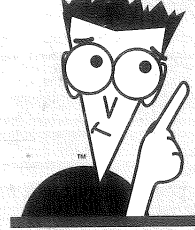


Free COBOL "Cheat Sheet" Inside!

Perform Complex Data Conversions with Simple, One-line Commands!

COBOL — the language of business data processing — has been used for accounting systems, inventory control, database maintenance, payroll systems, and many other applications. With the help of *COBOL For Dummies*®, you'll read and understand existing COBOL programs as well as write programs of your own. This book contains everything you need to know to store and retrieve data in files, perform calculations necessary for business operations, and organize and format data for presentation on paper and on the computer screen.

Inside, find helpful advice on how to:

- Choose the right solutions for the year 2000 problem
- Use simple, step-by-step processes for defining different types of file organizations and accesses
- Create PICTURE clauses to store and format data in ways that make the most sense for your applications
- Explore the data tables that you can build inside a COBOL program to model real-world events and situations
- Set up sorting procedures capable of handling millions of records

Technical Review

by Jeffrey R. Lagasse

Let These Icons Guide You!

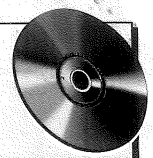
Points out places where the year 2000 problem can creep into a program

Summarizes the rules and regulations that COBOL imposes

Highlights obsolete forms found in existing COBOL code and offers better ways to complete the same task

About the Author

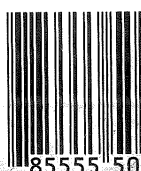
Arthur Griffith has worked with computer language compilers and interpreters since 1977. His first experience with COBOL was in 1980, when he worked on a special COBOL compiler with database extensions. His work with compilers has included C, FORTH, and special-purpose languages. All in all, he has worked on and in 23 computer languages.



Valuable Bonus CD Includes:

- Complete COBOL development system, including compilers for Windows, HP-UX, and Sun
- Freeware COBOL interpreter that runs on DOS, Linux, Windows, Solaris, SunOS, and AIX
- Source code for sample programs presented in the book, including programs that address different facets of the millennium problem

SYSTEM REQUIREMENTS: 486 or faster PC with Windows 3.1 or later; 16MB RAM; and a CD-ROM drive



READER LEVEL
Beginning to Intermediate

COMPUTER BOOK SHELVING CATEGORY
PCs/Programming/COBOL

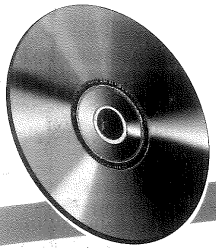
$29.99 USA
$42.99 Canada/£28.99 Incl. VAT UK

BESTSELLING BOOK SERIES FROM IDG

Dummies Press
a division of
IDG Books Worldwide, Inc.
An International
Data Group Company

see us at:
www.dummies.com
for info on other IDG Books titles:
www.idgbooks.com

ISBN 0-7645-0298-0



Perform Complex Data Conversions with Simple, One-line Commands!

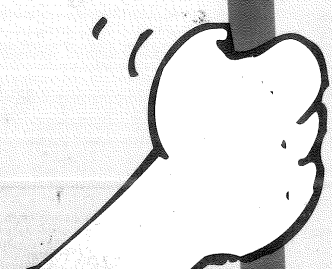
40 MILLION DUMMIES BOOKS IN PRINT

COBOL FOR DUMMIES®

Free COBOL Compiler on the CD!

A Reference for the Rest of Us!®

by Arthur Griffith

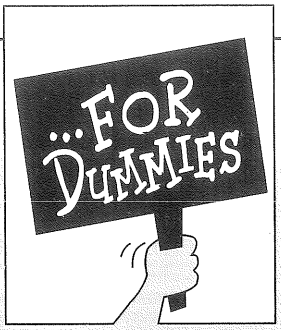


The Fun and Easy Way™ to Uncover the Inner Workings of COBOL

Your First Aid Kit® for Getting COBOL Programs from this Millennium into the Next

Industry Standard COBOL — Explained in Plain English

IDG BOOKS



References for the Rest of Us!®

COMPUTER BOOK SERIES FROM IDG

Are you intimidated and confused by computers? Do you find that traditional manuals are overloaded with technical details you'll never use? Do your friends and family always call you to fix simple problems on their PCs? Then the ...*For Dummies*® computer book series from IDG Books Worldwide is for you.

...*For Dummies* books are written for those frustrated computer users who know they aren't really dumb but find that PC hardware, software, and indeed the unique vocabulary of computing make them feel helpless. ...*For Dummies* books use a lighthearted approach, a down-to-earth style, and even cartoons and humorous icons to diffuse computer novices' fears and build their confidence. Lighthearted but not lightweight, these books are a perfect survival guide for anyone forced to use a computer.

"I like my copy so much I told friends; now they bought copies."

Irene C., Orwell, Ohio

"Quick, concise, nontechnical, and humorous."

Jay A., Elburn, Illinois

"Thanks, I needed this book. Now I can sleep at night."

Robin F., British Columbia, Canada

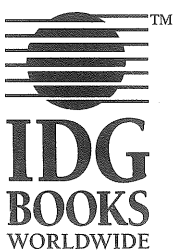
Already, millions of satisfied readers agree. They have made ...*For Dummies* books the #1 introductory level computer book series and have written asking for more. So, if you're looking for the most fun and easy way to learn about computers, look to ...*For Dummies* books to give you a helping hand.

IDG BOOKS WORLDWIDE™

5/97

COBOL FOR DUMMIES®

by Arthur Griffith



IDG Books Worldwide, Inc.
An International Data Group Company

Foster City, CA ♦ Chicago, IL ♦ Indianapolis, IN ♦ Southlake, TX

COBOL For Dummies®

Published by
IDG Books Worldwide, Inc.
An International Data Group Company
919 E. Hillsdale Blvd.
Suite 400
Foster City, CA 94404
www.idgbooks.com (IDG Books Worldwide Web site)
www.dummies.com (Dummies Press Web site)

Library of Congress Catalog Card No.: 97-808414

ISBN: 0-7645-0298-0

Printed in the United States of America

10 9 8 7 6 5 4 3 2

1O/QV/QS/ZY/IN

Distributed in the United States by IDG Books Worldwide, Inc.

Distributed by Macmillan Canada for Canada; by Transworld Publishers Limited in the United Kingdom; by IDG Norge Books for Norway; by IDG Sweden Books for Sweden; by Woodslane Pty. Ltd. for Australia; by Woodslane Enterprises Ltd. for New Zealand; by Longman Singapore Publishers Ltd. for Singapore, Malaysia, Thailand, and Indonesia; by Simron Pty. Ltd. for South Africa; by Toppan Company Ltd. for Japan; by Distribuidora Cuspide for Argentina; by Livraria Cultura for Brazil; by Ediciencia S.A. for Ecuador; by Addison-Wesley Publishing Company for Korea; by Ediciones ZETA S.C.R. Ltda. for Peru; by WS Computer Publishing Corporation, Inc., for the Philippines; by Unalis Corporation for Taiwan; by Contemporanea de Ediciones for Venezuela; by Computer Book & Magazine Store for Puerto Rico; by Express Computer Distributors for the Caribbean and West Indies. Authorized Sales Agent: Anthony Rudkin Associates for the Middle East and North Africa.

For general information on IDG Books Worldwide's books in the U.S., please call our Consumer Customer Service department at 800-762-2974. For reseller information, including discounts and premium sales, please call our Reseller Customer Service department at 800-434-3422.

For information on where to purchase IDG Books Worldwide's books outside the U.S., please contact our International Sales department at 650-655-3200 or fax 650-655-3295.

For information on foreign language translations, please contact our Foreign & Subsidiary Rights department at 650-655-3021 or fax 650-655-3281.

For sales inquiries and special prices for bulk quantities, please contact our Sales department at 650-655-3200 or write to the address above.

For information on using IDG Books Worldwide's books in the classroom or for ordering examination copies, please contact our Educational Sales department at 800-434-2086 or fax 817-251-8174.

For press review copies, author interviews, or other publicity information, please contact our Public Relations department at 650-655-3000 or fax 650-655-3299.

For authorization to photocopy items for corporate, personal, or educational use, please contact Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, or fax 978-750-4470.

About the Author

Arthur Griffith is author of the *Java Master Reference* and co-author of *Discover Visual Café*. He began working as a programmer in 1977, and after about five years, he became an independent software contractor. While working primarily with computer language compilers and interpreters (he has worked in and on more than 25 languages), he has involved himself with nearly all forms of computing. He has programmed in such diverse areas as health insurance company data processing, Defense Department missile guidance, and digital telephony. His primary area of expertise is in computer languages. In particular, he was involved in porting a COBOL compiler to several platforms.

Before becoming a programmer, he worked as a disc jockey, a tower climber, a TV newsman, a broadcast engineer, a stage and film actor, part-owner of a television station, a trade school teacher, and an airlines reservations agent. He is still involved with acting and can be seen in the movie *The Newton Boys* and in reruns of *Walker, Texas Ranger*.

ABOUT IDG BOOKS WORLDWIDE

Welcome to the world of IDG Books Worldwide.

IDG Books Worldwide, Inc., is a subsidiary of International Data Group, the world's largest publisher of computer-related information and the leading global provider of information services on information technology. IDG was founded more than 25 years ago and now employs more than 8,500 people worldwide. IDG publishes more than 275 computer publications in over 75 countries (see listing below). More than 60 million people read one or more IDG publications each month.

Launched in 1990, IDG Books Worldwide is today the #1 publisher of best-selling computer books in the United States. We are proud to have received eight awards from the Computer Press Association in recognition of editorial excellence and three from *Computer Currents*' First Annual Readers' Choice Awards. Our best-selling *...For Dummies®* series has more than 30 million copies in print with translations in 30 languages. IDG Books Worldwide, through a joint venture with IDG's Hi-Tech Beijing, became the first U.S. publisher to publish a computer book in the People's Republic of China. In record time, IDG Books Worldwide has become the first choice for millions of readers around the world who want to learn how to better manage their businesses.

Our mission is simple: Every one of our books is designed to bring extra value and skill-building instructions to the reader. Our books are written by experts who understand and care about our readers. The knowledge base of our editorial staff comes from years of experience in publishing, education, and journalism — experience we use to produce books for the '90s. In short, we care about books, so we attract the best people. We devote special attention to details such as audience, interior design, use of icons, and illustrations. And because we use an efficient process of authoring, editing, and desktop publishing our books electronically, we can spend more time ensuring superior content and spend less time on the technicalities of making books.

You can count on our commitment to deliver high-quality books at competitive prices on top'cs you want to read about. At IDG Books Worldwide, we continue in the IDG tradition of delivering quality for more than 25 years. You'll find no better book on a subject than one from IDG Books Worldwide.

IDG BOOKS WORLDWIDE

John Kilcullen
CEO
IDG Books Worldwide, Inc.

Steven Berkowitz
President and Publisher
IDG Books Worldwide, Inc.

VIII WINNER
Eighth Annual Computer Press Awards 1992

IX WINNER
Ninth Annual Computer Press Awards 1993

1994 COMPUTER CONSUMER 1995 READERS' CHOICE

X WINNER
Tenth Annual Computer Press Awards 1994

XI WINNER
Eleventh Annual Computer Press Awards 1995

IDG Books Worldwide, Inc., is a subsidiary of International Data Group, the world's largest publisher of computer-related information and the leading global provider of information services on information technology. International Data Group publishes over 275 computer publications in over 75 countries. Sixty million people read one or more International Data Group publications each month. International Data Group's publications include: **ARGENTINA:** Buyer's Guide, Computerworld Argentina, PC World Argentina; **AUSTRALIA:** Australian Macworld, Australian PC World, Australian Reseller News, Computerworld, IT Casebook, Network World, Publish, Webmaster; **AUSTRIA:** Computerwelt Osterreich, Networks Austria, PC Tip Austria; **BANGLADESH:** PC World Bangladesh; **BELARUS:** PC World Belarus; **BELGIUM:** Data News; **BRAZIL:** Annuário de Informática, Computerworld, Connections, Macworld, PC Player, PC World, Publish, Reseller News, Supergamepower; **BULGARIA:** Computerworld Bulgaria, Network World Bulgaria, PC & MacWorld Bulgaria; **CANADA:** CIO Canada, Client/Server World, ComputerWorld Canada, InfoWorld Canada, NetworkWorld Canada, WebWorld; **CHILE:** Computerworld Chile, PC World Chile; **COLOMBIA:** Computerworld Colombia, PC World Colombia; **COSTA RICA:** PC World Centro America; **THE CZECH AND SLOVAK REPUBLICS:** Computerworld Czechoslovakia, Macworld Czech Republic, PC World Czechoslovakia; **DENMARK:** Communications World Danmark, Computerworld Danmark, Macworld Danmark, PC World Danmark, Techworld Denmark; **DOMINICAN REPUBLIC:** PC World Republica Dominicana; **ECUADOR:** PC World Ecuador; **EGYPT:** Computerworld Middle East, PC World Middle East; **EL SALVADOR:** PC World Centro America; **FINLAND:** MikroPC, Tietoverkko, Tietoviikko; **FRANCE:** Distributique, Hebdo, Info PC, Le Monde Informatique, Macworld, Reseaux & Telecoms, WebMaster France; **GERMANY:** Computer Partner, Computerwoche, Computerwoche Extra, Computerwoche FOCUS, Global Online, Macwelt, PC Welt; **GREECE:** Amiga Computing, GamePro Greece, Multimedia World; **GUATEMALA:** PC World Centro America; **HONDURAS:** PC World Centro America; **HONG KONG:** Computerworld Hong Kong, PC World Hong Kong, Publish in Asia; **HUNGARY:** ABCD CD-ROM, Computerworld Szamitastechnika, Internetto online Magazine, PC World Hungary, PC-X Magazin Hungary; **ICELAND:** Tolvuheimur PC World Island; **INDIA:** Information Communications World, Information Systems Computerworld, PC World India, Publish in Asia; **INDONESIA:** InfoKomputer PC World, Komputek Computerworld, Publish in Asia; **IRELAND:** ComputerScope, PC Live!; **ISRAEL:** Macworld Israel, People & Computers/Computerworld; **ITALY:** Computerworld Italia, Macworld Italia, Networking Italia, PC World Italia; **JAPAN:** DTP World, Macworld Japan, Nikkei Personal Computing, OS/2 World Japan, SunWorld Japan, Windows NT World, Windows World Japan; **KENYA:** PC World East African; **KOREA:** Hi-Tech Information, Macworld Korea, PC World Korea; **MACEDONIA:** PC World Macedonia; **MALAYSIA:** Computerworld Malaysia, PC World Malaysia, Publish in Asia; **MALTA:** PC World Malta; **MEXICO:** Computerworld Mexico, PC World Mexico; **MYANMAR:** PC World Myanmar; **NETHERLANDS:** Computer! Totaal, LAN Internetworking Magazine, LAN World Buyers Guide, Macworld Netherlands, Net, WebWereld; **NEW ZEALAND:** Absolute Beginners Guide and Plain & Simple Series, Computer Buyer, Computer Industry Directory, Computerworld New Zealand, MTB, Network World, PC World New Zealand; **NICARAGUA:** PC World Centro America; **NORWAY:** Computerworld Norge, CW Rapport, Datamagasinet, Financial Rapport, Kursguide Norge, Macworld Norge, Multimediaworld Norge, PC World Ekspress Norge, PC World Netverk, PC World Norge, PC World ProduktGuide Norge; **PAKISTAN:** Computerworld Pakistan; **PANAMA:** PC World Panama; **PEOPLE'S REPUBLIC OF CHINA:** China Computer Users, China Computerworld, China InfoWorld, China Telecom World Weekly, Computer & Communication, Electronic Design China, Electronics Today, Electronics Weekly, Game Software, PC World China, Popular Computer Week, Software Weekly, Software World, Telecom World; **PERU:** Computerworld Peru, PC World Profesional Peru, PC World SoHo Peru; **PHILIPPINES:** Click!, Computerworld Philippines, PC World Philippines, Publish in Asia; **POLAND:** Computerworld Poland, Computerworld Special Report Poland, Cyber, Macworld Poland, Networld Poland, PC World Komputer; **PORTUGAL:** Cerebro/PC World, Computerworld/Correio Informatico, Dealer World Portugal, Mac*In/PC*in Portugal, Multimedia World; **PUERTO RICO:** PC World Puerto Rico; **ROMANIA:** Computerworld Romania, PC World Romania, Telecom Romania; **RUSSIA:** Computerworld Russia, Mir PK, Publish, Seti; **SINGAPORE:** Computerworld Singapore, PC World Singapore, Publish in Asia; **SLOVENIA:** Monitor; **SOUTH AFRICA:** Computing SA, Network World SA, Software World SA; **SPAIN:** Communicaciones World España, Computerworld España, Dealer World España, Macworld España, PC World España; **SRI LANKA:** Infolink PC World; **SWEDEN:** CAP&Design, Computer Sweden, Corporate Computing Sweden, Internetworld Sweden, it branschen, Macworld Sweden, MaxiData Sweden, MikroDatorn, Natverk & Kommunikation, PC World Sweden, PCaktiv, Windows World Sweden; **SWITZERLAND:** Computerworld Schweiz, Macworld Schweiz, PCtip; **TAIWAN:** Computerworld Taiwan, Macworld Taiwan, NEW VISION/Publish, PC World Taiwan, Windows World Taiwan; **THAILAND:** Publish in Asia, Thai Computerworld; **TURKEY:** Computerworld Turkiye, Macworld Turkiye, Network World Turkiye, PC World Turkiye; **UKRAINE:** Computerworld Kiev, Multimedia World Ukraine, PC World Ukraine; **UNITED KINGDOM:** Acorn User UK, Amiga Action UK, Amiga Computing UK, Apple Talk UK, Computing, Macworld, Parents and Computers UK, PC Advisor, PC Home, PSX Pro, The WEB; **UNITED STATES:** Cable in the Classroom, CIO Magazine, Computerworld, DOS World, Federal Computer Week, GamePro Magazine, InfoWorld, I-Way, Macworld, Network World, PC Games, PC World, Publish, Video Event, THE WEB Magazine, and WebMaster; online webzines: JavaWorld, NetscapeWorld, and SunWorld Online; **URUGUAY:** InfoWorld Uruguay; **VENEZUELA:** Computerworld Venezuela, PC World Venezuela; and **VIETNAM:** PC World Vietnam.

3/24/97

Dedication

For Mary.

Author's Acknowledgments

I want to thank John Pont — a kindly editor with infinite patience and a superb command of the English language. He's really picky, though. And there's Joe Jansen — a gentleman with some sort of mystical power that enables him to detect anything that is missing from a manuscript. It's kind of spooky the way he does it. And there's Jeff Lagasse, who used his knowledge of COBOL to catch me off base here and there and kindly guide me back. If I were scrupulously honest, I would include their names on the front of the book and share the royalties with them, but I have never been excessively scrupulous.

I want to thank Margot Maley at Waterside Productions. I also want to thank Gareth Hancock at IDG Books Worldwide for performing the difficult task of believing that I could write this book.

I want to thank Ron Souder for exposing me to some of the most peculiar COBOL programs in the world. I want to thank Ginger Mensik for letting me use some of her words. I don't need to tell you which ones are hers — she will point them out. I also need to thank about 50 people on the `comp.lang.cobol` newsgroup on the Internet for helping me find the answers to some difficult questions.

I want to thank Jim Grant for helping me gain the confidence needed for taking on a project like writing a book. Next to freedom, there is no greater treasure than self-confidence. Those who don't appreciate it have never been without it.

I want to thank my son Art Griffith for all the work he did rendering the syntax diagrams for Bonus Appendix A on the CD.

Publisher's Acknowledgments

We're proud of this book; please register your comments through our IDG Books Worldwide Online Registration Form located at: `http://my2cents.dummies.com`.

Some of the people who helped bring this book to market include the following:

Acquisitions, Development, and Editorial

Project Editor: John W. Pont

Acquisitions Editor: Gareth Hancock

Media Development Manager: Joyce Pepple

Associate Permissions Editor: Heather Heath Dismore

Copy Editor: Joe Jansen

Technical Editor: Jeffrey R. Lagasse

Editorial Manager: Mary C. Corder

Editoral Assistant: Donna Love

Production

Project Coordinator: Valery Bourke

Layout and Graphics: Angela F. Hunckler, Todd Klemme, Brent Savage, Cameron Booker, Mark Owens, Ian A. Smith

Proofreaders: Ethel M. Winslow, Christine Berman, Kelli Botta, Michelle Croninger, Nancy Price, Rebecca Senninger, Janet M. Withers

Indexer: David Heiret

Special Help

Suzanne Thomas, Associate Editor; Kevin Spencer, Associate Technical Editor; Stephanie Koutek, Proof Editor; Elizabeth Netedu Kuball, Copy Editor; Andrea C. Boucher, Copy Editor

---

General and Administrative

IDG Books Worldwide, Inc.: John Kilcullen, CEO; Steven Berkowitz, President and Publisher

IDG Books Technology Publishing: Brenda McLaughlin, Senior Vice President and Group Publisher

Dummies Technology Press and Dummies Editorial: Diane Graves Steele, Vice President and Associate Publisher; Mary Bednarek, Acquisitions and Product Development Director; Kristin A. Cocks, Editorial Director

Dummies Trade Press: Kathleen A. Welton, Vice President and Publisher; Kevin Thornton, Acquisitions Manager

IDG Books Production for Dummies Press: Beth Jenkins Roberts, Production Director; Cindy L. Phipps, Manager of Project Coordination, Production Proofreading, and Indexing; Kathie S. Schutte, Supervisor of Page Layout; Shelley Lea, Supervisor of Graphics and Design; Debbie J. Gates, Production Systems Specialist; Robert Springer, Supervisor of Proofreading; Debbie Stailey, Special Projects Coordinator; Tony Augsburger, Supervisor of Reprints and Bluelines; Leslie Popplewell, Media Archive Coordinator

Dummies Packaging and Book Design: Patti Crane, Packaging Specialist; Kavish + Kavish, Cover Design

♦

The publisher would like to give special thanks to Patrick J. McGovern, without whom this book would not have been possible.

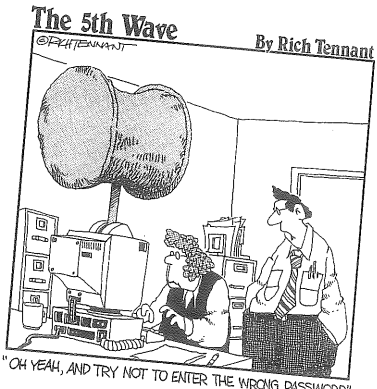
♦

Contents at a Glance

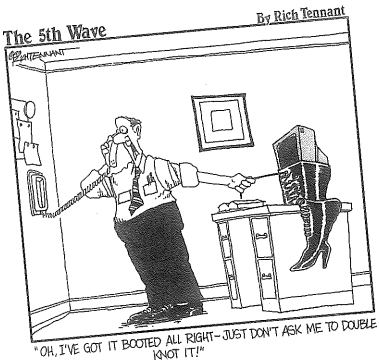
Cartoons at a Glance

By Rich Tennant

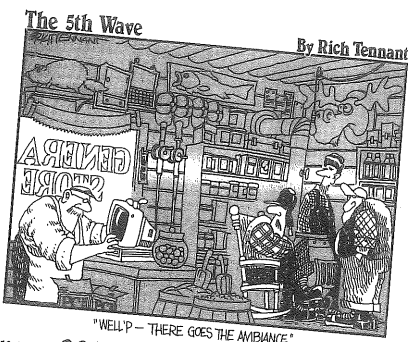
page 323



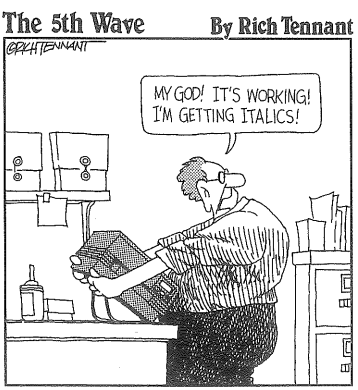
page 5



page 131



page 229



page 53

Fax: 978-546-7747 • E-mail: the5wave@tiac.net

Table of Contents

Introduction

I know why you're reading this book. It's a job thing, right? I thought so. You see, there are no COBOL hobbyists. I don't really know why that's the case — all the other languages have hobbyists who write programs, share code, and talk enthusiastically about the language. But not COBOL. COBOL programmers have their sleeves rolled up and they're doing a day's work.

Let me say right up front that I genuinely like COBOL. Really. Some people think of COBOL as being old-fashioned and outdated. I can't agree. COBOL has done, and is continuing to do, so many things so very well. It has made possible many wonders of our modern age. Really important things, like running the entire worldwide insurance industry, generating millions of utility bills every month, printing those little payment books for car loans, and keeping me employed during those lean years back there in the 1980s.

About This Book

COBOL For Dummies is a reference book, but it's organized so you can read it straight through if you want to. If you are new to programming (or just new to COBOL), you probably want to start from the beginning and read Parts I, II, and III, and then skim through the rest of the book just to see what's there.

The purpose of this book is to show you how to write COBOL programs, but you won't find lots of technical explanations. You will find simple examples and straightforward descriptions, written in plain English. This book is based on standard COBOL and is valid no matter which operating system and compiler you use.

The Portability of COBOL

Only in rare cases is a COBOL program portable from one environment to another. COBOL was not designed to be portable. It was designed to be *reusable,* but that's different — that's sort of like being recyclable.

This book covers the information you need to know about COBOL, but you will encounter cases in which each individual compiler goes in its own direction. This book is based on standard COBOL (as defined in the *American National Standard for Information Systems — Programming Language — COBOL,* which is also known as ANSI X3.23-1985 and as ISO 1989-1985); where the standard ends and the peculiarities begin, I briefly describe the possibilities and how you may proceed. In most of these cases, you need more information about your particular compiler. I suppose I could have included all the information from all the compilers, but really, would you buy a 4,000-page book?

How This Book Is Organized

I divide the description of COBOL into five parts. Part I covers the overall structure of the language. Parts II and III cover the mechanics of declaring data and writing code to process the data. Part IV is all about input and output. Part V contains descriptions and examples of things you can do with a COBOL program. The Appendix describes the contents of the CD that accompanies this book.

Part I: COBOL Has Structure; Boy, Does It!

Part I gives you information about the structure of the COBOL language. You can find simple examples that demonstrate the structure of a COBOL program and how its parts interact. I also provide descriptions of the basic building blocks of a COBOL program.

Part II: The DATA DIVISION Is Where You Put Things

The purpose of any computer program is to mess around with data. The chapters in Part II describe the ways in which you can use COBOL to declare areas of data. It turns out that logical formatting of data is one of COBOL's strong suits.

Part III: The PROCEDURE DIVISION Is Where You Do Things

A COBOL program runs on its verbs, and the chapters in Part III describe how you use verbs to construct statements and sentences that order COBOL to go forth and do your bidding. In other words, Part III covers the part of a COBOL program that messes around with your data.

Part IV: Input, Output, and Sorting

Unless a program communicates with the outside world, it doesn't matter whether the program messes around with data. In other words, if a program doesn't tell anybody what it has done, it hasn't done anything. Part IV is all about COBOL communicating with the outside world and putting data in order.

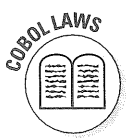
Part V: The Part of Tens

The millennium is here. Along with ten things you can do about the year 2000 problem, the chapters in Part V describe 10 other hard things you can do with COBOL.

Icons Used in This Book

Throughout the pages of this book, I use icons to flag important information.

COBOL programs have many places where year 2000 problems can hide. This icon marks the places in the book where I expose these hiding places.



COBOL has lots of rules about how things must be done. Some rules are intuitive and some are not — but wherever you see this icon, you can find the rules for using a particular COBOL keyword or structure.

While working on code that your company has been using for many years, you may run into some programming constructs that the COBOL standard now considers obsolete. This icon highlights not only an obsolete piece of COBOL, but also my description of a better way to write that COBOL code.

COBOL has lots of snares and traps that can catch the unwary, and I use this icon to point out those sneaky little rascals. Be careful.

This icon marks the presence of something useful. Usually, it points out something you can do that isn't obvious from the language definition itself.

You can skip anything marked by this icon, which flags information that is not directly required for you to write a COBOL program. This icon typically identifies something that you may want to know just because you are curious about what goes on backstage.

How'd I Do?

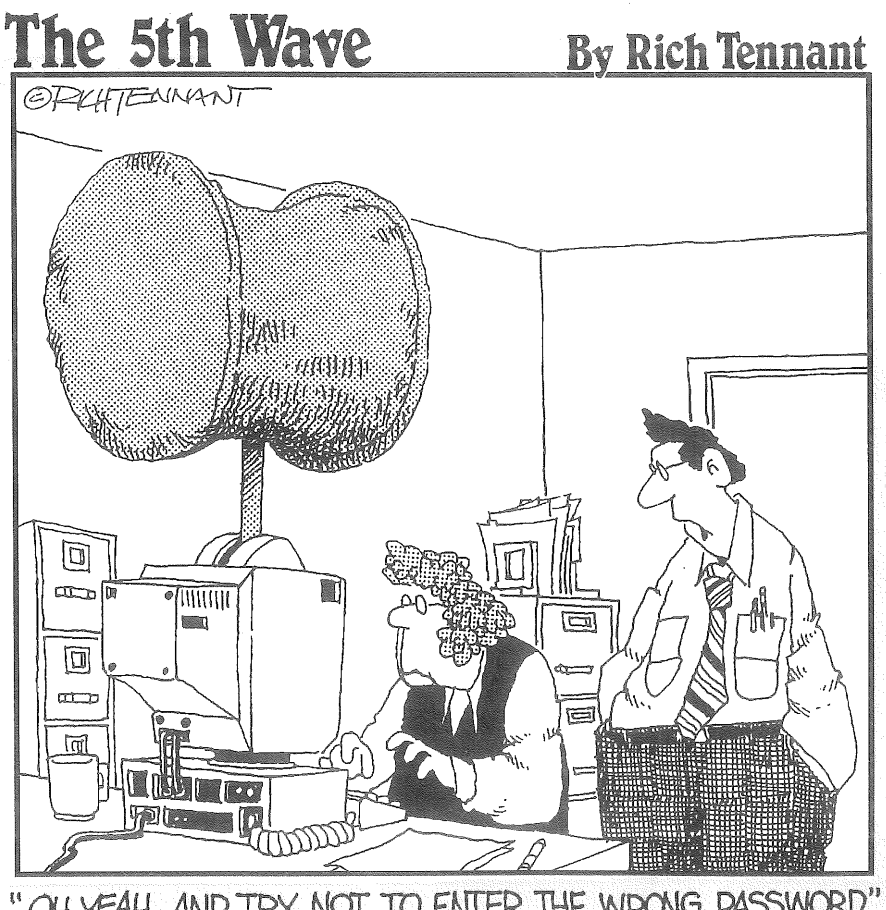
I am proud of this book. I enjoyed writing it, and I hope you enjoy reading it. I made every effort to make the book the easiest of all possible ways to understand COBOL programming. If you want to make a comment, or point out an error, or if you have some question, please feel free to send an e-mail message to me at the following address: `arthur98@airmail.net`.

I may take a week or more to answer because of schedules, deadlines, and how hard your question is. But I will answer.

I wish you the best of luck with your programming. By writing COBOL programs, you are continuing a long tradition that goes back to the dawn of business computing. Have a happy millennium!

Part I

COBOL Has Structure; Boy, Does It!



In this part . . .

A minimum structure — a skeleton if you will —must be in place for a collection of lines of text to be called a COBOL program. After you understand this basic structure, you can build on it to construct your programs. COBOL is verbose. Its verbosity is verbose. The chapters in this part of the book describe the basics of putting this verbosity to work for you in writing programs.

After you read this part, you will know how to write a simple COBOL program. Armed with the information I present in this part of the book, you can look at any COBOL program listing and identify its parts and understand its general structure.

Every COBOL program has four divisions, and each division serves a special purpose. In this part, I describe each of the divisions and give you a look at what goes into them. I also explain how the divisions interact with one another to create a COBOL program. Not only does a COBOL program have a fairly rigid structure overall, but each line of code also must conform to a specific format. All of these shape-and-size rules have been with COBOL since the days when code was punched on cards and fed through a slot.

COBOL has written laws and unwritten laws. The written laws have to do with character sets, keywords, and maximum sizes. The unwritten laws — COBOL traditions, really — have to do with character cases, line formatting, and paragraph naming. Throughout this and the other parts of the book, I show you how to become a law-abiding COBOL programmer.

Chapter 1

The Smallest COBOL Programs in the World

In This Chapter

- Getting started with programming and the COBOL language
- Understanding the basic form of a COBOL program
- Taking a look at some really tiny COBOL programs
- Compiling a COBOL program

COBOL is an acronym contrived from the phrase "COmmon Business Oriented Language." COBOL is a *computer language.* You use a computer language to create a collection of human-written instructions that you can input to a computer program called a *compiler.* A compiler translates the instructions you've written into *machine language* instructions. In other words, a compiler takes the instructions you write and translates them into a form the computer can understand. The gang of machine language instructions is known as a *program* — the set of instructions that tell a computer what to do. A *computer* is, uh, well, you know what a computer is.

Well, that just about covers the whole subject — except for a few hundred details. If you are interested in the details, read on. I mean, if you want to find out how to write COBOL programs, how to fix COBOL programs that have problems, how to modify COBOL programs so they won't fail when the millennium comes, and how to restructure COBOL programs to make them better, then read on. On the other hand, if you don't intend to actually *do* anything, you could consider yourself as having gone far enough. In fact, for certain levels of management, you may be over-trained.

A Program So Small That It Does Nothing

Have you ever seen a COBOL program? They are attractive little devils. Here is just about the simplest of all possible COBOL programs. This program is so simple, it doesn't even do anything:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Brunhilda.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
ParagraphName.
```

This is a skeleton COBOL program. In fact, you can leave out the ENVIRONMENT and DATA divisions and still have a valid program — the other divisions are the required parts of any COBOL program. This program doesn't actually do anything, but it is complete and correct. If you want to write your own program, from scratch, you need to include the structure you see here (of course, you will probably want to use your own program name and paragraph name).

Every COBOL program has four divisions. Each division has a specific purpose:

- Since the beginning of time, every COBOL program ever written has begun with the immortal words `IDENTIFICATION DIVISION`. You can use this convention to identify a COBOL program every time you see one — if a program doesn't start with those two magic words, it ain't COBOL. Lots of things can go into an `IDENTIFICATION DIVISION`, as I describe in Chapter 2, but the only one required is the `PROGRAM-ID`, which you use to name the program. Because the preceding program does absolutely nothing, I thought it was appropriate to name it `Brunhilda` — a pet name for someone I once knew.
- The `ENVIRONMENT DIVISION` is where you tell the COBOL program in what type of computer environment the program can run. For example, you can specify the computer model number, the compiler version, and stuff about the equipment that is connected to the system (printers, display screens, modems, TV sets, aooga horns, and such). I explore all the possibilities in Chapter 2.
- The `DATA DIVISION` is where your program stores things. The whole purpose of running a program is to have it fiddle with data, and this division is where your program keeps all the data. A lot of the data here is just stuff you work with and throw away when you are through with it. When you want some new data, you issue an order to have the data

brought in from disk, and it is delivered here. If you want to write data to a disk, this is where you put it to have it shipped out. You can think of the DATA DIVISION as your general data warehouse with a shipping and receiving department. I take a look at this division in Chapter 2, and the chapters in Part II of this book describe all the details.

- The PROCEDURE DIVISION is where the action takes place. If your program is going to do anything at all, this division is where it happens. Every program must have a PROCEDURE DIVISION, and every PROCEDURE DIVISION must have at least one paragraph. A paragraph is a bunch of sentences with a name at the top. A sentence is a COBOL command to make the computer do something. The preceding example has an empty paragraph — one without any sentences — named ParagraphName.

Because this example has nothing in the paragraph in the PROCEDURE DIVISION, when the program is run, it doesn't do anything. To make sure that was true, I poked the program into a computer and ran it. The test was a complete success — the program promptly did nothing. However, this program is very useful. You can use it as a sort of seed program to start editing your own COBOL program — just load it up and put in the parts that actually do stuff. In other words, this program is most useful because it does nothing at all.

A Small Program That Actually Does Something

I suppose it is now time to advance beyond the basic program that does nothing and get to one that actually does something. The following code shows an example of a program that doesn't only do something, it does something three times every time you run the program:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Siegfried.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  I PICTURE 99.
PROCEDURE DIVISION.
GET-OFF-MY-FOOT.
    PERFORM VARYING I FROM 1 BY 1 UNTIL I > 3
        DISPLAY "You're on my foot!".
```

Here you have a program named Siegfried that has one paragraph named GET-OFF-MY-FOOT. The PERFORM statement causes the DISPLAY statement to write a message to the screen three times. Having written the message, the program moves on, leaving this message on the screen for all to see:

```
You're on my foot!
You're on my foot!
You're on my foot!
```

In a COBOL program, the PROCEDURE DIVISION is made up of a collection of *paragraphs*. A paragraph can contain several *sentences*. Each COBOL sentence begins with a *verb* and ends with a period. Verbs are easy to spot. They are the action words like DISPLAY, MULTIPLY, PERFORM, and MOVE. You can also put verbs inside a sentence — for example, the DISPLAY verb in the preceding program appears inside a sentence that begins with the PERFORM verb — but a verb *always* appears at the beginning of a sentence.

Sentences are made out of *statements*. Any time you see a verb, it is at the beginning of a statement. If the statement ends with a period, the statement is also a sentence. A single sentence can be made up of more than one statement, meaning that sentences can contain more than one verb. The sentence in the preceding example contains two statements: One begins with PERFORM and the other begins with DISPLAY.

You may notice a similarity between the way the things are named in COBOL and the way things were named by your English teacher. This convention is no accident — the original framers of COBOL made a conscious effort to make the language as similar to English as possible. The idea was to make the language *self-documenting* (which means making the meaning of the code as evident, intuitive, and easy to understand as possible to a person new to computer lingo). This goal was admirable, but time has proven this English-like grammar to be of no real advantage in computer languages.

Making a Place to Put Things

For COBOL to do arithmetic, it must have a place to put the numbers. And for COBOL to keep track of things, it must have a place to store names and descriptions. When you do your own arithmetic with a pencil, you use a piece of paper. When you need to remember something, you jot it down. When COBOL does arithmetic, or stores something it needs to recall later, it uses WORKING-STORAGE the same way you use the paper.

Every time you need a place to put a number in your COBOL program, you just invent a name for the place and type the name and the description of the number into the WORKING-STORAGE area. After that, you can refer to it by name from anywhere in the PROCEDURE DIVISION. Handy, eh?

The following example demonstrates how to create an area of memory that holds numbers and how to do arithmetic:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ItFigures.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  Quantity PICTURE 999 COMPUTATIONAL.
PROCEDURE DIVISION.
ArithmeticDoneHere.
    MOVE 6 TO Quantity.
    ADD 4 TO Quantity.
    DIVIDE 2 INTO Quantity.
    DISPLAY Quantity.
```

In this example, I declare a number and name it `Quantity`. To *declare* something is to give it a name and assign it some space in the `WORKING-STORAGE` section of the program. That 01 number in front of `Quantity` has to do with its level — I tell you all about levels in Chapter 4. In COBOL, you can work with a bunch of different types and sizes of numbers. The number in the preceding example can be three digits long (as indicated by the `PICTURE` clause declaring it as 999) and it is `COMPUTATIONAL` (which means it is a special kind of number that does really quick arithmetic). I dedicate all of Part II in this book to explaining the different kinds of numbers and how you can put them in `WORKING-STORAGE`.

In the preceding example, you see some sentences in the `PROCEDURE DIVISION` that fiddle around with the number in `Quantity`. First, the `MOVE` verb is used to stuff a 6 into `Quantity`. The `ADD` verb is used to increase it by 4. The `DIVIDE` verb divides the number in `Quantity` in half. Finally, the `DISPLAY` verb is used to show you the result. When you run this program, it quickly performs all the calculations and proudly displays the following value:

```
5
```

You Have a Punched Card in Your Past

COBOL was originally designed to be punched onto cards. The cards were all exactly the same size and could hold a total of 80 characters. Each character position on the card was called a *column*. The character in each column was determined by the pattern of a vertical row of holes. The characters punched into these cards were then read into the computer and compiled.

Nobody uses punched cards any more — everybody use screens and keyboards (and sometimes a mouse) to create or modify their programs. This change has enabled programmers to achieve greater speed, a higher degree of accuracy, and carpal tunnel syndrome. However, the punched card legacy does hang around in COBOL, giving special meanings to certain column positions:

- Columns 1 through 6 are supposed to hold a *sequence number,* and column 7 is reserved for a *control character.* Just leave those columns blank — if your compiler is really picky and complains, go check out the rules in Chapter 3 and show the compiler who's boss.
- The next four columns (8 through 11) are known as *area A.* Only certain things can go into this area. You start the `DIVISION` and `SECTION` names in area A. You can also put `WORKING-STORAGE` data declarations and paragraph names in area A.
- Column 12 is the beginning of *area B,* which is where you put the verbs and sentences that make your program do stuff. Area B ends with column 72, but inside that range you are free to put things wherever you want. You can continue a statement from area B of one line right on into area B of the next line without telling COBOL — it looks there automatically.

Chapter 3 offers a detailed description of the card format, just in case your compiler is really stern about that sort of thing. Not all compilers adhere to these requirements, but I bring them up here because your compiler may require adherence to some part of them, and you may need to deal with them as you create your programs. The editors that come with compilers typically help you take care of most of this stuff automatically.

For the examples in this book, I leave out those seven columns on the left. They just contain a bunch of boring numbers that carry no information whatsoever. If those columns were included, all the examples would resemble the following one:

```
000001 IDENTIFICATION DIVISION.
000002 PROGRAM-ID. WetlandGrazer.
000003 ENVIRONMENT DIVISION.
000004 DATA DIVISION.
000010 WORKING-STORAGE SECTION.
000025 01  NumberOfMoose PICTURE 9999 COMPUTATIONAL.
000100 PROCEDURE DIVISION.
000105 ArithmeticDoneHere.
000110     MOVE 846 TO NumberOfMoose.
000112     DISPLAY NumberOfMoose.
```

The first six columns are the sequence numbers; the blank space between the number and the COBOL program is the control character column. Isn't that tacky? It may be traditional, but it is also polecat ugly. I could forgive it for being so absurd-looking if it had some purpose. Which it doesn't. It has been years since somebody spilled a card-punched COBOL program onto the floor and had to run it through a card sorter to put it back in order. That's right. That's the purpose of the numbers. Sorting the cards. Aren't you glad you know the truth now?

Going from What You See to What You Get

COBOL programming involves three steps:

1. Using a text editor of some kind to write the text of the program. The text is the sort of stuff that I talk about in this book — it's the form of the program that a human can read. You save this text as a file on the computer's disk. Most modern COBOL compilers supply a text editor.
2. Running the text of the program through the compiler. The compiler takes the program text and converts it to the bits-and-bytes format understood by the computer's hardware. If you want to think of some part of the programming process as being magic, this is the part. It's really not that big a deal, but the geeks and nerds who write compilers would like you to believe that it is difficult and mysterious.
3. Running the program.

This book is all about Step 1. The details involved with Steps 2 and 3 vary widely from computer to computer.

You can find all kinds of text editors. Some compilers supply super-duper, whiz-bang, graphical user interfaces that automatically color-code the parts of the COBOL language as you type them in. Some of these tools allow you to compile and run a program with a swift click of the mouse. The other extreme is a simple, barefooted text editor that saves the file to a disk and allows you to type in a command, naming the input and output files, that compiles the program.

Get Steps 2 and 3 out of the way now. These steps will be the same for every program that you write. If you don't know the process required to compile and run programs on your machine, there is no time like the present to learn how. Use some very simple program as a test case. Type it in and make it run. Choose a program that includes a `DISPLAY` statement so you can verify that it actually runs. Until you get these mechanics behind you, you can't concentrate on the details of writing programs.

Things to Consider While Programming in COBOL

Once you get started, you find that COBOL is pretty easy to work with. The odd truth is that COBOL's main advantage and its main disadvantage are the same thing: The language is easy and intuitive to read and write.

It's obvious why this feature is an advantage — folks can grab the basics of programming COBOL fairly quickly, and dive right into getting things done. The reason that the simplicity of the COBOL language is a disadvantage may be a bit more obscure.

You see, modern computing requires a lot of dynamic activity. Programs load and unload, change their characteristics according to their environment, dynamically allocate the space they need according to the task they are performing, and do a bunch of really weird, nonintuitive stuff. These things can make programming difficult in modern languages such as C++ and Java. Because COBOL was developed before all of these things became important, it doesn't have the same level of complexity as these more modern languages.

COBOL is a very static language — it doesn't change anything but the contents of some files and the values in `WORKING-STORAGE` while it is running. On the bright side, though, you don't usually need to do any of the really weird, nonintuitive stuff in COBOL. But when you do, COBOL has ways to get it done.

I want to pass on the best programming advice I ever received: Try it. (I'm not talking about breaking away from standard practices and procedures involved with writing good code.) Trust yourself. If you try something and it doesn't work, you will be working in the best Thomas Edison tradition. He tried thousands of light-bulb filaments before one of them lit.

Programmers always talk about "the portability of programs." Everybody wants to write a program and have it run on a bunch of different computers. This is one of the places where COBOL really shows its age — it is one of the least portable languages. It's kind of ironic, too, because one of the main purposes of the original COBOL design was the capability to write reusable code. It sure beat anything that came before it, so, to a large extent, it has succeeded — but not to the same extent as some of the more modern languages.

You need to consult the documentation of the compiler you are using. This book covers standard COBOL, but many places in the COBOL standard specification leave things flexible or undefined. This is true of all languages, but it is particularly true of COBOL. Throughout the book, when one of these open issues comes up, I point it out.

Chapter 2

The Anatomy of a COBOL Program

In This Chapter

- Exploring the basic structure of a COBOL program
- Displaying text on the screen
- Printing lines of text
- Sorting records in different ways

To help you understand the form of a COBOL program and how its parts interact, this chapter takes a close look at one relatively simple COBOL program. You can think of this chapter as a quick anatomy lesson. I lay a COBOL program on the table so you can examine its innards. If you are squeamish about anatomy, you can skip this chapter. In the chapters that follow this one, I examine all the parts of a COBOL program, but I carefully preserve the program parts. In this chapter, I rip the pieces right out of a complete, working program and hold them up for you to see. In the end, I put all the pieces back together and, as the program starts to run, I scream, "It's alive!"

The program you examine in this chapter first writes a bunch of names to a file. Each name has a number with it. A menu allows you to choose which action the program takes. You can display the names and numbers as an unsorted list, sorted by name, or sorted by number. You can also print the list. To perform all these tasks, the program must be able to display stuff on the screen, read from the keyboard, write to files, read from files, sort from one file to another, and print on the printer. All that, and it's pretty, too.

Program, Know Thyself: Looking at the IDENTIFICATION DIVISION

The sole purpose of the `IDENTIFICATION DIVISION` is to contain descriptive information about the program — other than assigning the program name, it doesn't participate in any of the program's activities. Like every COBOL program, the example program in this chapter starts with an `IDENTIFICATION DIVISION`:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SimpleSorterSample.
AUTHOR. Bertha D Blues.
DATE-WRITTEN. Long long ago.
DATE-COMPILED. Tuesday week.
SECURITY. Blanket.
```

Other than the name of the program, the IDENTIFICATION DIVISION doesn't contain much of anything. You can choose any program name you like. Heck, you can choose a name you don't like — just don't leave any spaces in the name and be sure you end it with a period.

The PROGRAM-ID is really the only thing you need in the IDENTIFICATION DIVISION. You may find some obsolete forms from time to time — mostly in older programs. Statements such as the following example don't mean much any more:

```
AUTHOR. Bertha D Blues.
```

An AUTHOR statement simply lists the name of the person, or persons, who perpetrated the program. Throughout the life of COBOL, it has been traditional for COBOL programmers to take credit here if the program is really, really good. This statement has also been a convenient place to blame somebody else for the lousy stuff. Its use has fallen into disfavor. I think you can understand why.

The following statements document the day that the program was completed, and the day it was compiled:

```
DATE-WRITTEN. Long long ago.
DATE-COMPILED. Tuesday week.
```

Both of these statements are really just comments and can hold anything. The compiler recognizes the DATE-COMPILED keyword; if you ask the compiler to output a listing of your program, the compiler inserts the current date in place of Tuesday week.

If you see a SECURITY entry in a COBOL listing, just fold it up and put it back where you got it:

```
SECURITY. Blanket.
```

You were wearing gloves, weren't you? This entry was designed to specify things like Secret, Top Secret, or If you read this, I will have to kill you. Programs like this should be read only by James Bond types. Even the programmer who wrote it is not allowed to read it. Rumor has it that this entry is also an obsolete COBOL form, but nobody knows that for sure. This rumor was spread by being printed in the COBOL 85 standard.

Creating a Safe ENVIRONMENT DIVISION

The ENVIRONMENT DIVISION is where you put descriptions of things that are external to the program — things like the kind of computer this program is supposed to run on and the names of files on the local disk:

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. ThisOne.
OBJECT-COMPUTER. ThisOne.
SPECIAL-NAMES.
    CURRENCY-SYMBOL IS "$".
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT BeanList ASSIGN TO "unsorted"
        ORGANIZATION IS SEQUENTIAL.
    SELECT Sorter ASSIGN TO "sortfile".
    SELECT SortedBeanList ASSIGN TO "sorted"
        ORGANIZATION IS SEQUENTIAL.
    SELECT PrintFile
        ASSIGN TO PRINTER
        ORGANIZATION IS LINE SEQUENTIAL.
```

The first things you can put in the CONFIGURATION SECTION of the ENVIRONMENT DIVISION are a couple of optional entries. One is the SOURCE-COMPUTER — the computer on which the program is being compiled. The other is the OBJECT-COMPUTER — the computer on which the compiled program is supposed to run. These names don't have any special meaning; these are just documentation comments. You can call your computer anything you want to. Be nice.

The SPECIAL-NAMES paragraph is, as you may suspect, for stuff that is a little special:

```
SPECIAL-NAMES.
    CURRENCY-SYMBOL IS "$".
```

In this example, I simply set the CURRENCY-SYMBOL to the dollar sign. This setting is odd for two reasons. First, no currency symbols are used in this program and, second, the dollar sign is the default anyway. You find many useful entries for the SPECIAL-NAMES paragraph, I just didn't need any for this example. Probably the most common use is for defining collating sequences for sorting. Chapter 16 includes an example of this use.

The FILE-CONTROL paragraph of the INPUT-OUTPUT SECTION contains a SELECT statement for each file that the COBOL program accesses:

```
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT BeanList ASSIGN TO "unsorted"
        ORGANIZATION IS SEQUENTIAL.
    SELECT Sorter ASSIGN TO "sortfile".
    SELECT SortedBeanList ASSIGN TO "sorted"
        ORGANIZATION IS SEQUENTIAL.
    SELECT PrintFile
        ASSIGN TO PRINTER
        ORGANIZATION IS LINE SEQUENTIAL.
```

You can think of a SELECT statement as a name map. It attaches an internal name (always a name that you make up) to an external file or device name (sometimes one that you make up). For example, the first SELECT statement in the list defines the name BeanList (a name used inside the program) as being associated with the name unsorted (the actual name of the file on the computer disk). The last SELECT statement in the list assigns the name PrintFile to the system printer. Anything written to PrintFile goes directly to the printer.

This mapping of names means that any references to the names of external files or devices appear only in the FILE-CONTROL paragraph of your program's INPUT-OUTPUT SECTION. You can easily change the name of a file that your program uses, because you know that the name appears in only one place: within the FILE-CONTROL paragraph, in the ASSIGN clause of a SELECT statement. It is best to use descriptive names in the SELECT — names that describe the data instead of the file holding the data. For example, you can use names like MonthEndSummary or SalesPerformance. The filenames change easily — the data does not.

Some ORGANIZATION stuff also goes along with each file. The organization describes the way in which the program accesses the file. In this example, the organization is SEQUENTIAL, which means that the files are written or read only in front-to-back, sequential order. I describe this and other file organizations in detail in Part IV of this book.

For the most part, a file is just a huge blob of bytes that the program must interpret before it has any meaning. You can use the SELECT statement to help with this interpretation. Your program can look at a single file in different ways — for example, text can be thought of as a collection of complete sentences or as just a simple stream of characters. If you need to look at a file in more than one way, just map two names to it with a pair of SELECT statements. I have lots more to say about all this in Part IV of this book.

Stashing Stuff in the DATA DIVISION

The DATA DIVISION is where a COBOL program stores and retrieves information. You may have heard of GIGO (garbage in, garbage out). Well, this division is where the garbage is kept. In fact, it is where everything is kept.

When you come right down to it, the sole purpose of a COBOL program is to manipulate the data that is sitting in the DATA DIVISION. The program can store names and numbers, do arithmetic, compare one data item to another, and write a record to disk and read it back — all in the DATA DIVISION.

The DATA DIVISION contains two sections. The first one specializes in data records that are read from and written to files. The second one is a local workspace internal to this program. They are laid out this way:

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

I fill in all the details about the FILE SECTION and the WORKING-STORAGE SECTION in the following sections of this chapter.

Talking to disk — the FILE SECTION

The first section in the DATA DIVISION is the FILE SECTION. This section is required if your program is going to do any disk I-O. Here's the FILE SECTION for this chapter's sample program:

```
FILE SECTION.
FD BeanList RECORD CONTAINS 16 CHARACTERS.
01  BeanListData.
    05  BeanOwner PIC X(12).
    05  BeanCount PIC ZZZ9.
SD Sorter RECORD CONTAINS 16 CHARACTERS.
01  SorterData.
    05  BeanOwner PIC X(12).
    05  BeanCount PIC ZZZ9.
FD SortedBeanList RECORD CONTAINS 16 CHARACTERS.
01  SortedBeanListData.
    05  BeanOwner PIC X(12).
    05  BeanCount PIC ZZZ9.
FD  PrintFile.
01  PrintLine PIC X(80).
```

The FILE SECTION contains one entry for each file or device that will be used for input or output during the run of the program. A file description begins with the keyword FD, which stands for either fuzzy dog, file description, fine day, or frilly doily — take your pick. The FD being defined in this program is given the name BeanList and has 16 characters in each of its records.

A record is one read-write unit. If your wallet can be considered a file, a record would be a bill in your wallet. The value of each bill could be different — it could be $5, $10, $20, or whatever — but all the bills are the same size and they fit nicely into the wallet. That's the way fixed-length records fit into a file. They can all contain different data, but they have the same form and size. Such a thing as a *variable-length record* also exists, but it isn't in this example and I don't want to talk about it right now. You can find everything you want to know about variable-length records in Part IV of this book.

The FD is immediately followed by an 01 entry that has the name BeanListData next to it. This entry defines the data record — a physical place in computer memory that the program can use to store data to be written to the file and to retrieve data that has been read from the file.

The 01 entry defines the exact form of the 16-byte records in the file. The record uses 12 characters for the BeanOwner — the name of the person who owns some beans — and 4 characters for BeanCount — the number of beans that are owned by this person. That X(12) data format just creates a spot in memory where any 12 characters can be stored. The ZZZ9 data format is for storage of a four-digit number that causes all leading zeroes to be changed to blanks. The capability to format data this way is one of the strengths of COBOL — I describe this capability in Chapters 4 and 5.

Immediately after the first field description, the program has an SD entry:

```
SD Sorter RECORD CONTAINS 16 CHARACTERS.
01  SorterData.
    05  BeanOwner PIC X(12).
    05  BeanCount PIC ZZZ9.
```

An SD is very similar to an FD, except that an SD defines a temporary work file that is to be created and used by COBOL's SORT verb as a work area. In this program, the data from BeanList passes through Sorter, which the program uses as temporary storage for sorting the records into the specified order.

A sort-file description begins with the keyword SD, which stands for either stupid dog, sort description, sad day, or soiled doily — take your pick. Like the FD entry, this SD also states that the records each contain 16 characters.

Also like the FD, an 01-level record definition immediately following it lays out the details of a 16-character record. In fact, the only thing different is the name of the record itself — the fields, BeanOwner and BeanCount, are identical to the other record. It's okay to have the same names on things inside a record, just as long as the names of the records themselves are different.

Following the SD entry, the program has another FD that has the same structure as the one defined for BeanList:

```
FD SortedBeanList RECORD CONTAINS 16 CHARACTERS.
01  SortedBeanListData.
    05  BeanOwner PIC X(12).
    05  BeanCount PIC ZZZ9.
```

The program reads from the BeanList file, passes stuff through Sorter, and deposits the resulting sorted file here. This file can then be read to retrieve the data in sorted order.

The final FD entry in the FILE SECTION differs from the other two:

```
FD  PrintFile.
01  PrintLine PIC X(80).
```

This one is intended for printing, as you can see by the last SELECT statement in the FILE-CONTROL paragraph of the program's INPUT-OUTPUT SECTION (which I discuss in a previous section of this chapter). What COBOL provides here is a printer that can be opened just like a file. No record layout is required for the printer because it normally needs to be able to print things that are formatted all sorts of different ways. The X(80) data format defines the printer as being able to print any old 80-character line you care to pass to it.

Pigeonholing data in WORKING-STORAGE

The WORKING-STORAGE SECTION is where you put all the stuff you want to work with. It is for holding names and numbers that your program can use while it does whatever it does. You can think of WORKING-STORAGE as your own personal closet space. Whenever you need to stash something, just make room for it in this section and cram it right in. This example program uses very little working storage, but COBOL programs often have pages and pages of the stuff.

Here's the WORKING-STORAGE SECTION for the example program:

```
WORKING-STORAGE SECTION.
77  Response PIC X VALUE " ".
    88  DisplayUnsorted       VALUE "1".
    88  DisplayNameSorted     VALUE "2".
    88  DisplayNumberSorted   VALUE "3".
    88  PrintUnsorted         VALUE "4".
    88  PrintNameSorted       VALUE "5".
    88  PrintNumberSorted     VALUE "6".
    88  QuitProgram           VALUE "q", "Q".
77  FileFlag PIC X.
    88  EndOfFile VALUE "E".
01  Heading PIC X(80).
```

This program is controlled by a menu that puts up a list of options and waits for the user to enter a choice. The program is menu controlled in the sense that it displays a menu and won't do anything else until a response from the menu commands it to take some action. The character entered by the user goes into `Response` — `PIC X` reserves space to hold one character. All those names with 88 on them are tags for the values that mean something special to the program. The menu selections that perform some action have the characters `1` through `6` assigned to them. The last entry under `Response` — the one that causes the program to exit — has both `q` and `Q` assigned to it, so the program doesn't care whether the user's big fat thumb has hit the Caps Lock key.

The sample program's `WORKING-STORAGE SECTION` uses another one of those `PIC X` things to indicate when the end of a file has been reached:

```
77  FileFlag PIC X.
    88  EndOfFile VALUE "E".
```

It's like a secret handshake — everybody involved knows the magic letter to stick into `FileFlag`. Whenever the end of a file comes up, the letter `E` is put into the `FileFlag` and tests can be made in other locations to see whether the `EndOfFile` has arrived.

The `WORKING-STORAGE SECTION` also defines something called `Heading`:

```
01  Heading PIC X(80).
```

Nothing fancy here. `Heading` is just a place to put stuff that needs to be printed as the heading of the data sent to the printer.

Going to Work in the PROCEDURE DIVISION

Paydirt. Everything that comes before the PROCEDURE DIVISION is just the necessary skeleton and configuration — this division is where the action is. This part of the program is filled with verbs. Only the actions taken by COBOL verbs bring a program to life and make it do things.

Here's the beginning of the PROCEDURE DIVISION for the sample program:

```
PROCEDURE DIVISION.
Mainline.
    PERFORM CreateList.
    PERFORM MenuLoop UNTIL QuitProgram.
    STOP RUN.
```

The PROCEDURE DIVISION is followed immediately by a paragraph name. COBOL verbs are used to construct COBOL sentences. All the COBOL sentences are grouped into paragraphs. All paragraphs start with a paragraph name. Whenever a COBOL program starts to run, it starts with the first paragraph in the PROCEDURE DIVISION.

The first paragraph of this example is named Mainline. The first sentence in the paragraph begins with the verb PERFORM:

```
    PERFORM CreateList.
```

The PERFORM verb in this example has the paragraph name CreateList as its option. This verb is an instruction to the COBOL program to go to the CreateList paragraph, execute every sentence in it, and return right back here. So that's what the program does. And when it returns, it goes to the next sentence:

```
    PERFORM MenuLoop UNTIL QuitProgram.
```

Another PERFORM verb. Once again, it has a paragraph name — MenuLoop — as its option. The COBOL program goes to the MenuLoop paragraph, executes every sentence in it, and returns right back to this PERFORM statement. But this PERFORM also has an UNTIL option.

The UNTIL verb checks the true or false condition of QuitProgram to determine whether it should move on or PERFORM MenuLoop again. If QuitProgram is not true, the PERFORM statement executes every statement

in the MenuLoop paragraph again. Each time the program completes MenuLoop and returns to this PERFORM, the status of QuitProgram is tested again. Whenever QuitProgram becomes true (that is, the letter Q or q is found in Response), the MenuLoop paragraph is not performed again and things proceed to the next sentence:

```
    STOP RUN.
```

The verb in this sentence is STOP. The option is RUN. This command tells the COBOL program to quit running — so it does. As a matter of fact, not running is one of COBOL's best tricks. As you become more and more familiar with COBOL, you find that failing to proceed is something it can do very well. The COBOL language includes many verbs, all of which fail at one time or another, except this one. It works.

COBOL gives you two ways to end a paragraph; neither one has a lot of pomp and circumstance to it. If a paragraph is the last one in the program, it ends the same way the program does. End of text; end of program; end of paragraph. The other way to end a paragraph is to start a new one. In this example program, the Mainline paragraph ends when the CreateList paragraph starts:

```
* Create a new file and write several records to it.
 CreateList.
     OPEN OUTPUT BeanList.
     MOVE "Alley" TO BeanOwner OF BeanListData.
     MOVE 87 TO BeanCount OF BeanListData.
     WRITE BeanListData.
     MOVE "Umpa" TO BeanOwner OF BeanListData.
     MOVE 341 TO BeanCount OF BeanListData.
     WRITE BeanListData.
     MOVE "Guz" TO BeanOwner OF BeanListData.
     MOVE 12 TO BeanCount OF BeanListData.
     WRITE BeanListData.
     MOVE "Foozy" TO BeanOwner OF BeanListData.
     MOVE 118 TO BeanCount OF BeanListData.
     WRITE BeanListData.
     MOVE "Ooola" TO BeanOwner OF BeanListData.
     MOVE 212 TO BeanCount OF BeanListData.
     WRITE BeanListData.
     MOVE "Wunmug" TO BeanOwner OF BeanListData.
     MOVE 88 TO BeanCount OF BeanListData.
     WRITE BeanListData.
     MOVE "Guz" TO BeanOwner OF BeanListData.
     MOVE 233 TO BeanCount OF BeanListData.
     WRITE BeanListData.
```

```
MOVE "Oscar" TO BeanOwner OF BeanListData.
MOVE 67 TO BeanCount OF BeanListData.
WRITE BeanListData.
MOVE "Dinny" TO BeanOwner OF BeanListData.
MOVE 891 TO BeanCount OF BeanListData.
WRITE BeanListData.
CLOSE BeanList.
```

The `CreateList` paragraph in the preceding code is the first one called from the `Mainline` paragraph when the program starts running. `CreateList` writes a collection of names and numbers to the `BeanList` file. The data stored in this file is used by the rest of the program to generate output. Normally, a COBOL program gets its data from some other location, such as a disk file or user input, but I did it this way to create a stand-alone example.

The `CreateList` paragraph begins with an `OPEN` verb that has the `OUTPUT` option and the name of the file to be opened:

```
OPEN OUTPUT BeanList.
```

This statement creates a new file; if the file already exists, this statement sets up the file to be overwritten.

As shown in the following excerpt, `CreateList` includes two `MOVE` statements and a `WRITE` for each record that goes out to the file:

```
MOVE "Alley" TO BeanOwner OF BeanListData.
MOVE 87 TO BeanCount OF BeanListData.
WRITE BeanListData.
```

Comment now or they'll call you later

I want to say something about the following line, which appears right before the `CreateList` paragraph:

```
* Create a new file and write
  several records to it.
```

A line of code that has an asterisk in column 7 is known as a *comment*. You can write anything you want on a line that has an asterisk in column 7 — anything. The compiler completely ignores whatever you put on that line. This type of line contains stuff that is strictly for human consumption. Considerate programmers use comment lines to explain the programs they write. Inconsiderate programmers don't write comments in their programs.

The MOVE statements put the name and number in the record to be written, and the WRITE sends the record out to the file. At the end of the paragraph, when all the records have been written, the CLOSE verb is used to clean up behind all the WRITE verbs and disassociate the program from the newly created file:

```
    CLOSE BeanList.
```

Anytime a program does reading or writing of a file, you see an OPEN-READ-CLOSE or OPEN-WRITE-CLOSE trio of verbs. They may be in separate paragraphs and seem unrelated, but, unless the program has a bug, they are always there.

After Mainline calls CreateList, the program has data to be processed. Mainline processes the data by calling MenuLoop to ask the user what to do. Take a look at the code from MenuLoop:

```
* Display a menu and act on the request from the user.
 MenuLoop.
     DISPLAY "".
     DISPLAY "    1  Display unsorted."
     DISPLAY "    2  Display sorted by name."
     DISPLAY "    3  Display sorted by number."
     DISPLAY "    4  Print unsorted."
     DISPLAY "    5  Print sorted by name."
     DISPLAY "    6  Print sorted by number."
     DISPLAY "    Q  Quit program."
     DISPLAY "     " WITH NO ADVANCING.
     ACCEPT Response.
     IF DisplayUnsorted
         PERFORM ShowUnsortedList
     ELSE IF DisplayNameSorted
         PERFORM ShowSortedByName
     ELSE IF DisplayNumberSorted
         PERFORM ShowSortedByNumber
     ELSE IF PrintUnsorted
         PERFORM PrintUnsortedList
     ELSE IF PrintNameSorted
         PERFORM PrintSortedByName
     ELSE IF PrintNumberSorted
         PERFORM PrintSortedByNumber.
```

This paragraph begins with a stream of DISPLAY statements. Each statement displays a single line of text on the screen. After each line displays, the screen looks like this:

```
    1  Display unsorted.
    2  Display sorted by name.
```

```
3  Display sorted by number.
4  Print unsorted.
5  Print sorted by name.
6  Print sorted by number.
Q  Quit program.
  _
```

Each line is indented by the four spaces that appear with the text in the DISPLAY statements. The last DISPLAY statement (the one with the WITH NO ADVANCING on it) displays a few space characters, but does not move to the next line when it finishes. The ACCEPT statement (which waits for the user to enter a letter or number from the keyboard) picks up where the DISPLAY statements leave off and waits with the cursor positioned beneath the last line of the menu.

The ACCEPT statement takes the user's entry and uses it to set the value of Response. For example, if the user enters 1, the program sets the value of Response to DisplayUnsorted. (As I discuss earlier in this chapter, the program's WORKING STORAGE section defines the Response value associated with each user entry.) Using a series of conditional statements that begins with the following lines, MenuLoop evaluates Response and determines which paragraph the program should PERFORM next:

```
IF DisplayUnsorted
   PERFORM ShowUnsortedList
```

Whether or not a paragraph is performed, the UNTIL clause on the following PERFORM statement in Mainline tests the Response value to decide whether to call MenuLoop again to display the menu or just quit:

```
PERFORM MenuLoop UNTIL QuitProgram.
```

The ShowUnsortedList paragraph displays all the names and numbers from BeanList without sorting them:

```
* Read the list and display it without sorting.
 ShowUnsortedList.
     OPEN INPUT BeanList.
     MOVE SPACE TO FileFlag.
     PERFORM UNTIL EndOfFile
         READ BeanList
             AT END MOVE "E" TO FileFlag
             NOT AT END DISPLAY BeanOwner OF BeanListData
                                BeanCount OF BeanListData
         END-READ
     END-PERFORM.
     CLOSE BeanList.
```

In the preceding example, the BeanList file (the one that contains the list of names and numbers) is opened for INPUT. The FileFlag is used to determine when the end of the file has been reached, so it must be cleared first to prevent some old leftover value from giving a false reading.

The PERFORM in this paragraph does not have a paragraph name, which means the program is going to perform something right here — in this case, the READ statement. The PERFORM has an UNTIL option on it, so the program will READ again and again until the EndOfFile condition has been set.

The END-READ and END-PERFORM keywords are used to structure the code by terminating the verbs READ and PERFORM, respectively. Many COBOL verbs have these END-whatsit keywords to help you structure your code.

Notice the OPEN-READ-CLOSE pattern. A file cannot be read until it has been opened, and it should be closed when the reading is done. The displayed output looks like this:

```
Alley          87
Umpa          341
Guz            12
Foozy         118
Ooola         212
Wunmug         88
Guz           233
Oscar          67
Dinny         891
```

In the ShowUnsortedList paragraph, this DISPLAY statement is used to display the name and count of each bean owner:

```
                NOT AT END DISPLAY BeanOwner OF BeanListData
                                   BeanCount OF BeanListData
```

The references are qualified as being BeanOwner OF BeanListData and BeanCount OF BeanListData. You must include an OF statement in the code in order to tell COBOL which BeanOwner you are referring to. The DATA DIVISION defines more than one BeanOwner and it is necessary to tell COBOL which you mean. It's sort of like having a bunch of guys named Joe. When you yell for one of them, you have to say "Joe Smith" or "Joe Bflspk" or whatever.

The PrintUnsortedList paragraph is very similar to ShowUnsortedList:

```
* Read the list and print it without sorting.
 PrintUnsortedList.
     OPEN INPUT BeanList.
     OPEN OUTPUT PrintFile.
     MOVE SPACE TO FileFlag.
     PERFORM UNTIL EndOfFile
         READ BeanList
             AT END MOVE "E" TO FileFlag
             NOT AT END WRITE PrintLine FROM BeanListData
         END-READ
     END-PERFORM.
     CLOSE BeanList.
     CLOSE PrintFile.
```

The only difference is that `PrintUnsortedList` prints and `ShowUnsortedList` displays. To be able to print, the program must open the printer. For this paragraph, two files are open. The one that's open for `OUTPUT` is the printer.

This paragraph has both the `OPEN-READ-CLOSE` sequence and the `OPEN-WRITE-CLOSE` sequence. Also, the `FROM` option on the `WRITE` verb is a convenience — without it, you would need to `MOVE` the data from `BeanListData` to `PrintLine` before printing it.

Take note of the relationships among the next few paragraphs. Some of this code sorts by name and some code sorts by numbers. These paragraphs also include code to display the list and code to print the list. The same sorting paragraphs are used for both displaying and printing. The same display paragraph is used for the different sorting orders. This type of code structure is known as *modular programming.* A single paragraph (or group of paragraphs) is designed to do one very specific task. It can be very useful to write paragraphs in such a way that they can serve the same purpose in different circumstances. Doing so requires a clear definition of the purpose of each paragraph.

The paragraph `ShowSortedByName` is called directly from a menu selection. This paragraph contains just two `PERFORM` statements:

```
* Display the list sorted by the names.
 ShowSortedByName.
     PERFORM SortByName.
     PERFORM ShowSortedList.
```

The SortByName paragraph does the sorting and the ShowSortedList does the displaying. The resulting display of the list sorted by names looks like this:

```
Alley          87
Dinny         891
Foozy         118
Guz            12
Guz           233
Ooola         212
Oscar          67
Umpa          341
Wunmug         88
```

The next paragraph, very similar to the previous one, does the same thing except that it uses a different sort order:

```
* Display the list sorted by the numbers.
ShowSortedByNumber.
    PERFORM SortByNumber.
    PERFORM ShowSortedList.
```

As I discuss a bit later in this chapter, the SortByName and SortByNumber paragraphs each include a keyword that specifies whether the list is sorted in ascending or descending order. The output from ShowSortedByNumber looks like this:

```
Dinny         891
Umpa          341
Guz           233
Ooola         212
Foozy         118
Wunmug         88
Alley          87
Oscar          67
Guz            12
```

The two previous paragraphs each display the list on the screen. The two following paragraphs each print the list to the printer:

```
* Print the list sorted by the names.
 PrintSortedByName.
     PERFORM SortByName.
     MOVE "Sorted by name..." TO Heading.
     PERFORM PrintSortedList.
```

```
* Print the list sorted by the numbers.
 PrintSortedByNumber.
     PERFORM SortByNumber.
     MOVE "Sorted by number..." TO Heading.
     PERFORM PrintSortedList.
```

The two paragraphs (`PrintSortedByName` and `PrintSortedByNumber`) are very similar to one another, as well as similar to the two preceding paragraphs that display the lists. They are all called directly from menu selections. They each have two `PERFORM` statements — one does the sorting and the other does the printing. Also, each paragraph uses a `MOVE` statement to move a string into the `Heading`. This string is used by `PrintSortedList` to put a heading line on the output.

The following paragraph displays the list from the sort file. The sorting order doesn't matter — this paragraph just displays the records in the order that it finds them:

```
* Display the list from the sorted bean list
 ShowSortedList.
     OPEN INPUT SortedBeanList.
     MOVE SPACE TO FileFlag.
     PERFORM UNTIL EndOfFile
         READ SortedBeanList
             AT END MOVE "E" TO FileFlag
             NOT AT END DISPLAY
                 BeanOwner OF SortedBeanListData
                 BeanCount OF SortedBeanListData
         END-READ
     END-PERFORM.
     CLOSE SortedBeanList.
```

Whenever the preceding paragraph is performed, the list has already been sorted and stored in the file `SortedBeanList`. This paragraph opens the file for `INPUT` and executes a `PERFORM` loop until the end of file is reached. Inside the loop, a record is read from the file and (if the end of file has not been reached) the `DISPLAY` statement displays one line containing the `BeanOwner` and `BeanCount` of `SortedBeanListData`. When the end of file has been reached, the `PERFORM` loop terminates, and the `SortedBeanList` is closed.

The paragraph `PrintSortedList` prints the list in much the same way as `ShowSortedList` displays it:

```
* Print the list from the sorted bean list
 PrintSortedList.
     OPEN INPUT SortedBeanList.
     OPEN OUTPUT PrintFile.
     MOVE SPACE TO FileFlag.
     WRITE PrintLine FROM Heading.
     PERFORM UNTIL EndOfFile
         READ SortedBeanList
             AT END MOVE "E" TO FileFlag
             NOT AT END
                 WRITE PrintLine FROM SortedBeanListData
         END-READ
     END-PERFORM.
     CLOSE PrintFile.
     CLOSE SortedBeanList.
```

In the preceding code, however, it is necessary to `OPEN` the printer before entering the loop and to `WRITE` to the printer using the data `FROM` the record of data that was read from the `SortedBeanList` file. At the end of this paragraph, the program must `CLOSE` both the file and the printer.

You must be careful to close everything you open. The `CLOSE` does more than just disconnect your program from a file or a device — it also cleans up things and makes sure the disconnection is tidy and no unwritten data is left hanging around. On many computer systems, your program has something known as a *file limit*. That is, your program can hold open only a certain number of files at any one time. After this number is reached, you cannot open another file until you close one. The file limit is normally quite large, but a runaway program can open the same file a number of times and easily exceed the limit.

At the very bottom of the program are the two paragraphs that sort the data:

```
* Sort the list by the names
 SortByName.
     SORT Sorter
         ON ASCENDING KEY BeanOwner OF SorterData
         USING BeanList
         GIVING SortedBeanList.

