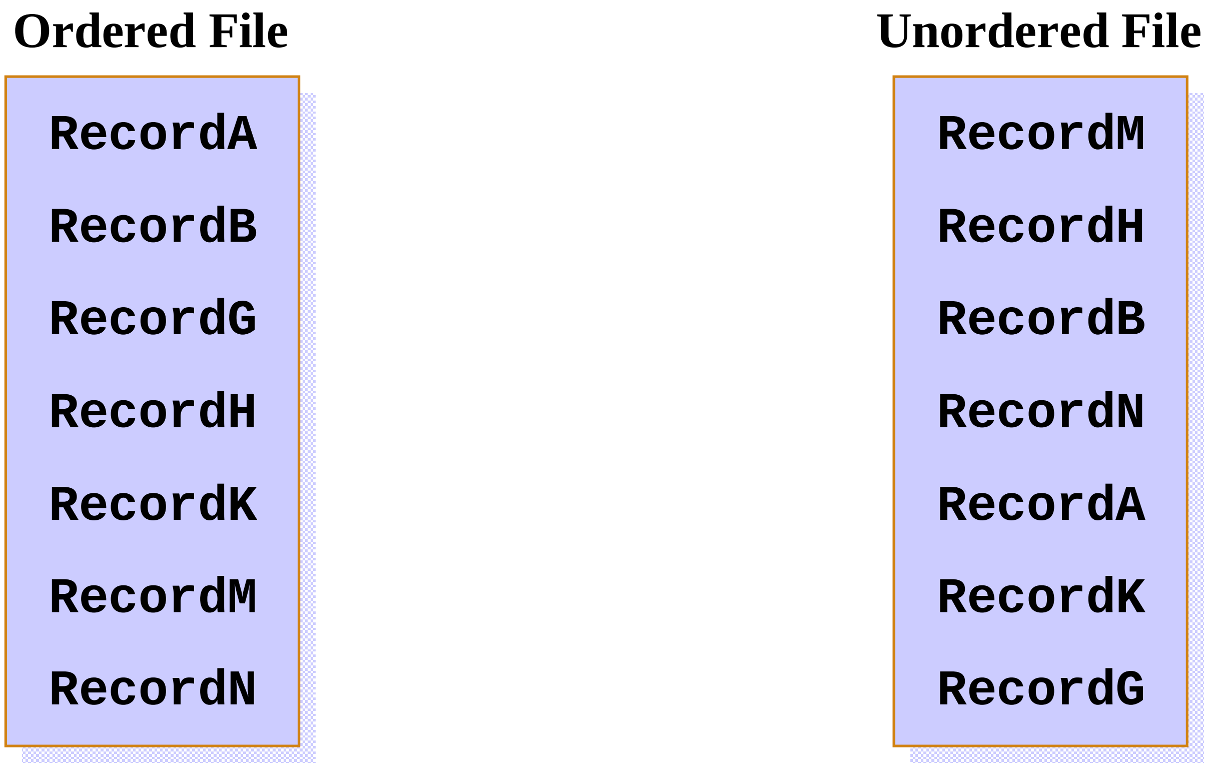
  or

  Unordered **(these should be called Serial files)**

- The ordering of the records in a file has a significant impact on the way in which it is processed and the processing that can be done on it.

Processing Sequential Files

Ordered and Unordered Files



Ordered File

- RecordA
- RecordB
- RecordG
- RecordH
- RecordK
- RecordM
- RecordN

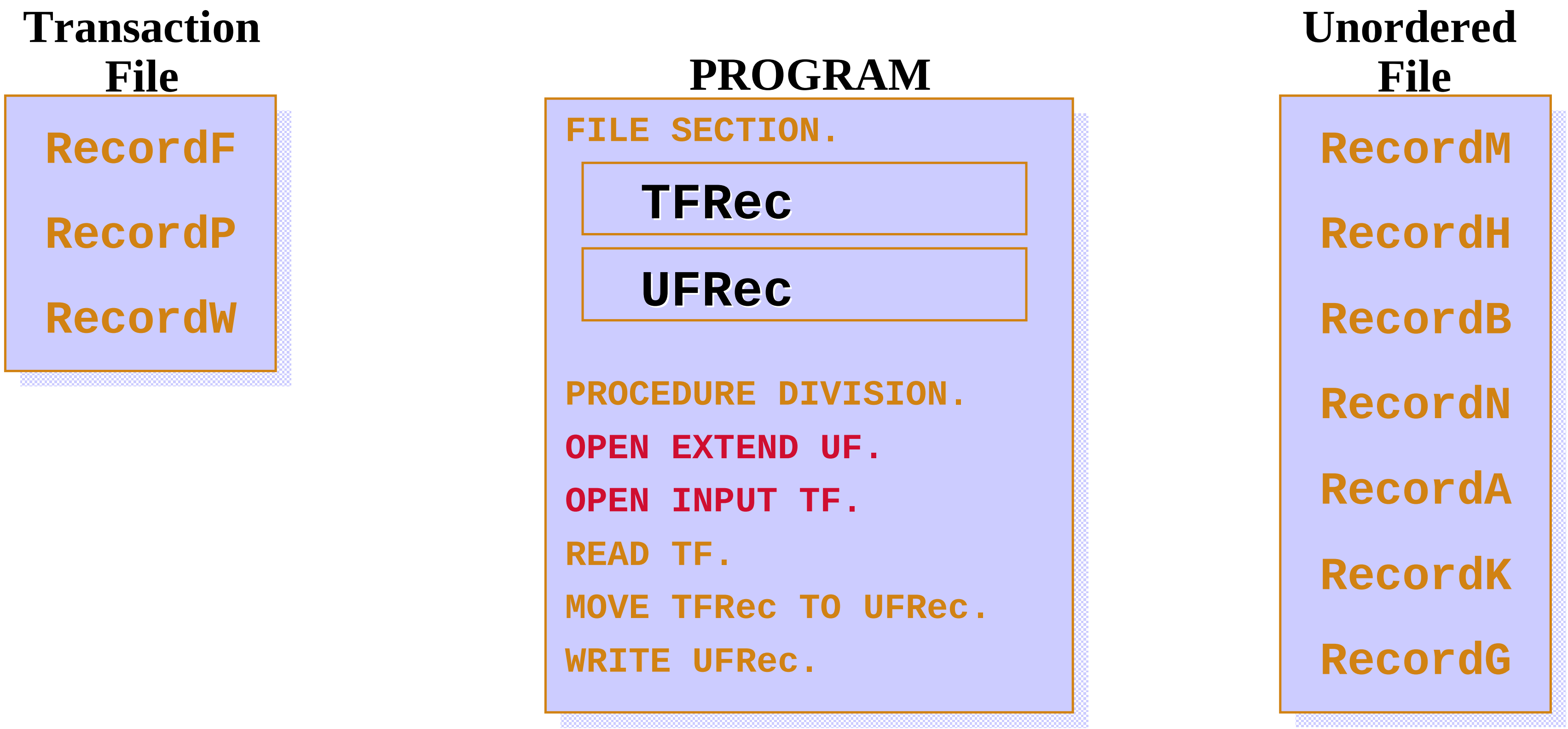
Unordered File

- RecordM
- RecordH
- RecordB
- RecordN
- RecordA
- RecordK
- RecordG

In an ordered file the records are sequenced on some field in the record.

Processing Sequential Files

Adding records to unordered files



Transaction File

RecordF

RecordP

RecordW

PROGRAM

```
FILE SECTION.
   TFRec
   UFRec

PROCEDURE DIVISION.
OPEN EXTEND UF.
OPEN INPUT TF.
READ TF.
MOVE TFRec TO UFRec.
WRITE UFRec.
```

Unordered File

RecordM

RecordH

RecordB

RecordN

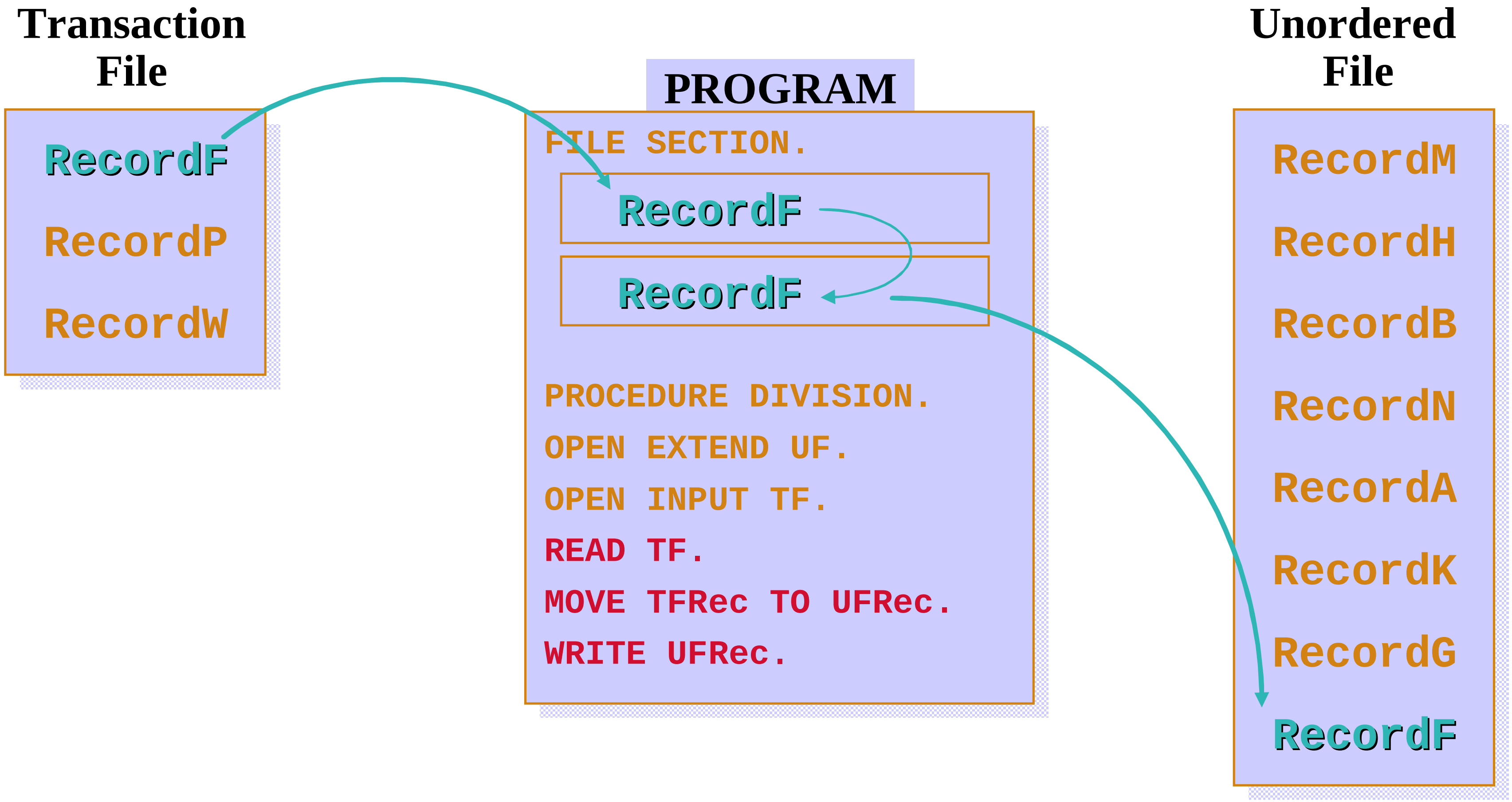
RecordA

RecordK

RecordG

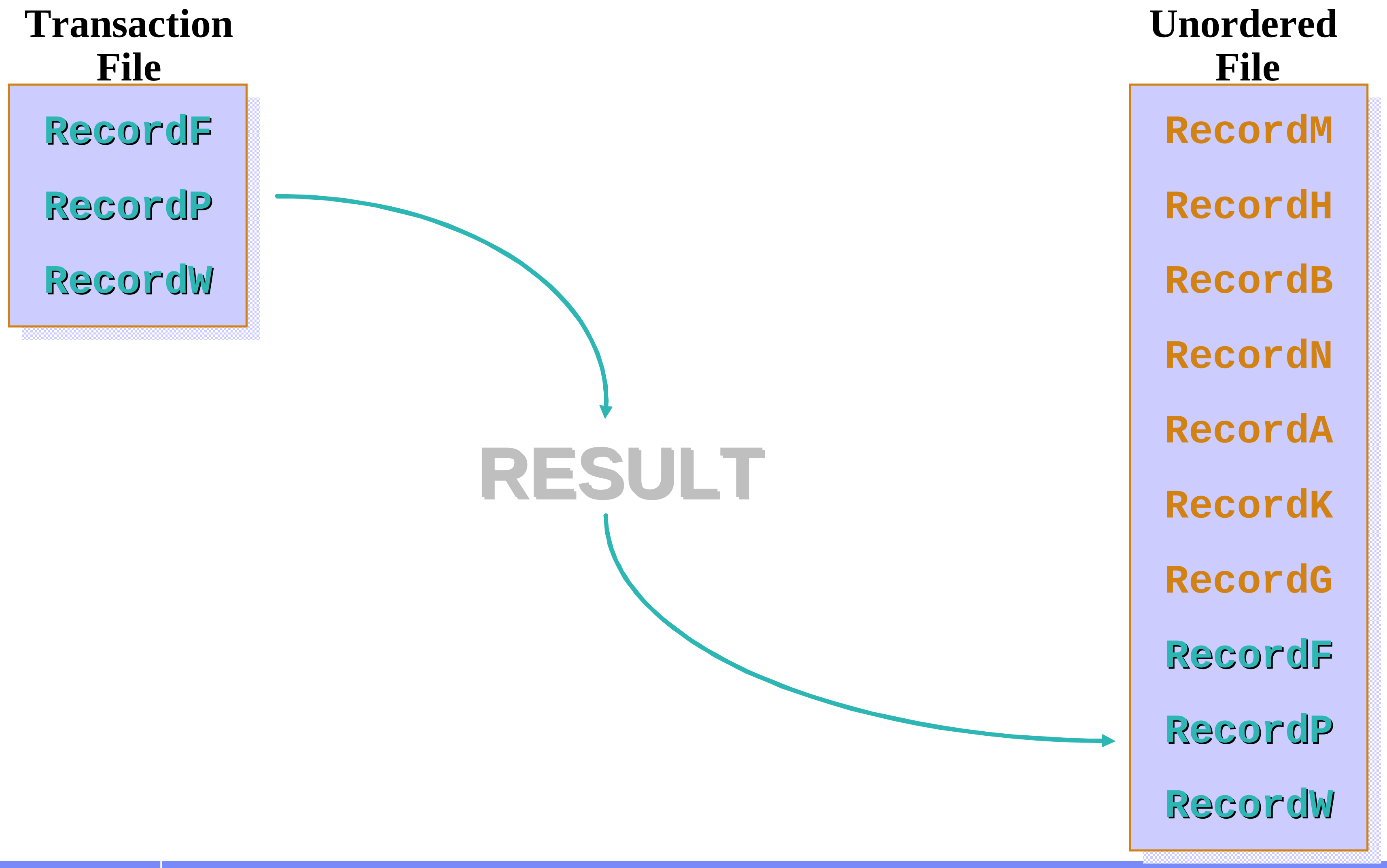
Processing Sequential Files

Adding records to unordered files



Processing Sequential Files

Adding records to unordered files



Transaction File

RecordF

RecordP

RecordW

RESULT

Unordered File

RecordM

RecordH

RecordB

RecordN

RecordA

RecordK

RecordG

RecordF

RecordP

RecordW

Processing Sequential Files

Problems with Unordered Sequential Files

- It is easy to add records to an unordered Sequential file.
- But it is not really possible to delete records from an unordered Sequential file.
- And it is not feasible to update records in an unordered Sequential file

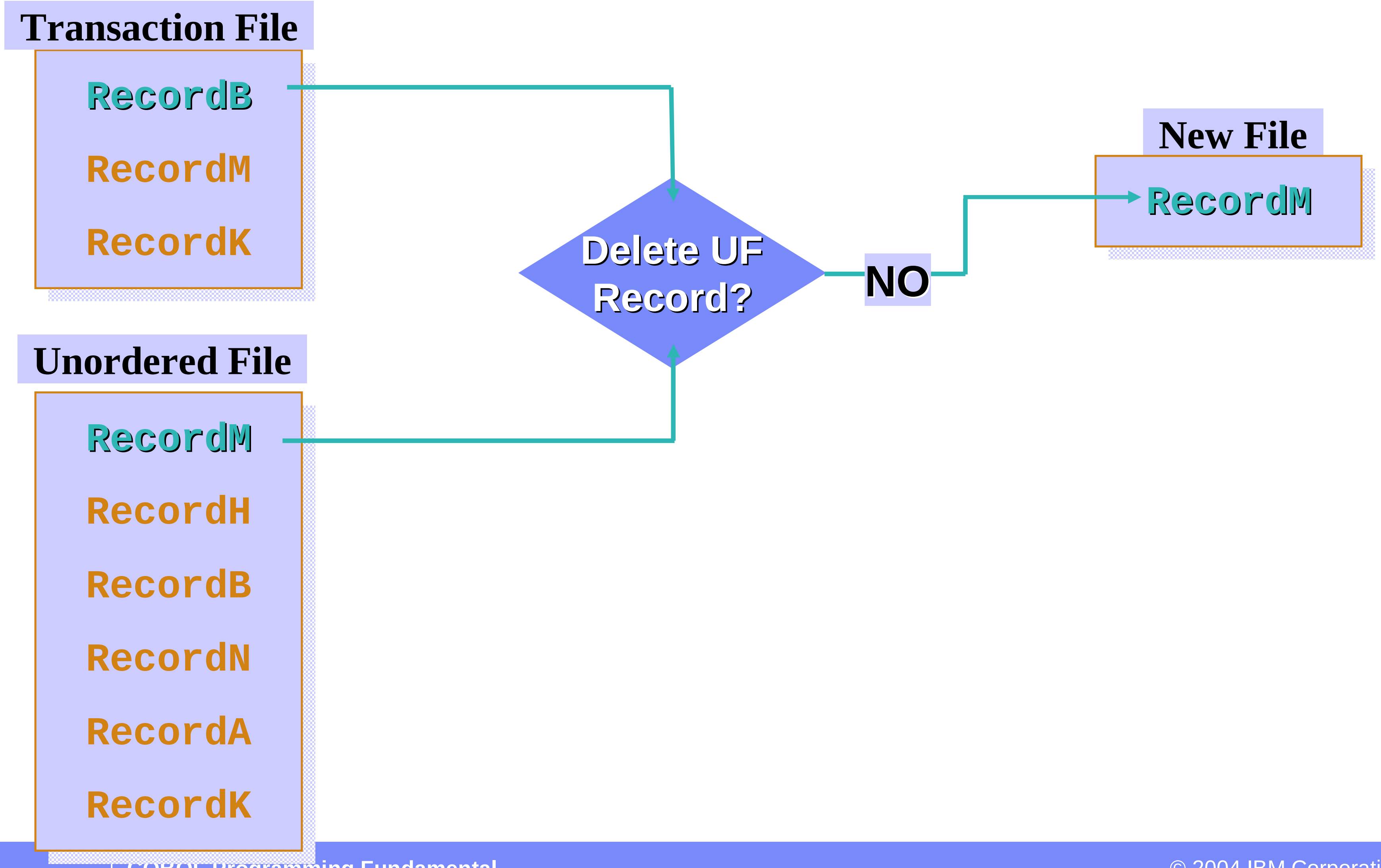
Processing Sequential Files

Problems with Unordered Sequential Files

- Records in a Sequential file can not be deleted or updated “in situ”.
- The only way to delete Sequential file records is to create a new file which does not contain them.
- The only way to update records in a Sequential File is to create a new file which contains the updated records.
- Because both these operations rely on record matching they do not work for unordered Sequential files.
- Why?

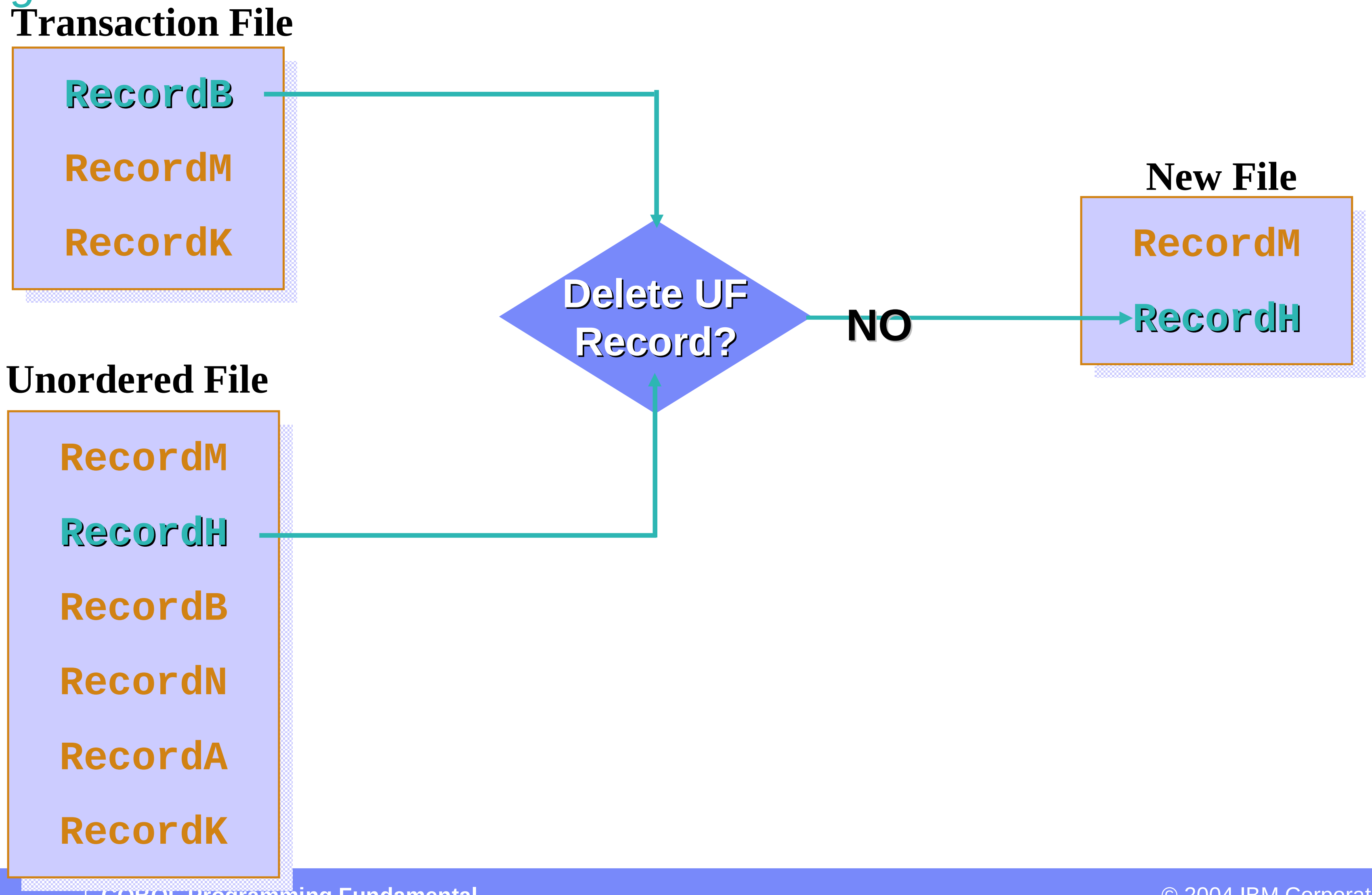
Processing Sequential Files

Deleting records from unordered files?



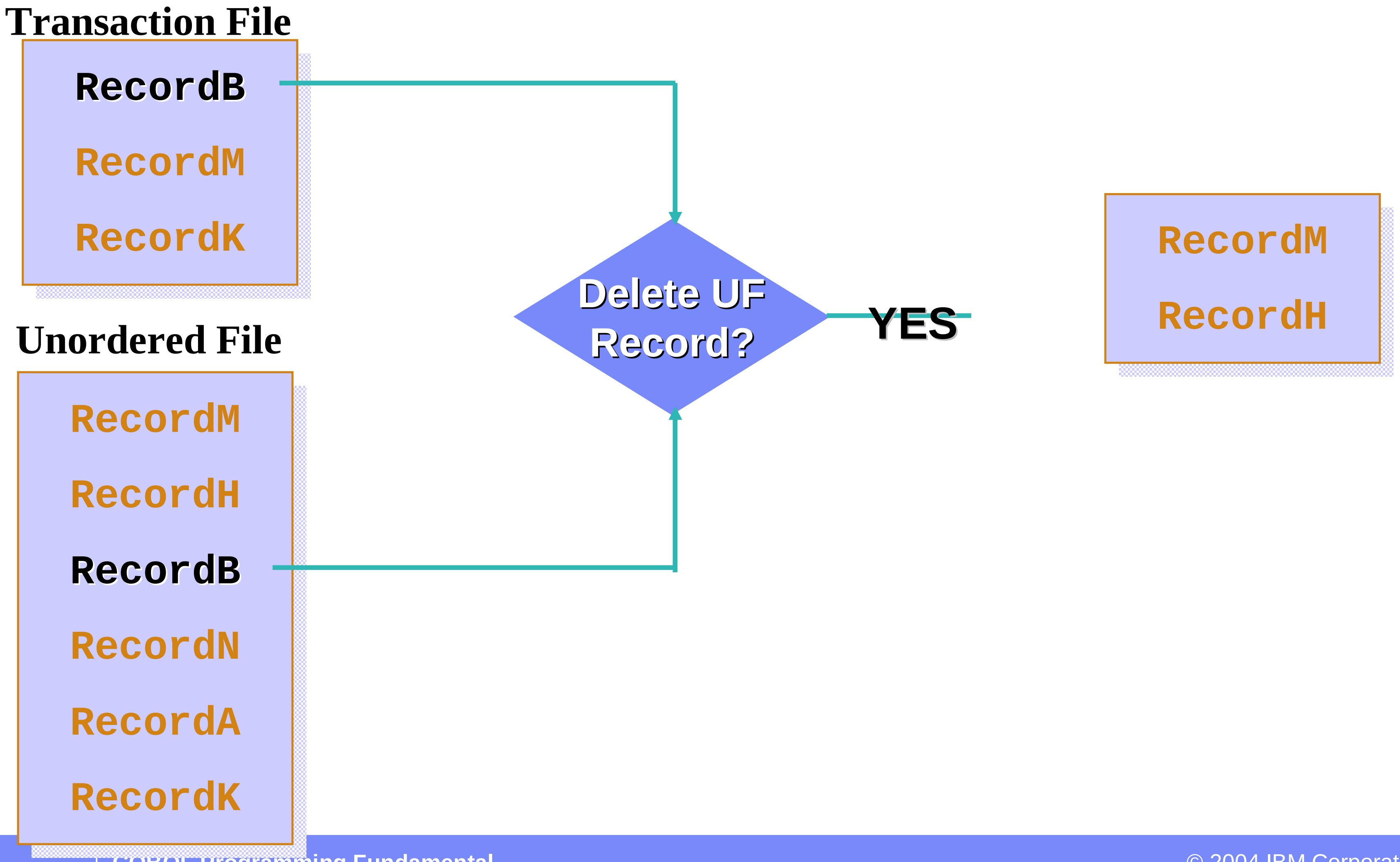
Processing Sequential Files

Deleting records from unordered files?



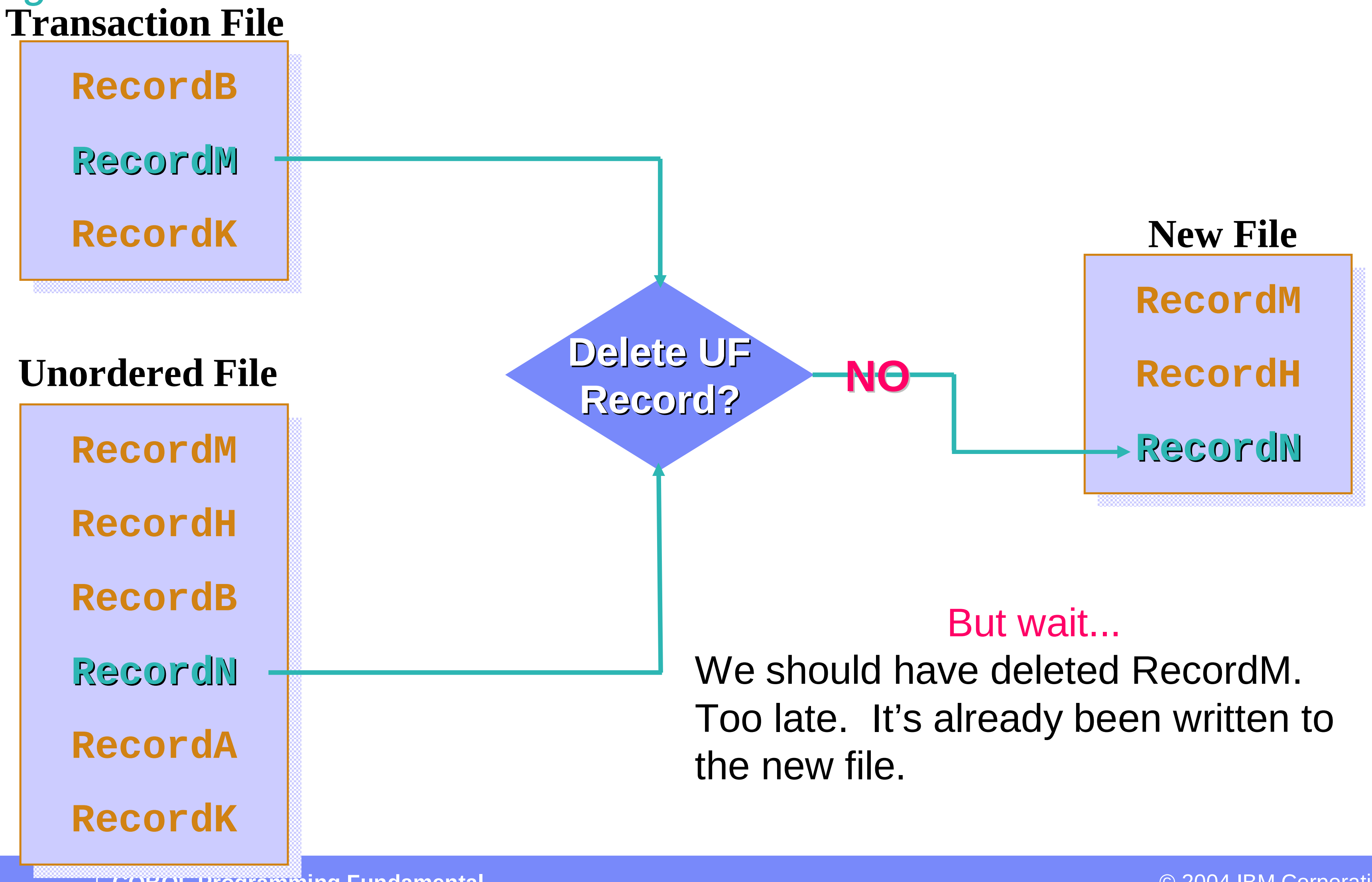
Processing Sequential Files

Deleting records from unordered files?



