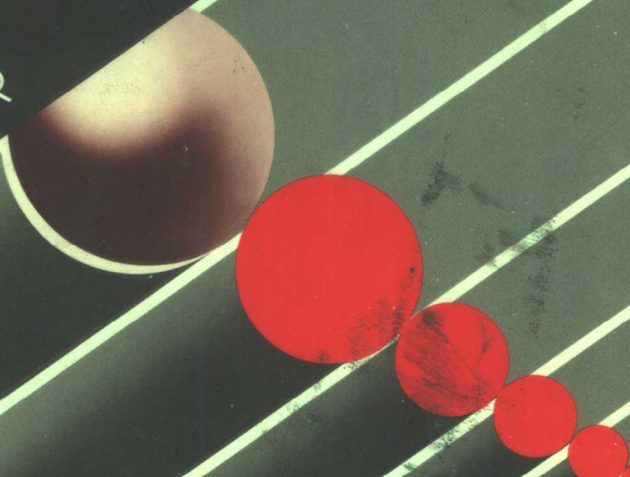


A. S. PHILIPPAKIS

A D V A N C E D  
**COBOL**



LEONARD J. KAZMIER

# ADVANCED COBOL

**A. S. Philippakis**

Arizona State University

**Leonard J. Kazmier**

Arizona State University

**McGraw-Hill Book Company**

New York St. Louis San Francisco Auckland  
Bogotá Hamburg Johannesburg London  
Madrid Mexico Montreal New Delhi Panama  
Paris São Paulo Singapore Sydney Tokyo Toronto

*To our wives  
Patricia and Lorraine*

**Library of Congress Cataloging in Publication Data**

Philippakis, Andreas S.  
Advanced COBOL.

Includes index.

1. COBOL (Computer program language)

I. Kazmier, Leonard J. II. Title.

QA76.73.C25P48 001.64'24 81-17126

ISBN 0-07-049806-7

AACR2

**ADVANCED COBOL**

Copyright © 1982 by McGraw-Hill, Inc. All rights reserved.

Printed in the United States of America. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the prior written permission of the publisher.

567890 DODO 8987654

**ISBN 0-07-049806-7**

This book was set in Optima by Cobb/Dunlop Publisher Services Incorporated. The editor was James E. Vastyan; the production supervisor was Leroy A. Young. The cover was designed by Jerry Wilke.  
R. R. Donnelley & Sons Company was printer and binder.

See Acknowledgment on page xi. Copyrights included on this page by reference.

# PREFACE

This book has been designed to provide material beyond an introduction to the COBOL language, and to serve as a reference for practicing professional programmers.

The book has been developed in response to the increased role of COBOL in college and university curricula. In most college-level programs COBOL is included in a two- and often three-semester sequence of courses. Yet, there is a very limited choice of text materials to support the second or third semester in such course sequences.

Although the title and contents of this book reflect an intent to support a more advanced study of COBOL, we have recognized the reality that in many cases the introductory course coverage or the student retention of the materials in that course may be minimal. For that reason Chapters 4 through 7 include a complete review of the language and can serve as either a review or an introduction, as needed.

The book devotes an extensive amount of coverage to new concepts in program design that go far beyond the well established ideas of structured programming, although the latter topic is also presented and applied throughout the text. The first three chapters deal with the new subjects of program *structure*, *cohesion*, and *design*. The reader will find that these chapters provide a comprehensive coverage of these fundamental topics. Audiences with a firm foundation in COBOL will find the first three chapters a solid beginning toward advanced study of the language. Others may prefer to review Chapters 4-7 to refresh their understanding of the language before proceeding to the study of structure, cohesion, and design.

Chapters 8 and 9 treat more advanced language topics in the area of numeric and character data processing and in subprogram structure and use. Then, Chapters 10 and 11 cover the concepts and techniques of structured programming and program testing.

The remaining chapters, 12 through 17, treat a number of separate topics: report generation, table handling, sequential files, sorting and merging, indexed sequential files, and relative files. These chapters have been designed to stand independently of one another, and therefore they may be

covered in any order desired. It is recognized that more advanced students will have the facility to use a book such as this eclectically, referring to partial contents of individual chapters in any order, as needed. Thus, the order of chapters is not a significant factor and any order can be chosen. For example, courses that emphasize file processing may wish to study Chapters 14 through 17 early in the semester.

Special features of the book include extensive use of self-study review items, numerous exercises, and ample use of illustrations.