* Sort the list by the numbers
 SortByNumber.
     SORT Sorter
         ON DESCENDING KEY BeanCount OF SorterData
         USING BeanList
         GIVING SortedBeanList.
```

COBOL has a built-in sort facility in the form of a SORT verb. SORT uses three files. The input is read from the USING file. The sorted output is written to the GIVING file. SORT uses the third file to do its work. This work file, also called the *sort file,* is the one named Sorter. Sorter is declared as an SD in the program's FILE SECTION. Just like any other file, it has a record associated with it. The names from this record are specified as options to the SORT verb to specify the keys used during the sort.

A program can sort things in two ways, and the preceding code example includes one of each. An ASCENDING sort puts small things first and big things last — a DESCENDING sort does just the opposite.

Are you ready for an oversimplification? Okay. The sorting process goes something like this:

1. Records are read from BeanList and written to Sorter.
2. The SORT verb examines the records in Sorter and swaps them around until they are all in order.
3. The records are then read from Sorter and written to SortedBeanList.

All Together Now

This chapter presents an explanation of the program SimpleSorterSample by cutting it apart and holding its various parts up to the light. The following code shows the program all together in one place, so you can get an idea of its size and its organization and see all its parts in one place at one time:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SimpleSorterSample.
AUTHOR. Bertha D Blues.
DATE-WRITTEN. Long long ago.
DATE-COMPILED. Tuesday week.
SECURITY. Blanket.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. ThisOne.
OBJECT-COMPUTER. ThisOne.
SPECIAL-NAMES.
    CURRENCY-SYMBOL IS "$".
INPUT-OUTPUT SECTION.
```

(continued)

(continued)

```
FILE-CONTROL.
    SELECT BeanList ASSIGN TO "unsorted"
        ORGANIZATION IS SEQUENTIAL.
    SELECT Sorter ASSIGN TO "sortfile".
    SELECT SortedBeanList ASSIGN TO "sorted"
        ORGANIZATION IS SEQUENTIAL.
    SELECT PrintFile
        ASSIGN TO PRINTER
        ORGANIZATION IS LINE SEQUENTIAL.
DATA DIVISION.
FILE SECTION.
FD  BeanList RECORD CONTAINS 16 CHARACTERS.
01  BeanListData.
    05  BeanOwner PIC X(12).
    05  BeanCount PIC ZZZ9.
SD  Sorter RECORD CONTAINS 16 CHARACTERS.
01  SorterData.
    05  BeanOwner PIC X(12).
    05  BeanCount PIC ZZZ9.
FD  SortedBeanList RECORD CONTAINS 16 CHARACTERS.
01  SortedBeanListData.
    05  BeanOwner PIC X(12).
    05  BeanCount PIC ZZZ9.
FD  PrintFile.
01  PrintLine PIC X(80).
WORKING-STORAGE SECTION.
77  Response PIC X VALUE " ".
    88  DisplayUnsorted      VALUE "1".
    88  DisplayNameSorted    VALUE "2".
    88  DisplayNumberSorted  VALUE "3".
    88  PrintUnsorted        VALUE "4".
    88  PrintNameSorted      VALUE "5".
    88  PrintNumberSorted    VALUE "6".
    88  QuitProgram          VALUE "q", "Q".
77  FileFlag PIC X.
    88  EndOfFile VALUE "E".
01  Heading PIC X(80).
PROCEDURE DIVISION.
Mainline.
    PERFORM CreateList.
    PERFORM MenuLoop UNTIL QuitProgram.
    STOP RUN.
```

```
* Create a new file and write several records to it.
 CreateList.
     OPEN OUTPUT BeanList.
     MOVE "Alley" TO BeanOwner OF BeanListData.
     MOVE 87 TO BeanCount OF BeanListData.
     WRITE BeanListData.
     MOVE "Umpa" TO BeanOwner OF BeanListData.
     MOVE 341 TO BeanCount OF BeanListData.
     WRITE BeanListData.
     MOVE "Guz" TO BeanOwner OF BeanListData.
     MOVE 12 TO BeanCount OF BeanListData.
     WRITE BeanListData.
     MOVE "Foozy" TO BeanOwner OF BeanListData.
     MOVE 118 TO BeanCount OF BeanListData.
     WRITE BeanListData.
     MOVE "Ooola" TO BeanOwner OF BeanListData.
     MOVE 212 TO BeanCount OF BeanListData.
     WRITE BeanListData.
     MOVE "Wunmug" TO BeanOwner OF BeanListData.
     MOVE 88 TO BeanCount OF BeanListData.
     WRITE BeanListData.
     MOVE "Guz" TO BeanOwner OF BeanListData.
     MOVE 233 TO BeanCount OF BeanListData.
     WRITE BeanListData.
     MOVE "Oscar" TO BeanOwner OF BeanListData.
     MOVE 67 TO BeanCount OF BeanListData.
     WRITE BeanListData.
     MOVE "Dinny" TO BeanOwner OF BeanListData.
     MOVE 891 TO BeanCount OF BeanListData.
     WRITE BeanListData.
     CLOSE BeanList.

* Display a menu and act on the request from the user.
 MenuLoop.
     DISPLAY "".
     DISPLAY "    1  Display unsorted."
     DISPLAY "    2  Display sorted by name."
     DISPLAY "    3  Display sorted by number."
     DISPLAY "    4  Print unsorted."
     DISPLAY "    5  Print sorted by name."
     DISPLAY "    6  Print sorted by number."
     DISPLAY "    Q  Quit program."
     DISPLAY "     " WITH NO ADVANCING.
     ACCEPT Response.
```

(continued)

(continued)

```
       IF DisplayUnsorted
           PERFORM ShowUnsortedList
       ELSE IF DisplayNameSorted
           PERFORM ShowSortedByName
       ELSE IF DisplayNumberSorted
           PERFORM ShowSortedByNumber
       ELSE IF PrintUnsorted
           PERFORM PrintUnsortedList
       ELSE IF PrintNameSorted
           PERFORM PrintSortedByName
       ELSE IF PrintNumberSorted
           PERFORM PrintSortedByNumber.

   * Read the list and display it without sorting.
    ShowUnsortedList.
       OPEN INPUT BeanList.
       MOVE SPACE TO FileFlag.
       PERFORM UNTIL EndOfFile
           READ BeanList
               AT END MOVE "E" TO FileFlag
               NOT AT END DISPLAY BeanOwner OF BeanListData
                                  BeanCount OF BeanListData
           END-READ
       END-PERFORM.
       CLOSE BeanList.

   * Read the list and print it without sorting.
    PrintUnsortedList.
       OPEN INPUT BeanList.
       OPEN OUTPUT PrintFile.
       MOVE SPACE TO FileFlag.
       PERFORM UNTIL EndOfFile
           READ BeanList
               AT END MOVE "E" TO FileFlag
               NOT AT END WRITE PrintLine FROM BeanListData
           END-READ
       END-PERFORM.
       CLOSE BeanList.
       CLOSE PrintFile.

   * Display the list sorted by the names.
    ShowSortedByName.
       PERFORM SortByName.
       PERFORM ShowSortedList.
```

```
* Display the list sorted by the numbers.
 ShowSortedByNumber.
     PERFORM SortByNumber.
     PERFORM ShowSortedList.

* Print the list sorted by the names.
 PrintSortedByName.
     PERFORM SortByName.
     MOVE "Sorted by name..." TO Heading.
     PERFORM PrintSortedList.

* Print the list sorted by the numbers.
 PrintSortedByNumber.
     PERFORM SortByNumber.
     MOVE "Sorted by number..." TO Heading.
     PERFORM PrintSortedList.

* Display the list from the sorted bean list
 ShowSortedList.
     OPEN INPUT SortedBeanList.
     MOVE SPACE TO FileFlag.
     PERFORM UNTIL EndOfFile
         READ SortedBeanList
             AT END MOVE "E" TO FileFlag
             NOT AT END DISPLAY
                 BeanOwner OF SortedBeanListData
                 BeanCount OF SortedBeanListData
         END-READ
     END-PERFORM.
     CLOSE SortedBeanList.

* Print the list from the sorted bean list
 PrintSortedList.
     OPEN INPUT SortedBeanList.
     OPEN OUTPUT PrintFile.
     MOVE SPACE TO FileFlag.
     WRITE PrintLine FROM Heading.
     PERFORM UNTIL EndOfFile
         READ SortedBeanList
             AT END MOVE "E" TO FileFlag
             NOT AT END
                 WRITE PrintLine FROM SortedBeanListData
         END-READ
     END-PERFORM.
     CLOSE PrintFile.
```

(continued)

(continued)

```
      CLOSE SortedBeanList.

* Sort the list by the names
 SortByName.
      SORT Sorter
          ON ASCENDING KEY BeanOwner OF SorterData
          USING BeanList
          GIVING SortedBeanList.

* Sort the list by the numbers
 SortByNumber.
      SORT Sorter
          ON DESCENDING KEY BeanCount OF SorterData
          USING BeanList
          GIVING SortedBeanList.
```

Chapter 3

COBOL Mechanics — A Look under the Hood

In This Chapter

- Understanding the characters and cases of COBOL
- Exploring the meaning of punctuation in COBOL
- Recognizing the words that COBOL reserves for special purposes
- Formatting a line of COBOL code

This chapter describes some of the mechanics you need to use when constructing a COBOL program. A COBOL program is made up of words surrounded by punctuation and shrewdly placed bunches of blank spaces. Some of the words are already defined by COBOL to have a special meaning — other words you make up as you go along. COBOL can get really cranky about how you format all the words and punctuation. This chapter describes the various moods and attitudes COBOL assumes as it reads your code.

The COBOL Cast of Characters

You can use lots of characters in a COBOL program, but COBOL doesn't even know that some characters exist. For example, COBOL has no place for the ! symbol (known as the *darnit*), or the # symbol (known as the *gridlet*), or even the @ symbol (known as the *atlet*). Don't be too disappointed, though, because you can still put all these characters inside quoted strings and display them on the screen or print them on paper — they are just not part of the COBOL language itself.

Table 3-1 lists all the characters that are officially sanctioned as being part of COBOL and one that is not sanctioned, but is used a lot.

Table 3-1 The COBOL Character Set

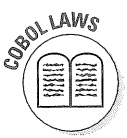
Character	*Name*
0, 1, . . . , 9	digit
A, B, . . . , Z	uppercase letter
a, b, . . . , z	lowercase letter
	space (I know you can't see it, but it's there. Honest.)
+	plus sign
-	minus sign (or hyphen if used within a word)
*	asterisk
**	exponentiation *(see note)*
/	slant, slash, or solidus (three silly names for the same thing)
=	equal sign
$	currency symbol (The dollar sign is the default currency symbol. You can specify other symbols, as necessary.)
,	comma (sometimes used as the decimal point)
;	semicolon
.	period (sometimes used as the decimal point)
"	quotation mark (also called the double quotation mark)
'	single quotation mark (This is not officially part of COBOL, but most compilers treat it as the same as the double quotation mark.)
(	left parenthesis
)	right parenthesis
>	greater than
>=	greater than or equal to *(see note)*
<	less than
<=	less than or equal to *(see note)*
:	colon

Note: The punctuation **, >=, and <= are not single characters. These character pairs have significance to COBOL.

That's What Little Programs Are Made Of

A COBOL program is a collection of words, punctuation, and spaces. My favorites are the spaces because it's harder to get them wrong. A word can be something that COBOL has defined and promises to recognize, or a word can be something you make up and inform COBOL as to its meaning. This neat arrangement works out quite well as long as you follow the set of rules that COBOL understands.

Whenever you want to come up with a word of your own (one that will carry a special meaning for you and for COBOL), you construct the word from a bunch of digits, letters, and hyphens. A letter is always a letter, a digit is always a digit, but a hyphen is the same character that you use for the minus sign. If you follow the rules of hyphenation, COBOL doesn't get confused.



The rules of hyphenation are simple: You never start or end a word with a hyphen. Also, you can never have a space inside a word, so a hyphen will always have another character right before it and right after it. A minus sign always must be preceded (and usually followed) by a space.

A semi-basic rule is that words can be no more than 30 characters long. I say "semi-basic" because lots of compilers don't worry about this rule — they let you make the words much longer. In fact, some compilers have no limit and let you make the words as long as you want. But you can always count on 30 characters being acceptable.

Here are some words that are valid in any COBOL program:

```
MAXIMUM
APPLE-BUTTER
ROCKET-SHIP-X15
4572-BOOMER
P156J
SOME-CAN-GET-PRETTY-LONG
```

As you can see in the preceding examples, COBOL names can both start with a digit and end with a digit. But you need to make sure that you include something other than digits somewhere inside the word because if it is all digits, COBOL thinks it's a number instead of a word. COBOL is right. Part II of this book is all about inventing and declaring your own names.

A tale of two cases

Long, long ago, when COBOL was but a child, the alphabet had only 26 letters. They were all uppercase letters. As time passed and COBOL grew into adulthood, more and more computer systems began to use a 52-letter alphabet — 26 uppercase letters and 26 lowercase letters. At first, COBOL was puzzled by this development and did not know what to do.

One day, while sorting a few million records for an insurance company, COBOL realized that the lowercase letters were exactly like the uppercase letters. The lowercase upstarts were slightly smaller, but COBOL saw no need to hold that against them. COBOL began treating the lowercase letters in exactly the same way as it has always treated the uppercase letters. This became COBOL's new alphabetic nondiscrimination policy — the little case-challenged letters are treated as equals.

For example, the following words are all exactly the same to COBOL:

```
Maximum-Value
maximum-value
MAXIMUM-VALUE
maxImum-vaLue
MAXIMUM-value
```

The same goes for keywords. These keywords are all identical in the eyes of COBOL:

```
PERFORM
perform
PerForm
perFORM
```

You can use the case-indifference of COBOL to create easily readable, non-hyphenated names by using a combination of upper- and lowercase letters. Here are some examples:

```
MaximumVelocity
JanuaryDailyAverage
LastReportedDate
```

You need to follow the coding style preferred by the place where you work. Code is much easier to read if everybody writes code the same way. This uppercase and lowercase thing is a matter of coding style, but it's only one part of an organization's preferred coding style.

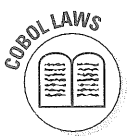
Hear the one about the space, the comma, and the semicolon?

Did you know that a COBOL compiler simply ignores every comma and semicolon that your program contains? It treats them like spaces. Any place you can use a space, you can use a comma or a semicolon. Also, anywhere you can use one space, you can use a bunch of them — that means any place you can use one comma or semicolon, you can use a bunch of them.

For example, these lines of code are exactly the same:

```
ADD ThisOne, ThatOne, TheOther TO Total.
ADD ThisOne ThatOne TheOther TO Total.
ADD ThisOne   ThatOne   TheOther TO Total.
```

Hear the one about the period, the number, and the sentence?



In COBOL, a period not followed by a space is assumed to be the decimal point in a number. A period followed by a space is taken to be the end of a sentence.

In the PROCEDURE DIVISION, paragraphs are composed of a name and one or more sentences. A statement begins with a verb, but may or may not end with a period. A sentence is a statement that ends with a period. That is, a statement — or a group of statements — with a terminating period is a sentence. Just as the clothes make the man, the period makes the sentence.

The period is very small in size and appearance, but very powerful in its effect on a COBOL program. You must take care and keep a short leash on any sentence-ending periods that you set loose into your program. If you find yourself working on some code that should be doing one thing but is doing something else, examine things closely for a misplaced period. The following example includes a misplaced period:

```
IF Apple < Orange
    ADD 3 TO Apple.
    ADD 2 TO Orange.
```

By looking at the indention pattern, it is apparent that the programmer wants both ADD statements to execute whenever Apple is less than Orange. Because of the period following Apple, however, the IF statement has no effect on the second ADD. The second statement — the one that adds two to Orange — always executes, regardless of whether Apple is less than Orange. Here is the correct form:

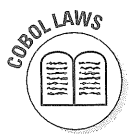
```
IF Apple < Orange
    ADD 3 TO Apple
    ADD 2 TO Orange.
```

The only difference is the period.

The Reserved Words

When you write your COBOL program, you get to make up all the words you are going to use. But you must remember that COBOL got here before you, so it has a head start in making up words. The words that COBOL made up and reserves for its own purposes are known, appropriately enough, as *reserved* words

A strange sort of thing can happen because COBOL has so many reserved words. You can easily use one without meaning to do so. If you get an error message from the compiler that says something really cryptic like `Sufficient optional differences unavailable in context` or `Expected literal ZERO conditional-expression or paragraph-name`, you may have accidentally used a reserved word without knowing it. The error message could be one that goes with the unintended reserved word, and that's why it makes no sense. This is one of COBOL's favorite practical jokes.



The law of COBOL states that you can't use, for your own purposes, any word that COBOL has already made up and is using for one of its own purposes. The good news is that it is very clear exactly what a word means in a COBOL program. The bad news is that all the good words are taken. Table 3-2 contains a complete list of the standard reserved words.

Table 3-2 The COBOL Standard Reserved Words

ACCEPT	DISPLAY	LESS	RETURN
ACCESS	DIVIDE	LIMIT	REVERSED
ADD	DIVISION	LIMITS	REWIND
ADVANCING	DOWN	LINAGE	REWRITE
AFTER	DUPLICATES	LINAGE-COUNTER	RF
ALL	DYNAMIC	LINE	RH
ALPHABET	EGI	LINE-COUNTER	RIGHT
ALPHABETIC	ELSE	LINES	ROUNDED
ALPHABETIC-LOWER	EMI	LINKAGE	RUN

ALPHABETIC-UPPER	ENABLE	LOCK	SAME
ALPHANUMERIC	END	LOW-VALUE	SD
ALPHANUMERIC-EDITED	END-ADD	LOW-VALUES	SEARCH
ALSO	END-CALL	MEMORY	SECTION
ALTER	END-COMPUTE	MERGE	SECURITY
ALTERNATE	END-DELETE	MESSAGE	SEGMENT
AND	END-DIVIDE	MODE	SEGMENT-LIMIT
ANY	END-EVALUATE	MODULES	SELECT
ARE	END-IF	MOVE	SEND
AREA	END-MULTIPLY	MULTIPLE	SENTENCE
AREAS	END-OF-PAGE	MULTIPLY	SEPARATE
ASCENDING	END-PERFORM	NATIVE	SEQUENCE
ASSIGN	END-READ	NEGATIVE	SEQUENTIAL
AT	END-RECEIVE	NEXT	SET
AUTHOR	END-RETURN	NO	SIGN
BEFORE	END-REWRITE	NOT	SIZE
BINARY	END-SEARCH	NUMBER	SORT
BLANK	END-START	NUMERIC	SORT-MERGE
BLOCK	END-STRING	NUMERIC-EDITED	SOURCE
BOTTOM	END-SUBTRACT	OBJECT-COMPUTER	SOURCE-COMPUTER
BY	END-UNSTRING	OCCURS	SPACE
CALL	END-WRITE	OF	SPACES
CANCEL	ENTER	OFF	SPECIAL-NAMES
CD	ENVIRONMENT	OMITTED	STANDARD
CF	EOP	ON	STANDARD-1
CH	EQUAL	OPEN	STANDARD-2
CHARACTER	ERROR	OPTIONAL	START
CHARACTERS	ESI	OR	STATUS

(continued)

Table 3-2 *(continued)*

CLASS	EVALUATE	ORDER	STOP
CLOCK-UNITS	EVERY	ORGANIZATION	STRING
CLOSE	EXCEPTION	OTHER	SUB-QUEUE-1
COBOL	EXIT	OUTPUT	SUB-QUEUE-2
CODE	EXTEND	OVERFLOW	SUB-QUEUE-3
CODE-SET	EXTERNAL	PACKED-DECIMAL	SUBTRACT
COLLATING	FALSE	PADDING	SUM
COLUMN	FD	PAGE	SUPPRESS
COMMA	FILE	PAGE-COUNTER	SYMBOLIC
COMMON	FILE-CONTROL	PERFORM	SYNC
COMMUNICATIONS	FILLER	PF	SYNCHRONIZED
COMP	FINAL	PH	TABLE
COMPUTATIONAL	FIRST	PIC	TALLYING
COMPUTE	FOOTING	PICTURE	TAPE
CONFIGURATION	FOR	PLUS	TERMINAL
CONTAINS	FROM	POINTER	TERMINATE
CONTENT	GENERATE	POSITION	TEST
CONTINUE	GIVING	POSITIVE	TEXT
CONTROL	GLOBAL	PRINTING	THAN
CONTROLS	GO	PROCEDURE	THEN
CONVERTING	GREATER	PROCEDURES	THROUGH
COPY	GROUP	PROCEED	THRU
CORR	HEADING	PROGRAM	TIME
CORRESPONDING	HIGH-VALUE	PROGRAM-ID	TIMES
COUNT	HIGH-VALUES	PURGE	TO
CURRENCY	I-O	QUEUE	TOP
DATA	I-O-CONTROL	QUOTE	TRAILING
DATE	IDENTIFICATION	QUOTES	TRUE
DATE-COMPILED	IF	RANDOM	TYPE
DATE-WRITTEN	IN	RD	UNIT
DAY	INDEX	READ	UNSTRING

DAY-OF-WEEK	INDEXED	RECEIVE	UNTIL
DE	INDICATE	RECORD	UP
DEBUG-CONTENTS	INITIAL	RECORDS	UPON
DEBUG-ITEM	INITIALIZE	REDEFINES	USAGE
DEBUG-LINE	INITIATE	REEL	USE
DEBUG-NAME	INPUT	REFERENCE	USING
DEBUG-SUB-1	INPUT-OUTPUT	REFERENCES	VALUE
DEBUG-SUB-2	INSPECT	RELATIVE	VALUES
DEBUG-SUB-3	INSTALLATION	RELEASE	VARYING
DEBUGGING	INTO	REMAINDER	WHEN
DECIMAL-POINT	INVALID	REMOVAL	WITH
DECLARATIVES	IS	RENAMES	WORDS
DELETE	JUST	REPLACE	WORKING-STORAGE
DELIMITED	JUSTIFIED	REPLACING	WRITE
DELIMITER	KEY	REPORT	ZERO
DEPENDING	LABEL	REPORTING	ZEROES
DESCENDING	LAST	REPORTS	ZEROS
DESTINATION	LEADING	RERUN	
DETAIL	LEFT	RESERVE	
DISABLE	LENGTH	RESET	

Table 3-2 may not include all the reserved words for your compiler. Most compilers add a few quirks and twists of their own, and some of them do so by adding special reserved words of their own. For example, it is common to add the keywords COMP-1, COMP-2, or COMP-3 to include special types of data that are peculiar to a particular computer. These words are not part of the standard and their implementation varies widely from one place to another. Check the documentation of your compiler.

Would you like to have some way of coming up with words that are guaranteed to not be reserved words? Okay — here are some rules you can follow to do that. These rules are part of the COBOL standard, so this should work even with compilers that have added their own words:

- You can begin a word with the digits 0 through 9 because no reserved word begins with a digit.
- You can use any word starting with X or Y, and any word starting with Z except ZERO, ZEROS, and ZEROES.

- You can use any single-character word because COBOL doesn't reserve any of them.
- You can use any word starting with a single character and a hyphen except I-O and I-O-CONTROL.
- You can use any word starting with two characters and a hyphen.

The END of Things

A group of reserved words are used for structuring COBOL sentences into neat and tidy little blocks of code. These special words are used to terminate a verb without terminating the sentence — the official name for these words is *explicit scope terminator* (which I refer to as scope terminator). That always sounds to me like some kind of deadly mouthwash. All the scope terminators start with END followed by a hyphen and the name of the verb being terminated. For example, the IF keyword has an END-IF and the READ verb has an END-READ. These terminators, when used correctly, can make for cleaner, block-structured code.

I can't emphasize enough the value of block-structured code. These scope terminators are the basic tools used to write structured code in COBOL — without them, such structure is just not possible. Take this example of structured code, which uses the scope terminators:

```
IF A > B THEN
    PERFORM VARYING J FROM 1 BY 1 UNTIL J > 3
        IF B > C THEN
            PERFORM BooBad
        ELSE
            PERFORM YayGood
        END-IF
    END-PERFORM
ELSE
    MULTIPLY A BY B
        GIVING J, K, L ROUNDED
        ON SIZE ERROR
            DISPLAY "Won't fit"
    END-MULTIPLY
END-IF.
```

This example is all one sentence — one complicated sentence. You can tell it's one sentence because it has only one period. The verbs are all terminated with END-*something* words, which makes the whole thing a clearer (and probably more correct) sentence than would be possible otherwise.

Use the scope terminators whenever it makes sense to do so — they come free with the compiler. Always use them whenever things start to get a little complicated. Just because a program is hard to write doesn't mean it should be hard to read. Also, whenever you use them, indent the code in some way that makes sense when you read it. The block-structure keywords alone don't do all the work. You need some kind of reasonable indention to help humans figure out what goes with what.

How do you determine which COBOL keywords have a scope terminator to go with them? That's easy — Table 3-3 is the complete list. In fact, the list may be too complete. These are the ones defined in the standard, but not all compilers implement all of them.

Table 3-3 The COBOL Block-Structuring Keywords

END-ADD	END-MULTIPLY	END-SEARCH
END-CALL	END-PERFORM	END-START
END-COMPUTE	END-READ	END-STRING
END-DELETE	END-RECEIVE	END-SUBTRACT
END-DIVIDE	END-RETURN	END-UNSTRING
END-EVALUATE	END-REWRITE	END-WRITE
END-IF		

When working with the block-structure keywords, don't use periods — just don't put one in there anywhere. If you do, the COBOL sentence — and with it, the scope — is terminated immediately. Usually, but not always, the compiler spews out an error when it finds a scope terminator that doesn't match up. There are insidious ways that the period can be inserted and pass right through the compiler. The program may run just fine, but it can occasionally do something weird.

Taking Action with COBOL Verbs

Every action statement in COBOL begins with a verb. You were told in school that the verbs were the action words. The same thing is true with COBOL. Some, but not all, of the COBOL reserved words are verbs. Table 3-4 lists all the verbs in COBOL.

Table 3-4 The Verbs of COBOL

ACCEPT	GENERATE	RELEASE
ADD	GO TO	REWRITE
ALTER	INITIALIZE	SEND
CALL	INITIATE	SET
CANCEL	INSPECT	SORT
CLOSE	MERGE	START
COMPUTE	MOVE	STOP
CONTINUE	MULTIPLY	STRING
DELETE	OPEN	SUBTRACT
DISABLE	PERFORM	SUPPRESS
DISPLAY	PURGE	TERMINATE
DIVIDE	READ	UNSTRING
ENABLE	RECEIVE	WRITE
EXIT		

The odds are pretty darned good that your compiler doesn't actually have all the verbs listed in this table. The table contains all the ones that are listed in the COBOL standard. In particular, the verbs DISABLE, ENABLE, GENERATE, INITIATE, PURGE, RECEIVE, SEND, SUPPRESS, and TERMINATE are part of a communication module that is defined as part of the standard but very rarely implemented.

Have you ever noticed that, if you say the word *verb* 15 times in a row, it sounds like you are trying to crank a '38 Chevy?

Zoning and the Indention Tradition

Each line of COBOL code must follow the rules of zones and margins. Certain character positions have a special meaning for the COBOL compiler. Figure 3-1 shows the format of a single line of COBOL code.

Here's a list of things you can put in each of the areas:

- Columns 1 through 6 are for the sequence numbers. Some compilers require these numbers, and some don't. Many compilers accept blanks here.

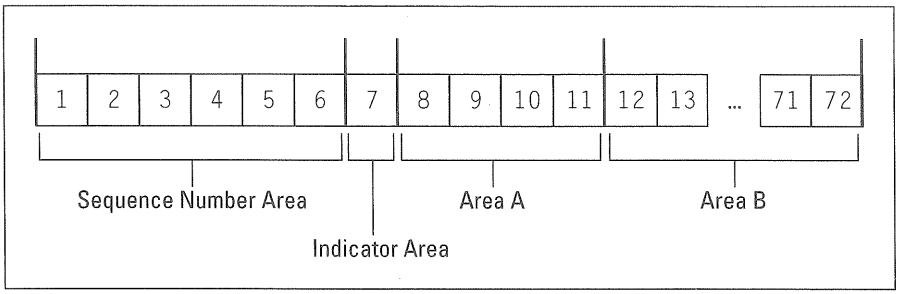


Figure 3-1: The reference format of a line of COBOL code.

- Column 7 is for an *indicator character.* It is sometimes called a *control character.* This column is normally blank, indicating nothing in particular. However, the following characters in column 7 have special functions:
  - An asterisk converts the entire line to a comment.
  - A slash character here is a comment, like an asterisk, except that it also causes the printer to jump to the top of the next page when you are printing the output directly from the compiler.
  - A minus sign indicates continuation — the first nonblank character of this line is appended to the last nonblank character of the previous line to make a really long line.
- Columns 8 through 11 are *area A.* This area is reserved for stuff like the names of divisions, sections, and paragraphs. You can also put certain data declarations here — things like 01, 66, and 77 levels, along with `SD` and `FD` file-level indicators.
- Columns 12 through the end of the line are *area B.* This area is for clauses, statements, and higher data levels (02, 03, and so on). Basically, this area includes anything that is not allowed in area A. There was a time (when COBOL was punched onto cards) when area B ended abruptly in column 72. Some compilers hold you to that, but mostly they don't mess with you after you go past column 12.

Notice that the leftmost position of any part of a COBOL program is in column 8, and that certain other parts of the program cannot begin before column 12. This format imposes a four-character indention on the leftmost parts of your code. You find that almost all programmers continue to indent in four-character steps. About the only time the indention is less than four is when things are being indented on a lot of levels and stuff gets squished too far off to the right.

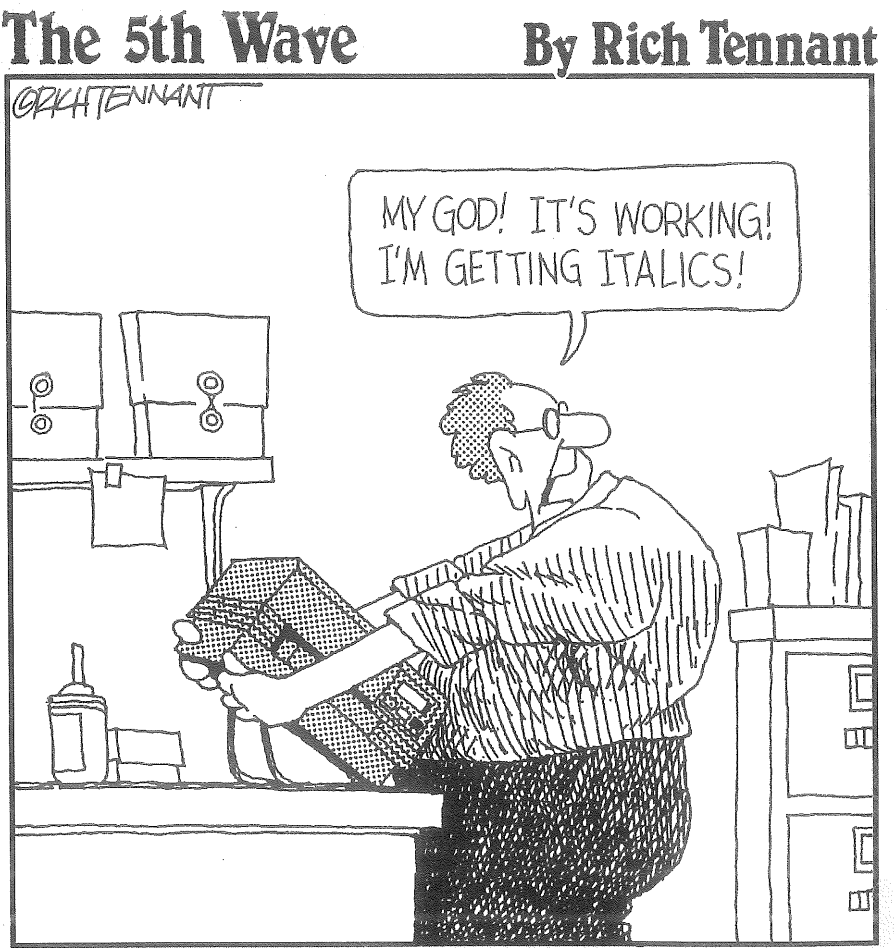
The following COBOL program example uses strict formatting:

```
000001 IDENTIFICATION DIVISION.
000002 PROGRAM-ID. Spacing.
000003 ENVIRONMENT DIVISION.
000004 DATA DIVISION.
000105 WORKING-STORAGE SECTION.
000150 01  Counters.
000160     02  I PICTURE 99.
000170     02  K PICTURE 99.
000300
007000 PROCEDURE DIVISION.
007010* This paragraph will always set K to 2
007020 MAIN-PARAGRAPH.
007025     PERFORM VARYING I FROM 1 BY 1 UNTIL I > 3
007030         IF I IS EQUAL TO 2
007035             COMPUTE K = I
007040         END-IF
007045     END-PERFORM.
```

The first six columns contain the sequence numbers — in order. Notice that while the numbers are in order, gaps exist. These gaps in the numbering make it possible to insert new lines of code without having to renumber all the rest of them. Back in the days of punched cards, it only took once to figure out this trick! Column 7 is blank except for a couple of comment lines that have an asterisk as the control character. Area A contains a paragraph name, an 01 level, the PROGRAM-ID, and the names of divisions and sections. All levels above 01, and all executable statements, begin in area B. A blank line — line 300 in the example — is simply skipped over by COBOL.

Part II

The DATA DIVISION Is Where You Put Things



In this part . . .

COBOL data records are constructed in the form of hierarchical records — one data item nested inside another. COBOL gives you special ways of constructing these records so that a data item forms associations with other data items. You can even have two distinct data items located at the same place in memory.

COBOL has many ways of describing numeric and character data. Some formats are for internal data manipulation and some are for external display. COBOL has more ways of representing data than any other major computer language in the world.

The chapters in this part show you how to create data records in COBOL. In these chapters, I give you all the details about formatting data for display and for speed of calculation. I describe the various data types that are available in COBOL, and I show you how to use all of them.

Chapter 4

Creating Data Descriptions: Describing the Real World or the Planet Pljfmyx

In This Chapter

- Defining records with numbered levels
- Overlapping records in memory
- Creating single-field definitions
- Defining conditional values with the 88 level
- Qualifying named references
- Calculating record sizes

The storage location for a single item of data (such as a name, a telephone number, or a batting average) is called a *field.* When you combine a group of related fields to create a single unit, this unit is called a *record.* A collection of records stored on disk is called a *file.* Part IV discusses files. This chapter focuses on fields and records.

In COBOL, the general term for a field or a record is a *data description.* This chapter introduces the different kinds of data descriptions you can create in COBOL. This chapter also explains how you can position and tag fields to give them special meanings and how you combine fields to create records. You define an individual field with a `PICTURE`, or `PIC`, clause. You can see `PICTURE` clauses throughout the data descriptions that I present in this chapter; Chapter 5 provides in-depth coverage of the `PICTURE` clause.

Assigning Level Numbers — And 01 and 02 and 03 . . .

Do you need a place to hold the name and address of a person? Okay, just create a great big place — one large enough to hold name, address, and maybe some other folderol. In COBOL, you create a place to hold stuff with a *data description*. This data description creates room enough to hold the name and address:

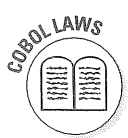
```
01  PERSON PIC X(300).
```

This data description creates a space to hold 300 characters — plenty of room for a name and address. It works, but it does have a couple of drawbacks. Say, for example, you want to print the address on an envelope. If you create a record using the preceding code, you have to write a bunch of additional code to break the address apart so you can then print the different parts (name, street address, zip code, and so on) on different lines of the envelope. However, a better way exists to store this information.

You can store the separate pieces of information about a person inside distinct and identifiable parts of a single unit. This single unit is called a *record*. Each separate piece of data about the person is stored in its own *field*. Here's an example showing how you combine the fields into records by assigning *level numbers* and placing several fields under one record:

```
01  PERSON.
    02  NAME       PIC X(64).
    02  ADDRESS-1  PIC X(60).
    02  ADDRESS-2  PIC X(60).
    02  CITY PIC   X(32).
    02  STATE PIC  XX.
    02  ZIP PIC    X(10).
```

This record — named `PERSON` — is made up of six fields. Each field in the record has its own name and size. For example, the `PIC` clause for the `CITY` field specifies that this field can hold 32 characters. It is considered polite to indent the fields so their relationships are visible.



The beginning level of a record is always an 01 level; it always has a name, and it never has a `PICTURE` clause. The subsequent levels — all levels other than the 01 level — can have any number from 02 through 49. COBOL has some other level numbers greater than 49, but they are used for some special purposes, which I describe in subsequent sections of this chapter. By the way, people who speak the COBOL language talk about these levels a lot, referring to them as the "oh-one level," the "oh-two level," and so on.

You can also include records inside of records. Just bump the level numbers and keep going. For example, the following example shows the PERSON record divided up further to give it a bit more pizzazz:

```
01  PERSON.
    02  NAME.
        03  FIRST-NAME    PIC X(32).
        03  LAST-NAME     PIC X(32).
    02  ADDRESS.
        03  ADDRESS-1     PIC X(60).
        03  ADDRESS-2     PIC X(60).
    02  CITY              PIC X(32).
    02  STATE             PIC XX.
    02  ZIP               PIC X(10).
```

This version of the PERSON record contains three fields and two records. The NAME has been converted to a record containing two fields: FIRST-NAME and LAST-NAME. The two address fields are now combined, creating a new record named ADDRESS. The entries with the PICTURE clauses on them are fields that are sometimes called *elementary items* — an elementary item cannot have any subordinate items.

I need to add one more refinement to this discussion of level numbers: Skip a few numbers when you are creating a data description. It isn't necessary, but it is a wise thing to do. COBOL lets you skip numbers going from one level to another — the only requirement is that each subsequent level number be larger than the one before it, and the largest number you can use is 49. Skipping a few numbers sure makes it easier to make changes if you have to insert some stuff later.

The following code shows the PERSON record with a new numbering scheme:

```
01  PERSON.
    05  NAME.
        10  FIRST-NAME    PIC X(32).
        10  LAST-NAME     PIC X(32).
    05  ADDRESS.
        10  ADDRESS-1     PIC X(60).
        10  ADDRESS-2     PIC X(60).
    05  CITY              PIC X(32).
    05  STATE             PIC XX.
    05  ZIP               PIC X(10).
```

The only difference between this version and the previous version of the code is that the numbers jump by five instead of by one. This change in numbering makes no difference whatsoever to the COBOL compiler, but it

can make things easier for you when the time comes to make changes to your data description. If you need to insert a new level somewhere, you don't need to renumber every field in the record — just stick in an 03 or 07 level wherever you need it.

Only two cases exist in which this number-skipping is a complete waste of time:

- Your programs are always perfect and complete and never need updating.
- Your programs are so lousy they will be thrown out and rewritten anyway.

Assigning New Field Names — The 66 Level and RENAMES

You use the 66 level to assign a new name to a field or to a group of fields. The 66 level doesn't get rid of the old name; it just adds a new one. The 66 level and `RENAMES` go together like corn pone and molasses — whenever you've got one, you've got the other. Here's an example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ReRose.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 Flower.
    05  Rose PIC X(20).
66  Nasturtium RENAMES Rose.
PROCEDURE DIVISION.
INIT.
    MOVE "sweet" TO Rose.
    DISPLAY Nasturtium.
    DISPLAY Rose.
    STOP RUN.
```

The 66-level `Nasturtium` is used to assign a new name to the field named `Rose` inside the `Flower` record. It doesn't remove the old name; it just adds a new one. When this program is run, it displays the following output:

```
sweet
sweet
```

This output proves once and for all that a Rose by any other name will print as sweet.

Situations arise in which you have data grouped in one way, but need it grouped in some other way. If this regrouping does not require reordering of the data (or converting it to another data format), you can use a 66 level. The following example shows one way to regroup your data using a 66 level:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DayOfReckoning.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  RECKONING.
    05  DAY-OF-MONTH PIC 9(2).
    05  FILLER PIC X VALUE "/".
    05  MONTH PIC 9(2).
    05  FILLER PIC X VALUE "/".
    05  YEAR PIC 9(4).
66  DAY-MONTH RENAMES DAY-OF-MONTH THRU MONTH.
66  MONTH-YEAR RENAMES MONTH THRU YEAR.
PROCEDURE DIVISION.
INIT.
    MOVE 31 TO DAY-OF-MONTH.
    MOVE 12 TO MONTH.
    MOVE 1999 TO YEAR.
    DISPLAY "RECKONING: " RECKONING.
    DISPLAY "DAY-MONTH: " DAY-MONTH.
    DISPLAY "MONTH-YEAR: " MONTH-YEAR.
    STOP RUN.
```

The record named RECKONING contains some date fields and uses slashes to improve the appearance whenever the dates are displayed. A couple of 66-level entries are defined in such a way that they create groupings that are not defined inside the record. Notice that the 66 levels are defined completely outside the record — they are an entirely separate entity using the RENAMES keyword to refer back to members of the record. When the program is run, the output display looks like this:

```
RECKONING: 31/12/1999
DAY-MONTH: 31/12
MONTH-YEAR: 12/1999
```

You use the 66 level in this way to create multiple views of the same data. For example, you may want to use it to combine cities with zip codes or to break apart phone numbers and area codes. You probably won't use this capability a lot, but it sure is handy when you need it.

You can think of a 66 level as nothing but a name you can use to refer to a collection of fields inside a record. If you think of it in this way, you are dead-spang-on accurate — that's all it is.

If you find that RENAMES comes up short for what you need, try REDEFINES.

Using REDEFINES

The REDEFINES clause performs the same sort of trick as the 66-level RENAMES. It puts two seemingly different things at the same place — that is, two data descriptions at the same storage location. REDEFINES has some special powers denied to RENAMES. The REDEFINES clause not only changes the name of a data description, but it can also change the data description's field type and format. You can use this capability to access a group of fields as a single field, a single field as a group of fields, or a group of fields as another group of fields. For example, you can use it to extract specific characters from the middle of a larger field or to overlay one record layout on top of another to store them in the same file.

COBOL has some pretty strict laws on REDEFINES:

- The redefining level in the record must be the same as the original defining level.
- The REDEFINES clause must immediately follow the thing it's redefining — that is, no 01, 77, or 66 records can come between them.
- The REDEFINES clause and the thing it is redefining must be declared at the same level.
- You cannot use the REDEFINES clause on an item with a VALUE clause.
- The data item being redefined cannot have an OCCURS clause.
- The REDEFINES clause can be used as an 01 level unless it is being used in the FILE SECTION or the COMMUNICATIONS SECTION.

But don't worry, even with all these laws, you can still shoot yourself in the foot. Using REDEFINES allows you to refer to any data description as any other kind of data description. You can trick COBOL into thinking that you have one type of data when you actually have another.

A rose is a rose — unless you REDEFINE it

You can use an 01 level to redefine another 01 level, as in the following example:

```
01  Flower.
    05  Rose            PIC X(20).
01  AnotherFlower REDEFINES Flower.
    05  Nasturtium      PIC X(20).
```

With the arrangement of flowers in this code, the `Rose` and the `Nasturtium` are exactly the same — in writing your code, it doesn't matter which flower you pick. Both flowers are the same size and type, and they reside at the same location in memory — they differ in name only.

As the following flower arrangement demonstrates, you can redefine one thing to also be another in the same record as long as the redefinor and the redefinee are at the same level:

```
01  Flower.
    05  Rose            PIC X(20).
    05  Nasturtium REDEFINES Rose.
        10  FILLER      PIC X(20).
```

The `Rose` and the `Nasturtium` are still one and the same.

You can even use `REDEFINES` to assign a whole bunch of names to one place, like this:

```
01  Flower.
    05  Rose              PIC X(20).
    05  Nasturtium REDEFINES Rose.
        10  FILLER        PIC X(20).
    05  Posey REDEFINES Rose.
        10  FILLER        PIC X(20).
    05  HighBiscuits REDEFINES Rose.
        10  FILLER        PIC X(20).
```

This example defines only a `Rose`, and then gives it all the other names. This technique can be useful if the logic of your program makes more sense when you call your `Rose` by another name. Most often, however, the names created by `REDEFINES` are used to impose a different `PICTURE` definition to provide a different way to access the data.

One fit sizes all

Size matters to REDEFINES. You can redefine something to make it smaller, but you can't redefine anything to make bigger. This is not a real limitation — if you make the original definition the largest one of the bunch, all your redefines work because you have plenty of room for them. As the following example demonstrates, however, a reference to one of the smaller members is a reference to only the smaller portion of the region shared between it and the one it is redefining:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. RushHour.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  Vehicle.
    05  EighteenWheeler PIC X(30).
    05  Coupe REDEFINES EighteenWheeler.
        10  FILLER PIC X(6).
    05  Sedan REDEFINES EighteenWheeler.
        10  FILLER PIC X(12).
PROCEDURE DIVISION.
INIT.
    MOVE "I have to go to the dentist" TO EighteenWheeler.
    PERFORM SHOW-CONTENT.
    MOVE "I love" to Coupe.
    PERFORM SHOW-CONTENT.
    STOP RUN.
SHOW-CONTENT.
    DISPLAY Coupe.
    DISPLAY Sedan.
    DISPLAY EighteenWheeler.
```

The EighteenWheeler is redefined with the assignment of two names representing smaller regions of memory. The program begins by filling up the EighteenWheeler with characters and then displays all three of its incarnations. Some are bigger than others, as the following output shows:

```
I have
I have to go
I have to go to the dentist
```

The program then comes to the second MOVEment — the characters I love are moved to Coupe. Although the MOVE statement completely overwrites the value of Coupe, it has no effect anywhere other than on the first six

characters (the size of Coupe) of EighteenWheeler and Sedan. The remainder of the storage area is left intact, causing the display statements to produce the following output:

```
I love
I love to go
I love to go to the dentist
```

Changing the data type

You can use REDEFINES to change data from one type to another. Well, it actually doesn't change anything — it lets you store the data as one type and read it back as if it were another. Chapter 18 includes an example of using this unchanging characteristic to do some bit twiddling.

Be careful with this capability — some data types don't make sense when interpreted as other data types. (It all depends on the details of the internal format of the data — which I discuss in more detail in Chapter 5.) The problem mostly applies to redefining that involves members of the COMPUTATIONAL family of data declarations, but lots of other places exist where a simple REDEFINES can cause you to spend some time debugging a garbage generator.

As the following example demonstrates, unless you declare data as COMPUTATIONAL or some other special USAGE, the data is stored in character format:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Digits.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  NameAndNumber.
    05  Digits PIC 9(5).
    05  FILLER REDEFINES Digits OCCURS 5 TIMES.
        10  OneCharacter PIC X.
77  Cindex USAGE IS INDEX.
PROCEDURE DIVISION.
INIT.
    MOVE 1998 TO Digits.
    DISPLAY Digits.
    PERFORM VARYING Cindex FROM 1 BY 1 UNTIL
            Cindex IS GREATER THAN 5
        DISPLAY OneCharacter(Cindex)
    END-PERFORM.
```

This example creates a five-digit number and REDEFINES it as a five-character array. By the way, notice that Digits is redefined as a FILLER. That's fine — you don't need to insert a name in a place where you never need one. The FILLER is a record that contains a single character, but the record occurs five times, so the array of characters is the same size as Digits. A four-digit number is moved to Digits and displayed once in its entirety and again — through its redefined name — one character at a time. The output looks like this:

```
01998
0
1
9
9
8
```

Chapter 7 has more information on working with arrays and REDEFINES.

Declaring Independent Data — The 77 Level

You use a 77 level to define an independent data item — one that is not related in any way to any of the data around it. There is nothing you can do with a 77 level you can't do with an 01 level. The following two declarations are absolutely identical:

```
01  Doober PIC $$,$$$9.99.
77  Doober PIC $$,$$$9.99.
```

If that's all it can do, why use a 77 level? Well, I have heard it said that it is a good idea to use the 77 level for the single-name independent declarations and reserve the use of 01 for defining a record that contains subordinate data items. The intention is to make the code easier to read, and I suppose it could.

I know what the 77 level is really for: It's for starting arguments. You will hear discussions flare up that have to do with 77 levels instructing COBOL to align the data on boundaries that speed access times or pack the data more tightly to save space. Some say that it allows COBOL to reorganize the most-often-used data into faster memory. There is even talk that it optimizes virtual memory. The truth is that, at one time, it was all of these things, but it isn't anymore. Hardware, operating systems, and compiler technology have advanced to the point that 01 levels and 77 levels are treated the same.

Declaring Conditional Data — The 88 Level

An 88 level is completely different from any other level. The main difference is that it does not create or address a storage location. You use an 88 level to declare a name for a possible value that a field may assume. Here's an example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Warmers.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  TYPE-OF-COAT PIC X.
    88  MINK              VALUE "M".
    88  RABBIT            VALUE "R".
    88  RECYCLED-LINT     VALUE "L".
PROCEDURE DIVISION.
WARMER-MAIN.
    MOVE "M" TO TYPE-OF-COAT.
    PERFORM TELL-TYPE.
    MOVE "R" TO TYPE-OF-COAT.
    PERFORM TELL-TYPE.
    STOP RUN.
TELL-TYPE.
    IF MINK
        DISPLAY "The coat is mink.".
    IF RABBIT
        DISPLAY "The coat is rabbit.".
    IF RECYCLED-LINT
        DISPLAY "The coat is ugly.".
```

The `TYPE-OF-COAT` can be set to any character — it is not limited to those named in the 88 levels. The 88-level declarations simply assign names to three of the possible characters. These names can then be used in an `IF` conditional. In the preceding example, the statement `IF MINK` is exactly the same as `IF TYPE-OF-COAT IS EQUAL TO "M"`.

The trick is to select meaningful names to simplify and clarify the COBOL statements that use those names.

A single 88 level name can be made to represent more than one value, as the following example shows:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ODD-EVEN.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  MAGIC-NUMBER PIC 9 COMP.
    88  OddMagicNumber   VALUES 1,3,5,7,9.
    88  EvenMagicNumber  VALUES 0,2,4,6,8.
PROCEDURE DIVISION.
ODD-EVEN-MAIN.
    PERFORM VARYING MAGIC-NUMBER FROM 1 BY 1
            UNTIL MAGIC-NUMBER > 9
        IF OddMagicNumber
            DISPLAY MAGIC-NUMBER " is odd"
        END-IF
        IF EvenMagicNumber
            DISPLAY MAGIC-NUMBER " is even"
        END-IF
    END-PERFORM.
    STOP RUN.
```

In this example, the value of MAGIC-NUMBER runs from 1 through 9, and each value is tested to determine whether it is odd or even. The test is made by using an 88-level name on an IF statement. Whenever the actual value matches a value defined on an 88-level name, the IF statement turns up true. This program correctly tags all the odd and even numbers.

This multiple-value setting for an 88 level can come in handy for all sorts of things. Take the situation in which your program reads some input from the user. Programming would be much easier if it weren't necessary to deal with those pesky, unpredictable humans. A simple yes-or-no question can have any one of a number of answers. The following example shows one way to try to deal with the unpredictable responses from the biological input unit (also known as the user):

```
IDENTIFICATION DIVISION.
PROGRAM-ID. yorn.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  Response PIC X(10).
    88  YesResponse VALUES "y", "Y", "Yes", "YES", "yes".
PROCEDURE DIVISION.
```

```
YES-OR-NO-MAIN.
    DISPLAY "Yes or No? " NO ADVANCING.
    ACCEPT Response.
    DISPLAY "You answered " Response.
    IF YesResponse
        DISPLAY "I assume that means yes"
    ELSE
        DISPLAY "I assume that means no"
    END-IF.
    STOP RUN.
```

This program accepts any one of several forms of "uh-huh." The program takes anything that does not match one of these forms to be "no." As you can see, extending the list is a simple matter. You can add `"sure"`, `"you betcha"`, `"right on"`, or anything else you like — all without making any changes to the code that tests whether the user has entered one of the values. Chapter 9 has a lot more information about this `IF` stuff.

Because it is possible to assign more than one value to an 88 level, a numeric 88 level allows the words `THROUGH` or `THRU` to specify a range of numbers. Here's an example:

```
01  RateOfIncrease PIC 9(4) COMP.
    88  MaximumRate VALUE 3157.
    88  MinimumRate VALUE 31.
    88  NormalRate VALUES 896 THRU 1143.
```

This example also demonstrates that an 88 level can be more than just a convenience for conditional tests. Well-considered 88-level names can make COBOL code easy to read and understand. Without knowing anything else about the program that includes the preceding example, just looking at this code tells you a lot. You immediately get information about the range of possible values, along with an idea of what the values may mean. Even if the 88-level names are never used in the program, they have already performed a service by simply being there.

Qualifying References with OF and IN

A COBOL program can have any number of records. Each of these records must have a unique name — that is, no two names at the 01 level can be the same. Inside these records, names can be duplicated all over the place, just as long as you have some way to uniquely address each name. Look at the following example:

```
01  COMPANY.
    05  NAME PIC X(30).
    05  TELEPHONE.
        10  OFFICE.
            15  AREA-CODE  PIC 9(3)
            15  TELEPHONE  PIC 9(7)
        10  800-NUMBER.
            15  TELEPHONE  PIC 9(7)
    05  LOCATION.
        10  NAME           PIC X(32).
        10  800-NUMBER     PIC 9(7).
01  PERSON.
    05  NAME.
        10  FIRST-NAME     PIC X(32).
        10  LAST-NAME      PIC X(32).
    05  TELEPHONE.
        10  AREA-CODE      PIC 9(3).
        10  TELEPHONE      PIC 9(7).
```

Some names (TELEPHONE, NAME, AREA-CODE) are duplicated in these records. This duplication works okay because each one can be uniquely addressed. You use the keywords IN or OF to create the address qualifications. Both IN and OF do the same thing — it is just a matter of personal preference which one you want to use. You can see where the following statement could bring up a question:

```
MOVE "Herbert" TO NAME.                 *** ERROR ***
```

The compiler takes one look at this statement, assumes a very righteous attitude, and asks you just which NAME you mean. You have one NAME in PERSON and one in COMPANY. A proper response from you is to qualify the reference this way:

```
MOVE "Herbert" TO NAME OF PERSON.
```

The preceding code works, but you can even be more specific by using the following code:

```
MOVE "Herbert" TO FIRST-NAME OF NAME OF PERSON.
MOVE "Frankenstein" TO LAST-NAME OF NAME OF PERSON.
```

Here's another one that doesn't work:

```
MOVE 972 TO AREA-CODE.                  *** ERROR ****
```

This code fails because you have an AREA-CODE in both the PERSON and the COMPANY records. The following valid statements all insert area codes into the records:

```
MOVE 972 TO AREA-CODE OF COMPANY.
MOVE 972 TO AREA-CODE OF OFFICE.
MOVE 972 TO AREA-CODE OF OFFICE OF COMPANY.
MOVE 972 TO AREA-CODE OF OFFICE OF TELEPHONE.
MOVE 972 TO AREA-CODE OF TELEPHONE OF COMPANY.
MOVE 972 TO AREA-CODE OF OFFICE OF TELEPHONE OF COMPANY.
```

Each of the preceding statements does exactly the same thing as the others. As you can see, any level references can be included or skipped, just as long as the reference is unique. If you can look at it and figure it out, COBOL can figure it out, too. What COBOL does is try every possible combination to make sure there is no way your statement can refer to more than one item.

Some things just don't work. At first, it looks like the name of the company could be set this way:

```
MOVE "Acme" TO NAME OF COMPANY.          *** ERROR ***
```

This is ambiguous because you have two things called NAME in the COMPANY record — one at the top level and another one down inside the LOCATION. The following code works:

```
MOVE "Acme" TO NAME OF LOCATION.
MOVE "Acme" TO NAME OF LOCATION OF COMPANY.
```

However, you can't address the other NAME — the one at the 05 level. The only way to fix this problem is to change the record. You must either change the name of something or rearrange the record in such a way that it can be unambiguously referenced. My suggestion is to just change it from NAME to something like COMPANY-NAME.