Processing Sequential Files

Deleting records from unordered files?

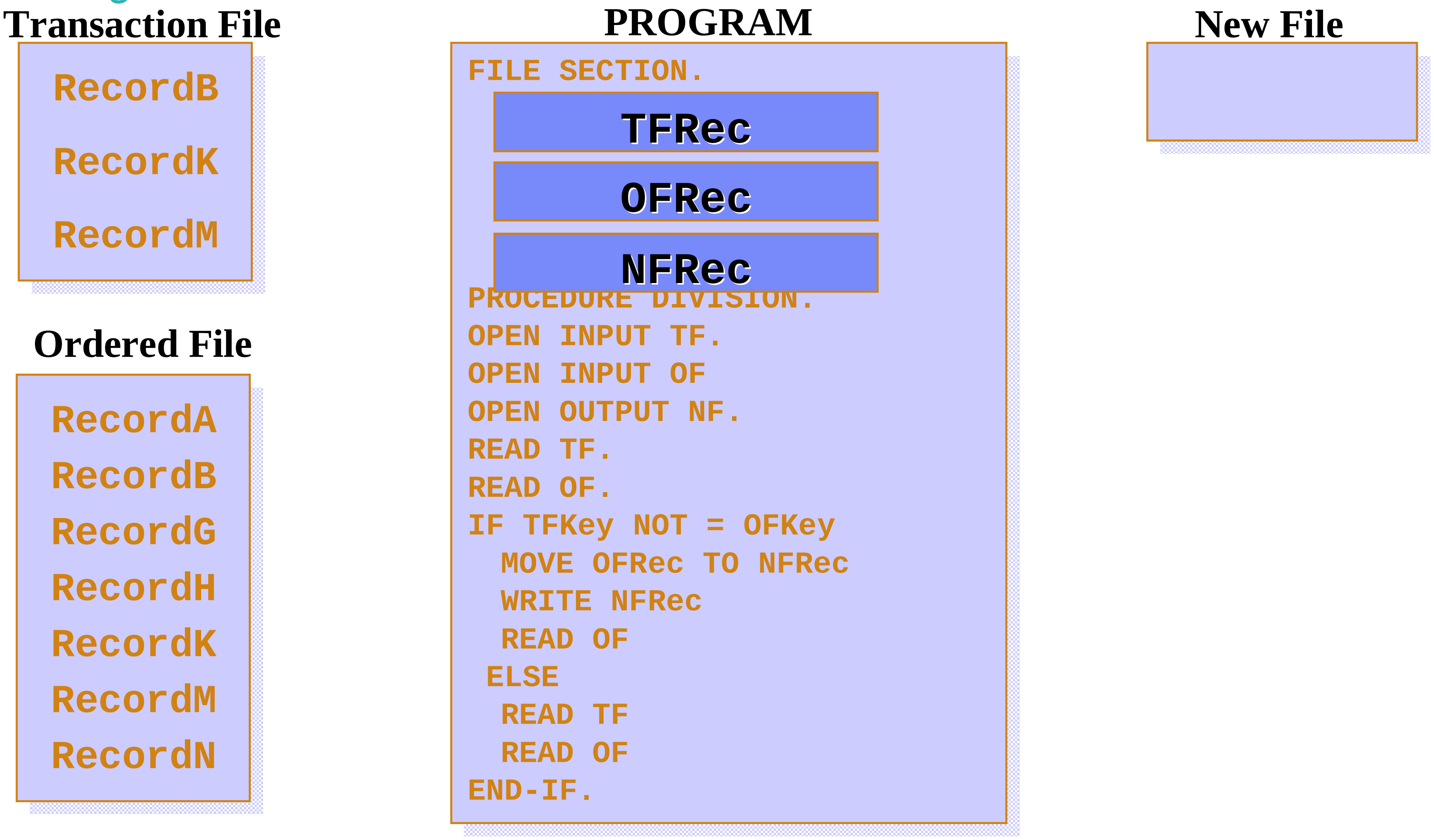


But wait...

We should have deleted RecordM. Too late. It's already been written to the new file.

Processing Sequential Files

Deleting records from an ordered file



Transaction File

RecordB

RecordK

RecordM

Ordered File

RecordA

RecordB

RecordG

RecordH

RecordK

RecordM

RecordN

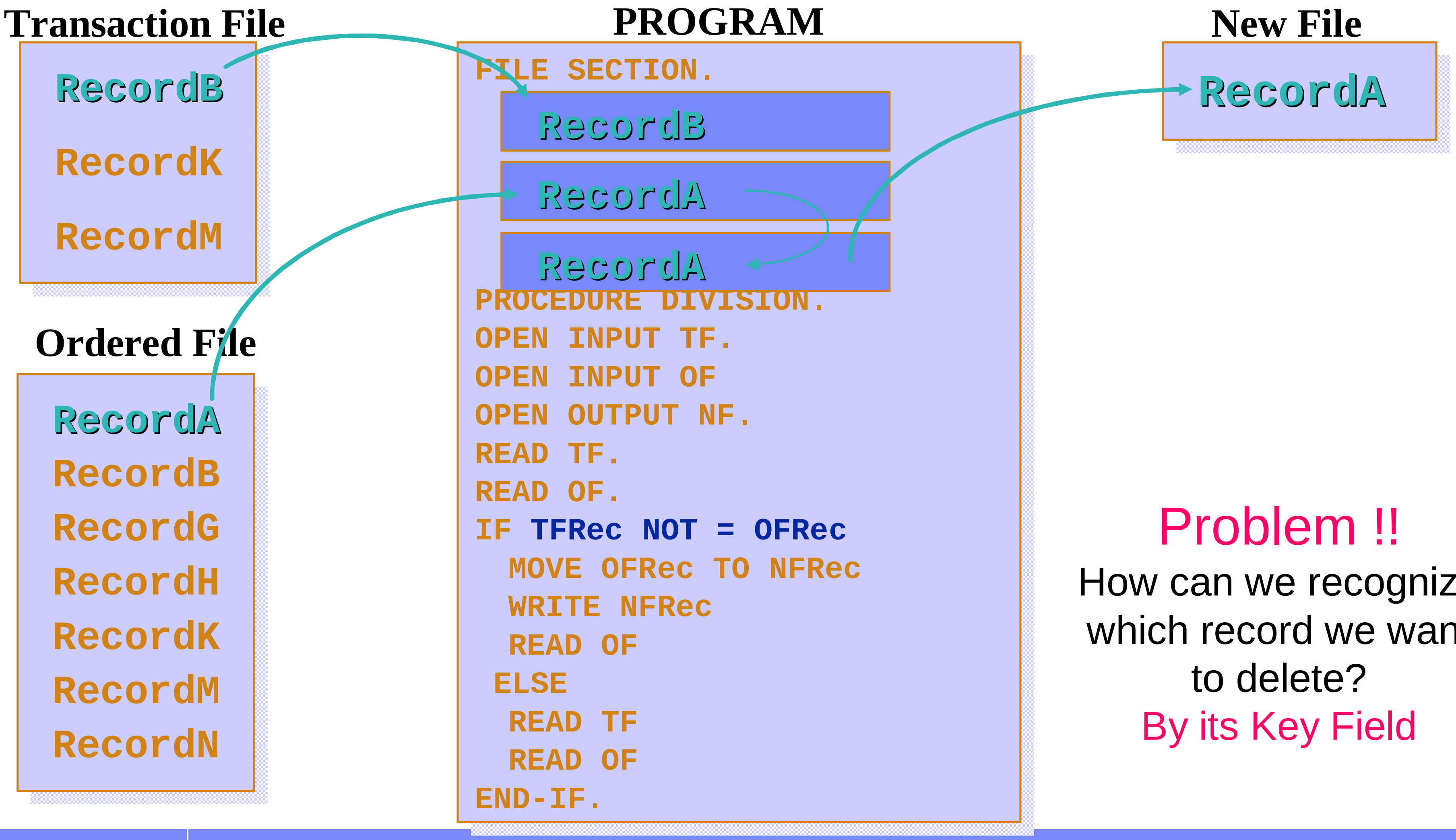
PROGRAM

```
FILE SECTION.
    TFRec
    OFRec
    NFRec
PROCEDURE DIVISION.
OPEN INPUT TF.
OPEN INPUT OF
OPEN OUTPUT NF.
READ TF.
READ OF.
IF TFKey NOT = OFKey
   MOVE OFRec TO NFRec
   WRITE NFRec
   READ OF
 ELSE
   READ TF
   READ OF
END-IF.
```

New File

Processing Sequential Files

Deleting records from an ordered file



Transaction File

Ordered File

PROGRAM

```
FILE SECTION.
    RecordB
    RecordA
    RecordA
PROCEDURE DIVISION.
OPEN INPUT TF.
OPEN INPUT OF
OPEN OUTPUT NF.
READ TF.
READ OF.
IF TFRec NOT = OFRec
  MOVE OFRec TO NFRec
  WRITE NFRec
  READ OF
 ELSE
  READ TF
  READ OF
END-IF.
```

New File

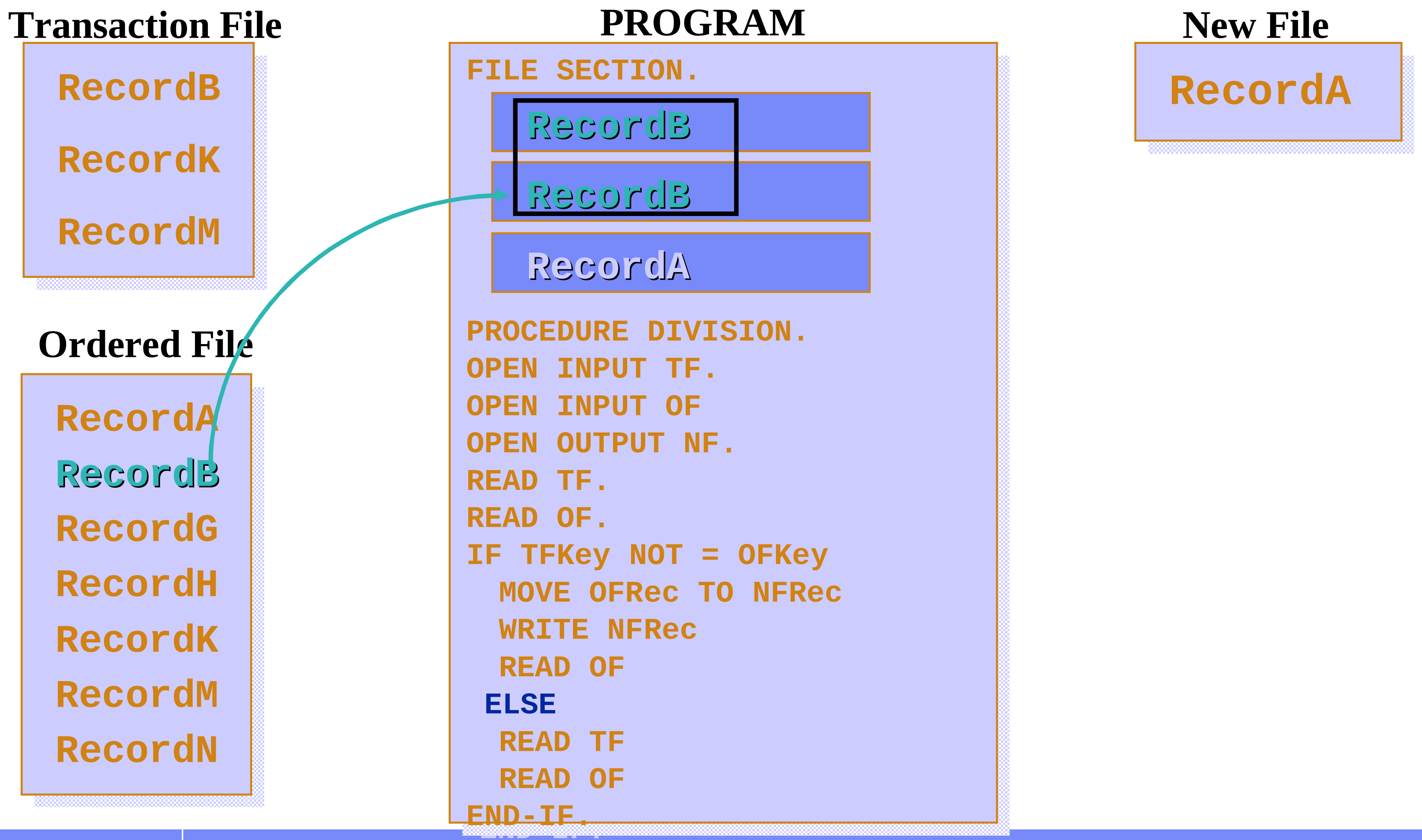
Problem !!

How can we recognize which record we want to delete?

By its Key Field

Processing Sequential Files

Deleting records from an ordered file



Transaction File

RecordB

RecordK

RecordM

Ordered File

RecordA

RecordB

RecordG

RecordH

RecordK

RecordM

RecordN

PROGRAM

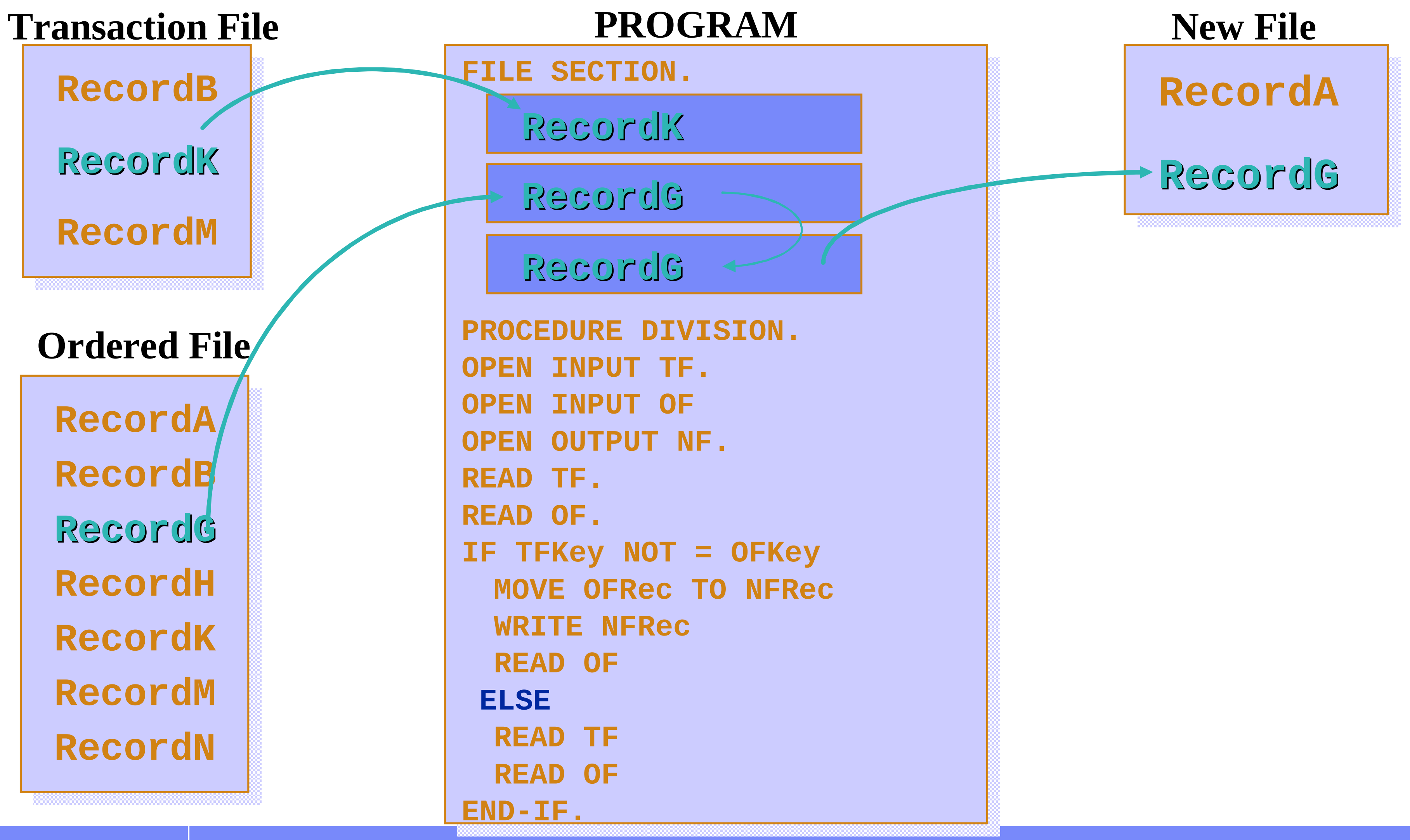
```
FILE SECTION.
   RecordB
   RecordB
   RecordA

PROCEDURE DIVISION.
OPEN INPUT TF.
OPEN INPUT OF
OPEN OUTPUT NF.
READ TF.
READ OF.
IF TFKey NOT = OFKey
  MOVE OFRec TO NFRec
  WRITE NFRec
  READ OF
 ELSE
  READ TF
  READ OF
END-IF.
```

New File

RecordA

Processing Sequential Files

Deleting records from an ordered file



Transaction File

RecordB
RecordK
RecordM

Ordered File

RecordA
RecordB
RecordG
RecordH
RecordK
RecordM
RecordN

PROGRAM

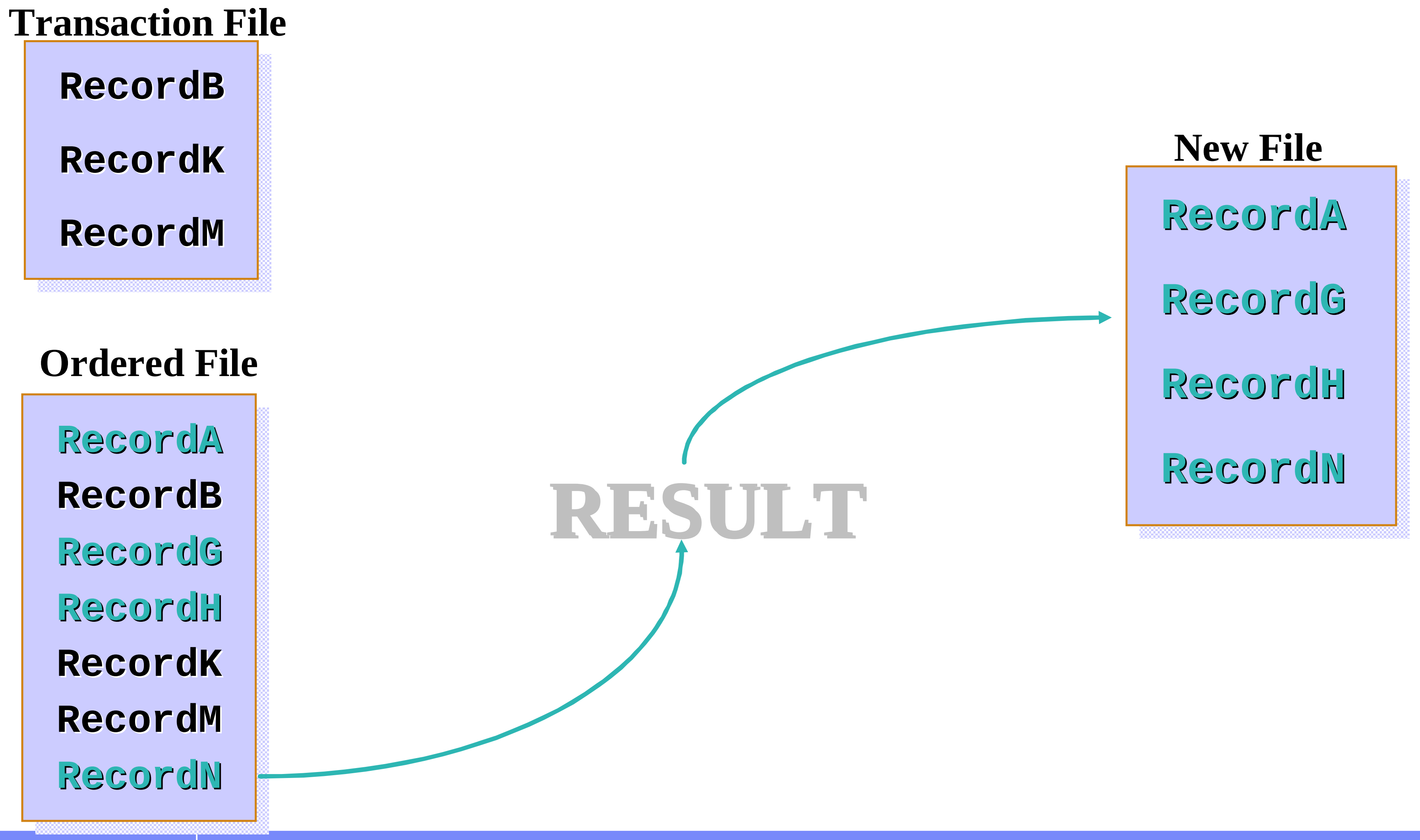
```
FILE SECTION.
    RecordK
    RecordG
    RecordG

PROCEDURE DIVISION.
OPEN INPUT TF.
OPEN INPUT OF
OPEN OUTPUT NF.
READ TF.
READ OF.
IF TFKey NOT = OFKey
  MOVE OFRec TO NFRec
  WRITE NFRec
  READ OF
 ELSE
  READ TF
  READ OF
END-IF.
```

New File

RecordA
RecordG

Processing Sequential Files

Deleting records from an ordered file



Transaction File

RecordB

RecordK

RecordM

Ordered File

RecordA

RecordB

RecordG

RecordH

RecordK

RecordM

RecordN

RESULT

New File

RecordA

RecordG

RecordH

RecordN

Processing Sequential Files

Updating records in an ordered file

Transaction File

RecordB

RecordH

RecordK

Ordered File

RecordA

RecordB

RecordG

RecordH

RecordK

RecordM

RecordN

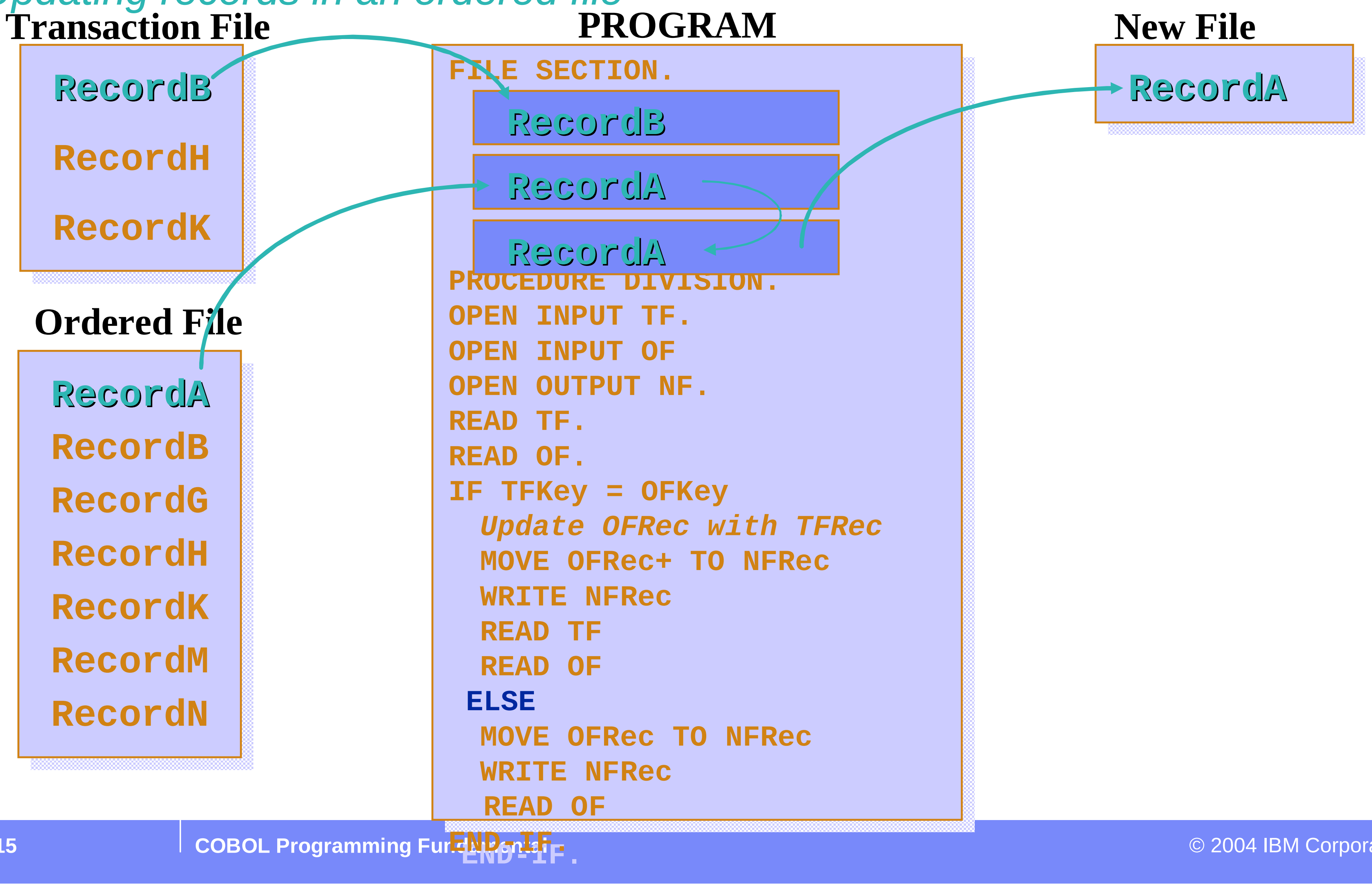
PROGRAM

```
FILE SECTION.
      TFRec
      OFRec
      NFRec
PROCEDURE DIVISION.
OPEN INPUT TF.
OPEN INPUT OF
OPEN OUTPUT NF.
READ TF.
READ OF.
IF TFKey = OFKey
  Update OFRec with TFRec
  MOVE OFRec+ TO NFRec
  WRITE NFRec
  READ TF
  READ OF
 ELSE
  MOVE OFRec TO NFRec
  WRITE NFRec
  READ OF
END-IF
 END-IF.
```

New File

Processing Sequential Files

Updating records in an ordered file



Transaction File

- RecordB
- RecordH
- RecordK

Ordered File

- RecordA
- RecordB
- RecordG
- RecordH
- RecordK
- RecordM
- RecordN

PROGRAM

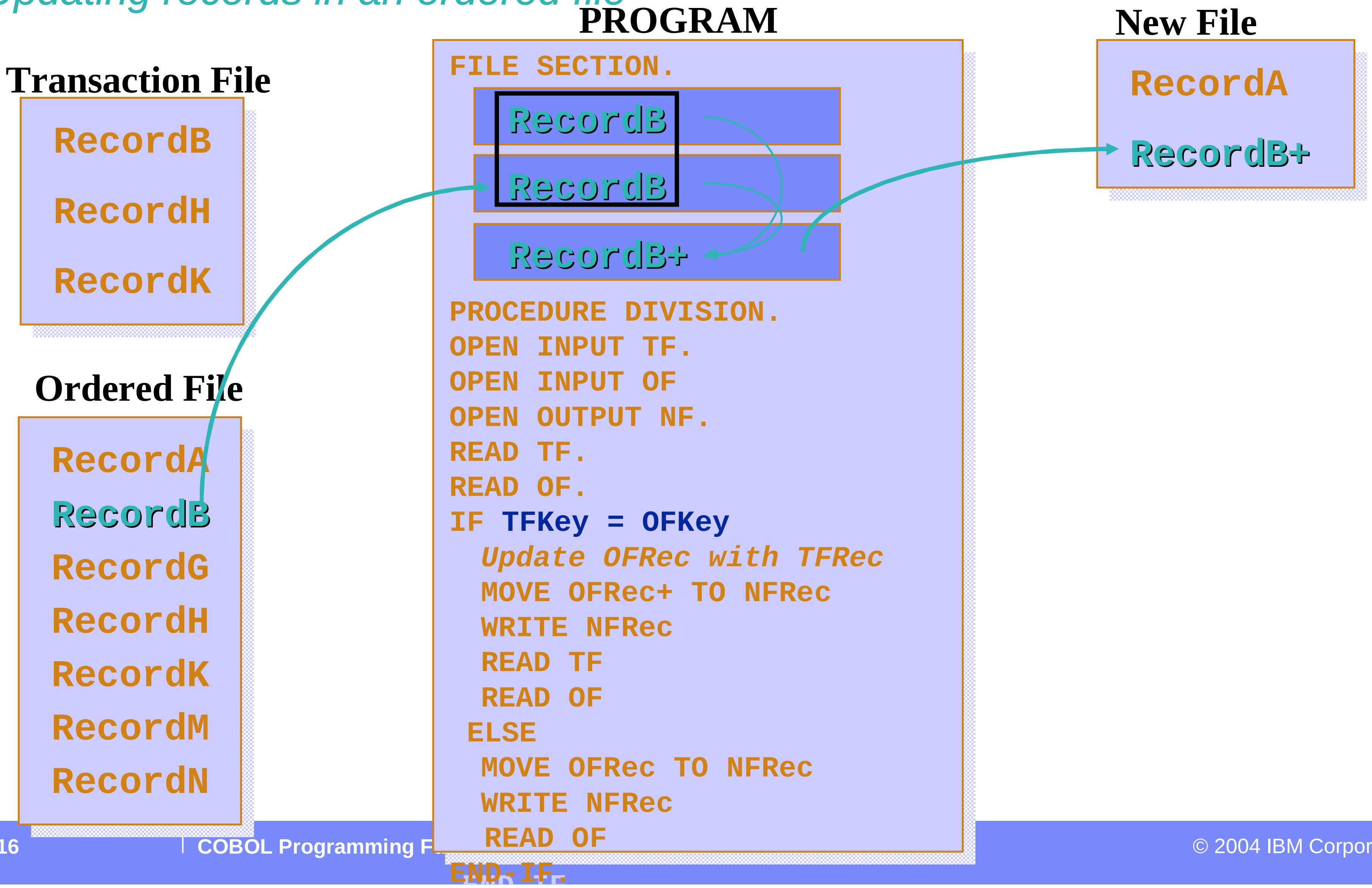
```
FILE SECTION.
    RecordB
    RecordA
    RecordA
PROCEDURE DIVISION.
OPEN INPUT TF.
OPEN INPUT OF
OPEN OUTPUT NF.
READ TF.
READ OF.
IF TFKey = OFKey
  Update OFRec with TFRec
  MOVE OFRec+ TO NFRec
  WRITE NFRec
  READ TF
  READ OF
 ELSE
  MOVE OFRec TO NFRec
  WRITE NFRec
  READ OF
END-IF
 END-IF.
```