The authors express their appreciation to Charles E. Stewart for his very capable supervision of this project. We also extend thanks to several anonymous reviewers for their comments and recommendations.

A. S. Philippakis  
Leonard J. Kazmier

# ACKNOWLEDGMENT

The following acknowledgment is reprinted from *American National Standard Programming Language COBOL, X3.23-1974* published by the American National Standards Institute, Inc.

Any organization interested in reproducing the COBOL standard and specifications in whole or in part, using ideas from this document as the basis for an instruction manual or for any other purpose, is free to do so. However, all such organizations are requested to reproduce the following acknowledgment paragraphs in their entirety as part of the preface to any such publication (any organization using a short passage from this document, such as in a book review, is requested to mention 'COBOL' in acknowledgment of the source, but need not quote the acknowledgment):

COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

No warranty, expressed or implied, is made by any contributor or by the CODASYL Programming Language Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection therewith.

The authors and copyright holders of the copyrighted material used herein

FLOW-MATIC (trademark of Sperry Rand Corporation), Programming for the UNIVAC I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM Commercial Translator Form No. F28-8013, copyrighted 1959 by IBM; FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell

have specifically authorized the use of this material in whole or in part, in the COBOL specifications. Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications.

# CONTENTS

PREFACE	ix
ACKNOWLEDGMENT	xi
<b>1 PROGRAM STRUCTURE</b>	<b>1</b>
Introduction	1
Characteristics of Good Programs	2
Partitioning	4
Hierarchies and Networks	5
Structure Charts	15
Exercises	19
<b>2 PROGRAM COHESION</b>	<b>21</b>
The Black Box Concept	21
Cohesion in Programs	23
Functional Cohesion	24
Coincidental Cohesion	25
Class-Oriented Cohesion	27
Time-Related Cohesion	30
Procedural Cohesion	31
Data-Related Cohesion	33
Sequential Cohesion	33
Levels of Cohesion	35
Exercises	36
<b>3 PROGRAM DESIGN</b>	<b>37</b>
Introduction	37
Content Coupling	38
Module Size	40
Span of Control	44
Fan-In	46
Levels of Decisions	50

Inversion of Authority	54
Control Based on Physical Contiguity	55
Top-Down Design	57
Example of Top-Down Design	60
Exercises	64
<b>4 ELEMENTS OF COBOL</b>	<b>67</b>
COBOL Characters and Words	67
Data-Names	69
Level Numbers	74
Variables and Constants	76
Statements and Sentences	80
COBOL Format Specifications	81
The Divisions of a COBOL program	84
The COBOL Coding Form	87
The COPY Verb	88
The Operating System and the Execution of COBOL Programs	92
Summary of Common COBOL Statements	94
Sample Program	103
Exercises	106
<b>5 DATA DIVISION FEATURES</b>	<b>111</b>
Introduction	111
The PICTURE Clause for Data Description	114
The PICTURE Clause for Data Editing	121
The BLANK WHEN ZERO Clause	130
Condition-Names	130
The VALUE Clause	132
Qualification	133
Multiple-Data Records	136
The REDEFINES Clause	138
The RENAMES Clause	140
The CURRENCY and DECIMAL-POINT Clauses	142
Exercises	143
<b>6 IMPERATIVE STATEMENTS</b>	<b>149</b>
Introduction	149
File Input and Output	149
WRITE with the LINAGE Clause	153
The ACCEPT and DISPLAY Verbs	156
Arithmetic Verbs	158
The COMPUTE Verb	163
Arithmetic Precision	167



Data Transfer	176
The MOVE CORRESPONDING	178
The PERFORM Verb	180
Exercises	185
<b>7    CONDITIONAL STATEMENTS</b>	<b>193</b>
Introduction	193
Relation Conditions	194
Nested Conditions	196
Class Conditions	207
Using Conditionals to Check Input Data	209
Sign Conditions	212
Condition-Name Conditions	213
Complex Conditions	214
Exercises	216
<b>8    NUMERIC AND CHARACTER DATA</b>	<b>221</b>
Data Representation	221
The USAGE Clause	226
The SYNCHRONIZED Clause	230
Alphabets and Collating Sequences	232
The JUSTIFIED RIGHT Clause	240
The STRING and UNSTRING Verbs	241
The INSPECT Verb	248
Exercises	252
<b>9    SUBPROGRAMS</b>	<b>255</b>
Introduction	255
Calling and Called Programs	257
Subprogram Data Linkage	262
Transfer of Control	269
Sample Main and Subprogram Structure	272
Exercises	281
<b>10    STRUCTURED PROGRAMMING</b>	<b>283</b>
Introduction	283
Three Basic Program Structures	284
Additional Program Structures	287
Formatting Rules for Programs	290
Program Layout	294
Converting Unstructured Programs to Structured Form	299
Exercises	309