Inserting the FILLER

One of the basic and most-used talents of a programmer is the ability to make up names. You must name the program, the paragraphs to be executed, the data records, and all the fields. Whenever records are grouped into subrecords, you need still more names. At times, you have to come up with names that are not used for anything but placeholders. In such cases, good ol' COBOL provides some relief. Take a look at the following record, designed to hold a social security number:

```
01  SS-NUMBER.
    03  FirstPart PIC 9(3).
    03  FirstHyphen PIC X VALUE "-".
    03  SecondPart PIC 9(2).
    03  SecondHyphen PIC X VALUE "-".
    03  ThirdPart PIC 9(4).
```

Inserting fields to hold the hyphens works. You can stick numbers into each of the three parts and pull them right back out again. If you print or display SS-NUMBER, it has the hyphens right where they belong — the number looks like this:

```
000-00-0000
```

However, you don't need to name the hyphens unless you plan on changing them or something. The following code shows another way of doing the same thing:

```
01  SS-NUMBER.
    03  FirstPart PIC 9(3).
    03  FILLER PIC X VALUE "-".
    03  SecondPart PIC 9(2).
    03  FILLER PIC X VALUE "-".
    03  ThirdPart PIC 9(4).
```

The names of the hyphens have now been replaced with the keyword FILLER. It's exactly as it was before, except for the fact that you have no way to address the hyphens.

You can use FILLER for all sorts of things. Suppose that you want to create some fields and, for convenience, you want to have them in a group, but you never refer to the entire group by its name. Instead of making up some name you will never use, you can use FILLER, like this:

```
05  FILLER.
    10  BeginningValue PIC 9(8).
    10  EndingValue PIC 9(8).
```

You find this type of code down in the middle of large record definitions to help clarify relationships among the fields. It works when you have a group of related fields but you don't need to call the group by a name.

Another use for FILLER is as a placeholder for future construction. For example, you may write a program that is going to create data files holding thousands of records. If you know that the records in these files are going to

be expanded someday, you can include a FILLER in the record to reserve a little space for future use, as in the following example:

```
01  DATA-FILE-RECORD.
    02  IDENTITY PIC X(2).
    02  LARRY-DATA PIC 9(5)V9(2).
    02  CURLY-DATA PIC 9(3)V9(3).
    02  MOE-DATA PIC X(8).
    02  YET-MOE-DATA PIC X(12).
    02  FILLER PIC X(120).
```

This example record is set up to include the data fields that you know about now, and it also reserves 120 characters for future expansion. Each time the entire data record is written to (or read from) the file, it carries the extra 120 characters. At some later date, some or all of this 120-character block can be used to add new fields to the record, and the record size of the existing file will not have to be changed. Sneaky, eh?

Determining the Size of a Record

Packing data fields into a record is like packing a suitcase. Two things determine the space required: the size of the individual items and the size of the gaps (if any) between them. The size of these gaps can vary from one compiler to the next. For some records, determining the actual size can be difficult to do.

If you don't have to take any special cases into consideration, the size of a record is simply the sum of the sizes of its parts. Take this record:

```
01  SIZER.
    03  Name PIC A(32).
    03  Address PIC X(80).
    03  Telephone PIC 9(10).
```

The fields of SIZER all default to USAGE DISPLAY, so that each one is contained in exactly the number of characters shown on the declaration — the size of this record is 122 bytes. When toting up the sizes, be aware of the meanings of the special formatting characters. In the following example, these two fields are capable of containing and displaying exactly the same range of values, but they are not the same size:

```
77  IMPLICIT PIC 9(4)V9(4).
77  EXPLICIT PIC 9(4).9(4).
```

In this example, one field has an implied decimal point and the other has an explicit decimal point. The length of IMPLICIT is 8 and the length of EXPLICIT is 9 — the decimal point takes up the space of a character, but the V does not. (For all the details about the PIC clause, check out Chapter 5.)

Sizing up COMP and BINARY

The actual storage size of a COMPUTATIONAL field varies from one compiler to another. Take this declaration:

```
01  RealNumber PIC 9(4) COMP.
```

You can deduce certain facts about the declaration of the field RealNumber: It is to be represented internally in some form other than the standard set of COBOL characters, and it is to be capable of working with values from 0 to 9,999. It has to be larger than one byte, but it doesn't have to be any larger than two. However, you can't assume it will be stored in two bytes. It could be stored in four because the machine you are using just works better with four. It could be done in three — it's not likely, but it is possible.

This sizing and alignment problem is true for BINARY as well as COMP. It is especially true for things like COMP-2 or COMP-3. The best you can do is refer to the manual of the compiler you are using. If the size information is not in the manual, call the person who sold you the compiler and say something rude. You can say I told you to call.

Allowing for synchronization and the slack byte

Some things can cause surprises in the size of records. Under some circumstances, the compiler intentionally leaves holes in the middle of a record, just as if you had inserted a FILLER. This intentional hole in the middle of a record is generally called *alignment,* and the added space is sometimes called the *slack byte.* Under normal circumstances, this added space doesn't make any difference, but sometimes you need to know the exact size of a record.

The most common case in which the size becomes important is in defining records to be written to files, as I describe in Part IV of this book. The automatic alignment almost exclusively applies to COMP and BINARY fields (which I discuss in Chapter 5) and OCCURS arrays (see Chapter 7).

Here's the deal. In the name of efficiency, some special data types (in particular, COMP and BINARY) are, on some computers, automatically aligned on certain address boundaries. This alignment can leave holes in a record. The following record is a candidate for having this happen:

```
01  Slacker.
    05  ThreeChar PIC X(3).
    05  Money PIC S9(5)V9(2) COMP.
```

The Slacker record contains a COMP field. If you have a compiler that automatically adjusts the position of COMP data to reside on a four-byte boundary, the record will be compiled as if you had written it this way:

```
01  Slacker.
    05  ThreeChar PIC X(3).
    05  FILLER PIC X.
    05  Money PIC S9(5)V9(2) COMP.
```

The preceding code makes the record length a total of eight bytes in length, instead of the expected seven bytes. In fact, if the record had been written this way

```
01  Slacker.
    05  ThreeChar PIC X.
    05  Money PIC S9(5)V9(2) COMP.
```

it would still be eight bytes long because three bytes would be inserted to align the COMP field, as in the following code:

```
01  Slacker.
    05  ThreeChar PIC X.
    05  FILLER PIC X(3).
    05  Money PIC S9(5)V9(2) COMP.
```

If you are constructing a record that has a bunch of COMP fields intermingled with some other fields, you may want to rearrange things. You can save a lot of space if you are careful to arrange the data so the COMP fields are all together, removing the necessity for the compiler to insert a bunch of slack bytes. This doesn't mean much for a single record in memory, but if you are going to write thousands of records to disk, you can save a lot of space.

An OCCURS statement can cause alignment. Take this example:

```
01  YearlySummary.
    02  Name PIC X(31).
    02  Month OCCURS 12 TIMES.
        03  MonthName PIC X(5).
        03  Amount PIC 9(5).
```

Month is addressed by an index because it is an array. (I discuss arrays in Chapter 7.) Some computers require special addressing modes for indexing. If arrays must appear on four-byte boundaries, the compiler handles the record as if it had been written this way:

```
01  YearlySummary.
    02  Name PIC X(31).
    02  FILLER PIC X.
    02  Month OCCURS 12 TIMES.
        03  MonthName PIC X(5).
        03  Amount PIC 9(5).
        03  FILLER PIC X(2).
```

The first FILLER exists to align the beginning of the first member of the OCCURS array. The second FILLER increases the size of each member of the array from 10 to 12, forcing all members of the array to reside on a four-byte boundary.

Chapter 5

Yes, Virginia, There Is a PICTURE Clause

In This Chapter

- Selecting symbols for PICTURE construction
- Creating five types of data
- Representing positive and negative values
- Forcing different internal data formats
- Automating field edits
- Handling international currency conversion

Computers — bless their little binary hearts — like to store data in all sorts of arcane ways. By default, COBOL always stores data as a string of man-readable characters. COBOL does this for numeric data as well as for data like names and addresses. However, you can request that COBOL store the data in other ways. In fact, the laws of COBOL leave a lot of latitude for the folks who implement the compiler — these folks have created lots of storage formats that they optimize for a specific piece of hardware.

As I discuss in this chapter, you can define how many characters COBOL uses to store each piece of data. You also specify what kinds of characters (alphabetic, numeric, and so on) each data location can hold. And you can set up a storage location so that it has special editing attributes — this editing automatically fiddles around with any data that you decide to store in that location.

You have a lot to remember about the PICTURE clause — all the letters and what they mean and how they react with one another and which ones make an edited field. That is, you *would* have a lot to remember if you tried to remember all of it. The truth is that you only need to remember a few of the symbols for the PICTURE clause. You find that you tend to use the same ones over and over (mostly X and 9).

Whenever I need to put together a format that is a bit out of the ordinary, I use the "rats and aha" method. I just make something up and shove it at the compiler. When it fails — and the friendly compiler tells me exactly where and why it fails — I just mumble, "rats" under my breath and try another combination. When I have one that works, I say, "aha." Sometimes the "rats" to "aha" ratio gets pretty high, but it usually works out quite well.

There is one advantage to "rats and aha." After you run through this trial-and-error method a few times, you begin to get a feel for how all this PICTURE stuff goes together. It becomes intuitive after you work with it some. Really.

A PICTURE Can Contain a Thousand Words

A PICTURE clause is the character-by-character definition of the format of data. COBOL assigns special meanings to characters that you use in the PICTURE clause. For example, an X character used in a PICTURE clause creates a position in the computer's memory that can hold one character; a pair of X characters can store two characters; a trio stores three characters; I can go on. For example, if you have a favorite four-letter word and you want to create a place to store it, just write this code:

```
77  dirty-word PICTURE IS XXXX.
```

This PICTURE clause creates a field called dirty-word that can contain any four-letter word you can think of. If you are a really creative person and have a ten-letter word, you can store it this way:

```
77  long-word PICTURE IS XXXXXXXXXX.
```

Right off, you can see a disadvantage to this format — you can get your eyes crossed trying to count how many X symbols you have. Fortunately, you can write this code in another way. Instead of writing one X after another, you can use a number in parentheses, like this:

```
77  long-word PICTURE IS X(10).
```

That's better. Easier to write — easier to read. However, you can still do a couple things to simplify this statement. The short form of PICTURE is PIC, and the IS keyword is optional. Almost every COBOL programmer uses PIC for PICTURE, and I am sure that lots of programmers don't even remember that they can include an IS. Using these shortcuts, your long-word code looks like the following:

2000
Y2K

The millennium problem starts here

The millennium problem can appear in many different forms inside a COBOL program. Of all the COBOL faces of the millennium problem, this one is the most obvious and most common — a two-digit year is often defined along with a two-digit day and a two-digit month, like this:

```
01  Date.
    02  Month  PIC 99.
    02  Day    PIC 99.
    02  Year   PIC 99.
```

Defining the month and day this way is not a problem — neither of these numbers can ever go beyond two digits. The problem is with the year. Does 00 represent the year 2000 or 1900? As a matter of fact, does the year 98 stand for 1898, 1998, or 2098? This question is one of those things that is quick and easy for a human to figure out, but a computer just goes "duh." I tell you about other forms of the COBOL millennium problem, and how to fix them, in Chapter 17.

```
77  long-word PIC X(10).
```

X is not the only possible option. While an X creates a space that can hold any character, a 9 creates a space that can hold only a numeric digit. For example, the following code creates a location that can hold an eight-digit number:

```
01 some-number PIC 9(8).
```

If you suspect that data formatting in COBOL involves a lot more than these simple examples, you're right. COBOL has more control over the details of data formatting than any other language that has ever been devised.

The Symbols That Make the PICTUREs

Table 5-1 lists each of the symbols that you can use to define the PICTURE of a data item. This table is for quick reference. The sections following the table offer more complete explanations and examples for all of these symbols.

Table 5-1 PICTURE Clause Symbols

Symbol	*Meaning*
A	Creates a position for an alphabetic character
B	Creates a blank character position
P	Provides a placeholder for scaling numbers that have their decimal points beyond either the right or left end of the string of digits
S	Creates a position for the sign of a number
V	Marks the position of an implied decimal point in a numeric field but does not take up a character position
X	Creates a position that can be filled by any character
Z	Creates a position that can be filled by any numeric digit other than a leading zero
9	Creates a position for a numeric digit
0	Inserts the zero character into the string
/	Inserts the slash character into the string
,	Inserts the comma character into the string
.	Inserts the period character into the string, and marks the location of the decimal point
-	Creates a position for an optional sign character
+	Creates a position for a required sign character
CR	Creates a two-character position that appears as CR on negative numbers
DB	Creates a two-character position that appears as DB on negative numbers
*	Inserts an asterisk in place of a leading zero in a numeric field
$	Creates a position to display the currency sign (You can change the currency symbol to other characters.)

A is for alphabetic

Inserting an A symbol in a PICTURE clause limits the contents of the field to only alphabetic characters. Take a look at the following code, for example:

```
77  Name PIC A(30)
```

This statement creates a location called Name, which holds 30 characters. These characters all must be alphabetic — A through Z, a through z, or space.

Using the A symbol to define a Name this way may cause problems, because names sometimes include nonalphabetic characters. For example, typical human names have commas and periods for initials and titles and stuff. Some company names include numbers — I have even seen one with an exclamation point and another with an *at* sign. Nothing is sacred — mostly programmers just use X.

Asterisk (*) replaces leading zeros

The asterisk works like the Z symbol (which I describe a bit later in this chapter), except that it replaces leading zeros with asterisks instead of blanks. This whole asterisk thing is a sort of low-tech defense against forgery. For example, it can prevent someone from using a typewriter to alter the value of a check. Table 5-2 shows some examples of how the asterisks can be used to fill in the space that would otherwise act as hacker bait.

Table 5-2 Leading Zero Suppression with Asterisks

Value	*PICTURE*	*Result*
456.02	****9.99	**456.02
456.02	***,**9.99	****456.02
23456.02	***,**9.99	*23,456.02

As you can see from the table, the asterisk deals intelligently with embedded commas. If an asterisk is displayed to the left of a comma, the comma also becomes an asterisk.

B is for blank

Moving data into a field that contains one or more B symbols in its definition causes a blank to be inserted into the data in place of each B. Check out the following code, for example:

```
77  Telephone 999B999B9999.
. . .
    MOVE 1235553456 TO Telephone.
    DISPLAY Telephone.
```

Here's the resulting display:

```
123 555 3456
```

The blank characters are inserted without loss of data (unless you MOVE more characters than can fit in the specified location — in such cases, the characters on the right end of the data are omitted).

Comma (,) displays a comma character

A comma (,) in a PICTURE clause inserts a comma in the data. (It can also appear as a blank if it gets mixed up in zero suppression.) Commas are handy for formatting big numbers, as in the following example:

```
77  SomeBigNumber PICTURE ZZ,ZZZ,ZZZ,ZZZ.
. . .
    MOVE 8765432 TO SomeBigNumber.
    DISPLAY SomeBigNumber.
```

The output from the DISPLAY statement looks like this:

```
8,765,432
```

Using a comma in combination with a bunch of Zs causes the comma to disappear whenever the Z to the comma's left disappears. If the MOVE statement in the preceding example had been this:

```
    MOVE 5432 TO SomeBigNumber.
```

The output from the DISPLAY would look like this:

```
 5,432
```

Currency ($) positions the currency symbol

The currency sign appears as itself. You can also use it as a leading-zero suppresser. Table 5-3 shows some different ways that you can use the currency sign.

Table 5-3	Positions of the Currency Sign	
Value	*PICTURE*	*Result*
82.45	$99.99	$82.45
82.45	$99,999.99	$00,082.45
82.45	$ZZ,ZZ9.99	$ 82.45
4382.45	$ZZ,ZZ9.99	$4,382.45
82.45	$$$,$$9.99	$82.45
4382.45	$$$,$$9.99	$4,382.45
.45	$$$,$$9.99	$0.45

DB and CR (Debit and Credit) indicate negative values

DB and CR are two-symbol pairs that you can use as minus signs. They display as blanks for a positive value, and as themselves for a negative value. Table 5-4 lists some examples.

Table 5-4	Debit and Credit Formatting	
Value	*PICTURE*	*Result*
113.65	9999.99DB	0113.65
-113.65	9999.99DB	0113.65DB
-113.65	9999.99CR	0113.65CR
2314.82	ZZ,ZZZCR	2,314.82
-2314.82	ZZ,ZZZCR	2,314.82CR
-2314.82	ZZ,ZZZDB	2,314.82DB

Minus sign (-) displays a minus sign or a blank

You can use the minus sign whenever you want to make sure that negative numbers are tagged with a sign character. You can use it on the front or the back of the PICTURE clause. On the front, you can use it for zero suppression.

You can shape a number just about any way you want by using a combination of - symbols and Z symbols. Table 5-5 shows some of the combinations.

Table 5-5 Zero Suppression for Signed Values

Value	*PICTURE*	*Result*
85	-99	85
-85	-99	-85
85	-9999	0085
-85	-9999	-0085
85	999-	085
-85	999-	085-
85	-ZZZ9	85
-85	-ZZZ9	-85
85	----9	85
-85	----9	-85
85	ZZZZ9-	85
-85	ZZZZ9-	85-

As you can see from the table, the - symbol causes a sign character to be displayed only if the number is negative. A group of two or more leading - symbols causes all leading zeroes to become blanks (unless one of them is to be the minus sign). This technique allows you to snuggle the minus sign right up next to the number.

Nine (9) displays a digit

The symbol 9 makes a place for a single digit. It displays the digit as a character from 0 through 9.

P is for placeholder

You use the P symbol as a placeholder for numbers that have lots of trailing or leading zeroes — for example, big numbers:

```
77  BigOldNumber PIC 9999PPPPPPPP.
```

A BigOldNumber has only four digits of accuracy, but those four digits are followed by an implied eight zeros. The number 819600000000 fits nicely, but it is stored and displayed as 8196. Here's a more sane way to write the same thing:

```
77  BigOldNumber PIC 9(4)P(8).
```

The P symbol also works in the other direction. You can use the P symbol to define really small numbers, too. For example:

```
77  LittleOldNumber PIC P(10)9(4).
```

Following the decimal point, the LittleOldNumber has ten zeros and then four digits. For example, it can hold the number .00000000008196.

What all this means is that you can use the P symbol either at the beginning or the end of a number, not both. Its purpose is to specify the position of the decimal point when the decimal point is a long way off.

Period (.) displays a decimal point character

A period displays a decimal point in a numeric PICTURE, as in the following example:

```
77  Average PIC 999.99.
. . .
    MOVE 486.3 TO Average.
    DISPLAY Average.
```

The displayed number looks like this:

```
486.30
```

Plus sign (+) displays a plus or minus sign

You can use the plus sign whenever you want to make sure that both positive and negative numbers are tagged with a sign character. You can use this symbol on the front or the back of the PICTURE clause. On the front, you can use the plus sign for zero suppression.

You can shape a number just about any way you want by using a combination of + symbols and Z symbols. Table 5-6 shows some of the combinations.

Table 5-6 Zero Suppression for Signed Values

Value	*PICTURE*	*Result*
85	+99	+85
-85	+99	-85
85	+9999	+0085
-85	+9999	-0085
85	999+	085+
-85	999+	085-
85	+ZZZ9	+ 85
-85	+ZZZ9	- 85
85	++++9	+85
-85	++++9	-85
85	ZZZZ9+	85+
-85	ZZZZ9+	85-

As you can see from the table, the + symbol causes a sign character to be displayed whether the number is positive or negative. A group of two or more leading + symbols causes all leading zeroes — except the one that becomes the sign — to be displayed as blanks. This technique allows you to snuggle the sign character right up next to the number.

S is for sign

You use the symbol S to make a number signed — that is, it can be both positive and negative. This is really an odd bird. An S may or may not create a position for the sign character and, if it does create a position, even though the S is always on the left, the sign could wind up appearing on the right. It could be that the sign is embedded in one of the digits, causing a digit to display as an L instead of a 3 on a system that uses the letter L to represent a negative 3. It must have been very interesting in the meeting room the day that the committee designed this one.

The S does have advantages. It takes up less space and calculates quicker than other methods of defining the sign. The S works in conjunction with the SIGN clause. I show you how the S and the SIGN work a little later in this chapter. I don't want to talk about it right now; I have a headache (but if you must know, see the section "You're a Cute Number; What's Your SIGN?" later in this chapter).

Slash (/) displays a slash character

How 'bout a date? I can fix you right up. Lookie here:

```
77  Date PICTURE 99/99/9999.
. . .
    MOVE 11231998 TO DATE.
    DISPLAY Date.
```

A slash character (/) in a PICTURE clause inserts a slash in the data. The displayed output of the preceding example looks like this:

```
11/23/1998
```

V is for implied decimal point

The symbol V marks an implied decimal point in a numeric value. The implied decimal point doesn't take up a character position, and it is never displayed. Here's an example:

```
77  Tensor PIC 999V99.
```

This PICTURE clause defines a field that holds five-digit numbers, with two of the digits to the right of the decimal point, but the decimal point is not included as one of the characters. For example, the value 891.45 is represented as 89145. What happens here is that COBOL remembers where the decimal point goes so that it can get the arithmetic right.

X is for any character

The symbol X defines a position that can hold any character in the computer's character set. This symbol is the simplest and most commonly used of all the PICTURE symbols. Any time you need to create a location to store data that has no particular format, this symbol is the one to use.

You can use the X symbol for big chunks of data. For example, if you need a place to hold 4 kilobytes of data, just write the following code:

```
77  FourK PIC X(4096).
```

Z is for suppressing zeros

When you use a Z as the leading character — or characters — in the PICTURE clause of a field, all leading zeros in the number show up as blanks. The characters 1 through 9 appear as themselves, and 0 displays as a blank if no nonblank digits appear to its left. Table 5-7 shows how the use of the Z in a PICTURE clause affects the display of various numbers.

Table 5-7 Suppressing Leading Zeros

Value	*PICTURE*	*Result*
25	9999.00	0025
25	ZZZ9.00	25
5	ZZZ9.00	5
5	ZZZZ.00	5
0	ZZZ9.00	0
0	ZZZZ.00	*(blank)*
25	ZZZ.Z9	25.00
0.05	ZZ9.99	0.05
0.05	ZZZ.99	.05
0.05	ZZZ.Z9	. 5
0.05	ZZZ.ZZ	. 5
0.0	ZZZ.ZZ	*(blank)*
8105.0	ZZ,ZZZ	8,105
43	ZZ,ZZZ	43

As you can see from the table, the comma and the period are special cases with the Z. If a comma finds itself to the right of a Z that is being shown as a blank, the comma also shows as a blank. I suppose they are both members of the same union. The decimal point is also in the union, but has a slightly different attitude — it shows as a blank only if all digits are suppressed.

Zero (0) displays a zero character

A 0 character in a PICTURE clause inserts a zero into the data — for example:

```
77  Name   PIC XXOXXX.
77  Number PIC 990999.
. . .
    MOVE "Leroy" TO Name.
    MOVE 12345 TO Number.
    DISPLAY Name.
    DISPLAY Number.
```

`Name` contains enough `X` symbols to hold all the characters, and `Number` has enough `9` symbols to hold all the digits, but they both have that pesky zero in their middles. The preceding code results in the following display:

```
LeOroy
120345
```

Don't look at me like that. I have absolutely no idea what this can be used for. It seems about as useful as an ashtray on a motorcycle.

Identifying the Five Kinds of Data

Using combinations of the editing symbols that I list in Table 5-1, you can declare five kinds of data with a `PICTURE` clause:

- Alphabetic
- Alphanumeric
- Alphanumeric edited
- Numeric
- Numeric edited

You find a good deal of overlap between things you can do in the different data types, but each one has its own reason for living. You don't learn a secret handshake here — the only thing that differentiates one type from another is the set of formatting symbols you use in the `PICTURE` clause.

To write COBOL programs, you don't really need to know all the little details necessary to tell one data type from another. The five kinds of data are important for only two reasons. First, if you learn what they are, you can sound really smart in COBOL discussion groups. Second, if you want to do any kind of arithmetic, you need to create a numeric data type — the laws of COBOL prohibit arithmetic on any other type. But, hey, if you try to do arithmetic on the wrong type, the compiler politely points it out to you and you can sneak a quick peek at this page in the book.

Alphabetic

Alphabetic data is the simplest of the data types. The PICTURE clause of alphabetic data can contain only A, as in the following example:

```
77  LongName PIC A(32).
77  ShortName PIC A(6).
```

Alphanumeric

A PICTURE that is composed of only A, X, and 9 symbols is alphanumeric. No active editing takes place, as with alphanumeric edited, but some limitations exist on what certain character positions can hold. Check the following code, for example:

```
77  ZipPlusFour PIC 5(9)X4(9).
77  SS-Number PIC 999X99X9999.
77  Catcher PIC AAAAAX99.
```

Each of these fields is designed to hold some data that exists in a special format that is a mixture of two or more character types. The first one, ZipPlusFour, can hold a 9-digit zip code like 75243-4096. The standard form of a social security number, like 123-45-6789 fits into SS-Number. The last one is a very specialized format — it is designed to hold the title of the book *Catch-22*.

Alphanumeric edited

A PICTURE that contains at least one A or X and at least one of /, B, or 0 (it may also contain one or more 9 symbols) is alphanumeric edited. That is, it is the same as an alphanumeric type with one of the editing characters /, B, or 0. For examples of using alphanumeric edited, see the sections on slash, blank, and zero, earlier in this chapter.

Numeric

The PICTURE clause for a numeric data type can contain only the following symbols:

- 9
- +
- -
- P
- S
- V

Numeric data also has a size limit — a numeric PICTURE can contain no more than 18 digits. If you don't specify +, -, or S, the number is unsigned — that is, it can contain only positive values.

You can perform arithmetic on numeric data. In fact, numeric data is the only data type that can be used with ADD, SUBTRACT, MULTIPLY, and DIVIDE. Table 5-8 shows some examples of numeric data.

Table 5-8 Formatting for Numeric Data

Value	*PICTURE*	*Result*
12.42	99	12
-12.42	99	12
12.42	99999999	00000012
12.42	9(8)	00000012
12.42	9999V999	0012420
12.42	9(4)V9(2)	001242
123.456	PP999	600
123.456	999PP	001
1234.00	+9(5)	+01234
-1234.00	+9(5)	-01234
1234.00	-9(5)	01234
-1234.00	-9(5)	-01234

As you can see in the table, any data that doesn't fit inside the confines of the format is simply discarded. This fact is especially true of the P symbol, which can cause numbers to be shifted over and fall right off the end. A negative number becomes positive when it is moved into a PICTURE without a sign indicator.

Moral: Write the numeric PICTURE clause carefully — the format needs to be able to hold the complete range of values you may throw at it.

Numeric edited

A numeric-edited PICTURE can contain any of the symbols except A, X, and S. The A and X symbols are not numeric, and the S symbol is used with the numeric type to enable signed arithmetic. This data type is where you use the currency symbol, leading-zero suppression, and all the other fancy stuff to show the numbers off to their best advantage.

You can't do arithmetic on numeric-edited data. In fact, about all you can do with numeric-edited data is shove a number into it and display it. You need to do your arithmetic somewhere else. Generally, you create numeric fields to do all the work and then set up edited fields just for the display, as in this example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MathEdit.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  EditedForm PIC $$$,$$9.99DB.
77  NumericForm PIC S9(7)V9(2).
PROCEDURE DIVISION.
CALC-SHOW.
    MOVE -893.81 TO NumericForm.
    PERFORM SHOW-VALUE.
    MULTIPLY 2 BY NumericForm.
    PERFORM SHOW-VALUE.
    STOP RUN.
SHOW-VALUE.
    MOVE NumericForm TO EditedForm.
    DISPLAY NumericForm.
    DISPLAY EditedForm.
```

The NumericForm is made up from characters that allow arithmetic to be performed, but its display is pole-cat ugly. The EditedForm prohibits arithmetic, but has a pretty snazzy display. You can do all the arithmetic you want in the NumericForm, but whenever you want to show off the result, you shove it into the EditedForm. The output from this example looks like this:

```
00008938J
   $893.81DB
00017876K
 $1,787.62DB
```

Personally, I find the edited form a bit more, shall we say, intuitive. Are you curious about that use of J and K in this output? Well, because the S implies a sign but doesn't set aside space to hold it, COBOL encodes the sign internally by using certain letters for certain digits to indicate positive and negative. Apparently, this compiler uses J for 1 and K for 2. Clever, yes; attractive, no. Thinking about it, shouldn't it be T for 2?

You're a Cute Number; What's Your SIGN?

If you are in the mood for something peculiar, you are in the right place. I don't mean that this section discusses anything difficult to understand, or even that something is wrong with the SIGN clause — it's just peculiar. The SIGN clause only has effect on a numeric type that begins with the symbol S. The symbol S at the front of a numeric PICTURE string may or may not cause a sign character to appear, it can cause the number 3 to display as the character L, and it may or may not even create a character position for the sign — and if it does, the sign can be at either end of the number.

I've got an idea. Why don't I explain this whole thing with a list of brilliantly contrived examples that cunningly expose the complete set of possibilities? Okay, would you believe two or three examples that give you some idea of what is going on? I'll start with a simple signed 3-digit number:

```
77  SignedValue PIC S9(3).
```

The preceding line of code is exactly the same as the following:

```
77  SignedValue PIC S9(3) SIGN IS TRAILING.
```

This whole thing is only three characters long — no separate character position is set aside for the sign. You don't need one because the compiler handles negative numbers by making internal changes to the value you store. The most common method for handling negative numbers in this way involves the use of a code that maps certain digits to other characters.

Character conversions for numeric signs

A reason actually does exist for why certain letters were originally chosen to be the indicators of negative values. In the EBCDIC codes (the character encoding used by IBM mainframes), the character 1 can be converted to a J by toggling one bit. Toggling the same bit in a 2 converts it to a K, or a 3 to an L, and so on.

One odd characteristic of EBCDIC is other characters are stuck right in the middle of its alphabet, so toggling that bit on a 0 results in the character }. The following table shows the bit-patterns of the digits and the characters for both ASCII and EBCDIC.

Character	*EBCDIC*	*ASCII*
0	11110000	00110000
1	11110001	00110001
2	11110010	00110010
3	11110011	00110011
4	11110100	00110100
5	11110101	00110101
6	11110110	00110110
7	11110111	00110111
8	11111000	00111000
9	11111001	00111001
}	11010000	01111101
J	11010001	01001010
K	11010001	01001010
L	11010001	01001010
M	11010001	01001010
N	11010001	01001010
O	11010001	01001010
P	11010001	01001010
Q	11010001	01001010
R	11010001	01001010

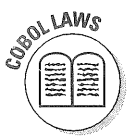
Toggling bits works okay for EBCDIC, but ASCII requires that the mapping be done with a table. To prevent this mapping, some ASCII implementations use a different set of characters.

To represent a negative number, the digit 1 becomes the letter J, the digit 2 becomes K, 3 becomes L, and so on. Because you have defined the sign as trailing (the default), you can just encode the last digit with the sign. For example, -141 is stored as 14J; and -142 is stored as 14K.

You can do the same thing on the front. Here's how you can put the sign into the leading digit:

```
77  SignedValue PIC S9(3) SIGN IS LEADING.
```

The same encoding applies; it just goes on the front instead of the back. For example, the value -350 is stored as L50 and -450 is M50.



Your mileage may vary. The particular encoding that I describe here is the most common, but it is not used by all compilers. The laws of COBOL specifically allow any particular compiler to encode the sign any way it chooses. Indicating whether a number is positive or negative takes only one bit, so it can be done in any number of ways.

The `SIGN` clause has one more option. It can be instructed to use an extra character for the sign. When this happens, no embedded character mapping occurs — the sign bit is just stuck on the front or back as an extra character. Take a look at these three different ways of defining `SignedValue`:

```
77  SignedValue PIC S9(3)
        SEPARATE CHARACTER.

77  SignedValue PIC S9(3) SIGN IS TRAILING
        SEPARATE CHARACTER.

77  SignedValue PIC S9(3) SIGN IS LEADING
        SEPARATE CHARACTER.
```

The first two are alike (because the `TRAILING` sign is the default), but all of these allocate a separate character for the sign. For example, with the sign trailing, the value 345 is stored as 345+, while -345 is stored as 345-. Leading signs work as you may expect — the value 345 is stored as +345 and -345 is stored as -345.

The words `IS` and `CHARACTER` are optional. The three previous examples can be written like this:

```
77  SignedValue PIC S9(3) SEPARATE.
77  SignedValue PIC S9(3) SIGN TRAILING SEPARATE.
77  SignedValue PIC S9(3) SIGN LEADING SEPARATE.
```

The USAGE Clause: Specifying How You Want Your Data Stored

A computer can use several different formats to store the same data. You can use the USAGE clause to tell the compiler exactly how you want it to store data (if you want to store it in something other than a string of characters). You may want to control the form of the data for any of several reasons:

- You may want to do this for efficiency because you know, on your particular machine, arithmetic on certain kinds of data is more efficient than others.
- You may want to do it to save space — if you save just one byte in a file that contains a million records, you save an entire megabyte.
- You may want to do it because you know that a particular format of data is the same on two computers and you want to be able to move your data from one machine to another.
- You may want to do it because you are a control freak and refuse to allow the compiler to make its own decisions.

A USAGE clause can be written on any data item except those of a 66 or 88 level. The PICTURE clause can contain only P, S, V, and 9 symbols.

If USAGE IS DISPLAY, it's the default

If you want something to be USAGE IS DISPLAY, just leave it alone — that's the default format. This format causes the data to be stored in a displayable character format. Each byte of the stored data contains a displayable character from the computer's character set.

If USAGE IS BINARY, it's base-2

Declaring USAGE IS BINARY creates a base-2 number. This is the normal representation of numbers in computers today, so it's no real strain on system resources to do this. Even though this format gives you a base-2 number on every system, they won't all be the same. Some systems store the bits from left to right, and others store them from right to left. Some swap the bytes around. It's weird.

The amount of storage used varies according to the number of digits you specify. You always have enough bits to hold the complete range of values in the PICTURE clause. The following list shows the maximum number of digits that can fit into each of these byte counts:

```
77  OneByte PIC 9(2) USAGE IS BINARY.
77  TwoBytes PIC 9(4) USAGE IS BINARY.
77  ThreeBytes PIC 9(7) USAGE IS BINARY.
77  FourBytes PIC 9(9) USAGE IS BINARY.
```

A single byte (8 binary bits) can hold any value up to 256, so it is only large enough to hold two digits of data. Two bytes of data can go up to a maximum of 65,535, so two bytes can only hold the full range of four digits. Three bytes can hold numbers up to seven digits, and four bytes can hold numbers up to nine digits. Although maximum and minimum ranges are different for negative numbers, it turns out that the digit counts are the same:

```
77  OneByte PIC S9(2) USAGE IS BINARY.
77  TwoBytes PIC S9(4) USAGE IS BINARY.
77  ThreeBytes PIC S9(7) USAGE IS BINARY.
77  FourBytes PIC S9(9) USAGE IS BINARY.
```

If USAGE IS COMP, it's probably binary

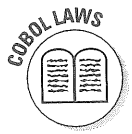
Specifying COMP (or COMPUTATIONAL) usage tells the compiler to use whatever method suits it best to store the data. This is quite often the same as BINARY, but it doesn't have to be — it can be any format the compiler writer decides to use to be most efficient on any particular computer.

This theme has lots of variations. Some computer systems have more than one nifty way of storing and handling data. Although standard COBOL only specifies COMP, you may see compilers with things like COMP-2 and COMP-3 being used to declare data in special formats. These names mean different things to different compilers, so you need to check the documentation on the one that you are using. One example of one of these names defining a special format is the IBM mainframe compilers that use COMP-3 to create packed decimal numbers.

If USAGE IS PACKED-DECIMAL, the size is cut in half

Declaring storage to be PACKED-DECIMAL can cause numbers to be stored in half (or almost half) the space of the default format. The digits are packed in like a bunch of sardines.

Here's how they do it: Normally, one character takes up a whole byte. However, because the data is numeric and all the characters are digits, only ten characters exist — well . . . 13 characters, counting the decimal point and the plus and minus signs. While a byte (8 bits) can hold 256 different characters, a half-byte (4 bits) can hold 16 different characters, so it's a simple matter of coming up with some sort of encoding for the characters and slipping two of them in each byte.



The COBOL standard does not specify the exact format for packed decimal data. It simply says that the representation must be in base-10 digits, and that each digit should take up as little space as possible. Exactly how this is done is up to the compiler writer. Some computers have some form of packed decimal implemented in hardware; this hardware feature dictates how the compiler works with packed decimals.

If USAGE IS INDEX, it's for use with OCCURS

The INDEX type is used for subscripting arrays. That's all. For examples and explanations of using the INDEX type, see Chapter 7.

This is a very special data type, and lots of rules exist about what you can and can't do with it. Because it is intended for one specific purpose, most of the laws fall into the "that's a no-no" category. For example, you can't declare one of these things with some of the more common specifiers, such as BLANK WHEN ZERO, JUSTIFIED, SYNCHRONIZED, VALUE, or even PICTURE. You can only refer to an INDEX type in a conditional statement, or in a SEARCH, SET, or USING statement — and it can be used as an index into an OCCURS array. You can't even DISPLAY it.

If you put it on a group, they all get it

It is okay to apply the USAGE clause to a whole group of items in one statement. Take this example:

```
01  Bounds USAGE IS COMP.
    02  Maximum PIC 9(5).
    02  Minimum PIC 9(5).
    02  Average PIC 9(5).
```

This code has exactly the same effect as if you had declared USAGE IS COMP on each member of the group, like this:

```
01  Bounds.
    02  Maximum PIC 9(5) USAGE IS COMP.
    02  Minimum PIC 9(5) USAGE IS COMP.
    02  Average PIC 9(5) USAGE IS COMP.
```

The JUSTIFIED Cause . . . er, Clause

Whenever data is moved into a nonedited, non-numeric field, the data is inserted beginning at the left-most position. For a reason unknown to anyone (except perhaps to a few retired keypunch operators), this is known as *justification.* Whenever things are placed against the left margin, they are known as left-justified; when they are placed against the right margin, they are right-justified.

The default for COBOL is to have the data left-justified and blank-filled on the right. This situation can be reversed by declaring a field as `JUSTIFIED RIGHT`. The short form of `JUSTIFIED` is just `JUST`, as the following example shows:

```
77  PapaBear PIC X(12).
77  BabyBear PIC X(12) JUST RIGHT.
. . .
        MOVE "Porridge" TO PapaBear.
        DISPLAY PapaBear.
        MOVE "Porridge" TO BabyBear.
        DISPLAY BabyBear.
```

The output from the two `DISPLAY` statements looks like this:

```
Porridge
    Porridge
```

Whenever `Porridge` is moved to `PapaBear`, it is justified left and blank-filled on the right. Moving `Porridge` to `BabyBear` justifies to the right and blank fills on the left because `BabyBear` is `JUST RIGHT`. Hey, don't look at me like that — I didn't make up the silly words for this language.

The BLANK WHEN ZERO Clause

The `BLANK WHEN ZERO` clause does exactly what it says it does. Adding this clause to the definition causes it to display all digits as blanks whenever the value is zero, as in the following example:

```
77  Blamout PIC 9(5) BLANK WHEN ZERO.
. . .
    MOVE 1 to Blamout.
    DISPLAY Blamout.
    MOVE 0 to Blamout.
    DISPLAY Blamout.
```

The first DISPLAY statement prints 00001. I would love to show you what the second one prints, but I can't — it's, well, like, invisible.

The Special Name CURRENCY

Let's talk money. By its very nature, COBOL likes to work the dollars. This is obvious from the fact that it recognizes the dollar sign in its PICTURE clause, as in the following example:

```
77  Expense PIC $999.99.
```

But COBOL — being the business-oriented language that it is — likes all kinds of money. It can be instructed to use F for French francs or £ for English pounds — of course, your computer has to have £ in its regular character set. You tell COBOL which currency symbol you want by setting the CURRENCY SIGN as one of the SPECIAL-NAMES.

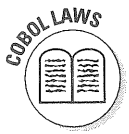
In fact, COBOL is willing to let you use any one of a number of symbols for the currency symbol. For example, the fictitious country of Slandovia uses the darnit for its currency. This example program shows how to set the ! symbol as the currency symbol, and how the symbol is then used in a PICTURE clause:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SlanCur.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    CURRENCY SIGN IS '!'.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  Expense PIC !,!!!,!!9.99.
PROCEDURE DIVISION.
0000-INIT.
    MOVE 44.10 TO Expense.
    DISPLAY Expense.
```

When this program runs, it displays the following output:

```
!44.10
```

This output represents 44 darnits and ten hecks.



According to COBOL law, certain characters cannot be used as the currency sign. You can't use any of the digits 0 through 9, nor any of the lowercase letters a through z. The uppercase letters A, B, C, D, P, R, S, V, X, and Z are prohibited, as are the following special characters:

- *
- +
- -
- ,
- .
- ;
- (
-)
- "
- =
- /

Any other characters are fair game. Of course, your compiler may be more restrictive. For example, MVS COBOL prohibits the characters `X'20'`, `X'21'` and even sometimes `G`, `N`, and `E`.

The Special Name DECIMAL-POINT

The roles of the comma and the period can be reversed. Some places in the world use the comma as the decimal point, and use the period to separate the digit-groupings for readability. COBOL has the capability to make this swap. For example, the value `34,561.98` can written as `34.561,98`.

The following example shows how you make the swap:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PeriodComma.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    DECIMAL-POINT IS COMMA.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  Expense PIC ZZ.ZZZ.ZZ9,99.
```

After the swap is made, it applies to numbers everywhere — even to literals in the PROCEDURE DIVISION. For example, the following two lines of code:

```
MOVE 29184362,81 TO Expense.
DISPLAY Expense.
```

display the following output:

```
29.184.362,81
```

Chapter 6

Literals, Constants, and Some Special Names

In This Chapter

- Defining numeric literals
- Working with nonnumeric literals
- Understanding the purposes of the predefined literals

A constant value — a name or number coded directly into a COBOL program — is called a *literal*. Two basic kinds of literals exist: numeric and nonnumeric. A numeric literal is a number, and a nonnumeric literal is some form of text.

COBOL gives you three ways to put a constant value into a field. First, you can use the VALUE clause to cause a field to take on an initial value when the program is compiled. Second, you can actively shove a value into the field while the program is running by using something like a MOVE or COMPUTE verb. Third, you can just leave the field alone and use whatever value the COBOL compiler decides to put in there for you.

The first two methods work just fine. The third method of putting a value into a field is frowned upon because the compiler normally just puts garbage in the field. Recent surveys have shown that most people don't like their programs to use garbage for data, so you may be better off sticking with one of the other methods for initializing variables.

This chapter shows you how to work with numeric and nonnumeric literals in your COBOL programs. The chapter also introduces figurative literals — COBOL keywords that have predefined values.

Playing the Numbers: Numeric Literals

As you write your programs, you will find yourself with an overwhelming urge to type in a number here and there. This number can be anything from the VIN number on your Henry J to the average weight of the American bald eagle. A number stuck into a COBOL program is called a *numeric literal.*

All numeric literals are made up primarily of the digits 0 through 9. They can have a + or - sign on the left, and they can include a decimal point. No commas or dollar signs are included in a numeric literal — you can include these editing characters in a PICTURE clause.

Here are some examples of numeric literals:

```
451    832.5    -22    +981.22
```

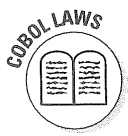
As shown in the following examples, you can use a literal in the VALUE clause:

```
01  BulletHoleCount PIC 9(5) VALUE 27.
01  AntsInPants PIC 9(3)V9(2) VALUE 88.12.
```

For a numeric literal to be used in a field's VALUE clause, the field must be numeric. It cannot be numeric edited. A numeric-edited field is a nonnumeric field — it holds a literal string of characters. Look to Chapter 5 for an explanation of the difference between numeric and numeric-edited data types. I explain the initialization of a numeric-edited field in the next section of this chapter.

It is not necessary to use a VALUE clause to poke a literal value into a field. It can be done while the program is running. The most common way to place a literal value into a field is with the MOVE statement, as in the following example:

```
MOVE 241 TO BulletHoleCount.
MOVE 22.87 TO AntsInPants.
```



Creating and using COBOL numeric literals is straightforward and intuitive — except for one little thing. The decimal point cannot be the last character in a numeric literal. That is, the following number is valid:

```
897.0
```

But the following number is not valid:

```
897.                                    *** error ***
```

This rule is necessary because of the tyrannical power of the *period* (.) in COBOL. This example shows what I mean:

```
SUBTRACT 82. FROM SomeValue.          *** error ***
```

The period following the 82 in the preceding code ends the COBOL sentence. The compiler then moves on to try to start a new sentence and discovers FROM. At this point, the compiler generates one of those rude "What the heck is this?" error messages. But that's not the worst of it. In some cases, you can put a period at the end of a number and the compiler (instead of generating an error message) misreads your intentions completely and generates something you never intended. This error can be one of those 3 a.m. head-scratcher problems. Here's an example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CrazyEight.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  Eight PIC 99 VALUE 8.
77  Seven PIC 99 VALUE 7.
77  EightDoubled PIC 999 VALUE 0.
PROCEDURE DIVISION.
CrazyEight-MAIN.
    IF Seven IS GREATER THAN Eight
        COMPUTE EightDoubled = Eight * 2.
        DISPLAY EightDoubled.
    STOP RUN.
```

The indentation of the DISPLAY statement suggests that the programmer wants to have the COMPUTE and DISPLAY statements executed whenever the conditional of the IF is true. The code is written so that the conditional expression is always false, but the program displays 000 — every time. The code generates this output because the period following the 2 ends the sentence and thus ends the IF block. Two quick ways exist to fix the program — both of which remove the period from the end of the COMPUTE statement. First, you can simply remove it, like this:

```
COMPUTE EightDoubled = Eight * 2
```

Or you can rewrite the literal so that it doesn't end with a decimal point, as in the following code:

```
COMPUTE EightDoubled = Eight * 2.0
```

Stringing Together Some Nonnumeric Literals

A *nonnumeric literal* is a string of quoted characters, like those shown in the following example:

```
"I am a nonnumeric literal"
```

A nonnumeric literal — also called *character literal* or a *character string,* or sometimes just *string* — can contain any character in the set of characters known to your COBOL compiler. All the following quoted strings are nonnumeric literals:

```
"You ain't kiddin'"
"Then he said #@!^*, so I hit him."
"972.41"
"If a > b OR c < d THEN SET m to 44."
```

COBOL doesn't do anything with the stuff inside the quotes. It doesn't even look at those characters. COBOL just takes the whole string and puts it wherever you say to put it. Of course, the act of putting it somewhere may cause COBOL to fiddle with it a bit. For example, if you have a quoted literal that is 50 characters long and you `MOVE` it to a location that holds only 20 characters, COBOL discards the other 30 characters to make the literal fit.

Because a string of characters is tagged on the left and right with the double-quote character, you can't just stick one in the middle of a string and expect anything worthwhile to come of it. The following code won't work:

```
"Billy "Joe Bob" Shakespeare"
```

To put a double-quote character inside a string, use a pair of double quotes, as in the following example:

```
"Billy ""Joe Bob"" Shakespeare"
```

The compiler looks at this code and figures out what you mean. It knows that a literal starts and ends with double quotes. As a special case, it will not end a literal with a pair of double quotes. Whenever a pair of double quotes appear inside a string, they are transformed into one double-quote character that is inside the string. This convention results in the output you were after in the first place:

```
Billy "Joe Bob" Shakespeare
```

Numeric-edited fields and the VALUE clause

Including one or more editing characters in a numeric field makes it become a numeric-edited field. Although numeric literals can be used in the VALUE clause of a numeric field, a numeric literal cannot be used in the VALUE clause of a numeric-edited field. This code doesn't work:

```
01  MonthlyGoal VALUE $$,$$9.99 VALUE 250.00. ** Error **
```

Because the field is an edited form, you need to define an appropriately edited literal value, like this:

```
01  MonthlyGoal VALUE $$,$$9.99 VALUE "   $250.00".
```

The term *numeric-edited* is probably a bad choice of words. A more appropriate name may be *character-string-that-edits-numbers.*

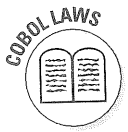
A MOVE statement converts a numeric literal to the character form of the edited format. For example, here's how you can MOVE a numeric literal into the edited MonthlyGoal field:

```
MOVE 250.00 TO MonthlyGoal.
```

On the other hand, don't MOVE a character literal into an edited literal field. The results are not predictable — for example, this may or may not work:

```
MOVE "$250.00" TO MonthlyGoal.
```

Double or single? And how long do you want it?



The COBOL 85 standard states that nonnumeric literals must be enclosed in double-quote characters. It also states that two double-quote characters in a row are to be interpreted as a single literal character. It also states that a compiler can allow strings to be of any length, but that they must allow for a minimum of 160 characters.

But the COBOL standard never once mentions using apostrophes (*single quotes,* if you prefer) to enclose a string. Since the dawn of COBOL, however, using apostrophes has been common practice. It seems that some primitive character sets didn't have double quotes, so the apostrophe was drafted to do the job. The practice continues to this very day. A compiler may exist somewhere that does not allow you to create strings using apostrophes for quotation marks, but I have never seen one. It could be that the COBOL standard has just gone into denial and is sulking.

Moving a literal to an edited field

When you move a literal string of characters into a numeric-edited location, COBOL edits the digits according to the `PICTURE` clause. For example:

```
77  HooHa PIC 99,999.99.
. . .
    MOVE "88" TO HooHa.
```

The preceding code causes COBOL to analyze the string to convert it to an edited number. The receiving field winds up containing the following number:

```
00,088.00
```

Not all compilers are rocket scientists about this process. Using the same `HooHa`, I tried the following statement on some different compilers:

```
    MOVE "ABC" TO HooHa.
```

The results differed from one compiler to the next. A couple compilers told me that I was wrong to mess around like that and I shouldn't do it again. Some of them just let it go through. Even those compilers that complained when I tried to put the string directly into the numeric field allowed me to put it in there indirectly, like this:

```
77  HooHa PIC 99,999.99.
77  Exes PIC X(3).
. . .
    MOVE "ABC" TO Exes.
    MOVE Exes TO HooHa.
```

In all cases, my attempt to treat letters like digits produced the following result:

```
00,ABC.00
```

The moral of this story is that you need to be careful about the contents of string literals that you shove into numeric-edited locations. COBOL may do what you tell it to do instead of what you want it to do.

Don't MOVE pre-edited literals into a numeric-edited field. For example, the following code may not work:

```
01  TotalCost PIC $,$$9.99.
. . .
    MOVE " $250.00" TO TotalCost.
```

The resulting string in TotalCost is unpredictable. You may succeed with some compilers, but some others may fail. Let COBOL do the numeric editing.

Figuring Out Figurative Literals

A figurative literal is sort of like a built-in constant value that you refer to by name. As I detail in the following sections, a figurative literal is a COBOL keyword that has a predefined value.

ZERO, ZEROS, and ZEROES

The COBOL keywords ZERO, ZEROS, and ZEROES mean nothing. That is, they have value; it's just that the value they have is nada, naught, nothing, zip, zed, the big goose egg. Although these words come in singular and plural versions, they don't differ from one another. All of the following MOVE statements are identical:

```
77  AverageIQ PIC 9(4).
. . .
    MOVE ZERO TO AverageIQ.
    MOVE ZEROS TO AverageIQ.
    MOVE ZEROES TO AverageIQ.
    MOVE ALL ZERO TO AverageIQ.
    MOVE ALL ZEROS TO AverageIQ.
    MOVE ALL ZEROES TO AverageIQ.
```

COBOL doesn't care how you spell ZERO. The addition of the word ALL is one of those COBOLisms that doesn't mean anything to the compiler, but it does seem to add certain emphasis, doesn't it? Just think of the word ALL as being built in to ZERO.

If this keyword seems a little too straightforward to last, you're right. Some odd things can happen. You can MOVE ALL ZEROES to a record and every field in the record will be initialized with the zero character. Fine. But you need to be sure that it makes sense to do this. This program offers an example of what can happen if the data types don't match:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ThreeNumbers.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  ThreeNumbers.
    02  FirstNumber PIC 9(4).
    02  SecondNumber PIC ZZ,ZZ9.99.
    02  ThirdNumber PIC 9(4) COMP.
PROCEDURE DIVISION.
ThreeNumbers-MAIN.
    DISPLAY "--Move One at a time--"
    MOVE ZERO to FirstNumber.
    MOVE ZERO to SecondNumber.
    MOVE ZERO to ThirdNumber.
    DISPLAY FirstNumber.
    DISPLAY SecondNumber.
    DISPLAY ThirdNumber.
    DISPLAY "--Move as a bunch--"
    MOVE ZERO TO ThreeNumbers.
    DISPLAY FirstNumber.
    DISPLAY SecondNumber.
    DISPLAY ThirdNumber.
    STOP RUN.
```

The preceding code first sets each of the fields to ZERO and then prints each one. The program then sets the whole record to ZERO in one fell swoop and prints the results. Here's what comes out:

```
--Move One at a time--
0000
     0.00
   0
--Move as a bunch--
0000
000000000
2336
```

The first three lines of code in the preceding example look about like you'd expect. (Well, this particular compiler automatically suppresses leading zeroes when printing COMP data, but it's within its rights to do so.)

The real fun begins in the second half of the output:

- The value displayed by FirstNumber is just what I expect — this field is defined as a four-character number, and those four characters are all zeroes.
- The SecondNumber doesn't fare as well. All the comma and period editing stuff is gone, and the leading zeros aren't suppressed. The MOVE ZERO statement just ignores the formatting and shoves a bunch of zero characters into this field.
- But the ThirdNumber is where things really get fouled up. You see, ThirdNumber is a COMP number — which means that COBOL uses some sort of internal binary form to represent the value — and the bit-patterns of the displayable zero characters get shoved into it. Your friendly neighborhood DISPLAY statement (mistaking the underlying bit patterns representing the characters as a binary number) dutifully converts the value it finds in ThirdNumber and you get . . . well, garbage.

SPACE and SPACES

The keywords SPACE and SPACES mean the same thing. They are synonyms for the space character. All the following MOVE statements do exactly the same thing:

```
77  SpaceRanger PIC X(10).
. . .
    MOVE " " TO SpaceRanger.
    MOVE SPACE TO SpaceRanger.
    MOVE SPACES TO SpaceRanger.
    MOVE ALL " " TO SpaceRanger.
    MOVE ALL SPACE TO SpaceRanger.
    MOVE ALL SPACES TO SpaceRanger.
```

Spaces and numbers don't really mix well. Here is an example that shows the sort of thing that can happen when you force spaces into a numeric field:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SpaceNumber.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  Frick.
    02  BinaryNumber PIC 9(4) BINARY.
    02  DisplayNumber PIC 9(4).
    02  EditedNumber PIC ZZZ,ZZ9.99.
    02  CompNumber PIC 9(8) COMP.
PROCEDURE DIVISION.
Mainline.
    MOVE SPACES TO Frick.
    DISPLAY BinaryNumber.
    DISPLAY DisplayNumber.
    DISPLAY EditedNumber.
    DISPLAY CompNumber.
```

COBOL won't let you `MOVE SPACES` directly to a numeric type, but you can fool it by moving the spaces into a record that contains numeric values. The output from the example looks like this:

```
8224


00000000
```

If you move spaces into a field that is supposed to hold a number, you get garbage. If the numeric field is `COMP`, you get your completely unexpected and meaningless-value type garbage. If the field is numeric-edited, all the editing characters vanish and you get a sort of blank-looking garbage. If you

move spaces into a regular numeric field, you get a sort of invisible garbage that, when arithmetic is performed on it, produces a whole new range of numeric garbage. On the other hand, if you really *want* garbage — this is a good way to get it.

QUOTE and QUOTES

To declare a literal character, and to put that character in a variable, you must enclose that character in quotes, like this:

```
01  TheLetterJ PIC X VALUE "J".
```

To declare a literal quote character, you must put the quotation mark in quotes, but the only way to quote a quote is to use quote-quote inside of the quotes (instead of just a quote), like this:

```
01  TheQuoteCharacter PIC X VALUE """".
```

Actually, COBOL gives you a better way to declare a literal quote character. As shown in the following example, you can use the keyword QUOTE or QUOTES (they both mean the same thing):

```
01  TheQuoteCharacter PIC X VALUE QUOTE.
```

The following MOVE statements are all equivalent:

```
77  FourQuotes PIC X(4).
. . .
    MOVE """""""""" TO FourQuotes.
    MOVE ALL """ TO FourQuotes.
    MOVE QUOTE TO FourQuotes.
    MOVE QUOTES TO FourQuotes.
    MOVE ALL QUOTE TO FourQuotes.
    MOVE ALL QUOTES TO FourQuotes.
```

Each of the preceding MOVE statements places four quote characters into FourQuotes. The first MOVE statement has ten quote characters in a row — the first and last ones are the beginning and ending quotes, and the COBOL compiler collapses the eight in the middle (each pair is combined into one) into a string of four quotes. The modifier ALL causes any character to be replicated in the receiving field, and the QUOTE and QUOTES keywords have an ALL built into them. The following statement produces different results from those of the previous examples:

```
    MOVE """ TO FourQuotes.
```

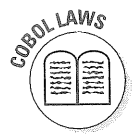
This statement puts a quote character into the first position of the field and a blank into the others.

Even though QUOTE is a quote character, you can't use it for quotes. For example, you can't write the literal "I'm literal" as follows:

```
QUOTE I'm literal QUOTE                    ** Error **
```

LOW-VALUES and HIGH-VALUES

If you want to set the contents of a field to its largest or smallest possible value, you have come to the right place. This feature can be very handy when you're sorting and you want to make sure that a certain field is either first or last in the sort order.



The name LOW-VALUES is a predefined constant of a value that has all its bits set to zero and is guaranteed to be smaller than any other character or value in COBOL. HIGH-VALUES is just the opposite — this predefined constant has all its bits set to one and is guaranteed to be larger than any other character or value in COBOL.

Like the other figurative constants, LOW-VALUES and HIGH-VALUES have an implied ALL built into them. The following two statements are identical:

```
MOVE LOW-VALUES TO SomeField.
MOVE ALL LOW-VALUES TO SomeField.
```

The following two statements are also identical:

```
MOVE HIGH-VALUES TO SomeField.
MOVE ALL HIGH-VALUES TO SomeField.
```

The capability to contain LOW-VALUES and HIGH-VALUES is valid for all the data types defined as part of the COBOL 85 standard. In fact, it should work for all types of fields — but your mileage may vary. It could be that your compiler has a special data type of its own (for example, COMP-3), which may be an exception. It would be best to check your compiler manual before you do anything exotic.

The SPECIAL-NAMES Clause

The SPECIAL-NAMES clause comes about as close to customization of the language as any self-respecting COBOL compiler allows. One common use of SPECIAL-NAMES is to change the character used for the currency sign or decimal point, as I demonstrate in Chapter 5. You can use the SPECIAL-NAMES clause to define special symbols that you use in the program. Probably the most common use of SPECIAL-NAMES is to mess around with the order in which characters of the alphabet are sorted, as I demonstrate in Chapter 16.

The truth is that you are very unlikely to use much of this stuff. Mostly, the things you can do with the SPECIAL-NAMES clause depend on the implementation — that is, each COBOL compiler can come up with its own definitions for things that you can do with the SPECIAL-NAMES clause. For example, a compiler may allow the following statement:

```
    DISPLAY "Hello" UPON CONSOLE.
```

In this case, the word CONSOLE — which is not a COBOL reserved word — has some special meaning to the particular compiler. The capability to use the name in this way implies that it may be possible to use some other name — something like this:

```
    DISPLAY "Hello" UPON LOCAL-SCREEN.
```

The next logical step is to be able to have some sort of global control over which screen is to receive the displayed output. For example, you can write the DISPLAY statement this way:

```
DISPLAY "Hello" UPON StandardOut.
```

It would then be possible to specify a device to receive the output, as in the following example:

```
SPECIAL-NAMES.
    CONSOLE IS StandardOut.
```

In some cases, it may be possible to redirect the output to a printer, as the following code does:

```
SPECIAL-NAMES.
    PRINTER IS StandardOut.
```

You need to check the documentation of your compiler for the possibilities.

If you find yourself with the job of locating some obscure bug and, when you pick up a copy of the offending program, you see some stuff in the SPECIAL-NAMES clause that you don't understand, it is time to take a break. Find a quiet place, get yourself something cool to drink, get out the compiler manual, take your shoes off, and slowly try to get a clear idea of what the programmer has done. Things defined in the SPECIAL-NAMES clause can cause other parts of the program to PERFORM TWILIGHT-ZONE UNTIL WEIRD.

Chapter 7

Several Things in One Place and Several Places for One Thing

In This Chapter

- Creating arrays by using OCCURS
- Addressing array members with indexes
- Setting and manipulating the values of indexes
- Defining arrays within arrays
- Setting initial values for tables and arrays

Generally speaking, data items don't come along one at a time. They always seem to come in bunches. In some cases, each one of these data items is unique — like the combination of a name, address, phone number, and so on. When this happens, the data can be neatly packaged into a record, as I describe in Chapter 4. In other cases, all the data items are alike; like a list of daily temperatures for the month or the names on Santa's naughty-kids list. Chapter 4 discusses how to handle the all-different case; this chapter describes how to work with the all-alike data.

When you have a bunch of pieces of all-alike data, you can store them in an array instead of a record. Each member of the array has the same name, so you have to distinguish them individually by a number. This number is called an index and is used to specify the position of an item of data that is stored in the array. This chapter shows you how to construct an array and use it to store data.

Using the OCCURS Clause to Define Arrays

It seems to be a natural tendency on the part of business data processing to keep things in tables. Like the example in Table 7-1, a table consists of rows and columns, and each column has a name at its head.

Table 7-1 Mayhaw Jelly Production

Region	*Last Month*	*This Month*	*12-Month Average*
North	43	46	40
West	81	78	88
South	71	72	70

Table 7-1 has four columns and three rows. You can represent this table in several ways in COBOL. The following code shows one way:

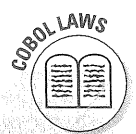
```
01  MAYHAW-JELLY-PRODUCTION.
    03  Region              PIC X(10) OCCURS 3 TIMES.
    03  LastMonth           PIC 9(4) OCCURS 3 TIMES.
    03  ThisMonth           PIC 9(4) OCCURS 3 TIMES.
    03  TwelveMonthAverage  PIC 9(4) OCCURS 3 TIMES.
```

The OCCURS keyword, along with its count, creates multiple copies of the field with which it is associated. That is, the preceding example has three distinct fields named Region — one for "North," one for "West," and one for "South." The example also has three LastMonth fields — one for 43, one for 81, and one for 71. The same is true for the ThisMonth and TwelveMonthAverage fields.

The layout in the preceding example works just fine, but you can do the same thing in other ways — some are worse, some are better, and some are just different. Making the code worse would be silly. I prefer this approach:

```
01  MAYHAW-DATA.
    03  MAYHAW-JELLY-PRODUCTION OCCURS 3 TIMES.
        05  Region              PIC X(10).
        05  LastMonth           PIC 9(4).
        05  ThisMonth           PIC 9(4).
        05  TwelveMonthAverage  PIC 9(4).
```

This example seems to be a bit friendlier than the first version. The OCCURS clause appears only once instead of on every field. It is certainly simpler to add a new region to this version — you have to modify only one OCCURS clause.



According to the standard definition of COBOL, an OCCURS clause cannot appear on a data item with an 01, 66, 77, or 88 level. In other words, an OCCURS clause can appear only on an item that is inside a record — levels 02 through 49. If you find code that breaks this law and has an OCCURS clause on an 01 or 77 level, don't be surprised. Not all compilers enforce this particular law. If you decide that you must know whether you can get away

with breaking this law, try it — maybe you, too, can break this COBOL law. I, however, obey the law throughout this book because I don't like being yelled at by the COBOL cops.

The preceding examples store a previously computed value (the average of the monthly production for the past year) in the table. To build the table, those programs must calculate the average and then stick that calculated average into the table. It would probably make more sense to store all the production information for the past 12 months and calculate whatever I need from that information. To accomplish this, I can shape the table as follows:

```
01  MAYHAW-DATA.
    03  MAYHAW-JELLY-PRODUCTION OCCURS 3 TIMES.
        05  Region PIC X(10).
        05  MonthlyProduction PIC 9(4) OCCURS 12 TIMES.
```

This example has an OCCURS within an OCCURS. I have three regions and each region has 12 months of production numbers — a total of 36 monthly production numbers. The average is no longer included, but I don't need it because it can be calculated from the data that I have. To actually mess around with data (to stick data into the table and get it back out again) you need subscripting, which I discuss in the following section.

Accessing the Data in an Array

To address a single member of an item defined by an OCCURS clause, you use the number for that item. The first item is number 1, the second is number 2, and so on. A number used for this purpose is called a *subscript* or an *index*. For example, by using the appropriate index (or subscript) number, you can access the production data for a specific month and region in the table I discuss in the preceding section of this chapter.

Simple indexing with an integer constant

Here's one way to create a record that can hold the names of the 50 states:

```
01  State.
    02  Name1 PIC X(30).
    02  Name2 PIC X(30).
    02  Name3 PIC X(30).
    . . .
    02  Name50 PIC X(30).
```

Naming each individual field this way does work, but the following example offers a much better way to do the same thing. This code declares a storage location capable of holding the names of 50 states:

```
01  State.
    02  Name PIC X(30) OCCURS 50 TIMES.
```

You can address the name of each state by using the number of its subscript or index. Here's how you can insert the names of the states into the array:

```
MOVE "Alabama" TO Name(1).
MOVE "Alaska" TO Name(2).
Move "Arkansas" TO Name(3).
. . .
```

And it's just as easy to get the data out as it is to put the data into the array. You can display the names of the states like this:

```
DISPLAY Name(1).
DISPLAY Name(2).
DISPLAY Name(3).
. . .
```

By the way, you can include the `OF` and `IN` qualifiers — which I describe in Chapter 4 — if you need them. For example, you can insert a name into the array and then display it like this:

```
MOVE "Alaska" TO Name OF State(2).
DISPLAY Name OF State(2).
```

Note the position of the parenthesized subscript. It seems to be on the wrong code element — after all, the `OCCURS` clause is on `Name`, not `State`. The subscript is sort of like the first name and last name of a person. Whenever you use `OF` or `IN`, it's like you're using the given name and the family name of the field. For example, if you have a bunch of guys named George Washington, you can either say `George(3)` or `George Washington(3)`. Same guy.