New File

- RecordA

Processing Sequential Files

Updating records in an ordered file



Transaction File

RecordB
RecordH
RecordK

Ordered File

RecordA
RecordB
RecordG
RecordH
RecordK
RecordM
RecordN

PROGRAM

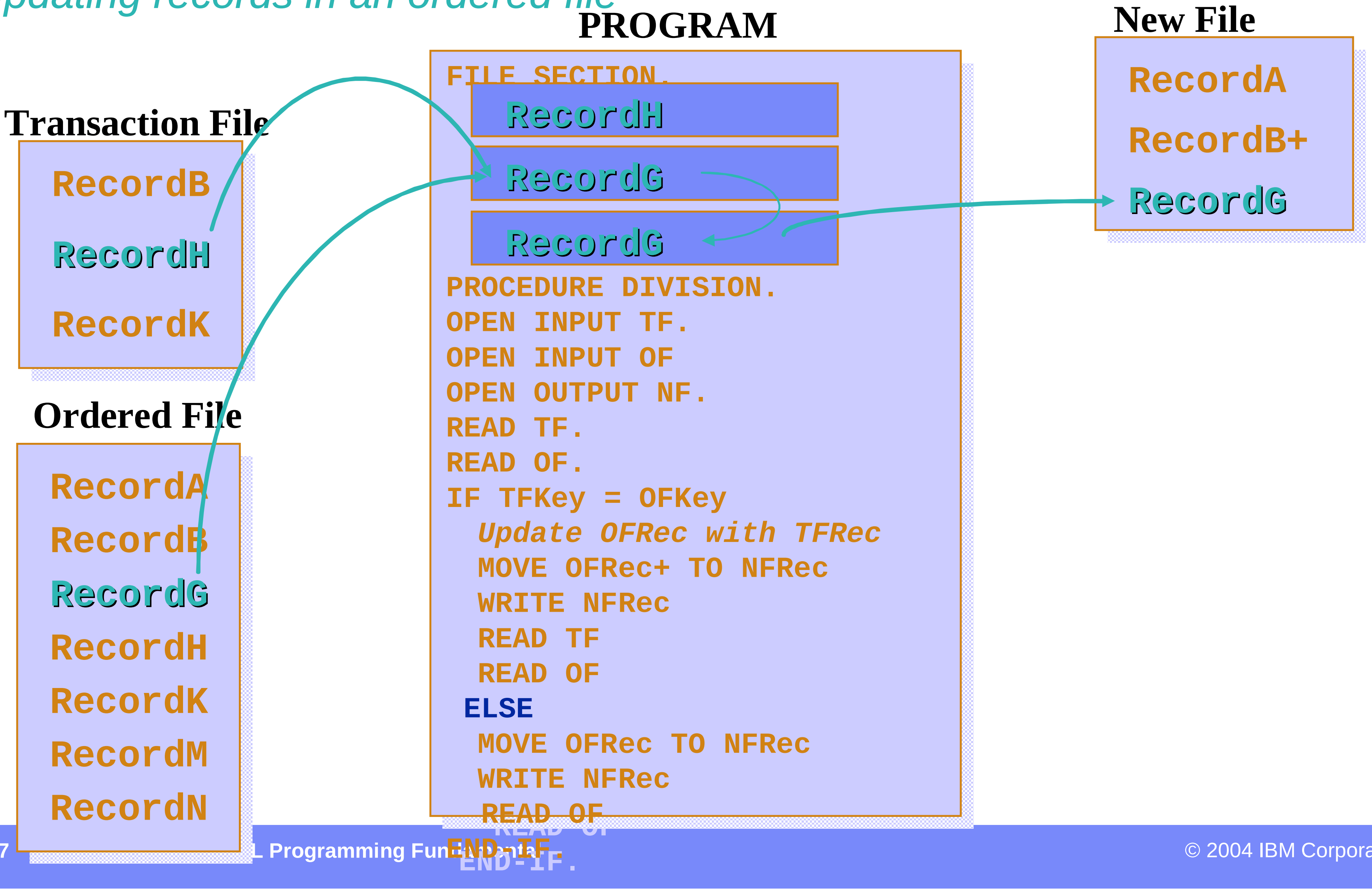
```
FILE SECTION.
    RecordB
    RecordB
    RecordB+
PROCEDURE DIVISION.
OPEN INPUT TF.
OPEN INPUT OF
OPEN OUTPUT NF.
READ TF.
READ OF.
IF TFKey = OFKey
   Update OFRec with TFRec
   MOVE OFRec+ TO NFRec
   WRITE NFRec
   READ TF
   READ OF
 ELSE
   MOVE OFRec TO NFRec
   WRITE NFRec
   READ OF
END-IF
```

New File

RecordA
RecordB+

Processing Sequential Files

Updating records in an ordered file



PROGRAM

```
FILE SECTION.
    RecordH
    RecordG
    RecordG
PROCEDURE DIVISION.
OPEN INPUT TF.
OPEN INPUT OF
OPEN OUTPUT NF.
READ TF.
READ OF.
IF TFKey = OFKey
   Update OFRec with TFRec
   MOVE OFRec+ TO NFRec
   WRITE NFRec
   READ TF
   READ OF
 ELSE
   MOVE OFRec TO NFRec
   WRITE NFRec
   READ OF
END-IF.
```

Transaction File

RecordB
RecordH
RecordK

Ordered File

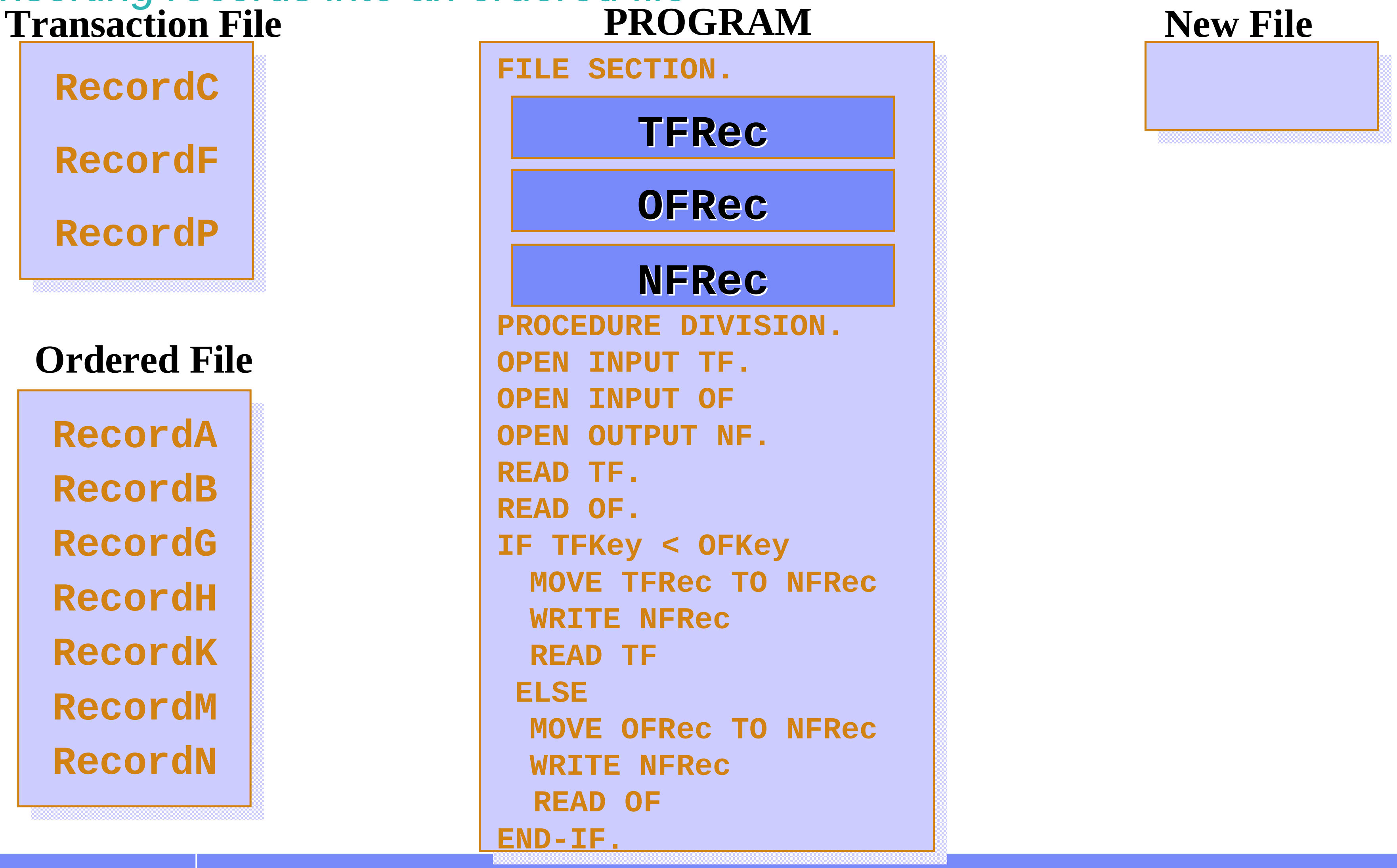
RecordA
RecordB
RecordG
RecordH
RecordK
RecordM
RecordN

New File

RecordA
RecordB+
RecordG

Processing Sequential Files

Inserting records into an ordered file



Transaction File

RecordC

RecordF

RecordP

Ordered File

RecordA

RecordB

RecordG

RecordH

RecordK

RecordM

RecordN

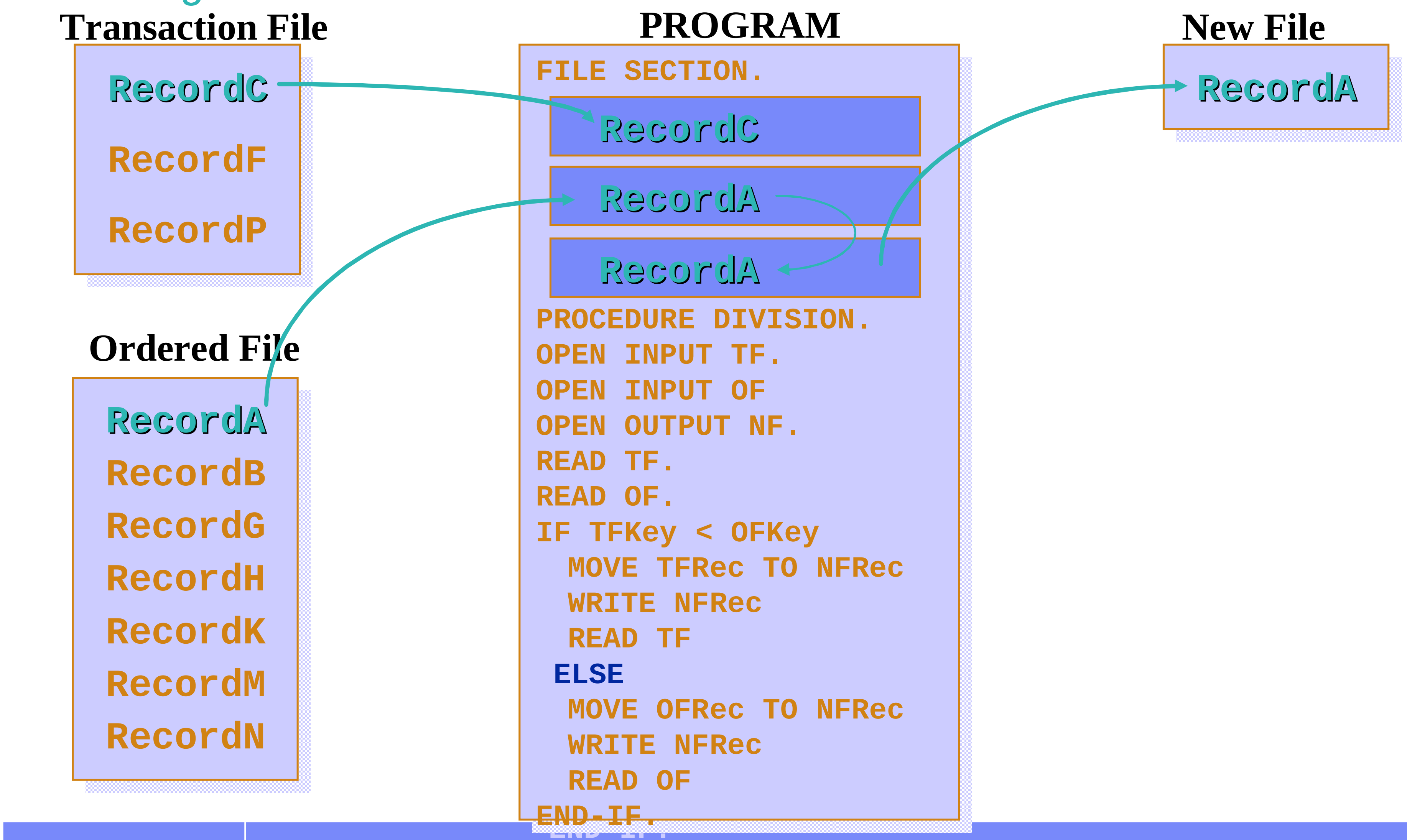
PROGRAM

```
FILE SECTION.
TFRec
OFRec
NFRec
PROCEDURE DIVISION.
OPEN INPUT TF.
OPEN INPUT OF
OPEN OUTPUT NF.
READ TF.
READ OF.
IF TFKey < OFKey
  MOVE TFRec TO NFRec
  WRITE NFRec
  READ TF
 ELSE
  MOVE OFRec TO NFRec
  WRITE NFRec
  READ OF
END-IF.
```

New File

Processing Sequential Files

Inserting records into an ordered file



Transaction File

- RecordC
- RecordF
- RecordP

Ordered File

- RecordA
- RecordB
- RecordG
- RecordH
- RecordK
- RecordM
- RecordN

PROGRAM

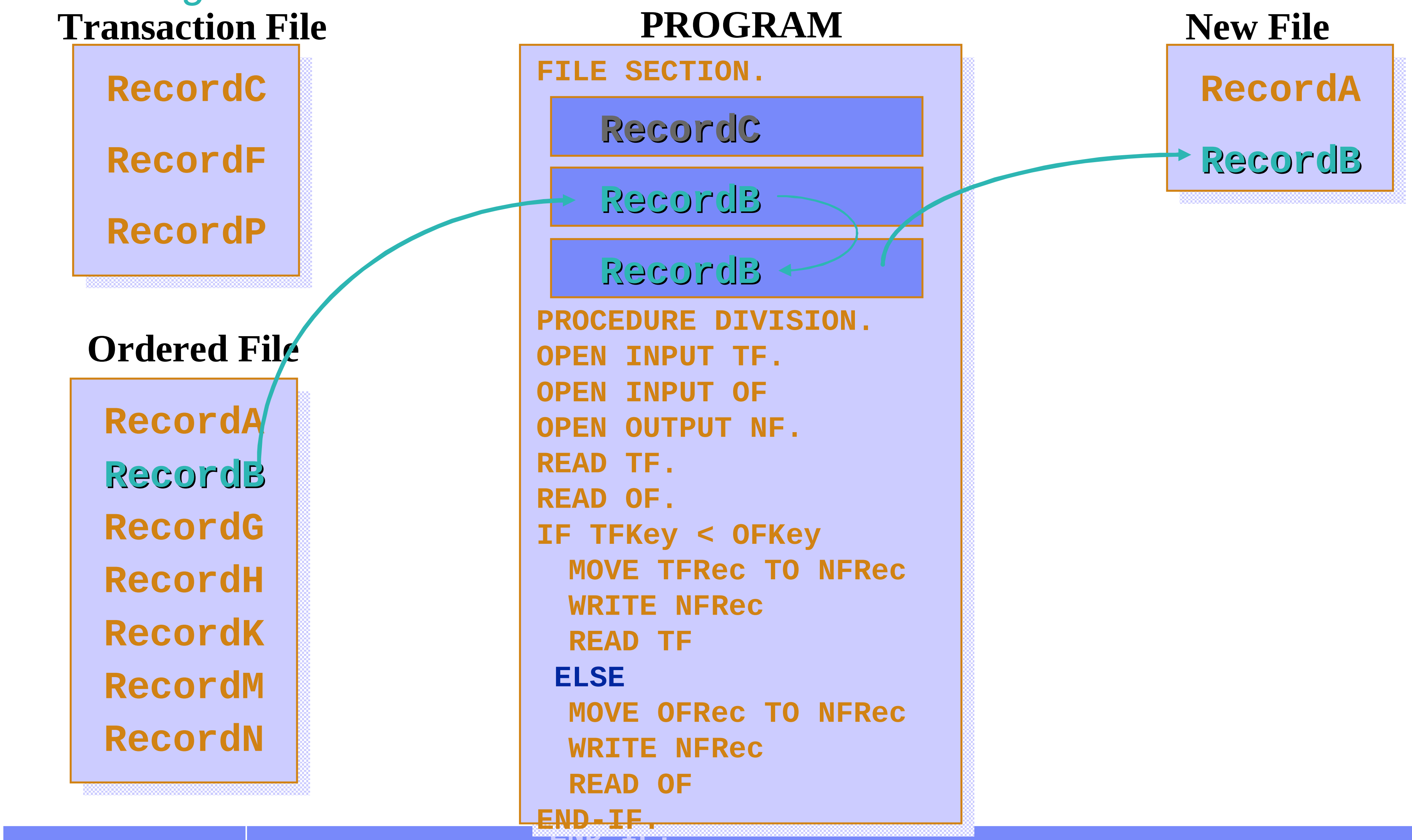
```
FILE SECTION.
    RecordC
    RecordA
    RecordA
PROCEDURE DIVISION.
OPEN INPUT TF.
OPEN INPUT OF
OPEN OUTPUT NF.
READ TF.
READ OF.
IF TFKey < OFKey
  MOVE TFRec TO NFRec
  WRITE NFRec
  READ TF
 ELSE
  MOVE OFRec TO NFRec
  WRITE NFRec
  READ OF
END-IF.
```

New File

- RecordA

Processing Sequential Files

Inserting records into an ordered file



Transaction File

RecordC
RecordF
RecordP

Ordered File

RecordA
RecordB
RecordG
RecordH
RecordK
RecordM
RecordN

PROGRAM

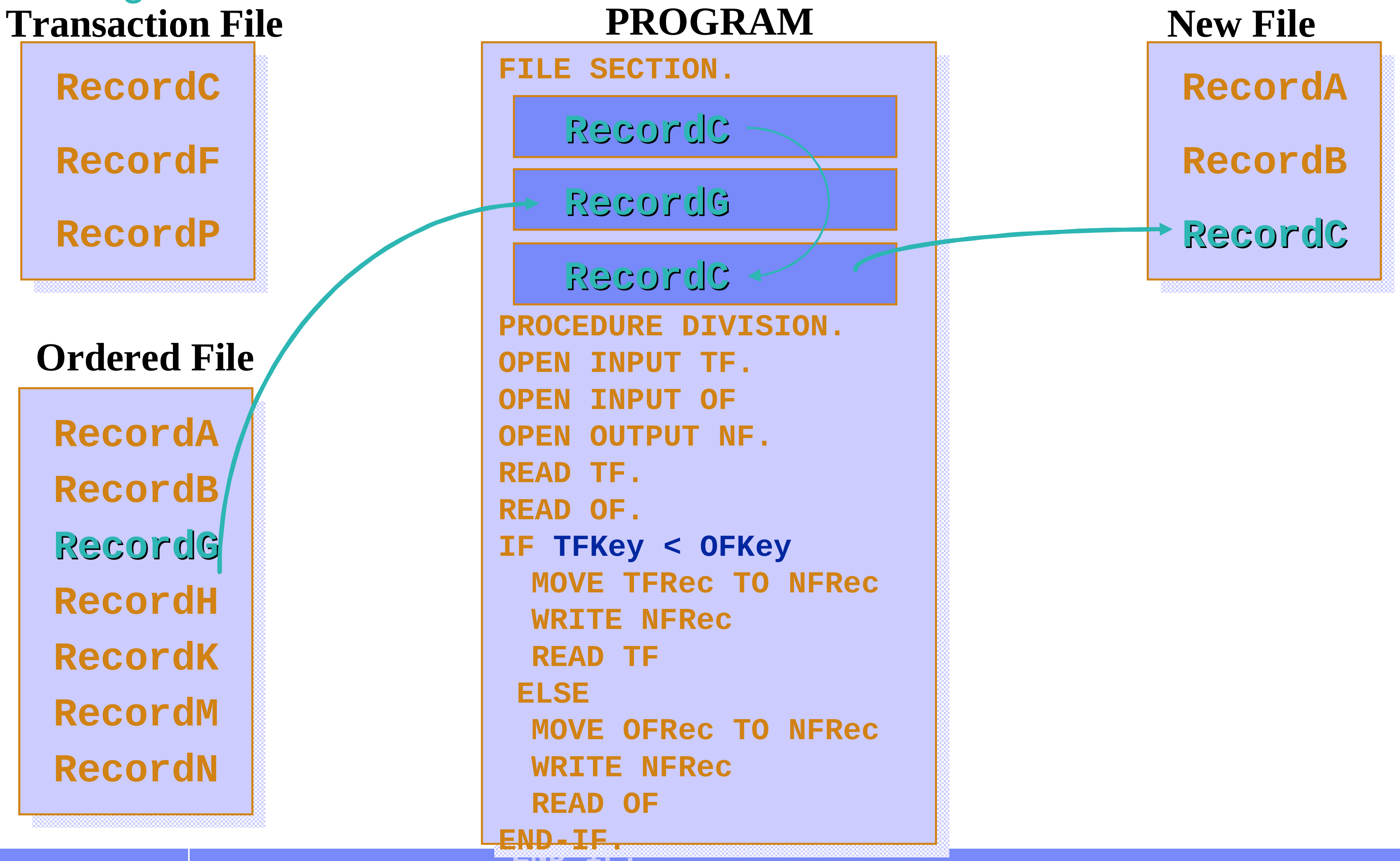
```
FILE SECTION.
    RecordC
    RecordB
    RecordB
PROCEDURE DIVISION.
OPEN INPUT TF.
OPEN INPUT OF
OPEN OUTPUT NF.
READ TF.
READ OF.
IF TFKey < OFKey
  MOVE TFRec TO NFRec
  WRITE NFRec
  READ TF
 ELSE
  MOVE OFRec TO NFRec
  WRITE NFRec
  READ OF
END-IF.
```

New File

RecordA
RecordB

Processing Sequential Files

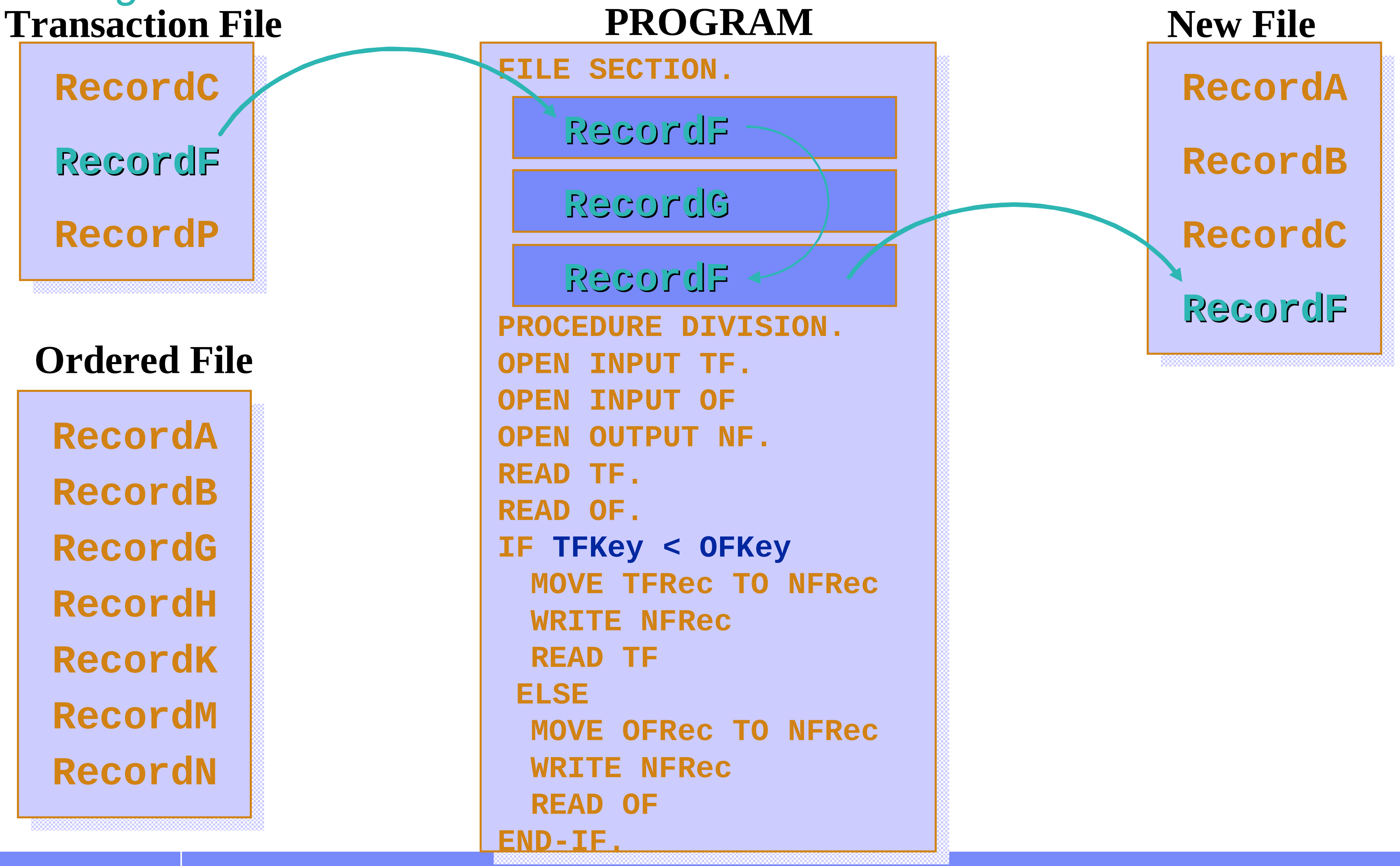
Inserting records into an ordered file



```
FILE SECTION.
PROCEDURE DIVISION.
OPEN INPUT TF.
OPEN INPUT OF
OPEN OUTPUT NF.
READ TF.
READ OF.
IF TFKey < OFKey
  MOVE TFRec TO NFRec
  WRITE NFRec
  READ TF
 ELSE
  MOVE OFRec TO NFRec
  WRITE NFRec
  READ OF
END-IF.
```

Processing Sequential Files

Inserting records into an ordered file



Transaction File

RecordC
RecordF
RecordP

Ordered File

RecordA
RecordB
RecordG
RecordH
RecordK
RecordM
RecordN

PROGRAM

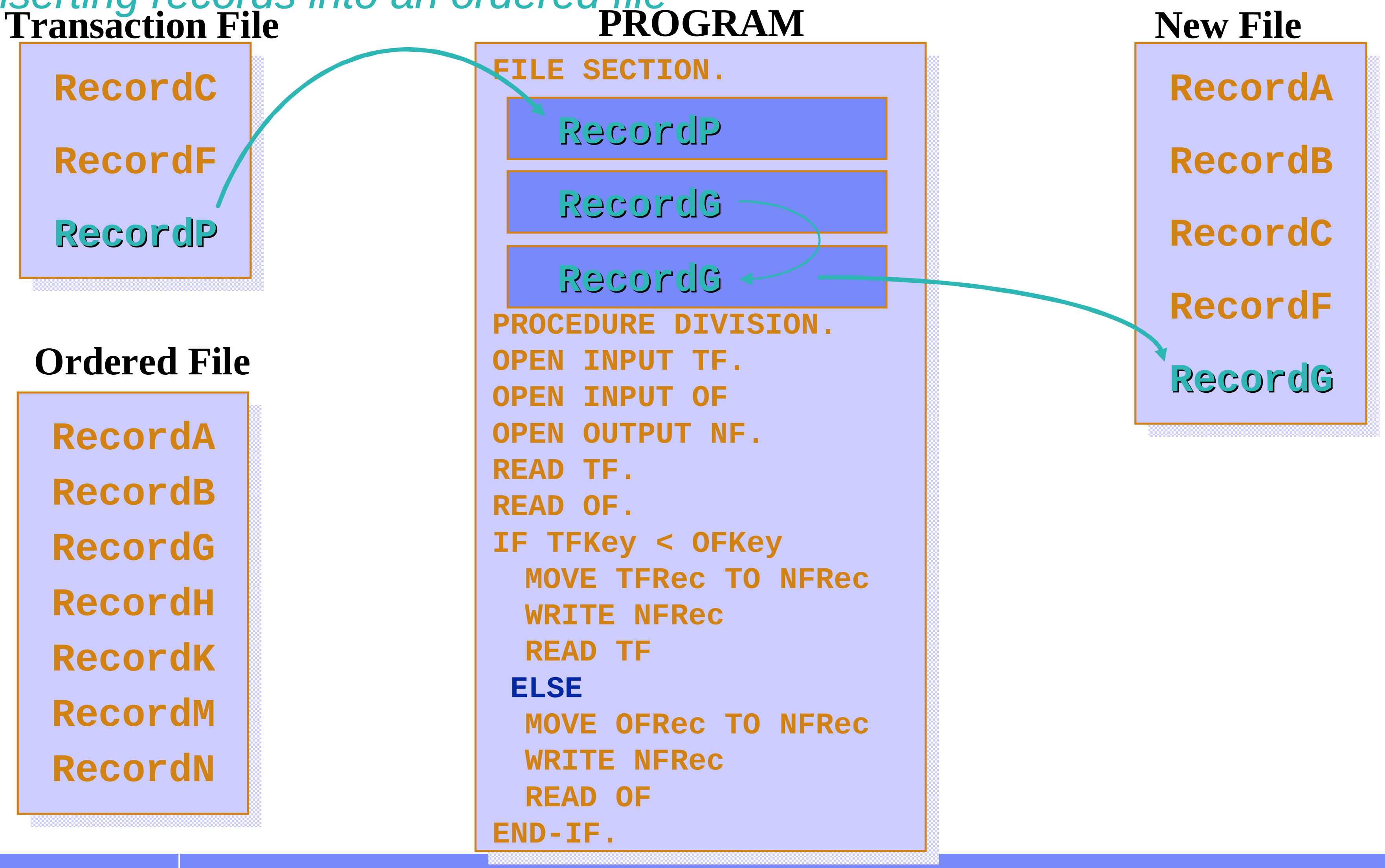
```
FILE SECTION.
  RecordF
  RecordG
  RecordF
PROCEDURE DIVISION.
OPEN INPUT TF.
OPEN INPUT OF
OPEN OUTPUT NF.
READ TF.
READ OF.
IF TFKey < OFKey
   MOVE TFRec TO NFRec
   WRITE NFRec
   READ TF
 ELSE
   MOVE OFRec TO NFRec
   WRITE NFRec
   READ OF
END-IF.
```

New File

RecordA
RecordB
RecordC
RecordF

Processing Sequential Files

Inserting records into an ordered file



Transaction File

RecordC

RecordF

RecordP

Ordered File

RecordA

RecordB

RecordG

RecordH

RecordK

RecordM

RecordN

PROGRAM

```
FILE SECTION.
    RecordP
    RecordG
    RecordG
PROCEDURE DIVISION.
OPEN INPUT TF.
OPEN INPUT OF
OPEN OUTPUT NF.
READ TF.
READ OF.
IF TFKey < OFKey
   MOVE TFRec TO NFRec
   WRITE NFRec
   READ TF
 ELSE
   MOVE OFRec TO NFRec
   WRITE NFRec
   READ OF
END-IF.
```

New File

RecordA

RecordB

RecordC

RecordF

RecordG

EXERCISE 2

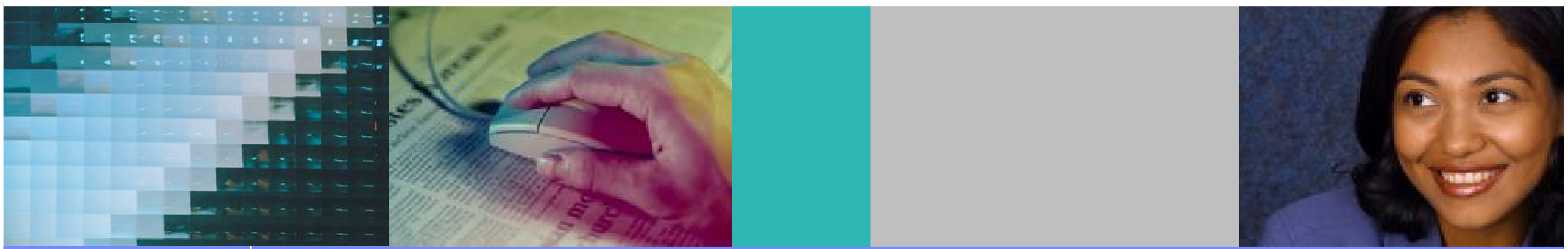


Table of contents

Introduction to COBOL
COBOL Basics 1
COBOL Basics 2
Introduction to Sequential Files
Processing Sequential Files
Simple iteration with the PERFORM verb
Arithmetic and Edited Pictures
Conditions
Tables and the PERFORM ... VARYING
Designing Programs

Simple iteration with the PERFORM verb

Overview

- Non-Iteration PERFORM.
- GO TO and PERFORM....THRU.
- In line and out of line PERFORM.
- PERFORM n TIMES.
- PERFORM .... UNTIL.
- Using the PERFORM...UNTIL in processing files.