<b>11</b>	<b>PROGRAM TESTING</b>	<b>313</b>
	Introduction	313
	Top-Down Program Development and Testing	316
	Bottom-Up Program Development and Testing	320
	Top-Down vs. Bottom-Up Approaches to Testing	322
	Testing Procedures	323
	Common Errors	328
	COBOL Debugging Feature	334
	Exercises	341
<b>12</b>	<b>REPORT GENERATION</b>	<b>343</b>
	Introduction	343
	Control Breaks in Report Writing	345
	Logic of Report Programs	347
	The Report Writer Feature—A Basic Example	350
	Report Writer with Control Breaks	364
	Report Writer Using Declaratives	376
	Language Specifications for the COBOL Report Writer	380
	Exercises	386
<b>13</b>	<b>TABLE HANDLING</b>	<b>389</b>
	Table Definitions in COBOL	389
	An Example of a Table of Constant Values	397
	The OCCURS . . . DEPENDING ON Option	398
	The PERFORM Verb and Table Handling	400
	Sample Program with a Two-Dimensional Table	403
	Sample Program with Graphic Output	405
	Table Searching	408
	COBOL Language Options in Table Searching	414
	Sample Program with Indexing and Searching	421
	Exercises	424
<b>14</b>	<b>SEQUENTIAL FILES</b>	<b>433</b>
	File Organization	433
	File Labels	435
	Record Blocking	439
	COBOL Instructions for Sequential Files	444
	Sample Program to Create a Sequential File	454
	I/O Exception Processing	454
	Master File Maintenance	461
	A General Programming Model for Updating Sequential Files	467
	Transaction Records and File Maintenance	472

Activity Ratios and File Maintenance	476
Exercises	479
<b>15 SORTING AND MERGING</b>	<b>489</b>
Introduction	489
Internal Sorting	489
COBOL File-Sort Feature	498
SORT Statement Formats	505
File Merging	509
File Merging in COBOL	513
Exercises	516
<b>16 INDEXED SEQUENTIAL FILES</b>	<b>521</b>
Introduction	521
Indexed Sequential File Organization	521
Adding Records to an Indexed Sequential File	526
Sequential and Random Access with an Indexed Sequential File	530
VSAM—An Alternate Index Structure	532
An Example of the Creation of an Indexed File	537
COBOL Language Instructions for Indexed Files	540
An Example of Processing an Indexed File	548
Exercises	549
<b>17 RELATIVE FILES</b>	<b>555</b>
Relative File Organization	555
The Division Remainder Method	558
Other Key-to-Address Transformation Methods	561
COBOL Statements for Relative Files	564
An Example of Creating A Relative File	567
An Example of Updating A Relative File	570
Exercises	576
APPENDIX A	
ANS COBOL RESERVED WORDS	579
APPENDIX B	
COMPLETE ANS COBOL LANGUAGE FORMATS	583
INDEX	603

# 1

## Program structure

INTRODUCTION

CHARACTERISTICS OF GOOD PROGRAMS

PARTITIONING

HIERARCHIES AND NETWORKS

STRUCTURE CHARTS

EXERCISES

### INTRODUCTION

During the past decade the concepts of *structured programming* were developed based on the collective experiences of practicing programmers. Structured programming emphasizes the principles and methods for developing good program code. As such, the approach has been very beneficial in increasing the productivity of programmers, but has fallen short in terms of the broader objective of developing good programs. Perhaps an analogy from another field would be useful. In the context of constructing a building we could say that structured programming is the counterpart of developing improved methods of constructing walls, floors, ceilings, and the like. But an architectural plan is needed to identify the various rooms and spaces and to consider their interrelationships in terms of the total design of the building. What is needed in the field of programming is the counterpart of the architectural plan in the construction of a building.

The major new programming developments of the current decade are likely to be in the area of *program structure* and *design*. From this viewpoint,

the principal objectives are to identify the functions that constitute a program and to map the interrelationships among these functions. The result is then a *purposeful, coordinated structure* that can be implemented in programming code. The concepts of program structure and design are recent developments that continue to be reshaped and refined. Nevertheless, the present state of development of these concepts, as presented in this book, will serve as a *conceptual foundation for continued developments* in this field.