Using a data item as a subscript

It's okay to use a numeric data item as an index. It's even easy to do. Here's an example program that allows the user to enter the numbers for a five-day Mayhaw forecast, and then uses the data to display a summary report:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. FiveDayForecast.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  Counter PIC 9.
01  Forecast.
    02  Mayhaws PIC 9(2) OCCURS 5 TIMES.
77  Total PIC 9(4).
PROCEDURE DIVISION.
Begin.
    PERFORM VARYING Counter FROM 1 BY 1
            UNTIL Counter > 5
       DISPLAY "Enter day " Counter " amount:"
               NO ADVANCING
       ACCEPT Mayhaws(Counter)
    END-PERFORM.
    MOVE ZERO TO Total.
    PERFORM VARYING Counter FROM 1 BY 1
            UNTIL Counter > 5
        DISPLAY " Day " Counter ": " Mayhaws(Counter)
           NO ADVANCING
        ADD Mayhaws(Counter) TO Total
    END-PERFORM.
    DISPLAY " ".
    DISPLAY "      Total: " Total.
    STOP RUN.
```

This example uses the same counter in two loops. The first loop accepts the data and the second one displays it. Both loops use the numeric value of `Counter` as a subscript to reference the individual members of `Mayhaws`. (Chapter 9 has more information about `PERFORM` and looping.)

The following display from a run of this program shows some input and its resulting output:

```
Enter day 1 amount:42
Enter day 2 amount:19
Enter day 3 amount:02
Enter day 4 amount:44
Enter day 5 amount:31
 Day 1: 42 Day 2: 19 Day 3: 02 Day 4: 44 Day 5: 31
      Total: 0138
```

You can declare the Counter in this example in several different ways. Any numeric (not numeric-edited!) data item works. Here are some of the ways in which you can declare the Counter:

```
77  Counter PIC 99 COMP.
77  Counter PIC 99 BINARY.
77  Counter USAGE INDEX.
```

That third line in the preceding example is a bit special, and is actually the best of the bunch. USAGE INDEX allows the COBOL compiler itself to choose whatever type it likes to use for romping about in arrays. You can use any one of these statements in the previous example — simply change the Counter declaration.

When you use a numeric item for subscripting, it's a good idea to declare it as COMP or BINARY if you are not going to declare it as an INDEX. The process that COBOL goes through to work with a subscript value can be a bit complicated, and COBOL normally works more efficiently with COMP or BINARY than it does with just regular old default DISPLAY. It's best to use an INDEX type instead of a numeric type — the INDEX type tells the compiler exactly what you are going to do and allows it to be a little smarter about the way it goes about working with the subscript.

Using INDEX or INDEXED BY

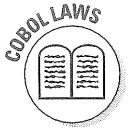
You have two ways of creating an INDEX data type. You can declare one separately from the OCCURS array, or you can use INDEXED BY as part of the OCCURS declaration. If you declare the index as a separate data item, you can use it with any array; creating it with the INDEXED BY clause dedicates the index to that one array.

You can create the array and the index for the example in the preceding section like this:

```
77  Counter USAGE INDEX.
02  Mayhaws PIC 9(2) OCCURS 5 TIMES.
```

Note that the Counter has no PICTURE clause. COBOL knows how to create indexes without you having to tell it anything else. By letting the compiler decide on the exact format of the data, you are guaranteed to get the most efficient of all possible indexes. The following code shows another way to do the same thing:

```
02  Mayhaws PIC 9(2) OCCURS 5 TIMES
        INDEXED BY Counter.
```



Using the INDEXED BY clause creates an index that is limited to the OCCURS array for which it is declared. Well, to be fair, I suppose I should tell you that not all COBOL compilers worry about where it's declared — some of them let you declare an INDEXED BY on one array and use it on another. I think it would be just fine for you to go ahead and do this, unless, of course, you want to be able to figure out what your program is doing when you read it next week.

Sometimes, you need to look at more than one member of the array at a time. For example, you may need to sort an array, which means you need to compare one member to another and then play swapsies. (By the way, sorting is one of COBOL's best tricks, and I devote all of Chapter 16 to this topic.) You can be in two places at once by using two indexes, and you can create them this way:

```
02  Mayhaws PIC 9(2) OCCURS 5 TIMES
          INDEXED BY Counter, ReCounter.
```

In fact, you can have all the indexes you want. You can declare some on the OCCURS clause and some others in other places. And you can mix and match the declarations — you can use one type of index in one place and another type in another place. COBOL is an equal opportunity indexer.

Spend some time experimenting with indexes. Subscripting is one of the two areas in COBOL (the other being keyed files) that seem to cause beginners the most problems. The problems don't occur because subscripting is that difficult; it's just a new way of looking at things. Create simple programs from the code in this chapter and experiment with them. The compiler tells you when you do something that isn't right, and the output from the program tells you what is going on when your program runs. Subscripting is important, and any amount of time you spend learning it will pay you huge dividends in the future — both in COBOL and in the next language you decide to learn.

How to diddle with the values of an INDEX data type

Index variables can be modified by the VARYING clause on a PERFORM statement (as I describe in Chapter 9), or by using the SET verb as I describe here. Placing values in an INDEX variable (and using the value) differs a bit from performing these tasks with other variables. For example, you can't use MOVE or COMPUTE to put values into an index.

You can only modify the value of an INDEX by using SET or VARYING. You can only access the value of an index by using it as a subscript, by numerically comparing it to another value, or in a VARYING FROM phrase. You can't qualify index names with OF or IN. I can see that question mark over your head, and all I have to say is, "I don't know why either, but it's the law." This must be part of the law that came to us on clay tablets.

Take a simple index like this one:

```
77  Counter USAGE INDEX.
```

To stuff a value into an index, use the SET verb, as in this example:

```
SET Counter TO 12.
```

You're not limited to constants. You can use a numeric variable, or another INDEX, to set the value, like this:

```
SET Counter TO MinimumValue.
```

Remember how you call the plays on the COBOL football team: SET to the left; MOVE to the right:

```
MOVE A TO B.
SET B TO A.
```

Both of these statements take the value from A and copy it into B. Getting these confused can cause you to lose complete track of your original line of scrimmage, and your team will beat itself.

You also use the SET verb to add or subtract values to an index. That's right; you can't use ADD or SUBTRACT. As you can see in the following examples, you use the much more intuitive terms of UP and DOWN — UP meaning addition and DOWN meaning subtraction:

```
SET Counter UP BY 2.
SET Counter UP BY IncrementAmount.
SET Counter DOWN BY 1.
```

Placing tables within tables

An OCCURS clause can contain another OCCURS clause. This table-nesting feature can be very handy in an attempt to describe the real world. The following program contains nested tables that describe an apartment complex:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ApartmentComplex.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  BIndex PIC 9(4) COMP.
01  ApartmentComplex.
    03  Street OCCURS 5 TIMES INDEXED BY SIndex.
        05  StreetName PIC X(20).
        05  Building OCCURS 12 TIMES.
            07  BuildingAddress PIC 9(3).
            07  Apartment OCCURS 4 TIMES INDEXED BY AIndex.
                09  UnitNumber PIC 9(4).
                09  NumberOfBedrooms PIC 99.
                09  NumberOfBathrooms PIC 99.
                09  NumberOfOccupants PIC 99.
77  NumberMaker PIC 9(3).
PROCEDURE DIVISION.
Begin.
    MOVE 100 TO NumberMaker.
    PERFORM VARYING SIndex FROM 1 BY 1 UNTIL SIndex > 5
        PERFORM VARYING BIndex FROM 1 BY 1
            UNTIL BIndex > 12
            PERFORM VARYING AIndex FROM 1 BY 1
                UNTIL AIndex > 4
                MOVE NumberMaker TO UnitNumber OF Apartment
                    OF Building OF
                    Street(SIndex,BIndex,AIndex)
                ADD 1 TO NumberMaker
            END-PERFORM
        END-PERFORM
    END-PERFORM.
    PERFORM VARYING SIndex FROM 1 BY 1 UNTIL SIndex > 5
        PERFORM VARYING BIndex FROM 1 BY 1
            UNTIL BIndex > 12
            PERFORM VARYING AIndex FROM 1 BY 1
                UNTIL AIndex > 4
                DISPLAY UnitNumber(SIndex,BIndex,AIndex)
            END-PERFORM
        END-PERFORM
    END-PERFORM.
    STOP RUN.
```

The table ApartmentComplex holds some information about a large apartment complex. The complex has five streets. Each street is named and has 12 buildings. Each building has an address and contains four apartments. Each apartment has a unit number, a number of bedrooms, a number of bathrooms, and a number of occupants. These are, of course, the St. Ives apartments.

The preceding program does two things: It numbers all the apartments, and then it displays the list of numbers. I know this isn't a lot to be doing, but I just want to show you how subscripting works. The program uses three indexes — one for each array. The program also uses a counter named NumberMaker that is initialized with a value of 100 (the number of the first apartment), and the looping begins (as the William Tell Overture plays in the background).

The outermost loop uses SIndex to step through all the streets. Inside the outer loop is a loop using BIndex to step swiftly through all the buildings in the list. The innermost loop uses AIndex to step through all the apartments and stamp a number on each one. (Music out.) The first set of nested loops sets the numbers. The second loop, using the same set of three nested loops, displays the numbers. If this is too exhilarating for you, take a break.

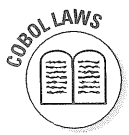
Welcome back. A slight difference exists in how things are addressed in the two loops. The reference to UnitNumber inside the first loop is fully qualified like this:

```
MOVE NumberMaker TO UnitNumber OF Apartment
                    OF Building OF
                    Street(SIndex,BIndex,AIndex)
```

The UnitNumber in the second loop is not fully qualified, and looks like this:

```
                DISPLAY UnitNumber(SIndex,BIndex,AIndex)
```

Either way is okay. The rules for the OF and IN qualifications are the same for arrays as for any other items in a record: if you can, without ambiguity, figure out which one you are referring to, so can COBOL.



Three things are worth noting in the preceding subscripting. First, all three index values are included in a single comma-separated list between one pair of parentheses. Second, the indexes are included at the very end of the name, no matter how many levels of qualification are required. Finally, the order of the indexes is from outer to inner. The leftmost index is the outermost loop — the rightmost index is the innermost.

Doing COBOL indexing the old-fashioned way

The PERFORM statements in the preceding ApartmentComplex example use a form based on ANSI 85 COBOL. Prior to that standard — under ANSI 74 and before — you could not have a PERFORM statement without a paragraph name. You will find older programs that were written with this limitation.

To satisfy the older syntax requirements, the final loop — the one at the bottom of the program — would be written as follows:

```
. . .
    PERFORM ParaA VARYING SIndex FROM 1 BY 1
        UNTIL SIndex > 5.
    STOP RUN.
ParaA.
    PERFORM ParaB VARYING BIndex FROM 1 BY 1
        UNTIL BIndex > 12.
ParaB.
    PERFORM ParaC VARYING AIndex FROM 1 BY 1
        UNTIL AIndex > 4.
ParaC.
    DISPLAY UnitNumber(SIndex,BIndex,AIndex).
```

This version does the same thing as the previous example, except that it requires the presence of some paragraph names to satisfy the fact that the PERFORM verb must execute a paragraph. The two programs do the same thing.

I have more to say about using PERFORM for looping in Chapter 9.

The order of the subscripts inside the parentheses is one of those things that can be hard to remember. You can find yourself sitting at the terminal with your fingers suspended in the air just above the keyboard and a big yellow question mark in the air above your head. Just think of it this way: The one on the right is the one that moves the fastest when you are looping through the data. It's like the odometer on your Harley.

Setting Initial Values for a Table

The purpose of any table is to hold values. In some cases, these values never change while a program is running — the table just sits there so you can read the values and use them for your own purposes elsewhere.

To be able to read the values from the table, you have to do something to put them in there. These values — the ones that you would like to have magically appear whenever your program starts to run — are called *initial values*.

The good news is that several ways exist to have initial values appear in an OCCURS array. The bad news is that none of them are straightforward and obvious. The good news is that your table can have any kind and any range of initial values you want. The bad news is that you are going to have to figure out how to put the initial values into the table. The good news is that I have included some examples of the way I do it. The bad news is that I have included some examples of the way I do it.

Using a VALUE clause on the OCCURS

The following example shows one way you can set all the members of an array to the same value:

```
01  SixFours.
    02  Fours PIC 9(3) COMP OCCURS 6 TIMES VALUE 4.
```

The preceding code gives you an array of six numeric values, all of which contain 4. You can also use the same approach to initialize a group of character stuff, as in the following code:

```
WORKING-STORAGE SECTION.
01  SixDates.
    05  Deadline OCCURS 6 TIMES VALUE "01012000".
        10  MM PIC 99.
        10  DD PIC 99.
        10  YY PIC 9999.
```

Each member of the Deadline array holds the date January 1, 2000 — a date that will live in infamy. This technique works fine as long as things are simple. If the record gets long, or if it contains data that is not stored as characters, this trick can become cumbersome or useless.

Using REDEFINES and a flat list

Here's a little trick that can be quite convenient when you're setting up small tables of constant values: Create a flattened form of your array. That is, declare a record that holds a bunch of fields that are just like the ones in your array. Each one of these fields has a VALUE clause to initialize it. After you have this record, simply use REDEFINES to make it into an array, like this:

```
01  Days.
    03 EachDayName.
        05  FILLER PIC X(3) VALUE "Sun".
        05  FILLER PIC X(3) VALUE "Mon".
        05  FILLER PIC X(3) VALUE "Tue".
        05  FILLER PIC X(3) VALUE "Wed".
        05  FILLER PIC X(3) VALUE "Thu".
        05  FILLER PIC X(3) VALUE "Fri".
        05  FILLER PIC X(3) VALUE "Sat".
    03  FILLER REDEFINES EachDayName.
        05  DayName PIC X(3) OCCURS 7 TIMES.
```

The preceding example constructs an array of the names of the days of the week so that "Sun" can be addressed as DayName(1), "Mon" as DayName(2), and so on. Notice that each member of the record EachDayName has exactly the same PICTURE as the array itself, and both of them have seven entries.

Here is a slight variation on the REDEFINES theme. The entire array is defined as one long character string, and the array then REDEFINES the character string:

```
01  Days.
    03 EachDayName.
        05  FILLER PIC X(21) VALUE "SunMonTueWedThuFriSat".
    03  FILLER REDEFINES EachDayName.
        05  DayName PIC X(3) OCCURS 7 TIMES.
```

The result is the same whether you use a list of separate names or jam them all into one character string. It's just a matter of personal taste. Making this kind of choice is one of those things that folks refer to as "the creative part of programming"; the single jewel set into your crown of brilliant coding decisions; code that, a hundred years from now, will be gold-embossed and displayed in the programmer's hall of fame. Okay, enough of that — back to work.

Blammit! Clearing out an array

Leftovers are okay for lunch, but not for data. If you are going to be using a record in such a way that some old, leftover data could intermingle with your new data, you need to just clean the whole thing out before you start. Also, don't forget that your program doesn't clean up the data areas when it starts running; if you don't clean up behind COBOL, you are almost guaranteed to have a bunch of random garbage.

On occasion, you have an array that is made up of data that is all one type, and you just need to clear the whole thing out. You can do this in one big MOVE, as in the following example. Here's the array that you want to clear out:

```
01  BigList.
    02  Suspect OCCURS 30 TIMES.
        03  FirstName PIC X(20).
        03  LastName PIC X(20).
```

You can clear this entire array with the following statement:

```
MOVE SPACES TO BigList.
```

Blam! This code clears the whole thing quicker'n a skunk clears a phone booth. Of course, moving spaces works only for character data. You need a slightly different approach if your array is all BINARY or COMP data, as in this example:

```
01  BettingList.
    02  Bet OCCURS 50 TIMES.
        03  Amount PIC 9(6)V9(2) COMP.
        03  Odds PIC 9(3)V9(3) COMP.
```

Here's how you can instantly clear this table:

```
MOVE LOW-VALUES TO BettingList.
```

This statement clears out the entire array quicker'n a rotten egg clears out a Sunday school.

Making one record and then looping and moving

In some cases, your array is made up of mixed data types, or you need to initialize each member of an array to a different value, or you're just one of those people who likes to do things the long way. Look at this example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. InitInit.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  Template.
    03  BadWord PIC X(4).
```

```
    03  Altitude PIC ZZZ9.
    03  MinimumValue PIC 9(10) COMP.
    03  MaximumValue PIC 9(10) COMP.
    03  Excuse PIC X(120).
01  Stuff.
    03  BadExcuse OCCURS 25 TIMES.
        05  BadWord PIC X(4).
        05  Altitude PIC ZZZ9.
        05  MinimumValue PIC 9(10) COMP.
        05  MaximumValue PIC 9(10) COMP.
        05  Excuse PIC X(120).
77  AltitudeCounter PIC 9(4).
77  i USAGE INDEX.
PROCEDURE DIVISION.
Init.
    PERFORM BuildTemplate.
    MOVE 123 TO AltitudeCounter.
    PERFORM VARYING i FROM 1 BY 1 UNTIL i > 25
        MOVE Template TO BadExcuse(i)
        MOVE AltitudeCounter TO Altitude of BadExcuse(i)
        ADD 12 TO AltitudeCounter
    END-PERFORM.
    STOP RUN.
BuildTemplate.
    MOVE "Pooh" TO BadWord of Template.
    MOVE ZEROES TO MinimumValue of Template.
    MOVE ZEROES TO MaximumValue of Template.
    MOVE SPACES TO Excuse of Template.
```

This example contains a `Template` that is used for the sole purpose of setting up initial values. Notice that the `Template` contains exactly the same collection of field definitions as each member of the `BadExcuse` array. The `Init` paragraph performs `BuildTemplate`, which initializes the fields of the `Template`.

An `AltitudeCounter` is set to the value of `Altitude` that goes into the first member of the array. The loop then moves copies of the `Template` into each member of the array. Also in the loop, the `Altitude` value is set to the current value of `AltitudeCounter`, and then the `AltitudeCounter` is bumped up by 12 — this new value is used for the next element in the `BadExcuse` array.

You can make your initialization paragraphs just as fancy as you want. You can read files, ask questions of the user, or if you want, do some really exotic calculations. You can use `INITIALIZE`, and some of the other fancy COBOL data-manipulator verbs, to help construct the template — I describe them in Chapter 11.

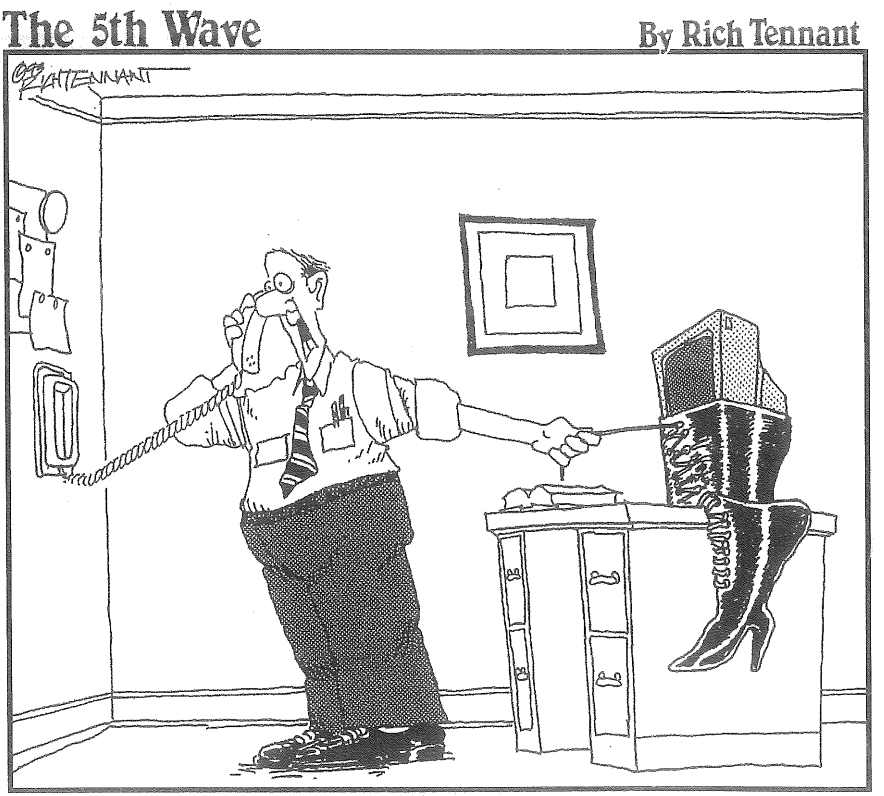
It may seem rather redundant to create a whole record just to initialize an array. I mean, the record is exactly like a member of the array. Well, that's true. In fact, in most cases it's quite convenient to use the first member of the array as the template. The initializer loop would look something like this:

```
PERFORM VARYING i FROM 2 TO HowMany
    MOVE ArrayMember(1) TO ArrayMember(i)
```

This works just fine — initialize the first member of the array and then loop `FROM 2 BY 1 UNTIL` they are all set up.

Part III

The PROCEDURE DIVISION Is Where You Do Things



"OH, I'VE GOT IT BOOTED ALL RIGHT—JUST DON'T ASK ME TO DOUBLE KNOT IT!"

In this part . . .

COBOL executable code is divided into paragraphs. A paragraph is made up of one or more sentences. A sentence always begins with a verb, may or may not contain other verbs, and ends with a period. When you write a COBOL program, you write a series of paragraph labels followed by the sentences contained in each paragraph.

COBOL has verbs that change the flow of execution. For example, some verbs command the program to jump from one paragraph to another. Other verbs allow blocks of code to execute only under certain conditions.

Several verbs specialize in data manipulation. Some of these verbs change the value of a variable, and others copy a value from one location to another.

The chapters in this part show you how to use COBOL verbs to manipulate data. In these chapters, I help you understand the details required to construct the paragraphs, statements, and sentences that do all the work in a COBOL program.

Chapter 8

It's PARAGRAPHs and SECTIONs THROUGH and THRU

In This Chapter

- Understanding COBOL sentence structure
- Exploring COBOL paragraph structure
- Examining COBOL section structure
- Understanding how EXIT and CONTINUE are alike
- Recognizing how STOP and END differ

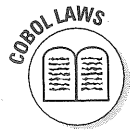
This chapter describes the structure of the PROCEDURE DIVISION, where all the action takes place in a COBOL program. Every executable statement of every COBOL program is in the PROCEDURE DIVISION. You can write the PROCEDURE DIVISION as one long, linear list of statements that are executed, in order, one after the other, from top to bottom.

The truth is that a normal COBOL program is not this linear. The PROCEDURE DIVISION makes some pretty interesting structures available to you. You can break the whole thing into sections, you can break each section into paragraphs, and each paragraph can be made up of a bunch of sentences. A sentence is made up of one or more statements. As you become familiar with this structure, you will find that while all these pieces and parts are the stars of the production, the real genius behind the artistry is the director — a verb named PERFORM.

Understanding COBOL Sentence Structure

The smallest complete thing you can write in COBOL is a *statement.* A statement begins with a COBOL verb, which is immediately followed by all the stuff the verb needs. In every case, the syntax of the stuff following the

verb is enough to determine where the statement ends, so a period is not really necessary. However, if you decide to put a period on the end of a statement, or at the end of a bunch of statements, everything in front of the period becomes a *sentence*.



In a line of COBOL code, area A begins with column 8 and area B begins with column 12. The law states that an entire sentence must be in area B. Some compilers are a little lax with this law, but for most of them if you allow even one little character to appear in one of the four spaces allocated to area A, your compiler won't play. I explain the A and B areas in Chapter 3; Figure 3-1 in that chapter offers a diagram of the areas in a line of COBOL code.

The following example shows three COBOL sentences:

```
MOVE 34 TO Maximum.
PERFORM CALCULATE-AVERAGE.
ADD 15 TO START-VALUE GIVING MID-VALUE.
```

Each sentence begins with a verb; each verb is followed by some stuff that the verb understands; and each sentence ends with a period. The periods are not strictly necessary — each verb knows what it needs and stops without being told by a period. As far as COBOL is concerned, you can omit the periods from the first two sentences, like this:

```
MOVE 34 TO Maximum
PERFORM CALCULATE-AVERAGE
ADD 15 TO START-VALUE GIVING MID-VALUE.
```

By deleting all but the last period, you change what were three, distinct sentences into one single sentence made up of three statements. COBOL can tell where the `MOVE` ends and the `PERFORM` begins, and where the `PERFORM` ends and the `ADD` begins, without a period.

As long as you keep to area B (the area between column 12 and the right margin), you can continue a sentence to the next line and indent things any way you like. You can write the previous example this way:

```
MOVE A TO B PERFORM CALCULATE-AVERAGE ADD 15
TO START-VALUE GIVING MID-VALUE.
```

I don't recommend this approach; I just thought it only fair to show you what's possible. You can even do this:

```
    MOVE A
TO B PERFORM
        CALCULATE-AVERAGE ADD
    15 TO START-VALUE GIVING
MID-VALUE.
```

These bad examples (I'm *very* good at bad examples) demonstrate the flexibility you have in formatting COBOL. You should only write code this bad in the following cases:

- You know who is going to work on the program next and you happen to owe that person a wet willie.
- You never, ever want to be asked to write another program.

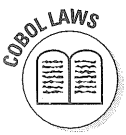
On the other hand, you will find this formatting freedom very handy when you have long sentences.

As the following example demonstrates, sentences and statements are very important whenever COBOL is trying to determine where things begin and end:

```
IF A < B
    NEXT SENTENCE
END-IF
DISPLAY "The next statement"
DISPLAY "Still in the same sentence".
DISPLAY "The next sentence".
```

In this example, whenever A is less than B, the only line printed is `The next sentence`. What happens is that execution of `NEXT SENTENCE` sends COBOL on a period search. It doesn't find one until the end of the next-to-last line, at which point it picks up execution again. For more exciting episodes in the continuing adventure series with `IF` and the period, turn to Chapter 9.

Paragraphs Contain Sentences



A COBOL paragraph has a name followed by a period, and it contains zero or more sentences. The end of one paragraph is determined by the beginning of the next paragraph (or the next section; or the end of the `PROCEDURE DIVISION`; or a really, really bad hair day). A paragraph name must begin in area A, as diagrammed in Figure 3-1 (see Chapter 3). If a paragraph is not empty — that is, if it contains at least one statement — the last statement must end in a period (which changes the statement into a sentence).

Here is an example of a PROCEDURE DIVISION that contains two paragraphs:

```
PROCEDURE DIVISION.
Begin.
    DISPLAY "The begin paragraph"
    DISPLAY "Contains two statements".
Ending.
    DISPLAY "The ending paragraph".
    DISPLAY "Contains three sentences".
    STOP RUN.
```

Paragraphs are great little code organizers. By grouping things into paragraphs, you can give pieces of code special names that have to do with the function they perform. You can have your program leap about from one paragraph to another by using PERFORM or GO TO. If your program includes a statement like, say, PERFORM PRINT-GENERAL-LEDGER, you have at least some idea of what is going to happen.

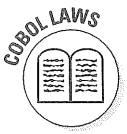
You can even name paragraphs according to their location in the program. In fact, because COBOL programs can get very large, it is not uncommon to assign names to the paragraphs that tell you exactly where they are in the program, as in the following example:

```
1000-INIT.
    PERFORM 3400-TOTE-BARGE.
    PERFORM 6000-LIFT-BALE.
    PERFORM 1500-GET-LITTLE-DRUNK.
    PERFORM 1550-GET-BIG-DRUNK.
    PERFORM 8125-LAND-IN-JAIL.
```

You find code that looks something like this in the initialization paragraph of a large program. The names of all the paragraphs start with a number and have a title that gives you an idea of what they are supposed to do. By including the paragraphs in the program in numeric order, it is a much simpler task to locate the paragraph either in a listing or with a text editor.

Using this sort of naming convention leads to paragraphs of a similar nature having a tendency to reside in the same area. This procedure (or something like it) is fairly common and can be the standard format for all the programs in a programming shop. Normally, this type of standard also requires that some sort of header block appear at the top of the program, listing what each range of numbers means.

Sections Contain Paragraphs



A SECTION of the PROCEDURE DIVISION begins with a section header and continues until the beginning of the next section, the end of the PROCEDURE DIVISION, or until it achieves escape velocity and entirely escapes from planet COBOL. The section header always begins in area A (see Figure 3-1 in Chapter 3). If one or more of your paragraphs are included in a SECTION, you are required to include all of them in a SECTION. I think they get jealous.

You can think of a SECTION as a sort of super paragraph. Not only can it enclose a bunch of paragraphs and be performed as if it were a paragraph, it also has some special powers when dealing with SORT and MERGE. Here's an example showing a program split up into sections:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Sectionalize.
DATA DIVISION.
WORKING-STORAGE SECTION.
PROCEDURE DIVISION.
CONDUCTOR SECTION.
    PERFORM PERCUSSION.
    PERFORM BRASS.
    STOP RUN.
PERCUSSION SECTION.
DRUMMING-BEGINS.
    DISPLAY "Distant drums".
    DISPLAY "They're getting closer".
DRUMMING-ENDS.
    DISPLAY "Timpani rumbles disconcertingly".
    DISPLAY "Hold it colonel, a message is coming through".
BRASS SECTION.
MOUTHPIECE.
    DISPLAY "This takes lots of brass".
    DISPLAY "Do you toot or tutor tooters?".
```

This example shows a COBOL program in three sections. Execution starts, as usual, with the first statement in the PROCEDURE DIVISION. This first statement also happens to be the first statement in the CONDUCTOR SECTION. You may want to take note of the fact that the entire CONDUCTOR SECTION consists of three sentences — not a paragraph to be found — the moral being that it is just fine to begin a section without a paragraph name.

The CONDUCTOR SECTION executes a PERFORM of the other two sections and then stops the program from running any further. All the DISPLAY statements are executed because performing a SECTION is the same as performing each paragraph in the SECTION.

EXIT Is a Lonely Statement

The EXIT statement has a definite attitude problem. If you use an EXIT statement, it must be the first one in a paragraph — it also must be the last one. That's right, it absolutely insists on being the only thing in the whole paragraph. The interesting twist to this requirement is that EXIT doesn't actually *do* anything at all. Nothing. It serves only as a placeholder for an empty paragraph. Here is a direct quote from the COBOL standard: "Such an EXIT statement has no other effect on the compilation or execution of the program."

Would you believe that the EXIT statement turns out to be used quite often? Just to prove that anything is possible, the EXIT statement, even with its poor attitude and lack of skills, has carved out a very special place in the world of COBOL. Many experienced COBOL programmers use it as the body of a termination paragraph on the PERFORM statement — that is, the final paragraph that the PERFORM statement tells the program to execute. Take this example:

```
PROCEDURE DIVISION.
    PERFORM 0100-TOP-NOTCH THROUGH 0299-TOP-NOTCH-EXIT.
    STOP RUN.

0100-TOP-NOTCH.
    DISPLAY "Something".
0150-TOP-NOTCH-AGAIN.
    DISPLAY "Something else".
0200-TOP-NOTCH-YETAGAIN.
    DISPLAY "Yet something else".
0299-TOP-NOTCH-EXIT.
    EXIT.
```

The EXIT statement provides a common end point for a series of paragraphs. You see this format quite often. The PERFORM statement executes THROUGH a list of paragraphs, and the last one in the list contains the EXIT statement.

Using the PERFORM and EXIT statements this way has more than one advantage:

- The PERFORM statement (containing both the beginning and ending paragraph names) tells you, at a glance, the full range of paragraphs being performed.
- Because the terminating paragraph contains no executable statements, you can include new paragraphs and sentences anywhere between the beginning and ending paragraphs without being forced to rename or reorganize.

CONTINUE Does Nothing, and Does It Very Well

The CONTINUE statement is the double-first-cousin to the EXIT statement — it doesn't actually do anything. It can be intermingled with other COBOL statements and will silently have no effect whatsoever on anything around it. I've heard it said that you can't get something for nothing, but, with the CONTINUE statement, you can certainly get nothing for something.

Well, to be fair, the CONTINUE statement does fill a role. You can use it as a placeholder whenever you have a place that needs — well, holding. I sometimes find it handy when I need to change code around inside a complicated IF. For example, while searching through some code, I found this:

```
IF (A > B) AND ((B EQUALS J) OR (M > 0)) THEN
    MOVE 15 TO K
ELSE
    SUBTRACT 4 FROM L
END-IF.
```

Right now, I want to say two things. First, I changed the variable names to simple letters, but otherwise this code came out of a real program. Second, I did not write it. Anyway, if you come across something like this code, and you need to remove the statement that shoves 15 into K, you may want to do it like this:

```
IF (A > B) AND ((B EQUALS J) OR (M > 0)) THEN
    CONTINUE
ELSE
    SUBTRACT 4 FROM L
END-IF.
```

The only other way to make this change would be to rearrange the IF statement. Can you say yuck? I knew you could. Suddenly the CONTINUE statement has completely justified its existence.

Here's a really obscure use of the CONTINUE statement. If you have to deal with one of those officious characters who counts lines of code to determine whether or not you are being productive, and you get the word that you need another thousand lines of code, just say, "Sure. You betcha." Then lumber off down the hall with images of hundreds of CONTINUE statements dancing through your head.

STOP RUN: A Self-Contradiction

The STOP RUN statement sounds like something from Simon says, except you are supposed to do two things at once. Actually, this statement has more dire consequences — you can use it to convince your program to commit suicide and silently cease executing. It's easy to do — just PERFORM or GO TO this paragraph:

```
99999-TERMINATOR.
    STOP RUN.
```

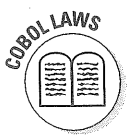
The STOP RUN sentence doesn't have to be in a paragraph by itself — it can be anywhere that you have executable code. However, it is really handy if you put only one of these sentences in your program and use GO TO whenever you need to shut things down. This way, you can easily come back in later and add code to clean up work files, turn off the lights, and say, "Hasta la vista, baby."

END PROGRAM

You can use the END PROGRAM statement at the end of your program. It looks like this:

```
END PROGRAM Fred.
```

Well, it looks like this if you name your program Fred. The ironic thing here is that END PROGRAM doesn't end a program as much as it begins another one. I mean, if you don't put END PROGRAM as the last line of your source code, COBOL just figures it out. However, if you want to include another program in the same source file, you need to end the first one before you can start a new one. Maybe a better keyword for this purpose would be END-OF-OLD-AND-BEGINNING-OF-NEW PROGRAM.



If anything follows the END PROGRAM statement, it must be the IDENTIFICATION DIVISION of another program.

Chapter 9

Verbs That Change the Direction in Which COBOL Runs

In This Chapter

- Branching to a new location with GO TO
- Executing a paragraph with PERFORM
- Creating local and remote loops with PERFORM
- Deciding what to do with an IF
- Examining the anatomy of the conditional expression
- Using EVALUATE as the conditional expression

When you run a COBOL program, execution starts with the first sentence of the first paragraph in the PROCEDURE DIVISION. After the first sentence, execution moves on to the second sentence, and then to the third, and so on, until the program gets to the bottom of the PROCEDURE DIVISION. The program then quits. At least, that's what the program does if you let it. But you also can control the order in which COBOL executes the sentences in your program.

This chapter describes things you can do to make your program loop back on itself. You can tell the program things like, "Go over there and do that and come right back here after you're finished." Or, "Go over there and do that 43 times and then come back." Or, "Skip this part and do that part instead."

This jumping about in the code is called *flow control.* Be careful that you don't let all this power go to your head, or your program will come down with a case of the dreaded Spaghetti Code-itis. This disease can lead to premature code-death, and can earn the programmer who spawned the disease a less-than-complimentary reputation. Throughout this chapter, I tell you more about spaghetti code, and how to avoid it.

Leaping about with Your Basic GO TO

The easiest way to tell COBOL that you want it to go do something else is to just tell it where to go. You do this by using GO TO.

Spaghetti Alert! Since the advent of the block-structured statements (for example, IF and END-IF, or PERFORM and END-PERFORM), folks tend to frown on programmers who actually *use* a GO TO in their programs. I can't think of any other statement that can result in a confusing program more readily than GO TO does. However, every programming language in the world has the equivalent of a GO TO, and it exists for a reason. A time will come when you just gotta GO. For example, you may write a large procedure (several paragraphs long) and find that you need to jump from a place in the middle of the procedure directly to the end. About the only way to jump cleanly to the end of the procedure is by using GO TO.

In the following sections, I discuss the two forms of the GO TO statement.

Plain vanilla GO TO

The simplest form of a GO TO just uses the name of a paragraph as its target — when the GO TO executes, the flow of program execution jumps directly to the named paragraph. The following example uses this form of the GO TO:

```
PROCEDURE DIVISION.
THIS-PARAGRAPH.
    GO TO THAT-PARAGRAPH.
SKIPPED-PARAGRAPH.
    DISPLAY "This will not display".
THAT-PARAGRAPH.
    DISPLAY "This will display".
```

This PROCEDURE DIVISION has three one-line paragraphs. When the program starts to run, it starts with THIS-PARAGRAPH — the one with the GO TO in it. The flow of the program is immediately transported from THIS-PARAGRAPH to THAT-PARAGRAPH. The SKIPPED-PARAGRAPH never runs. Of course, the SKIPPED-PARAGRAPH can be the target of a GO TO somewhere else in the program — that's how spaghetti rumors get started.

A GO TO with a DEPENDING clause

This form of a GO TO has a list of two or more paragraph names followed by a DEPENDING clause that determines which of the paragraphs is to be the target of the GO TO. At first glance, this type of GO TO statement looks like you can use it to go a lot of places all at once. Nope. This statement lets you

choose where to go by using MOVE to place a value into the *identifier*. The identifier is a kind of magic number used by GO TO — the identifier selects where to go. As the following example shows, the GO TO statement lists all the places the program can go:

```
01  BRANCHER PIC 9 VALUE 1.
    . . .
PROCEDURE DIVISION.
THIS-PARAGRAPH.
    GO TO ANOTHER-PARAGRAPH YET-ANOTHER-PARAGRAPH
        DEPENDING ON BRANCHER.
    . . .
ANOTHER-PARAGRAPH.
    MOVE A to B.
YET-ANOTHER-PARAGRAPH.
    MOVE B TO A.
```

Whenever THIS-PARAGRAPH starts running, one of three things happens. If BRANCHER has a value of 1, the program goes to ANOTHER-PARAGRAPH. If BRANCHER has a value of 2, the program goes to YET-ANOTHER-PARAGRAPH. If BRANCHER has any other value, the GO TO does nothing — it just drops through to the next sentence after the GO TO. In other words, the identifier (in this case, BRANCHER) typically takes on a value in the range of 1 to the number of paragraphs listed in the GO TO. For example, if the GO TO lists three different paragraphs, the identifier typically takes a value of 1, 2, or 3.

Taking Action with the PERFORM Verb

The PERFORM verb is the COBOL way of saying, "Do something!" The PERFORM statement is probably the second-most used in COBOL (MOVE being the runaway number one). PERFORM is sort of a GO TO with a "Ya'll come back now!" attached to it. You use PERFORM to do something over there, do something right here, or do something over and over again.

The PERFORM verb is your strongest ally in structuring your program. For example, if you have a paragraph somewhere in your program that initializes all your variables, another that creates work files, a group of paragraphs that do the main processing, and a final paragraph that closes all the files, your main paragraph can look like this:

```
MainParagraph.
    PERFORM INITIALIZE-VARIABLES.
    PERFORM CREATE-WORK-FILES.
    PERFORM PROCESS-ITEMS THROUGH PROCESS-ITEMS-EXIT.
    PERFORM CLOSE-ALL-FILES.
    STOP RUN.
```

This kind of organization can give your program a great deal of clarity. Whenever you come across discussions about structured COBOL, they are referring to the strategic use of PERFORM.

The traditional PERFORM

You supply the PERFORM verb with a paragraph name, it goes off and runs every sentence in the paragraph, and then comes right back, like a dog in a game of fetch. Here's a simple example:

```
PROCEDURE DIVISION.
OVER-HERE.
    DISPLAY "I'm over here."
    PERFORM OVER-THERE.
    DISPLAY "I'm over here again."
    . . .
OVER-THERE.
    DISPLAY "Now I'm over there."
```

This example starts running with the paragraph named OVER-HERE. The PERFORM statement causes the flow of execution to jump to OVER-THERE, run to the bottom of that paragraph, and then jump back to the statement following the PERFORM. The result is that the following three lines are displayed:

```
I'm over here.
Now I'm over there.
I'm over here again.
```

It's just as if the OVER-THERE paragraph were right inside OVER-HERE.

A well-designed COBOL program uses the PERFORM verb frequently. Take a look at the program listing at the end of Chapter 2. That program provides an example of organizing the code elements of the PROCEDURE DIVISION into related groups and using PERFORM to execute each of the groups.

PERFORM allows you to organize paragraphs in a logical way, to isolate special functions and algorithms, to read the code and easily understand the flow of logic, and to gain the respect and admiration of your coworkers. Well, that last part may be pushing it a bit, but well laid-out code will do wonders for you next week. What happens next week? That's when somebody comes in and explains how your program has fouled up the entire chicken-pluckers' database and that you must fix it before you can go home. You can smugly pop open your listing and read your code like it was the quick-reference guide to boiling water. You can then do a quick fix-up of the program and stroll off confidently into the oncoming pay raise.

Here is a straightforward example of performing three paragraphs in a row:

```
PROCEDURE DIVISION.
MAIN-PARAGRAPH.
    PERFORM SHOW-ONE.
    PERFORM SHOW-TWO.
    PERFORM SHOW-THREE.
    . . .
SHOW-ONE.
    DISPLAY "One for the money".
SHOW-TWO.
    DISPLAY "Two for the show".
SHOW-THREE.
    DISPLAY "Three to get ready".
```

The very first thing this program does is run the MAIN-PARAGRAPH, which displays these three lines:

```
One for the money
Two for the show
Three to get ready
```

The three PERFORM verbs in a row perform three paragraphs in a row. You don't have to perform them in a row; you can perform them in any order you want.

The traditional PERFORM THROUGH

If you want to PERFORM more than one paragraph, and you want to PERFORM them in order, you can do so by just naming the first one and the last one and telling the program to PERFORM THROUGH all of them, like this:

```
PROCEDURE DIVISION.
MAIN-PARAGRAPH.
    PERFORM SHOW-ONE THROUGH SHOW-THREE.
    . . .
SHOW-ONE.
    DISPLAY "One for the money".
SHOW-TWO.
    DISPLAY "Two for the show".
SHOW-THREE.
    DISPLAY "Three to get ready".
```

MAIN-PARAGRAPH runs SHOW-ONE and, because the THROUGH clause ends with a paragraph name other than SHOW-ONE, execution continues. Instead of returning from SHOW-ONE, the flow of execution goes right down into

SHOW-TWO and runs the DISPLAY statement there. This still isn't the paragraph specified on the THROUGH clause, so the flow of execution just keeps on going. The DISPLAY statement of SHOW-THREE does its thing. At this point, the paragraph specified in the THROUGH clause of the PERFORM statement has finished running, so the flow of execution returns to the statement immediately after PERFORM (represented by ellipses in the sample listing). COBOL doesn't care how many paragraphs exist between the two paragraphs listed in the THROUGH clause — they all get run.

PERFORMing over and over

You can perform the same paragraph numerous times with a single PERFORM verb. Actually, COBOL gives you more than one way to do this. I show you how with several examples, progressing from the simple through the complicated and to the completely bizarre, committee-designed ways of doing it.

Here is a nice, simple way to PERFORM a paragraph six times:

```
    PERFORM SOME-PARAGRAPH 6 TIMES.
```

That's all you need to do. SOME-PARAGRAPH is executed by the PERFORM verb exactly six times, one right after the other.

Of course, you will encounter cases in which you need to calculate the number of times to perform a paragraph. In such cases, the following approach is appropriate:

```
01  P-COUNTER PIC 99.
    . . .
    MOVE 12 TO P-COUNTER.
    PERFORM SOME-PARAGRAPH P-COUNTER TIMES.
```

The preceding example performs SOME-PARAGRAPH 12 times. Of course, you can stick any old number in P-COUNTER and have SOME-PARAGRAPH performed that number of times.

Here's another way to control how many times your program performs a paragraph: You can run a counter from some value to some other value and have the paragraph performed once for each of the counts. The following example demonstrates this technique:

```
01  ITER PIC 99.
    . . .
    PERFORM SOME-PARAGRAPH VARYING ITER FROM 20 BY 2
        UNTIL ITER IS GREATER THAN 33.
```

The first thing the PERFORM verb does is set ITER to 20. It then compares ITER to 33. If ITER is less than 33, SOME-PARAGRAPH is performed. After SOME-PARAGRAPH runs, the PERFORM verb adds 2 to ITER. Once again, if the value of ITER is less than 33, SOME-PARAGRAPH is executed. This process continues until ITER is greater than 33.

You can do the same thing in performing several paragraphs, like this:

```
01  ITER PIC 99.
    . . .
    PERFORM SOME-PARAGRAPH THROUGH ANOTHER-PARAGRAPH
        VARYING ITER FROM 20 BY 2
        UNTIL ITER IS GREATER THAN 33.
```

You can also specify when COBOL should test the value of your counter. COBOL normally checks the value of the counter before performing the paragraph. In the preceding example, the program makes sure that ITER is less than 33 and then it performs the paragraph.

By using the TEST LAST clause, you can tell COBOL to make the test after it runs the paragraph. This technique guarantees that your program performs the paragraph at least once. The following example shows how you use the TEST LAST clause:

```
01  ITER PIC 99.
    . . .
    PERFORM SOME-PARAGRAPH THROUGH ANOTHER-PARAGRAPH
        WITH TEST LAST
        VARYING ITER FROM 20 BY 2
        UNTIL ITER IS GREATER THAN 33.
```

The TEST LAST clause tells COBOL to perform the paragraph first and ask questions later — the test of whether ITER has gotten larger than 33 is not made until *after* the paragraph has been performed. If you don't specify TEST LAST, the PERFORM statement assumes TEST BEFORE. You can include TEST BEFORE if you wish, but COBOL just mumbles, "I knew that."

If you have any experience in working with the Year 2000 problem, you may notice that some examples in this book use the dreaded PIC 99 for the declaration of the VARYING counter on the PERFORM statements. A declaration of PIC 99 can only hold numbers from 0 through 99. This is another one of those hidden places where storing a year as a two-digit number can bite you. Even though the two-digit field is just a temporary variable set up to do the looping, if the value is supposed to be a year, it will simply be wrong after the year 1999. Moral: Make sure the PICTURE of a counter in a PERFORM VARYING loop is large enough to handle all possible values.

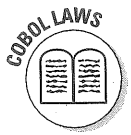
PERFORMing nothing

If you have been discovering the magical powers of PERFORM by reading the preceding sections of this chapter, the following information may come as a bit of a surprise: It is possible to execute a PERFORM statement without specifying the name of a paragraph to be performed. Actually, once you see how this works, it makes perfect sense, and is very handy indeed. It is the only way of executing a loop in line — that is, executing a loop inside a paragraph.

Here is an example of a paragraph with an embedded loop:

```
TWO-MOVES-AND-A-LOOP.
    MOVE A TO B.
    PERFORM VARYING I FROM 1 BY 1
            UNTIL I IS EQUAL TO 5
        DISPLAY "This will display five times"
        DISPLAY "And so will this"
    END-PERFORM.
    MOVE P TO Q.
```

The paragraph name missing from the PERFORM statement tells the COBOL compiler that this is going to be a loop right here in this paragraph. This paragraph moves A to B, displays each of the two lines five times, and then moves P to Q. This little looping trick can really be handy. Without it, you can wind up creating dozens of little paragraphs all around the program just so you can perform them in loops. Spaghetti.



Notice the presence of the END-PERFORM in the preceding example. END-PERFORM is required for the in-line form of the PERFORM. A period won't work here.

The PERFORM and the GO TO

Take care when combining a PERFORM and a GO TO statement — this is a potentially explosive mixture. The PERFORM verb is a round-trip ticket. The PERFORM statement causes the execution of a paragraph elsewhere in the program, and then execution pops right back to the statement immediately following the PERFORM statement. It is possible to use a GO TO in such a way that the PERFORM becomes completely befuddled and never returns. This example shows how it can happen:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. GoWrong.
DATA DIVISION.
WORKING-STORAGE SECTION.
PROCEDURE DIVISION.
Begin.
    PERFORM LowRoad.
    DISPLAY "Back home again".
    STOP RUN.
LowRoad.
    DISPLAY "LowRoad".
    GO TO HighRoad.
MiddleRoad.
    DISPLAY "This is skipped".
HighRoad.
    DISPLAY "HighRoad".
```

This program begins with a PERFORM of the LowRoad paragraph, displaying LowRoad and then jumping to the HighRoad paragraph. The GO TO statement in the LowRoad paragraph causes the flow of execution to jump out of the range specified in the PERFORM statement. Consequently, the program flow doesn't return to the statement immediately following the PERFORM, and the string Back home again never gets displayed.

A one-line change fixes the problem. Change the PERFORM line from this:

```
PERFORM LowRoad.
```

to this:

```
PERFORM LowRoad THROUGH HighRoad.
```

This change moves the return point from the end of LowRoad to the end of HighRoad. The PERFORM returns when it gets to the end of HighRoad, no matter how it got there.

Here's an interesting variation. In the following program, one GO TO takes the program flow outside the range of the PERFORM, and another one brings it back again:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. GoOutIn.
DATA DIVISION.
WORKING-STORAGE SECTION.
PROCEDURE DIVISION.
```

(continued)

(continued)

```
Begin.
    PERFORM LowRoad THRU MiddleRoad.
    DISPLAY "Back home again".
    STOP RUN.
LowRoad.
    DISPLAY "LowRoad".
    GO TO HighRoad.
MiddleRoad.
    DISPLAY "MiddleRoad".
HighRoad.
    DISPLAY "HighRoad".
    GO TO MiddleRoad.
```

The flow goes like this:

1. The PERFORM statement, by using THRU, sets up a return position at the bottom of MiddleRoad.
2. The PERFORM statement then jumps to the beginning of LowRoad.
3. LowRoad, using GO TO, leaps out of the range of the PERFORM to HighRoad.
4. HighRoad has its own GO TO that leaps back into the range of the PERFORM.
5. On reaching the end of MiddleRoad, the return position — the one originally set up by the PERFORM — is found and the flow returns to the statement right after the PERFORM in Begin.

Whew. The output looks like this:

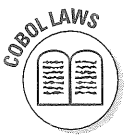
```
LowRoad
HighRoad
MiddleRoad
Back home again
```

This example shows how spaghetti code is born. This simple program is difficult enough to follow; a larger program can become impossible to fathom. In the preceding example, and in almost any other program, replacing the GO TO statements with PERFORM statements is straightforward. This change can simplify life, clarify code, and clear up your complexion.

Creating Old-Fashioned Spaghetti with ALTER

Read my lips: The ALTER statement can produce industrial-strength, canonized, four-star, killer-quality spaghetti code. By using an ALTER statement, you can cause a statement that appears to PERFORM one paragraph to actually PERFORM another.

One of the main purposes of writing a program in a higher-level language such as COBOL is that you can come back and read it to figure out what it does. The use of an ALTER statement obscures this information — it deliberately changes things in such a way to make the program look like it is doing one thing when, in fact, it is doing something else. Only a geek could love this sort of facility. The ALTER verb — which has been declared obsolete, by the way — has been in COBOL for a long time. Some primitive geek designed it.



The paragraph to be altered must contain only one sentence, and that sentence must begin with the verb GO TO. The GO TO statement cannot contain a DEPENDING phrase.

Here's an example of a program that uses ALTER:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. AlterBoy.
DATA DIVISION.
WORKING-STORAGE SECTION.
PROCEDURE DIVISION.
Begin.
    PERFORM Coronation THROUGH EndOfReign.
    ALTER Coronation TO Henry.
    PERFORM Coronation THROUGH EndOfReign.
    STOP RUN.
Coronation.
    GO TO Fred.
Fred.
    DISPLAY "I am Frederick the First I am".
    GO TO EndOfReign.
Henry.
    DISPLAY "I am Henry the Eighth I am".
    GO TO EndOfReign.
EndOfReign.
    EXIT.
```

The paragraph being altered is CORONATION. Strictly speaking, it isn't the paragraph that is altered — it is the GO TO inside the paragraph that is altered. The original target of the GO TO is a paragraph named Fred. After the ALTER statement executes, it is just as if the statement had been written GO TO Henry instead of GO TO Fred.

The ALTER statement has been declared obsolete. And for good reason. It is generally considered to be a poor programming practice. Some old COBOL code uses it, so you may encounter it from time to time. If you need to modify the ALTER code in any way, you may want to consider an ALTER-ectomy to have it removed.

Making Simple Decisions with an IF Statement

The IF verb runs a test and then takes action according to the results of the test. Compared to the tests you and I took in school, IF is a very simple one — it has only one true-false question. The test itself is in the form of a *conditional expression* — for example, a comparison between the values of two numbers or determining which one of two names comes first alphabetically. An IF statement can do one of two things — one for true and another for false. An IF statement is shaped like this:

```
IF condition THEN
    do this if the condition is true
ELSE
    do this if the condition is false
```

The condition usually has to do with the comparison of one thing to another. (I tell you a lot more about conditions in the section, "Writing Conditional Expressions," later in this chapter.)

Take a look at a simple example of an IF statement. In the following code, if the value of LIMIT-COUNT is 21 or more, the result of the test is true and the two MOVE statements set the LIMIT-COUNT to zero and set the LIMIT-COUNT-EXCEEDED to Yes:

```
IF LIMIT-COUNT IS GREATER THAN 20 THEN
    MOVE ZERO TO LIMIT-COUNT
    MOVE "Yes" TO LIMIT-COUNT-EXCEEDED.
```

Beware the tyrannical power of the period. In the previous example, all statements after the IF and before the period are considered one sentence. That is, all statements from the IF to the period run whenever the condition is true. Both MOVE statements are skipped whenever the condition is false. It is a fact of life that the smallest and most difficult-to-see character in the COBOL character set is the most syntactically powerful.

A better way exists to write the code for the preceding IF statement. Using the block-structured form of IF takes the scary period out of the picture and allows you to see exactly where the thing ends. The block-structured form of the same IF looks like this:

```
IF LIMIT-COUNT IS GREATER THAN 20 THEN
    MOVE ZERO TO LIMIT-COUNT
    MOVE "Yes" TO LIMIT-COUNT-EXCEEDED
END-IF
```

Don't be too smug, though — a period stuck anywhere between the IF and the END-IF still gums up the work!

Here's a debugging tip: Anytime you are fooling around with some code that has more than one line following an IF, and the code just doesn't run right, take a very close look to see if you can find a period that got stuck in there by accident. If it seems that some of the statements inside the IF are not being executed, or if they all seem to ignore the IF statement and execute every time, you can have an unwanted period on the end of one or more statements inside the IF block of code. I have a special name for a period that I put in the code this way, but I can't tell you what it is or this book loses its PG rating.

Take a look at the second part of the IF statement. You can add an optional ELSE clause to an IF statement, and the statements associated with the ELSE only execute if the result of the test is false. This clause is handy when you want your code to do only one of two different things. Here's a simple example of using IF with an ELSE to do just that:

```
IF LIMIT-COUNT IS GREATER THAN 20 THEN
    MOVE ZERO TO LIMIT-COUNT
    MOVE "Yes" TO LIMIT-COUNT-EXCEEDED
ELSE
    MOVE "No" TO LIMIT-COUNT-EXCEEDED.
```

After this block of code has finished executing, the LIMIT-COUNT-EXCEEDED is guaranteed to be either Yes or No. Notice again the careful placement of the period following the entire IF/ELSE block. A period anywhere in there

would wreak havoc with the logic flow. Again, you can do it this way if you want to, but I believe you will find that life is much more pleasant if you write things that look more like the following example:

```
IF LIMIT-COUNT IS GREATER THAN 20 THEN
    MOVE ZERO TO LIMIT-COUNT
    MOVE "Yes" TO LIMIT-COUNT-EXCEEDED
ELSE
    MOVE "No" TO LIMIT-COUNT-EXCEEDED
END-IF
```

That just feels better. This version has an `END-IF` instead of a period to terminate the `IF/ELSE` statement. If you decide to add a new statement following the `ELSE`, you will not have to be careful about how you juggle that period around. The `END-IF` version is less error-prone and it is a lot easier to read.

By the way, this `END-IF` sort of stuff is called *block-structured* (or just *structured*) COBOL. Not only does it make the code less buggy, but if you are overheard using words like "structured COBOL," you can make your boss happy. The boss doesn't necessarily know what *structured* actually means, but it came up in a seminar once and it is known to be something good.

Decisions within Decisions: Nesting IF Statements

You can nest `IF` statements inside other `IF` statements. You can do this without using an `END-IF`, but it's so difficult to do, and so error-prone, that almost no reason exists in the world to ever try to do it. You can try it if you want to — just don't put a period anywhere between the first `IF` and the very last statement of the entire nested block of `IF` statements. By the time you get your `IF` statements all nested and properly period-matched and balanced with a corresponding `ELSE` statement, you probably will have completely lost track of your line of scrimmage.

To avoid these problems when you have one or more `IF` statements nested inside of other `IF` statements, use `END-IF`, as in the following example:

```
IF LIMIT-COUNT IS GREATER THAN 20 THEN
    MOVE ZERO TO LIMIT-COUNT
    MOVE "Yes" TO LIMIT-COUNT-EXCEEDED
    IF DRIBBLE-COUNT IS GREATER THAN 1 THEN
        MOVE "Yes" TO DOUBLE-DRIBBLE
    END-IF
END-IF
```

Of course, the nesting of ELSE statements fits right into this pattern, as the following example shows:

```
IF LIMIT-COUNT IS GREATER THAN 20 THEN
    MOVE ZERO TO LIMIT-COUNT
    MOVE "Yes" TO LIMIT-COUNT-EXCEEDED
    IF DRIBBLE-COUNT IS GREATER THAN 1 THEN
        MOVE "Yes" TO DOUBLE-DRIBBLE
    ELSE
        MOVE "No" TO DOUBLE-DRIBBLE
    END-IF.
ELSE
    MOVE "No" TO LIMIT-COUNT-EXCEEDED
END-IF
```

Writing Conditional Expressions

In this section, I deal only with the truth. A conditional statement is a question that the programmer poses to COBOL — COBOL answers either yes or no. Every time. The most common form of a conditional statement is the conditional expression (which I describe in this section), but conditional statements can appear in other places in COBOL.

Here's a list of places where you can use a conditional statement:

- The argument to an IF or EVALUATE, as I describe in this chapter
- The RETURN statement of SORT, which I discuss in Chapter 16
- The AT END or INVALID KEY phrase of a READ verb, as I describe in Part IV of this book
- The INVALID KEY or END-OF-PAGE phrase of a WRITE verb (see Part IV)
- The INVALID KEY phrase of a DELETE, REWRITE, or START verb (see Part IV)
- The ON SIZE ERROR phrase of an ADD, SUBTRACT, MULTIPLY, DIVIDE, or COMPUTE verb, as I describe in Chapter 11
- The ON OVERFLOW phrase of a STRING or UNSTRING verb, as I discuss in Chapter 12

In the real world, some questions are harder than others. You can ask simple questions like, "What color is your other sock?" and more complicated questions like, "What is your mother's father's mother's maiden name?" The same is true for the world of COBOL. You can ask simple questions like,

"Is five greater than four?" and harder questions like, "Is the maximum less than ten and the minimum greater than eight, or is half of the average more than Thursday's median?" The bad news is that you have to make up the questions — the good news is that COBOL has to figure out the answers.

Making a simple comparison

In the simplest and the most common form of a conditional expression, the statement simply compares two numeric values. The two numeric values can be numeric literals, the name of a numeric type in the DATA DIVISION, or even an algebra-like expression. Comparisons are made between the sizes of the two numbers. For example, to determine whether A is greater than B, just write the following code:

```
IF A IS GREATER THAN B . . .
```

Just like chickens and eggs, it doesn't matter which comes first. By reversing both the type of comparison and the values, the expression works this way:

```
IF B IS LESS THAN A . . .
```

Of course, with COBOL being the verbose language that it is, you can write a comparison in any of several ways. The following examples show all the possible ways that you can write the test to determine whether A is greater than B:

```
A IS GREATER THAN B
A GREATER THAN B
A IS GREATER B
A GREATER B
A IS > B
A > B
```

The result of these expressions is false whenever A is equal to B, as well as whenever A is less than B. You can reverse the result of the expression by using the keyword NOT, as in the following examples:

```
A IS NOT GREATER THAN B
A NOT GREATER THAN B
A IS NOT GREATER B
A NOT GREATER B
A IS NOT > B
A NOT > B
```

You can use the same number of variations when asking whether A is less than B, or if A is not less than B. These two statements are the same:

```
A IS LESS THAN B
A < B
```

And so are these:

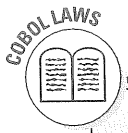
```
A IS NOT LESS THAN B
A NOT < B
```

This same sort of thing applies to all the comparison operators. The possible combinations go on and on. Here is a smattering of examples from the rest of the seemingly endless list of possible combinations:

```
A IS EQUAL TO B
A IS NOT EQUAL TO B
A IS GREATER THAN OR EQUAL TO B
A IS LESS THAN OR EQUAL TO B
A = B
A NOT = B
A >= B
A <= B
```

By having the syntax defined this way, you are free to form your conditional expressions just about any way you like. You may say that COBOL provides a certain freedom of expression.

A NOT isn't available for the combined types of comparison — such as the compound <= and >= operators. Why not NOT? A reason exists, but it's really weird and boring, so if you don't care very much, I suggest you just skip the rest of this paragraph. As you know, putting a NOT in the front of a conditional expression switches its value from true to false, or from false to true. But with an OR operator in the middle of a conditional expression, you no longer have such a simple statement. A NOT brings up questions about whether or not the NOT should just switch the stuff on the left of the OR, or the stuff on both sides of the OR, or should not the NOT apply to the results of the OR. As I discuss a bit later in this chapter (in the section titled "NOT is okay, but NOT NOT is not"), COBOL does offer a way to NOT your OR operators. See, I told you it was boring.



A quick visit to the Land of NOT

The laws of NOT have been relaxed a bit. The COBOL standard now states that you can put a NOT in front of any conditional expression. In the preceding examples, notice that the NOT always immediately precedes the comparison verb (LESS, EQUAL, and so on). This approach is fine, but it is also just as effective to put the NOT out front, like this:

```
IF NOT A > B THEN . . .
IF NOT M IS LESS THAN OR EQUAL
   TO N THEN . . .
```

This approach has a bit of class, don't you think? I mean, if you write your COBOL this way and read it out loud, it sort of sounds like Shakespeare. By adding a couple of new verbs, you can get really classy with your code. Imagine something like LEST A OUT-SHINE B WHENCE . . . .

Comparing nonnumerics

The same bunch of conditional expressions that you use for numerics can be used for comparing nonnumerics. A *nonnumeric* is simply a string of characters one after the other. Two nonnumeric items are compared character-by-character. Although some comparisons make sense, the results of some of the comparisons may surprise you.

Letters closer to the beginning of the alphabet are considered to be *less* than those later in the alphabet. Also, the character 0 is considered to be less than 1, 1 is less than 2, and so on. So far, so good. But, after that, some surprises are in store.

You see, deep down inside its little digital heart, a computer doesn't know anything about letters or any other characters — it can only store numbers. What we humans do is assign each letter and punctuation character its own number and ask the computer to store those for us. This little trick works just fine, but when the computer is asked to compare stuff, it obliges by comparing the numbers that were given to it. This works out just fine — usually.

The actual ordering depends on the encoding of the characters — the internal numeric value that your computer uses for each character. Almost all computers use either ASCII or EBCDIC, the former being the more common. ASCII and EBCDIC make slightly different, arbitrary decisions regarding the ordering of nonnumeric characters, but both coding schemes present the same problem for comparing such characters.

This process of ordering things by the numeric value of the letters is called *lexicographic order*. You may want to take a minute and learn that term. I'm not saying that it will be of any real use in your work, but it can come in handy when you're dealing with a computer nerd. If you find yourself in a situation where a nerd is looking right at you and saying something like, "Virtual buffer interruption spooling inversions," all you have to do is wait for a pause and ask, "Lexicographic order?" Suddenly, you're brilliant. Not only that, the nerd won't have an answer. The phrase *lexicographic order* seems to apply to just about everything.

Anyway, in lexicographic order, the digits all group together, the uppercase letters all group together, and the lowercase letters all group together. Here is an example of an ASCII lexicographic sorting order:

```
10
7
73
8
88
89
9
90
91
Apple
Mango
Orange
apple
banana
```

In the preceding example of lexicographic order, the numbers all come before the letters. Because the comparisons actually involve the character values of the numbers — and not the numeric values — `73` comes before `8`, and `89` comes before `9`. All uppercase letters come before the lowercase letters, so `M` and `O` come before `a` and `b`. If the list included any punctuation characters, they would be sorted first, last, or in the middle, depending on which character it is and whether you are using ASCII or EBCDIC. If you think this may become an issue for you, COBOL comes to the rescue with the alternate alphabets used by `SORT`, as I discuss in Chapter 16.

This discussion about making comparisons strikes very near the heart of the millennium problem: Sorting years. If a year is declared as a `PIC 99`, the year 99 always comes after the year 00. Doing some kind of sort-by-date operation can cause the newest data records to show up as being the oldest. This ordering can do some funny stuff with employee seniority and delinquent account collections.

Determining the class of a field

A field can be classified according to the kind of data it contains. COBOL has a few built-in classes, and you can define classes of your own. Here is an example of a program that asks about the classes of data and gets the answers back:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. FieldContents.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    CLASS Vowel IS 'A' 'E' 'I' 'O' 'U' 'a' 'e' 'i' 'o' 'u'.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  FredField PIC X(4).
PROCEDURE DIVISION.
Begin.
    MOVE "2345" TO FredField.
    PERFORM Identify.
    MOVE "Pq s" TO FredField.
    PERFORM Identify.
    MOVE "PQ S" TO FredField.
    PERFORM Identify.
    MOVE "pq s" TO FredField.
    PERFORM Identify.
    MOVE "aIUo" TO FredField.
    PERFORM Identify.
    STOP RUN.
Identify.
    DISPLAY FredField.
    IF FredField IS NUMERIC
        DISPLAY "      NUMERIC".
    IF FredField IS ALPHABETIC
        DISPLAY "      ALPHABETIC".
    IF FredField IS ALPHABETIC-LOWER
        DISPLAY "      ALPHABETIC-LOWER".
    IF FredField IS ALPHABETIC-UPPER
        DISPLAY "      ALPHABETIC-UPPER".
    IF FredField IS Vowel
        DISPLAY "      Vowel".
```

The four-character field `FredField` is stuffed with different combinations of letters and digits, and each one is tested to determine what it contains. The output looks like this:

```
2345
       NUMERIC
Pq s
       ALPHABETIC
PQ S
       ALPHABETIC
       ALPHABETIC-UPPER
pq s
       ALPHABETIC
       ALPHABETIC-LOWER
aIUo
       ALPHABETIC
       Vowel
```

The IF IS statements examine the field to see if every character is a member of the set of named characters. As you can see, one field can fit the requirements of more than one type. COBOL has four different kinds of predefined character sets:

- A NUMERIC string of characters contains only digits. It cannot contain spaces.
- An ALPHABETIC string of characters contains any combination of upper- and lowercase characters, as well as spaces.
- An ALPHABETIC-UPPER string of characters contains any combination of uppercase letters and spaces.
- An ALPHABETIC-LOWER string of characters contains any combination of lowercase letters and spaces.