Simple iteration with the PERFORM verb

The PERFORM Verb

- Iteration is an important programming construct. We use iteration when we need to repeat the same instructions over and over again.
- Most programming languages have several iteration keywords (e.g. WHILE, FOR, REPEAT) which facilitate the creation different ‘types’ of iteration structure.
- COBOL only has one iteration construct; PERFORM.
- But the PERFORM has several variations.
- Each variation is equivalent to one of the iteration ‘types’ available in other languages.
- This lecture concentrates on three of the PERFORM formats. The PERFORM..VARYING, the COBOL equivalent of the FOR , will be introduced later.

Simple iteration with the PERFORM verb

Paragraphs :- Revisited

- A Paragraph is a block of code to which we have given a name.
- A Paragraph Name is a programmer defined name formed using the standard rules for programmer defined names (A-Z, 0-9, -).
- A Paragraph Name is ALWAYS terminated with a ‘full-stop’.
- Any number of statements and sentences may be included in a paragraph, and the last one (at least) must be terminated with a ‘full-stop’.
- The scope of a paragraph is delimited by the occurrence of another paragraph name or the end of the program text.

Simple iteration with the PERFORM verb

Paragraph Example

```
ProcessRecord.
    DISPLAY StudentRecord
    READ StudentFile
            AT END MOVE HIGH-VALUES TO StudentRecord
    END-READ.

ProduceOutput.
    DISPLAY “Here is a message”.
```

NOTE

The scope of ‘ProcessRecord’ is delimited by the occurrence the paragraph name ‘ProduceOutput’.

Simple iteration with the PERFORM verb

Format 1 Syntax

$$\underline{\text{PERFORM}}\left[\text{1stProc}\left[\left\{\begin{array}{l}\underline{\text{THRU}}\\ \underline{\text{THROUGH}}\end{array}\right\}\text{EndProc}\right]\right]$$

- This is the only type of PERFORM that is not an iteration construct.
- It instructs the computer to transfer control to an out-of-line block of code.
- When the end of the block is reached, control reverts to the statement (not the sentence) immediately following the PERFORM.
- 1stProc and EndProc are the names of Paragraphs or Sections.
- The PERFORM..THRU instructs the computer to treat the Paragraphs or Sections from 1stProc TO EndProc as a single block of code.

Simple iteration with the PERFORM verb

Format 1 Example

Run of PerformFormat1

```
In TopLevel. Starting to run program
>>>> Now in OneLevelDown
>>>>>>>> Now in TwoLevelsDown.
>>>> Back in OneLevelDown
Back in TopLevel.
```

```
PROCEDURE DIVISION.
TopLevel.
    DISPLAY "In TopLevel. Starting to run program"
    PERFORM OneLevelDown
    DISPLAY "Back in TopLevel.".
    STOP RUN.

TwoLevelsDown.
    DISPLAY ">>>>>>>> Now in TwoLevelsDown."

OneLevelDown.
    DISPLAY ">>>> Now in OneLevelDown"
    PERFORM TwoLevelsDown
    DISPLAY ">>>> Back in OneLevelDown".
```

Simple iteration with the PERFORM verb

Format 1 Example

Run of PerformFormat1

```
In TopLevel. Starting to run program
>>>> Now in OneLevelDown
>>>>>>>> Now in TwoLevelsDown.
>>>> Back in OneLevelDown
Back in TopLevel.
```

```
PROCEDURE DIVISION.
TopLevel.
    DISPLAY "In TopLevel. Starting to run program"
    PERFORM OneLevelDown
    DISPLAY "Back in TopLevel.".
    STOP RUN.

TwoLevelsDown.
    DISPLAY ">>>>>>>> Now in TwoLevelsDown."

OneLevelDown.
    DISPLAY ">>>> Now in OneLevelDown"
    PERFORM TwoLevelsDown
    DISPLAY ">>>> Back in OneLevelDown".
```

Simple iteration with the PERFORM verb

Format 1 Example

Run of PerformFormat1

```
In TopLevel. Starting to run program
>>>> Now in OneLevelDown
>>>>>>>> Now in TwoLevelsDown.
>>>> Back in OneLevelDown
Back in TopLevel.
```

```
PROCEDURE DIVISION.
TopLevel.
    DISPLAY "In TopLevel. Starting to run program"
    PERFORM OneLevelDown
    DISPLAY "Back in TopLevel.".
    STOP RUN.

TwoLevelsDown.
    DISPLAY ">>>>>>>> Now in TwoLevelsDown."

OneLevelDown.
    DISPLAY ">>>> Now in OneLevelDown"
    PERFORM TwoLevelsDown
    DISPLAY ">>>> Back in OneLevelDown".
```

Simple iteration with the PERFORM verb

Format 1 Example

Run of PerformFormat1

```
In TopLevel. Starting to run program
>>>> Now in OneLevelDown
>>>>>>>> Now in TwoLevelsDown.
>>>> Back in OneLevelDown
Back in TopLevel.
```

```
PROCEDURE DIVISION.
TopLevel.
    DISPLAY "In TopLevel. Starting to run program"
    PERFORM OneLevelDown
    DISPLAY "Back in TopLevel.".
    STOP RUN.

TwoLevelsDown.
    DISPLAY ">>>>>>>> Now in TwoLevelsDown."

OneLevelDown.
    DISPLAY ">>>> Now in OneLevelDown"
    PERFORM TwoLevelsDown
    DISPLAY ">>>> Back in OneLevelDown".
```

Simple iteration with the PERFORM verb

Format 1 Example

Run of PerformFormat1

```
In TopLevel. Starting to run program
>>>> Now in OneLevelDown
>>>>>>>>> Now in TwoLevelsDown.
>>>> Back in OneLevelDown
Back in TopLevel.
```

```
PROCEDURE DIVISION.
TopLevel.
    DISPLAY "In TopLevel. Starting to run program"
    PERFORM OneLevelDown
    DISPLAY "Back in TopLevel.".
    STOP RUN.

TwoLevelsDown.
    DISPLAY ">>>>>>>>> Now in TwoLevelsDown."

OneLevelDown.
    DISPLAY ">>>> Now in OneLevelDown"
    PERFORM TwoLevelsDown
    DISPLAY ">>>> Back in OneLevelDown".
```

Simple iteration with the PERFORM verb

Format 1 Example

Run of PerformFormat1

```
In TopLevel. Starting to run program
>>>> Now in OneLevelDown
>>>>>>>> Now in TwoLevelsDown.
>>>> Back in OneLevelDown
Back in TopLevel.
```

```
PROCEDURE DIVISION.
TopLevel.
    DISPLAY "In TopLevel. Starting to run program"
    PERFORM OneLevelDown
    DISPLAY "Back in TopLevel.".
    STOP RUN.

TwoLevelsDown.
    DISPLAY ">>>>>>>> Now in TwoLevelsDown."

OneLevelDown.
    DISPLAY ">>>> Now in OneLevelDown"
    PERFORM TwoLevelsDown
    DISPLAY ">>>> Back in OneLevelDown".
```

Simple iteration with the PERFORM verb

Format 1 Example

Run of PerformFormat1

```
In TopLevel. Starting to run program
>>>> Now in OneLevelDown
>>>>>>>> Now in TwoLevelsDown.
>>>> Back in OneLevelDown
Back in TopLevel.
```

```
PROCEDURE DIVISION.
TopLevel.
    DISPLAY "In TopLevel. Starting to run program"
    PERFORM OneLevelDown
    DISPLAY "Back in TopLevel.".
    STOP RUN.

TwoLevelsDown.
    DISPLAY ">>>>>>>> Now in TwoLevelsDown."

OneLevelDown.
    DISPLAY ">>>> Now in OneLevelDown"
    PERFORM TwoLevelsDown
    DISPLAY ">>>> Back in OneLevelDown".
```

Simple iteration with the PERFORM verb

Why use the PERFORM Thru?

```
PROCEDURE DIVISION.
Begin.
   PERFORM SumSales
   STOP RUN.

SumSales.
   Statements
   Statements
   IF NoErrorFound
     Statements
     Statements
      IF NoErrorFound
       Statements
       Statements
       Statements
      END-IF
   END-IF.
```

Simple iteration with the PERFORM verb

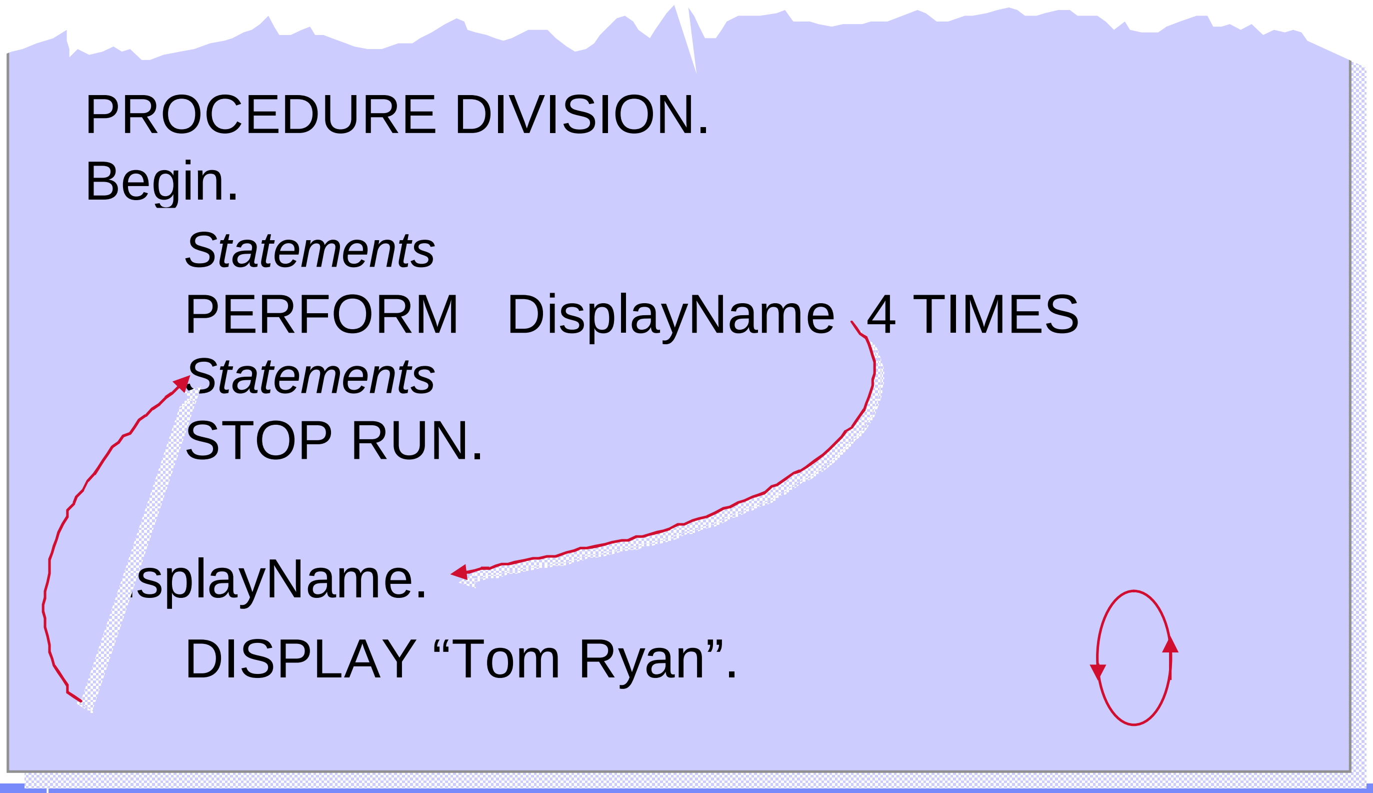
Go To and PERFORM THRU

```
PROCEDURE DIVISION
Begin.
    PERFORM SumSales THRU SumSalesExit
    STOP RUN.

SumSales.
   Statements
   Statements
    IF ErrorFound GO TO SumSalesExit
    END-IF
   Statements
   Statements
   Statements
    IF ErrorFound GO TO SumSalesExit
    END-IF
   Statements
SumSalesExit.
    EXIT.
```

Simple iteration with the PERFORM verb

Format 2 - Syntax

```
PERFORM  [ 1stProc  [ { THRU    } EndProc ] ]
                      { THROUGH }
   RepeatCoun   t TIMES
   [StatementB   lock  END  - PERFORM ]
```



```
PROCEDURE DIVISION.
Begin.
     Statements
     PERFORM   DisplayName  4 TIMES
     Statements
     STOP RUN.

 splayName.
     DISPLAY “Tom Ryan”.
```

Simple iteration with the PERFORM verb

Format 2 Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  PerformExample2.
AUTHOR.  Michael Coughlan.

DATA DIVISION.
WORKING-STORAGE SECTION.
01 NumofTimes      PIC 9 VALUE 5.

PROCEDURE DIVISION.
Begin.
    DISPLAY "Starting to run program"
    PERFORM 3 TIMES
       DISPLAY ">>>>This is an in line Perform"
    END-PERFORM
    DISPLAY "Finished in line Perform"
    PERFORM OutOfLineEG NumOfTimes TIMES
    DISPLAY "Back in Begin. About to Stop".
    STOP RUN.

OutOfLineEG.
    DISPLAY ">>>> This is an out of line Perform".
```

Run of PerformExample2

```
Starting to run program
>>>>This is an in line Perform
>>>>This is an in line Perform
>>>>This is an in line Perform
Finished in line Perform
>>>> This is an out of line Perform
>>>> This is an out of line Perform
>>>> This is an out of line Perform
>>>> This is an out of line Perform
>>>> This is an out of line Perform
Back in Begin. About to Stop
```

Simple iteration with the PERFORM verb

Format 3 - Syntax

$$\underline{\text{PERFORM}}\;\left[\text{1stProc}\;\left[\left\{\begin{array}{l}\underline{\text{THRU}}\\ \underline{\text{THROUGH}}\end{array}\right\}\text{EndProc}\right]\right]\left[\text{WITH}\;\;\underline{\text{TEST}}\;\left\{\begin{array}{l}\underline{\text{BEFORE}}\\ \underline{\text{AFTER}}\end{array}\right\}\right]$$

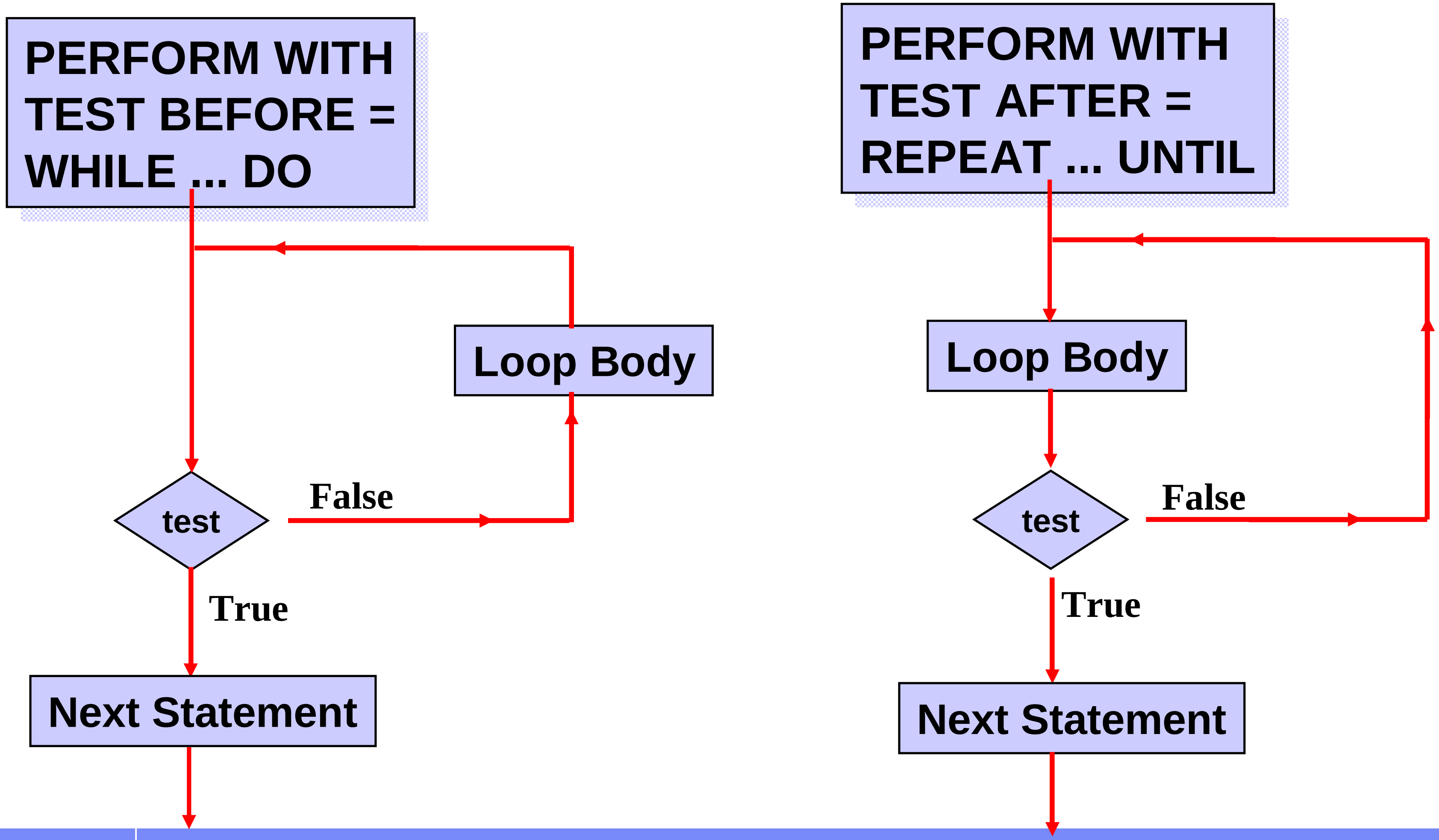
$$\underline{\text{UNTIL}}\;\;\text{Condition}$$

$$[\text{StatementB}\;\;\;\text{lock}\;\;\underline{\text{END - PERFORM}}\;]$$

- This format is used where the WHILE or REPEAT constructs are used in other languages.
- If the WITH TEST BEFORE phrase is used the PERFORM behaves like a WHILE loop and the condition is tested before the loop body is entered.
- If the WITH TEST AFTER phrase is used the PERFORM behaves like a REPEAT loop and the condition is tested after the loop body is entered.
- The WITH TEST BEFORE phrase is the default and so is rarely explicitly stated.

Simple iteration with the PERFORM verb

Format 3 - Sample



PERFORM WITH TEST BEFORE = WHILE ... DO

Loop Body

test

False

True

Next Statement

PERFORM WITH TEST AFTER = REPEAT ... UNTIL

Loop Body

test

False

True

Next Statement

Simple iteration with the PERFORM verb

Sequential File Processing

- In general terms, the WHILE loop is an ideal construct for processing sequences of data items whose length is not predefined.
- Such sequences of values are often called “streams”.
- Because the ‘length’ of the stream is unknown we have to be careful how we manage the detection of the end of the stream.
- A useful way for solving this problem uses a strategy known as “read ahead”.

Simple iteration with the PERFORM verb

The READ Ahead

- With the “read ahead” strategy we always try to stay one data item ahead of the processing.

- The general format of the “read ahead” algorithm is as follows;

```
Attempt to READ first data item
WHILE NOT EndOfStream
    Process data item
    Attempt to READ next data item
ENDWHILE
```

- Use this to process any stream of data.

Simple iteration with the PERFORM verb

Reading a Sequential File

- Algorithm Template

```
READ StudentRecords
    AT END MOVE HIGH-VALUES TO StudentRecord
END-READ
PERFORM UNTIL StudentRecord = HIGH-VALUES
    DISPLAY StudentRecord
    READ StudentRecords
       AT END MOVE HIGH-VALUES TO StudentRecord
    END-READ
END-PERFORM
```

- This is an example of an algorithm which is capable of processing any sequential file; ordered or unordered!

Simple iteration with the PERFORM verb

Sample

RUN OF SeqRead

```
9456789 COUGHLANMS LM51
9367892 RYAN     TG LM60
9368934 WILSON   HR LM61
```

```
PROCEDURE DIVISION.
Begin.
   OPEN INPUT StudentFile

   READ StudentFile
      AT END MOVE HIGH-VALUES TO StudentDetails
   END-READ
   PERFORM UNTIL StudentDetails = HIGH-VALUES
      DISPLAY StudentId SPACE StudentName SPACE CourseCode
      READ StudentFile
         AT END MOVE HIGH-VALUES TO StudentDetails
      END-READ
   END-PERFORM

   CLOSE StudentFile
   STOP RUN.
```

Table of contents

Arithmetic and Edited Pictures

Overview

- ROUNDED option.
- ON SIZE ERROR option.
- ADD, SUBTRACT, MULTIPLY, DIVIDE and COMPUTE.
- Edited PICTURE clauses.
- Simple Insertion.
- Special Insertion.
- Fixed Insertion.
- Floating Insertion.
- Suppression and Replacement.

Arithmetic and Edited Pictures

Arithmetic Verb Template

$$\text{VERB}\left\{\begin{matrix}\text{Identifier}\\ \text{Literal}\end{matrix}\right\}\left\{\begin{matrix}\text{TO}\\ \text{FROM}\\ \text{BY}\\ \text{INTO}\end{matrix}\right\}\left\{\begin{matrix}\text{Identifier}\ \ \mathbf{K}\\ \text{Identifier}\ \ \text{GIVING}\ \ \text{Identifier}\ \ \mathbf{K}\end{matrix}\right\}[\text{ROUNDED}\ ]$$

$$[\text{ON SIZE ERROR}\ \ \text{StatementB lock}\ \ \text{END - VERB}\ ]$$

- Most COBOL arithmetic verbs conform to the template above. For example;

```
ADD Takings TO CashTotal.
ADD Males TO Females GIVING TotalStudents.
SUBTRACT Tax FROM GrossPay.
SUBTRACT Tax FROM GrossPay GIVING NetPay.
DIVIDE Total BY Members GIVING MemberAverage.
DIVIDE Members INTO Total GIVING MemberAverage.
MULTIPLY 10 BY Magnitude.
MULTIPLY Members BY Subs GIVING TotalSubs.
```

- The exceptions are the COMPUTE and the DIVIDE with REMAINDER.

Arithmetic and Edited Pictures

The ROUNDED option

Receiving Field	Actual Result	Truncated Result	Rounded Result
PIC 9(3)V9.	123.25	123.2	123.3
PIC 9(3).	123.25	123	123

- The ROUNDED option takes effect when, after decimal point alignment, the result calculated must be truncated on the right hand side.

- The option adds 1 to the receiving item when the leftmost truncated digit has an absolute value of 5 or greater.

Arithmetic and Edited Pictures

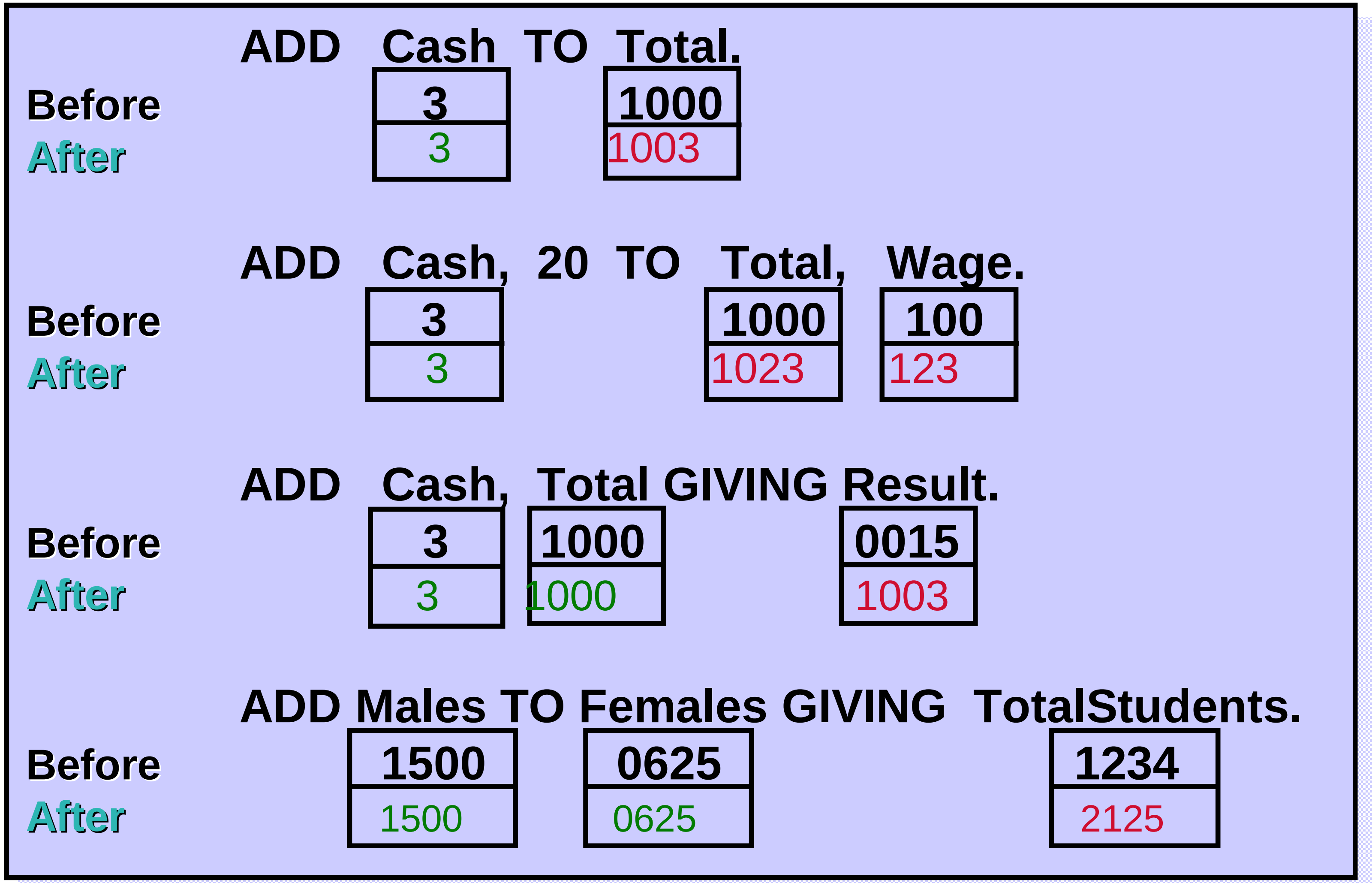
The ON SIZE ERROR option

Receiving Field	Actual Result	SIZE ERROR
PIC 9(3)V9.	245.96	Yes
PIC 9(3)V9.	1245.9	Yes
PIC 9(3).	124	No
PIC 9(3).	1246	Yes
PIC 9(3)V9 **Not Rounded**	124.45	Yes
PIC 9(3)V9 **Rounded**	124.45	No
PIC 9(3)V9 **Rounded**	3124.45	Yes

- A size error condition exists when, after decimal point alignment, the result is truncated on either the left or the right hand side.
- If an arithmetic statement has a rounded phrase then a size error only occurs if there is truncation on the left hand side (most significant digits).

Arithmetic and Edited Pictures

ADD Examples



ADD Cash TO Total.

	Cash	Total.
Before	3	1000
After	3	1003

ADD Cash, 20 TO Total, Wage.

	Cash,	Total,	Wage.
Before	3	1000	100
After	3	1023	123

ADD Cash, Total GIVING Result.

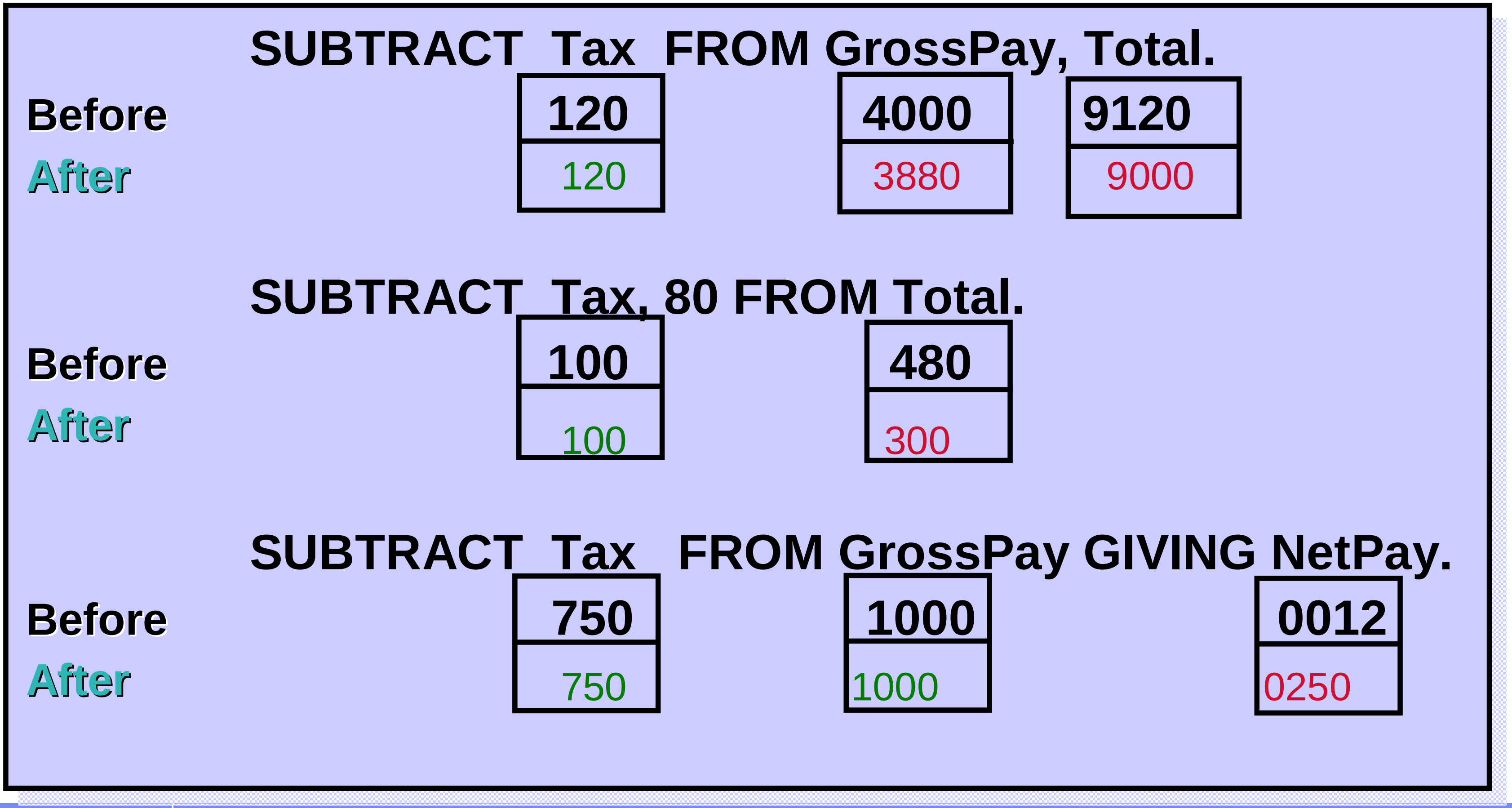
	Cash,	Total	Result.
Before	3	1000	0015
After	3	1000	1003

ADD Males TO Females GIVING TotalStudents.

	Males	Females	TotalStudents.
Before	1500	0625	1234
After	1500	0625	2125

Arithmetic and Edited Pictures

SUBTRACT Examples



SUBTRACT Tax FROM GrossPay, Total.

	Tax	GrossPay	Total
Before	120	4000	9120
After	120	3880	9000

SUBTRACT Tax, 80 FROM Total.

	Tax	Total
Before	100	480
After	100	300

SUBTRACT Tax FROM GrossPay GIVING NetPay.

	Tax	GrossPay	NetPay
Before	750	1000	0012
After	750	1000	0250

Arithmetic and Edited Pictures

MULTIPLY and DIVIDE Examples