The present chapter considers the concept of program structure as it relates specifically to COBOL programming. Chapter 2, "Program Cohesion," describes the properties associated with good modules. Finally, Chapter 3, "Program Design," presents operational guidelines for achieving well-designed programs.

## CHARACTERISTICS OF GOOD PROGRAMS

As discussed above, the first three chapters of this text are concerned with program design. Specifically, we describe concepts and methods for developing well-designed programs. However, good program design is not the ultimate objective. The end result being sought is a written, functioning program. Therefore the ultimate objective in programming is to develop *good programs*, and we study program design as a foundation for program development.

The first and usually most important characteristic of a good program is that it be *correct*. The program should carry out the task for which it was designed and do so without error. In order to achieve this objective, a complete and clear specification of the purpose and functions of the program must be obtained. Thus, program "errors" can be due to outright mistakes on the part of a programmer, or be due to a lack of a clear description regarding the required output of the program.

The second characteristic of a good program is that it be *understandable*. Although a computer program is a set of instructions for a computer, it should also be comprehensible to other people. A person other than the author should be able to read and understand the purpose and functions of the program. Higher-level programming languages such as COBOL are intended for human use and interpretation in their direct form, and are intended for machine use only indirectly, through compilation.

Next, a computer program should be *easy to change*. Changes in products, changes in company procedures, new government regulations, and the like all lead to the necessity of modifying existing computer programs. As a

consequence, most established computer installations devote considerable time and effort to changing existing programs. Thus, a good program not only fulfills its original purpose, but also is easily adaptable in response to a changing environment.

The fourth characteristic of a good program is that it be *written efficiently*, which concerns the amount of time spent in writing the program. Of course, this objective is secondary to the program being correct, understandable, and easy to change. In practice, the easiest way to write a program quickly is to write it partly correct, leave it difficult to understand, or allow its obscurity to make it difficult to change. Still, the principal cost of a programming project is the programmer's time, and programming techniques which economize this time while still satisfying the objectives that the program be correct, understandable, and easy to change are preferred.

The final characteristic of a good program which we consider is that it should *execute efficiently*. The program should be so written that it does not use more computer storage nor more computer processing time than is necessary. Again, this objective also is secondary to the primary objectives that a program be correct, understandable, and easy to change. Furthermore, as hardware costs have decreased relative to programmers' salaries, overall cost considerations often justify minimizing the concern about a high level of efficiency in program execution. Nevertheless, a programmer should be alert to the techniques by which efficient program execution can be achieved.

## REVIEW

- 1 The most important characteristic of a good computer program is that it be \_\_\_\_\_.  
*correct*
- 2 Probable reference to the program by individuals other than the original programmer dictates that it should be \_\_\_\_\_, while unavoidable changes in data processing requirements in most organizations make it desirable that the program be \_\_\_\_\_.  
*understandable; easy to change*
- 3 In order to economize the programmer's time, a program should be \_\_\_\_\_ efficiently; in order to economize the use of computer hardware, the program should \_\_\_\_\_ efficiently.  
*written; execute*

## PARTITIONING

A fundamental concept of program design is that of *partitioning*, which refers to the process of subdividing a large programming task into smaller parts or functions.

Partitioning is a pervasive phenomenon in human activities. One common form of partitioning in organizations is based on the division of labor, or functional specialization. For example, an automobile manufacturing plant includes departmental units which may be further subdivided according to specific functions. A Painting Department, for instance, could include such separate functions as cleaning, spraying, baking, inspecting, and the like. Similarly, an Electronic Data Processing Department could include such separate functions as programming, systems analysis, data entry, and input-output control. The common occurrence of partitioning in a variety of situations is reflective of the physical and mental limits of human beings. A given person can only do so much and attend to so much at a given time. Therefore, we find it not only beneficial, but also necessary, to partition large and complex tasks into smaller and more specialized tasks.

A computer programming task generally is complex enough to make partitioning desirable. From the standpoint of the individual programmer, the partitioning of the overall task allows the programmer to concentrate on particular program functions. From the standpoint of the organization, partitioning makes it possible to complete complex programming tasks in a shorter time by having a team of programmers working simultaneously on different specific tasks that constitute the overall program.

In the context of computer programming, a widely used term associated with partitioning is *modularity*. A program module is a well-defined program segment. Modular programming has been recognized as a desirable practice for many years. In practice, all programs include some degree of modularity by necessity: no programmer can write a monolithic program that is not partitioned into some kinds of parts, or modules. Thus, it is not just presence of modularity that is important. Rather, we need to develop an understanding of how to design programs whose modules are so constructed as to lead to good programs, where "program goodness" is defined by the attributes discussed in the preceding section of this chapter.