In the preceding example program, another character set is defined in the SPECIAL-NAMES paragraphs as a CLASS. In the example, the Vowels class contains all the upper- and lowercase vowels. You can build a CLASS containing as many characters as you wish, including all the punctuation characters and any other characters available to the version of COBOL that you use. For convenience, you can specify a range of characters on the CLASS statement — for example:

```
CLASS FrontHalf 'A' THROUGH 'M'.
```

This class includes all the uppercase letters in the first half of the alphabet.

This field classification capability has all kinds of possibilities. It is most popular among programs that like to test the validity of data. The user was supposed to enter numbers here; did he just put in his mother's maiden name? Or, the user was supposed to enter the name of the county, so it had better be all letters. A phone number should be all digits with maybe a hyphen and some parentheses.

Naming your own conditions

The 88-level trick turns out to be one of the handiest things in COBOL. You can tell that it works well because it is used so much. Almost every program of any size has some 88-level stuff in it. Here's a quick look at how this trick works:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Conditions.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  LimitMark PIC X.
    88  AtTop VALUE 'T'.
    88  AtBottom VALUE 'B'.
    88  Cruisin VALUE 'C'.
PROCEDURE DIVISION.
Begin.
    MOVE 'T' TO LimitMark.
    PERFORM ShowLimit.
    MOVE 'B' TO LimitMark.
    PERFORM ShowLimit.
    MOVE 'C' TO LimitMark.
    PERFORM ShowLimit.
    MOVE 'J' TO LimitMark.
    PERFORM ShowLimit.
    STOP RUN.
ShowLimit.
    DISPLAY LimitMark WITH NO ADVANCING.
    IF AtTop
        DISPLAY " is the top"
    ELSE IF AtBottom
        DISPLAY " is the bottom"
    ELSE IF Cruisin
        DISPLAY " is the cruisin"
    ELSE
        DISPLAY " is an unknown condition".
```

In this example, a field with a `PICTURE` clause has some 88-level names defined with it. The 88-level names tag the possible contents of the field, and the field's contents can be tested by simply testing for the 88-level name. The name of the 88-level item is the entire conditional expression. The output from running this program looks like this:

```
T is the top
B is the bottom
C is the cruisin
J is an unknown condition
```

This technique is often used to specify the status of something. For example, if you are maintaining a billing database, you can have an 88-level field in the record that indicates whether each bill is past due, due, paid, or turned over to some guy named Murray for collection. Chapter 4 offers more information on the care and feeding of the 88-level data.

Checking the sign

Here's a quick way to test whether the result of an arithmetic operation is positive, negative, or zero. The test looks like this:

```
IF A IS ZERO . . .
```

The test doesn't have to involve a simple value. You can use some kind of algebra-looking thing, as in this example:

```
IF (A + B - C) / 22 IS POSITIVE . . .
```

COBOL gives you a few different ways to ask the simple sign question. Here is a list of them using a field named A:

```
A IS POSITIVE
A IS NEGATIVE
A IS ZERO
A IS NOT POSITIVE
A IS NOT NEGATIVE
A IS NOT ZERO
```

This positive and negative thing sounds like part of COBOL from the '60s, when girls would always ask the question, "What's your sign?" I never did figure out what was going on — I always answered, "Yield."

Combining conditions with AND and OR

You can combine two conditions into one by gluing them together with AND and OR operators. For example:

```
IF A < B OR C > D THEN . . .
```

This pair of conditional expressions with an OR between them creates a new, and larger, conditional expression. The new conditional expression results in a true answer if either A is less than B or C is greater than D.

The following example shows the other basic form for combining conditionals:

```
IF A < B AND C > D THEN . . .
```

This pair of conditional expressions is glued together with an AND operator to create a larger conditional expression. This one is true only if A is less than B, and C is also greater than D.

Whenever you combine conditional expressions this way, you get a new conditional expression, which you can also combine into a new conditional expression as in the following example:

```
IF A < B AND C > D AND E = F THEN . . .
```

With this statement, three different things must be true for the whole statement to be true. If any one of the simple conditionals is false, the whole thing is false. The OR works about the same way:

```
IF A < B OR C > D OR E = F THEN . . .
```

This entire statement is true if any one of the simple conditionals is true. For this statement to be false, all three conditionals must be false.

Reading from left to right

If a conditional contains only OR operators, or it contains only AND operators, life is simple. You can read the statement any way you want to and you can see exactly what is going to happen. However, when you have a mixture of OR and AND operators, COBOL follows some rules as it evaluates the statement. Here's an example:

```
IF A = B AND C < D OR E > F AND G >= H THEN . . .
```

Right off, two things are obvious. First, the operation of this statement is not obvious; and second, the programmer who wrote this code has a permanently wrinkled forehead. If you know the trick, however, this statement is not that hard to read. It's simple: read it left to right.

To see how COBOL reads this statement, set A and C to 1, set B, D, E, F, and G to 2, and H to 0. The following example shows a diagram of the step-by-step process that COBOL follows to evaluate the statement.

```
A = B AND C < D OR E > F OR G >= H
false AND C < D OR E > F OR G >= H
false AND true  OR E > F OR G >= H
      false     OR E > F OR G >= H
      false     OR false OR G >= H
                false    OR G >= H
                false     OR true
                          true
```

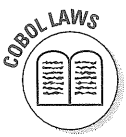
Reading in any direction you want

COBOL does offer a way to overcome this left-to-right order of doing things. By inserting parentheses, you can tell COBOL that you want to do things in a different order. The following example shows the preceding code, rewritten with some parentheses:

```
IF (A = B AND C < D) OR (E > F AND G >= H) THEN . . .
```

This code still has some complexity, but it is easier to read than the original version. More important, though, is the effect that the parentheses have on the order of evaluation. Using the same values as before, the expression evaluates this way:

```
(A = B) AND (C < D OR E > F OR G >= H)
 false  AND (C < D OR E > F OR G >= H)
 false  AND (true  OR E > F OR G >= H)
 false  AND (true  OR false OR G >= H)
 false  AND (      true      OR G >= H)
 false  AND (      true      OR false )
 false  AND (                true      )
       false
```



Logical AND and OR operators are evaluated left to right. Using parentheses causes everything inside a pair of parentheses to be considered as a single unit.

If you are one of those people who has the uncanny ability to take something simple and make something difficult out of it, you have just found your tool kit. With proper care and pruning, complex Boolean expressions (that's geek talk for the AND and OR operations) can be kept quite simple. However, this is a good place to issue a spaghetti alert. An otherwise well-designed and completely readable COBOL program can be completely bollixed up with convoluted AND and OR operators going on for a few lines. If you must write complex conditional expressions, be sure you stay well out of rifle range of anyone who has to work on your code later. I don't have to supply you with an example of this sort of spaghetti — you will immediately recognize it when you see it.

NOT is okay, but NOT NOT is not

The keyword NOT reverses the result of a conditional expression. For example:

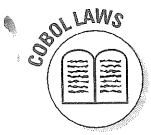
```
IF A > B THEN . . .
IF NOT A > B THEN . . .
```

If A actually is greater than B, the first statement is true and the second one is false. On the other hand, if A is not greater than B, the first statement is false and the second is true.

Using NOT with AND and OR causes the left-to-right rule to come into play:

```
IF NOT A > B OR C = D THEN . . .
```

This example brings up the question of where the NOT applies itself. Does the NOT reverse the comparison of A > B, or does it wait to pounce after the C = D has been evaluated and the OR has reached its decision? May I have the envelope, please?



The reach of a NOT is limited to the conditional expression on its immediate right. If you want to make the NOT apply itself to a larger expression, you need to use parentheses to group them into a single statement to the immediate right of the NOT. Also, the statement to the right of a NOT cannot be another NOT. In other words, NOT NOT is a no no.

Setting the values A, B, C, and D all to 1, here is the evaluation sequence of the previous example:

```
NOT A > B OR C = D
NOT false OR C = D
true      OR C = D
true      OR true
true
```

You can change the order of the evaluation by using parentheses. Enclosing everything following the NOT inside a set of parentheses causes the sequence to go like this:

```
NOT (A > B OR C = D)
NOT (false OR C = D)
NOT (false OR true)
NOT       true
false
```

Combining and compacting conditionals

COBOL has this facility for combining conditionals in such a way that they sound a bit more like English. For example, the following example shows one way to write a couple of conditionals:

```
IF A < B AND A > C THEN . . .
```

You can write the same thing in the following way:

```
IF A < B AND > C THEN . . .
```

Of course, this whole thing can also be written out in elegant, verbose COBOLese, like this:

```
IF A IS LESS THAN B AND GREATER THAN C THEN . . .
```

And you can add other things, resulting in a statement like this:

```
IF A IS LESS THAN B AND GREATER THAN C
    AND NOT EQUAL TO 4 THEN . . .
```

The preceding example sounds enough like English that it can trick the unwary into thinking it has some sort of artificial intelligence behind it. However, it really has a pretty rigorous syntax. There's a trick to getting the syntax right and, once you get it, you can write these conditionals like an old pro.

Here's how the trick works. Notice that the leftmost variable (the variable A in the previous examples) is the one being compared to other stuff throughout the expression. That's the first part of the trick: Have one variable that you want to run through several comparisons.

The second part of the trick is that you have either a conditional (possibly preceded by a NOT) or a field name immediately following each AND and OR. Now you know the whole trick. Any time you have a conditional keyword immediately following an AND or OR, COBOL reaches all the way back to the beginning of the statement and grabs what it finds and sticks it in as the thing on the left side of the conditional. In effect, COBOL rewrites the statement for you.

Here are some examples. You write these statements:

```
A < B AND NOT = 44 OR NOT = 10
A = B OR C OR D
NOT M < J AND K
M NOT < J AND K
NOT M < J AND NOT > K
```

COBOL reads them as if you had written these statements:

```
(A < B) AND (A NOT = 44) OR (A NOT = 10)
(A = B) OR (A = C) OR (A = D)
(NOT M < J) AND (M < K)
(M NOT < J) AND (M NOT < K)
(NOT M < J) AND (M NOT > K)
```

Choosing a Course of Action with EVALUATE

An EVALUATE statement has the power to evaluate an expression and, from the results of the evaluation, select one statement out of many that should be executed. It then executes the statement. The truth is, anything that you can do with an EVALUATE you can also do with an IF/THEN/ELSE sequence of statements. But the EVALUATE statement looks nice, is mostly housebroken, and almost never leaves a mess around the house. With the EVALUATE statement, you can write code that is easier to understand — and easier to modify — than anything you can write using a long string of IF/THEN/ELSE blocks.

Here is an example showing how really neat and tidy EVALUATE can be:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EvaluatingHours.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  HourOfDay PIC 9(2) COMP.
PROCEDURE DIVISION.
Begin.
    PERFORM Morning VARYING HourOfDay FROM 6 BY 1
        UNTIL HourOfDay IS GREATER THAN 12.
    STOP RUN.
Morning.
    DISPLAY HourOfDay "  " NO ADVANCING.
    EVALUATE HourOfDay
        WHEN 8
           DISPLAY "Where are the doughnuts?"
        WHEN 9
           DISPLAY "We're out of coffee."
        WHEN 10
           DISPLAY "It is mid-morning. Break time."
        WHEN 11
           DISPLAY "Isn't it about time for lunch?"
        WHEN 6 THRU 9
           DISPLAY "Too early."
        WHEN OTHER
           DISPLAY "Nothing scheduled for this time."
    END-EVALUATE.
```

This example executes the EVALUATE statement for the HourOfDay values 6 through 12. The program takes specific actions for specific hours of the day.

Here's what the EVALUATE statement does. It evaluates the expression — in this case, the value of the HourOfDay — and then starts looking through the WHEN statements to find a match. As soon as it finds a match, EVALUATE executes the statement associated with that WHEN.

Only one WHEN statement is executed. If more than one WHEN statement matches the criteria, the ones lower down in the list are just out of luck. The first one that matches is used and the others are ignored. The next-to-last WHEN in the example is set to execute for hours 6 through 9, but the hours 8 and 9 are caught higher up in the list — this WHEN statement comes into play only for the hours 6 and 7.

The final entry in the list, the WHEN OTHER entry, is the one that is executed if none of the others are. You don't have to include a WHEN OTHER entry in the list, but if you don't, and if none of the others are executed, nothing at all happens. If you do include a WHEN OTHER, it must be the last entry in the list.

The output from the program looks like this:

```
 6  Too early.
 7  Too early.
 8  Where are the doughnuts?
 9  We're out of coffee.
10  It is mid-morning. Break time.
11  Isn't it about time for lunch?
12  Nothing scheduled for this time.
```

EVALUATE a Conditional

An EVALUATE statement can be made to respond to the true or false result of a conditional expression, as in the following example:

```
EVALUATE A < B
    WHEN TRUE
        DISPLAY "It is true"
    WHEN FALSE
        DISPLAY "It is false"
END-EVALUATE.
```

Doing it this way is about the same as writing an IF/THEN/ELSE sequence, so it really isn't a technological breakthrough. However, you can turn it upside down, as in the following example, to test the result of multiple conditional expressions. You can put a conditional on each of the WHEN statements and find the first one that is either true or false.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ReEvaluatingHours.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  HourOfDay PIC 9(2) COMP.
PROCEDURE DIVISION.
Begin.
    PERFORM Morning VARYING HourOfDay FROM 6 BY 1
        UNTIL HourOfDay IS GREATER THAN 12.
    STOP RUN.
Morning.
    DISPLAY HourOfDay "  " NO ADVANCING.
    EVALUATE TRUE
        WHEN HourOfDay < 8
           DISPLAY "Too early."
        WHEN HourOfDay = 8
           DISPLAY "Where are the doughnuts?"
        WHEN HourOfDay = 9
           DISPLAY "We're out of coffee."
        WHEN HourOfDay = 10
           DISPLAY "It is mid-morning. Break time."
        WHEN HourOfDay = 11
           DISPLAY "Isn't it about time for lunch?"
        WHEN HourOfDay > 11
           DISPLAY "Nothing scheduled for this time."
    END-EVALUATE.
```

The EVALUATE statement starts at the top of the list of WHEN statements and evaluates each of them. The program runs the first WHEN statement that matches the conditional specified on the EVALUATE statement.

Chapter 10

Using MOVE to Put Data in Its Place

In This Chapter

- Moving data from one field to another
- Moving data into a field that is too small to hold it
- Moving data into a field that is too large
- Moving an entire record as a block
- Using MOVE to initialize a record
- Using a CORRESPONDING MOVE to reorganize data

MOVE is the most popular verb in all of COBOL. You use the verb MOVE to copy data from one place to another in the WORKING-STORAGE DIVISION of your program.

It's pretty easy to understand what MOVE does and how it does its job, but it was incorrectly named. It's true that when MOVE finishes working on data, the data is in a new location, so in that sense the data has been moved. But the data also still exists in the original location. MOVE could have been called DUPLICATE or COPY, but those names would be wrong too, because MOVE sometimes modifies the data to make it fit into its new home.

Maybe this verb should have been called REPLICATE or FACSIMILE, because sometimes it makes an exact copy, and other times it changes data in such a way that the data's own mother doesn't recognize it. Maybe it should have been called TRANSMOGRIFY. I don't know what that word means exactly, but I found it in a thesaurus, and it passed the spell checker, so I decided to leave it in here.

This chapter discusses the different ways in which you can MOVE data. You can MOVE data one field at a time, or you can MOVE the whole record. You can even use the MOVE verb to change the format of data and rearrange the order of the fields in a record.

Making a Simple MOVE

An elementary data item — also called an elementary field, or just a field — is one that has a PICTURE clause. It can be a stand-alone 77- or 01-level item, or it can be a member of a record, but it always has a PICTURE clause. A MOVE that puts data into an elementary field from either a literal or another elementary field is called an *elementary* MOVE. (An example of an elementary move would be Holmes giving Watson a hotfoot.)

You are pretty much free to make an elementary MOVE however you want, but you do have to remember a few restrictions having to do with some of the field types I describe in Chapter 5. Here is a list of the things you can't do with a MOVE statement:

- You can't MOVE the constant SPACE to a numeric field.
- You can't MOVE alphanumeric-edited or alphabetic fields to a numeric field.
- You can't MOVE a numeric literal, the constant ZERO, a numeric field, or a numeric-edited field to an alphabetic field.
- You can't MOVE a noninteger numeric value to an alphanumeric or alphanumeric-edited field.
- You can't MOVE a numeric-edited field to a numeric or numeric-edited field.

If you just read two or three items in this list and then skipped on down here, give yourself two extra points. All you really need to do is go ahead and try to make the MOVE in your program — the compiler tells you if you try to break one of the rules.

I think I should make some comments about the preceding list. First, not all COBOL compilers adhere to these restrictions — sometimes you can get away with things in one compiler that are forbidden in another. Second, if you actually understand the preceding limitations in using the MOVE statement, you may be reading the wrong book — you should be reading *Relativity Theory For Dummies*.

The PICTURE clause has so many options that thousands of possible combinations exist when moving data from one field to another. From time to time, you need to make a MOVE that looks dubious. The best way to learn the limitations that your compiler imposes is by following this simple, three-step process:

1. **Type the MOVE statement into the COBOL program you are writing.**

2. **Compile the program.**

   If your compiler doesn't like the MOVE statement, it rewards you by presenting you with an error message. Go back to Step 1 and try something else.

3. **Run the program with some potentially unfriendly data and check the results.**

   If you don't get what you wanted, go back to Step 1 and try something else.

Personally, I find that this process works quite well. It should enable you to find out about the MOVE statement at your own pace. I have used this method for years. I call it the *rats-and-right* method. Each time I get it wrong, I say, "Rats!" When it finally works, I say, "Right!" As time goes by and you become more familiar with COBOL and your compiler, you naturally improve your rats-to-right ratio.

It's time for you to make your first MOVE. Here is a very simple, but MOVEing, program:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SimpleNumberMove.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  A PIC 9(2).
01  B PIC 9(2).
01  C PIC 9(2).
01  D PIC 9(4).
PROCEDURE DIVISION.
Begin.
    MOVE 32 TO A.
    MOVE A TO B.
    MOVE ZERO TO C.
    MOVE A TO D.
    DISPLAY "A=" A " B=" B " C=" C " D=" D.
    STOP RUN.
```

The program SimpleNumberMove declares four numeric fields and then uses a series of MOVE statements to put data in those fields. Running the program generates this output:

```
A=32 B=32 C=00 D=0032
```

The first line of the program moves the literal value of 32 into A. By the way, this statement is also correct if you write it this way:

```
MOVE "32" TO A.
```

This statement is okay because A defaults to USAGE DISPLAY and holds the number as a pair of characters. If you declare A as USAGE COMP (as I describe in Chapter 5), putting quotes around the number may or may not work. I tried this on a couple of compilers and got mixed results. So will you. This example shows why I recommend the rats-and-right method whenever you need to do something that seems like it may not be entirely straightforward.

When you need to MOVE data from one field to another, you need to watch for three basic kinds of fields: nonnumeric, numeric, and edited. The simplest is nonnumeric stuff. It just moves around from place to place without any conversion. Numeric data, on the other hand, can have different USAGE types, and you may run into problems if you try to move data from one USAGE type to another. Also, if you MOVE data between numeric types that are not USAGE DISPLAY and the nonnumeric types, things can get a bit crazy. The third group is the edited data (you know, leading Z characters, commas, and stuff like that). Any time you MOVE something edited to something numeric, you can expect fireworks.

Making a MOVE TO a Bigger Place

If you MOVE data from a smaller numeric field to a larger numeric field, the value settles right into its new home and doesn't change anything (except its appearance). You have all the data you started with, but it has more zeroes or spaces around it when you print it. Here is an example:

```
01  Small PIC 9(4).9(2).
01  Large PIC 9(8).9(3).
      . . .
    MOVE 8192.16 TO Small.
    MOVE Small TO Large.
    DISPLAY "Small=" Small " Large=" Large.
```

The number that you initially MOVE into Small is large enough to fill up this location. When you MOVE the same number into Large, the number has enough room to rattle around a bit. The output looks like this:

```
Small=8192.16 Large=00008192.160
```

As you can see in the preceding example, the number lines itself up around the decimal point. Using MOVE to put character data into a larger field is pretty straightforward. If the receiving field has room left over, MOVE just fills it with blanks on the right. However, you should know a couple of things about JUSTIFIED RIGHT. Take a look at this example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. BiggerMove.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  SmallSize PIC X(5).
01  MediumSize PIC X(10).
01  MediumSizeRight PIC X(10) JUSTIFIED RIGHT.
01  LargeSize PIC X(30).
01  LargeSizeRight PIC X(30) JUSTIFIED RIGHT.
PROCEDURE DIVISION.
Begin.
    MOVE "Stuff" TO SmallSize.
    MOVE SmallSize TO MediumSize.
    MOVE SmallSize TO MediumSizeRight.
    MOVE MediumSizeRight TO LargeSize.
    MOVE MediumSizeRight TO LargeSizeRight.
    DISPLAY "'" SmallSize "' (SmallSize)".
    DISPLAY "'" MediumSize "' (MediumSize)".
    DISPLAY "'" MediumSizeRight "' (MediumSizeRight)".
    DISPLAY "'" LargeSize "' (LargeSize)".
    DISPLAY "'" LargeSizeRight "' (LargeSizeRight)".
    STOP RUN.
```

The preceding example just takes the data from some character fields and moves it around. The output displays single quotes around the fields, to mark where each field begins and ends — in this way, you can see where the spaces have been added. The output looks like the following example:

```
'Stuff' (SmallSize)
'Stuff     ' (MediumSize)
'     Stuff' (MediumSizeRight)
'     Stuff                    ' (LargeSize)
'                         Stuff' (LargeSizeRight)
```

A five-letter word in SmallSize — which is a five-character field — displays with no blanks. A MOVE places the contents of SmallSize into MediumSize. Because MediumSize is larger than SmallSize, the MOVE fills in blanks to the right of the word. Another MOVE takes the word from MediumSize and

puts it into MediumSizeRight, which is the same size but JUSTIFIED RIGHT. In MediumSizeRight, the rightmost nonblank character is shifted as far as it will go to the right, and the blanks are filled in on the left. This is one of those places where MOVE actually fiddles around with the data a little bit, according to the wishes of the receiving field.

Next comes an unjustified MOVE. The move from MediumSizeRight to LargeSize blank-fills on the right. The leading blanks — the ones that were created by justifying to the right in MediumSizeRight — MOVE untouched into LargeSize. However, because LargeSize is bigger than the data it receives, the MOVE must fill with blanks on the right. This leaves your Stuff suspended between two groups of spaces.

The final MOVE takes the data into another JUSTIFIED RIGHT field. Once again, the rightmost nonblank character of the sending field is shoved as far as it can go over to the right of the receiving field.

Making an Unfit MOVE

If you MOVE something into a place where it doesn't fit, MOVE just trims off the data's tail and forces the fit. Here's an example:

```
01  LongName PIC A(32).
01  ShortName PIC A(6).
      . . .
    MOVE "Rumpelstiltskin" TO LongName.
    MOVE LongName TO ShortName.
    DISPLAY LongName.
    DISPLAY ShortName.
```

LongName has plenty of room to hold the name, but the name can't squeeze into the ShortName. It gets its tail trimmed. The output looks like this:

```
Rumpelstiltskin
Rumpel
```

I originally created this example using a PIC A(4) for the ShortName, but the editor looked at the resulting output and became concerned that I was taking an attitude.

Trimming names and stuff is one thing — but numbers are a different matter. The only way to describe the action taken by COBOL whenever you try to cram a large number into a small space is to say that it is vicious. I could say

that this type of MOVE trims numbers, or I could say that it clips them, or I could be really nice and say that it truncates them. All of these terms are kind euphemisms for what it really does — it makes them completely non-fixable and dead wrong. Here is an example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. NumberTrimmer.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  LittleNumber PIC 9(3).9(2).
01  BigNumber PIC 9(5).9(4).
01  FourDigitYear PIC 9(4).
01  TwoDigitYear PIC 9(2).
PROCEDURE DIVISION.
Begin.
    MOVE 12345.6789 TO BigNumber.
    MOVE BigNumber TO LittleNumber.
    DISPLAY BigNumber " became " LittleNumber.
    MOVE LittleNumber TO BigNumber.
    DISPLAY LittleNumber " returned as " BigNumber.
    MOVE 2006 TO FourDigitYear.
    MOVE FourDigitYear TO TwoDigitYear.
    DISPLAY FourDigitYear " became " TwoDigitYear.
    MOVE TwoDigitYear TO FourDigitYear.
    DISPLAY TwoDigitYear " returned as " FourDigitYear.
    STOP RUN.
```

The NumberTrimmer program shows two examples of what can happen when you don't make enough room to hold your values. Here is the output that the program produces:

```
12345.6789 became 345.67
345.67 returned as 00345.6700
2006 became 06
06 returned as 0006
```

The first line of the output gives you some idea of what happens during the trimming ceremonies. The most significant digits, the ones farthest to the left of the decimal point, are unceremoniously and permanently deleted. You can do that if you are sending me bills, but don't mess with my checks! The second line of the output shows that you can't MOVE the number back to the larger place and recover the values — they are gone forever and your numbers could not be more wrong.

The last two lines of the output from the preceding example show another way that a date can be fouled up. This sort of thing can happen to a program that was supposed to have been modified for the year 2000 problem, but a little something was missed during the conversion. A four-digit year (with its front end missing like this) indicates that somewhere in the calculations, data was stored in a field that has never been converted from two to four digits. The program inadvertently trims the date when it has to MOVE the date from one place to another.

Shoving Entire Records Around with MOVE

You normally use a MOVE statement to move the value of one field at a time, but you aren't limited to only one field. You can use a MOVE statement to make block moves of entire data records.

A MOVE has no size limit — if you can get something into your program, MOVE can shove it from one place to another. It is said that if you listen closely in a quiet room with your ear very close to the computer, you can hear the thumping sound made by huge records of data being slapped around by COBOL programs. The next time you are discovered with your head down on your desk with your eyes closed, and somebody nudges you to ask what you are doing, you can just say, "I was listening to the block transfers on the data bus." Nobody will ever bother you again.

The following code declares (and initializes) two records:

```
01  FromRecord.
    02  FirstName PIC X(10) VALUE "Grunion".
    02  LastName PIC X(10) VALUE "Run".
    02  LandSpeed PIC 9(4) VALUE 98.
    02  AwardsWon.
        03  Emmies PIC 9(2) VALUE 3.
        03  Oscars PIC 9(2) VALUE 1.
01  ToRecord.
    02  FirstName PIC X(10).
    02  LastName PIC X(10).
    02  BodyTemperature PIC 9(4).
    02  AwardsWon.
        03  Emmies PIC 9(2).
        03  Oscars PIC 9(2).
```

The two records in the preceding code have identical layouts. Everything about them is the same except for some names and values. One is called FromRecord and the other is called ToRecord. FromRecord has a bunch of initial values. The PIC 9(4) field in FromRecord named LandSpeed is, in ToRecord, named BodyTemperature. Here is a MOVE statement that copies the whole thing:

```
MOVE FromRecord TO ToRecord.
```

After this MOVE, the value of BodyTemperature is 98 because this field just happens to be sitting on the spot where MOVE puts the data. The MOVE statement picks up the data in FromRecord as one big blob and plops it down into ToRecord without doing any conversions whatsoever. The MOVE statement pays no attention whatsoever to the individual fields. As far as MOVE is concerned, it is just as if the two records had been declared like this:

```
01  FromRecord PIC X(28).
01  ToRecord PIC X(28).
```

The following example shows the records again, but this time they are not so nearly alike. The FromRecord is the same as in the preceding example, but I changed the ToRecord organization:

```
01  FromRecord.
    02  FirstName PIC X(10) VALUE "Grunion".
    02  LastName PIC X(10) VALUE "Run".
    02  LandSpeed PIC 9(4) VALUE 98.
    02  AwardsWon.
        03  Emmies PIC 9(2) VALUE 3.
        03  Oscars PIC 9(2) VALUE 1.
01  ToRecord.
    02  FirstName PIC X(10).
    02  LastName PIC X(10).
    02  AwardSpeed PIC 9(8).
```

Both records have the same total size as in the previous example; the new version of ToRecord converts its predecessor's three numeric fields into a single eight-digit numeric field named AwardSpeed. After the MOVE statement has been applied, the data from the sending fields LandSpeed, Emmies, and Oscars are all combined into AwardSpeed, which now contains this data:

```
00980301
```

Creating a working record that breaks up the data in different ways can be quite handy. For example, look at the two records in the following program:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MoveSame.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  DateForm1 PIC 99/99/9999.
01  DateForm2.
    02  MM PIC 99.
    02  FILLER PIC X.
    02  DD PIC 99.
    02  FILLER PIC X.
    02  YYYY PIC 9999.
PROCEDURE DIVISION.
Begin.
    MOVE 02041998 TO DateForm1.
    MOVE DateForm1 TO DateForm2.
    DISPLAY "The day: " DD OF DateForm2.
    STOP RUN.
```

The first MOVE statement puts the date value into DateForm1; this edited field automatically inserts the slashes into the date value. The second MOVE transfers the field as a block into DateForm2, which effectively assigns a name to each of the numeric parts of the date. The DISPLAY statement can then access a single member field, the DD field of the date, resulting in this output:

```
The day: 04
```

You can also do this kind of thing without moving the data from one place to another by using REDEFINES and RENAMES, which I describe in Chapter 4. You may need to use the technique that I describe here, however, because the record containing the field you need to break apart has already been defined and, for some reason, you can't change it. That happens sometimes with legacy code (legacy code is a fancy term that means *old programs that your company still uses*). Of course, you may want to use MOVE just because you want to use MOVE. Okay, fine. That works for me.

Using MOVE as a Record Initializer

Chapter 6 describes the figurative literals and how they can be used to initialize fields. You can use this same bunch of figurative literals in a MOVE statement to initialize an entire record in one fell swoop. Well, sometimes it actually takes a couple of fell swoops.

Things must be initialized. You need to avoid using data from a field before you have put any data into the field. You can't just assume it is going to be spaces or zeroes — it could be #~!5&[or 4@qJ% or anything else. Anything. Uninitialized data has been the source of millions and millions of software bugs over the years. I have done my part in adding to this number. You've heard the expression, "Garbage in, garbage out." It is just as true to say, "Nothing in, garbage out."

Initializing with SPACES and ZEROES

The most common form of record initialization puts spaces in the character data fields and zeroes in the numeric data fields. The MOVE verb can handle this task easily. Take the following record, for example:

```
01  WHO-DAT.
    04  FIRST-NAME PIC X(10).
    04  LAST-NAME PIX X(10).
    04  ADDRESS.
        07  STREET PIC X(10).
        07  CITY PIC X(8).
        07  STATE PIC X(2).
```

You can clear the entire record to spaces in one simple statement, like this:

```
MOVE SPACES TO WHO-DAT.
```

Like most things in COBOL, this code can be phrased in more than one way. COBOL offers an alternate spelling for SPACES, and the word ALL is optional. Here are all the possible combinations:

```
MOVE SPACES TO WHO-DAT.
MOVE ALL SPACES TO WHO-DAT.
MOVE SPACE TO WHO-DAT.
MOVE ALL SPACE TO WHO-DAT.
```

It isn't just SPACES that can be used to fill records. You can do the same thing with ZEROES if you have a record that consists solely of character numeric data, like the following code:

```
01  Quantities.
    03  NumberOfJars PIC 9(4).
    03  PercentOfFullJars PIC 9(2).
    03  NumberOfBrokenJars PIC 9(3).
    03  KidsSpankedForBreakingJars PIC 9(2).
```

You can set all the numeric fields of the entire record to the character 0 with any one of the following statements:

```
MOVE ZERO TO Quantities.
MOVE ZEROS TO Quantities.
MOVE ZEROES TO Quantities.
MOVE ALL ZERO TO Quantities.
MOVE ALL ZEROS TO Quantities.
MOVE ALL ZEROES TO Quantities.
```

All these statements do exactly the same thing — some just use the alternate spelling of ZERO along with the optional keyword ALL.

I mention at the beginning of this section that you may need to use more than one fell swoop to initialize an entire record. Take this example record:

```
01  Driver.
    02  Name PIC X(50).
    02  TypeOfCar PIC X(4).
    02  AverageSpeed PIC 9(3).
    02  MothersMaidenName PIC X(20).
```

If you just slam SPACES into this record, you fill the numeric AverageSpeed with the space character. This may not be what you want. You can handle this field separately by doing this:

```
MOVE SPACES TO Driver.
MOVE ZEROES To AverageSpeed OF Driver.
```

The first MOVE slams SPACES into everything, and the second line overwrites the SPACES with ZEROES for the AverageSpeed. Sneaky, eh?

Although this multiple-MOVE method does work, it can become unwieldy for a large record with lots of fields. For example, if you have something with 30 or 40 character fields and 10 or 12 numeric fields, you wind up writing 10 or 12 MOVE statements — one for each numeric field. As I discuss in Chapter 11, INITIALIZE offers a better way to accomplish the same thing.

You take the HIGH-VALUE, and I'll take the LOW-VALUE

HIGH-VALUES and LOW-VALUES are figurative literals that can be used as special key values in sorting. I describe these and other figurative literals in Chapter 6, and I discuss sorting in Chapter 16.

You can MOVE HIGH-VALUES or LOW-VALUES into an entire record. The ALL keyword is optional when you do this. This type of MOVE isn't as commonly used as MOVE SPACES or MOVE ZEROES, but it can be quite useful in some cases. It can be handy, for example, if you have a sort key and you want to guarantee that the sort key comes either first or last in the sorting order. For example:

```
. . .
10  SortKey.
    15  PrigType PIC X.
    15  MorkType PIC X.
    15  DrumCode PIC 9(3).
```

Here are the ways to make sure that SortKey comes first in the sort:

```
MOVE LOW-VALUE TO SortKey.
MOVE LOW-VALUES TO SortKey.
MOVE ALL LOW-VALUE TO SortKey.
MOVE ALL LOW-VALUES TO SortKey.
```

And here are the ways to make sure that it comes last in the sort:

```
MOVE HIGH-VALUE TO SortKey.
MOVE HIGH-VALUES TO SortKey.
MOVE ALL HIGH-VALUE TO SortKey.
MOVE ALL HIGH-VALUES TO SortKey.
```

Filling records with anything at all

You can fill fields and records with anything you want. You can make up any bunch of characters you would like, put them between a pair of double quotes, and spread them across a record like peanut butter on toast. Here is a sample program that shows you how to do this:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. FillErUp.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  FilledRecord.
    02  Camels PIC X(5).
    02  Humps PIC 99 USAGE COMP.
    02  Hooves PIC 9999V99.
    02  FILLER PIC $ZZ,ZZ9.99.
    02  Oasis PIC A(20).
```

(continued)

(continued)

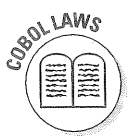
```
PROCEDURE DIVISION.
Begin.
    MOVE ALL "#" TO FilledRecord.
    DISPLAY FilledRecord.
    MOVE ALL "J" TO FilledRecord.
    DISPLAY FilledRecord.
    MOVE ALL "dbqp" TO FilledRecord.
    DISPLAY FilledRecord.
    MOVE ALL "Peanut Butter " TO FilledRecord.
    DISPLAY FilledRecord.
    STOP RUN.
```

Each MOVE statement in the preceding example is followed by the keyword ALL and a literal — this is the sending side of the MOVE statement. The name of a record is the receiving side. The MOVE statement grabs the literal and sort of makes a rubber stamp out of it. It then makes as many duplicates as it can, until it runs out of space in the record. The output from the program looks like this:

```
#############################################
JJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJ
dbqpdbqpdbqpdbqpdbqpdbqpdbqpdbqpdbqpdbqpdbqpd
Peanut Butter Peanut Butter Peanut Butter Pea
```

Notice that the program fills the whole record with characters, no matter what types the individual fields are. This means, of course, if you were to try to read the data from the edited or COMP fields you would get garbage, but, hey, you can't have everything, right? I mean, what good is it to have a powerful language like COBOL if you can't produce a little garbage data from time to time?

Putting incorrectly formatted data into a record may not seem like a good thing to do, but in some cases, it may come in handy. These cases are, fortunately, rather rare and serve a special purpose. For example, you can use a MOVE ALL to determine the record size. (Chapter 21 includes an example program that uses this trick to calculate the record size.) The printing of the characters from the preceding example shows that the record size is 45 characters long, but you can't determine this size by examining the record itself. First, the V character in a PICTURE clause doesn't take up a character space. Also, the COMP field, even though it is a PIC 99, for the compiler used in the example, actually consumes four character spaces.



Whenever the receiving end of a MOVE statement is a record, COBOL treats it just as if it were a nonnumeric type. For example, if the receiving record is a total of 45 characters long, MOVE treats it just as if it were declared PIC X(45).

Making Your MOVE to Lots of Places

Moving the same thing into more than one place is possible with a single MOVE statement. Here is a simple program that does it:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. OneToMany.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  A PIC X(4) VALUE "fred".
77  B PIC X(4).
77  C PIC X(4).
77  D PIC X(4).
77  E PIC X(4).
77  F PIC X(4).
PROCEDURE DIVISION.
Begin.
    MOVE A TO B C D E F.
    DISPLAY A B C D E F.
    STOP RUN.
```

The output of the program looks like this:

```
fredfredfredfredfredfred
```

This technique mostly comes in handy for setting up initial values. You can use this method to set a whole bunch of numeric values to zero, or to set a group of records to blanks, as does the following code:

```
MOVE SPACES TO InputRecord OutputRecord WorkData.
```

Some Sneaky Stuff about MOVE and OCCURS

The process by which you MOVE things into or out of an OCCURS array is straightforward enough. Any MOVE that you can put on data without an OCCURS can also be applied to data with an OCCURS. Here's an example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HaHa.
DATA DIVISION.
```

(continued)

(continued)

```
WORKING-STORAGE SECTION.
01  FILLER.
    04  A PIC X(4) OCCURS 10 TIMES.
    04  i PIC 9(2) COMP.
PROCEDURE DIVISION.
Begin.
    MOVE 4 TO i.
    MOVE "ha" TO A(i).
    MOVE A(i) TO A(6).
    DISPLAY A(i) A(6).
    STOP RUN.
```

This example shows how you can use a MOVE to set the value that the program uses to index an OCCURS array. The example then shows how you can use MOVE to put stuff into an array and get it back out again. This program puts ha into the locations A(4) and A(6) — the program accesses these array elements by using a variable index and a constant index, respectively. The output from the program looks like this:

```
ha  ha
```

So far, so good. But what happens if you diddle around with the index value in the same MOVE statement where you diddle around with the value in the array being indexed? Here's an example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. IndexDiddle.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  FILLER.
    04  A PIC 9(2) COMP OCCURS 10 TIMES.
    04  i PIC 9(2) COMP.
PROCEDURE DIVISION.
Begin.
    MOVE 4 TO i.
    MOVE 8 TO A(i).
    MOVE A(i) TO i A(i).
    DISPLAY i A(4) A(8).
    STOP RUN.
```

The output of the program looks like this:

```
080808
```

The program is pretty straightforward except for this one statement:

```
MOVE A(i) TO i A(i).
```

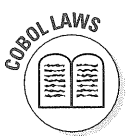
Good ol' COBOL starts this MOVE statement by retrieving the value from A(i), and, because i has been set to 4, the retrieved value is 8. So far, so good. The next thing COBOL does is shove the 8 into the i. Still no problem. The final act is to shove the 8 into A(i) — but does this mean the old value of i or the new value of i? The output of the program shows that COBOL chose the new value of i.

Don't write code like this example. You should have no real need to ever put confusing stuff like this in your program. Some folks do it, however, so I include this example to help you figure out what is going on when you read stuff like this. If you find something like this in a program you are working on, it is considered bad form to actually go and injure the programmer who put it there. You may cause people to talk about you if you go around hurting programmers.

Rewriting the statement the way that COBOL actually does things results in this code:

```
MOVE A(i) TO temporary.
MOVE temporary TO i.
MOVE temporary TO A(i).
```

This version is much easier to follow than the previous example. The first MOVE statement stores the value from A(i) into temporary. The second MOVE changes the index. The third MOVE stores the value in its new location.



When you MOVE data from a subscripted field, COBOL evaluates the subscripting and the parts of the statement to the left of the TO only once. The results of the evaluation are then stored temporarily and applied to all the receiving identifiers to the right of the TO in left-to-right order.

Reformatting Data with MOVE CORRESPONDING

COBOL considers things that are in two separate records, but have the same name, as corresponding to one another. If a record contains fields that have

the same names as fields in another record, you can move data between the corresponding fields in the two records by using (you guessed it) MOVE CORRESPONDING. Here is an example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. IndexDiddle.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  FromBunch.
    05  DEPT PIC X(2) VALUE "DE".
    05  DIV PIC X(2) VALUE "DI".
    05  IRREGARDLESS PIC X(3) VALUE "SAG".
    05  ENDING-DATE VALUE "01021999".
        10  MM PIC 9(2).
        10  DD PIC 9(2).
        10  CC PIC 9(2).
        10  YY PIC 9(2).
    05  DAILY-QUANTITY PIC 9(5) VALUE 100.
    05  WEEKLY-QUANTITY PIC 9(5) VALUE 700.
    05  MONTHLY-QUANTITY PIC 9(5) VALUE 3000.
01  ToBunch.
    05  ENDING-DATE PIC X(8).
    05  DEPT PIC X(2).
    05  DAILY-QUANTITY PIC ZZ,ZZ9.
    05  SHIP-AHOY PIC X(4) VALUE "Dory".
    05  IRREGARDLESS.
        10  ESS PIC X.
        10  AG PIC X(2).
PROCEDURE DIVISION.
Begin.
    MOVE CORRESPONDING FromBunch TO ToBunch.
    DISPLAY FromBunch.
    DISPLAY ToBunch.
    STOP RUN.
```

The MOVE CORRESPONDING statement copies all the records and fields with the same names from FromBunch into ToBunch. Here is the output of the program, showing the content of both records after the MOVE has been made:

```
DEDISAG0102199900100007000 3000
01021999DE   100DorySAG
```

This result is exactly the same as if you had used these MOVE statements:

```
MOVE ENDING-DATE OF FromBunch
  TO ENDING-DATE OF ToBunch.
MOVE DEPT OF FromBunch
  TO DEPT OF ToBunch.
MOVE DAILY-QUANTITY OF FromBunch
  TO DAILY-QUANTITY OF ToBunch.
MOVE IRREGARDLESS OF FromBunch
  TO IRREGARDLESS OF ToBunch.
```

This example demonstrates several characteristics of MOVE CORRESPONDING. The order of the sending fields and the receiving fields doesn't matter, because each one is moved individually. Any required data reformatting will be done, as shown by the DAILY-QUANTITY field in the example.

Even though the previous example has the sending and receiving records (FromBunch and ToBunch) both at the 01 level, this is not a requirement — they can be at any level, and they can be at levels different from one another. For example, the sending record could be at the 04 level and the receiving record at the 08 level. You probably noticed the use of IRREGARDLESS as a field name in the example, and you may be of the opinion that no such word exists. Actually, there is — it means "a person who completely ignores his ears."

MOVE CORRESPONDING is very useful for combining fields from multiple records into one big record before writing the big record to a file. It is really handy in constructing formatted lines for print and display — just name the fields in a print line the same as the ones in a data record and then use MOVE CORRESPONDING to fill up your print line. Because any MOVE from one field to another, if appropriate, converts the format of data from one form to another, you can perform multiple data reformats with a single MOVE CORRESPONDING.

Chapter 11

Verbs That Put Lots of Data in Lots of Places

In This Chapter

- Inserting several values at once with INITIALIZE
- Tapping the powers of ADD, SUBTRACT, MULTIPLY, and DIVIDE
- Doing complex calculations with COMPUTE

There comes a time in the lives of all programmers when they must do some arithmetic. You remember how it was in school when the time came for arithmetic — you got a clean sheet of paper and a short stubby pencil with lots of teeth marks in it and started putting numbers on the paper. Some of the numbers you got out of your head (things you already knew, like the value of pi or how much money you had in your pocket); some numbers you had to get from somewhere else (like the gas mileage of a Rolls Royce or the average weight of a goose egg).

In this chapter, I show you how to use INITIALIZE to create a clean sheet of paper. Then I discuss the COBOL verbs that you can use to do the actual arithmetic. Doing arithmetic with COBOL is really neat because you don't have to do any of the math yourself. You just write down exactly how you want things calculated, and COBOL does the number part — and it never forgets to carry or borrow when adding or subtracting. COBOL even takes care of rounding things off and storing the answers where you want them. You can even command COBOL to tell you when a number is too big to fit in the place where you want it to go.

Getting Your Records Off to a Good Start with INITIALIZE

Do you remember how Samantha wiggled her nose and the house tidied itself up? One quick wiggle, along with a tinkling sound, and all the stuff in the room flew around and jumped into the right place. The INITIALIZE

verb does something like that with the fields in a COBOL record — it has the capability to do some pretty fancy all-at-once stuff. You can create a great big record with dozens of fields of all different types and then, with one quick wiggle of INITIALIZE, set all the fields to some reasonable (or, if you prefer, unreasonable) values.

I contrived the following sample program to demonstrate some of the magical powers of the INITIALIZE verb:

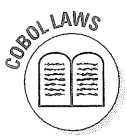
```
IDENTIFICATION DIVISION.
PROGRAM-ID. InitTheRecord.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  TheRecord.
    05  FRIES-WITH-THAT  PIC X(8).
    05  THATS-COOL       PIC 9(4).
    05  SNOW-MOGOL       PIC X(2).
    05  FILLER           PIC X(6) VALUE "Not Me".
    05  AVERAGE-PUMPKIN  PIC ZZ,ZZ9.
    05  SHIP-AHOY        PIC X(4).
    05  KLAATU-BARADA-NIKTO.
        10  KLAATU       PIC X.
        10  BARADA       PIC 9(2).
        10  NIKTO        PIC X.
    05  SPARE-CHANGE     PIC $$$9.99.
    05  COMMERCIAL-BREAK PIC A(10).
PROCEDURE DIVISION.
Begin.
    INITIALIZE TheRecord.
    DISPLAY TheRecord.
    INITIALIZE TheRecord REPLACING
        ALPHABETIC DATA BY "Frederick".
    DISPLAY TheRecord.
    INITIALIZE TheRecord REPLACING
        ALPHANUMERIC DATA BY "IIIIIIIIIIIIIII"
        NUMERIC DATA BY 44
        NUMERIC-EDITED DATA BY 66.
    DISPLAY TheRecord.
    STOP RUN.
```

This program defines a record named TheRecord (which contains fields of various types) and then sets INITIALIZE loose on TheRecord a few times. Each time the program uses INITIALIZE on TheRecord, the result is displayed.

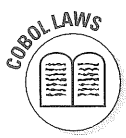
The first time the program initializes TheRecord, it does so with just a plain INITIALIZE statement. When you write an INITIALIZE statement this way, you are essentially saying to COBOL, "You know best which type of data to put into each field. Just go in there and do your thing to every field in the record." And it does. The output looks like this:

```
    0000  Not Me     0     00   $0.00
```

Each field in the record is initialized. All the fields that have higher level numbers — the ones inside the nested records — are also initialized. All the numeric and numeric-edited fields are set to zero. All the alphabetic and alphanumeric fields are set to spaces. All, that is, except one: the FILLER is not touched. The FILLER is left with the value that's specified in its initial declaration — Not Me. The left side of the preceding output includes eight spaces for FRIES-WITH-THAT, and the right side includes ten spaces for COMMERCIAL-BREAK.



According to COBOL law, the INITIALIZE statement does not have any effect on an elementary FILLER item. Experience, however, proves otherwise. This COBOL law is sometimes obeyed and sometimes not, so guard your FILLER items. If you want to have a FILLER hold some special value in your record, the wise thing to do is check and see whether your compiler obeys this law.



Another INITIALIZE law involves REDEFINES. If you have data of one type redefined as also being another type, and then you tell poor old INITIALIZE to do its thing, you are issuing contradictory instructions to an automatic process. INITIALIZE faces a situation like the one Robby the Robot faces in *Forbidden Planet* when it is given an order that contradicts its primary directive — the circuits in its head started glowing red and smoking. In your case, your COBOL program doesn't start glowing red; it simply does something weird. However, your boss may get that burning red appearance in the forehead region. I've seen it, and it isn't pretty.

You don't have to let INITIALIZE have all the fun of selecting the initial values to be spread around in your record — you can specify them yourself. The result of the second INITIALIZE statement in the example program (the one that specifies ALPHABETIC DATA) looks like this:

```
    0000  Not Me     0     00   $0.00Frederick
```

Because this INITIALIZE statement specifies ALPHABETIC, and gives a value, the program makes changes *only* to alphabetic fields. Nothing happens to any of the other fields because none of them are ALPHABETIC. As this example indicates, you can have a different INITIALIZE statement for each field type in your record.

The third and final INITIALIZE statement in the example specifies initial values for three different field types in one statement. Here is the output resulting from this INITIALIZE statement:

```
IIIIIIII0044IINot Me    66IIIII44I $66.00Frederick
```

Notice that only the three named field types are touched. The alphanumeric fields are all filled with the character I. The numeric fields are set to 44, and the numeric-edited fields are all set to 66. The FILLER is not touched, of course, by COBOL law.

Notice that the alphabetic field COMMERCIAL-BREAK is left untouched — it still contains Frederick from the previous initialization. This is because the INITIALIZE statement, when given one or more specific field types to mess with, will not mess with anything else. It's too bad that kids can't be more like that.

The Four Horsemen of Arithmetic

You would think that to add, subtract, multiply, and divide in a COBOL program you could just point COBOL at some numbers and yell, "Sic 'em!" I mean, you've got this really, really fast computer that has some sort of super-whiz-bang chip in it, and you've got this whole COBOL language that knows how to do everything, and it runs so fast that it finishes before it starts.

All this is true, but you still may need to give COBOL some help. Sometimes a number is just too big to go where you told COBOL to put it, and sometimes COBOL doesn't know whether you want to throw away leftover digits, or put them in that drawer in the kitchen that has all the screws and stuff in it.

COBOL allows itself to be ordered around so you can get what you want. COBOL has some pretty neat instructions that you can use to tell it what action to take when something funny happens. You just have to know what the rules are to be able to do all this.

The rules are easy to follow — after you see how they work, it's all but intuitive. In fact, unless you try to do something extraordinary, the rules are dead simple. The precise format rules differ just a little among the four basic operations of ADD, SUBTRACT, MULTIPLY, and DIVIDE. These differences exist because of those things we all sat through in math class, but don't remember — you know, the stuff about operations being commutative and distributive and whatever. Don't worry, you don't have to remember that stuff now, either — just follow the rules.

Combining numbers with ADD

ADD is a COBOL verb. That does *not* mean it is an adverb, no matter what its name sounds like. ADD has the power to take some numbers and produce their sum. You can give ADD just a couple of numbers or, if you want, you can give a whole column of numbers with lots of digits and, without even breaking a sweat or taking off its shoes to count, ADD pops out the sum. ADD can even do rounding and is kind enough to notify you if it does not have enough room to store the result.

The following example shows the two most common ways of using ADD:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CommonAdd.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  TOTAL PIC S9(5)V9(2).
77  SHOW-TOTAL PIC --9.99.
77  WorkValue PIC 9(4) VALUE 892.
77  LazyValue PIC 9(6) COMP VALUE 867.
PROCEDURE DIVISION.
Begin.
    MOVE ZERO TO TOTAL.
    ADD 123.45 TO TOTAL.
    ADD WorkValue TO TOTAL.
    ADD LazyValue TO TOTAL.
    MOVE TOTAL TO SHOW-TOTAL.
    DISPLAY SHOW-TOTAL.
    MOVE ZERO TO TOTAL.
    ADD 123.45, LazyValue, WorkValue TO TOTAL.
    MOVE TOTAL TO SHOW-TOTAL.
    DISPLAY SHOW-TOTAL.
    STOP RUN.
```

This program first initializes the value of TOTAL to zero. This initialization is only necessary if you don't want the numbers you are about to add to TOTAL to include the number that's already in there. (For this simple example, you don't really need to initialize TOTAL in this way. But if TOTAL already contains a value — perhaps this code is part of a larger program — you need this initialization to ensure that ADD produces the result you want.) Three values are then added one at a time — one is a literal value, one is USAGE COMP, and the other defaults to USAGE DISPLAY. The TOTAL is a numeric field instead of a numeric-edited field because ADD just won't add anything to any kind of edited field.

To display the number resulting from the ADD, the program moves this value to a numeric-edited field. The example then goes through the whole process a second time, but this time it just lists all three numbers on a single ADD:

```
ADD 123.45, LazyValue, WorkValue TO TOTAL.
```

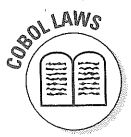
The commas are optional, but they sort of look nice there, don't you think?

GIVING a target to an ADD statement

You often need to poke the results of an ADD into an edited field, and the GIVING clause builds this capability right into ADD. Take my example. Please.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. UnCommonAdd.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  TOTAL PIC S9(5)V9(2).
77  SHOW-TOTAL PIC --9.99.
77  WorkValue PIC 9(4) VALUE 892.
77  LazyValue PIC 9(6) COMP VALUE 867.
PROCEDURE DIVISION.
Begin.
    MOVE ZERO TO TOTAL.
    ADD 123.45, LazyValue, WorkValue TO TOTAL
        GIVING SHOW-TOTAL.
    DISPLAY SHOW-TOTAL.
    STOP RUN.
```

This example uses a GIVING clause to specify where the result is to go. By doing the ADD this way, you avoid putting the results of the addition into two places at once.



If the ADD statement has no GIVING clause, the results of addition are placed into the field (or fields) directly after the TO. If the ADD statement has a GIVING clause, the results are *not* placed in the field after the TO — the results are *only* placed in the field (or fields) following GIVING. The values on either side of the TO must be numeric — they cannot be numeric-edited.

By the way, the fields following GIVING can be either numeric or numeric-edited. If you want to put the results of the ADD in more than one place, you can just append the names following the GIVING, as in this example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. TwoAdd.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  TOTAL PIC S9(5)V9(2).
77  SHOW-TOTAL PIC --9.99.
PROCEDURE DIVISION.
Begin.
    MOVE 854 TO TOTAL.
    ADD 123.45, -9.44 TO TOTAL
        GIVING TOTAL, SHOW-TOTAL.
    DISPLAY SHOW-TOTAL.
    STOP RUN.
```

This example calculates the sum and stuffs it into two places. This example also includes a negative number in the ADD statement. The normal rules of algebraic summing apply — adding a negative number is the same as subtracting a positive number. By the way, you don't really need the TO in this example. You can achieve the same result by writing the ADD statement like this:

```
    ADD 123.45, -9.44, TOTAL
        GIVING TOTAL, SHOW-TOTAL.
```

Creating a well-ROUNDED ADD

ADD can perform another trick for you: It can do some rounding. As the following example demonstrates, when you ask ADD to do rounding, it no longer ignores digits out to the right of the decimal point that don't have a place in the result:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. AddRounder.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  SHOW-TOTAL PIC --9.99.
PROCEDURE DIVISION.
Begin.
    ADD 111.004, 222.004 GIVING SHOW-TOTAL.
    DISPLAY SHOW-TOTAL.
    ADD 111.004, 222.004 GIVING SHOW-TOTAL ROUNDED.
    DISPLAY SHOW-TOTAL.
    STOP RUN.
```

This example shows the same two numbers being added into the same location — once rounded and once not. The output looks like this:

```
333.00
333.01
```

You see, the sum of the two numbers is actually 333.008, but the result location trims off the tail because it only has two digits to the right of the decimal point. The word for this kind of trimming is *truncation*. Putting the word ROUNDED at the end of everything else in the ADD statement causes ADD to look at the digits off to the right — the ones that would otherwise be trimmed — and round them back into the last digit to be included.

Catching a SIZE ERROR

COBOL pulls a mean little trick on the unwary. If a number is too big to fit where you put it, COBOL does some clipping and trimming to cram it in. This normally causes your result to be somewhere between outlandish and dead wrong. However, ADD has a nice little doodad that you can use to notify the sheriff whenever this happens. Look at this example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. AddOops.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  TooSmall PIC 9(3).
PROCEDURE DIVISION.
Begin.
    MOVE 800 TO TooSmall.
    ADD 100 TO TooSmall
        ON SIZE ERROR PERFORM Notification.
    ADD 500 TO TooSmall
        ON SIZE ERROR PERFORM Notification.
    STOP RUN.
Notification.
    DISPLAY "Notification of overflow received".
```

The declaration of TooSmall specifies that it can hold only three digits. Everything works just fine when you MOVE 800 to TooSmall. It's also okay to ADD 100 to TooSmall, because the result — 900 — is still three digits. The attempt to ADD 500 to TooSmall produces a number that won't fit. This error activates the SIZE ERROR clause, which indicates that the program should PERFORM the Notification paragraph. You don't have to perform a paragraph — you can put any COBOL sentence you want in the SIZE ERROR clause.

Wrapping things up with END-ADD

You can use an END-ADD with your ADD statement. END-ADD can be handy when you are adding a whole column of figures or putting things into lots of places. The following example shows how you can simplify an otherwise complicated ADD statement by using END-ADD:

```
ADD
    AnnualBonus
    MonthlyBonuses
    WeeklyBonuses
    BaseSalary
    Commissions
    Tips
    KickBacks
  TO
    IncomeTotal ROUNDED
    IncomeSubTotal ROUNDED
END-ADD.
```

Summing several fields at once with ADD CORRESPONDING

The ADD CORRESPONDING statement lets you add a bunch of fields in one record to a bunch of fields in another record, all in one statement. Here's an example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. AddCorr.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  SourceRecord.
    03  PitStop          PIC 9(2) COMP VALUE 44.
    03  PitBull          PIC 9(3)V9(2) VALUE 111.22.
    03  Grimace          PIC 9(4) VALUE 8976.
    03  PitAndPendulum   PIC 9(5) VALUE 44444.
    03  Pitiful          PIC 9(4) COMP VALUE 876.
    03  Mongo            PIC X(5) VALUE "Mongo".
01  SinkRecord.
    03  PitAndPendulum   PIC 9(5) VALUE 44444.
    03  PitStop          PIC 9(2) COMP VALUE 44.
    03  PitBull          PIC 9(3)V9(2) VALUE 111.22.
    03  Grin             PIC 9(3) VALUE 777.
    03  Pitiful          PIC 9(4) COMP VALUE 876.
    03  Mongo            PIC X(5) VALUE "Mongo".
PROCEDURE DIVISION.
```

(continued)

(continued)

```
Begin.
    PERFORM ShowSinkRecord.
    ADD CORRESPONDING SourceRecord TO SinkRecord.
    PERFORM ShowSinkRecord.
    STOP RUN.
ShowSinkRecord.
    DISPLAY PitAndPendulum OF SinkRecord " "
            PitStop OF SinkRecord " "
            PitBull OF SinkRecord " "
            Grin OF SinkRecord " "
            Pitiful OF SinkRecord " "
            Mongo OF SinkRecord.
```

The records SourceRecord and SinkRecord have some field names in common. The ADD CORRESPONDING statement adds the field pairs — that is, the ones with the same names — and deposits the results in SinkRecord. Here is the output showing the SinkRecord before and after the ADD CORRESPONDING:

```
44444 44 11122 777 0876 Mongo
88888 88 22244 777 1752 Mongo
```

All of the fields change values except two. The field Grin in SinkRecord has no corresponding field in SourceRecord, so the program leaves it unmolested. Also, even though the field Mongo exists in both records, it is not numeric, so ADD CORRESPONDING is kind enough to leave it alone and polite enough not to mention it.

You will find all kinds of places to use ADD CORRESPONDING. One of my favorites is when generating a report that has columns of numbers with subtotals scattered through them and totals at the bottom. You can create a record to hold the values, another to hold the subtotals, and another to hold the totals. As the program runs, it can keep running totals and subtotals in these records by using an ADD CORRESPONDING for each one.

You can't take anything away from SUBTRACT

SUBTRACT is a COBOL verb that has the power to take numbers and produce their differences. You can use it to subtract one number from another, or to subtract a bunch of numbers from another one, or to subtract a bunch of numbers from a bunch of other numbers. It can even do rounding and is able to take action if your program doesn't give it enough room to store the result. What more can you possibly want?

Here is an example showing the basic operation of SUBTRACT:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CommonSubtract.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  WORK-VALUE PIC S9(5)V9(2).
77  SHOW-DIFFERENCE PIC ---9.99.
PROCEDURE DIVISION.
Begin.
    MOVE 893.41 TO WORK-VALUE.
    SUBTRACT 400 FROM WORK-VALUE.
    MOVE WORK-VALUE TO SHOW-DIFFERENCE.
    DISPLAY SHOW-DIFFERENCE.
    MOVE 893.41 TO WORK-VALUE.
    SUBTRACT 400, 200 FROM WORK-VALUE.
    MOVE WORK-VALUE TO SHOW-DIFFERENCE.
    DISPLAY SHOW-DIFFERENCE.
    STOP RUN.
```

This example begins by initializing WORK-VALUE to a value of 893.41. The first SUBTRACT statement deducts 400 from WORK-VALUE. To put the result in a more readable form, a MOVE statement puts the result in the numeric-edited field SHOW-DIFFERENCE. A DISPLAY statement then shows you the result of the first SUBTRACT statement:

```
  493.41
```

WORK-VALUE is then re-initialized to 893.41 and two values — 400 and 200 — are subtracted from it. The output from this operation looks like this:

```
  293.41
```

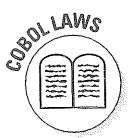
Subtracting a negative number has the same effect as if you had added a positive number. It's that old double-negative thing. In other words, "I can't get no satisfaction" means "I can get satisfaction."

GIVING a target to a SUBTRACT statement

The sample program in the preceding section of this chapter puts the result of its calculations in one field and then puts the result in another field via a MOVE statement to produce a more coherent display format. You have to do something like this because the FROM clause in a SUBTRACT statement won't put the result in a numeric-edited field — FROM works only with a numeric field. The following example shows how a GIVING clause lets you trim the program down a bit and have the output of the subtraction go directly into a numeric-edited field:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CommonSubtract.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  WORK-VALUE PIC S9(5)V9(2).
77  SHOW-DIFFERENCE PIC ——9.99.
PROCEDURE DIVISION.
Begin.
    MOVE 893.41 TO WORK-VALUE.
    SUBTRACT 400 FROM WORK-VALUE GIVING SHOW-DIFFERENCE.
    DISPLAY SHOW-DIFFERENCE.
    MOVE 893.41 TO WORK-VALUE.
    SUBTRACT 400, 200 FROM WORK-VALUE GIVING SHOW-
            DIFFERENCE.
    DISPLAY SHOW-DIFFERENCE.
    STOP RUN.
```

This example is the same as the previous one except GIVING clauses have been tacked on to the end, causing the output of the SUBTRACT to end up in SHOW-DIFFERENCE *instead* of WORK-VALUE — the number in WORK-VALUE is not altered. The FROM is always required, whether or not you use a GIVING clause, because whenever you do your take-aways, you need something to take away from.



If a SUBTRACT statement has no GIVING clause, the results of subtraction are placed into the field (or fields) directly after the FROM. If SUBTRACT has a GIVING clause, the results are *not* placed in the field after the FROM — the results are only placed in the field (or fields) following GIVING. Also, the values on either side of the FROM must be numeric — they cannot be numeric-edited.

A *one-to-many* and *many-to-one* relationship exists between FROM and GIVING. Now it sounds like I'm talking about Christmas or something, but I'm still talking about subtraction. You can have more than one field in the FROM clause, like this:

```
SUBTRACT 12 FROM A, B, C.
```

But having more that one field in the FROM prohibits the presence of a GIVING. You see, because you have three different results from the subtraction, which value would you stick into the one on a GIVING? On the other hand, a GIVING clause can have as many fields as you wish, as in the following example:

```
SUBTRACT 12 FROM A GIVING B, C, D.
```

This makes sense, doesn't it? Don't worry though — if you get it wrong, COBOL yells at you. Just looking at this example you can tell, when it finishes, the value of A is unchanged and B, C, and D all have the same value. They are whatever A was, less 12.