```
MULTIPLY   Subs BY Members GIVING TotalSubs
      ON SIZE ERROR DISPLAY "TotalSubs too small"
END-MULTIPLY.
```

	Subs	Members	TotalSubs
Before	15.50	100	0123.45
After	15.50	100	1550.00

```
MULTIPLY 10 BY Magnitude,  Size.
```

Before	355	125
After	3550	1250

```
DIVIDE  Total   BY  Members GIVING Average  ROUNDED.
```

Before	9234.55	100	1234.56
After	9234.55	100	92.35

Arithmetic and Edited Pictures

The Divide Exception

$$\underline{\text{DIVIDE}} \begin{Bmatrix} \textit{Identifier} \\ \textit{Literal} \end{Bmatrix} \underline{\text{INTO}} \begin{Bmatrix} \textit{Identifier} \\ \textit{Literal} \end{Bmatrix} \underline{\text{GIVING}} \; \{\text{Identifier} \;\; [\; \underline{\text{ROUNDED}} \;]\} \;\; \underline{\text{REMAINDER}} \;\; \text{Identifier}$$

$$\left[\begin{Bmatrix} \text{ON } \; \underline{\text{SIZE}} \;\; \underline{\text{ERROR}} \\ \underline{\text{NOT}} \;\; \text{ON } \; \underline{\text{SIZE}} \;\; \underline{\text{ERROR}} \end{Bmatrix} \text{StatementB} \;\; \text{lock} \;\; \underline{\text{END}} \;\; \underline{\text{- DIVIDE}} \right]$$

$$\underline{\text{DIVIDE}} \begin{Bmatrix} \textit{Identifier} \\ \textit{Literal} \end{Bmatrix} \underline{\text{BY}} \begin{Bmatrix} \textit{Identifier} \\ \textit{Literal} \end{Bmatrix} \underline{\text{GIVING}} \; \{\text{Identifier} \;\; [\; \underline{\text{ROUNDED}} \;]\} \;\; \underline{\text{REMAINDER}} \;\; \text{Identifier}$$

$$\left[\begin{Bmatrix} \text{ON } \; \underline{\text{SIZE}} \;\; \underline{\text{ERROR}} \\ \underline{\text{NOT}} \;\; \text{ON } \; \underline{\text{SIZE}} \;\; \underline{\text{ERROR}} \end{Bmatrix} \text{StatementB} \;\; \text{lock} \;\; \underline{\text{END}} \;\; \underline{\text{- DIVIDE}} \right]$$

DIVIDE 201 BY 10 GIVING Quotient REMAINDER Remain.

Before	209	424
After	020	001

Arithmetic and Edited Pictures

The COMPUTE

$$\underline{\text{COMPUTE}}\ \{\text{Identifier}\ [\ \underline{\text{ROUNDED}}\ ]\}\ldots = \text{ArithmeticExpression}$$

$$\left[\begin{Bmatrix}\text{ON}\ \underline{\text{SIZE ERROR}}\\ \underline{\text{NOT}}\ \text{ON}\ \underline{\text{SIZE ERROR}}\end{Bmatrix}\ \text{StatementBlock}\ \underline{\text{END-COMPUTE}}\right]$$

Precedence Rules.

1.	**	=	**POWER**	N^N
2.	*	=	**MULTIPLY**	x
	/	=	**DIVIDE**	÷
3.	+	=	**ADD**	+
	-	=	**SUBTRACT**	-

Compute IrishPrice = SterlingPrice / Rate * 100.

Before	1000.50	156.25	87
After	179.59	156.25	87

Arithmetic and Edited Pictures

Edited Pictures

- Edited Pictures are PICTURE clauses which format data intended for output to screen or printer.
- To enable the data items to be formatted in a particular style COBOL provides additional picture symbols supplementing the basic 9, X, A, V and S symbols.
- The additional symbols are referred to as “Edit Symbols” and PICTURE clauses which include edit symbols are called “Edited Pictures”.
- The term edit is used because the edit symbols have the effect of changing, or editing, the data inserted into the edited item.
- Edited items can not be used as operands in a computation but they may be used as the result or destination of a computation (i.e. to the right of the word GIVING).

Arithmetic and Edited Pictures

Editing Types

- COBOL provides two basic types of editing
  - Insertion Editing - which modifies a value by including additional items.
  - Suppression and Replacement Editing - which suppresses and replaces leading zeros.

- Each type has sub-categories
  - Insertion editing
    - Simple Insertion
    - Special Insertion
    - Fixed Insertion
    - Floating Insertion
  - Suppression and Replacement
    - Zero suppression and replacement with spaces
    - Zero suppression and replacement with asterisks

Arithmetic and Edited Pictures

Editing Symbols

Edit Symbol	Editing Type
, B 0 /	Simple Insertion
.	Special Insertion
+ - CR DB $	Fixed Insertion
+ - S	Floating Insertion
Z *	Suppression and Replacement

Arithmetic and Edited Pictures

Simple Insertion

Sending		Receiving	
Picture	**Data**	**Picture**	**Result**
PIC 999999	123456	PIC 999,999	123,456
PIC 9(6)	000078	PIC 9(3),9(3)	000,078
PIC 9(6)	000078	PIC ZZZ,ZZZ	□ □ □78 □
PIC 9(6)	000178	PIC ***,***	****178
PIC 9(6)	002178	PIC ***,***	**2,178
PIC 9(6)	120183	PIC 99B99B99	12□01□83
PIC 9(6)	120183	PIC 99/99/99	12/01/83
PIC 9(6)	001245	PIC 990099	120045

Arithmetic and Edited Pictures

Special Insertion

Sending		Receiving	
Picture	Data	Picture	Result
PIC 999V99	12345	PIC 999.99	123.45
PIC 999V99	02345	PIC 999.9	023.4
PIC 999V99	51234	PIC 99.99	12.34
PIC 999	456	PIC 999.99	456.00

Arithmetic and Edited Pictures

Fixed Insertion - Plus and Minus

Sending		Receiving	
Picture	Data	Picture	Result
PIC S999	-123	PIC -999	-123
PIC S999	-123	PIC 999-	123-
PIC S999	+123	PIC -999	□123
PIC S9(5)	+12345	PIC +9(5)	+12345
PIC S9(3)	-123	PIC +9(3)	-123
PIC S9(3)	-123	PIC 999+	123-

Arithmetic and Edited Pictures

Fixed Insertion - Credit, Debit, $

Sending		Receiving	
Picture	**Data**	**Picture**	**Result**
PIC S9(4)	+1234	PIC 9(4)CR	1234□ □
PIC S9(4)	-1234	PIC 9(4)CR	1234CR
PIC S9(4)	+1234	PIC 9(4)DB	1223□ □
PIC S9(4)	-1234	PIC 9(4)DB	1234DB
PIC 9(4)	1234	PIC $99999	$01234
PIC 9(4)	0000	PIC $ZZZZZ	$□ □ □ □ □

Arithmetic and Edited Pictures

Floating Insertion

Sending		Receiving	
Picture	Data	Picture	Result
PIC 9(4)	0000	PIC $$,$$9.99	$0.00
PIC 9(4)	0080	PIC $$,$$9.00	$80.00
PIC 9(4)	0128	PIC $$,$$9.99	$128.00
PIC 9(5)	57397	PIC $$,$$9	$7,397
PIC S9(4)	- 0005	PIC +++++9	-5
PIC S9(4)	+0080	PIC +++++9	+80
PIC S9(4)	- 0080	PIC -----9	-80
PIC S9(5)	+71234	PIC -----9	ž1234

Arithmetic and Edited Pictures

Suppression and Replacement

Sending		Receiving	
Picture	Data	Picture	Result
PIC 9(5)	12345	PIC ZZ,999	12,345
PIC 9(5)	01234	PIC ZZ,999	□1,234
PIC 9(5)	00123	PIC ZZ,999	□ □123
PIC 9(5)	00012	PIC ZZ,999	□ □012
PIC 9(5)	05678	PIC **,**9	*5,678
PIC 9(5)	00567	PIC **,**9	***567
PIC 9(5)	00000	PIC **,***	******

EXERCISE 3

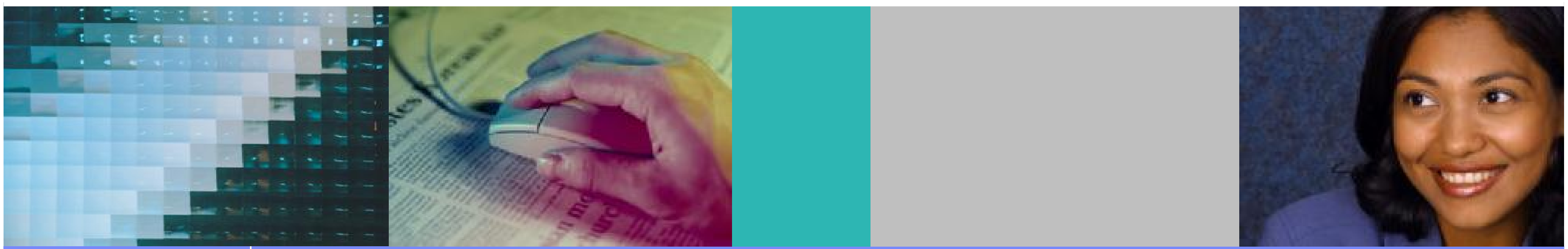


Table of contents

Introduction to COBOL
COBOL Basics 1
COBOL Basics 2
Introduction to Sequential Files
Processing Sequential Files
Simple iteration with the PERFORM verb
Arithmetic and Edited Pictures
Conditions
Tables and the PERFORM ... VARYING
Designing Programs

Conditions

Overview

- IF..THEN...ELSE.
- Relation conditions.
- Class conditions.
- Sign conditions.
- Complex conditions.
- Implied Subjects.
- Nested IFs and the END-IF.
- Condition names and level 88's.
- The SET verb.

Conditions

IF Syntax

$$\underline{\mathit{IF}}\ \text{Condition}\quad \text{THEN}\ \begin{Bmatrix}\text{StatementB lock}\\ \underline{\text{NEXT SENTENCE}}\end{Bmatrix}$$

$$\underline{\text{ELSE}}\ \begin{Bmatrix}\text{StatementB lock}\\ \underline{\text{NEXT SENTENCE}}\end{Bmatrix}\ \underline{[\text{END - IF}]}$$

CONDITION TYPES

- Simple Conditions
  - Relation Conditions
  - Class Conditions
  - Sign Conditions
- Complex Conditions
- Condition Names

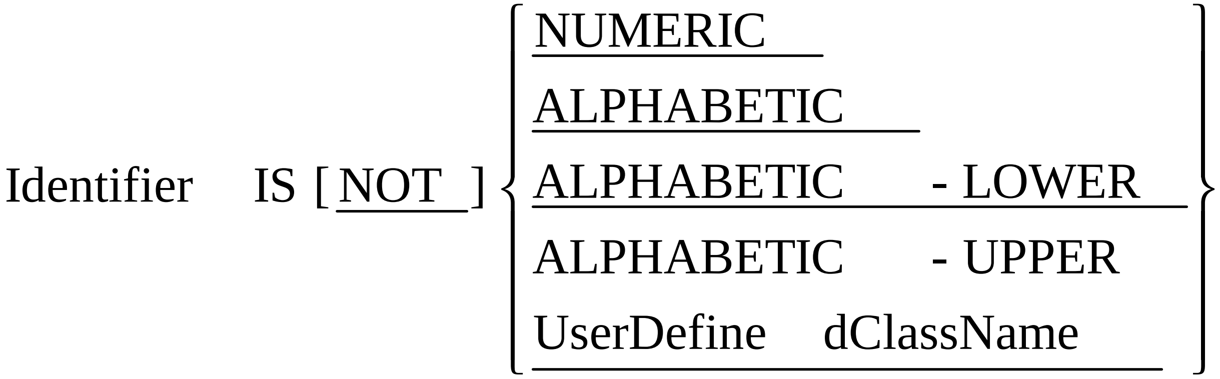
Conditions

Relation Conditions

$$\left\{\begin{array}{l} \textit{Identifier} \\ \textit{Literal} \\ \textit{Arithmetic Expression} \end{array}\right\} \text{IS} \left\{\begin{array}{l} [\underline{\text{NOT}}]\ \underline{\text{GREATER}}\ \text{THAN} \\ [\underline{\text{NOT}}] > \\ [\underline{\text{NOT}}]\ \underline{\text{LESS}}\ \text{THAN} \\ [\underline{\text{NOT}}] < \\ [\underline{\text{NOT}}]\ \underline{\text{EQUAL}}\ \text{TO} \\ [\underline{\text{NOT}}] = \\ \underline{\text{GREATER}}\ \text{THAN}\ \underline{\text{OR EQUAL}}\ \text{TO} \\ >= \\ \underline{\text{LESS}}\ \text{THAN}\ \underline{\text{OR EQUAL}}\ \text{TO} \\ <= \end{array}\right\} \left\{\begin{array}{l} \textit{Identifier} \\ \textit{Literal} \\ \textit{Arithmetic Expression} \end{array}\right\}$$

Conditions

Class Conditions



```
                                 ┌ NUMERIC                        ┐
                                 │ ALPHABETIC                     │
Identifier   IS [ NOT ]         {  ALPHABETIC        - LOWER      }
                                 │ ALPHABETIC        - UPPER      │
                                 └ UserDefine   dClassName        ┘
```

- Although COBOL data items are not ‘typed’ they do fall into some broad categories, or classes, such a numeric or alphanumeric, etc.
- A Class Condition determines whether the value of data item is a member of one these classes.

Conditions

Sign Conditions

$$\text{ArithExp} \qquad \text{IS} \; [\,\underline{\text{NOT}}\,] \left\{ \begin{array}{l} \underline{\text{POSITIVE}} \\ \underline{\text{NEGATIVE}} \\ \underline{\text{ZERO}} \end{array} \right\}$$

- The sign condition determines whether or not the value of an arithmetic expression is less than, greater than or equal to zero.
- Sign conditions are just another way of writing some of the Relational conditions.

Conditions

Complex conditions

$$\mathit{Condition}\ \left\{ \left\{ \begin{array}{l} \underline{\text{AND}} \\ \underline{\text{OR}} \end{array} \right\} \text{Condition} \right\} \mathbf{K}$$

- Programs often require conditions which are more complex than single value testing or determining a data class.
- Like all other programming languages COBOL allows simple conditions to be combined using OR and AND to form composite conditions.
- Like other conditions, a complex condition evaluates to true or false.
- A complex condition is an expression which is evaluated from left to right unless the order of evaluation is changed by the precedence rules or bracketing.

Conditions

Complex conditions have precedence rules too

Precedence Rules.

1.	**NOT**	=	**
2.	**AND**	=	* or /
3.	**OR**	=	+ or -

- Just like arithmetic expressions, complex conditions are evaluated using precedence rules and the order of evaluation may be changed by bracketing.

- Examples

```
IF  ( Row > 0)   AND   (Row < 26)   THEN
      DISPLAY “On Screen”
END-IF

IF ( VarA > VarC )   OR ( VarC = VarD  )  OR  ( VarA NOT = VarF )
      DISPLAY “Done”
END-IF
```

Conditions

Implied Subjects

- When a data item is involved in a relation condition with each of a number of other items it can be tedious to have to repeat the data item for each condition. For example,

```
IF TotalAmt > 10000 AND TotalAmt < 50000 THEN
IF Grade = “A” OR Grade = “B+” OR GRADE = “B” THEN
IF VarA > VarB AND VarA > VarC AND VarA > VarD
        DISPLAY “VarA is the Greatest”
END-IF
```

- In these situations COBOL provides an abbreviation mechanism called implied subjects.
- The statements above may be re-written using implied subjects as;

```
IF TotalAmt > 10000 AND  < 50000 THEN
IF Grade=“A” OR “B+” OR “B” THEN
IF VarA > VarB AND VarC AND VarD
        DISPLAY “VarA is the Greatest”
END-IF
```

Implied Subjects
TotalAmt
Grade =
VarA >

Conditions

Nested IFs

```
IF ( VarA < 10 ) AND ( VarB NOT > VarC ) THEN
      IF VarG = 14 THEN
         DISPLAY “First”
      ELSE
         DISPLAY “Second”
      END-IF
ELSE
   DISPLAY “Third”
END-IF
```

VarA	VarB	VarC	VarG	DISPLAY
3 T	4 T	15	14 T	First
3 T	4 T	15	15 F	Second
3 T	4 F	3	14	Third
13 F	4 T	15	14	Third

Conditions

Condition Names

$$\underbrace{\textit{IF}\ \text{VarA GREATER THAN VarB THEN}\ \textit{Action}}_{\substack{\text{Condition is either} \\ \text{TRUE or False}}}$$

- Wherever a condition can occur, such as in an IF statement or an EVALUATE or a PERFORM..UNTIL, a CONDITION NAME (Level 88) may be used.
- A Condition Name is essentially a BOOLEAN variable which is either TRUE or FALSE.
- Example.

  IF StudentRecord = HIGH-VALUES THEN *Action*

  The statement above may be replaced by the one below. The condition name EndOfStudentFile may be used instead of the condition StudentRecord = HIGH-VALUES.

  IF EndOfStudentFile THEN *Action*

Conditions

Defining Condition Names

$$88\ \text{ConditionN ame}\left\{\begin{matrix}\underline{\text{VALUE}}\\ \underline{\text{VALUES}}\end{matrix}\right\}\left\{\begin{matrix}\text{Literal}\\ \text{LowValue}\left\{\begin{matrix}\underline{\text{THROUGH}}\\ \underline{\text{THRU}}\end{matrix}\right\}\text{HighValue}\end{matrix}\right\}\mathbf{K}$$

- **Condition Names are defined in the DATA DIVISION using the special level number 88.**

- **They are always associated with a data item and are defined immediately after the definition of the data item.**

- **A condition name takes the value TRUE or FALSE depending on the value in its associated data item.**

- **A Condition Name may be associated with ANY data item whether it is a group or an elementary item.**

- **The VALUE clause is used to identify the values which make the Condition Name TRUE.**

Conditions

Example

```
01  CityCode                       PIC 9 VALUE 5.
    88  Dublin                     VALUE 1.
    88  Limerick                   VALUE 2.
    88  Cork                       VALUE 3.
    88  Galway                     VALUE 4.
    88  Sligo                      VALUE 5.
    88 Waterford                   VALUE 6.
    88 UniversityCity              VALUE 1 THRU 4.
```

City Code

5

```
IF Limerick
   DISPLAY "Hey, we're home."
END-IF
IF UniversityCity
   PERFORM CalcRentSurcharge
END-IF
```

Dublin	FALSE
Limerick	FALSE
Cork	FALSE
Galway	FALSE
Sligo	TRUE
Waterford	FALSE
UniversityCity	FALSE

Conditions

Example

```
01  CityCode                          PIC 9 VALUE 5.
    88  Dublin                        VALUE 1.
    88  Limerick                      VALUE 2.
    88  Cork                          VALUE 3.
    88  Galway                        VALUE 4.
    88  Sligo                         VALUE 5.
    88 Waterford                      VALUE 6.
    88 UniversityCity                 VALUE 1 THRU 4.
```

```
IF Limerick
   DISPLAY "Hey, we're home."
END-IF
IF UniversityCity
   PERFORM CalcRentSurcharge
END-IF
```

City Code

2

Dublin	FALSE
Limerick	TRUE
Cork	FALSE
Galway	FALSE
Sligo	FALSE
Waterford	FALSE
UniversityCity	TRUE

Conditions

Example

```
01  CityCode                     PIC 9 VALUE 5.
    88  Dublin                   VALUE 1.
    88  Limerick                 VALUE 2.
    88  Cork                     VALUE 3.
    88  Galway                   VALUE 4.
    88  Sligo                    VALUE 5.
    88 Waterford                 VALUE 6.
    88 UniversityCity            VALUE 1 THRU 4.
```

City Code

6

```
IF Limerick
   DISPLAY "Hey, we're home."
END-IF
IF UniversityCity
   PERFORM CalcRentSurcharge
END-IF
```

Dublin	**FALSE**
Limerick	**FALSE**
Cork	**FALSE**
Galway	**FALSE**
Sligo	**FALSE**
Waterford	**TRUE**
UniversityCity	**FALSE**

Conditions

Example

```
01  InputChar           PIC X.
    88  Vowel           VALUE   "A","E","I","O","U".
    88  Consonant       VALUE   "B" THRU "D", "F","G","H"
                                "J" THRU "N", "P" THRU "T"
                                "V" THRU "Z".
    88 Digit            VALUE "0" THRU "9".
    88 LowerCase        VALUE   "a" THRU "z".
    88  ValidChar       VALUE   "A" THRU "Z","0" THRU "9".
```

```
IF ValidChar
   DISPLAY "Input OK."
END-IF
IF LowerCase
     DISPLAY "Not Upper Case"
END-IF
IF Vowel
   Display "Vowel entered."
END-IF
```

Input Char

E

Vowel	TRUE
Consonant	FALSE
Digit	FALSE
LowerCase	FALSE
ValidChar	TRUE

Conditions

Example

```
01  InputChar        PIC X.
    88  Vowel        VALUE   "A","E","I","O","U".
    88  Consonant    VALUE   "B" THRU "D", "F","G","H"
                             "J" THRU "N", "P" THRU "T"
                             "V" THRU "Z".
    88 Digit         VALUE "0" THRU "9".
    88 LowerCase     VALUE   "a" THRU "z".
    88  ValidChar    VALUE   "A" THRU "Z","0" THRU "9".
```

```
IF ValidChar
   DISPLAY "Input OK."
END-IF
IF LowerCase
     DISPLAY "Not Upper Case"
END-IF
IF Vowel
   Display "Vowel entered."
END-IF
```

Input Char

4

Vowel	FALSE
Consonant	FALSE
Digit	TRUE
LowerCase	FALSE
ValidChar	TRUE

Conditions

Example

```
01  InputChar          PIC X.
    88  Vowel          VALUE   "A","E","I","O","U".
    88  Consonant      VALUE   "B" THRU "D",  "F","G","H"
                               "J" THRU "N",  "P" THRU "T"
                               "V" THRU "Z".
    88 Digit           VALUE "0" THRU "9".
    88 LowerCase       VALUE   "a" THRU "z".
    88  ValidChar      VALUE   "A" THRU "Z","0" THRU "9".
```

```
IF ValidChar
   DISPLAY "Input OK."
END-IF
IF LowerCase
     DISPLAY "Not Upper Case"
END-IF
IF Vowel
   Display "Vowel entered."
END-IF
```

Input Char

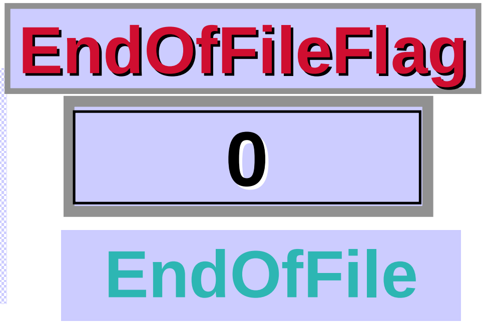
g

Vowel	FALSE
Consonant	FALSE
Digit	FALSE
LowerCase	TRUE
ValidChar	FALSE

Conditions

Example

```
01 EndOfFileFlag   PIC 9 VALUE 0.
   88 EndOfFile     VALUE 1.
```

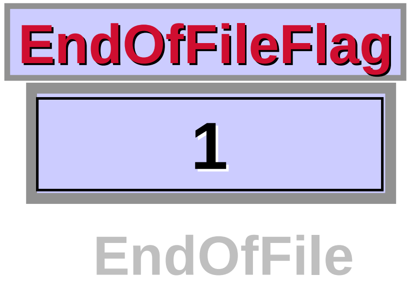


```
READ InFile
    AT END MOVE 1 TO EndOfFileFlag
END-READ
PERFORM UNTIL EndOfFile
    Statements
    READ InFile
        AT END MOVE 1 TO EndOfFileFlag
    END-READ
END-PERFORM
```

Conditions

Example

```
01  EndOfFileFlag  PIC 9 VALUE 0.
    88  EndOfFile        VALUE 1.
```

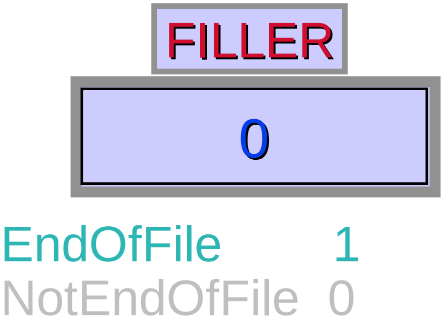


```
READ InFile
    AT END MOVE 1 TO EndOfFileFlag
END-READ
PERFORM UNTIL EndOfFile
    Statements
    READ InFile
        AT END MOVE 1 TO EndOfFileFlag
    END-READ
END-PERFORM
```

Conditions

Using the SET verb

```
01  FILLER              PIC 9 VALUE 0.
    88  EndOfFile     VALUE 1.
    88 NotEndOfFile VALUE 0.
```



FILLER

0

EndOfFile 1

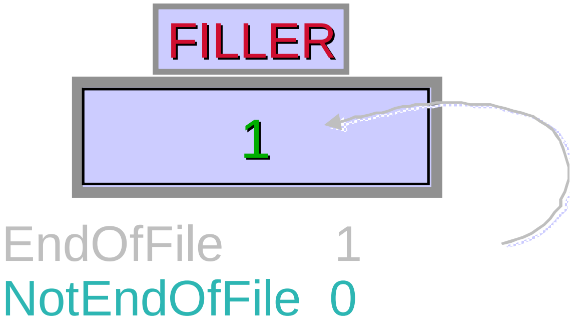
NotEndOfFile 0