To be useful, a module should not only be a program segment, but a *well-defined* program segment. More specifically, a module should be a named program segment that carries out a specific program function. In the context of COBOL programming, a module eventually is represented in one of four forms in the program:

- 1 As a single paragraph
- 2 As a series of two or more consecutive paragraphs which are the object

of a PERFORM PA THRU PZ, where PA and PZ stand for the first and last paragraphs

- 3 As a single section
- 4 As a program subroutine

## REVIEW

- 1 The process of subdividing a large programming task into smaller, more specific tasks is called \_\_\_\_\_.  
*partitioning*
- 2 In terms of human endeavors, partitioning is a [long-standing/recently developed] concept.  
*long-standing*
- 3 "A named program segment that carries out a specific program function" is a definition of a program \_\_\_\_\_.  
*module*
- 4 Program modularity can be described as being effective when it leads to the development of \_\_\_\_\_ programs.  
*good*

## HIERARCHIES AND NETWORKS

As described in the preceding section, partitioning is the process by which a large programming task can be subdivided into smaller parts. But there must be an integrating force in order to attain coordinated results with respect to the parts. This force is provided by *structure*, which refers to the identification of system components and their interrelationships. The two basic forms of structure are *hierarchies* and *networks*.

Hierarchies are often referred to as tree structures, especially in the context of data bases. A hierarchy, or tree, is a structure such that there is a single module at the top with one or more *subordinate* modules. The singular top module is *superordinate* or *superior* to its subordinate modules, and these subordinate modules may themselves have additional subordinate modules to which they are superior. Any given module can be subordinate to only one superior, but may be superordinate to one or more subordinate modules. Figure 1-1 portrays a typical hierarchical structure. As can be



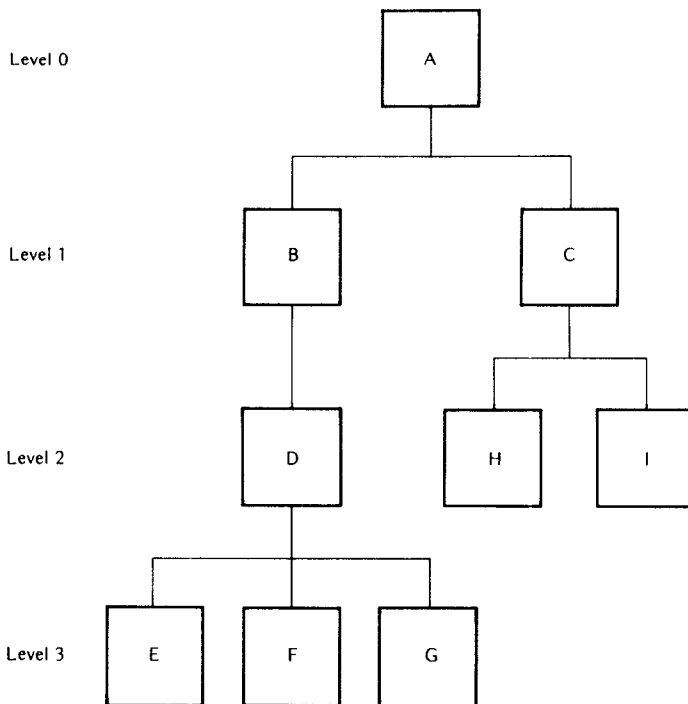


FIGURE 1-1 REPRESENTATION OF HIERARCHY (TREE) STRUCTURES.

observed, the single module A is at the top (level 0) of the structure. This module has two subordinate modules, B and C, which comprise level 1 in the hierarchy, and which have further subordinates. The designation of superordinates and subordinates constitutes the specification of the relationships. Two basic characteristics of any hierarchy structure is that there is a single superordinate module for the entire hierarchy and that there is only one superordinate module for each subordinate module.

As contrasted to a hierarchy structure, in a *network structure* there is no single module that is superordinate to all others, and relationships among modules are unrestricted. In other words, two modules may relate to each other in both directions, so that we cannot say that one is superordinate to the other. Figure 1-2 includes two examples of networks. In the first network module A is superior to all three of the other modules B, C, and D, as indicated by the direction of the arrows. But notice that B is also subordinate to C and C is also subordinate to D, which violates the rule for hierarchies that a subordinate should have only one superordinate. The second diagram in Figure 1-2 illustrates a network in which every possible relationship is