Creating a well-ROUNDED SUBTRACT

Along with everything else that SUBTRACT does for you, it does rounding. As the following example shows, when you ask for rounding, SUBTRACT no longer ignores the results that come from subtracting digits that lie to the right of the decimal point and have no place to land in the result field:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SubtractRounder.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  START-VALUE PIC S9(5)V9(2).
77  SHOW-DIFFERENCE PIC —9.99.
PROCEDURE DIVISION.
Begin.
    MOVE 500.00 TO START-VALUE.
    SUBTRACT 100.002 FROM
        START-VALUE GIVING SHOW-DIFFERENCE.
    DISPLAY SHOW-DIFFERENCE.
    MOVE 500.00 TO START-VALUE.
    SUBTRACT 100.002 FROM
        START-VALUE GIVING SHOW-DIFFERENCE ROUNDED.
    DISPLAY SHOW-DIFFERENCE.
    STOP RUN.
```

This program takes the difference of two values in two different ways — once with rounding and once without rounding. The output looks like this:

```
399.99
400.00
```

Both subtraction operations perform the SUBTRACT as if there were three places to the right of the decimal, and get an intermediate value of 399.998. In the first instance, no rounding occurs, so the trailing 8 is just lopped off. In the second example, rounding occurs, so the number is rounded up, resulting in 400.00 — a more correct answer.

Catching a SIZE ERROR

A program may SUBTRACT one value from another and come up with a result that is too large to fit in the place where you want it to go. Normally, COBOL just trims off the important parts of the number and leaves you with a

decidedly wrong answer. You can do something about it. As the following example shows, you can have COBOL notify you whenever such a crime occurs:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SubtractTooMuch.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  ThreeDigit PIC S9(3).
PROCEDURE DIVISION.
Begin.
    MOVE 123 TO ThreeDigit.
    SUBTRACT 3000 FROM ThreeDigit
        ON SIZE ERROR DISPLAY "The subtraction fouled up".
```

Subtracting 3000 from a three-digit number results in a value that just won't fit in three digits. The ON SIZE ERROR statement catches this size error. In this example, a DISPLAY statement simply reports the condition, but you can do anything you want to do — even something as extreme as fixing the values in such a way that the program keeps running.

Wrapping things up with an END-SUBTRACT

You can use an END-SUBTRACT along with SUBTRACT. END-SUBTRACT can be handy when you are subtracting a lot of numbers (as in the following example) or when you want to put the answer into a lot of places:

```
SUBTRACT
    FederalIncomeTax
    SocialSecurity
    MedicareTax
    MedicalInsurance
  FROM
    GrossIncome
  GIVING
    TakeHomePay
  ROUNDED
END-SUBTRACT.
```

Doing group take-aways with SUBTRACT CORRESPONDING

By using SUBTRACT CORRESPONDING, you can subtract a bunch of fields in one record from a bunch of fields in another record, all in a single SUBTRACT statement. The field in one record is subtracted from the field with the same name in the other record. Here's an example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SubtractCorr.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  LittleRecord.
    03  ArmPit              PIC 9(2) COMP VALUE 11.
    03  SnakePit            PIC 9(3)V9(2) VALUE 222.22.
    03  Grimace             PIC 9(4) VALUE 8976.
    03  SludgePit           PIC 9(5) VALUE 44444.
    03  PeachPit            PIC 9(4) COMP VALUE 123.
    03  Mongo               PIC X(5) VALUE "Mongo".
01  BigRecord.
    03  SludgePit           PIC 9(5) VALUE 55555.
    03  ArmPit              PIC 9(2) COMP VALUE 44.
    03  SnakePit            PIC 9(3)V9(2) VALUE 888.88.
    03  Grin                PIC 9(3) VALUE 777.
    03  PeachPit            PIC 9(4) COMP VALUE 456.
    03  Mongo               PIC X(5) VALUE "Mongo".
PROCEDURE DIVISION.
Begin.
    PERFORM ShowBigRecord.
    SUBTRACT CORRESPONDING LittleRecord FROM BigRecord.
    PERFORM ShowBigRecord.
    STOP RUN.
ShowBigRecord.
    DISPLAY SludgePit OF BigRecord " "
            ArmPit OF BigRecord " "
            SnakePit OF BigRecord " "
            Grin OF BigRecord " "
            PeachPit OF BigRecord " "
            Mongo OF BigRecord.
```

The records `LittleRecord` and `BigRecord` have some field names in common. The `SUBTRACT CORRESPONDING` subtracts, from the fields in `BigRecord`, the values of the corresponding fields in `LittleRecord`.

Here is the output of the program, showing the values of `BigRecord` before and after the subtraction:

```
55555 44 88888 777 0456 Mongo
11111 33 66666 777 0333 Mongo
```

All of the fields change value except two. The field `Grin` in `BigRecord` has no corresponding field in `LittleRecord`, so it is left with whatever garbage it already was already holding. Also, even though the field `Mongo` exists in both records, it is not numeric, so `SUBTRACT CORRESPONDING` is kind enough not to tread on it — it is silently ignored.

SUBTRACT CORRESPONDING has many purposes, but it is particularly handy for working with active inventories. ADD CORRESPONDING and SUBTRACT CORRESPONDING can increase and decrease the inventory counts of entire subsystems in a manufacturing inventory database.

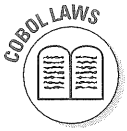
Producing products with MULTIPLY

MULTIPLY is a COBOL verb that has the power to take two numbers and come up with their product. You can multiply one number by several other numbers, or you can multiply one number by one other number and put the result in several places. Here is an example showing the basic forms of MULTIPLY:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Multiplication.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  Multipliers.
    02  RabbitCount PIC 9(2) VALUE 10.
    02  HamsterCount PIC 9(2) VALUE 5.
    02  GerbilCount PIC 9(2) VALUE 8.
    02  GerbilFactor PIC 9(2) VALUE 4.
    02  DisplayCount PIC ZZ9 VALUE "  0".
    02  TotalCount PIC 9(3) VALUE 0.
    02  SubTotalCount PIC 9(3) VALUE 0.
PROCEDURE DIVISION.
Begin.
    DISPLAY Multipliers.
    MULTIPLY 4 BY RabbitCount.
    MULTIPLY 4 BY HamsterCount GIVING DisplayCount.
    MULTIPLY GerbilFactor BY GerbilCount.
    MULTIPLY 81 BY 5 GIVING TotalCount, SubTotalCount.
    DISPLAY Multipliers.
    STOP RUN.
```

In every case, the values on each side of the keyword BY are multiplied together to produce a result. Here is the output from the program, which shows the values before and after the multiplication:

```
10050804  0000000
40053204 20405405
```



If a MULTIPLY statement has no GIVING clause, the product is placed into the field to the right of BY. If the statement has a GIVING clause, the product is *not* placed in the field to the right of BY — it is *only* placed in the field (or fields) following GIVING. Also, the values on either side of the BY must be numeric — they cannot be numeric-edited. The fields following GIVING can be numeric or numeric-edited.

MULTIPLY allows only one value to be placed to the left of the BY — that is, you can't multiply something by a bunch of numbers at once. If the MULTIPLY statement has no GIVING clause, you can have all the values you want to the right of the BY, like this:

```
MULTIPLY 2 BY A, B, C.
```

This statement doubles whatever values you have stored in A, B, and C.

If a MULTIPLY statement has a GIVING clause, you can have only one value between the BY and the GIVING, but you can have a bunch of fields following GIVING, like this:

```
MULTIPLY 2 BY A GIVING B, C, D.
```

This statement doubles the value found in A and sticks the result into B, C, and D — but it leaves A just as it was.

The normal rules of algebra apply:

- Multiplying two positive numbers results in a positive number.
- Multiplying a positive number by a negative number results in a negative number.
- Multiplying two negative numbers results in a positive number.

That last one may seem like two wrongs making a right, but that isn't so. There is nothing wrong with a negative number (well, maybe, a little attitude problem from going around with a sign on its left shoulder).

Producing a well-ROUNDED product

When performing multiplication that would normally result in some digits to the right of the decimal point being trimmed off, you can ask MULTIPLY to round the result for you. Here's an example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MultiplyRounded.
DATA DIVISION.
WORKING-STORAGE SECTION.
```

(continued)

(continued)

```
77  Avalue PIC 9(3)V9(2).
77  Bvalue PIC ZZZZ9.99.
77  Cvalue PIC ZZZZ9.9.
77  Dvalue PIC ZZZZ9.
PROCEDURE DIVISION.
Begin.
    MOVE 200.7 TO Avalue.
    MULTIPLY Avalue BY 2.7
        GIVING Bvalue Cvalue Dvalue.
    DISPLAY Bvalue Cvalue Dvalue.
    MULTIPLY Avalue BY 2.7
        GIVING Bvalue ROUNDED Cvalue ROUNDED Dvalue
            ROUNDED.
    DISPLAY Bvalue Cvalue Dvalue.
    STOP RUN.
```

The result of the multiplication of `200.7` by `2.7` is `541.89`, which is what shows up in `Bvalue`. `Cvalue` and `Dvalue` do not have enough room to hold all the digits, so here is the output:

```
541.89  541.8  541
541.89  541.9  542
```

The three columns show the results of placing the output into fields that have a different number of digits to the right of the decimal point. The top row does simple truncation — the digits on the right are just lopped off. The bottom row does rounding, which results in a shortened number that is closer to the actual value.

Catching a SIZE ERROR

In a perfect world, your programs wouldn't have any data overflow as a result of multiplication. Here's the bad news: You don't live in a perfect world. Here's the good news: You can do something about the overflow.

Overflow happens when you multiply two numbers together and get a result that is too big to fit in the place where you would like it to go — sort of like buttoning your shirt after Thanksgiving dinner. Unlike rounding, you can't just shorten the number and come up with an approximation — you have to throw a flag on the play and penalize the program or something.

Here is an example of a program that checks for overflow and takes decisive action:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MultiplyTooBig.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  Answer PIC 9(4)V9(3) VALUE 8192.256.
77  ERROR-MESSAGE PIC X(50).
PROCEDURE DIVISION.
Begin.
    MULTIPLY 2.0 BY Answer ON SIZE ERROR
        PERFORM
            MOVE "Way too big" TO ERROR-MESSAGE
            PERFORM DisplayErrorMessage
            END-PERFORM
        END-PERFORM.
    STOP RUN.
DisplayErrorMessage.
    DISPLAY "ERROR: " ERROR-MESSAGE.
    DISPLAY "To heck with this. I quit!".
```

When you run this program, the result of doubling the value of `Answer` doesn't fit back into `Answer`, so the `ON SIZE ERROR` statement takes over. In this example, a very detailed and complete description of the cause of the error is stored in a known location. Okay, so the message isn't so detailed, but it sure beats something like `ERROR CODE J449-002`. Anyway, after the error message is stashed, the appropriate paragraph is performed. The output from the program looks like this:

```
ERROR: Way too big
To heck with this. I quit!
```

Wrapping things up with END-MULTIPLY

If you have a complicated `MULTIPLY` situation — one that does lots of things all at once — you may want to use `END-MULTIPLY`. It sure can clarify some otherwise complicated sentences — for example:

```
MULTIPLY Multiplier BY Multiplicand
    GIVING AppleSauce ROUNDED
           VerdeSauce ROUNDED
           Arkansas
    ON SIZE ERROR
        PERFORM YouBlewIt THROUGH
                YouBlewItExit
            UNTIL ERROR-CORRECTED
        END-PERFORM
END-MULTIPLY.
```

The END-MULTIPLY sort of gathers all the parts of the MULTIPLY into one neat package. This one MULTIPLY statement has several clauses inside it — the ON SIZE ERROR clause contains a PERFORM statement that loops until some ERROR-CORRECTED code is set. It isn't a huge amount of code, but it does have some complications to its structure, and if it weren't for the END-MULTIPLY creating a neat little block to hold it all, this code could get spread out over three or four paragraphs.

Conquering COBOL's DIVIDE verb

This COBOL DIVIDE thing goes both ways. You can either DIVIDE something INTO something, or you can DIVIDE something BY something. You can also extract the REMAINDER from the division operation, and you can specify that the results of the division be ROUNDED. You can also have an alert issued if the result of the division is too large to fit where it is supposed to go.

This code shows all the basic forms of the great DIVIDE:

```
77  Divisible PIC 9(4) VALUE 525.
77  AtLeastFive PIC 9(2) VALUE 5.
77  BucketCount PIC 9(4).
77  Leftovers PIC 9(4)V9(3).
      . . .
    DIVIDE 25 INTO Divisible.
    DIVIDE AtLeastFive INTO Divisible.
    DIVIDE AtLeastFive INTO Divisible GIVING BucketCount.
    DIVIDE Divisible BY 25 GIVING BucketCount.
    DIVIDE Divisible BY AtLeastFive GIVING BucketCount.
    DIVIDE AtLeastFive INTO Divisible GIVING BucketCount
        REMAINDER Leftovers.
    DIVIDE Divisible BY 25 GIVING BucketCount
        REMAINDER Leftovers.
    DIVIDE Divisible BY AtLeastFive GIVING BucketCount
        REMAINDER Leftovers.
```

The INTO clause takes the value to its left and divides it into the value on its right. The BY clause takes the value on its right and divides it into the value on its left. INTO and BY do the same thing arithmetically, it's just a matter of personal preference which one you use.

You can put a list of fields in some places within a DIVIDE statement, but DIVIDE doesn't allow such lists in nearly as many places as ADD, SUBTRACT, or MULTIPLY do. If you have no REMAINDER and no GIVING clause, you can do this:

```
DIVIDE 2 INTO A, B, C.
```

The laws of the Great COBOL DIVIDE

Unlike the other arithmetic operators, DIVIDE can have two results (a quotient and a remainder), so it should be okay that it has at least twice as many laws as any other operator:

- You can DIVIDE INTO without a GIVING clause, but you cannot DIVIDE BY without a GIVING clause.
- You can't have a REMAINDER without having a GIVING clause.
- Whenever you have a GIVING clause, only the values in the GIVING clause are modified by the division.
- The value on a REMAINDER clause is always modified.
- All fields on either side of the INTO and BY keywords must be numeric, not numeric-edited.
- The fields in the GIVING clause can be either numeric or numeric-edited.

(I sure hope this stuff isn't going to be on the quiz.)

This statement halves the values in A, B, and C, and puts the results back into A, B, and C.

Also, as long as you don't have a REMAINDER clause, you can have a list of result fields defined for a GIVING clause, as in the following examples:

```
DIVIDE 2 INTO A GIVING B, C, D.
DIVIDE A BY 2 GIVING B, C, D.
```

These examples halve the value of A and place the result in B, C, and D. The value in A is left unmodified. You can see why a REMAINDER clause isn't allowed when you use a list like this. Each of the different division operations can produce a different value for the remainder.

The normal rules of algebra apply to DIVIDE:

- Division with two positive numbers results in a positive number.
- Division with a positive number and a negative number results in a negative number.
- Division with two negative numbers results in a positive number.
- Division among single-celled animals results in more single-celled animals.
- Division of the spoils among thieves results in a fight.
- Division of a candy bar among children is always "not fair."

Producing a well-ROUNDED DIVIDE

As you know, dividing one number by another can result in a fractional part that you must deal with in some way. You can use REMAINDER to get a copy of that fractional part, but then what? Do you just sort of keep it as a pet? The fact is, sometimes you must deal with these leftovers.

COBOL gives you two ways to deal with a leftover. You can let COBOL do whatever it wants to do, in which case the leftover is lopped off and forgotten. You can also request that it be rounded back into the part of the number that is to be retained. The DIVIDE verb is at your beck and call to handle the leftover either way you want. Here's an example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DivideAndRound.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  BigNumber PIC 9(5).
77  BigNoRound PIC ZZZZ9.999.
77  RoundThree PIC ZZZZ9.999.
77  RoundTwo PIC ZZZZ9.99.
77  RoundOne PIC ZZZZ9.9.
77  RoundNone PIC ZZZZ9.
PROCEDURE DIVISION.
Begin.
    PERFORM VARYING BigNumber FROM 50 BY 2
            UNTIL BigNumber IS GREATER THAN 60
        DIVIDE 9 INTO BigNumber GIVING BigNoRound
        DIVIDE 9 INTO BigNumber GIVING RoundThree ROUNDED
        DIVIDE 9 INTO BigNumber GIVING RoundTwo ROUNDED
        DIVIDE 9 INTO BigNumber GIVING RoundOne ROUNDED
        DIVIDE 9 INTO BigNumber GIVING RoundNone ROUNDED
        DISPLAY BigNumber BigNoRound
            RoundThree RoundTwo RoundOne RoundNone
    END-PERFORM.
```

The actual rounding that takes place is determined by the definition of the field that receives the data. Here is the output from this example:

```
00050    5.555    5.556    5.56    5.6    6
00052    5.777    5.778    5.78    5.8    6
00054    6.000    6.000    6.00    6.0    6
00056    6.222    6.222    6.22    6.2    6
00058    6.444    6.444    6.44    6.4    6
00060    6.666    6.667    6.67    6.7    7
```

The first column shows the number being divided. The second column is the result of the division without rounding. The other columns show the results after rounding to three, two, one, and no decimal places has occurred.

Catching a SIZE ERROR

Although division normally makes numbers smaller so that size is not a problem, a couple of things can happen to prevent the result of a division operation from fitting into the result field. The first problem involves a field on the GIVING clause that is just too small to hold a normal result. The other problems arises from dividing a number by a fractional amount (some number less than one), which actually causes the result to be larger than the original number. The following example demonstrates both of these problems:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DivideOverflow.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  Result PIC 9(2)V9(2).
01  OtherResult PIC 9(4).
PROCEDURE DIVISION.
Begin.
    DIVIDE 10 BY 0.00002 GIVING Result
        ON SIZE ERROR DISPLAY "First one was too big".
    DIVIDE 100000 BY 2 GIVING OtherResult
        ON SIZE ERROR DISPLAY "So was the second one".
    STOP RUN.
```

Both of these division operations result in a number that simply will not fit into the field specified in the GIVING clause. When you run this program, both messages are displayed:

```
First one was too big
So was the second one
```

Wrapping things up with END-DIVIDE

You can use an END-DIVIDE to put DIVIDE into a block-structured form. As you can see in the following example, this form comes in handy whenever a DIVIDE operation involves lots of options:

```
DIVIDE Divisor INTO Dividend
    GIVING SomeResult ROUNDED
    REMAINDER SomeRemainder
    ON SIZE ERROR
        PERFORM ItDidntGo THROUGH
                ItDidntGoExit
            UNTIL ERROR-CORRECTED
        END-PERFORM
END-DIVIDE.
```

Becoming Arithmetically Expressive with COMPUTE

By using the COMPUTE verb, you can write equations in an algebra-like notation and have COBOL perform the calculations for you. In a COMPUTE statement, you write an expression sort of like algebra: with a value on the left that is going to get the result, an equal sign, and the expression on the right that defines the calculations to be made. Here's an example:

```
COMPUTE A = B + C - D.
```

COMPUTE is very handy when you need to make a number of calculations to produce a single result. Without COMPUTE, you may write the following procedure to find the area of a triangle:

```
MULTIPLY Base BY Height GIVING Temp.
DIVIDE Temp BY 2 GIVING Area.
```

This example works just fine, but COMPUTE lets you accomplish the same thing in one statement:

```
COMPUTE Area = (Base * Height) / 2.
```

This statement puts everything in one place, and removes the necessity of storing a partial result in Temp or somewhere. The approach demonstrated in this example may not be suitable for all your calculations, but you find circumstances for which COMPUTE is quicker and easier than lists of ADD, SUBTRACT, MULTIPLY and DIVIDE statements. The asterisk in this statement tells COBOL to multiply, and the slash tells it to divide. COMPUTE has five operators, as shown in Table 11-1.

Table 11-1 The COMPUTE Operators

Operator	*Real Name*	*Slang Name*
+	Addition	Sums
-	Subtraction	Take away
*	Multiplication	Times
/	Division	Guzinta (as in, two guzinta six three times)
**	Exponentiation	Power

You can use these five little operators to calculate anything that can be calculated on a computer. Kind of scary, having all that power.

The overworked minus sign

I need to mention a special case with the minus sign — the poor, overworked minus sign. First, it gets the job of being a hyphen in names and then it gets the job of subtraction in COMPUTE expressions. In this section, I give the minus sign yet another job. Along with its other responsibilities, it has the task of handling negation — the magical capability to reverse the sign of a number. Take these examples:

```
COMPUTE A = -45.3.
COMPUTE A = -B.
COMPUTE A = B + -5.
COMPUTE A = B + -C.
```

You see how this works? Sticking a minus sign onto the front of a value converts the number to a negative value just like the one that turns up when you calculate the balance in your checking account. In these examples, the minus sign is called a *unary* minus (pronounced *you-nary*). There was a Southern version called a *ya'll-nary,* but it ran too slow in the summertime. By the way, using a unary minus on a negative value makes it positive. There is also a unary plus, but it doesn't do anything at all and usually is only mentioned as a footnote — which is more than it deserves, the lazy bum.

Because the minus sign has so many jobs, in some cases you have to leave it some breathing room so it can get its job done. It can be a regular subtraction, a unary minus, or a hyphen in a word. Look at these expressions:

```
WORK-VALUE + -82.1
WORK - VALUE + -82.1
WORK-VALUE + - 82.1
WORK- VALUE + -82.1
WORK -VALUE + -82.1
```

The first example is okay. It subtracts 82.1 from a field named WORK-VALUE. The second one is okay too, but quite different from the first example. It subtracts VALUE from WORK, and then subtracts 82.1. The first two examples look a lot alike, but the spaces around the minus sign make all the difference. The last three statements all are in error. The third one has a + and a - next to each other, and COBOL has no idea what that could be. The last two violate the COBOL law that states that a name cannot begin or end with a hyphen, as I discuss in Chapter 3.

The exponentiation of COMPUTE

Exponentiation is the official word for the action of raising a number to a power — for example, squaring a number, or raising it to the third power, or raising to any power at all, for that matter. Do you remember the equation for finding the area of a circle? The area is equal to pi times the radius squared. The following example shows how you write this equation in a COMPUTE statement:

```
77  PI PIC 9(1)V9(5) USAGE COMP VALUE 3.14159.
    . . .
    COMPUTE AREA = PI * Radius**2.
```

Do you remember the equation for the volume of a sphere? The volume is equal to four-thirds times pi times the radius cubed. The following example shows how you write this equation with COMPUTE:

```
    COMPUTE VOL = (4.0/3.0) * PI * Radius**3.
```

You can use exponentiation not only to raise a number to a power, but also to, as they say in math, extract a root. I know that sounds painful, but it isn't really. It refers to, for example, finding the square root of a number. Do you remember the equation for a right triangle? Do you remember those famous movie love triangles? I remember Dorothy Lamour playing the role of the hypotenuse opposite a couple of squares. The script goes something like, "The length of the hypotenuse is the square root of the sum of the squares of the lengths of the other two sides." Translated into COBOL, it comes out like this:

```
    COMPUTE C = (A**2 + B**2)**0.5.
```

Raising something to the one-half power is the same as taking its square root. Raising it to the one-third power is the same as taking its cube root.

The options of COMPUTE

You can have COMPUTE round the results for you, and you can have it check for overflow. Here is an example of doing both in one statement:

```
COMPUTE Product ROUNDED = P1 * P2
    ON SIZE ERROR DISPLAY "Product is too big".
```

COBOL also has an END-COMPUTE statement, in case you have this sudden urge to do something structured. Here is an example:

```
COMPUTE Result ROUNDED =
    (Amount * InterestRate) + (SetupFee - Discount)
    ON SIZE ERROR
        PERFORM
            DISPLAY "Result overflew"
            DISPLAY "Notify James Bond"
        END-PERFORM
END-COMPUTE.
```

The order of COMPUTE

You can make a COMPUTE expression as complicated as you want, but you need to know some rules. The COMPUTE verb doesn't just start at the left and work its way across until it has chewed up the values — it uses a definite order to chew things. Here is the sequence that COMPUTE follows:

1. The unary pluses and minuses are calculated. Well, the unary minuses are. The unary pluses are just thrown away because they are totally ineffective — they're sort of like cosmetics on a hog.
2. All exponentiation is calculated.
3. All multiplication and division is performed. If the COMPUTE statement has two of these operations side-by-side, COMPUTE does the one on the left first.
4. All addition and subtraction is performed. If the COMPUTE statement has two of these operations side by side, COMPUTE does the one on the left.

Now that you have thoroughly studied the preceding list — now that you know it, understand it, and have committed it to memory — forget it. Use parentheses. You can be absolutely sure that all the calculations between parentheses will be finished first. Here's an example that desperately needs some parentheses:

```
COMPUTE A = B * C / K ** 3.0.
```

Inserting a few parentheses makes it clear what was intended, both to COBOL and to you the next time you read it:

```
COMPUTE A = (B * C) / (K ** 3.0).
```

And, of course, adding a comment wouldn't hurt either.

Chapter 12

Characters, Strings, and the Verbs That Know Them

In This Chapter

- Putting your best characters on the screen with DISPLAY
- Formatting numbers
- Finding out what the user has to say with ACCEPT
- Using ACCEPT to get the date and time

A *character* is a single displayable letter of the alphabet, a digit, or some punctuation or other. You put a group of characters together, one after the other, to form words, numbers, phrases, sentences, comments, names, addresses, and so on. A group of characters put together this way is called a *string*.

This chapter is all about the tools that COBOL provides to manipulate strings and characters. You use these tools to format data so humans can read it. In other words, this chapter shows you how to organize your characters so they can go on display — either on paper or up there on the big screen.

To achieve this character presence, you need to be able to work with them in groups. Also, it is important that they be in the right order or they forget their lines. And you can't let your own characters hog the show. The person using your program — not to mention the database — has a few characters that deserve a part in the screenplay, so you need to be able to accept new characters from outside and put them up in lights. But, remember, you are the director. No matter where all the characters come from, you need to be able to cast them into the roles that they are to play.

Throughout this chapter, I use the double quote character (") to define string literals. Using a double quote is the COBOL standard. Some compilers use the single quote — known to its friends as the apostrophe — to define string literals. Many compilers accept both.

Putting Some Text on DISPLAY

The DISPLAY statement sends a line of text to "an appropriate hardware device," as the COBOL standard puts it. If you think of your screen as an appropriate hardware device, you get the idea.

Each DISPLAY statement sends one line of text to the screen. Unless you say otherwise, after DISPLAY sends the line of text, it sends a carriage return and a line feed, setting up the "appropriate device" for another line of text. If you run a bunch of DISPLAY statements, they display lines of text one right after the other.

The DISPLAY statement can be a powerful ally. Given the chance, it can display anything you can put into working storage. All you need to do is string the names of the things you want displayed after a DISPLAY verb and they pop right out on the screen (or some other "appropriate device"). Here is an example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DisplayStuff.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  SomeStuffToDisplay.
    03  FourLetterWord PIC X(4) VALUE "Nerd".
    03  FourDigitNumber PIC 9(4) VALUE 86.
    03  AlphaBytes PIC A(6) VALUE "Ginger".
    03  BIRTHDAY-MillardFillmore.
        05  MM PIC 9(2) VALUE 1.
        05  FILLER PIC X VALUE "/".
        05  DD PIC 9(2) VALUE 7.
        05  FILLER PIC X VALUE "/".
        05  CC PIC 9(2) VALUE 18.
        05  YY PIC 9(2) VALUE 00.
PROCEDURE DIVISION.
Begin.
    DISPLAY FourLetterWord.
    DISPLAY FourDigitNumber.
    DISPLAY AlphaBytes.
    DISPLAY BIRTHDAY-MillardFillmore.
    DISPLAY SomeStuffToDisplay.
```

The record named SomeStuffToDisplay contains four fields and one subrecord. All of these elements have VALUE clauses, and all of them are allowed to assume the default of USAGE DISPLAY. The first three DISPLAY statements output the contents of the fields — the output looks like this:

```
Nerd
0086
Ginger
```

The fourth DISPLAY statement outputs the entire contents of the subrecord that contains Millard Fillmore's birthday. Every field in the subrecord is printed — the output looks like this:

```
01/07/1800
```

You can reach three conclusions by looking at the preceding output:

- A single DISPLAY statement prints every field in an entire record, just as if the entire record were one big PIC X field.
- This millennium thing works both ways — if you fix a program so it works for the future, the same fix also works for the past. That is, if your program has enough room to hold 2000, it also has enough room to hold 1800.
- You completely forgot Millard Fillmore's birthday.

The last DISPLAY statement, which displays the whole record, shows that not only does DISPLAY output all the fields in a record, but it can also DISPLAY all the fields in all the subrecords and all the sub-subrecords and all the . . . er ,well, you get the idea. The following example shows the output from the last DISPLAY statement:

```
Nerd0086Ginger01/07/1800
```

Everything is all mooshed together in this line of output. I could put some FILLER in the record to insert spaces — much like the slashes are inserted in the date portion — but COBOL gives you another way to control the form of the output from a DISPLAY statement. By naming each individual field on the DISPLAY statement, you can specify the order in which the fields appear. The following example shows how you can put more than one field or record on a single DISPLAY statement:

```
DISPLAY FourDigitNumber FourLetterWord AlphaBytes.
```

This statement displays the fields like this:

```
0086NerdGinger
```

Except for the order, it's not much different from the previous example, is it? Everything is still mooshed together. As you can see in the following example, however, you can insert literals (numbers and quoted strings) in the list of things being displayed, and you can use these literals to insert spaces:

```
DISPLAY FourLetterWord " " FourDigitNumber " " AlphaBytes.
```

The pairs of quotation marks in this DISPLAY statement each define a literal consisting of a single space. By including these literals, this DISPLAY statement produces the following output:

```
Nerd 0086 Ginger
```

In fact, you can put all sorts of things in between the displayed data. For example, you can use text to tag the fields. And by using more than one DISPLAY statement, you can put things on more than one line, as in the following example:

```
DISPLAY "Today's word is " FourLetterWord.
DISPLAY "Today's number is " FourDigitNumber.
DISPLAY "Today's flavor is " AlphaBytes.
```

These three DISPLAY statements print the following output:

```
Today's word is Nerd
Today's number is 0086
Today's flavor is Ginger
```

You can put as many fields, records, and literals as you like in a single DISPLAY statement. You can simply separate those things with spaces, or you can separate them with commas, as you prefer. Each field in the list is converted — if a conversion is necessary — to a displayable format. This conversion applies only to fields, not to records. That is, the DISPLAY of an entire record never causes the conversion of any of the record's fields — the entire record is displayed as if everything in it were already in character format.

Formatting numbers for output

If your numbers are important to you (and why are you reading this section if numbers are not your life?), and if you want your numbers always to show up looking their best, MOVE them into numeric-edited fields before putting them on DISPLAY. A numeric-edited field washes their ears and shines their shoes before putting them into the lineup. (I describe numeric-edited fields in detail in Chapter 5.)

Displaying characters is seldom a problem — just stick the field name on a DISPLAY statement and out go the characters from the field to the screen. Numbers, however, can cause some odd things to happen. For example, the letter V in a PICTURE clause implies a decimal point, but doesn't include a displayable character position for it. This example shows some of the things that can happen when you decide to DISPLAY some numbers:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DisplayNumbers.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  CaseOne PIC 9(6) VALUE 22.
77  CaseTwo PIC ZZZZ9.
77  CaseThree PIC S9(4)V9(2) VALUE 82.13.
77  CaseFour PIC S9(4)V9(2) VALUE -82.14.
77  CaseFive PIC -9(4).9(2) VALUE "82.15".
77  CaseSix PIC -9(4).9(2) VALUE "-82.16".
77  CaseSeven PIC 9(5) USAGE BINARY VALUE 123.
77  CaseEight PIC 9(5) USAGE COMP VALUE 123.
PROCEDURE DIVISION.
Begin.
    DISPLAY "CaseOne: " CaseOne.
    MOVE 22 to CaseTwo.
    DISPLAY "CaseTwo: " CaseTwo.
    DISPLAY "CaseThree: " CaseThree.
    DISPLAY "CaseFour: " CaseFour.
    DISPLAY "CaseFive: " CaseFive.
    DISPLAY "CaseSix: " CaseSix.
    DISPLAY "CaseSeven: " CaseSeven.
    DISPLAY "CaseEight: " CaseEight.
    STOP RUN.
```

This example declares some fields that hold some numbers in different forms, and then displays each one of them. Some work just fine, but others hold surprises. The output looks like this:

```
CaseOne: 000022
CaseTwo:    22
CaseThree: 008213
CaseFour: 00821M
CaseFive: 82.15
CaseSix: -82.16
CaseSeven: 00123
CaseEight: 00123
```

Here is a description of the numbers that this example displays:

- `CaseOne` shows how a number is displayed when it is just a plain-vanilla `PIC 9` kind of thing. It defaults to all digit characters that are `USAGE DISPLAY`. If you like leading zeroes on your numbers, you've got it made.

- CaseTwo is a numeric-edited field — these characters are always showing off by cleanly suppressing the leading zeroes. The program uses a MOVE statement to put the value into the field because a VALUE clause won't convert numeric data into a numeric-edited field.
- CaseThree is a real problem. This field has a decimal point in there somewhere, but the DISPLAY keeps it a secret because V doesn't set aside a character space for it.
- CaseFour is the same as CaseThree, only a bit worse. The decimal point is missing and a letter M appears where the digit 4 should be — this is part of that internal format stuff with numbers, which I describe in Chapter 5.
- The next two, CaseFive and CaseSix, look pretty good because the editing characters add some clarity, and a bit of pizzazz.
- CaseSeven and CaseEight are sort of special. These two are converted by the DISPLAY statement from some internal representation to a string of characters. No standard way is defined for this conversion to happen, but every COBOL compiler comes up with something — the one used to generate this example formats the numerals with stunningly attractive leading zeroes.

If you want the real skinny on exactly why numbers behave this way, check out Chapter 5.

Lining up multiple DISPLAY statements

A DISPLAY statement, unless told otherwise, ends the current output line and starts a new one. This format is not always convenient; sometimes, you want to start the DISPLAY of a line from one place and continue it from another place. By using the ADVANCING clause, you can break DISPLAY of this one-line-at-a-time habit and use multiple DISPLAY statements to build up one line of output text. You don't need to do this very often, but it can come in handy. Here is an example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SDRAWKCAB.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  BrokenRecord.
    05  FrontWord PIC X(9).
    05  FILLER REDEFINES FrontWord OCCURS 9 TIMES
        10  OneCharacter PIC X.
77  i USAGE INDEX.
PROCEDURE DIVISION.
```

```
Begin.
    MOVE "backwards" TO FrontWord.
    DISPLAY FrontWord.
    PERFORM VARYING i FROM 9 BY -1
            UNTIL i IS LESS THAN 1
        DISPLAY OneCharacter(i) WITH NO ADVANCING
    END-PERFORM.
    DISPLAY "".
    STOP RUN.
```

The preceding code displays `backwards` backwards — as `sdrawkcab`. The output looks like this:

```
backwards
sdrawkcab
```

The first line of output (the one with `backwards` frontwards) comes from the first `DISPLAY` statement in the program. To produce the second line of output, the program runs through the second `DISPLAY` statement nine times (once for each letter in the word). The index `i` is set to the last character in the word and then reduced by one for each pass through the loop.

The second `DISPLAY` statement uses the `NO ADVANCING` option, which prevents `DISPLAY` from starting a new line after it sends a character to the screen. This is fine, except that nothing tells `DISPLAY` when the last character has been written and a new line would be okay. That's what the last little `DISPLAY` statement does. By trying to display a zero-length character literal, that statement manages to `DISPLAY` nothing at all — but it does act like a regular `DISPLAY` statement and causes the line to come to an end.

Some notes about quotes

A time will come when you want to display quotes around something you display. COBOL provides you with a couple of ways of displaying quotes. For example, assume that you want to display the following line of text:

```
He said, "Fooey!"
```

When you write the `DISPLAY` statement, just double up the quotation marks that you want to display, like this:

```
DISPLAY "He said, ""Fooey!""".
```

COBOL converts the double quotation marks into a single quotation mark. If you don't like the looks of that statement, you can accomplish the same thing in another way. You can use the QUOTE figurative literal, as in the following example:

```
DISPLAY "He said, " QUOTE "Fooey!" QUOTE.
```

This way, as the DISPLAY statement pastes the pieces together to build a string of characters to display, it pastes in a couple of quote characters for you.

This double-quote technique can be a bit confusing. You can get your eyes crossed trying to keep track of double quotes and double double quotes and the unquoted QUOTE. If at first you don't succeed, edit and try again. By the way, Chapter 6 has more information on nonumeric literals, and how the quote works.

Reading Data with ACCEPT

The ACCEPT verb reads data. It has magical powers and can read from the tips of your fingers (well, okay, the keyboard). Also, ACCEPT reads the predefined date and time registers that are built into COBOL.

Reading keyboard entries with ACCEPT

A hush falls over your program, as the ACCEPT verb takes control of your entire system:

```
01  InComing PIC X(40).
    . . .
    ACCEPT InComing.
```

ACCEPT listens intently to the keyboard, waiting for a keystroke — any keystroke. After a period of time, a key is pressed. Then another. And another. The ACCEPT statement makes a record of every keystroke entered until finally, almost unexpectedly, the Return key is pressed. The ACCEPT verb immediately stops listening to the keyboard, takes all the characters it has gathered up, and puts them into the InComing field. We have data entry!

Normally, when you ACCEPT input from the keyboard, it is polite to DISPLAY something on the screen that lets the human know what is expected. Here's an example:

```
01  BirdCondition PIC X(40).
01  BirdFeathers PIC 9(4).
    . . .
    DISPLAY "How's your bird? "
        WITH NO ADVANCING.
    ACCEPT BirdCondition
    DISPLAY "How many feathers does it have? "
        WITH NO ADVANCING.
    ACCEPT BirdFeathers.
```

The two DISPLAY statements each put a prompt on the screen, and because they use the NO ADVANCING option, the cursor remains at the right end of the prompt, waiting for keyboard activity. The only thing left to do is type something, and then press Return. When you run this program and you have a dirty bird, here is what shows up on the screen:

```
How's your bird? dirty
How many feathers does it have? _
```

Your answer to the first question remains in place. When you press the Return key after answering the first question, the screen scrolls up and the second question appears. While you are off counting feathers, the ACCEPT statement politely and patiently waits for you to enter a number.

Getting the date and time with ACCEPT

You can use four special names — names that are predefined by COBOL — with ACCEPT to retrieve the current date and time from the computer running your COBOL program.

You shouldn't use the capability of ACCEPT that I describe in this section — even though it is a part of the COBOL language. This capability is a part of the millennium problem that's built right into standard COBOL. The exact format of the dates and times are a part of the COBOL standard. No standard COBOL way exists to determine the actual year — all you can do is use ACCEPT to get the last two digits of the current year. But don't despair. Take a look at the documentation of the compiler you are using and you will probably find an alternative method for acquiring the current date. If not, I present a solution in Chapter 17.

Here is an example program showing how these special names work:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. WhatTimeIsIt.
DATA DIVISION.
```

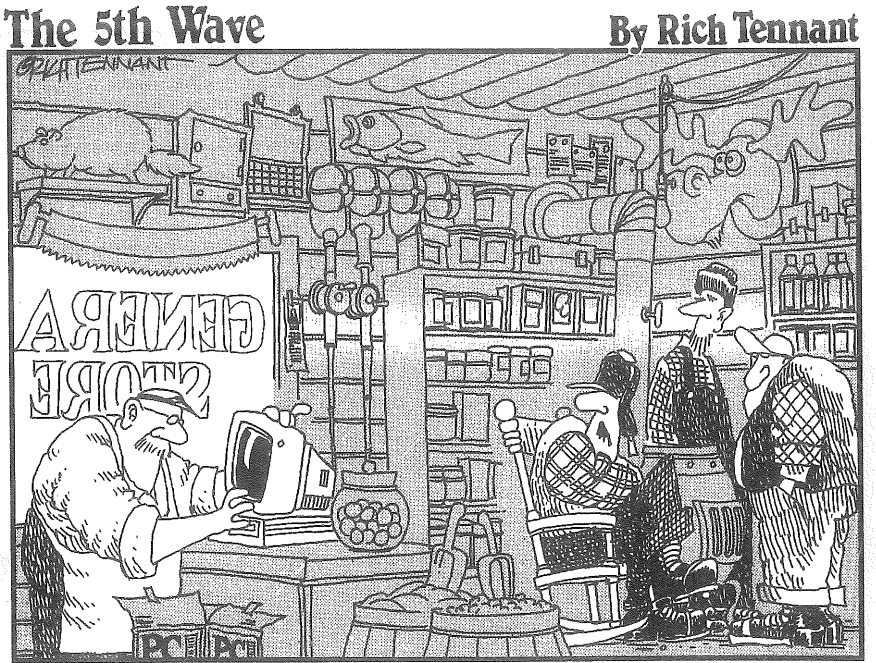
```
WORKING-STORAGE SECTION.
01  SystemDate.
    02  YY PIC 99.
    02  MM PIC 99.
    02  DD PIC 99.
01  SystemDay.
    02  Year PIC 99.
    02  DayOfYear PIC 999.
77  DayOfWeek PIC 9.
01  SystemTime.
    02  HH PIC 99.
    02  MM PIC 99.
    02  SS PIC 99.
    02  Hundredths PIC 99.
PROCEDURE DIVISION.
Begin.
    ACCEPT SystemDate FROM DATE.
    DISPLAY SystemDate.
    ACCEPT SystemDay FROM DAY.
    DISPLAY SystemDay.
    ACCEPT DayOfWeek FROM DAY-OF-WEEK.
    DISPLAY DayOfWeek.
    ACCEPT SystemTime FROM TIME.
    DISPLAY SystemTime.
    STOP RUN.
```

The program uses ACCEPT statements to obtain the current date and time. When you run the preceding program, the DISPLAY statements produce the following output:

```
970720
97201
1
10204354
```

The first line of the output shows the date in the form YYMMDD. The second line shows the date in the form YYDDD, where DDD is the day-number of the year. The third line lists the day of the week, with Monday as 1, Tuesday as 2, and so on. The last line shows the time in a 24-hour format, with hours, minutes, seconds, and hundredths of seconds. In the example, each of these numbers is read into a record that allows access to the individual fields.

Part IV
Input, Output, and Sorting



In this part . . .

COBOL has three basic types of files: sequential, relative, and indexed. A *sequential file* contains a series of records that can only be read or written sequentially — that is, one right after the other. A *relative file* contains a series of records that you can access by their serial numbers — that is, by their relative positions in the file. An *indexed file* contains a series of records that can be located by a *key value* — that is, by the value of one of the fields that each record contains. With a separate chapter for each of COBOL's three basic file types, the first three chapters in this part of the book show you how to write data to a file and how to read data from a file.

COBOL has a built-in sorting facility that you can use to sort data records from one file into another, from your program into a file, from a file into your program, or from one part of your program to another. You can also call on COBOL's merge facility to combine two files that are already sorted. The final chapter in this part of the book describes how to sort data that resides either in a file or inside your program.

Chapter 13

Working with Sequential Input and Output

In This Chapter

- Defining a sequential file
- Discovering how to OPEN and CLOSE a sequential file
- Understanding how to READ and WRITE a sequential file
- Figuring out how to REWRITE a sequential file

You can think of a sequential file as a reel of tape. It can only be written and read front to back or back to front. In fact, sequential files were originally designed for storing data on reels of tape. Many of the COBOL verbs and statements that you encounter (such as REWIND, REEL, and REMOVAL) are relics of the days when all data was stored sequentially on reels of tape.

A sequential file holds records in a fixed order. The order in which you write records to the file is the order in which you can read from the file. The records are kept in the file one right behind the other. Each time you write a new record to a sequential file, COBOL appends the new record to the end of the file, just as an additional song that you record on an audio tape goes onto the tape after the most recently recorded song. To read from a file, you start at the beginning and read the first record, and then the next, and the next, and so on, until you have either read them all or you are just sick and tired of reading and decide to quit.

You can use a sequential file to keep data on disk if you really don't care about the order in which you keep the data. Sequential disk files are especially good for holding temporary working data. Although a sequential disk file does a fine job of holding a large number of records — this happens quite often, actually — it is particularly good at storing data that has just a few records. If you need to write just a few pieces of information to a disk for later use, a sequential disk file is your huckleberry.

Because of the file organization — one record right behind another — you cannot delete a record in a sequential file. You can read records and write new records to the end of the file. You can even write data into the middle of a file by writing new data on top of the old data, but you cannot delete a record.

Sequential access, also called sequential I-O, can be used for different types of devices. It can be used for `INPUT` to read from devices such as tape drives, disks, and CD-ROMs. It can be used for `OUTPUT` to write to devices such as tape drives, disks, and printers.

One of the most common uses of sequential output is for printing reports. In fact, some folks say that printing reports is what COBOL was born to do. At any rate, special capabilities are built into sequential writing that make printing work quite well. I don't cover these printer doodads in this chapter — if it's printing you're after, check out Bonus Appendix B on the CD that accompanies this book.

Defining a Sequential File

In the following sections, I describe a step-by-step process that you can follow to write the code that defines a sequential file. Some of the steps are required and some are not. If you go through these steps and put in the things that are required and select what you want from the things that are optional, you can set up your sequential file exactly the way you want.

Before you can read or write a sequential file — in fact, before you can even open a sequential file — you have to define the file. Right there in your program, you have to specify everything anybody would ever want to know about that file. And you have to tell COBOL how you want to read or write the file.

In the `ENVIRONMENT DIVISION`, you have to put file names and access methods in the `INPUT-OUTPUT SECTION` for both `FILE-CONTROL` and `I-O CONTROL`. On top of that, you have to go to the `FILE SECTION` of the `DATA DIVISION` and come up with an `FD` statement specifying the things like the record size and what sort of label you want on the file, as well as the record layout itself. And then you have to set options — this thing has more options than a kid with a water pistol at a garden party.

Step 1: SELECT an access method and names

The first step in the process of defining a sequential file is to specify the name of the file you want to use. You complete this required step by putting a SELECT statement in the FILE-CONTROL paragraph of your program. ACCESS MODE is not required, but it is almost always included. Along with the ACCESS MODE, here is the minimum form of a SELECT:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT RefName
        ASSIGN TO "RealName"
        ACCESS MODE IS SEQUENTIAL.
```

This statement defines *RefName* as the name used inside the program to refer to the file. It also specifies *RealName,* the name of the physical file. You need to supply your own *RefName* and *RealName*.

The ACCESS MODE defines how COBOL reads and writes files, not how the file itself is organized. The ACCESS MODE specification is optional; if you don't specify an ACCESS MODE, it defaults to SEQUENTIAL, but not everybody knows that and somebody may need to read your program some day. Be nice.

The file itself does not have to be organized as a sequential file for you to access it in a sequential way. The physical file can be RELATIVE or INDEXED (I describe these file organizations in the next two chapters) and still be accessed as if it were sequential.

Step 2: Specify the ORGANIZATION

As I discuss a bit later in this chapter, the COBOL OPEN verb creates a new file. You need to define the ORGANIZATION of a file that you are going to create. Choose wisely, though — after you specify the ORGANIZATION and create the file, you can never change the ORGANIZATION. And, yes, if you don't specify the ORGANIZATION, it automatically defaults to SEQUENTIAL, but why not say what you mean?

The following code shows how to use the ORGANIZATION statement to define the physical structure of a sequential file:

```
    SELECT RefName
        ASSIGN TO "RealName"
        ORGANIZATION IS SEQUENTIAL
        ACCESS MODE IS SEQUENTIAL.
```

The last two lines of the preceding code may look a bit redundant, but they really aren't. The ORGANIZATION has to do with the physical structure of the file and the ACCESS MODE has to do with how the program reads and writes the file.

Step 3: SELECT an OPTIONAL file

It only makes sense that the keyword OPTIONAL would itself be optional. You may want to be able to open a nonexistent file for I-O, or even for INPUT. That sounds a little silly at first, but sometimes you really need to do it. For some insight into the reasons why you may want to open a nonexistent file, see the "When is a file not a file?" sidebar. Later in this chapter, I also provide some sample code showing how and why you would want to open a nonexistent file for I-O.

Briefly, here's the situation: Normally, you want to open a file, read it to the end, and then quit. If you declare the file as OPTIONAL and it doesn't exist, everything works fine, and you get an end-of-file notification the first time you read it. Your program runs the way you intend for it to run (by doing the end-of-file stuff). However, if the file does not exist and you have not declared it as OPTIONAL, the OPEN statement detects and reports an error.

The word OPTIONAL comes right after the word SELECT, like this:

```
SELECT OPTIONAL RefName
    ASSIGN TO "RealName"
    ACCESS MODE IS SEQUENTIAL.
```

This statement is standard COBOL, but some compilers may not support it. The only way to find out whether yours does is . . . well, you know, type it in and see what happens.

Step 4: RESERVE some extra space

You can optionally allocate some extra space in the computer's memory to hold input and output records. A program can access data from the computer's internal memory much faster than from a file on disk or tape. By setting aside some space in memory for holding input and output records, you can improve your program's efficiency. The following example sets aside enough space to hold 15 records:

When is a file not a file?

Answer: When it is OPTIONAL.

You can successfully command COBOL to OPEN a file that does not exist. Not only that, but you can then use the READ verb on the open, but nonexistent, file. If this capability seems a bit odd to you, consider yourself normal. When you take a closer look at why it works this way, however, it starts to make some sense.

Only one way exists to create a file in COBOL: OPEN it. You simply open a file for OUTPUT and, wham, a brand-new, and very empty, file comes into existence on the disk. If a file of the same name already existed, it's history. And any data that was already in the file is history right along with the old file. This may sound brutal, but it's a file-eat-file world out there on that disk.

If you OPEN a file for INPUT, you can't WRITE to it — you can only READ from it. Opening a file for INPUT never creates a file. Never. But this can be an inconvenience.

Imagine this situation: You have a program that opens five files for INPUT and does stuff with the data it reads from those files. The number of records of data varies from day to day — one day, the program may process 1,000 records, and the next day it processes 5,000 records. On some days, one or more of the files doesn't even exist — that is, the program may need to open only three files instead of five. But the program contains the COBOL code to open all five. That's where the OPTIONAL part comes into play.

If you don't declare any of the files as OPTIONAL, the program has no option but to open and process data from all five. A missing file causes an error. If you declare the files as OPTIONAL, the program can still open them all, but if any files are missing, the program won't even know. Or care. A missing optional file acts exactly like a file that exists but doesn't contain any data. Whenever your program reads from the missing OPTIONAL file, the program reacts exactly as if it had already read some data and has now come to the end of the file.

```
SELECT RefName
       ASSIGN TO "RealName"
       ACCESS MODE IS SEQUENTIAL
       RESERVE 15 AREAS.
```

The keyword AREA is optional; it can be just left off, or it can also be spelled AREAS.

Take two buffers and call me in the morning

The nerd term for holding stuff in memory is *buffering*. The word buffering has nothing to do with taking aspirin or polishing fingernails — it has to do with stashing records in memory that are about to be written or have just been read.

Sometimes it is more efficient (at the hardware level) to read or write a whole block of records at once. Take the case in which you are writing lots of records and you have a very small record size. Each time you do a WRITE, the output record is actually just moved into this specially reserved storage space (called a buffer), but it is not written to disk. Then, after you have written a bunch of records, and the buffers are all filled up, COBOL writes the whole wad of them to the tape or disk in one swipe, which is much more efficient.

With most modern operating systems, a capability like RESERVE AREAS doesn't really mean much because this buffering is done automatically whether or not you have asked COBOL to do it. You may have little or no say in the matter. Oh, well. So much for programmers' rights.

Step 5: Set the character used for padding

It is possible that records written to disk don't evenly fill the space allocated for the file. The leftover area on disk is filled with the padding character. For example, if the smallest chunk of data that you can physically write to your disk drive is 256 characters, and you write a record that is 150 characters long, 106 character spaces are filled with the padding character. Not all COBOL compilers implement this option.

You can specify the character to be used to fill up any space that you don't use. Whenever you read a record into a place that has some extra room — that is, your program's record size is larger than the file's record size — the READ fills the extra space with whatever character you have named. You can use any character you would like.

Here's an example that sets the letter Q as the padding character:

```
SELECT RefName
    ASSIGN TO "RealName"
    ACCESS MODE IS SEQUENTIAL
    PADDING CHARACTER IS "Q".
```

If you don't set the padding character, your COBOL compiler picks whatever it wants to.

One interesting side note here: If you have a record completely filled with the specified padding character, it will be ignored by the READ verb. Hmm. That means the record *can* be deleted from the middle of a sequential file, but you have to be really devious to do it. Because you can't delete a record from the middle of a sequential file, you can trick COBOL into ignoring a record by filling it with padding characters. This is by no means a recommended practice, but you may come across some old program that uses it. If you decide to try something like this, you need to experiment by using REWRITE, which I describe at the end of this chapter.

Step 6: Define the record delimiter

You can specify the record delimiter — the stuff that goes between the records in the file — but this step is definitely optional. By defining the record delimiter for a sequential file, you specify markers to be written in between the records as you write to disk or tape. Although this option is really an artifact left over from the days of files on tape that required delimiters to mark the beginning and ending of sequentially stored records, it can still have its uses.

Here's the deal: You plan to write variable-length records and a programmer on some other computer system needs to be able to read the file and figure out the length of each record. For example, you could be writing data to a file so it can be read with a C program. Or maybe you are writing data to a tape that will be read by a Studebaker 1900 tape drive on a WunMug computer. Basically, you want to make a chalk mark at the end of each record in such a way that anything can find it.

Fortunately, COBOL gives you a standard way of defining the record delimiter. Here is what you do to read or write to a standard tape:

```
SELECT RefName
    ASSIGN TO "RealName"
    ACCESS MODE IS SEQUENTIAL
    RECORD DELIMITER IS STANDARD-1.
```

The STANDARD-1 delimiter causes the file written to the tape to use the format defined as the ANSI standard X3.27-1978. But forget all that. If somebody says, "Can you produce a standard tape for me?" you just grin and say, "Sure." Then you just stick in the DELIMITER statement and go. One important note: If you use STANDARD-1, it *must* be a tape — this one won't work on a disk file. But that's okay, because you don't need delimiters on disk files.

Now I need to throw a little grit in the gears. Each COBOL compiler that implements this capability has a secret handshake all its own. In place of the STANDARD-1 in the preceding example, each compiler has to make up its own names for any different kinds of delimiters it provides. Hey, don't look at me, I didn't make that decision. To find out what is available, you have to look at the documentation for your compiler. Then comes the interesting part: Haul out your documentation and explain the format of the tape delimiters to the programmer who wants to read the file in C.

Step 7: Create a place to stick the file status

If you are one of those people who likes to know more about the status of a file than its own mother, I've got just the thing for you. Your program can include a two-digit *sequential I-O status*. This is a value that changes every time any operation whatsoever takes place on the file. Using this value, your program can determine whether any file operation succeeded or failed — and if it failed, why it failed. Here's how you add the file status to your definition of the sequential file:

```
SELECT RefName
    ASSIGN TO "RealName"
    ACCESS MODE IS SEQUENTIAL
    FILE STATUS IS RefFileStatus.
```

Somewhere in WORKING-STORAGE, you have to define a place to hold the status. (I describe WORKING-STORAGE in Chapter 4.) You can make it part of a record or a standalone 77 level, like this:

```
77  RefFileStatus PIC 99.
```

The COBOL program updates the variable in this location every time your program performs (or attempts to perform) any activity on your file. It tracks the status of the result of every file OPEN, CLOSE, READ, WRITE, and REWRITE. Table 13-1 lists the possible values that can wind up here as the status of a file. Any value less than 30 is okay, but anything 30 or over is not good. Within your program, you can display the values to the screen or process them in any way you see fit.

Your COBOL compiler may have a special file-handling subsystem that does not use these codes. It can be that your compiler operates with some special file system that has its own set of codes and some special way of handling file error conditions.

Table 13-1 The Possible Values of Sequential I-O Status

Value	*Meaning*
00	Whatever you did last worked. In fact, it worked so well that no further comment is necessary.
04	The READ worked, except that the record was either a little longer or a little shorter than was expected.
05	The OPEN worked, but the file doesn't exist. It's okay, though, because you have the file declared as OPTIONAL. If the OPEN was for I-O or EXTEND, the file was created — otherwise, it still doesn't exist.
07	If the last thing you did was a READ, WRITE, or REWRITE statement, it worked. If it was an OPEN or CLOSE, it worked but the read or write operation was on a disk file and you included some kind of tape-drive option (NO REWIND, REEL/UNIT, or FOR REMOVAL). Careless, maybe, but no harm done.
10	The READ failed. But it's no big deal — either the READ was at the end of the file or it involved an optional file that doesn't exist. Either way, there's no data to be had.
30	A permanent error exists and you should just give up. No way exists for you to do anything with this file. COBOL has no way to determine the cause — it can be something to do with the operating system.
34	You just attempted to WRITE beyond the boundary limits of the file. You're out of space. "No room, no room," shouted the Mad Hatter.
35	You tried to OPEN a file for INPUT, I-O, or EXTEND, and the file doesn't exist. You can't do that unless you declare the file as OPTIONAL.
37	You can't OPEN that file. It could be that you tried to open a read-only file for EXTEND, OUTPUT, or I-O. It could be that you tried to open for I-O a file that just can't do that (some files can be opened for reading or for writing, but not both at the same time). It could be that you tried to read a file that doesn't allow reading — you know, an attempt to breach security. Maybe it's just you — have you had your clearance checked lately?
38	You tried to OPEN a file that was locked by a CLOSE WITH LOCK statement.

(continued)

Table 13-1 *(continued)*

Value	*Meaning*
39	You tried to OPEN a file that has a completely different set of attributes than the one you have defined in your program. Guess again.
41	You tried to OPEN a file that's already open.
42	You tried to CLOSE a file that wasn't open.
43	You surprised the file when it wasn't ready. You have to do a READ right before you do a REWRITE.
44	The record you are trying to REWRITE has a size problem. It is certainly not the same size as the one you just read — in fact, it can be larger than the largest record allowed or smaller than the smallest allowed. Where did you get that thing?
46	You just had to try it again, didn't you! The previous READ statement failed, and so did this one. Unless you change your ways, so will the next one. One of the READ statements back there hit the end of file or something.
47	You can't READ from a file that you have OPEN for OUTPUT or EXTEND.
48	You can't WRITE to a file that you have OPEN only for reading.
49	You can't REWRITE a record to a file that you did not OPEN in I-O mode.
9x	Any error message in the 90s is one that is peculiar to your compiler.

Step 8: Add an I-O CONTROL paragraph

You can put an I-O CONTROL paragraph in the INPUT-OUTPUT SECTION (this step is optional):

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
I-O CONTROL.
```

You can just include the heading and put nothing into the I-O CONTROL paragraph, because everything you can put in it is optional. The following section in this chapter describes an optional clause you can add to your I-O CONTROL paragraph.

If you are reading through some old code, you may find entries in the I-O CONTROL paragraph for RERUN and MULTIPLE FILE TAPE. Both of these entries are considered obsolete — they have to do with some really weird ways of handling multiple files on tape. If you are riding out across the plains on an old program that has some of this stuff in it, and the program breaks a leg, and the documentation for your compiler doesn't give you the information you need, shoot the program. It may be time to rewrite.

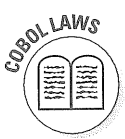
Step 9: Add the SAME clause

The SAME clause in the I-O CONTROL paragraph has some limited usefulness. I say "limited" because RAM is not as precious as it once was, and this clause is a space-saving device. Unless you are a real efficiency freak, skip to the next step. In fact, you may not be able to use this clause even if you want to, because many COBOL compilers don't even have it.

You can use the SAME statement to have your file operations share some RAM. Every file you define in your program has its own workspace set aside where it performs the mechanics of input and output, and it also has a record used to hold the input and output data. With the SAME statement, you can have your files share one or both of these areas:

```
I-O CONTROL.
    SAME AREA FOR ThisFile ThatFile.
    SAME RECORD AREA FOR RefName UmpName BooName.
```

The first sentence causes the two files ThisFile and ThatFile to share internal work areas for the mechanics of performing input and output. The second sentence (the one with RECORD in its middle) causes the three files RefName, UmpName, and BooName to share the same data record locations for reading and writing. In both cases, the filenames are the ones defined on the SELECT statement.



Keep the following laws in mind when you want to use a SAME statement:

- Your program can have multiple SAME AREA statements, but a file can only be named in one of them.
- You can only OPEN the files listed on a SAME AREA statement one at a time.
- You can OPEN the files listed on a SAME RECORD AREA statement all at once, but be aware that the same data location is shared among all of them.
- The files named on either type of SAME statement don't have to be of the same ORGANIZATION or ACCESS.

A SAME RECORD AREA statement can have the same bunch of files that you have in a SAME AREA statement — they share record layouts and work areas in a sort of ultimate giving-and-sharing, hippie-like environment. Heavy.

Step 10: Describe the structure in the FILE SECTION

In Step 2, you define the general organization of the file as being SEQUENTIAL. In this step, the FILE SECTION holds the detailed information describing the structure of the file itself. This detailed description is in two parts: the *file description* and the *record description*.

The file description is affectionately known as the "eff-dee" statement because its keyword is FD. The record description has no defining keyword — it is defined as an 01 level. Well, I suppose you could say it's called an "oh-one," but that would confuse it with the "oh-ones" of WORKING-STORAGE.

The record description comes right after the FD, like this:

```
DATA DIVISION.
FILE SECTION.
FD RefName . . .
01  RefFileDataRecord . . .
```

The name on the FD is the same one that appears on the SELECT statement. Several options exist for FD, and I describe them in the following steps. By the way, you can have a whole bunch of the 01 record descriptions, and they don't even have to be all the same size. I know this sounds spooky, but I clear it up in the next few steps.

Step 11: Define the RECORD size

You can define the record size for your sequential file with the optional RECORD statement. This statement is optional because the record description following the FD statement determines the record size. However, COBOL programs normally include a RECORD statement, if for nothing other than documentation.

The simplest form of a RECORD statement is for a file that has all fixed-size records, like this example:

```
FD  RefName
    RECORD CONTAINS 84 CHARACTERS.
```

No matter what else happens, every record in this file has exactly 84 characters. If you write something too small, the PADDING characters fill out the record. If you write something too big, COBOL simply clips the tail off the data and shoves what's left into the file. It may be brutal, but that's life — it's survival of the data that fits.

On the other hand, if you want to control this size thing yourself (if you feel that you need to take control of your own sizing destiny), or if you need to allow for variable-length records, you just have to let COBOL know the maximum and minimum sizes, as in the following example:

```
FD  RefName
    RECORD CONTAINS 16 TO 96 CHARACTERS.
```

With this definition, you can read or write a record as small as 16 characters and as large as 96 characters. Heady stuff. It is up to you to determine the size of every record. In many cases, you can easily add up the size of each field to determine the size of a record, but in some other cases, this technique doesn't work. Chapter 18 includes a program that you can use to determine the size of a record.

But wait, there's more. For you complete control freaks (for those of you who want to micromanage the input and output), take a look at this baby:

```
FD  RefName
    RECORD IS VARYING IN SIZE FROM 16 TO 96 CHARACTERS
        DEPENDING ON RefSizeValue.
```

The preceding definition not only allows you to read and write records that vary in size from 16 to 96 characters, but it also lets you dictate the exact size of every individual READ and WRITE. All you have to do is stuff some number into a variable — in this example, it is RefSizeValue. As an added bonus, whenever you READ a record from a file, and the DEPENDING variable has been defined, the DEPENDING variable holds the number of characters actually read. This capability allows you to monitor the exact size of each and every READ.

I suppose it is only fair to tell you that COBOL gives you another way to specify the minimum and maximum sizes. All you have to do is declare record definitions of different sizes and, if you don't mention the sizes anywhere, COBOL sets the minimum and maximum sizes to the smallest and largest of the records you define. For example, you can do this:

```
FD  RefName.
01  Record1 PIC X(16).
01  Record2 PIC X(96).
```

In the preceding example, and in the one that follows, it is exactly as if you had defined a RECORD IS VARYING clause with the size ranging from 16 to 96. You can even do this while using the controlling variable. Here's how:

```
FD  RefName
    RECORD IS VARYING DEPENDING ON RefSizeValue.
01  Record1 PIC X(16).
01  Record2 PIC X(96).
```

Step 12: Specify the BLOCK size

You can optionally specify the number of records that will be written or read with each output or input operation. This setting is an efficiency concern — it never has any effect on how you write your program or what it does. Some advantage may exist in physically reading or writing a bunch of records at once. If so, you can specify how many records COBOL writes or reads as a block, as in this example:

```
FD  RefName
    BLOCK CONTAINS 20 RECORDS.
```

With the preceding statement, as you WRITE, COBOL tries to stash 20 records in memory before it does a physical write. Every time you do a READ, COBOL tries to grab 20 records at once. It doesn't give you all 20 — it just gives you one and sticks the other 19 in memory somewhere so the next time you ask for a record it already has one.

You don't have to limit yourself to a number of records. If you happen to know that your computer just loves to have its files written in blocks of 4,096 characters at one time, you can make it really happy this way:

```
FD  RefName
    BLOCK CONTAINS 4096 CHARACTERS.
```

If you don't include a BLOCK statement, the compiler makes the decision about the block size. Some compilers take the statement BLOCK CONTAINS 0 CHARACTERS to be the same as if no BLOCK statement were specified.

To make your BLOCK definitions work right, you need to study your system. Exactly what your compiler does about this BLOCK stuff can be a little mysterious. It may be that the compiler completely ignores any BLOCK statement, and the BLOCK statement makes no difference at all. On the other hand, its effects may be dramatic. You may find that with the addition of a BLOCK statement, a program that took four hours to run now scoots through the system in just a few minutes. If you have some speed problems, this is a

good thing to try. You need to do some research on the file-blocking characteristics of your computer before you can do something — shooting in the dark usually won't work.

Step 13: Define the LABEL RECORDS

A LABEL RECORDS statement is another one of those obsolete things. This statement can cause COBOL to write a special label record at the beginning of an output file and to expect to read a label from the beginning of an input file. These labels are something left over from the days when the libraries of magnetic tapes needed to have identifying labels. Don't put this statement in your code, but if you find it in some older code, you may want to leave it alone. It can mean that some files out there include label records that need to be handled when the program reads them.

The LABEL sentence has two forms. The keyword OMITTED indicates that the file simply has no label records:

```
FD  RefName
    LABEL RECORDS ARE OMITTED.
```

The other form specifies that the file has standard label records:

```
FD  RefName
    LABEL RECORDS ARE STANDARD.
```

Referring to label records as STANDARD is probably one of the most misleading things in COBOL. This statement means that the labels are standard for *this* compiler on *this* computer — but not necessarily for any other. In today's world, the word *standard* refers to a set of specifications that globally defines the architecture of a specific program, software, or hardware system. The old meaning of *standard* as used in COBOL is strictly proprietary — you can refer to a tape with STANDARD label as belonging to a particular brand of computer, such as "This is a standard tape for a Zephyr 4200."