```
READ InFile
    AT END SET EndOfFile TO TRUE
END-READ
PERFORM UNTIL EndOfFile
    Statements
    READ InFile
        AT END SET EndOfFile TO TRUE
    END-READ
END-PERFORM
Set NotEndOfFile TO TRUE.
```

Conditions

Using the SET verb

```
01   FILLER              PIC 9 VALUE 0.
     88   EndOfFile      VALUE 1.
     88 NotEndOfFile VALUE 0.
```



FILLER

1

EndOfFile 1

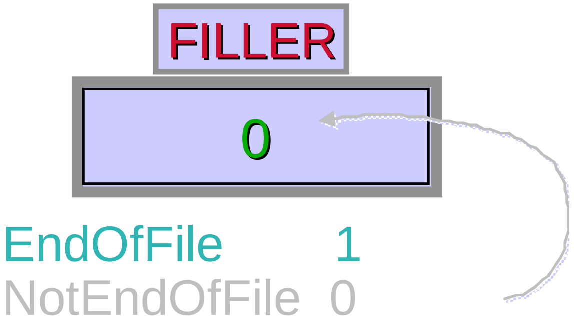
NotEndOfFile 0

```
READ InFile
    AT END SET EndOfFile TO TRUE
END-READ
PERFORM UNTIL EndOfFile
    Statements
    READ InFile
        AT END SET EndOfFile TO TRUE
    END-READ
END-PERFORM
Set NotEndOfFile TO TRUE.
```

Conditions

Using the SET verb

```
01   FILLER               PIC 9 VALUE 0.
     88   EndOfFile    VALUE 1.
     88 NotEndOfFile VALUE 0.
```



FILLER

0

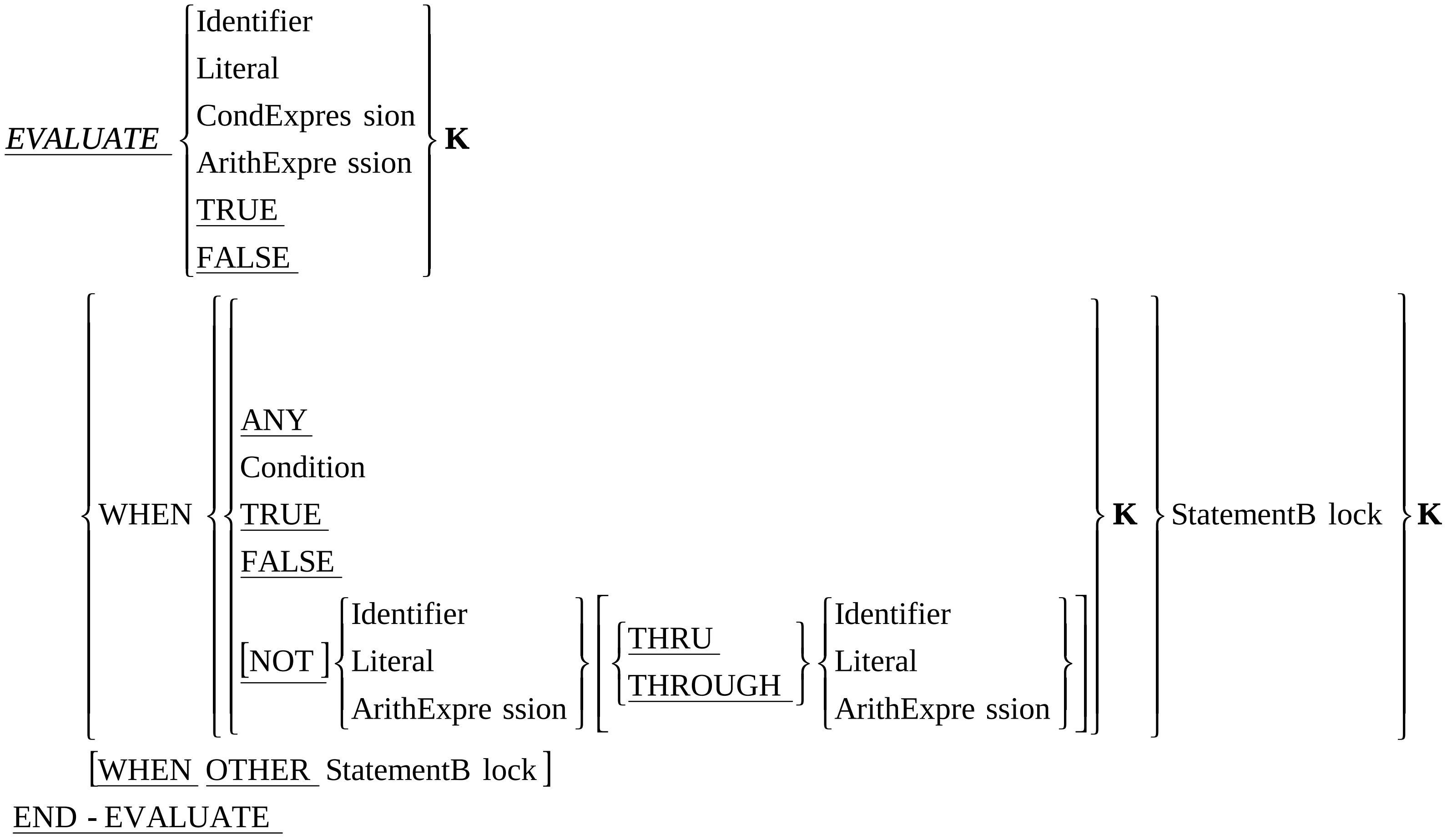
EndOfFile 1

NotEndOfFile 0

```
READ InFile
     AT END SET EndOfFile TO TRUE
END-READ
PERFORM UNTIL EndOfFile
     Statements
     READ InFile
          AT END SET EndOfFile TO TRUE
     END-READ
END-PERFORM
Set NotEndOfFile TO TRUE.
```

Conditions

The Evaluate



EVALUATE

Identifier

Literal

CondExpres sion

ArithExpre ssion

TRUE

FALSE

K

WHEN

ANY

Condition

TRUE

FALSE

[NOT]

Identifier

Literal

ArithExpre ssion

THRU

THROUGH

Identifier

Literal

ArithExpre ssion

K

StatementB lock

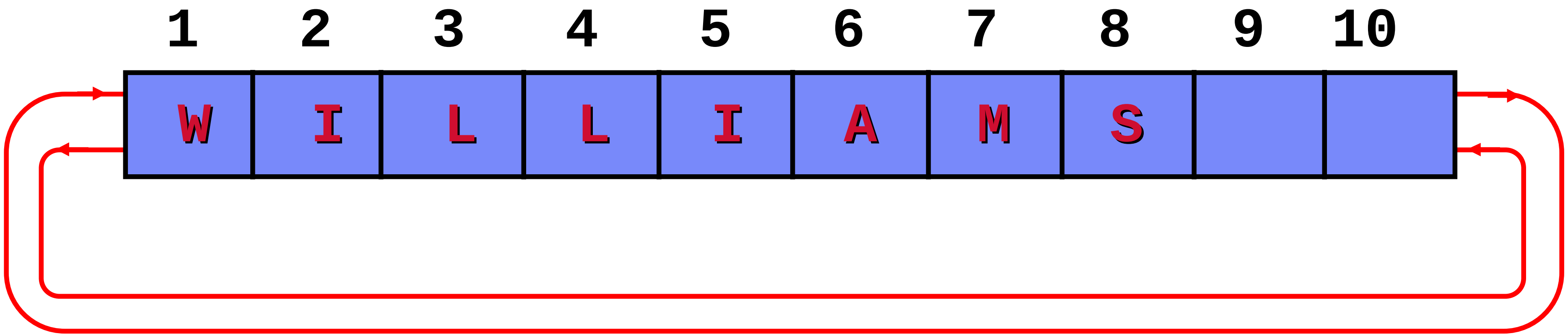
K

[WHEN OTHER StatementB lock]

END - EVALUATE

Conditions

The Evaluate



1	2	3	4	5	6	7	8	9	10
W	I	L	L	I	A	M	S		

```
EVALUATE  TRUE  Position
    WHEN L-Arrow   2 THRU 10PERFORM MoveLeft
    WHEN R-Arrow   1 THRU  9PERFORM MoveRight
    WHEN L-Arrow       1    MOVE 10 TO Position
    WHEN R-Arrow      10    MOVE  1 TO Position
    WHEN DeleteKey     1    PERFORM CantDelete      WHEN Character
ANY PERFORM InsertChar      WHEN OTHER PERFORM DisplayErrorMessage
END-EVALUATE
```

Conditions

Decision Table Implementation

Gender	M	F	M	F	M	F	M	F	
Age	<20	<20	20-40	20-40	40>	40>	20-40	20-40	etc
Service	Any	Any	<10	<10	<10	<10	10-20	10-20	etc
% Bonus	5	10	12	13	20	15	14	23	

```
EVALUATE Gender      TRUE              TRUE
    WHEN        "M"    Age<20                    ANY          MOVE  5 TO Bonus
    WHEN        "F"    Age<20                    ANY          MOVE 10 TO Bonus
    WHEN        "M"    Age>19 AND <41         Service<10      MOVE 12 TO Bonus
    WHEN        "F"    Age>19 AND <41         Service<10      MOVE 13 TO Bonus
    WHEN        "M"    Age>40                 Service<10      MOVE 20 TO Bonus
    WHEN        "F"    Age>40                 Service<10      MOVE 15 TO Bonus
        :       :      :                      :               :
        :       :      :                      :               :
    WHEN        "F"        ANY                Service>20      MOVE 25 TO Bonus
END-EVALUATE.
```

Table of contents

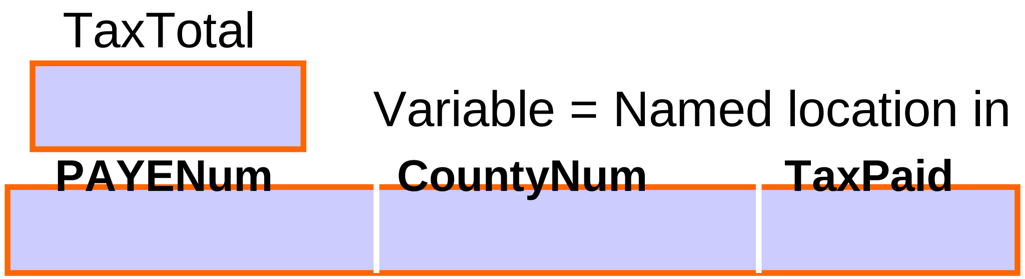
Introduction to COBOL
COBOL Basics 1
COBOL Basics 2
Introduction to Sequential Files
Processing Sequential Files
Simple iteration with the PERFORM verb
Arithmetic and Edited Pictures
Conditions
Tables and the PERFORM ... VARYING
Designing Programs

Tables and the PERFORM ... VARYING

Overview

- Introduction to tables.
- Declaring tables.
- Processing tables using the PERFORM..VARYING.

Tables and the PERFORM ... VARYING



Variable = Named location in memory

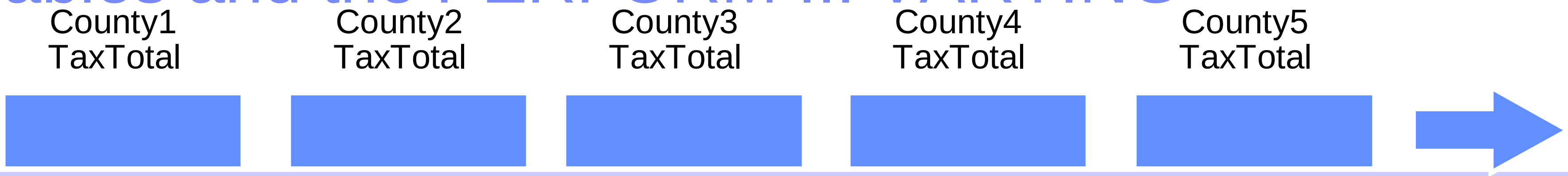
```
PROCEDURE DIVISION.
Begin.
   OPEN INPUT TaxFile
   READ TaxFile
      AT END SET EndOfTaxFile TO TRUE
   END-READ
   PERFORM UNTIL EndOfTaxFile
      ADD TaxPaid TO TaxTotal
      READ TaxFile
         AT END SET EndOfTaxFile TO TRUE
      END-READ
   END-PERFORM.
   DISPLAY "Total taxes are ", TaxTotal
   CLOSE TaxFile
   STOP RUN.
```

The program to calculate the total taxes paid for the country is easy to write.

BUT.

What do we do if we want to calculate the taxes paid in each county?

Tables and the PERFORM ... VARYING



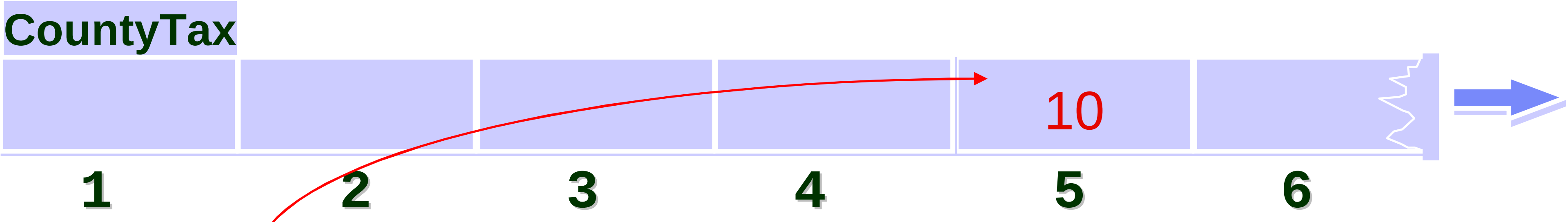
```
PROCEDURE DIVISION.
Begin.
   OPEN INPUT TaxFile
   READ TaxFile
      AT END SET EndOfTaxFile TO TRUE
   END-READ

   PERFORM SumCountyTaxes UNTIL EndOfTaxFile

   DISPLAY "County 1 total is ", County1TaxTotal
      :    :    : 24 Statements  :    :    :
   DISPLAY "County 26 total is ", County26TaxTotal
   CLOSE TaxFile
   STOP RUN.

SumCountyTaxes.
   IF CountyNum = 1 ADD TaxPaid TO County1TaxTotal
   END-IF
      :    :    : 24 Statements  :    :    :
   IF CountyNum = 26 ADD TaxPaid TO County26TaxTotal
   END-IF
   READ TaxFile
      AT END SET EndOfTaxFile TO TRUE
   END-READ
```

58 Statements

Tables and the PERFORM ... VARYING

Tables/Arrays

A table is a contiguous sequence of memory locations called elements, which all have the same name, and are uniquely identified by that name and by their position in the sequence.

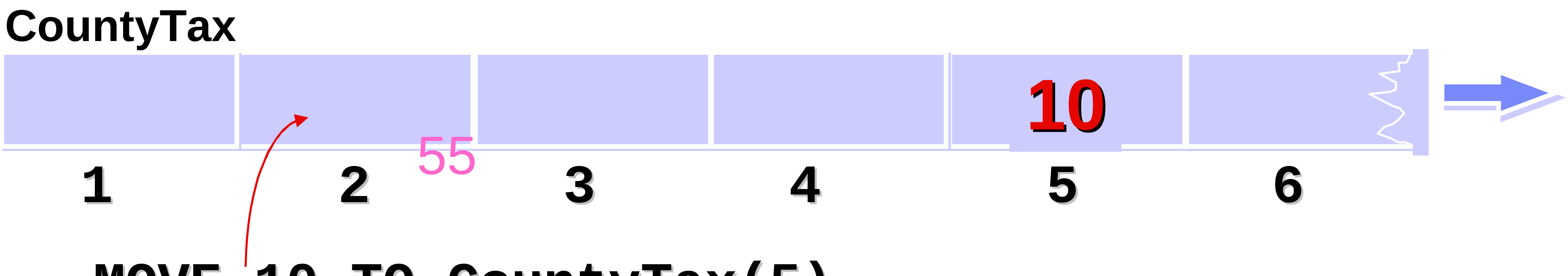


```
MOVE 10 TO CountyTax(5)
ADD TaxPaid TO CountyTax(CountyNum)
ADD TaxPaid TO CountyTax(CountyNum + 2)
```

Tables and the PERFORM ... VARYING

Tables/Arrays

A table is a contiguous sequence of memory locations called elements, which all have the same name, and are uniquely identified by that name and by their position in the sequence.

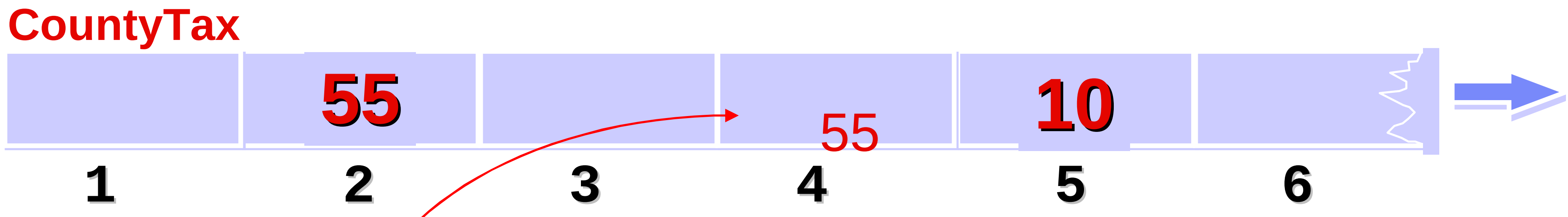


```
MOVE 10 TO CountyTax(5)
ADD TaxPaid TO CountyTax(CountyNum)
ADD TaxPaid TO CountyTax(CountyNum + 2)
```

Tables and the PERFORM ... VARYING

Tables/Arrays

A table is a contiguous sequence of memory locations called elements, which all have the same name, and are uniquely identified by that name and by their position in the sequence.



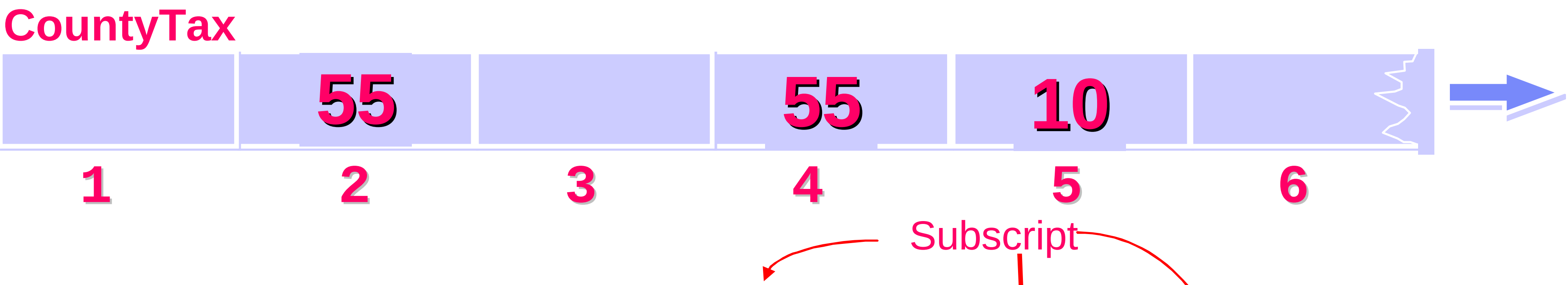
```
MOVE 10 TO CountyTax(5)
ADD TaxPaid TO CountyTax(CountyNum)
ADD TaxPaid TO CountyTax(CountyNum + 2)
```

55 2

Tables and the PERFORM ... VARYING

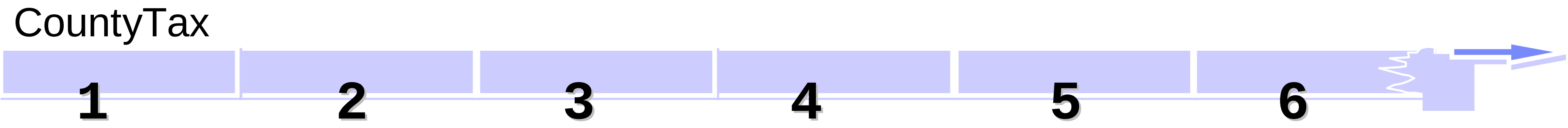
Tables/Arrays

A table is a contiguous sequence of memory locations called elements, which all have the same name, and are uniquely identified by that name and by their position in the sequence. The position index is called a subscript.



```
MOVE 10 TO CountyTax(5)
ADD TaxPaid TO CountyTax(CountyNum)
ADD TaxPaid TO CountyTax(CountyNum + 2)
```

Tables and the PERFORM ... VARYING

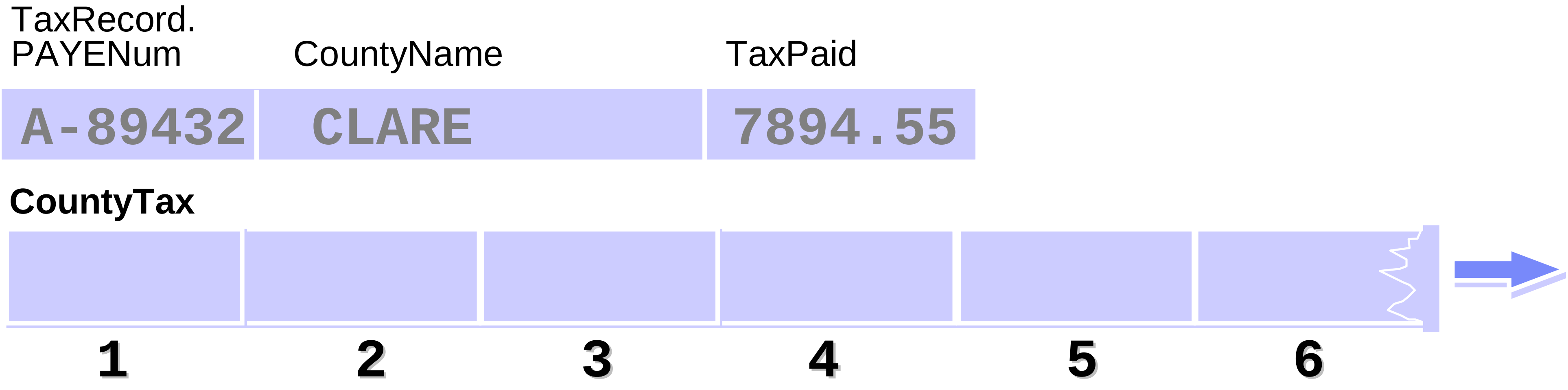


```
PROCEDURE DIVISION.
Begin.
   OPEN INPUT TaxFile
   READ TaxFile
      AT END SET EndOfTaxFile TO TRUE
   END-READ

   PERFORM UNTIL EndOfTaxFile
      ADD TaxPaid TO CountyTax(CountyNum)
      READ TaxFile
         AT END SET EndOfTaxFile TO TRUE
      END-READ
   END-PERFORM.

   PERFORM VARYING Idx FROM 1 BY 1
           UNTIL Idx GREATER THAN 26
      DISPLAY "County ", CountyNum
              " tax total is " CountyTax(Idx)
   END-PERFORM

   CLOSE TaxFile
   STOP RUN.
```

Subscript

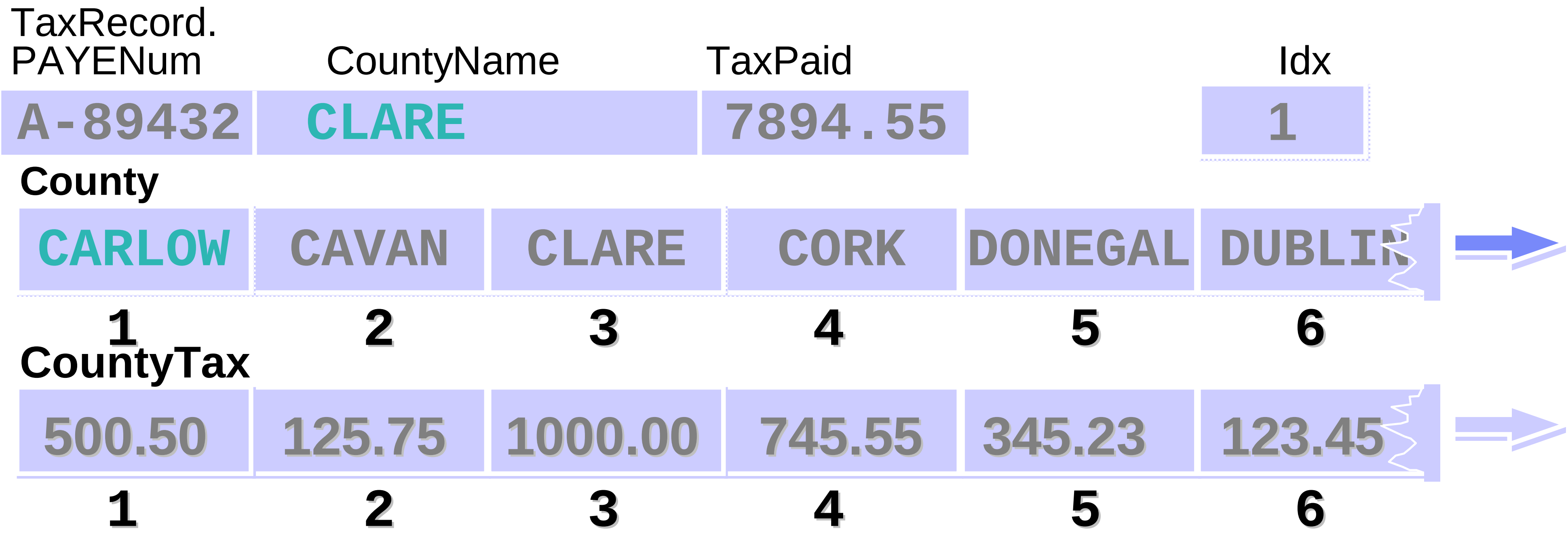
9 Statements

Tables and the PERFORM ... VARYING



```
IF CountyName = "CARLOW"
   ADD TaxPaid TO CountyTax(1)
END-IF
IF CountyName = "CAVAN"
   ADD TaxPaid TO CountyTax(2)
END-IF
 :     :     :      :      :
 :     :     :      :      :
          24 TIMES
```

Tables and the PERFORM ... VARYING



TaxRecord.

PAYENum	CountyName	TaxPaid	Idx
A-89432	CLARE	7894.55	1

County

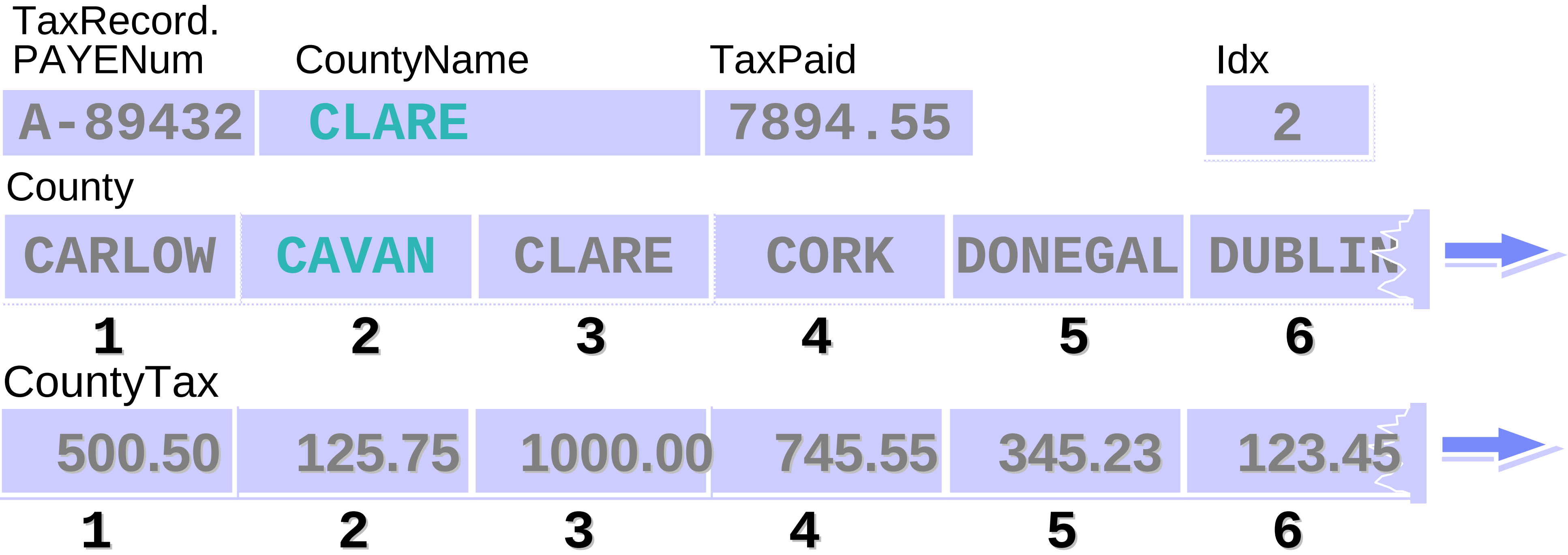
CARLOW	CAVAN	CLARE	CORK	DONEGAL	DUBLIN
1	2	3	4	5	6

CountyTax

500.50	125.75	1000.00	745.55	345.23	123.45
1	2	3	4	5	6

```
PERFORM VARYING Idx FROM 1 BY 1
     UNTIL County(Idx) = CountyName
END-PERFORM
ADD TaxPaid TO CountyTax(Idx)
```

Tables and the PERFORM ... VARYING



TaxRecord.

PAYENum	CountyName	TaxPaid
A-89432	CLARE	7894.55

Idx
2

County

1	2	3	4	5	6
CARLOW	CAVAN	CLARE	CORK	DONEGAL	DUBLIN

CountyTax

1	2	3	4	5	6
500.50	125.75	1000.00	745.55	345.23	123.45

```
PERFORM VARYING Idx FROM 1 BY 1
     UNTIL County(Idx) = CountyName
END-PERFORM
ADD TaxPaid TO CountyTax(Idx)
```

Tables and the PERFORM ... VARYING

TaxRecord.

PAYENum	CountyName	TaxPaid	Idx
A-89432	CLARE	7894.55	3

County

1	2	3	4	5	6
CARLOW	CAVAN	CLARE	CORK	DONEGAL	DUBLIN

CountyTax

1	2	3	4	5	6
500.50	125.75	1000.00	745.55	345.23	123.45

```
PERFORM VARYING Idx FROM 1 BY 1
        UNTIL County(Idx) = CountyName
END-PERFORM
ADD TaxPaid TO CountyTax(Idx)
```

Tables and the PERFORM ... VARYING

TaxRecord.

PAYENum	CountyName	TaxPaid
A-89432	CLARE	7894.55

Idx
3

County

CARLOW	CAVAN	CLARE	CORK	DONEGAL	DUBLIN
1	2	3	4	5	6

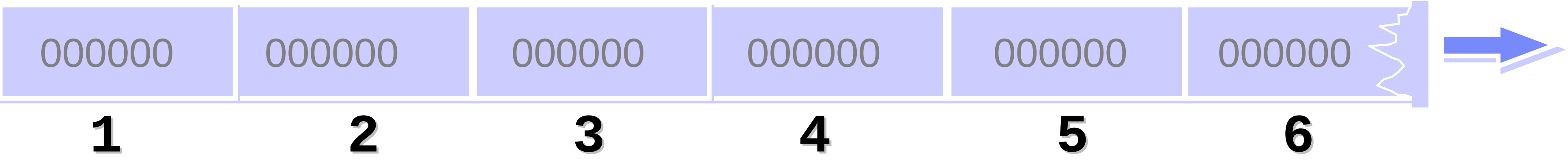
CountyTax

500.50	125.75	8894.55	745.55	345.23	123.45
1	2	3	4	5	6

```
PERFORM VARYING Idx FROM 1 BY 1
     UNTIL County(Idx) = CountyName
END-PERFORM
ADD TaxPaid TO CountyTax(Idx)
```

Tables and the PERFORM ... VARYING

Declaring Tables



000000	000000	000000	000000	000000	000000
1	2	3	4	5	6

```
01  TaxTotals.
     02  CountyTax   PIC 9(10)V99
                     OCCURS 26 TIMES.
```

or

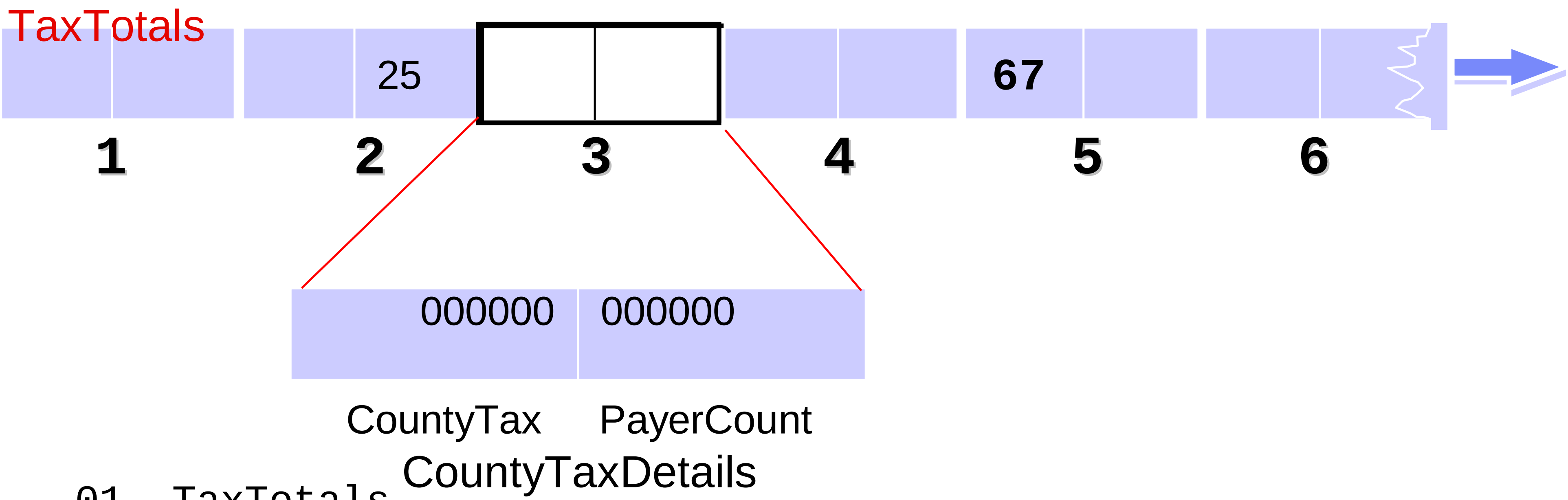
```
    02  CountyTax   OCCURS 26 TIMES
                     PIC 9(10)V99.
```

e.g.