Step 14: Create the DATA RECORDS clause

Hey. Do you feel like typing in some extra code? I mean, do you want to put in some COBOL code that doesn't do anything at all? You're in the right place. You have to look long and hard to find something as special as the DATA RECORDS clause — there is no way that anything except a committee could have produced anything this elegant. Not only does this clause not do anything, it has now been officially declared obsolete.

This clause just lists the names of the data definition records — the ones defined as 01 levels that follow this FD statement. Here's what it looks like:

```
FD  RefName
    DATA RECORDS RefData1 RefData2.
    01  RefData1 . . .
    01  RefData2 . . .
```

Yep. That's it. COBOL doesn't care whether you do this, but (and here's the really funny part) if for some reason you decide to go ahead and do this, and you get the list wrong, you get an error message.

Opening a Sequential File

Before your program can do anything with a file, the program must OPEN the file. Four ways exist to OPEN a sequential file: You can OPEN it for INPUT, OUTPUT, EXTEND, or I-O. The OPEN verb doesn't do any reading or writing of data — it just gets the file ready for action and notifies your program that the file is ready.

Opening a file for INPUT

If you want to read the file from front to back and then quit, you OPEN the file for INPUT. For the OPEN to succeed, the file must already exist or be declared as OPTIONAL. If it is OPTIONAL and does not exist, the first READ statement reports that the end of the file has been reached.

You can do a simple OPEN this way:

```
OPEN INPUT RefName.
```

The RefName is the name you gave to the file on the SELECT statement. The code contains no bells and whistles — just open the little rascal up and start reading. That is not to say that you can't do a couple of tricks. For one thing, you can open the file this way:

```
OPEN INPUT RefName NO REWIND.
```

This statement opens the file for reading without rewinding to the beginning of the file. This statement is primarily for a sequential device, such as a tape drive, that maintains some sort of physical position. Unless you specify NO REWIND, the tape automatically rewinds to its beginning when you OPEN the file. If you use NO REWIND on something that doesn't rewind, it's not an error, but nothing happens.

By using REVERSED, you can force a file to start at the end:

```
OPEN INPUT RefName REVERSED.
```

This statement causes the file to OPEN and position itself at the last record in the file. I realize that starting at the end could make the file seem incredibly short when you READ it, but REVERSED does something else, too. It causes each READ to move backward to the previous record, instead of the normal mode of moving forward to the next record. I can count all the reasons I'd want to read a file backwards on zero fingers. The capability to read a file from back to front could be one of those "really good ideas" that was adopted because nobody could think of a good reason to throw it out.

Opening a file for OUTPUT

If you just want to WRITE records to a file and have the file end at the point where you quit and CLOSE the file, you should OPEN the file for OUTPUT. If the file does not already exist, COBOL creates a new file. If the file does exist, OPEN empties the file of any data it may contain — in effect, it becomes a new file.

You can do a simple OPEN this way:

```
OPEN OUTPUT RefName.
```

That's all it takes. RefName is the name found on the SELECT statement.

The only option on this kind of OPEN has to do with tape:

```
OPEN OUTPUT RefName NO REWIND.
```

Unless you specify NO REWIND, the tape automatically rewinds to the beginning when you OPEN the file. NO REWIND effectively leaves intact the data already on the tape prior to the point at which you start writing — but to do this, you must have had some way to position the tape. For example, you can OPEN the tape for INPUT, READ a few records, CLOSE the tape, and immediately OPEN it again for OUTPUT with NO REWIND.

Opening a file for EXTEND

If you want to open a file and add some stuff to the end of the file without messing with any data that's already in there, use OPEN EXTEND. This statement is just about like OPEN OUTPUT, except for the positioning thing. Any data that's already in the file remains untouched. This mode has no options whatsoever.

Here is an example:

```
OPEN EXTEND RefName.
```

If the file does not exist, it must have been declared OPTIONAL for the OPEN to work. Also, if it is declared as OPTIONAL, a new file is created if it doesn't already exist.

Opening a file for I-O

If you want to read through a file and make changes to the records in it, use OPEN I-O. With this mode, you can READ a record, and if you don't like the data you find there, REWRITE the record to change it to whatever you would like. If the file doesn't exist when you OPEN it, COBOL creates it — regardless of whether you have declared the file as OPTIONAL. Here's how you OPEN a file for I-O:

```
OPEN I-O RefName.
```

Closing a Sequential File

The first thing you do to a file is to OPEN it; the last thing you do is to CLOSE it. For the most part, the statement you use to CLOSE a file is simple and straightforward:

```
CLOSE RefName.
```

After you CLOSE a file, you can't do any more reading and writing unless you OPEN it again. However, if you CLOSE the file with the following statement, you can't OPEN the file again:

```
CLOSE RefName WITH LOCK.
```

What this WITH LOCK clause means is, during only *this* run of only *this* program, you cannot OPEN the file again. Any other program can open it and, if you run this program again, you can open the file again. This clause is such a weird thing to have in the language that there must be a really good reason for it. There just has to be. Surely. I wish I could think of it.

With a disk file, you can just CLOSE the file and forget about it. The file will just lay there and spin and wait for you to come back to it. A tape, however, is a different beast. You can cause the system to issue a permission for the tape to be unloaded from the tape drive and put back in the closet (or wherever tapes are kept at your place) by closing it this way:

```
CLOSE RefName REEL FOR REMOVAL.
```

If you don't particularly care for the word REEL, you can use UNIT to do the same thing, like this:

```
CLOSE RefName UNIT FOR REMOVAL.
```

Normally, when you close a tape, it automatically rewinds. You can prevent the rewind by including the following COBOL statement:

```
CLOSE RefName WITH NO REWIND.
```

If things work right, the tape just stops wherever it is and waits for something else to happen. You can reopen it in some other mode, or you could just let it sit there until some other program opens it.

Writing to a Sequential File

Here's an example program that writes to a sequential file. If the file named phones doesn't exist, COBOL creates it when the program tries to OPEN it. If phones does exist, COBOL overwrites it with a new file.

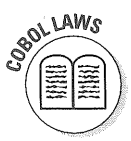
```
IDENTIFICATION DIVISION.
PROGRAM-ID. SequentialWrite.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT OPTIONAL PhoneList
        ASSIGN TO "phones"
        ACCESS MODE IS SEQUENTIAL
        ORGANIZATION IS SEQUENTIAL
        FILE STATUS IS PhoneFileStatus.
DATA DIVISION.
FILE SECTION.
FD PhoneList
    RECORD CONTAINS 30 CHARACTERS.
01  PhoneData.
    05  FirstName PIC X(10).
    05  LastName PIC X(10).
    05  AreaCode PIC 999.
    05  PhoneNumber PIC 9(7).
WORKING-STORAGE SECTION.
77  PhoneFileStatus PIC XX VALUE "00".
```

(continued)

(continued)

```
01  FredsPhone.
    05  FirstName PIC X(10) VALUE "Fred".
    05  LastName PIC X(10) VALUE "Anonymous".
    05  AreaCode PIC 999 VALUE 711.
    05  PhoneNumber PIC 9(7) VALUE  5559438.
PROCEDURE DIVISION.
Mainline.
    OPEN OUTPUT PhoneList.
    PERFORM WriteFourRecords.
    CLOSE PhoneList.
    STOP RUN.

WriteFourRecords.
    MOVE "Billy Bob" TO FirstName OF PhoneData.
    MOVE "Hamhock" TO LastName OF PhoneData.
    MOVE 817 TO AreaCode OF PhoneData.
    MOVE 5558016 TO PhoneNumber OF PhoneData.
    WRITE PhoneData.
    MOVE "Tim       Shane     2055550918" TO PhoneData.
    WRITE PhoneData.
    MOVE "Fanny     Brice     9725558381" TO PhoneData.
    WRITE PhoneData.
    WRITE PhoneData FROM FredsPhone.
```

In the preceding example, COBOL writes four records to the `PhoneData` file. `MOVE` statements put the information into the record associated with the `FD` and then the `WRITE` statements send the information to the file. As demonstrated by the last `WRITE` statement in the example, the `WRITE` statement can move the data by using the `FROM` option. The example in the next section of this chapter shows how to `READ` the records written by this example.



To `WRITE` to a file, you must `OPEN` it in `OUTPUT` or `EXTEND` mode. An old saying exists that goes along with COBOL's reading and writing: "Write records and read files." This maxim refers to the fact that the COBOL `WRITE` verb uses the name of a record as its argument, and the COBOL `READ` verb uses the filename as its argument. This makes some kind of sense because in both cases you specify the source of the data.

Reading from a Sequential File

The example in the preceding section of this chapter writes to a sequential file. The following example reads that file and displays the records:

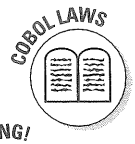
```
IDENTIFICATION DIVISION.
PROGRAM-ID. SequentialRead.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT PhoneList
        ASSIGN TO "phones"
        ACCESS MODE IS SEQUENTIAL
        FILE STATUS IS PhoneFileStatus.
DATA DIVISION.
FILE SECTION.
FD PhoneList
    RECORD CONTAINS 30 CHARACTERS.
01  PhoneData.
    05  FirstName PIC X(10).
    05  LastName PIC X(10).
    05  AreaCode PIC 999.
    05  PhoneNumber PIC 9(7).
WORKING-STORAGE SECTION.
01  FileFlag PIC X.
    88 EndOfFile VALUE "E".
77  RecordCount PIC 99 COMP.
77  PhoneFileStatus PIC XX VALUE "00".
PROCEDURE DIVISION.
Mainline.
    OPEN INPUT PhoneList.
    IF PhoneFileStatus IS NOT EQUAL TO "00"
       DISPLAY "Open failed: " PhoneFileStatus
       STOP RUN
    END-IF.
    PERFORM ReadAllRecords.
    CLOSE PhoneList.
    STOP RUN.

ReadAllRecords.
    MOVE SPACE TO FileFlag.
    PERFORM VARYING RecordCount FROM 1 BY 1
            UNTIL EndOfFile
        READ PhoneList
            AT END MOVE "E" TO FileFlag
            NOT AT END DISPLAY PhoneData
        END-READ
    END-PERFORM.
```

This example is a little more robust than the previous one: It checks the status code after it attempts to OPEN the file for INPUT. In this example, the file is not declared as OPTIONAL, so an error occurs on OPEN if the file doesn't exist. The displayed error looks like this:

```
Open failed: 35
```

By looking at this message, you immediately know that the file doesn't exist. The reason you immediately know this is because you took a peek at the error codes in Table 13-1. However, if the file does exist, the program reads it and displays this information:

```
Billy Bob Hamhock    8175558016
Tim       Shane      2055550918
Fanny     Brice      9725558381
Fred      Anonymous  7115559438
```



To READ a file, you must OPEN it in INPUT or I-O mode. The READ verb requires the name of the file, not the name of the record as the WRITE verb does.

Catch errors before they catch you

You really do need to write your COBOL code to check for error conditions on files. If you don't do anything in your program to check for errors, they go by unknown and unnoticed. To have a really robust program — one that runs right every time — you need to check the error condition resulting from every OPEN, CLOSE, READ, and WRITE. Every one.

But you won't. I know you won't. I know what will happen. You will ignore a possible error condition here and there just like I left some out of the examples in this chapter. Then, one day, your great masterwork program will crash unexpectedly without an error message. You will be visited by those-who-must-be-answered with questions like "Why?" and "What good are you?" You will lose your double-oh prefix, which licenses you to be a killer programmer. Your life will be in ruins. You will wind up all alone trudging barefoot through the snow. All because you left out one little IF statement that could have tested for the existence of an error code.

How do I know this is true? Would you like to see my frostbite? Ah, well, "Better to have programmed and crashed than never to have programmed at all."

You don't have to check for every possible error code value. In fact, it is just not practical to do so. You can check for simple success or failure, or you can check for certain ranges of error codes and decide whether they are acceptable or unacceptable. Looking at Table 13-1, you can see that certain error numbers can be considered okay — in particular, notice that all error codes less than 30 indicate some kind of success.

Rewriting a Sequential File

You can change data in the middle of a sequential file by using the REWRITE verb. To be able to use the REWRITE verb, you must OPEN the file for I-O. Then, starting from the beginning, you READ each record and use REWRITE to change any of them that strike you as needing a change. Just think of your records as a bunch of diapers — you've got to check them all to find the ones that need changing.

The following example uses the file created and displayed by the two previous examples. The change made is to the area code field — all phone numbers have their area codes changed to 972.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SequentialRewrite.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT PhoneList
        ASSIGN TO "phones"
        ACCESS MODE IS SEQUENTIAL
        FILE STATUS IS PhoneFileStatus.
DATA DIVISION.
FILE SECTION.
FD PhoneList
    RECORD CONTAINS 30 CHARACTERS.
01  PhoneData.
    05  FirstName PIC X(10).
    05  LastName PIC X(10).
    05  AreaCode PIC 999.
    05  PhoneNumber PIC 9(7).
WORKING-STORAGE SECTION.
01  FileFlag PIC X.
    88 EndOfFile VALUE "E".
77  RecordCount PIC 99 COMP.
77  PhoneFileStatus PIC XX VALUE "00".
PROCEDURE DIVISION.
Mainline.
    OPEN I-O PhoneList.
    IF PhoneFileStatus IS NOT EQUAL TO "00"
       DISPLAY "Open failed: " PhoneFileStatus
       STOP RUN
    END-IF.
    PERFORM ReadAllRecords.
```

(continued)

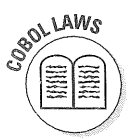
(continued)

```
    CLOSE PhoneList.
    STOP RUN.

ReadAllRecords.
    MOVE SPACE TO FileFlag.
    PERFORM VARYING RecordCount FROM 1 BY 1
            UNTIL EndOfFile
        READ PhoneList
            AT END MOVE "E" TO FileFlag
            NOT AT END PERFORM UpdateRecord
        END-READ
    END-PERFORM.

UpdateRecord.
    MOVE 972 TO AreaCode of PhoneData.
    REWRITE PhoneData.
    IF PhoneFileStatus IS NOT EQUAL TO "00"
       DISPLAY "Rewrite failed: " PhoneFileStatus
       STOP RUN
    END-IF.
```

This program opens the file for `I-O` and then performs a `READ` on each record. Each time a record is successfully read, the paragraph `UpdateRecord` is performed to stick a new value in for the area code and to `REWRITE` the record. The `REWRITE` verb puts the data right back in the file where it came from. Nothing else changes — the next `READ` goes sequentially to the next record just as if the `REWRITE` had never happened.

After COBOL rewrites all the area codes to 972, the program displays the following output:

```
Billy Bob Hamhock   9725558016
Tim       Shane     9725550918
Fanny     Brice     9725558381
Fred      Anonymous 9725559438
```



To `REWRITE` a record, you must `OPEN` the file in `I-O` mode. Before you can `REWRITE` a record, you must `READ` it. You can use the `FROM` option to `REWRITE` a record `FROM` a different location, but you must ensure that the record fits — the physical size of the slot in the sequential file cannot be changed.

Chapter 14

Working with Relative Files

In This Chapter

- Defining a relative file
- Discovering how to OPEN and CLOSE a relative file
- Understanding how to READ and WRITE records in a relative file
- Figuring out how to REWRITE and DELETE records in a relative file

A relative file, like any other file, consists of a bunch of data records. The *relative* thing comes about because each record in the file is assigned a unique number. These are not magic numbers — they are position numbers. The first record is number 1, the second is number 2, and so on. A relative file is like the menu in a Chinese restaurant — when your program wants to eat a record, it can order by number.

A relative file stores each record in a fixed position. It is the position that's numbered. The number is referred to as the *relative key* for that record. If you know the number, you can get the record. Because the number of a record is derived from the record's position in the file, the numbers also indicate how the records relate to one another in the file.

This chapter includes a step-by-step guide that you can use to define a relative file. I also describe the different ways that you can open or create a relative file. Finally, some sample programs demonstrate the various ways you can read and write records in a relative file.

What Is a Relative File Good for, Really?

The organization and capabilities of a relative file are based on those of a sequential file, which I describe in Chapter 13. You can do everything with a relative file that you can do with a sequential file, plus a little more. Even though you can randomly access the records by their relative numbers, you can still read them sequentially — you can OPEN and READ the whole file from front to back just as if it were a sequential file. Relative file organization adds the capability to select a specific record in the file. By using START to select a record by its relative file position, you can have COBOL start in the

middle of a file and READ records sequentially from that location. You can use START any number of times to move freely from any record to any other record. One other trick that you can perform with a relative file that you can't do with a sequential file is to DELETE a record right out of the middle of the file.

The most useful feature of a relative file is the completely random way in which you can access the records. After you store a record in the file, its number doesn't change, so you can save the numbers somewhere and come right back to the same record any time you want. For example, if you keep track of your automobile collection in a relative file, you can tag each of your cars with a number corresponding to its relative file number. If you want to retrieve the information on your Rolls Royce, you check its key ring and discover that it is, say, auto number 43. You can tell COBOL to go directly to its record in the file.

Defining a Relative File

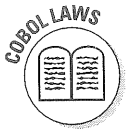
In the following sections of this chapter, I provide step-by-step instructions for setting up your program to read and write a relative file. This procedure is similar to the one that I describe in Chapter 13 to set things up for a sequential file. In fact, the file definitions are similar enough that several of the steps to define a relative file are the same as those for a sequential file — but a few important differences exist. For example, you need to tell COBOL that you are dealing with a relative file and you need to specify the name of the field that is to serve as the key.

Step 1: SELECT the file you want to use

To work with a relative file, your program must specify two names: the name of the file on your disk and the name you want your program to use internally for referring to that file. You complete this required step by putting a SELECT statement in the FILE-CONTROL paragraph of your program, like this:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT RefName
        ASSIGN TO "RealName"
        ORGANIZATION IS RELATIVE.
```

RealName represents the name of the file on your disk. *RefName* represents the name you use for the file within your program. In your own programs, you insert the name of a real file in place of *RealName* and *RefName*.



The OPEN verb creates a file if one does not already exist, so you should always declare the ORGANIZATION. Declaring the ORGANIZATION as RELATIVE is actually required only if you know you are going to create a new file. If the file already exists, it already has some ORGANIZATION, and you can't do anything to change the file's ORGANIZATION (short of deleting the file and starting over from scratch).

Step 2: Decide on your ACCESS MODE

The ACCESS MODE determines how you can READ from and WRITE to the file. If you just want to read the thing from front to back without any of the fancy acrobatics that you can perform with a relative file, you can specify the ACCESS MODE like this:

```
SELECT RefName
    ASSIGN TO "RealName"
    ACCESS MODE IS SEQUENTIAL.
```

In SEQUENTIAL mode, you can read right through the file without having to mess with the record numbers or anything else. In fact, if you don't tell COBOL anything about the ACCESS MODE, it assumes SEQUENTIAL as its default setting.

Say you want your program to be able to open a relative file, position the file at some specific record, READ that record (or possibly READ several records sequentially beginning with that record), and then CLOSE the file. To do this, you can define a RELATIVE KEY for SEQUENTIAL access, like this:

```
SELECT RefName
    ASSIGN TO "RealName"
    ACCESS MODE IS SEQUENTIAL RELATIVE KEY IS RecNo.
```

That looks a little odd. Is it sequential or is it relative? Okay, it *is* a little odd, but the START verb needs to have something to hang its hat on when you want to specify that the reading is to begin at some point other than the beginning of the file. The physical organization of the file on disk is relative, and you can use START to cause reading to commence at a relative record location, but after the initial positioning, access is limited to sequential.

After you declare the name of the RELATIVE KEY, you also need to declare the actual key in WORKING-STORAGE, something like this:

```
WORKING-STORAGE SECTION.
77  RecNo PIC 9(2) COMP.
```

Note: You can declare your record number counter to be any size you want. The example RecNo here is only two digits, which means the program has no way to address any record number greater than 99. If you are going to have, say, 5,000 records, you need to declare the record number as PIC 9(4) to be able to hold a number that large.

If you want to be able to move around in the file — to select which record gets read or written — you can set it up this way:

```
SELECT RefName
    ASSIGN TO "RealName"
    ACCESS MODE IS RANDOM RELATIVE KEY IS RecNo.
```

Defining the ACCESS MODE as RANDOM requires that you use the built-in record numbers (the ones that are a part of every relative file) to jump around in the file to find what you want.

COBOL lets you define one more ACCESS MODE. If you want the code in your program to be able choose between SEQUENTIAL access and RANDOM access, use DYNAMIC, like this:

```
SELECT RefName
    ASSIGN TO "RealName"
    ACCESS MODE IS DYNAMIC RELATIVE KEY IS RecNo.
```

If you define the file as having DYNAMIC access, you can access records pretty much any way you want. You can use START with built-in record numbers to jump around in the file and read the records randomly. You can also use the record numbers to jump into the middle of the file with a START command and then READ sequentially. DYNAMIC access gives you, at once, everything from both SEQUENTIAL access and RANDOM access.

The RELATIVE KEY is always optional — you only need to declare it if you are going to use it. It is against the law to specify RANDOM access on a file that is being used on the USING or GIVING phrase of a SORT or MERGE statement, which I discuss in Chapter 16. The laws of COBOL state that the record numbers must be nonzero, so you can always use zero to indicate that you have no record.

Step 3: Specify whether the file is OPTIONAL

You have the option of defining your file as OPTIONAL on the SELECT statement. Defining your file as OPTIONAL allows you to open a nonexistent file for I-O, or even for INPUT. The reasons you may want to do so are the same as those for a sequential file, which I describe in Step 3 of Chapter 13.

The word OPTIONAL comes right after the word SELECT, like this:

```
SELECT OPTIONAL RefName
    ASSIGN TO "RealName"
    ACCESS MODE IS DYNAMIC RELATIVE KEY IS RecNo.
```

Step 4: Create a place to stick the FILE STATUS

COBOL gives you a way to keep track of every little thing that happens to your file. A two-digit status value changes every time any operation whatsoever occurs on the file. For example, if you try to WRITE a record that is too big for the file, the two-digit code will be set to 44. (I describe the various status codes just a bit later in this section.)

Here's how you specify that you want to track the file status:

```
    SELECT RefName
        ASSIGN TO "RealName"
        ACCESS MODE IS RELATIVE
        FILE STATUS IS RefFileStatus.
```

The FILE STATUS clause creates a reference to RefFileStatus, which receives the new file status value each time the value changes. Then, of course, somewhere in WORKING-STORAGE, you have to actually define RefFileStatus, like this:

```
WORKING-STORAGE SECTION.
77  RefFileStatus PIC 99.
```

This location is updated every time you do something to your file. Table 14-1 lists the possible values that can wind up in this field as the status of a file. A status value of 00 indicates a complete success. A nonzero value less than 30 indicates that an exceptional condition exists, but it's nothing out of the ordinary. Any value 30 or over is not good. Later in this chapter, I present some example programs showing how the codes are used when processing files.

Steps 5–11: Complete the file definition

The last seven steps in the process of defining a relative file are identical to those for defining a sequential file. To complete the process, follow Steps 8 through 14 of Chapter 13.

Table 14-1 The Possible Values of Relative I-O Status

Value	*Meaning*
00	Whatever you did last worked. In fact, it worked so well that no further comment is necessary.
04	The READ worked, except that the record was either a little longer or a little shorter than expected.
05	The OPEN worked, but the file doesn't exist. It's okay though, because you have the file declared as OPTIONAL. If your program tried to OPEN the file for I-O or EXTEND, the file was created — otherwise, it still doesn't exist.
10	You tried to do a sequential READ and it failed. But it's no big deal — it's either the end of the file or it is an OPTIONAL file that doesn't exist. Either way, no data is to be had.
14	You tried to do a sequential READ and it failed. You can just consider this status an end-of-file condition that is a little weird. One thing it could be is that the relative key isn't big enough — for example, you could have declared it as PIC 9(2) and then tried to read sequentially to record number 100.
22	Duplicate key alert: You tried to write a record that would have created a duplicate key value in a relative file. You should know better — each record must have a unique key.
23	You tried to READ a record that doesn't exist. Or you tried to READ or START an OPTIONAL file that doesn't exist.
24	You tried to WRITE way beyond the maximum capacity of the file.
30	A permanent error exists with this file. COBOL doesn't know what it is. If you don't know, you have to ask somebody. A problem can exist with the file system.
34	You just attempted to WRITE beyond the boundary limits of the file. You're out of space. Either get permission from the system to write bigger files, or have your data resized.
35	You tried to OPEN a file for INPUT, I-O, or EXTEND, and the file doesn't exist. You can't do that unless you declare the file as OPTIONAL.
37	You can't OPEN that file. It could be that it's a read-only file you tried to open for EXTEND, OUTPUT, or I-O. It could be that you tried to open a file for I-O that just can't do that sort of thing (some files can't, you know). It could be that you tried to read a file that the system won't give you permission to read.
38	You tried to OPEN a locked file.
39	You tried to OPEN a file that has a completely different set of attributes than the one you have defined in your program.

Value	Meaning
41	You tried to OPEN a file that's already open.
42	You tried to CLOSE a file that wasn't open.
43	You surprised the file when it wasn't ready. You have to do a READ right before you do a DELETE or REWRITE.
44	The record you are trying to WRITE or REWRITE has a size problem. It is certainly not the same size as the one you just read — in fact, it could be larger that the largest record allowed or smaller than the smallest allowed. This can happen when you shop at those cut-rate data places.
46	You tried to execute a READ immediately following another failure. The previous READ statement failed, or the previous START statement failed.
47	You tried to START or READ a file that you have OPEN for OUTPUT or EXTEND. You can only do this with a file open for INPUT or I-O.
48	You tried to WRITE to a file that you have OPEN only for reading.
49	You tried to DELETE a record, or REWRITE a record, in a file that you did not OPEN in I-O mode.
9x	Any error message in the 90s is one that is peculiar to your compiler.

Opening a Relative File

Before your program can do anything with a file, it must OPEN the file. Your program can OPEN a relative file in four ways: for INPUT, OUTPUT, EXTEND, or I-O. The OPEN verb doesn't do any reading or writing of data — it just gets the file ready for action and notifies your program that the file is ready.

Opening a file for INPUT

If you only want to read data from the file, you OPEN it for INPUT. You can OPEN a file this way for all the access modes — SEQUENTIAL, RANDOM, and DYNAMIC. You can't write to the file at all, but you can read data from it any way you want. If the ACCESS MODE is either SEQUENTIAL or DYNAMIC, you can use START to position the file to any record and then read it. You can do an OPEN for INPUT this way:

```
OPEN INPUT RefName.
```

That's it — no options or anything. Just OPEN it and move on. RefName is the name found on the SELECT statement. If you declared the file as OPTIONAL and it does not exist, the first READ statement results in an end-of-file notification or an invalid key condition.

Opening a file for OUTPUT

If you just want to WRITE records to a file, you should OPEN the file for OUTPUT. If the file doesn't exist, COBOL creates a new file with the name from the SELECT statement that you designate in the OPEN statement. If the file already exists, COBOL overwrites the old file and creates a new and empty file. Here's how you OPEN a file for OUTPUT:

```
OPEN OUTPUT RefName.
```

That's all there is to it. No options. All you can do is WRITE to the file. It doesn't matter whether the ACCESS mode is SEQUENTIAL, RANDOM, or DYNAMIC — all you get to do is WRITE. You can't START or REWRITE because you can't READ. It doesn't matter whether you declared the file as OPTIONAL, because OPEN OUTPUT always either starts with a new file or empties the one that already exists.

Opening a file for EXTEND

If you already have a file, the file already contains some data, and all you want to do is add some new data onto the end of the file, you're in the right place. EXTEND only works if the ACCESS MODE is SEQUENTIAL — EXTEND won't work if the ACCESS MODE is RANDOM or DYNAMIC. Also, if the file doesn't already exist, you need to declare it as OPTIONAL. If you meet all these qualifications, you can then write the following code:

```
OPEN EXTEND RefName.
```

After you OPEN the file for EXTEND, all you can do is WRITE. You can't position the file in any way. You can't modify any existing records. All you can do is WRITE new records that you add to the end of the file.

Opening a file for I-O

If you want to have as much power as possible over the file, OPEN it for I-O. Consider this one the *everything* way to OPEN a file. And here's all it takes:

```
OPEN I-O RefName.
```

After you OPEN the file for I-O, you can do just about what you want to. I say "just about" because certain combinations impose a limit or two:

- If the ACCESS MODE is SEQUENTIAL, you can't WRITE new records to the file (although you can REWRITE and DELETE).
- If the ACCESS MODE is RANDOM, you can't START the file at some specific record number (you put the relative record number into the RELATIVE KEY and the READ or WRITE verb positions the file).
- If you try to OPEN a file that does not exist, and you didn't declare it as OPTIONAL, it won't work.

One combination overcomes all limitations. If you declare your relative file according to this prescription, you are able to do everything that can be done to a relative file:

- Declare the ACCESS MODE as DYNAMIC.
- Make the file OPTIONAL.
- OPEN the file for I-O.

COBOL creates the file if it doesn't exist, and you can START, READ, WRITE, REWRITE, and DELETE the file. You can also store values in the RELATIVE KEY value and the READ and WRITE verbs position the file for you.

Closing a Relative File

The first thing you do to a file is to OPEN it; the last thing you do is to CLOSE it. Although you can CLOSE a file WITH LOCK, under almost all circumstances the statement that you use to CLOSE a file is simple and straightforward:

```
CLOSE RefName.
```

After you CLOSE a file, no more activity can occur on that file unless you OPEN it again (which you are allowed to do). Well, you are usually allowed to OPEN the file again. If you CLOSE the file with the following statement, you can't OPEN the file again:

```
CLOSE RefName WITH LOCK.
```

What this WITH LOCK clause means is, for only *this* run of only *this* program, you cannot OPEN the file again. Any other program can open the file and, if you run this program again, you can open the file again.

Writing to a Relative File

The code example I present in this section shows you how to WRITE to a relative file. This example creates a file and populates it with data. Subsequent examples in this chapter use the file (and the data it contains). The file is a very simple database that contains the names of some Earthlings along with some of their characteristics.

This example program creates the file and fills it with seven data records:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. RelativeWrite.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT OPTIONAL AnimalList
        ASSIGN TO "animals"
        ORGANIZATION IS RELATIVE
        ACCESS MODE IS SEQUENTIAL
            RELATIVE KEY IS RecordNumber.
DATA DIVISION.
FILE SECTION.
FD AnimalList
    RECORD CONTAINS 16 CHARACTERS.
01  AnimalData.
    05  Name        PIC X(8).
    05  NormalColor PIC X(6).
    05  LegCount    PIC 9.
    05  Friendly    PIC X.
        88  IsFriendly VALUE "Y".
        88  IsNotFriendly VALUE "N".
        88  MayBeFriendly VALUE "M".
WORKING-STORAGE SECTION.
01  InitialValueText.
    02  FILLER PIC X(16) VALUE "ElephantGray  4Y"
    02  FILLER PIC X(16) VALUE "Spider  Black 8M"
    02  FILLER PIC X(16) VALUE "Fire antRed   6N"
    02  FILLER PIC X(16) VALUE "Panther Black 4N"
    02  FILLER PIC X(16) VALUE "Shark   Gray  0N"
    02  FILLER PIC X(16) VALUE "Human   Varies2M"
    02  FILLER PIC X(16) VALUE "CardinalRed   2Y"
01  InitialValueArray REDEFINES InitialValueText.
    02  ValueArray PIC X(16) OCCURS 7 TIMES.
77  RecordNumber PIC 9(2) COMP.
77  AnimalFileStatus PIC XX VALUE "00".
77  I PIC 9(2).
```

```
PROCEDURE DIVISION.
Mainline.
    OPEN OUTPUT AnimalList.
    IF AnimalFileStatus IS NOT EQUAL TO "00"
       DISPLAY "Open failed: " AnimalFileStatus
       STOP RUN
    END-IF.
    PERFORM VARYING I FROM 1 BY 1
            UNTIL I IS GREATER THAN 7
        MOVE ValueArray(I) TO AnimalData
        WRITE AnimalData
            INVALID KEY
                DISPLAY "Invalid key error"
            NOT INVALID KEY
                DISPLAY Name " is record number "
                        RecordNumber
        END-WRITE
    END-PERFORM.
    CLOSE AnimalList.
    STOP RUN.
```

This example declares the values in `InitialValueText` for the contents of a bunch of records and writes them to the relative file. Because the program asks to `OPEN` the file for `OUTPUT`, the file is either created or completely emptied of data on the first `WRITE`.

The `WRITE` statement is in a loop that executes once for each record. A pair of `DISPLAY` statements follows the `WRITE` statement — one for success and one for failure. Whenever a `WRITE` succeeds, the animal's name and its record number are printed. The output looks like this:

```
Elephant is record number 01
Spider   is record number 02
Fire ant is record number 03
Panther  is record number 04
Shark    is record number 05
Human    is record number 06
Cardinal is record number 07
```

Notice how the `WRITE` statement has an `INVALID KEY` clause to check the possibility of a failure because of an invalid key condition. You may well wonder how that could happen, because the program itself is generating the key values. Assume that I had really gotten industrious in this example and instead of writing seven records I had tried to write 150 records. That would work except for the little detail that only two digits are available in `RecordNumber`, the field the program uses to hold the key values. Oops. Of course, the `WRITE` could fail for other reasons also — the disk drive could be having a bad hair day.

Reading a Relative File in a Sequential Way

You can treat a relative file just like a sequential file. Just OPEN the file for INPUT and start to READ the records. As an added bonus, you can start somewhere in the middle of the file by using the START command to skip as many records as you would like. This capability can be useful if a program simply needs to read straight through the file and extract records — for example, to produce a printed report, copy the records to another file, or perform a simple search for records with certain characteristics.

The following code shows an example of reading the records both ways — once from the beginning and once from a place in the middle. This example uses the file created in the preceding example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. RelSeqRead.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT AnimalList
        ASSIGN TO "animals"
        ORGANIZATION IS RELATIVE
        ACCESS MODE IS SEQUENTIAL
            RELATIVE KEY IS RecordNumber.
DATA DIVISION.
FILE SECTION.
FD AnimalList
    RECORD CONTAINS 16 CHARACTERS.
01  AnimalData.
    05  Name         PIC X(8).
    05  NormalColor  PIC X(6).
    05  LegCount     PIC 9.
    05  Friendly     PIC X.
        88  IsFriendly VALUE "Y".
        88  IsNotFriendly VALUE "N".
        88  MayBeFriendly VALUE "M".
WORKING-STORAGE SECTION.
01  EndOfFileFlag PIC X.
    88  EndOfFile VALUE "E".
77  RecordNumber PIC 9(2) COMP.
77  AnimalFileStatus PIC XX VALUE "00".
```

```
PROCEDURE DIVISION.
Mainline.
    DISPLAY "   The entire file..."
    OPEN INPUT AnimalList.
    IF AnimalFileStatus IS NOT EQUAL TO "00"
       DISPLAY "Open failed: " AnimalFileStatus
       STOP RUN
    END-IF.
    PERFORM ReadAndDisplay.
    CLOSE AnimalList.

    DISPLAY "   Starting at 3..."
    OPEN INPUT AnimalList.
    MOVE 3 TO RecordNumber.
    START AnimalList.
    PERFORM ReadAndDisplay.
    CLOSE AnimalList.
    STOP RUN.

ReadAndDisplay.
    MOVE "N" TO EndOfFileFlag.
    PERFORM UNTIL EndOfFile
        READ AnimalList
            AT END MOVE "E" TO EndOfFileFlag
            NOT AT END PERFORM DisplayAnimal
        END-READ
    END-PERFORM.

DisplayAnimal.
    DISPLAY "The " LegCount " legged " Name " "
            WITH NO ADVANCING.
    IF IsFriendly
        DISPLAY "is friendly."
    ELSE IF IsNotFriendly
        DISPLAY "is not friendly."
    ELSE IF MayBeFriendly
        DISPLAY "is sometimes friendly.".
```

Because the file was open for `INPUT`, the program must `CLOSE` the file after it gets to the end of the file. You cannot use `START` to move somewhere else after you start to `READ` the file sequentially. Look, when you `OPEN` for `INPUT`, sequential is all you get.

Here's the output of this program:

```
   The entire file...
The 4 legged Elephant is friendly.
The 8 legged Spider   is sometimes friendly.
The 6 legged Fire ant is not friendly.
The 4 legged Panther  is not friendly.
The 0 legged Shark    is not friendly.
The 2 legged Human    is sometimes friendly.
The 2 legged Cardinal is friendly.
   Starting at 3...
The 6 legged Fire ant is not friendly.
The 4 legged Panther  is not friendly.
The 0 legged Shark    is not friendly.
The 2 legged Human    is sometimes friendly.
The 2 legged Cardinal is friendly.
```

Notice how the positioning trick works. The value of the KEY is set to the number of the desired record, the START verb is used to position the file, and then the program proceeds normally with the sequential READ.

This example uses the default KEY setting on the START verb, like this:

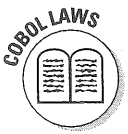
```
    MOVE 3 TO RecordNumber.
    START AnimalList.
```

In other words, the START verb uses the current value of the KEY — RecordNumber — to determine where the program should begin the sequential READ. Unless you specify otherwise, the program goes directly to the record number you specify. The following examples show the other ways in which you can tell the program where to START the READ:

```
START AnimalList KEY IS EQUAL TO RecordNumber.
START AnimalList KEY IS = RecordNumber.
START AnimalList KEY IS GREATER THAN RecordNumber.
START AnimalList KEY IS > RecordNumber.
START AnimalList KEY IS NOT LESS THAN RecordNumber.
START AnimalList KEY IS NOT < RecordNumber.
START AnimalList KEY IS GREATER THAN
    OR EQUAL TO RecordNumber.
START AnimalList KEY IS >= RecordNumber.
```

If you let the KEY expression just default, as in the example, it will always be KEY IS EQUAL TO, which has almost always been fancy enough for my taste.

I mean, we're talking relative record positions in a file — just how fancy can you get? The only time you may want to use one of the other KEY IS expressions is when some records may have been deleted and the one you are looking for may not exist.



If you OPEN a file for INPUT, you must use the START command before you READ any records. After you start to READ, you must either continue to the end of the file or CLOSE the file. If you are going to use the START command, you must have a RELATIVE KEY defined as part of the ACCESS MODE in the SELECT statement.

Reading in a Relative Way

Anything you can do sequentially you can also do relatively; you just have to take a little more control over the KEY value. Every time you do a relative READ, you must set the key value yourself. You can read the records in any order you wish — front to back, back to front, from the middle to either end, or completely at random. Just set the key value to the record you want and then read the record.

The following example — by just adding one to the record number for each read — uses relative READ operations to do the same thing as the previous example does with sequential READ operations:

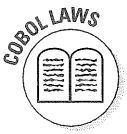
```
IDENTIFICATION DIVISION.
PROGRAM-ID. RelRelRead.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT AnimalList
        ASSIGN TO "animals"
        ORGANIZATION IS RELATIVE
        ACCESS MODE IS DYNAMIC
            RELATIVE KEY IS RecordNumber.
DATA DIVISION.
FILE SECTION.
FD AnimalList
    RECORD CONTAINS 16 CHARACTERS.
01  AnimalData.
    05  Name        PIC X(8).
    05  NormalColor PIC X(6).
    05  LegCount    PIC 9.
    05  Friendly    PIC X.
```

(continued)

(continued)

```
        88  IsFriendly VALUE "Y".
        88  IsNotFriendly VALUE "N".
        88  MayBeFriendly VALUE "M".
WORKING-STORAGE SECTION.
01  EndOfFileFlag PIC X.
    88  EndOfFile VALUE "E".
77  RecordNumber PIC 9(2) COMP.
77  AnimalFileStatus PIC XX VALUE "00".
PROCEDURE DIVISION.
Mainline.
    DISPLAY "   The entire file..."
    OPEN INPUT AnimalList.
    IF AnimalFileStatus IS NOT EQUAL TO "00"
       DISPLAY "Open failed: " AnimalFileStatus
       STOP RUN
    END-IF.
    MOVE 1 TO RecordNumber.
    PERFORM ReadAndDisplay.

    DISPLAY "   Starting at 3..."
    MOVE 3 TO RecordNumber.
    PERFORM ReadAndDisplay.
    CLOSE AnimalList.
    STOP RUN.

ReadAndDisplay.
    MOVE "N" TO EndOfFileFlag.
    PERFORM UNTIL EndOfFile
        READ AnimalList
            INVALID KEY MOVE "E" TO EndOfFileFlag
            NOT INVALID KEY PERFORM DisplayAnimal
        END-READ
        ADD 1 TO RecordNumber
    END-PERFORM.

DisplayAnimal.
    DISPLAY "The " LegCount " legged " Name " "
            WITH NO ADVANCING.
    IF IsFriendly
        DISPLAY "is friendly."
    ELSE IF IsNotFriendly
        DISPLAY "is not friendly."
    ELSE IF MayBeFriendly
        DISPLAY "is sometimes friendly.".
```

This program declares the file DYNAMIC and proceeds to OPEN the file for INPUT. This example reads the same set of data records as the preceding example does, and prints exactly the same output, but the mechanics are a bit different. When ACCESS MODE IS DYNAMIC, to read any record in the file, you must set the value of its key. You can set the key value to any record you would like to read — this example simply adds one to the key value each time to read the next record. This level of control gives you completely random access to every record in the file.



Whenever you perform a relative READ with DYNAMIC or RANDOM access, you must set the key value yourself. The key value is never modified by a DYNAMIC or RANDOM READ as it is with a SEQUENTIAL READ.

Rewriting a Record in a Relative File

You can read a record from the file and write it right back again, which you would presumably do after making some change to the data. The following example shows how to REWRITE a record in a file that already exists. The file is the one that was originally created by the example in the section "Writing to a Relative File" earlier in this chapter:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. RelativeRewrite.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT AnimalList
        ASSIGN TO "animals"
        ORGANIZATION IS RELATIVE
        ACCESS MODE IS DYNAMIC
        RELATIVE KEY IS RecordNumber.
DATA DIVISION.
FILE SECTION.
FD AnimalList
    RECORD CONTAINS 16 CHARACTERS.
01  AnimalData.
    05  Name        PIC X(8).
    05  NormalColor PIC X(6).
    05  LegCount    PIC 9.
    05  Friendly    PIC X.
        88  IsFriendly VALUE "Y".
        88  IsNotFriendly VALUE "N".
        88  MayBeFriendly VALUE "M".
WORKING-STORAGE SECTION.
```

(continued)

(continued)

```
01  InvalidKey PIC X.
    88  IsInvalid VALUE "Y".
77  RecordNumber PIC 9(2) COMP.
77  AnimalFileStatus PIC XX VALUE "00".
PROCEDURE DIVISION.
Mainline.
    OPEN I-O AnimalList.
    IF AnimalFileStatus IS NOT EQUAL TO "00"
       DISPLAY "Open failed: " AnimalFileStatus
       STOP RUN
    END-IF.
    MOVE 5 TO RecordNumber.
    PERFORM ReadAndDisplay.
    MOVE "M" TO Friendly.
    MOVE 3 TO LegCount.
    REWRITE AnimalData
        INVALID KEY DISPLAY "Bad location for REWRITE"
    END-REWRITE.
    PERFORM ReadAndDisplay.
    CLOSE AnimalList.
    STOP RUN.

ReadAndDisplay.
    READ AnimalList
        INVALID KEY MOVE "Y" TO InvalidKey
        NOT INVALID KEY PERFORM DisplayAnimal
    END-READ.

DisplayAnimal.
    DISPLAY "The " LegCount " legged " Name " "
            WITH NO ADVANCING.
    IF IsFriendly
        DISPLAY "is friendly."
    ELSE IF IsNotFriendly
        DISPLAY "is not friendly."
    ELSE IF MayBeFriendly
        DISPLAY "is sometimes friendly.".
```

This example reads relative record number 5, makes some changes to the data, and then writes it back to the file, replacing the old data record with the new one. To do this, the `RecordNumber` — the relative key value for the file — is set to 5 and the record is read and displayed, producing this line of output:

```
The 0 legged Shark    is not friendly.
```

The record is now in memory. Two MOVE statements change the values in the record (the level of friendliness and the leg count) and a REWRITE statement writes the data back to the disk. The key value is left untouched from the previous READ because it is still addressing the correct record. After the REWRITE successfully completes its mission, the record can be read and displayed again, producing this output:

```
The 3 legged Shark    is sometimes friendly.
```

Deleting a Record from a Relative File

One thing you can do with a relative file that you cannot do with a sequential file is DELETE a record. It's really easy, too, as the following example shows:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. RelativeDelete.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT AnimalList
        ASSIGN TO "animals"
        ORGANIZATION IS RELATIVE
        ACCESS MODE IS DYNAMIC
        RELATIVE KEY IS RecordNumber.
DATA DIVISION.
FILE SECTION.
FD AnimalList
    RECORD CONTAINS 16 CHARACTERS.
01  AnimalData.
    05  Name        PIC X(8).
    05  NormalColor PIC X(6).
    05  LegCount    PIC 9.
    05  Friendly    PIC X.
        88  IsFriendly VALUE "Y".
        88  IsNotFriendly VALUE "N".
        88  MayBeFriendly VALUE "M".
WORKING-STORAGE SECTION.
01  InvalidKey PIC X.
    88  IsInvalid VALUE "Y".
77  RecordNumber PIC 9(2) COMP.
77  AnimalFileStatus PIC XX VALUE "00".
PROCEDURE DIVISION.
Mainline.
```

(continued)

(continued)

```
OPEN I-O AnimalList.
IF AnimalFileStatus IS NOT EQUAL TO "00"
   DISPLAY "Open failed: " AnimalFileStatus
   STOP RUN
END-IF.
MOVE 5 TO RecordNumber.
DELETE AnimalList
    INVALID KEY DISPLAY
        "Record number " RecordNumber " invalid."
    NOT INVALID KEY DISPLAY
        "Record number " RecordNumber " deleted."
END-DELETE.
CLOSE AnimalList.
STOP RUN.
```

All you need to do is OPEN the file for I-O, stick the number of the record you want to delete in the key, and pull the trigger on the DELETE verb. It's gone. Of course, like the other file operations, you need to check the results to see what happened. The statement on the INVALID KEY clause is executed if the value in the relative key specifies the number of a record that does not exist. If the record is successfully deleted, the statement on the NOT INVALID KEY clause is executed.

An old COBOL adage states, "Write records; read files." In other words, whenever you have a WRITE statement, you use the name of the record that you defined following the FD statement in the FILE SECTION. Whenever you have a READ statement, you use the name of the file you defined on the SELECT statement in the FILE-CONTROL paragraph. And then comes DELETE. It has to go one way or the other, and COBOL chooses to group it with READ, so to execute a DELETE, you use the name of the file from the SELECT statement. It kind of looks like you are trying to delete the whole file, but if you think of the DELETE verb as being DELETE RECORD FROM, it makes more sense.

The preceding example demonstrates something called a *logical delete.* If you had a way of peeking into the file, you'd see that the file still has a place for the record, and it may actually still contain the data. COBOL puts an evil-eye mark on the record and your program can't read it. If you try to use the KEY value to read the record (or even if you try to DELETE it again), COBOL simply tells you that no such record exists. If you READ sequentially through the file without using record numbers, the record is skipped. If you use a record number to READ it directly, the READ may or may not work — it depends on your compiler. The space is available — you can use WRITE to add a new record with that key value.

Chapter 15

Working with Indexed Files

In This Chapter

- Defining an indexed file
- Discovering how to OPEN and CLOSE an indexed file
- Understanding how to WRITE records to an indexed file
- Figuring out how to READ specific records, using primary and alternate keys
- Examining how to REWRITE and DELETE records in an indexed file

Files contain data. The world produces lots and lots of data for these files. As time goes by, and more data arrives for storage, some files become very large. So large, in fact, that finding things in those files gets to be cumbersome and time consuming. But finding things is what indexed files are all about.

Say you write a program that has the job of finding one record in a file and displaying information from that record on the screen. Without an indexed file, every time you run the program, it opens the file and reads through the records until the program finds the record to be displayed. If you have only a few records in the file, or even a few hundred records, the program quickly finds the record it needs and displays it. On the other hand, if you have multiple thousands of records, you could end up waiting several minutes for each record to be displayed. Imagine reading a telephone book from the beginning every time you wanted to look up a number.

To the rescue come the key and the index. These two items, working together, can help you find your data very quickly.

Here's how indexed files work. You designate one of the fields in your data record to act as the key. For example, the key for a telephone book would be the last name of the person with a phone.

Every time you write a record to your file, the value of the key and the record number in the data file are supplied to the index. (The record number is used to locate a record, like the relative file record numbers that I describe in Chapter 14.) The index, which keeps the keys in some logical

order for quick lookup, adds the new key value (along with the record number) to its list of key values. Also, every time you delete a record from the file, the corresponding key and record number are removed from the index.

Now comes the fun part. Whenever you need a record, you simply supply the key value to the index, the index answers with the record number, and then you can go directly to the record in the data file. Actually, as you see in this chapter, this process is even easier than I describe it here because the whole key/index/record relationship is built directly into COBOL. You never need to manipulate the key values or record numbers.

The indexed file is one of COBOL's best tricks. It is one of the things that make COBOL the international star that it is today.

Defining an Indexed File

To read and write an indexed file, you must define one in your program. In the following sections, I present a step-by-step procedure that you can use as a checklist for defining an indexed file. Subsequent sections in this chapter show you how to handle the input and output of data that's stored in indexed files. You can also use the checklist to figure out the index file definitions of a program you just happened to find lying around on your desk with a note attached to it saying, "Fix me."

The keys to the millennium problem

COBOL's built-in key capability is a main contributor to one of the most talked-about issues in computing today — the millennium problem.

An indexed file uses a specified field in each record to keep all the records in sorted order. If that field contains a two-digit year, when the year changes from 99 to 00, the records in the file are going to be in the wrong order. For example, bills that are past due will show up as the first ones paid. Bills that are currently due will show up as being 100 years past due. There is no end to the combination of confusion that will come from scrambling dates. This is a particularly wicked form of the millennium problem because it lives in the keys that index the files on disk — you can find two-digit years in keys of files that have been stashed in the archives for years. If a program is modified to work with four-digit years, it has no way to read the old archived data.

Is it feasible to spend the time and effort needed to convert all that data — data that may never be used? If the data isn't converted, it could become inaccessible and crucial at the same time. Oh dear, what to do? Solutions exist, but they all take some work. I describe some COBOL solutions in Chapter 17.

This key and indexing thing is something of an *OIC* (pronounced "Oh, I see"). Working with indexed files doesn't involve anything that's really difficult to understand — just lots of little things that you need to remember. To start out, just go through the steps I describe in this chapter, and use the parts you need. You may find yourself coming back to this chapter (and to your compiler's documentation) quite often at first. As you work with indexed files a little bit, you get the feel of things. Later, when you come back here to check out something, you can say, "OIC!"

Step 1: SELECT the file you want to use

To work with an indexed file, your program must specify four things:

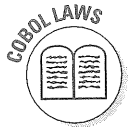
- The name of the file on your disk
- The name you want your program to use for referring to that file
- The organization of the file — that is, `INDEXED`
- The name of the key field on which the records in the file are indexed

You complete this required step by putting a `SELECT` statement in the `FILE-CONTROL` paragraph of your program. Here's the minimal form of a `SELECT` for an indexed file:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT RefName
        ASSIGN TO "RealName"
        ORGANIZATION IS INDEXED
        RECORD KEY IS NameKey.
```

RealName represents the name of the file on your disk. *RefName* represents the name you use for the file within your program. *NameKey* represents the name of the key field by which the records in the file are indexed.

You must pick the name of a key, the *NameKey*. The name you pick is a field in the record you define as part of the record layout in the program's `FILE SECTION`. This key is known as the *primary key;* you can have only one primary key. As I discuss in the next step of this procedure, you can have other keys — known as `ALTERNATE` keys. For example, the primary key of a file containing a list of invoices can be the invoice number, and an alternate key can be the invoice date.



COBOL has some basic laws involving the keys in an INDEXED file:

- The field declared as the RECORD KEY must be an alphanumeric member of a record description for this file in the FILE SECTION.
- You can't change the definition of the key after you create the file.
- If the file allows variable-length records, the primary key must be completely contained inside the smallest possible record. (I define variable-length records in Chapter 13, in the section "Step 11: Define the RECORD size.")
- Duplicate values are not allowed for primary keys — that is, no two records with the same key value can be stored in the file.

Step 2: Add an ALTERNATE Key

An indexed file can have more than one key, and any key other than the primary key is known as an ALTERNATE key. In fact, if you want, you can have two or more ALTERNATE keys. The truth is, if you really want to be obsessive, you can make every alphanumeric field in the data record a key — COBOL doesn't care. However, this is almost never useful — my experience has been that three keys in one record are about the most you ever need.

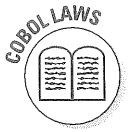
This keying of a file is one of the cornerstones of COBOL's power as a business language. For example, say you are running a business that needs to keep detailed track of numerous invoices. You can put all the invoices in one file, using the invoice number as the primary key. An alternate key can be the name of the customer. Because alternate keys allow duplicates, you can have the same customer name on several invoices. You can use the customer-name alternate key to find all the invoices for any one customer. You can have the date of the invoice as another alternate key, and use the invoice-date key to locate all the invoices that were written during a certain week, or month, or whatever.

You define an ALTERNATE key in the SELECT statement, like this:

```
SELECT RefName
    ASSIGN TO "RealName"
    ORGANIZATION IS INDEXED
    RECORD KEY IS NameKey
    ALTERNATE RECORD KEY IS CityKey.
```

Unlike the primary key, the ALTERNATE key can be defined to allow for duplicates. For example, if you want to allow two entries in your file to be in the same city, the definition line looks like this:

```
    ALTERNATE RECORD KEY IS CityKey WITH DUPLICATES.
```



COBOL has a few laws that you must obey when dealing with keys:

- If you do not specify WITH DUPLICATES for an ALTERNATE key, every record in the file must hold a unique value for that key.
- The primary key cannot also be used as an ALTERNATE key.
- An ALTERNATE key and a primary key can overlap and use some of the same characters (for the mechanics of doing this, see the discussion of REDEFINES in Chapter 4), but they cannot both start in the same character position.
- If the file contains variable-length records (which I describe in Step 11 of Chapter 13), all keys must be contained within the shortest possible record.
- The definitions of the keys — primary or alternate — cannot be changed after the file has been created.
- Sequential access with duplicate keys retrieves the records in the order in which they were placed in the file.
- Performing random access on a duplicate key retrieves only the first record that was written with the requested key value.

Step 3: Specify whether the file is OPTIONAL

You have the option of defining your file as OPTIONAL on the SELECT statement. Defining your file as OPTIONAL allows you to open a nonexistent file for I-O, EXTEND, or even for INPUT. The reasons you may want to do this are the same as they are for a sequential file. I describe those reasons in Step 3 of Chapter 13.

The word OPTIONAL comes right after the word SELECT, like this:

```
SELECT OPTIONAL RefName
    ASSIGN TO "RealName"
    ORGANIZATION IS INDEXED
    RECORD KEY IS NameKey.
```

Step 4: RESERVE some extra space

You can optionally allocate some extra space in the computer's memory to hold input and output records. A program can access data from the computer's internal memory much faster than from a file on disk or tape. By setting aside some space in memory for holding input and output records, you can improve your program's efficiency.

The following example sets aside space enough to hold 15 records:

```
SELECT RefName
    ASSIGN TO "RealName"
    RESERVE 15 AREAS.
```

The keyword AREAS is optional; you can just omit it, or you can spell it AREA. If you want more information on reserving space, see the discussion of reserving space for a sequential file in Step 4 of Chapter 13.

Step 5: Select the ACCESS MODE

The ACCESS MODE determines the methods available for the program to READ and WRITE the file. Even though the file's ORGANIZATION is INDEXED, you can have your program treat it as a sequential file by declaring the ACCESS MODE this way:

```
SELECT RefName
    ASSIGN TO "RealName"
    ORGANIZATION IS INDEXED
    ACCESS MODE IS SEQUENTIAL.
```

Using SEQUENTIAL access allows you to READ and WRITE the file while you completely ignore any keys. This can be necessary, for example, if you need to print a report listing all the customers in a certain city, and the name of the city is not one of the keys — the only thing to do is read the whole file and pull out the records you need. On the other hand, if you are only going to READ and WRITE by using key values, you can declare the ACCESS MODE this way:

```
SELECT RefName
    ASSIGN TO "RealName"
    ORGANIZATION IS INDEXED
    ACCESS MODE IS RANDOM.
```

Having RANDOM access to an INDEXED file gives you the full power to access any and all records by key values.

It's rare, but you may find yourself in the situation in which you need RANDOM access to a file for some operations, but SEQUENTIAL access for other operations. You can get that sort of privilege this way:

```
SELECT RefName
```

```
        ASSIGN TO "RealName"
        ORGANIZATION IS INDEXED
        ACCESS MODE IS DYNAMIC.
```

With DYNAMIC access, you can do some SEQUENTIAL stuff, stop and do some RANDOM stuff, and then come back and do some more SEQUENTIAL stuff. For example, if you want to extract all the records with addresses within a specific zip code, you use the zip code alternate key to get the first matching record, then read sequentially to get all the rest of them. This mode gives you the power to do whatever you want to do. It's heady stuff I know, but I'm sure you can handle it. After all, it's obvious that you are discerning, intelligent, and very capable of solving real-world problems — I can tell by the kind of books you read.

Step 6: Create a place to stick the file status

In case you were wondering where to put your COBOL status symbol, your curiosity is about to be slaked. Your program can include a two-digit status value that changes every time something happens to the file — good or bad. Here's how you specify that you want to track the file status:

```
    SELECT RefName
        ASSIGN TO "RealName"
        ORGANIZATION IS INDEXED
        RECORD KEY IS NameKey
        FILE STATUS IS RefFileStatus.
```

After you specify that you want to track the file status, you just sneak over into WORKING-STORAGE, and define the place you would like to award the responsibility of holding your status symbol, which you define with the following code:

```
WORKING-STORAGE SECTION.
77  RefFileStatus PIC 99.
```

This is the declaration of a field named RefFileStatus, which saves the new file status value each time the value changes. Your program updates RefFileStatus every time something noteworthy happens to your file. Table 15-1 lists the possible values that can wind up here as the status of a file. The operations that I describe later in this chapter (READ, WRITE, and so on) cause these status values to be created. If you check the value and get zero, that's perfect. If the value is less than 30, it may not be perfect, but it is the sort of thing that happens during normal operations. If the value is 30 or more, file it under "oops."

Table 15-1 The Possible Values of Sequential I-O Status

Value	*Meaning*
00	Whatever you did last worked. In fact, it worked so well that no further comment is necessary.
02	Whatever you did last worked. If you just did a READ on an ALTERNATE key, you allowed duplicates for it and the next record you read is going to be a duplicate key value of this one. If you just did a WRITE or a REWRITE, some ALTERNATE key or another has a duplicate value — but that's okay because you said it could.
04	The READ worked, except that the record was either a little long or a little short. In fact, it was just a smidgen too long or too short for the file definition. But that's okay — COBOL just snipped off its tail or stuck on some data and went ahead.
05	The OPEN worked, but the file didn't exist before the OPEN was executed. It's okay though, because you have the file declared as OPTIONAL. If your OPEN was for I-O or EXTEND, the file was created — otherwise, it still doesn't exist.
10	The sequential READ failed. But nothing is wrong — it's either the end of the file or you are reading an OPTIONAL file that doesn't exist. Either way, you're not missing anything because if this file contained any information, you have already read it.
21	You caused yourself a problem with the primary key. You executed a READ, which worked okay. But then you changed the value of the primary key and tried to execute a REWRITE. You can't do that. The only way you can change a primary key value is to DELETE the record and WRITE a new one.
22	You have a key value problem. You tried to execute a WRITE or REWRITE that would have created a duplicate key. COBOL won't tell you which key — it is either the primary key (which can never be duplicated) or an ALTERNATE key that isn't defined to allow duplicates.
23	You tried to READ something that isn't there. You either tried to use a key value that is not in the file, or you tried to READ an OPTIONAL input file that does not exist.
24	You have just run out of space. Your computer system imposed some sort of size limitation on this file and you have just reached the limit. You can either get permission to use more disk space, or try to store less data.
30	A really bad file error exists. In fact, things are so fouled up that even COBOL has no idea what the problem is — you're on your own with this one.

Value	Meaning
35	You tried to OPEN a file for INPUT, I-O, or EXTEND, and the file doesn't exist. You can only do that if you declare the file as OPTIONAL. Do you have a reason for not declaring the file as OPTIONAL?
37	You can't OPEN that file in that manner. It could be that it's a read-only file you tried to open for EXTEND, OUTPUT, or I-O. It could be that you tried to open a file for I-O, but it can't be opened that way. It could be that you tried to read from a file that, at the system level, prohibits you from reading it.
38	You tried to OPEN a file that you closed WITH LOCK. This is known as shooting yourself in the file.
39	You tried to OPEN a file that has a completely different set of attributes than the one you defined in your program. Go find the program that created the file and plagiarize the file definitions from it.
41	You tried to OPEN a file that's already open.
42	You tried to CLOSE a file that wasn't open.
43	You surprised the file when it wasn't ready. You have to do a READ right before you do a REWRITE, or you tried to DELETE something without a valid primary key value (the value you put into the primary key must match the value of the one in the record you want to delete).
44	The record you are trying to REWRITE has a size problem. It could be larger that the largest record allowed or smaller than the smallest record allowed. Before you try this REWRITE again, have your record resized to fit.
46	The previous READ statement failed, and so did this one. Why not just try it again? And again? Look, one of the READ statements back there hit the end of file (or possibly encountered some error condition) and trying to READ it again and again is not going to fix it.
47	You tried to READ from a file that you have OPEN for OUTPUT or EXTEND.
48	You tried to WRITE to a file that you have OPEN only for reading.
49	You tried to DELETE or REWRITE a record in a file that you did not OPEN in I-O mode.
9*x*	Any error message in the 90s is one that is peculiar to your compiler.

Steps 7–13: Complete the file definition

To complete the file definition, follow Steps 8 through 14 in Chapter 13 — the rest of the steps are identical to the ones for defining sequential files. Well, almost. The data records that you define for the file must include the

keys you defined in the SELECT statement. For example, a record definition that works with the keys that I define in Step 2 (earlier in this chapter) can look like this:

```
FD  RefName
    RECORD CONTAINS 121 CHARACTERS.
01  RefDataRecord.
    02  NameKey     PIC X(16).
    02  CompanyName PIC X(10).
    02  CityKey     PIC X(8).
    02  StateCode   PIC X(2).
    02  Address1    PIC X(40).
    02  Address2    PIC X(40).
    02  ZipCode     PIC 9(5).
```

The SELECT statement in Step 2 defines the primary key as NameKey and the alternate key as CityKey. This record definition defines both the record and the keys that go in it. A key can be located anywhere in a record — this example puts the primary key as the first field and an alternate key as the third field, but COBOL doesn't really care where you put them.

A key doesn't have to be coded as a single field — you can bust a key up into parts and make a record out of it, like this:

```
01  RefDataRecord.
    02  NameKey.
        03  LastName   PIC X(8).
        02  FirstName  PIC X(8).
```

You need to take great care when picking friends, melons, deodorant, and indexed-file keys. After you select your keys and create a file, the only way to change the keys (or anything else about the file, for that matter) is to delete the file and start over. No matter how well you plan, you need to do this from time to time, but it's a bummer when you have to do it because of carelessness.

To continue the step-by-step procedure for setting up your file access, go through Steps 8 through 14 in Chapter 13. Just remember to put the keys from the SELECT statement into the data record.

Opening an Indexed File

You can OPEN an indexed file in any of four ways: you can OPEN it for INPUT, OUTPUT, EXTEND, or I-O. If you open the file for OUTPUT, EXTEND, or I-O, you can write to it. If you open it for INPUT or I-O, you can read from it.

Opening a file for INPUT

If you only want to read data from the file, you OPEN it for INPUT. You are able to read from it any way you would like, but you can't write to it. Choosing to OPEN a file this way works just fine with any ACCESS MODE — SEQUENTIAL, RANDOM, and DYNAMIC. If the ACCESS MODE is either RANDOM or DYNAMIC, you can read data based on values in the keys. If the ACCESS MODE is either SEQUENTIAL or DYNAMIC, you can use START to position the file to any record and then read straight through from that record to the end of the file.

Here's how you OPEN a file for INPUT:

```
OPEN INPUT RefName.
```

That's it — no options or anything, just OPEN it and move on. RefName is the name found on the SELECT statement. If you have declared the file as OPTIONAL and the file does not exist, the first READ statement results in an end-of-file notification or an invalid key condition.

Opening a file for OUTPUT

If you just want to WRITE records to a file, you should OPEN the file for OUTPUT. If the file doesn't exist, COBOL creates a new file with the filename that you designate in the OPEN statement. If the file already exists, COBOL overwrites the old file and creates a brand-new file.

Here's how you OPEN a file for OUTPUT:

```
OPEN OUTPUT RefName.
```

You don't have any options to choose when opening a file for OUTPUT. All you can do is WRITE to the file. You have to be careful that you don't try to write duplicate key values for the keys that have not been declared WITH DUPLICATES. But other than that, you can write anything you want to write — the index files keep track of the key values for you.

Opening a file for EXTEND

If you already have a file, and the file already contains some data, and all you want to do is add some new data to it, and the ACCESS MODE is SEQUENTIAL, then open it for EXTEND. Also, if the file doesn't already exist and you want to create it, you need to declare it as OPTIONAL. If you meet all these qualifications, do this:

```
OPEN EXTEND RefName.
```

After you OPEN the file for EXTEND, all you can do is WRITE. You can't position the file in any way. You can't modify any existing records. All you can do is WRITE new records — and you have to be careful about duplicate key values.

Opening a file for I-O

If you want to be able to read and write a file, and have complete, random access to any record in the file, you want to OPEN the file for I-O, like this:

```
OPEN I-O RefName.
```

After you OPEN the file for I-O, you can do just about whatever you need to do. In fact, it is easier to list the things you can't do than to list the things you can do. Here are the things you can't do:

- If the ACCESS MODE is SEQUENTIAL, you can't WRITE new records to the file (although you can REWRITE and DELETE existing records).
- If the ACCESS MODE is RANDOM, you can't START the file at some specific record number to read sequentially.
- If you try to OPEN a file that does not exist, and you didn't declare it as OPTIONAL, the OPEN fails.

If you want to avoid all the limitations — if you want to OPEN an indexed file so you have complete control over it, then do these three things:

- Declare the ACCESS MODE as DYNAMIC.
- Define the file as OPTIONAL on the SELECT statement.
- OPEN the file for I-O.

This sequence is like a magic potion that gives you complete power over the file. You can START, READ, WRITE, REWRITE, and DELETE. And if you try to OPEN a file that doesn't exist, COBOL creates the file.

Closing an Indexed File

When you are finished using a file, you need to CLOSE it. Under almost all circumstances, the statement you use to CLOSE a file is simple and straightforward:

```
CLOSE RefName.
```

After you CLOSE a file, no more activity can occur on that file unless you OPEN it again, which you are allowed to do. Well, you are usually allowed to OPEN it again. If you CLOSE the file with the following statement, you can't OPEN the file again:

```
CLOSE RefName WITH LOCK.
```

The preceding code means that for only *this* run of only *this* program, you cannot OPEN the file again. Any other program can open the file and, if you run this program again, you can open the file again.

Writing to an Indexed File

The program I describe in this section creates an indexed file and writes a herd of records to the file. Each record in the file holds information on one cow. The *primary key* to a cow is her name. The file also has alternate keys that track the pounds of milk that the cow produces and the cow's refresh date. For those of you who are not up on politically correct cow-talk, the *refresh date* is when the cow has a calf, causing her to produce milk. And yes, dairy products — milk, butter, and cheese — are measured in pounds until they get to the store and become quarts, sticks, and slices.

The following code defines cows as an indexed file and writes data to the file. Here's the code for this program:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. IndexedWrite.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT OPTIONAL CowList
        ASSIGN TO "cows"
        ORGANIZATION IS INDEXED
        ACCESS MODE IS RANDOM
        RECORD KEY IS Name
        ALTERNATE RECORD KEY IS RefreshDate
        ALTERNATE RECORD KEY IS MilkPoundsPerDay
            WITH DUPLICATES.
DATA DIVISION.
FILE SECTION.
FD CowList
    RECORD CONTAINS 22 CHARACTERS.
01  CowData.
```

(continued)

(continued)

```
    05  Name                PIC A(8).
    05  RefreshDate.
        10  CC              PIC 99.
        10  YY              PIC 99.
        10  MM              PIC 99.
        10  DD              PIC 99.
    05  MilkPoundsPerDay    PIC 99.
    05  PercentButterfat    PIC 9.9.
    05  Attitude            PIC X.
        88  Contented VALUE "C".
        88  Indifferent VALUE "I".
        88  Mean VALUE "M".
        88  Republican VALUE "R".
        88  Democrat VALUE "D".
        88  Undecided VALUE "U".
WORKING-STORAGE SECTION.
01  InitialValueText.
    02  FILLER PIC X(22) VALUE "Dancer  19970319473.1R".
    02  FILLER PIC X(22) VALUE "Grammy  19961130223.4M".
    02  FILLER PIC X(22) VALUE "Phydeaux19970214324.6C".
    02  FILLER PIC X(22) VALUE "Bessie  19970824382.7R".
    02  FILLER PIC X(22) VALUE "Amelia  19961212404.3D".
    02  FILLER PIC X(22) VALUE "Veronica19970205245.5U".
    02  FILLER PIC X(22) VALUE "EmmyLou 19970806364.2C".
    02  FILLER PIC X(22) VALUE "Mudder  19970727393.9I".
    02  FILLER PIC X(22) VALUE "Dasher  19970911432.8C".
01  InitialValueArray REDEFINES InitialValueText.
    02  ValueArray PIC X(22) OCCURS 9 TIMES.
77  I PIC 9(2).
77  CowFileStatus PIC XX VALUE "00".
PROCEDURE DIVISION.
Mainline.
    OPEN OUTPUT CowList.
    IF CowFileStatus IS NOT EQUAL TO "00"
       DISPLAY "Open failed: " CowFileStatus
       STOP RUN
    END-IF.
    PERFORM VARYING I FROM 1 BY 1
            UNTIL I IS GREATER THAN 9
        MOVE ValueArray(I) TO CowData
        WRITE CowData
            INVALID KEY
                DISPLAY "Invalid key error" CowFileStatus
        END-WRITE
```

```
    END-PERFORM.
    CLOSE CowList.
    STOP RUN.
```

For convenience, the data that the program writes to the file is coded directly in the example program as `InitialValueText`. This data is not in any particular order, but the indexed file wants to have its primary key inserted in order. To do this, the `SELECT` statement defines the `ACCESS MODE` as `RANDOM`. This way, the program is free to write the records in any order, and it's up to COBOL to keep things organized and keep track of the keys.