```
MOVE ZEROS TO TaxTotals.

MOVE 20 TO CountyTax(5).
```

Tables and the PERFORM ... VARYING

Group Items as Elements



```
01  TaxTotals.
     02  CountyTaxDetails OCCURS 26 TIMES.
         03  CountyTax     PIC 9(10)V99.
         03  PayerCount    PIC 9(7).

e.g.  MOVE 25 TO PayerCount(2).
       MOVE 67 TO  CountyTax(5).
       MOVE ZEROS TO CountyTaxDetails(3).
```

Tables and the PERFORM ... VARYING

PERFORM..VARYING Syntax

$$\underline{\text{PERFORM}}\ \left[\text{1stProc}\ \left[\left\{\begin{matrix}\underline{\text{THRU}}\\ \underline{\text{THROUGH}}\end{matrix}\right\}\text{EndProc}\right]\right]\left[\text{WITH}\ \ \underline{\text{TEST}}\left\{\begin{matrix}\underline{\text{BEFORE}}\\ \underline{\text{AFTER}}\end{matrix}\right\}\right]$$

$$\underline{\text{VARYING}}\left\{\begin{matrix}\text{Identifer1}\\ \text{IndexName1}\end{matrix}\right\}\underline{\text{FROM}}\left\{\begin{matrix}\textit{Identifier}\ \ 2\\ \textit{IndexName}\ \ 2\\ \textit{Literal}\end{matrix}\right\}$$

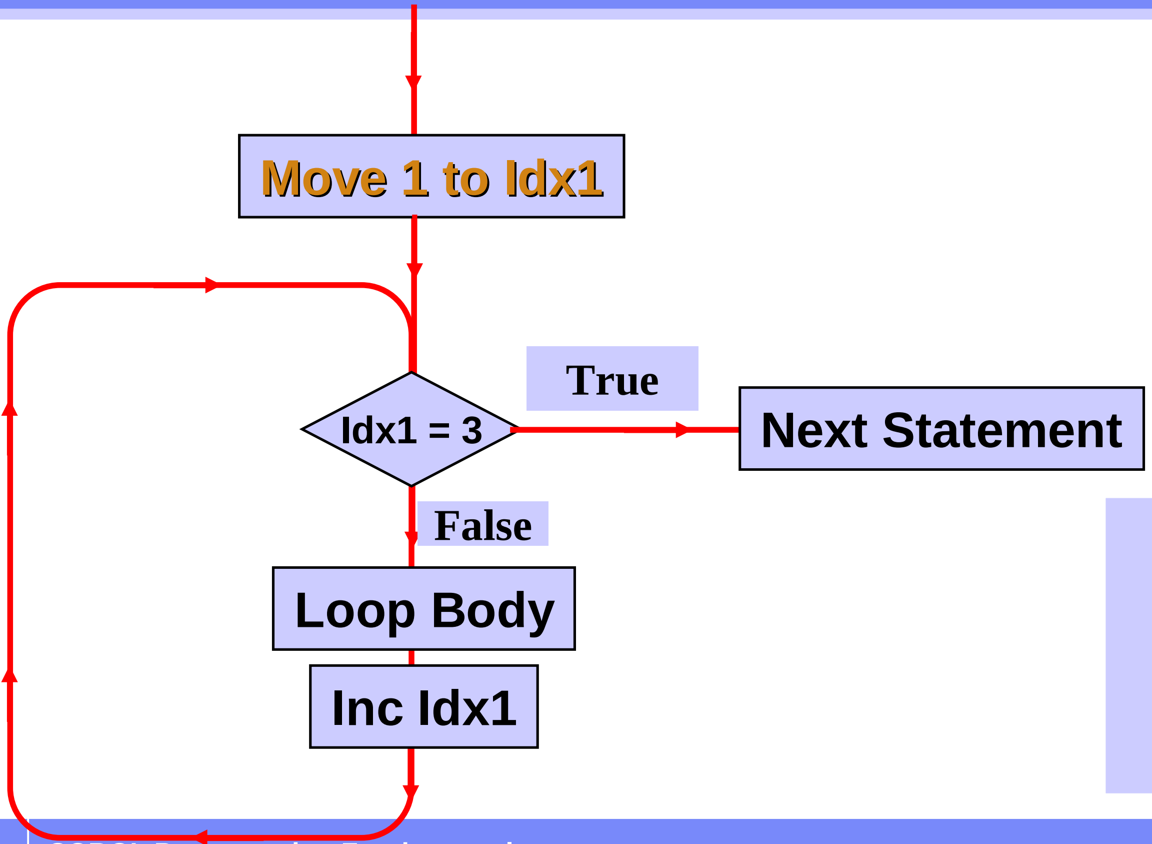
$$\underline{\text{BY}}\left\{\begin{matrix}\text{Identifier}\ \ 3\\ \text{Literal}\end{matrix}\right\}\underline{\text{UNTIL}}\ \ \text{Condition1}$$

$$\left[\begin{matrix}\underline{\text{AFTER}}\left\{\begin{matrix}\text{Identifier}\ \ 4\\ \text{IndexName3}\end{matrix}\right\}\underline{\text{FROM}}\left\{\begin{matrix}\textit{Identifier}\ \ 5\\ \textit{IndexName}\ \ 4\\ \textit{Literal}\end{matrix}\right\}\\ \underline{\text{BY}}\left\{\begin{matrix}\text{Identifier}\ \ 6\\ \text{Literal}\end{matrix}\right\}\underline{\text{UNTIL}}\ \ \text{Condition2}\end{matrix}\right]\mathbf{K}$$

$$\left[\text{StatementB}\ \ \ \text{lock}\ \ \underline{\text{END}\ \ \text{- PERFORM}}\right]$$

Tables and the PERFORM ... VARYING

```
PERFORM VARYING Idx1 FROM 1 BY 1 UNTIL
        Idx1 EQUAL TO 3
     DISPLAY Idx1
END-PERFORM.
```

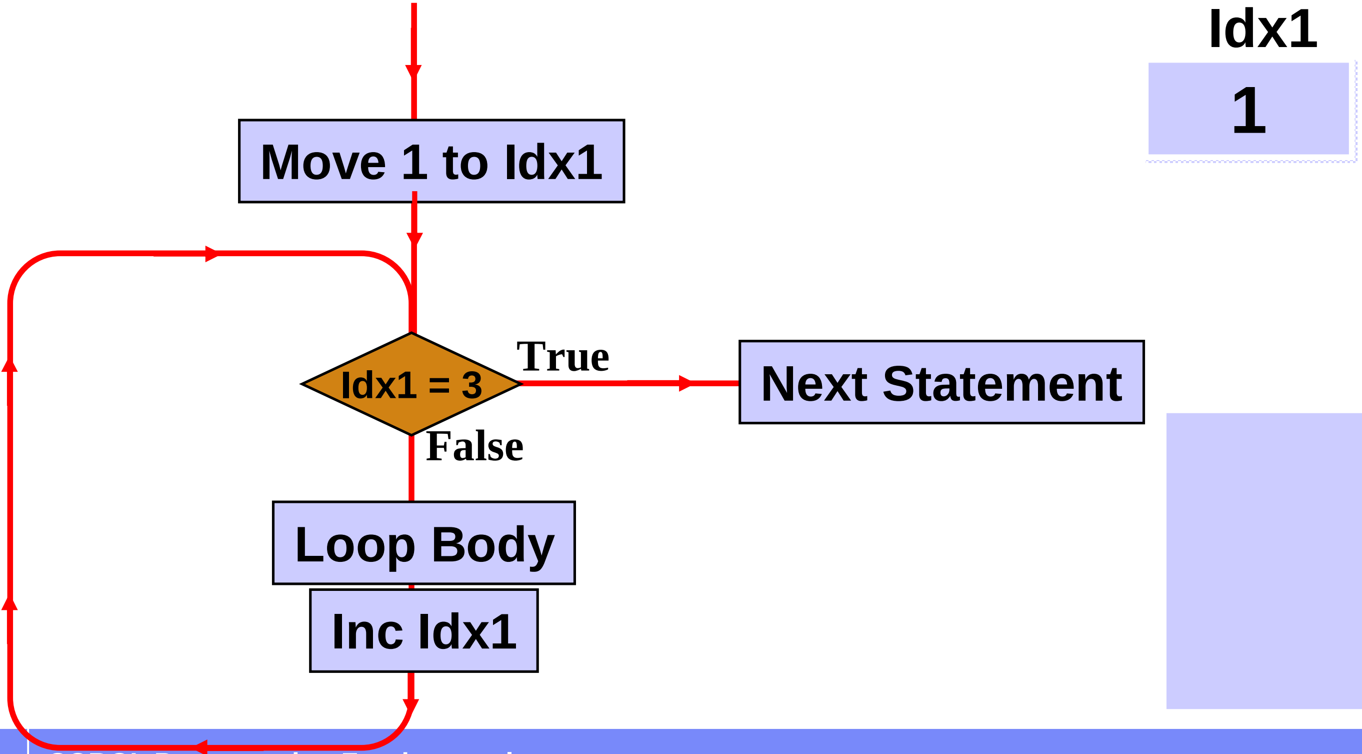


Idx1

1

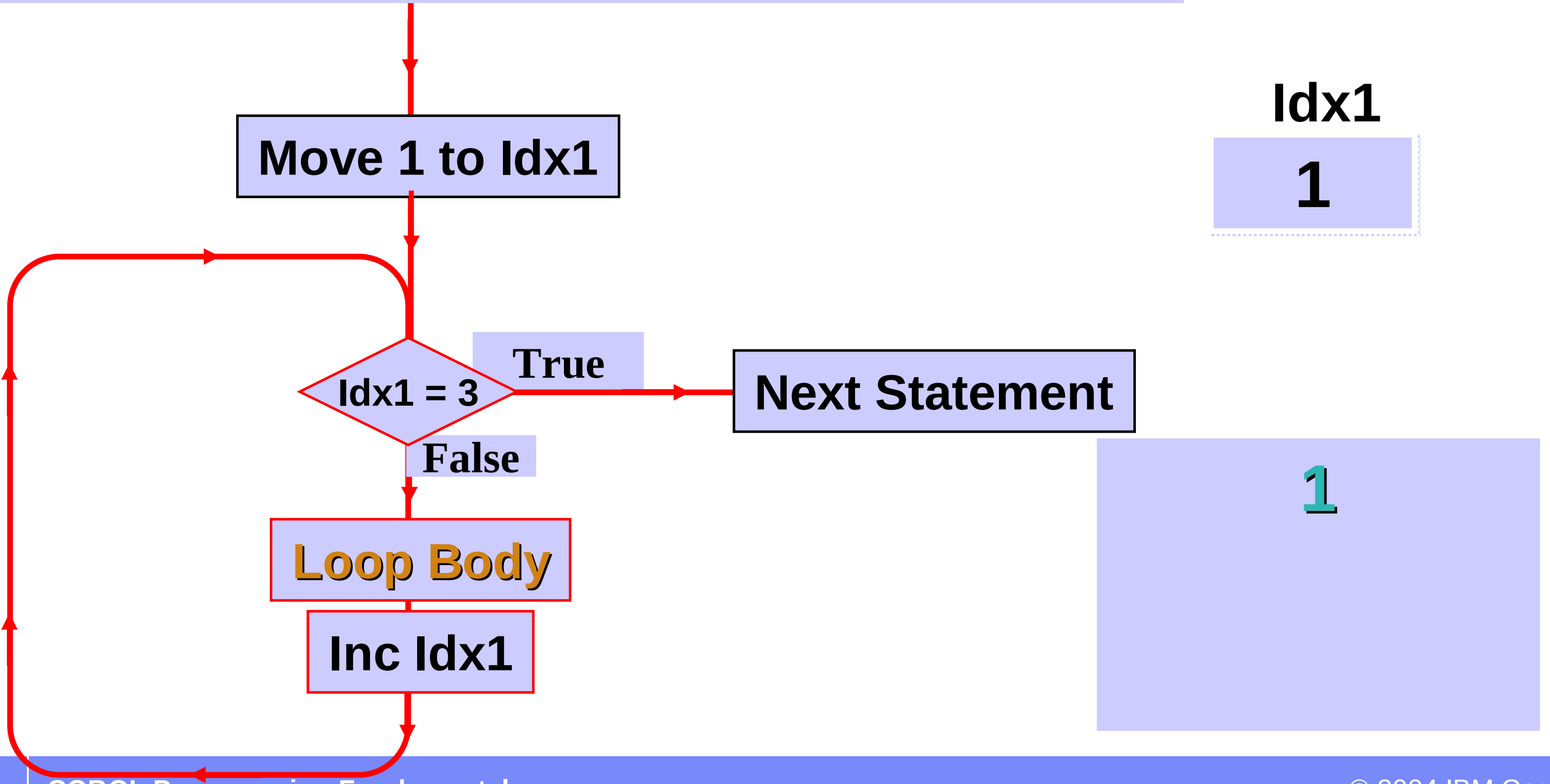
Tables and the PERFORM ... VARYING

```
PERFORM VARYING Idx1 FROM 1 BY 1 UNTIL
        Idx1 EQUAL TO 3
      DISPLAY Idx1
END-PERFORM.
```



Tables and the PERFORM ... VARYING

```
PERFORM VARYING Idx1 FROM 1 BY 1 UNTIL
        Idx1 EQUAL TO 3
      DISPLAY Idx1
END-PERFORM.
```



Move 1 to Idx1

Idx1 = 3

True

False

Next Statement

Loop Body

Inc Idx1

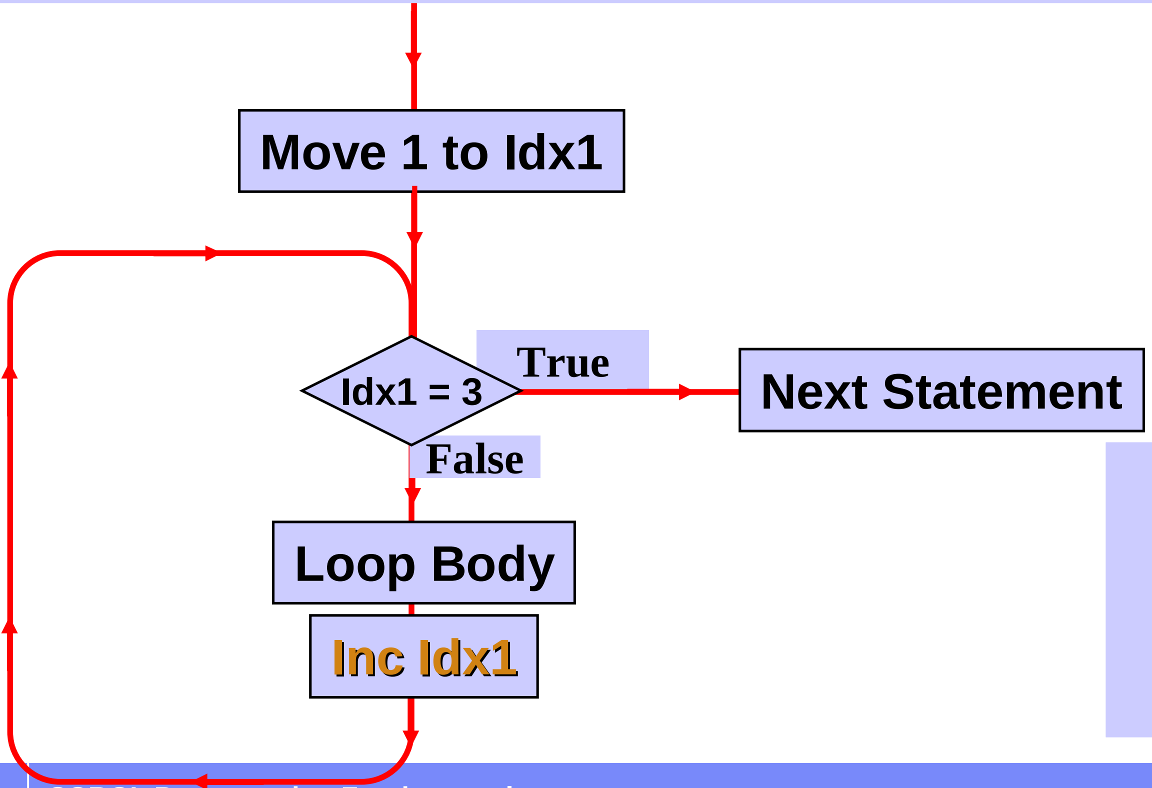
Idx1

1

1

Tables and the PERFORM ... VARYING

```
PERFORM VARYING Idx1 FROM 1 BY 1 UNTIL
        Idx1 EQUAL TO 3
      DISPLAY Idx1
END-PERFORM.
```



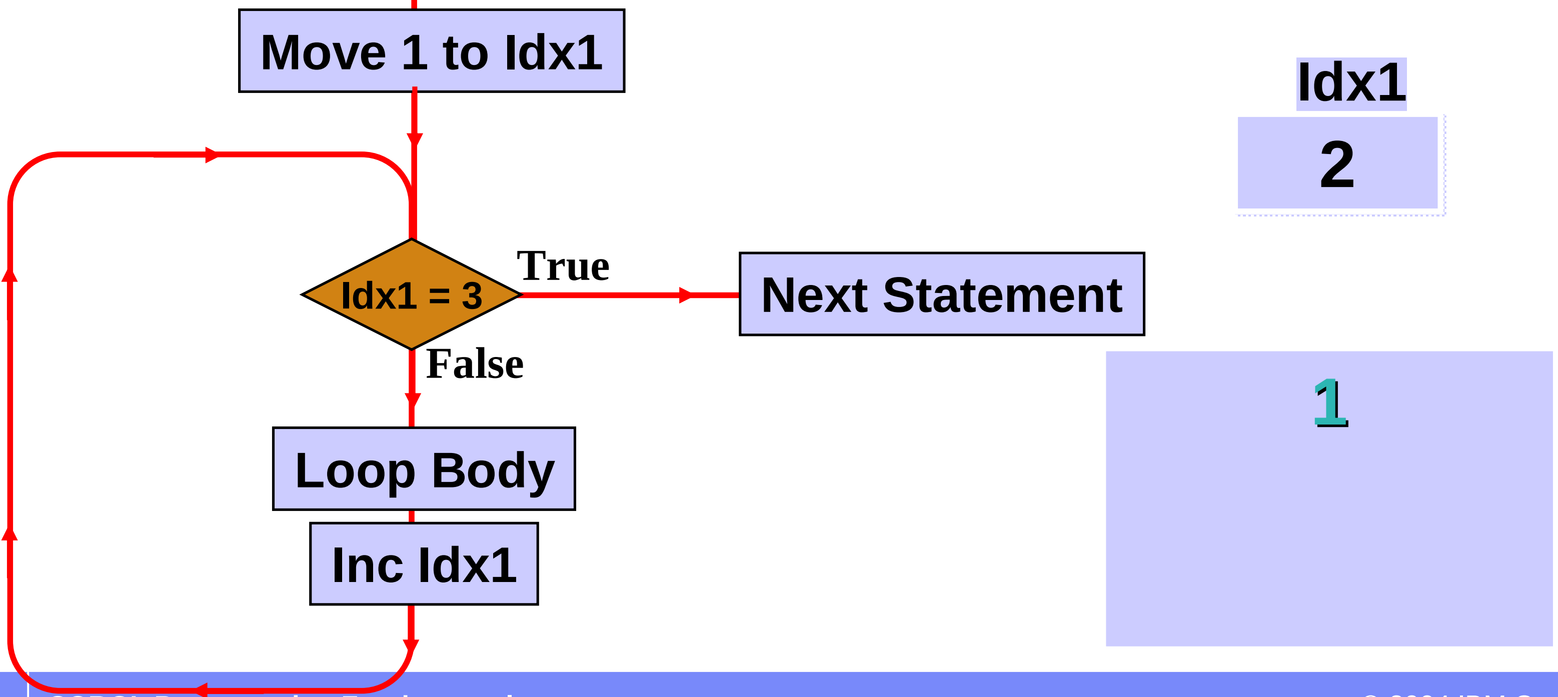
Idx1

2

1

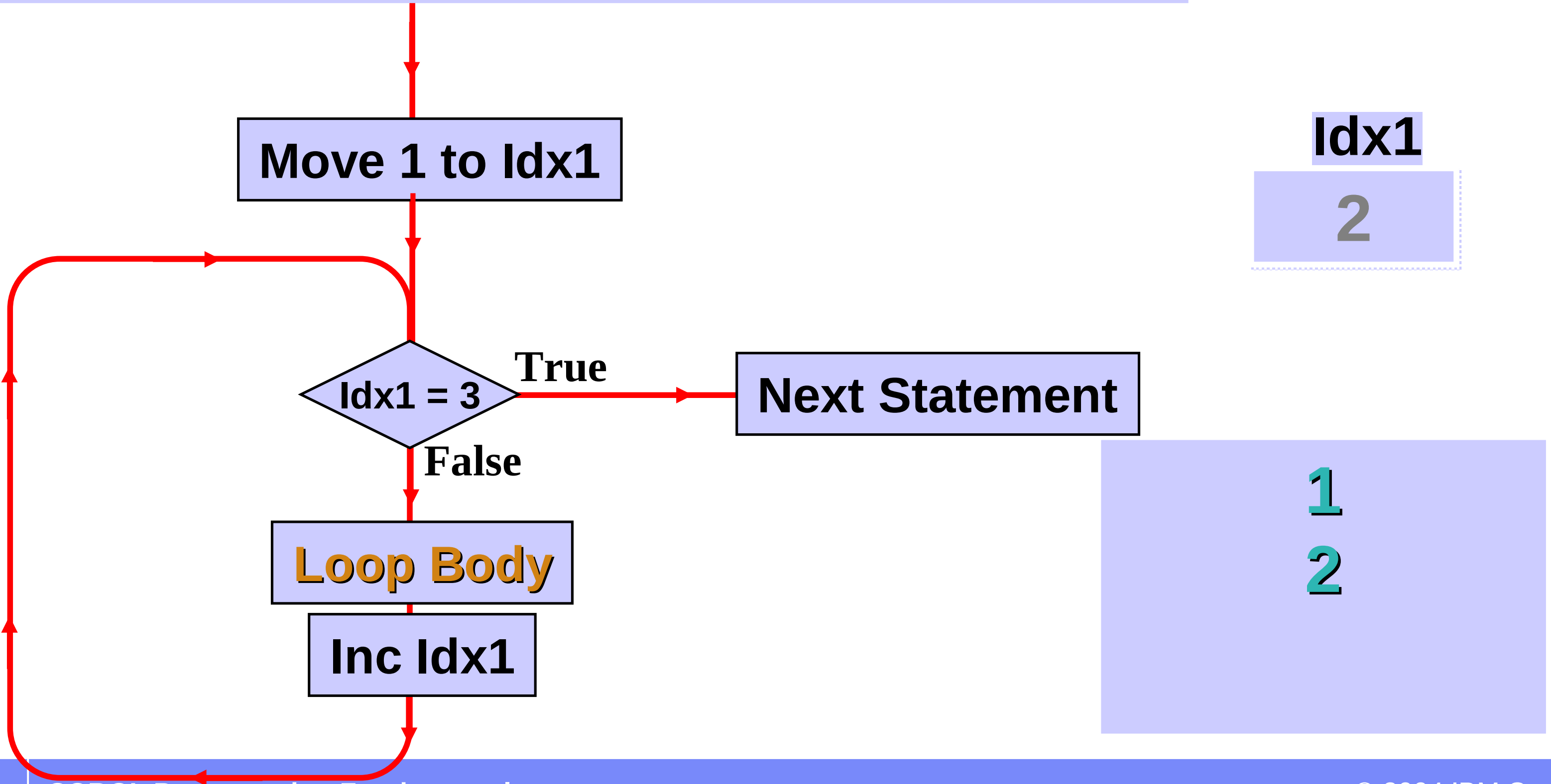
Tables and the PERFORM ... VARYING

```
PERFORM VARYING Idx1 FROM 1 BY 1 UNTIL
        Idx1 EQUAL TO 3
     DISPLAY Idx1
END-PERFORM.
```



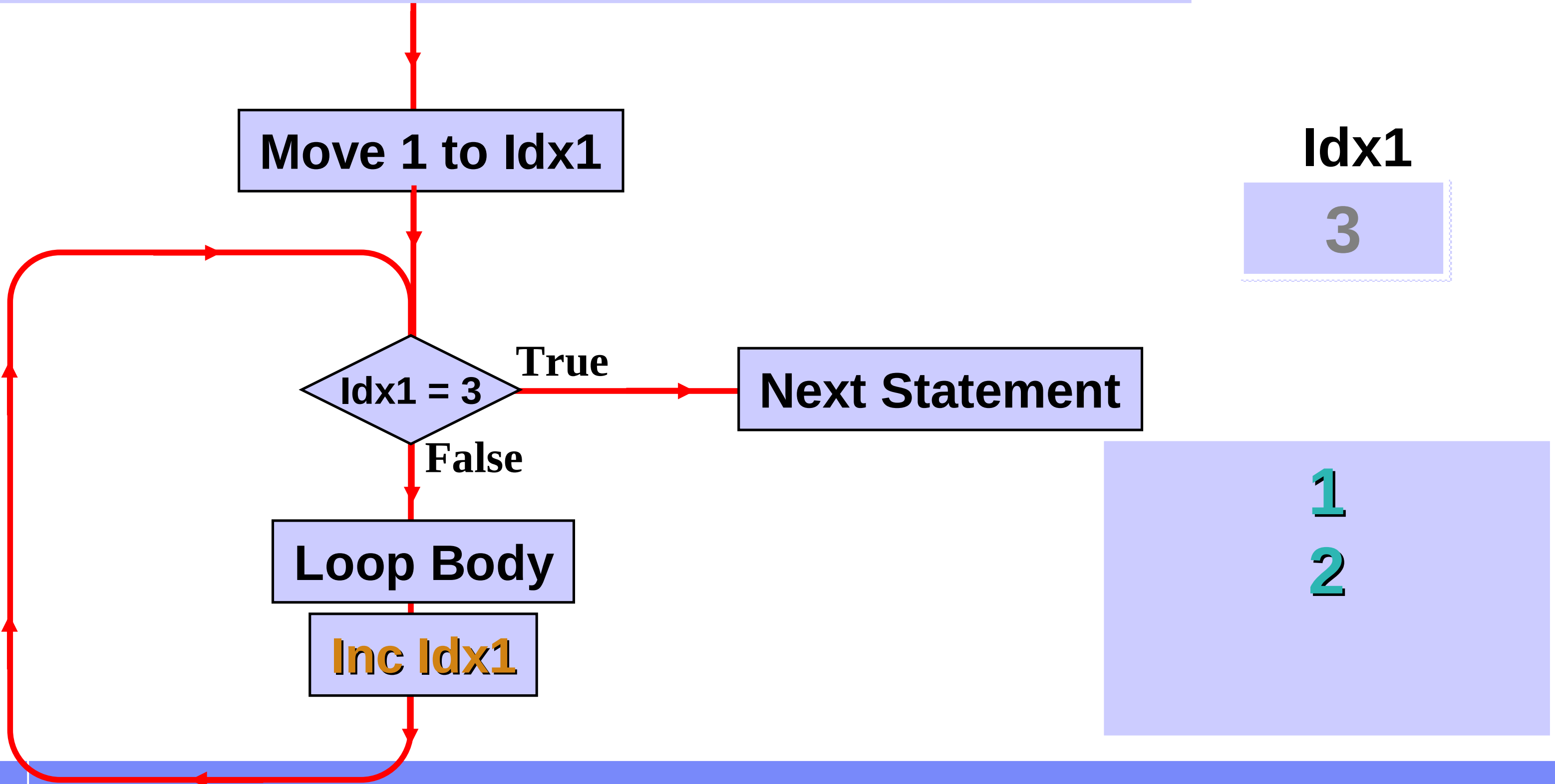
Tables and the PERFORM ... VARYING

```
PERFORM VARYING Idx1 FROM 1 BY 1 UNTIL
        Idx1 EQUAL TO 3
      DISPLAY Idx1
END-PERFORM.
```



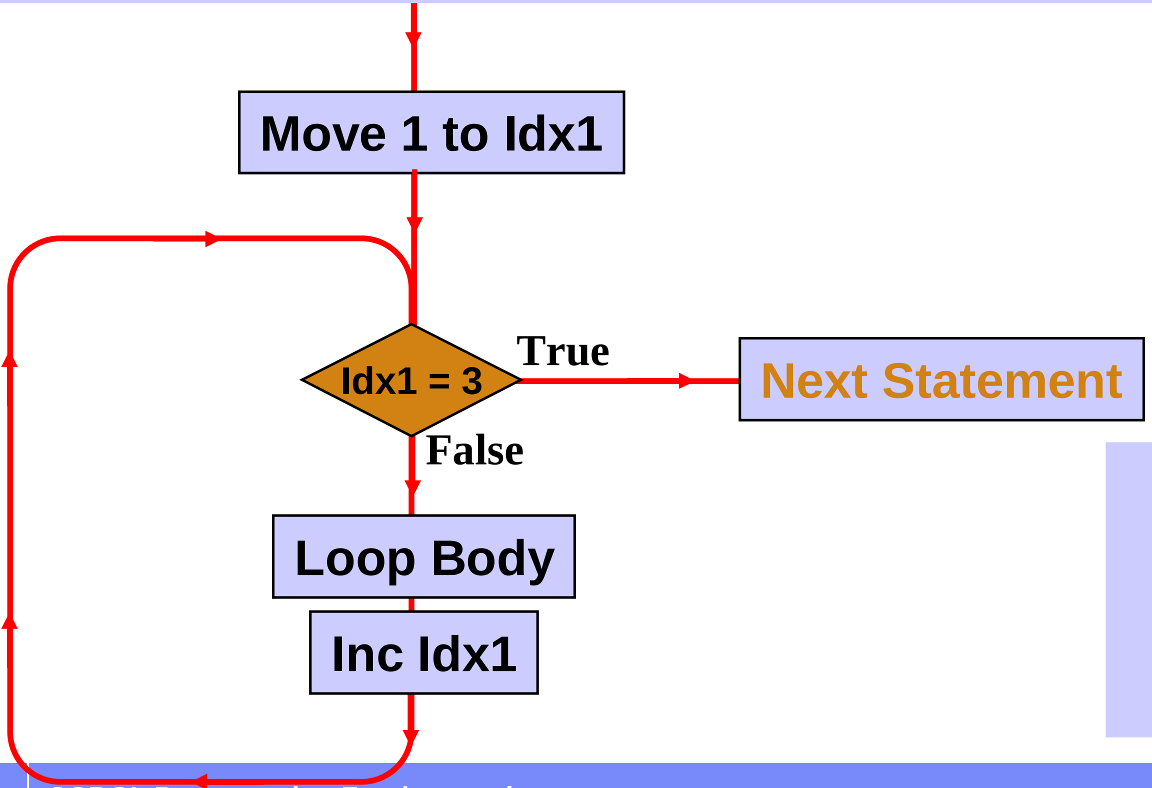
Tables and the PERFORM ... VARYING

```
PERFORM VARYING Idx1 FROM 1 BY 1 UNTIL
        Idx1 EQUAL TO 3
     DISPLAY Idx1
END-PERFORM.
```



Tables and the PERFORM ... VARYING

```
PERFORM VARYING Idx1 FROM 1 BY 1 UNTIL
        Idx1 EQUAL TO 3
     DISPLAY Idx1
END-PERFORM.
```



Idx1

3

1

2

Exit value = 3

Table of contents

Introduction to COBOL
COBOL Basics 1
COBOL Basics 2
Introduction to Sequential Files
Processing Sequential Files
Simple iteration with the PERFORM verb
Arithmetic and Edited Pictures
Conditions
Tables and the PERFORM ... VARYING
Designing Programs

Designing Programs

Overview

- Why we use COBOL.
- The problem of program maintenance.
- How Cobol programs should be written.
- Efficiency vs Clarity.
- Producing a good design.
- Introduction to design notations.
- Guidelines for writing Cobol programs.

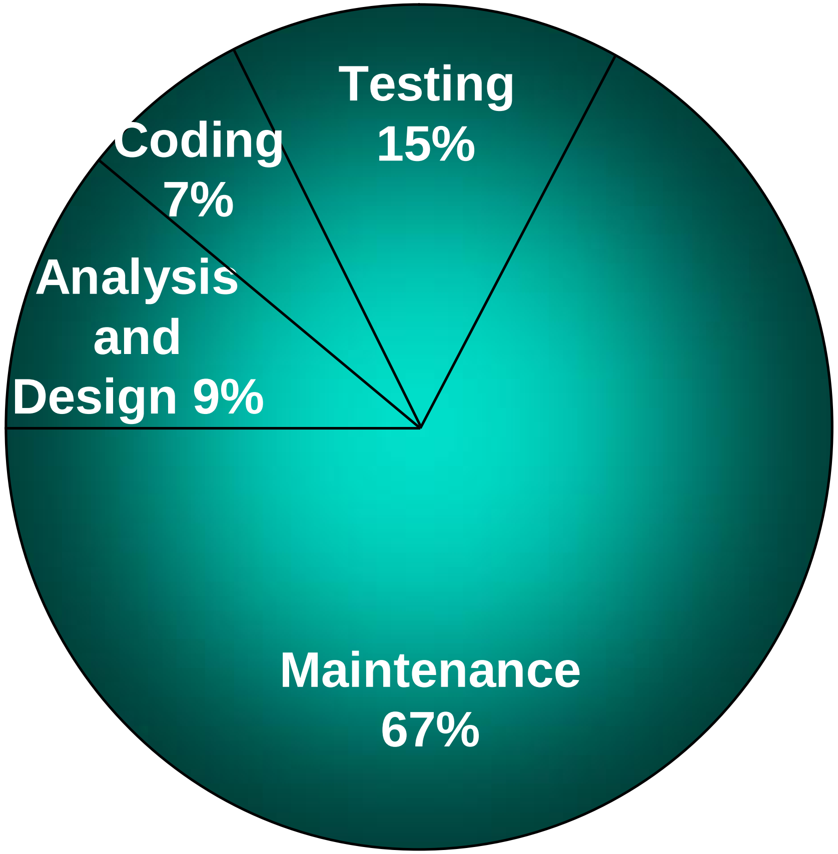
Designing Programs

COBOL

- COBOL is an acronym standing for Common Business Oriented Language.
- COBOL programs are (mostly) written for the vertical market.
- COBOL programs tend to be long lived.
- Because of this longevity ease of program maintenance is an important consideration.
- Why is program maintenance important?

Designing Programs

Cost of a system over its entire life



Zelkowitz
ACM 1978
p202

Maintenance Costs are only as low as this because many systems become so unmaintainable early in their lives that they have to be SCRAPPED !!
:- B. Boehm

Designing Programs

Program Maintenance

- Program maintenance is an umbrella term that covers;

  1. Changing the program to fix bugs that appear in the system.
  2. Changing the program to reflect changes in the environment.
  3. Changing the program to reflect changes in the users perception of the requirements.
  4. Changing the program to include extensions to the user requirements (i.e. new requirements).

- What do these all have in common?

CHANGING THE PROGRAM.

Designing Programs

How should write your programs?

- You should write your programs with the expectation that they will have to be changed.

- This means that you should;
  - write programs that are easy to read.
  - write programs that are easy to understand.
  - write programs that are easy to change.

- You should write your programs as you would like them written if you had to maintain them.

Designing Programs

Efficiency vs Clarity

- Many programmers are overly concerned about making their programs as efficient as possible (in terms of the speed of execution or the amount of memory used).
- But the proper concern of a programmer, and particularly a COBOL programmer, is not this kind of efficiency, it is clarity.
- As a rule 70% of the work of the program will be done in 10% of the code.
- It is therefore a pointless exercise to try to optimize the whole program, especially if this has to be done at the expense of clarity.
- Write your program as clearly as possible and then, if its too slow, identify the 10% of the code where the work is being done and optimize it.

Designing Programs

When shouldn’t we design our programs?

- We shouldn’t design our programs, when we want to create programs that do not work.
- We shouldn’t design when we want to produce programs that do not solve the problem specified.
- When we want to create programs that;

  get the wrong inputs,

  or perform the wrong transformations on them

  or produce the wrong outputs

  then we shouldn’t bother to design our programs.
- But if we want to create programs that work, we cannot avoid design.
- The only question is;

  will it be a good design or a bad design

Designing Programs

Producing a Good Design

- The first step to producing a good design is to design consciously.
- Subconscious design means that design is done while constructing the program. This never leads to good results.
- Conscious design starts by separating the design task from the task of program construction.
- Design, consists of devising a solution to the problem specified.
- Construction, consists of taking the design and encoding the solution using a particular programming language.

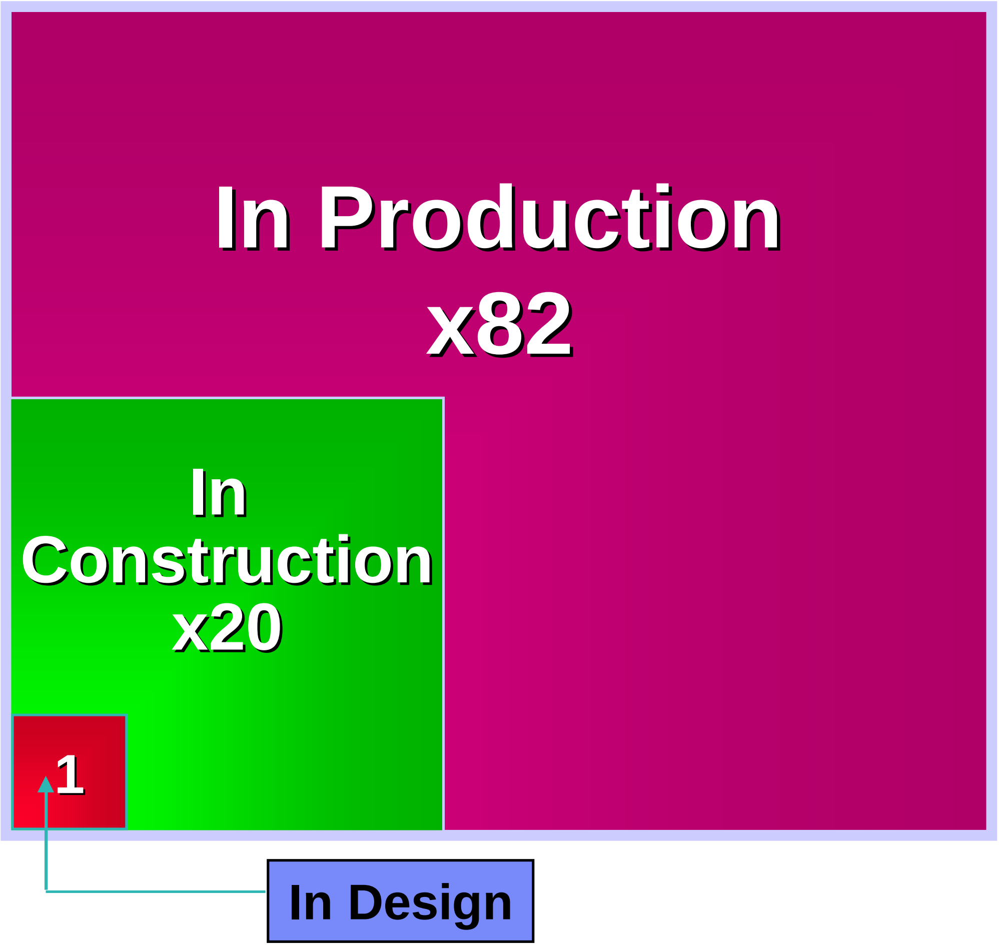
Designing Programs

Why separate design from construction?

- Separating program design from program construction makes both tasks easier.
- Designing before construction, allows us to plan our solution to the problem - instead of stumbling from one incorrect solution to another.
- Good program structure results from planing and design. It is unlikely to result from ad hoc tinkering.
- Designing helps us to get an overview of the problem and to think about the solution without getting bogged down by the details of construction.
- It helps us to iron out problems with the specification and to discover any bugs in our solution before we commit it to code (see next slide).
- Design allows us to develop portable solutions

Designing Programs

Relative cost of fixing a bug



Figures from IBM in Santa Clara.

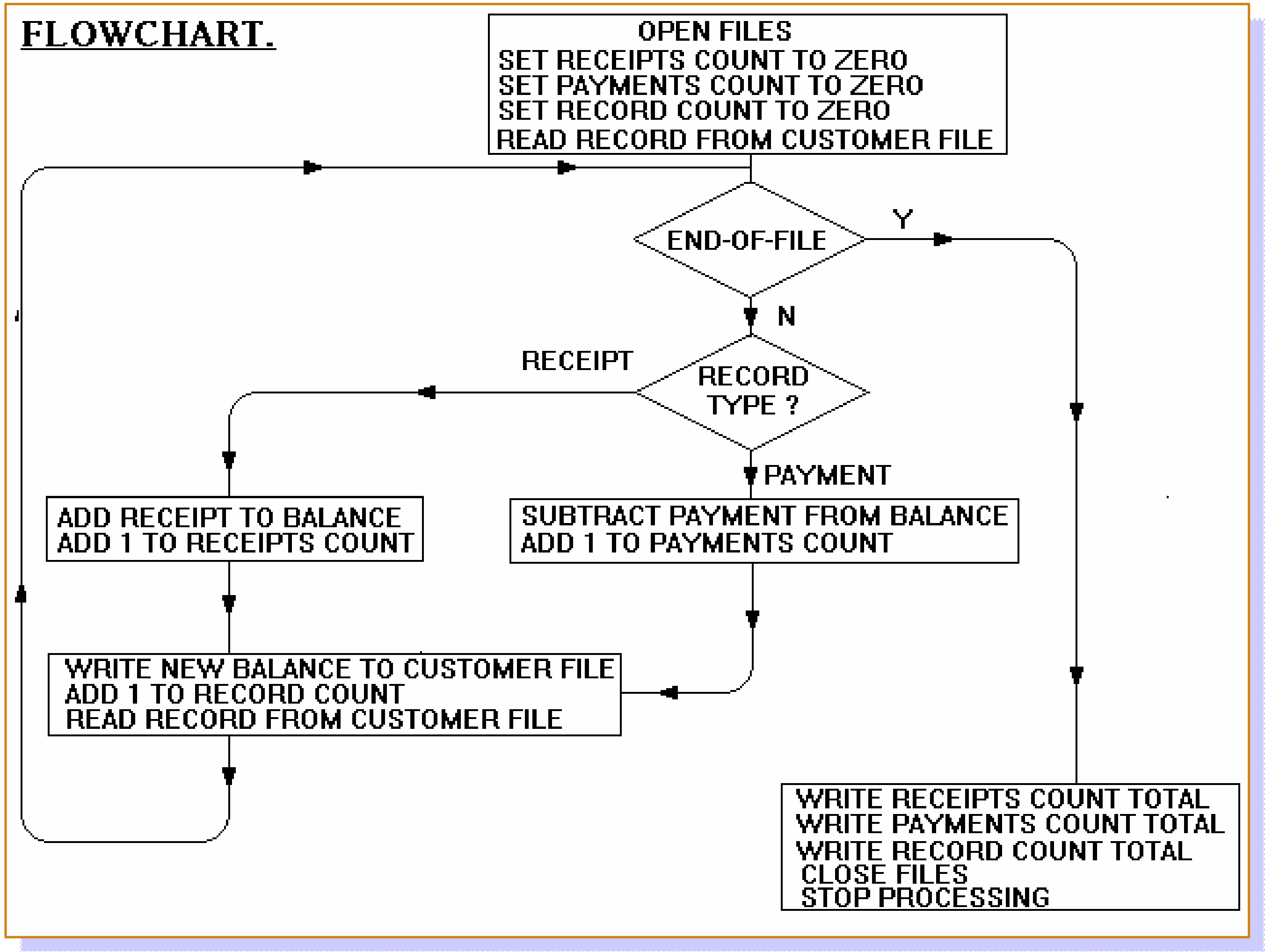
Designing Programs

Design Notations

- A number of notations have been suggested to assist the programmer with the task of program design.
- Some notations are textual and others graphical.
- Some notations can actually assist in the design process.
- While others merely articulate the design.

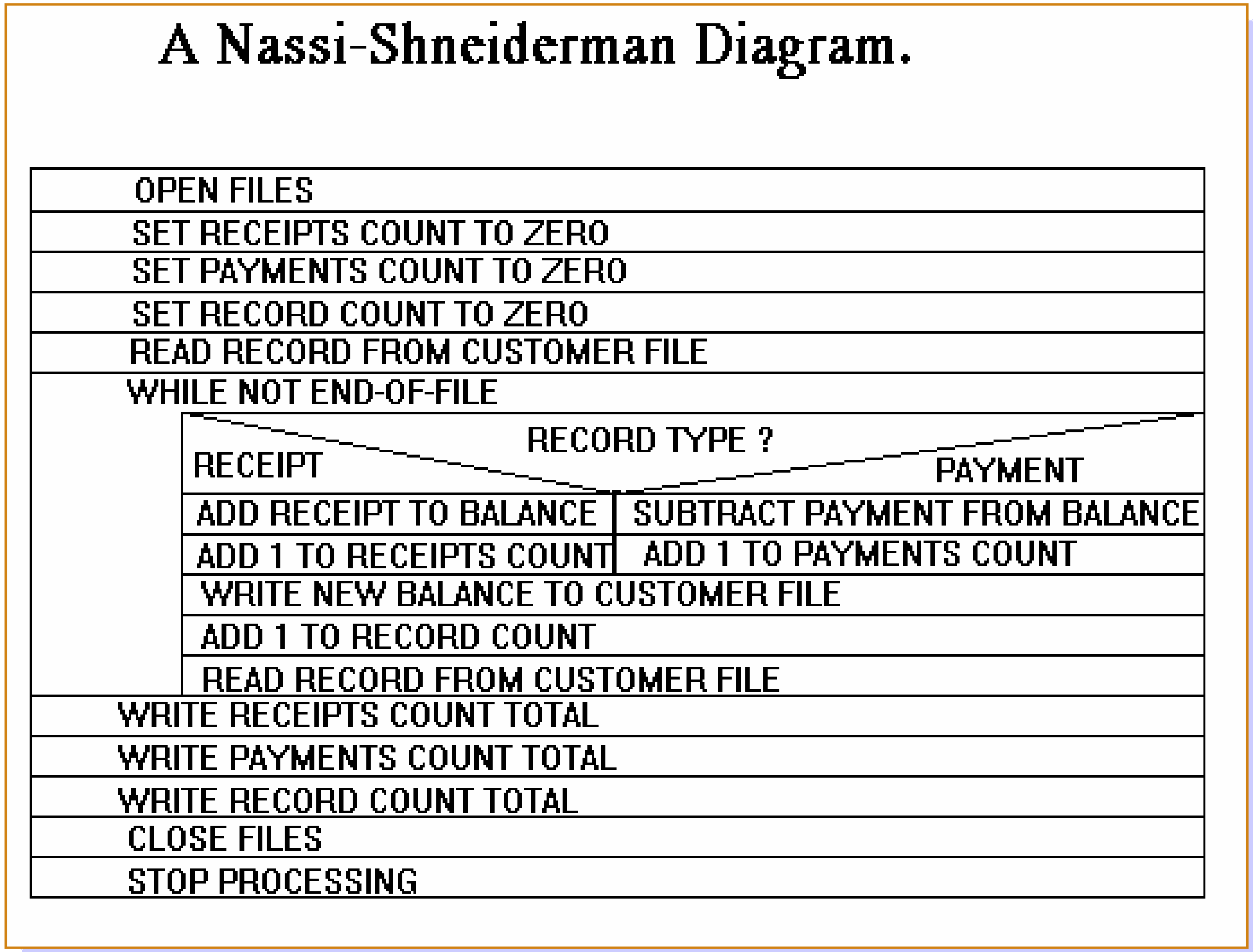
Designing Programs

Flowcharts as design tools


FLOWCHART.
OPEN FILES
SET RECEIPTS COUNT TO ZERO
SET PAYMENTS COUNT TO ZERO
SET RECORD COUNT TO ZERO
READ RECORD FROM CUSTOMER FILE
END-OF-FILE
Y
N
RECEIPT
RECORD TYPE ?
PAYMENT
ADD RECEIPT TO BALANCE
ADD 1 TO RECEIPTS COUNT
SUBTRACT PAYMENT FROM BALANCE
ADD 1 TO PAYMENTS COUNT
WRITE NEW BALANCE TO CUSTOMER FILE
ADD 1 TO RECORD COUNT
READ RECORD FROM CUSTOMER FILE
WRITE RECEIPTS COUNT TOTAL
WRITE PAYMENTS COUNT TOTAL
WRITE RECORD COUNT TOTAL
CLOSE FILES
STOP PROCESSING

Designing Programs

Structured Flowcharts as design tools



A Nassi-Shneiderman Diagram.

```
OPEN FILES
SET RECEIPTS COUNT TO ZERO
SET PAYMENTS COUNT TO ZERO
SET RECORD COUNT TO ZERO
READ RECORD FROM CUSTOMER FILE
WHILE NOT END-OF-FILE
    RECORD TYPE ?
    RECEIPT
        ADD RECEIPT TO BALANCE
        ADD 1 TO RECEIPTS COUNT
    PAYMENT
        SUBTRACT PAYMENT FROM BALANCE
        ADD 1 TO PAYMENTS COUNT
    WRITE NEW BALANCE TO CUSTOMER FILE
    ADD 1 TO RECORD COUNT
    READ RECORD FROM CUSTOMER FILE
WRITE RECEIPTS COUNT TOTAL
WRITE PAYMENTS COUNT TOTAL
WRITE RECORD COUNT TOTAL
CLOSE FILES
STOP PROCESSING
```

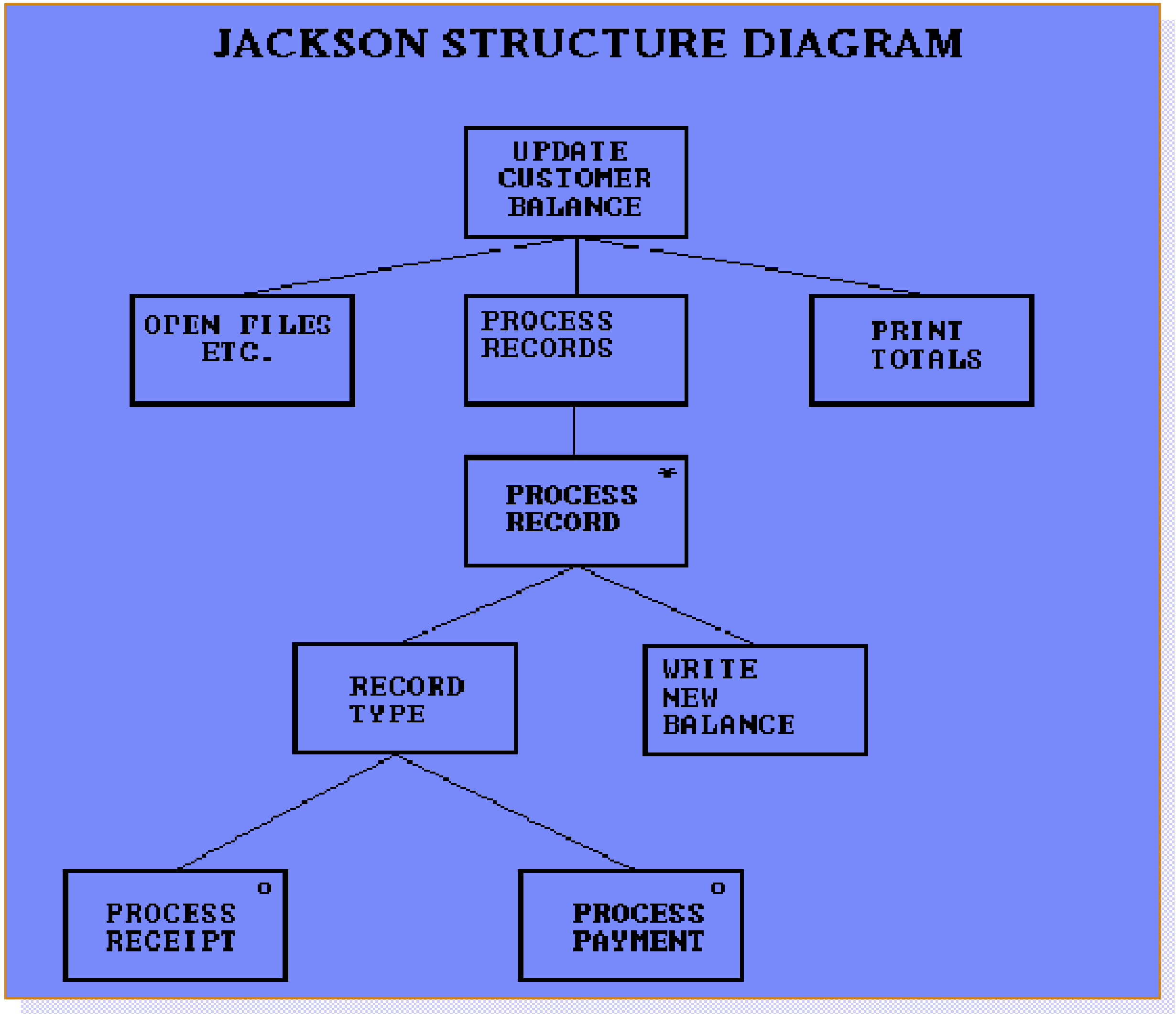
Designing Programs

Structured English

```
For each transaction record do the following
    IF the record is a receipt then
        add 1 to the ReceiptsCount
        add the Amount to the Balance
    otherwise
        add 1 to the PaymentsCount
        subtract the Amount from the Balance
    EndIF
    add 1 to the RecordCount
    Write the Balance to the CustomerFile

When the file has been processed
    Output    the ReceiptsCount
                  the PaymentsCount
                  and the RecordCount
```

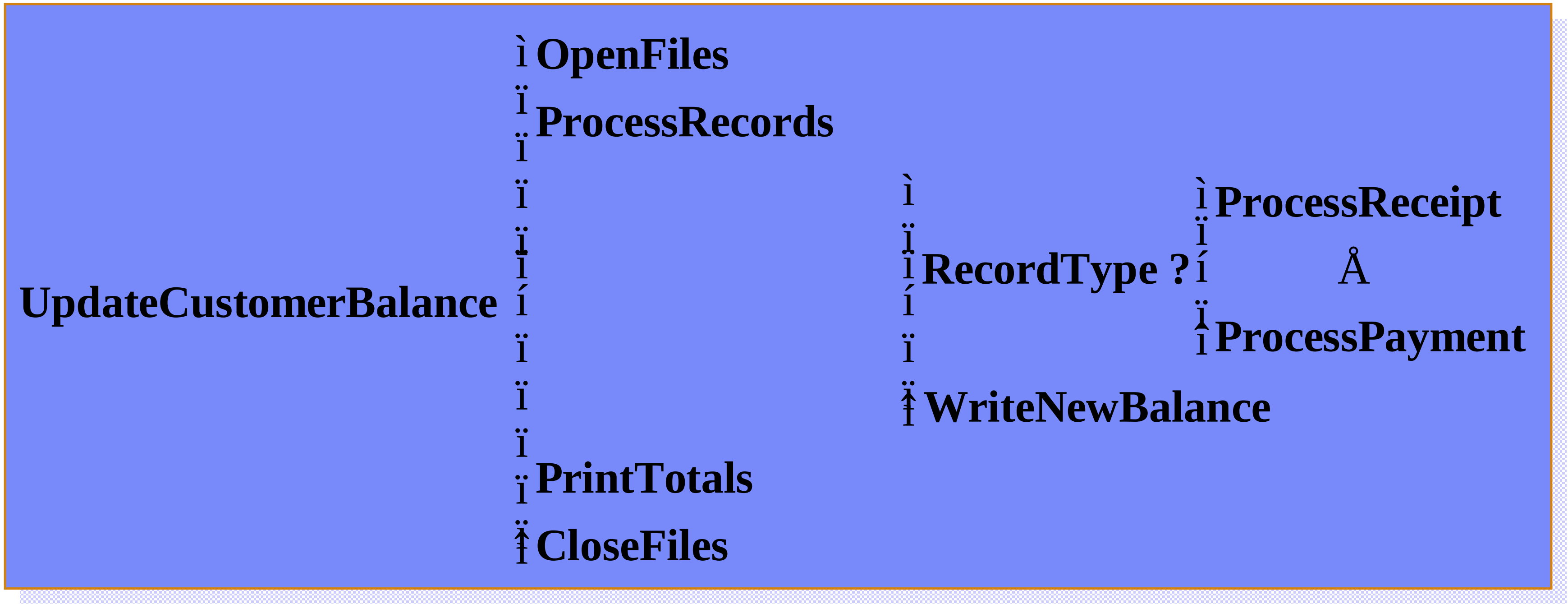
Designing Programs

The Jackson Method



Designing Programs

Warnier-Orr Diagrams



UpdateCustomerBalance

OpenFiles

ProcessRecords

RecordType ?

ProcessReceipt

Å

ProcessPayment

WriteNewBalance

PrintTotals

CloseFiles

Any Existing Process Could Be Improved!

Thank you very much!