The program's first task is to `OPEN CowList` for `OUTPUT`; because `CowList` doesn't already exist, this `OPEN` statement creates a new file with nothing in it. The program then executes one `WRITE` statement for each cow. After the program runs, the data and the keys sit on disk in one big cow file.

Notice that the refresh date in the preceding code includes a two-digit `CC` value. This value is the *century number,* which is necessary to have a four-digit date and thus avoid the millennium problem. Even though in this example, the refresh date is likely to be very near (less than a year away), the millennium problem persists. When the year 2000 dawns and all your cows dry up, it won't be witchcraft — it'll be a computer glitch. The point is this: Every date — no matter how trivial it may seem at the moment — must include the complete century number. Doing otherwise is just being short-sighted — and it is shortsightedness that got us into this millennium mess in the first place.

Reading from an Indexed File

If you know the key value, you can get the record you want from an indexed file. The following example reads a record from `CowList` (the file created by the example program in the preceding section of this chapter) by searching for the indexed key value (in this case, the cow's name):

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ReadDancer.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT OPTIONAL CowList
        ASSIGN TO "cows"
        ORGANIZATION IS INDEXED
        ACCESS MODE IS RANDOM
        RECORD KEY IS Name
```

(continued)

(continued)

```
        ALTERNATE RECORD KEY IS RefreshDate
        ALTERNATE RECORD KEY IS MilkPoundsPerDay
           WITH DUPLICATES.
DATA DIVISION.
FILE SECTION.
FD CowList
    RECORD CONTAINS 22 CHARACTERS.
01  CowData.
    05  Name               PIC A(8).
    05  RefreshDate.
        10  CC             PIC 99.
        10  YY             PIC 99.
        10  MM             PIC 99.
        10  DD             PIC 99.
    05  MilkPoundsPerDay   PIC 99.
    05  PercentButterfat   PIC 9.9.
    05  Attitude           PIC X.
        88  Contented VALUE "C".
        88  Indifferent VALUE "I".
        88  Mean VALUE "M".
        88  Republican VALUE "R".
        88  Democrat VALUE "D".
        88  Undecided VALUE "U".
WORKING-STORAGE SECTION.
77  CowFileStatus PIC XX VALUE "00".
PROCEDURE DIVISION.
Mainline.
    OPEN INPUT CowList.
    IF CowFileStatus IS NOT EQUAL TO "00"
       DISPLAY "Open failed: " CowFileStatus
       STOP RUN
    END-IF.
    MOVE "Dancer" TO Name OF CowData.
    READ CowList
        KEY IS Name OF CowData
        INVALID KEY
            DISPLAY "Invalid key"
        NOT INVALID KEY
            DISPLAY CowData
    END-READ.
    CLOSE CowList.
    STOP RUN.
```

This example reads from the file that I describe in the preceding section of this chapter. Notice that the SELECT statement in this program is identical to the one in the previous program. Furthermore, the 01 level that defines the record layout is also identical. These points are important because the file has been created and resides out there on disk, and it has definite opinions about the size of its records and the placement and size of its keys.

If you create an indexed file in one program and then read that file in another program, keeping track of the record layouts and keys is very important. If the definition of the file in your program doesn't match the actual file on disk, you won't be able to read the file. If you need to write a program that reads data from an indexed file, and you don't have all the information about the layout of the file, find someone and ask, "Can I have the keys to the cow?" About the only practical thing you can do is copy the information from another program that uses the file — preferably from the program that originally created the file because, that way, you have less chance of making an error.

The program defines the file with its access as RANDOM, which allows reading on keys, and an OPEN statement indicates that the program wants to access the file for INPUT. A MOVE statement puts the name of the cow into the key field, so the READ statement can find the desired record. The KEY clause tells the READ statement which key fields to use (actually, READ being rather simple-minded, the KEY clause needs to tell the READ statement which key field to use even if you define only one key for the file). The READ fails udderly for an unknown cow and displays the INVALID KEY message. In this example, however, the cow is known, so the displayed line looks like this:

```
Dancer   19970319473.1R
```

Reading from a Specific Starting Point in an Indexed File

If you use START on one of the keys in an indexed file, it's as if the entire file lines up in a row behind the first record. Then, to get all the records sorted according to the key you named in the START statement, all you need to do is READ sequentially to the end of the file.

This capability — to read data from a file based on a key — could be the centerpiece of COBOL. If it isn't the most important capability of the language, it's in the top three. As the following example demonstrates, this capability makes it very easy to get all your ducks (and cows) in a row:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CowsInOrder.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT OPTIONAL CowList
        ASSIGN TO "cows"
        ORGANIZATION IS INDEXED
        ACCESS MODE IS DYNAMIC
        RECORD KEY IS Name
        ALTERNATE RECORD KEY IS RefreshDate
        ALTERNATE RECORD KEY IS MilkPoundsPerDay
            WITH DUPLICATES.
DATA DIVISION.
FILE SECTION.
FD CowList
    RECORD CONTAINS 22 CHARACTERS.
01  CowData.
    05  Name               PIC A(8).
    05  RefreshDate.
        10  CC             PIC 99.
        10  YY             PIC 99.
        10  MM             PIC 99.
        10  DD             PIC 99.
    05  MilkPoundsPerDay   PIC 99.
    05  PercentButterfat   PIC 9.9.
    05  Attitude           PIC X.
        88  Contented VALUE "C".
        88  Indifferent VALUE "I".
        88  Mean VALUE "M".
        88  Republican VALUE "R".
        88  Democrat VALUE "D".
        88  Undecided VALUE "U".
WORKING-STORAGE SECTION.
77  CowFileStatus PIC XX VALUE "00".
01  CowCatcher PIC X.
    88  LastCow VALUE "L".
01  FormatDate.
    02  MM PIC Z9.
    02  FILLER PIC X VALUE "/".
    02  DD PIC Z9.
    02  FILLER PIC X VALUE "/".
    02  CC PIC 99.
    02  YY PIC 99.
PROCEDURE DIVISION.
```

```
Mainline.
    OPEN INPUT CowList.
    IF CowFileStatus IS NOT EQUAL TO "00"
       DISPLAY "Open failed: " CowFileStatus
       STOP RUN
    END-IF.
    PERFORM ShowByName.
    PERFORM ShowByRefreshDate.
    PERFORM ShowByPoundsOfMilk.
    CLOSE CowList.
    STOP RUN.

ShowByName.
    DISPLAY " * * * Sort by name * * *".
    MOVE SPACES TO CowData.
    START CowList
        KEY IS GREATER THAN Name OF CowData
        INVALID KEY
            DISPLAY "Invalid key on the START"
    END-START.
    MOVE SPACE TO CowCatcher.
    PERFORM UNTIL LastCow
        READ CowList
            AT END
                MOVE "L" TO CowCatcher
            NOT AT END
                PERFORM ShowCow
        END-READ
    END-PERFORM.

ShowByRefreshDate.
    DISPLAY " * * * Sort by refresh date * * *".
    MOVE ALL ZEROES TO RefreshDate.
    START CowList
        KEY IS GREATER THAN RefreshDate OF CowData
        INVALID KEY
            DISPLAY "Invalid key on the START"
    END-START.
    MOVE SPACE TO CowCatcher.
    PERFORM UNTIL LastCow
        READ CowList
            AT END
                MOVE "L" TO CowCatcher
            NOT AT END
                PERFORM ShowCow
```

(continued)

(continued)

```
        END-READ
    END-PERFORM.
    CLOSE CowList.

ShowByPoundsOfMilk.
    DISPLAY " * * * Sort by pounds of milk * * *".
    MOVE 35 TO MilkPoundsPerDay.
    START CowList
        KEY IS GREATER THAN MilkPoundsPerDay OF CowData
        INVALID KEY
            DISPLAY "Invalid key on the START"
    END-START.
    MOVE SPACE TO CowCatcher.
    PERFORM UNTIL LastCow
        READ CowList
            AT END
                MOVE "L" TO CowCatcher
            NOT AT END
                PERFORM ShowCow
        END-READ
    END-PERFORM.

ShowCow.
    MOVE CORRESPONDING RefreshDate TO FormatDate.
    DISPLAY Name " " FormatDate " "
        MilkPoundsPerDay " pounds of milk at "
        PercentButterfat "% butterfat".
```

The three paragraphs (`ShowByName`, `ShowByRefreshDate`, and `ShowByPoundsOfMilk`) each read and display the data in a different order. Each of these paragraphs reads data from the file using a different key.

The paragraph `ShowByName` uses `START` to put all the cows in line. This paragraph sets the position of the file to the record with the lowest primary key value, and positions all the other records, in order, behind the first one. The program positions the records in order by sticking `SPACES` into `CowData` and using the `START` verb to tell COBOL, "Using the `Name` field of `CowData` as the key field, start with the value it contains, line up every record you have, and get ready for `READ`." This example uses `SPACES` as the key value to be `GREATER THAN`, so that every record in the file qualifies and gets in line. All your cows are now standing there looking over each other's shoulders. It's time to milk the file for all it's worth.

The `READ` doesn't use any kind of key. It doesn't have to — all the cows are organized into a sequential line by `START`. The `READ` is sequential. Each record, one after the other, comes ambling in from the file and is put on display by the `ShowCow` paragraph.

The paragraph ShowByRefreshDate does about the same thing as ShowByName does, except that it uses a different key and places the cows in a different order. This time, the key — and thus the sorted order — is on the RefreshDate. I could use SPACES as the START key value, just like I did in ShowByName, but because all the parts of RefreshDate are PIC 99, I thought it would be more polite to use ZEROES. Lots of people use LOW-VALUES when they want to make sure the key value is less than any actual key in file. After the START verb completes its chore, the READ verb does its sequential thing and, once again, here come all the cows — this time, in the order of calving.

Finally, using the same mechanism as ShowByRefreshDate, the paragraph ShowByPoundsOfMilk uses the third key to list the cows in the order of milk production. One difference exists, though. The key value is set to 35 and the START verb is used to select key values that are GREATER THAN the starting value. The READ does not begin with the first record this time — READ skips all those records with key values of 35 or less.

The output from the program looks like this:

```
 * * * Sort by name * * *
Amelia   12/12/1996 40 pounds of milk at 4.3% butterfat
Bessie    8/24/1997 38 pounds of milk at 2.7% butterfat
Dancer    3/19/1997 47 pounds of milk at 3.1% butterfat
Dasher    9/11/1997 43 pounds of milk at 2.8% butterfat
EmmyLou   8/ 6/1997 36 pounds of milk at 4.2% butterfat
Grammy   11/30/1996 22 pounds of milk at 3.4% butterfat
Mudder    7/27/1997 39 pounds of milk at 3.9% butterfat
Phydeaux  2/14/1997 32 pounds of milk at 4.6% butterfat
Veronica  2/ 5/1997 24 pounds of milk at 5.5% butterfat
 * * * Sort by refresh date * * *
Grammy   11/30/1996 22 pounds of milk at 3.4% butterfat
Amelia   12/12/1996 40 pounds of milk at 4.3% butterfat
Veronica  2/ 5/1997 24 pounds of milk at 5.5% butterfat
Phydeaux  2/14/1997 32 pounds of milk at 4.6% butterfat
Dancer    3/19/1997 47 pounds of milk at 3.1% butterfat
Mudder    7/27/1997 39 pounds of milk at 3.9% butterfat
EmmyLou   8/ 6/1997 36 pounds of milk at 4.2% butterfat
Bessie    8/24/1997 38 pounds of milk at 2.7% butterfat
Dasher    9/11/1997 43 pounds of milk at 2.8% butterfat
 * * * Sort by pounds of milk * * *
EmmyLou   8/ 6/1997 36 pounds of milk at 4.2% butterfat
Bessie    8/24/1997 38 pounds of milk at 2.7% butterfat
Mudder    7/27/1997 39 pounds of milk at 3.9% butterfat
Amelia   12/12/1996 40 pounds of milk at 4.3% butterfat
Dasher    9/11/1997 43 pounds of milk at 2.8% butterfat
Dancer    3/19/1997 47 pounds of milk at 3.1% butterfat
```

Good ol' DYNAMIC ACCESS lets you open a file one time and do just about anything you want with it. In this example, the program asks to OPEN the file only once — because the file is INDEXED and the ACCESS MODE is DYNAMIC, that's enough. Each START command repositions the file based on a different key, almost as if the program had asked to OPEN the file from scratch before each READ — the file is always ready to go.

The setting of the KEY on the START command is kind of flexible. In every case, you shove a value into the location of the key you are going to use as the starting point, and then you tell START how you want to use the value to position the file. The following code shows the possible options that the example could have used:

```
KEY IS EQUAL TO RecordNumber.
KEY IS = RecordNumber.
KEY IS GREATER THAN RecordNumber.
KEY IS > RecordNumber.
KEY IS NOT LESS THAN RecordNumber.
KEY IS NOT < RecordNumber.
KEY IS GREATER THAN OR EQUAL TO RecordNumber.
KEY IS >= RecordNumber.
```

These examples all do pretty much the same thing — each one enables you to eliminate all records with key values less than some specified value and read from there to the end of file. Even the one that uses the expression IS EQUAL TO really means IS EQUAL TO ALONG WITH ALL THAT ARE GREATER THAN. All of these examples simply define the starting point for a sequential READ. If you want to skip the files with key values larger than a certain amount, you have to roll your own — just quit reading when a key value gets too large for you. For example, here's how you read all the rec-ords that show cows that produce between 30 and 40 pounds of milk per day:

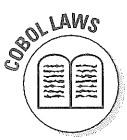
```
MOVE 30 TO MilkPoundsPerDay.
START CowList
    KEY IS >= MilkPoundsPerDay OF CowData
    INVALID KEY
        DISPLAY "Invalid key on the START"
END-START.
MOVE SPACE TO CowCatcher.
PERFORM UNTIL LastCow
    READ CowList
        AT END
            MOVE "L" TO CowCatcher
        NOT AT END
            IF MilkPoundsPerDay > 40
```

```
                NEXT SENTENCE
            ELSE
                PERFORM ShowCow
            END-IF
        END-READ
    END-PERFORM.
```

The START verb lines up the cows beginning with the first one that produces 30 pounds of milk or better. After each READ, an IF statement makes a test to see if the value has gone beyond 40, and if it has, processing goes to the NEXT SENTENCE (which means COBOL skips forward to the first thing it comes to after the next period, thus bringing a halt to the reading of records). The output looks like this:

```
Phydeaux  2/14/1997 32 pounds of milk at 4.6% butterfat
EmmyLou   8/ 6/1997 36 pounds of milk at 4.2% butterfat
Bessie    8/24/1997 38 pounds of milk at 2.7% butterfat
Mudder    7/27/1997 39 pounds of milk at 3.9% butterfat
Amelia   12/12/1996 40 pounds of milk at 4.3% butterfat
```

Rewriting a Record in an Indexed File



COBOL uses the REWRITE verb to make modifications to the current record. You READ the record, make the changes to it, and REWRITE it back to the file.

To get COBOL to obey your REWRITE instructions, you must do certain things:

- To perform a REWRITE, you must OPEN the file for I-O.
- The REWRITE statement must immediately follow a successful READ (which means that you can't perform a second REWRITE after a single READ statement). If you do a REWRITE, you have to do another READ before you can REWRITE again.
- You cannot change the value of the primary key with REWRITE. The only way to change the primary key value is to DELETE the record and WRITE it again with a new key value.
- You can change the values of the ALTERNATE keys with a REWRITE statement.

Here is an example that changes one of the ALTERNATE key values for a cow:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Recow.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT OPTIONAL CowList
        ASSIGN TO "cows"
        ORGANIZATION IS INDEXED
        ACCESS MODE IS DYNAMIC
        RECORD KEY IS Name
        ALTERNATE RECORD KEY IS RefreshDate
        ALTERNATE RECORD KEY IS MilkPoundsPerDay
            WITH DUPLICATES.
DATA DIVISION.
FILE SECTION.
FD CowList
    RECORD CONTAINS 22 CHARACTERS.
01  CowData.
    05  Name                PIC A(8).
    05  RefreshDate.
        10  CC               PIC 99.
        10  YY               PIC 99.
        10  MM               PIC 99.
        10  DD               PIC 99.
    05  MilkPoundsPerDay    PIC 99.
    05  PercentButterfat    PIC 9.9.
    05  Attitude            PIC X.
        88  Contented VALUE "C".
        88  Indifferent VALUE "I".
        88  Mean VALUE "M".
        88  Republican VALUE "R".
        88  Democrat VALUE "D".
        88  Undecided VALUE "U".
WORKING-STORAGE SECTION.
77  CowFileStatus PIC XX VALUE "00".
01  CowCatcher PIC X.
    88  LastCow VALUE "L".
01  FormatDate.
    02  MM PIC Z9.
    02  FILLER PIC X VALUE "/".
    02  DD PIC Z9.
    02  FILLER PIC X VALUE "/".
    02  CC PIC 99.
    02  YY PIC 99.
PROCEDURE DIVISION.
```

```
Mainline.
    OPEN I-O CowList.
    IF CowFileStatus IS NOT EQUAL TO "00"
       DISPLAY "Open failed: " CowFileStatus
       STOP RUN
    END-IF.
    MOVE "Phydeaux" TO Name OF CowData.
    READ CowList
        KEY IS Name OF CowData
        INVALID KEY
           DISPLAY "Invalid key on READ"
        NOT INVALID KEY
           PERFORM ShowCow
    END-READ.
    MOVE 29 TO MilkPoundsPerDay OF CowData.
    REWRITE CowData
        INVALID KEY
           DISPLAY "Invalid key on REWRITE"
        NOT INVALID KEY
           PERFORM ShowCow
    END-REWRITE.
    CLOSE CowList.
    STOP RUN.

ShowCow.
    MOVE CORRESPONDING RefreshDate TO FormatDate.
    DISPLAY Name " " FormatDate " "
        MilkPoundsPerDay " pounds of milk at "
        PercentButterfat "% butterfat".
```

If you follow the basic laws of COBOL, doing a `REWRITE` is quite easy. In the preceding example, the program first asks to `OPEN` the file for `I-O`; then a record is read on the primary key. It doesn't matter how the record is actually read — you can read on an alternate key or even read sequentially.

After the `READ` does its thing (and the data from the disk is now sitting inside the program), just change anything you want — except, of course, the primary key. In this example, the alternate key value of `MilkPoundsPer` day is changed to 29, and `REWRITE` replaces the existing record in the file with the new one. The output from the example — a copy of the data before and after the change — looks like this:

```
Phydeaux  2/14/1997 32 pounds of milk at 4.6% butterfat
Phydeaux  2/14/1997 29 pounds of milk at 4.6% butterfat
```

That's it — new data is shoved onto the disk in place of the old data. It's that simple. If there were any more to it, I would say some more. There isn't, so I won't.

Deleting a Record from an Indexed File

There comes a time in the life of every cow when she must be removed from the file. A cow could be beyond her prime, or could have moved on to the next ranch. And there are always rustlers. The COBOL DELETE verb is always standing by to remove a record from a file.

Here's an example of a program that removes a cow from the file:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Decow.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT OPTIONAL CowList
        ASSIGN TO "cows"
        ORGANIZATION IS INDEXED
        ACCESS MODE IS DYNAMIC
        RECORD KEY IS Name
        ALTERNATE RECORD KEY IS RefreshDate
        ALTERNATE RECORD KEY IS MilkPoundsPerDay
            WITH DUPLICATES.
DATA DIVISION.
FILE SECTION.
FD CowList
    RECORD CONTAINS 22 CHARACTERS.
01  CowData.
    05  Name               PIC A(8).
    05  RefreshDate.
        10  CC             PIC 99.
        10  YY             PIC 99.
        10  MM             PIC 99.
        10  DD             PIC 99.
    05  MilkPoundsPerDay   PIC 99.
    05  PercentButterfat   PIC 9.9.
    05  Attitude           PIC X.
        88  Contented VALUE "C".
        88  Indifferent VALUE "I".
```

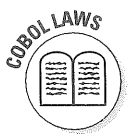
```
        88  Mean VALUE "M".
        88  Republican VALUE "R".
        88  Democrat VALUE "D".
        88  Undecided VALUE "U".
WORKING-STORAGE SECTION.
77  CowFileStatus PIC XX VALUE "00".
01  CowCatcher PIC X.
    88  LastCow VALUE "L".
01  FormatDate.
    02  MM PIC Z9.
    02  FILLER PIC X VALUE "/".
    02  DD PIC Z9.
    02  FILLER PIC X VALUE "/".
    02  CC PIC 99.
    02  YY PIC 99.
PROCEDURE DIVISION.
Mainline.
    OPEN I-O CowList.
    IF CowFileStatus IS NOT EQUAL TO "00"
       DISPLAY "Open failed: " CowFileStatus
       STOP RUN
    END-IF.
    MOVE "Phydeaux" TO Name OF CowData.
    READ CowList
        KEY IS Name OF CowData
        INVALID KEY
            DISPLAY "Invalid key on READ"
    END-READ.
    DELETE CowList
        INVALID KEY
            DISPLAY "Invalid key on DELETE"
    END-DELETE.
    CLOSE CowList.
    STOP RUN.
```

Step 1: READ the record. Step 2: DELETE the record. As you can see, removing a record is just about the easiest thing that you can do with an indexed file. What could be easier? I'll tell you what — you don't even have to READ the record first. Just MOVE the primary key value into the record and yell DELETE at it.

The following code — an alternate version of the PROCEDURE DIVISION for this example — shows you how to perform this shortcut when you want to delete a record:

```
PROCEDURE DIVISION.
Mainline.
    OPEN I-O CowList.
    IF CowFileStatus IS NOT EQUAL TO "00"
       DISPLAY "Open failed: " CowFileStatus
       STOP RUN
    END-IF.
    MOVE "Phydeaux" TO Name OF CowData.
    DELETE CowList
        INVALID KEY
            DISPLAY "Invalid key on DELETE"
    END-DELETE.
    CLOSE CowList.
    STOP RUN.
```

The MOVE verb sets the value of the primary key. The DELETE verb takes the record, looks for a match on the primary key, and deletes a record that matches it. The truth is, the only reason you need to bother with reading a record is to find out the value of the primary key. You just name the cow and the DELETE verb cuts her right out of the herd.



If you read the record with ACCESS SEQUENTIAL, you can neither specify INVALID KEY nor NOT INVALID KEY on the DELETE statement. On the other hand, if you aren't reading with ACCESS SEQUENTIAL, you are required to put in something that handles the INVALID KEY situation. You can put any kind of COBOL statement you would like on the INVALID KEY clause. I know these are strict rules, but they exist for your own good.

Chapter 16

Using SORT and MERGE

In This Chapter

- Defining a sort file description
- Sorting data that comes from a file
- Sorting data that comes from an internal procedure
- Sorting data and writing the results to a file
- Sorting data and sending the results to an internal procedure
- Merging several sorted files into one sorted file

Humans have this overwhelming urge to organize things. Well, most humans, anyway. Although some adults and most teenagers are organizationally challenged, for the most part the human brain seems to work better with things that are arranged in a recognizable pattern. I'll never forget the day I discovered that the dictionary is in alphabetical order — this discovery speeded me up no end.

One of the primary purposes of computing is to organize data. I have seen estimates that as much as 80 percent of all computing time is spent in sorting. I don't know whether those estimates include time spent crashing and rebooting.

When you sort data records, sometimes you sort them with little ones first and the big ones last, and sometimes you sort them the other way around. If you sort your data records and then format them for print on a piece of paper (or show them on the screen), a human can almost immediately see the pattern you created. All you have to do is take one quick glance at ordered data, see how it is organized, and go directly to the data record you're looking for.

This chapter describes the mechanics that COBOL uses to get data in order. A COBOL program can be made to take the input data records (retrieved from a file or from an internal procedure) and, by using a temporary work file, sort the records to the output (either another file or an internal procedure).

SORT and MERGE Work Together

Everybody knows what sorting is. We've all done it. Imagine a file drawer filled with manila folders, each folder with a label on it. You can look at the labels and move the folders around until they are all in alphabetical order. That's sorting. Sorting takes some brain power — you have to come up with a process by which you move one folder at a time, and each time you move one of the folders, you move one step closer to having them in the right order.

Computers don't have one particular way to sort — in fact, enough different sorting methods exist that every COBOL compiler could use a different one. Nobody has discovered the perfect sorting technique yet — but many have tried.

Sorting out sorting and merging

Computers spend lots of time sorting and merging, and the various sorting algorithms have been the focus of extensive studies. These studies involve lots of really weird math, but their key findings come down to this point: You can speed up the sorting process if you can (1) reduce the number of times required to compare two records, and (2) reduce the number of times you have to move a record to a new location. At least a bazillion algorithms exist for sorting.

No sorting algorithm is as efficient as merging. When you merge two files, each record is moved only once. Every time a comparison is made, a record is sent directly to its final resting place. The only drawback to merging is that the records to be merged must arrive in sorted order. But because the need to combine two sorted files arises pretty often, merging is a handy thing to have available.

You can use merging as an aid to help with sorting when you need to sort a large number of records — and situations can arise in which you need to sort millions of records. If you have to do a huge sort, try a *sort-merge sequence*. First, pick a file size (say, 10,000 records) and start reading your input data. Each time you have 10,000 records, put them into a sorted file. Then create a new file, and do it again. After you have processed all the data, you have a bunch of files that contain sorted records. You can merge these files quickly into one large sorted file. If you have too many files to merge them all at once, start merging them into larger and larger sorted files until you wind up with one huge file.

This process may seem counterintuitive at first. Sorting records into multiple files and then merging them together may seem like more work rather than less, but with a large number of records, it is actually a savings. If you try to handle hundreds of thousands of records as one large sort, you could wind up making thousands of comparisons on each record and move each record from one place to another thousands of times. The savings from doing a sort-merge can be dramatic — I worked on a project in which the implementation of a multiple-file sort-merge reduced the execution time from 36 hours to 90 minutes.

It is only fair to tell you that other solutions exist for sorting large files. Many computer systems have built-in sort capabilities that can really speed things up. Also, some companies specialize in software that will do the sorting for you. You're not alone out there.

Merging is something of a first cousin to sorting. In fact, you can think of merging as a form of sorting that works only when certain pre-existing conditions are met. To understand how merging works, imagine two file drawers that are already in sorted order. Between the two sorted file drawers is an empty file drawer. The two full drawers are the sources of input data and the empty drawer is to hold the output data — that is, the merged files.

To start merging the files, you grab the first folder from each of the full file drawers, look at the two folders to decide which one should go first, and put it in the front of the empty drawer. You then grab the next folder from whichever file drawer supplied the one you put into the empty drawer. Again, you compare the two folders and put one in the output drawer. You continue this process until you transfer all the folders from the input file drawers to the output file drawer. This process creates a new file drawer that holds all the folders from the original two, and, if you made no mistakes, it is in sorted order. You have just merged.

Creating a Sort File Definition

Sorting is not only hard, it also takes up lots of space. To sort your records, the SORT verb scatters them all over the floor and the tables of a work file. You, being the benevolent and helpful programmer that you are, will gladly supply the floor and the table space for SORT to do its work. And of course, this being COBOL, you need to jump through a few hoops to set up the file that SORT uses for doing its job. You set up this sort file in much the same way as you declare a sequential file — except it's different. (See Chapter 13 for all the details about declaring a sequential file.) They are just similar enough to look alike, but different enough to be deceptive.

In the following sections, I describe a step-by-step process that you can follow to write the code that defines a file that the SORT verb uses to add order to your chaotic world. All you need to do is go through these steps and put in all the required parts, along with the optional parts you want to have, and then you are ready to sort.

Step 1: SELECT your sort file

You use a SELECT statement to give the sort file its name. You put the SELECT statement in the FILE-CONTROL paragraph, like this:

```
ENVIRONMENT DIVISION.
```

```
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT RefName ASSIGN TO "ActualName".
```

You define the names of the sort file here — the *RefName* that you will be calling it inside your program and the *ActualName* that will be its real name out there on disk. That's all you can do for a sort file — you don't have any fancy options like the ones for other kinds of files. If you want to, you can use the same name for both the internal and the external name, but that is such an organized and logical thing to do that almost nobody does it.

Step 2: Decide whether to put several sort files in the SAME space

If you are going to have lots of sort files, and you are not going to be using them all at once, and you know which ones are not going to be open at the same time, you can tell COBOL about all this and it shares the space that the sort files use. You specify that several sort files can share the same work area by putting the SAME clause in the I-O CONTROL paragraph, like this:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
I-O CONTROL.
    SAME SORT AREA FOR ThisFile ThatFile.
```

You can specify a bunch of sort files to share the same work area. In fact, they don't all have to be sort files — if at least one of them is a sort file, the others can be sequential, relative, or indexed files. Using the reserved word SORT-MERGE is exactly the same as using SORT.

You can also use SAME RECORD AREA to have files share the data record location. You can find more information on this capability in Chapter 13, in the description of sequential files.

This sharing option is an efficiency issue and really doesn't do much of anything on modern computers. It is designed to save space in memory, which is not nearly as precious as it once was. Not only that, but modern operating systems automatically do this sharing stuff. Unless you are obsessive, just skip to the next step.

Step 3: Define the record layout for the sort file

The FILE SECTION holds the details of the sort file's record layout. It does this with two things: the sort description and the record description. The

sort description is known to its friends as "ess-dee" because its keyword is SD. The *record description* is simply an 01-level entry defining all the fields that make up the record. The 01 level is associated with the SD level by following right behind it, like this:

```
DATA DIVISION.
FILE SECTION.
SD  RefName . . .
01  SortFileDataRecord . . .
```

The *RefName* is the same as the one you define on the SELECT statement.

Specifying RECORD CONTAINS is optional because the record description following the SD statement determines the record size, but you normally include a RECORD statement, if for nothing other than documentation. The simplest form of a RECORD statement is for a file that has all fixed-size records, like this example:

```
SD  RefName
    RECORD CONTAINS 84 CHARACTERS.
```

No matter what else happens in your life, if you put this statement in your code, every record in the file will have exactly 84 characters. On the other hand, if you want to vary the size as you go — if you have this urge to sort things even though some of them are bigger than others — if you need to allow for variable-length records, COBOL gives you a way. You just have to let COBOL know the minimum and maximum sizes, as in the following example:

```
SD  RefName
    RECORD CONTAINS 16 TO 96 CHARACTERS.
```

With this statement, you can sort records as small as 16 characters and as large as 96 characters. You can define the upper and lower size limits to be anything you would like, but after you define them, your program is limited to that range. It is now up to you to specify the size of every record you sort. COBOL is picky about one thing — the smallest record must completely contain whatever field you decided to use for the sort key.

For you complete control freaks who want to be in control of every sorted detail, take a look at this definition:

```
SD  RefName
    RECORD IS VARYING IN SIZE FROM 16 TO 96 CHARACTERS
        DEPENDING ON RefSizeValue.
```

This definition not only allows you to sort records that vary in size from 16 to 96 characters, but it also lets you dictate the exact size of each and every record. All you have to do is stuff some number into a variable — in this example, it is `RefSizeValue` — and records of that exact size are thrown into the sort. If you have some records of another size, just change `RefSizeValue` and throw them in. This technique can be useful when the data to be sorted is coming from a file that contains variable-length records — it's a simple matter of reading each record from the file, storing its size as the value in `RefSizeValue`, and passing the record on to be sorted.

You can define the minimum and maximum sort record sizes in yet another way. You can use multiple record definitions, and the smallest one and the largest one define the minimum and maximum record sizes. You don't have to mention the sizes anywhere. For example, you can do this:

```
SD  RefName.
01  Record1 PIC X(16).
01  Record2 PIC X(96).
```

The preceding definition implies that the record size ranges from 16 to 96. The example before that uses a variable that can be set to determine the record size. You can combine the two techniques and use them both at once, like this:

```
SD  RefName
    RECORD IS VARYING DEPENDING ON RefSizeValue.
01  Record1 PIC X(16).
01  Record2 PIC X(96).
```

Sorting One File into Another

This section presents an example program that sorts records by sorting the value of a key in ascending order. The program reads the data from one file, uses a second file to do the sorting, and then writes the result to a third file. Here's the source code for this sample program:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SimpleSort.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT SortWorkFile ASSIGN TO "sortfile".
```

```
    SELECT InFile ASSIGN TO "infile".
    SELECT OutFile ASSIGN TO "outfile".
DATA DIVISION.
FILE SECTION.
SD  SortWorkFile
    RECORD CONTAINS 12 CHARACTERS.
01  WorkWidgets.
    02  Color PIC X(6).
    02  Height PIC 9(2).
    02  Weight PIC 9(2).
    02  IQ PIC 9(2).
FD  InFile
    RECORD CONTAINS 12 CHARACTERS.
01  InWidgets.
    02  Color PIC X(6).
    02  Height PIC 9(2).
    02  Weight PIC 9(2).
    02  IQ PIC 9(2).
FD  OutFile
    RECORD CONTAINS 12 CHARACTERS.
01  OutWidgets.
    02  Color PIC X(6).
    02  Height PIC 9(2).
    02  Weight PIC 9(2).
    02  IQ PIC 9(2).
WORKING-STORAGE SECTION.
PROCEDURE DIVISION.
Mainline.
    SORT SortWorkFile
        ON ASCENDING KEY Color OF WorkWidgets
        USING InFile
        GIVING OutFile.
    STOP RUN.
```

This example shows one instance in which COBOL steps out of character: Where COBOL is normally verbose and wordy, here it is terse and efficient. Sorting a file just doesn't require much code.

The input and output files to `SORT` can be of any type, but for this example I use sequential files. (For more information on sequential files, see Chapter 13.) This program specifies the input file on an `FD` statement, specifies the output file with another `FD` statement, and then gives COBOL a work file with an `SD` statement. Then with one simple `SORT` statement, the program aims the power of the internal sort engine at these three files, and COBOL takes charge of things. `InFile` contains the following data, which it supplies to the program:

```
Blue   341844
Red    712582
Orange843924
Green  130214
Puce   645285
Aqua   672405
Bondo  572041
```

The program produces a file that contains this output:

```
Aqua   672405
Blue   341844
Bondo  572041
Green  130214
Orange843924
Puce   645285
Red    712582
```

The program chooses this particular ordering because of the `ON ASCENDING KEY` phrase in the `SORT` statement. The program could just as easily have sorted things the other way around. To reverse the direction of the sort, you just change the word `ASCENDING` to `DESCENDING`, like this:

```
ON DESCENDING KEY Color OF WorkWidgets
```

With this change, the order of the output file looks like this:

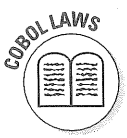
```
Red    712582
Puce   645285
Orange843924
Green  130214
Bondo  572041
Blue   341844
Aqua   672405
```

You can pick any field in the record to act as the key for sorting. For example, this `KEY` statement specifies the `Height` field as the key for sorting the file:

```
ON ASCENDING KEY Height OF WorkWidgets
```

This statement causes the `SORT` to completely ignore the color of widget and sort by the height (which is the first two numbers in each record). The resulting output file holds the records in this order:

```
Green 130214
Blue  341844
Bondo 572041
Puce  645285
Aqua  672405
Red   712582
Orange843924
```



The COBOL SORT verb has some rules and some peculiarities:

- The input and output files must be FD files (sequential, relative, or indexed files) — neither one can be an SD file.
- You can have as many keys as you like — just list them left to right in decreasing order of importance. The left-most name is the primary key, the next name is the secondary key, and so on.
- If the SD statement has more than one record description associated with it, a key need only be defined for one of them. While you define the keys in the code by specifying the name of a field, COBOL translates that definition into a location in the record that spans a certain number of characters — this internal key definition is applied to all records in the sort file.
- You can't reference a subscripted member of an OCCURS clause as the key. (I discuss OCCURS in Chapter 7.)
- The record sizes (or size ranges) of the input, output, and sort files all must be compatible. That is, the sort file must be able to hold any record from the input file, and the output file must be able to hold any record from the sort file. The minimum record size of all three of them must be large enough to include all key fields.

Making the collating go like you want

If you only sort numbers, or stuff that is all uppercase letters, or all lowercase letters, just skip this section. If, however, you are a mixed-case kind of person, you may want to read this section. The bad news is that COBOL may not sort your data in the way that you want. The good news is that you can do something about it.

To demonstrate the problem you face in sorting upper- and lowercase letters, the following code shows an input file that I fed to the SimpleSort example program, which does a neat job of putting the colors in order in the preceding section of this chapter:

Ask me about ASCII

I don't know how to break this to you except just to be blunt. The fact is, a computer doesn't really store information as letters and digits. It stores data as numbers that *represent* the letters and digits.

You remember when you were a kid and you came up with a secret code that used a number in place of each letter — 1 for A, 2 for B, and so on? Well, because computers can only hold numbers, the computer industry has agreed on a code that is used to represent letters inside a computer. The truth is, we have all agreed on three codes:

- **ASCII:** Stands for American Standard Code for Information Interchange
- **EBCDIC:** Stands for Extended Binary Coded Decimal Interchange Code
- **Unicode:** Just a cute name that doesn't really stand for anything

No matter whether your COBOL compiler uses ASCII, EBCDIC, or Unicode, all three codes have the same problem. These codes all sort data by the numbers assigned to the letters. For one thing, this feature causes all the lowercase letters to wad up in one place and the uppercase letters to gather up in another. This happens because of the numbers chosen to represent the letters. For example, the ASCII codes for A through Z are 65 through 90 and the codes for a through z are 97 through 122. Consequently, the uppercase letters always come before the lowercase letters.

```
Blue   341844
Red    712582
orange843924
Green  130214
Puce   645285
aqua   672405
Bondo  572041
```

The only difference between this input file and the one in the preceding section of this chapter is that a couple of these color names start with lowercase letters. Running the new input file through the program produces the following output:

```
Blue   341844
Bondo  572041
Green  130214
Puce   645285
Red    712582
aqua   672405
orange843924
```

The uppercase letters are all sorted before the lowercase letters. If this sort order is what you want, fine; if not, you have a bit of work to do. The good news is that I don't have any more bad news.

By making two additions to the previous example, you can cause the sorting to be done in alphabetical order. First, you need to define an alphabet in the SPECIAL-NAMES paragraph, like this:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. AlphabetSort.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    ALPHABET MyOrder IS
     "AaBbCcDdEeFfGgHhIiJjKkLlMm
-    "NnOoPpQqRrSsTtUuVvWwXxYyZz".
```

I call the alphabet MyOrder because the program uses the order of the characters in the alphabet to control the collation of the sort in just the way I want. It kinda looks like a big kid and a little kid reciting the alphabet at the same time — the upper- and lowercase letters are intermingled — but that's the way I want them to sort.

Note: The definition of the alphabet in the preceding code brings up a special situation. The quoted string is long enough that it needs to be split across two lines of COBOL. As I discuss in Chapter 3, you use a special COBOL method for continuing lines that break a quoted string. The string on the line to be continued does not have an ending quote, but the beginning of the one on the continuation line has a beginning quote. Also, you must use a continuation character (-) used as the control character in column 7 of the continuation line.

On the SORT verb, using the COLLATING SEQUENCE phrase, I tell COBOL to sort the alphabet according to MyOrder — first sorting the colors that begin with an uppercase *A*, then those that begin with a lowercase *a*, uppercase *B*, lowercase *b*, and so on. With the new set of options, the SORT verb looks like this:

```
    SORT SortWorkFile
        ON ASCENDING KEY Color OF WorkWidgets
        COLLATING SEQUENCE IS MyOrder
        USING InFile
        GIVING OutFile.
    STOP RUN.
```

SORT assumes that the alphabet referenced on the COLLATING SEQUENCE phrase is the order of the letters you want to use for comparing upper- and lowercase letters. If this example gives you a feeling of power over SORT, you are right to feel that way. You can actually put the letters in any order you want and the SORT verb obeys your every whim.

Here's the order in which this version of the program sorts the records:

```
aqua  672405
Blue  341844
Bondo 572041
Green 130214
orange843924
Puce  645285
Red   712582
```

This is a more human-like alphabetical order. Instead of COBOL first displaying all records that begin with uppercase letters, and then displaying the records that begin with lowercase letters, the output now shows all records in alphabetical order.

Sorting with DUPLICATES

From time to time, it happens that two of the records you're sorting have identical values for your chosen key. Don't worry; it happens in the best of SORT routines. For example, if you have a file full of names and addresses and need to sort them by zip code, you should expect to have several people with the same zip code. The same is true of the name of the state, the name of the city, and even birthdays.

The SORT verb handles duplicates just fine, and if you really don't care what kind of decision your sort makes about duplicate keys, just skip this section. I mean, your records with the duplicate keys are all going to be right there together just where you would expect them to be.

If you want complete control — if you are one of those people who has a real problem leaving any decisions at all to the SORT verb, read on. Although you can't do a whole lot about the existence of duplicate key values, you can guarantee that any duplicates appear in the order in which they were originally discovered during the sorting process. You do that by specifying the DUPLICATES option on the SORT verb:

```
    SORT SortWorkFile
        ON ASCENDING KEY Color OF WorkWidgets
        WITH DUPLICATES IN ORDER
```

```
        USING InFile
        GIVING OutFile.
```

Boy, that looks official, doesn't it? For all its impressive looks, all it does is cause SORT to go to the extra trouble of keeping track of where the records came from and make sure that the only shuffling done is when two records have different key values. Big deal.

Sorting from a File to a Procedure

The SORT verb doesn't require an output file. It requires some kind of output, but the output doesn't have to be a file. The output can go to a procedure. This technique is commonly used for sorting data into a procedure that calculates totals and subtotals to generate a report. For example, to generate a sales report with subtotals for each department, you need to sort on the department names so you can run independent totals for each department.

Here's an example program that reads from a file and delivers sorted output to a procedure:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. FileToProcedure.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT SortWorkFile ASSIGN TO "sortfile".
    SELECT InFile ASSIGN TO "infile".
DATA DIVISION.
FILE SECTION.
SD  SortWorkFile
    RECORD CONTAINS 12 CHARACTERS.
01  WorkWidgets.
    02  Color PIC X(6).
    02  Height PIC 9(2).
    02  Weight PIC 9(2).
    02  IQ PIC 9(2).
FD  InFile
    RECORD CONTAINS 12 CHARACTERS.
01  InWidgets.
    02  Color PIC X(6).
    02  Height PIC 9(2).
    02  Weight PIC 9(2).
    02  IQ PIC 9(2).
```

(continued)

(continued)

```
WORKING-STORAGE SECTION.
77  AT-END-MARKER PIC X VALUE "N".
    88  EndOfFile VALUE "Y".
PROCEDURE DIVISION.
Mainline.
    SORT SortWorkFile
        ON ASCENDING KEY Color OF WorkWidgets
        USING InFile
        OUTPUT PROCEDURE IS ShowStuff.
    STOP RUN.
ShowStuff.
    PERFORM UNTIL EndOfFile
        RETURN SortWorkFile
            AT END
                MOVE "Y" TO AT-END-MARKER
            NOT AT END
                DISPLAY WorkWidgets
        END-RETURN
    END-PERFORM.
```

The input to the SORT is a file defined as an FD. The work file has the same record layout as the input file and is defined as an SD. The SORT verb gets its input by USING the InFile, and depends on the PROCEDURE named ShowStuff to take care of the output.

Just in case you want to know all the sorted details, here's a brief synopsis of the plot:

1. The SORT verb assumes control and reads stuff from the input file and does its magic trick. The magically sorted records are left to fend for themselves in the work file.
2. The SORT verb then tells the program to PERFORM the OUTPUT PROCEDURE. The procedure ShowStuff (knowing that it must PERFORM its work only once) uses the RETURN verb to retrieve all the sorted records from the work file.
3. The procedure ShowStuff (in this example, it is a single paragraph) retrieves the records one by one by using RETURN. Each time the RETURN verb is used, a single record comes forth from the sort file and becomes resident in the sort record in memory. COBOL checks for AT END and NOT AT END just as if you were reading a regular ol' dumb sequential file.
4. Each time the RETURN verb pulls another record into memory, a DISPLAY statement gives you a look at the contents of that record.

The output from running this program looks like this:

```
Aqua   672405
Blue   341844
Bondo  572041
Green  130214
Orange843924
Puce   645285
Red    712582
```

In this example, the procedure that processes the data is a single paragraph. In the real world, this is usually not the case. Most programs want to do something that involves a little more than just a quick DISPLAY.

The following example shows how you can include a whole bunch of paragraphs as the OUTPUT PROCEDURE:

```
OUTPUT PROCEDURE IS ShowStuff THRU ShowStuffExit.
```

Using THRU allows you to include all kinds of fancy PERFORM and GO TO statements that can really bounce things around from one paragraph to another. Probably the best way to specify the procedure that's to receive the output from SORT is to use the name of a SECTION for the procedure instead of using a paragraph, like this:

```
OUTPUT PROCEDURE IS SORT-OUTPUT.
  . . .
SORT-OUTPUT SECTION.
    Code here to process records.
SORT-OUTPUT-EXIT.
    EXIT.
ANOTHER-SECTION SECTION.
  . . .
```

This code defines the OUTPUT PROCEDURE as being the section named SORT-OUTPUT.

Sorting from a Procedure to a File

In this section, I present a sample program that uses a procedure for its input and a file as its output. The SORT verb includes a clause that specifies the INPUT PROCEDURE the program uses for generating records. The INPUT PROCEDURE uses RELEASE statements to pass these records to the SORT verb, which puts the records in the correct order. You can use a process like this in a program that generates data — for example, by consolidating information from several files at once — and needs to have its output sorted. Here's the source code for this sample program:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ProcedureToFile.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT SortWorkFile ASSIGN TO "sortfile".
    SELECT OutFile ASSIGN TO "outfile".
DATA DIVISION.
FILE SECTION.
SD  SortWorkFile
    RECORD CONTAINS 12 CHARACTERS.
01  WorkWidgets.
    02  Color PIC X(6).
    02  Height PIC 9(2).
    02  Weight PIC 9(2).
    02  IQ PIC 9(2).
FD  OutFile
    RECORD CONTAINS 12 CHARACTERS.
01  OutWidgets.
    02  Color PIC X(6).
    02  Height PIC 9(2).
    02  Weight PIC 9(2).
    02  IQ PIC 9(2).
WORKING-STORAGE SECTION.
PROCEDURE DIVISION.
Mainline.
    SORT SortWorkFile
        ON ASCENDING KEY Color OF WorkWidgets
        INPUT PROCEDURE MakeRecords
        GIVING OutFile.
    STOP RUN.

MakeRecords.
    MOVE "Blue  341844" TO WorkWidgets.
    RELEASE WorkWidgets.
    MOVE "Red   712582" TO WorkWidgets.
    RELEASE WorkWidgets.
    MOVE "Orange843924" TO WorkWidgets.
    RELEASE WorkWidgets.
    MOVE "Green 130214" TO WorkWidgets.
    RELEASE WorkWidgets.
    MOVE "Puce  645285" TO WorkWidgets.
    RELEASE WorkWidgets.
```

```
MOVE "Aqua  672405" TO WorkWidgets.
RELEASE WorkWidgets.
MOVE "Bondo 572041" TO WorkWidgets.
RELEASE WorkWidgets.
```

The INPUT PROCEDURE defined on the SORT verb has the responsibility for generating records and passing them on to the SORT with RELEASE.

Here is a plot synopsis:

1. The SORT verb gets itself ready to receive records and then tells the INPUT PROCEDURE to PERFORM its assigned tasks.
2. The INPUT PROCEDURE is a single paragraph named MakeRecords, and MakeRecords knows that it is only going to be called once, so it produces all the records that it will ever produce and passes them to the SORT by using the magic word RELEASE.
3. The INPUT PROCEDURE then returns control back to the SORT verb (which has been patiently gathering up the records from the INPUT PROCEDURE) and the actual sorting takes place.
4. The completed records are then written to the output file in sorted order.

The INPUT PROCEDURE can be a single paragraph, as in the preceding example, or it can be a group of paragraphs, this way:

```
INPUT PROCEDURE IS MakeRecords THRU MakeRecordsExit.
```

In similar fashion, the INPUT PROCEDURE can be the name of a SECTION. Just name the SECTION the same way as you would a paragraph, like this:

```
INPUT PROCEDURE IS InputSection.
```

Sorting from a Procedure to a Procedure

With the techniques I describe in the previous two sections, you have everything you need for sorting from a procedure to a procedure. All you need is an SD file to use as the work area by the sort — the program internally produces the data to be sorted and then uses the sorted records internally for its own purposes. Everything except the sort itself is done in memory. The SORT sentence looks like this:

```
SORT SortWorkFile
    ON ASCENDING KEY Color OF WorkWidgets
    INPUT PROCEDURE MakeRecords
    OUTPUT PROCEDURE ShowStuff.
```

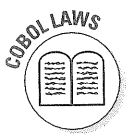
The INPUT PROCEDURE uses the RELEASE statement to send records to SORT, and the OUTPUT PROCEDURE uses RETURN to retrieve them. I can't imagine a program that would need to use an external file to sort its own internally generated data for its own internal use. However, I can imagine a program with a complicated INPUT PROCEDURE that reads data from multiple files to create consolidated data that needs to be sorted and sent directly to an OUTPUT PROCEDURE that summarizes and formats the data into reports.

Merging the Sorted Files

After you understand SORT, understanding MERGE is a walk in the park.

If you have a collection of files that contain sorted records, you can use MERGE to combine the records into one big file that contains sorted records. You just name the output file and the list of input files, and MERGE glues them all together in one big wad.

You can't merge if you are out of sorts. You can only MERGE files that have already been sorted.



The MERGE verb has almost the same set of options as the SORT verb has:

- You must name an SD file as a work file.
- You can specify ASCENDING or DESCENDING keys (but the order you specify had better match the order in the files being merged).
- You can specify the COLLATING SEQUENCE alphabet (but it had better match the one that was used to sort the files being merged).
- You can have the output go to a file or to an OUTPUT PROCEDURE, just like you can with SORT.
- As a matter of fact, only one difference exists between MERGE and SORT: the source of the input. The only possible source of input for MERGE is a collection of presorted files.

Here is an example program that merges three files into one:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Merger.
ENVIRONMENT DIVISION.
```

```
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT MergeWorkFile ASSIGN TO "mergefile".
    SELECT OutFile ASSIGN TO "outfile".
    SELECT InFile1 ASSIGN TO "infile1".
    SELECT InFile2 ASSIGN TO "infile2".
    SELECT InFile3 ASSIGN TO "infile3".
DATA DIVISION.
FILE SECTION.
SD  MergeWorkFile
    RECORD CONTAINS 12 CHARACTERS.
01  WorkWidgets.
    02  Color PIC X(6).
    02  Height PIC 9(2).
    02  Weight PIC 9(2).
    02  IQ PIC 9(2).
FD  OutFile
    RECORD CONTAINS 12 CHARACTERS.
01  OutWidgets.
    02  Color PIC X(6).
    02  Height PIC 9(2).
    02  Weight PIC 9(2).
    02  IQ PIC 9(2).
FD  InFile1
    RECORD CONTAINS 12 CHARACTERS.
01  InWidgets1.
    02  Color PIC X(6).
    02  Height PIC 9(2).
    02  Weight PIC 9(2).
    02  IQ PIC 9(2).
FD  InFile2
    RECORD CONTAINS 12 CHARACTERS.
01  InWidgets2.
    02  Color PIC X(6).
    02  Height PIC 9(2).
    02  Weight PIC 9(2).
    02  IQ PIC 9(2).
FD  InFile3
    RECORD CONTAINS 12 CHARACTERS.
01  InWidgets3.
    02  Color PIC X(6).
    02  Height PIC 9(2).
    02  Weight PIC 9(2).
    02  IQ PIC 9(2).
WORKING-STORAGE SECTION.
PROCEDURE DIVISION.
```

(continued)

(continued)

```
Mainline.
    MERGE MergeWorkFile
        ON ASCENDING KEY Color OF WorkWidgets
        USING InFile1 InFile2 InFile3
        GIVING OutFile.
    STOP RUN.
```

The three input files all have identical record layouts and are sorted on the same key. The MERGE sentence reads from all three input files and copies their contents to the output file. The result is one large file containing all the records in sorted order.

As you can see, MERGE really does look a lot like the SORT sentence. The following code shows a MERGE sentence that uses an OUTPUT PROCEDURE:

```
    MERGE MergeWorkFile
        ON ASCENDING KEY Color OF WorkWidgets
        USING InFile1 InFile2 InFile3
        OUTPUT PROCEDURE DooWahDiddy.
```

The data from the three input files are merged into the work file and are available to the OUTPUT PROCEDURE by using the RETURN verb. The output procedure works just like it does on the SORT sentence.

Part V
The Part of Tens

In this part . . .

After you read the first chapter in this part of the book, you can face the millennium with confidence that you understand the solution to the year 2000 problem. Executing the solution still requires lots of work, but I help you understand what you need to do to make your COBOL programs millennium-safe. I also show you how to convert existing data, or write your program so it can live with the existing data through the end of this millennium and into the next one.

The other chapter in the Part of Tens present a total of 10 different things you can do in your COBOL programs. Some of these things are not so easy because they run up against some limitations in the design of the COBOL language — but anything that you can do in any other computer language, you also can do in COBOL.

Chapter 17

Ten Faces of the Millennium Problem

In This Chapter

- Converting code from two- to four-digit years
- Converting files to accommodate the year 2000
- Windowing and pivoting dates into the next century
- Clearing up the leap-year misconception
- Correcting the system dates

Relaxing beneath the palms of the oasis, the Bedouin peddler, chewing slowly while studying a COBOL program listing, mumbled softly to himself, "Bad dates." No, this desert wanderer isn't complaining about the quality of his snack; he has realized that his COBOL program — like most existing programs — can't deal with the year 2000.

Here's a quick rundown on how this year 2000 problem became such a big deal for COBOL programmers:

1. During this century, it became common practice to write the year by using only its last two digits. Instead of 1953, we all just put down 53. Everybody knows the first two digits are 19, right? Printed forms that include places to put the date have "19__" printed on them. It just makes life easier not having to write 19 every time. We probably saved a thousand gallons of ink.
2. Enter the computer. Computer memory was expensive and everybody already knew that 19 was a totally unnecessary part of the name of the year. Leave it off!
3. The COBOL language was soon born and programmers gathered around it and said, "Ooie! That's pretty!" They started using it. Now these programmers were the same people who had been perpetrating the two-digit year on paper for years. They brought that tradition to COBOL programming.

4. Time passes and lots of COBOL programs are written. By "lots," I'm talking about numbers that would get the attention of Carl Sagan — billions and billions and billions of lines of COBOL code. Here's the kicker: Almost every one of these COBOL programs uses a date in one way or another, and almost every one records the date using only two digits. Oops.

Programmers working in the 1960s and 1970s saw the year 2000 as being far into the future. I can remember having discussions about the situation, but they were all of the "grin and wink" variety, and nobody believed their programs would last that long. During the 1980s, more discussions took place, and an occasional paper was written on the subject, but still, nobody took the problem seriously. The international COBOL language standard of 1985 — a standard that was expected to be in place for more than ten years — has the millennium problem built right into its system date. We still use this standard today. I have personally seen code written as recently as 1995 — new code, written to modify an existing legacy system — that uses two digits for the year.

"What is going to happen when the year 2000 comes?" That's a hard question to answer. Everybody who tries to answer it comes up with something different. The answers vary from minor inconvenience to a major global disaster. Fixing this problem is certainly going to cost a lot of money. I feel pretty sure that some companies will disappear (ones that depend on their computers and have taken years to develop their software). Some utilities and services could be disrupted for a while — a month or so. Certainly, billing foul-ups will occur (incorrect dates and past-due notices and such). A few computers will simply crash and just won't do anything until they are fixed. Most predictions are sheer speculation, but one thing I know to be a fact: You won't find one unemployed programmer on the face of the Earth.

The year 2000 problem exists in other places besides COBOL, but because this book is about COBOL, this chapter concentrates on the problem only as it applies to COBOL. The most straightforward solution is simply to convert all the files and all the programs from two-digit to four-digit years, but in some circumstances, that approach is just not practical. This chapter describes several things you can do to solve the problem.

The millennium problem is not difficult to understand, nor is it difficult to solve. It is, however, huge — an enormous amount of work must be done. Some of it can be automated, but no silver bullet exists that can magically solve the problem. Some help is available, however. Here are some resources you may want to explore:

- The Web site `www.infogoal.com/cbd/cbdhome.htm` contains links to numerous COBOL sites, including many sites that have COBOL-specific year 2000 information.

- The Web site `www.mitre.org/research/y2k` contains lots of year 2000 information. Oriented primarily toward government systems, this site also has links to dozens of other sites that discuss the year 2000.
- For more information on the millennium problem, see *Year 2000 Solutions For Dummies,* by K.C. Bourne (published by IDG Books Worldwide, Inc.).

Understanding the Two-Digit Year

The most common face of the millennium problem in COBOL is the `YYMMDD` grouping. In almost every program that has the year 2000 problem, you find a record that looks like this:

```
01  MunchingDate.
    02  YY PIC 99.
    02  MM PIC 99.
    02  DD PIC 99.
```

The year 2000 problem originates in this type of date record. In most cases, the year comes first so the dates are in the correct order when sorted. The only problem is that when the year 2000 comes, the value of `YY` will be `00` and the sorting order is blown. Also, if the dates are displayed or printed in a four-digit format, the 19 is just stuck on the front of the two-digit year, resulting in the year 2000 being displayed as the year 1900. A program that needs to calculate the number of days between two dates could come up with a hundred years worth of days. You could have records of people who died before they were born — or people who were married at the age of 4. A bill could be considered to be a hundred years past due — and the appropriate late fees added.

I could go on and on about all the different problems the two-digit YY can cause — that could take up an entire book — but that's not what this chapter is about. This chapter describes ways to fix the problem, so roll up your sleeves and read on.

For the simplest fix of all, just add a two-digit field to hold the century number (19 or 20). You can change the record to something like this:

```
01  MunchingDate.
    02  CC PIC 99.
    02  YY PIC 99.
    02  MM PIC 99.
    02  DD PIC 99.
```

This change leaves the record as it was except for the addition of a line of code that defines a century field. After you make this change, you need to locate every reference to MunchingDate in the program and change the code to deal with the new CC field. You also need to write code that inserts the correct century number. I heard about one instance of a program that had a correct record format (it had the CC field for the century), but throughout the program the value 19 was being moved into that field. The CC field may as well have not been there. The exact method you use to determine the correct value for CC is the main subject I cover in this chapter — the method you decide to use depends on your circumstances.

You can use an alternate approach when adding space to hold the two digits of the century. Instead of adding a two-digit CC field, you can just change the YY field to four digits, like this:

```
01  MunchingDate.
    02  YYYY PIC 9999.
    02  MM   PIC 99.
    02  DD   PIC 99.
```

I prefer the first version (the code that includes a CC field), but it's all a matter of personal taste and the characteristics of the program you are working on. Using YYYY has one small advantage: Any direct reference to the YY field causes a compiler error, and you can quickly locate the code and change it. Of course, no error occurs for MOVE CORRESPONDING or if the entire record is moved as a block, so you still have to inspect the code closely.

When you are making this change from two digits to four, remember that you are changing the size of the record. Normally this change doesn't matter, but if a copy of this record is moved to some other location, or if it is written to (or read from) a file, you have more work to do. You must change the record sizes in the file (as I describe in the upcoming section "Converting a File that Contains YY").

Totally Obscure Names for YY

Not all date fields have the YY name. Programmers can call the year anything they want to call it. You can find two-digit year fields with names like these:

```
05  Terminator PIC 99.
05  Boudreaux PIC 99.
05  Ending PIC 99.
```

And the field may contain more than just the year. The year, month, and day fields can be combined into a six-digit field like these:

```
01  Temporary PIC 9(6).
01  Frammis PIC X(6).
```

More than likely, you find dates like this being used as temporary work areas. A date is moved into one of these work areas from somewhere else, and later, it is moved to another regular date field.

Finding this problem can be difficult because it operates silently. When you move an eight-digit date into one of these beauties, COBOL just trims off the last two digits — which is usually the DD part. Then, when you move the date back into an eight-digit date, the DD part doesn't go with it. The result is that you only have about one chance in 30 of having the right day. This is a very sneaky bug because trimming things like this is normal for COBOL — it truncates the data swiftly and without complaint — so check the work areas used to get dates.

Converting a File that Contains YY

Millions of records stored in disk files have two-digit years in them. It is possible to convert the data in the files, but to do so, you must convert the two-digit year data to four-digit year data in every record in the file. The only way to change the size of the records is to create a new file.

The following example shows the kind of processing that you must do to create a new file with a record size and format that can handle a four-digit year field:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. FromYYtoCCYY.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT OldFile ASSIGN TO "oldfile".
    SELECT NewFile ASSIGN TO "newfile".
DATA DIVISION.

FILE SECTION.
FD  OldFile
    RECORD CONTAINS 20 CHARACTERS.
01  OldRecord.
    05  Name              PIC X(10).
```

(continued)

(continued)

```
    05  OldDateFormat.
        10  YY PIC 99.
        10  MM PIC 99.
        10  DD PIC 99.
    05  EggplantSalad    PIC X(4).
FD  NewFile
    RECORD CONTAINS 22 CHARACTERS.
01  NewRecord.
    05  Name             PIC X(10).
    05  NewDateFormat.
        10  CC PIC 99.
        10  YY PIC 99.
        10  MM PIC 99.
        10  DD PIC 99.
    05  EggplantSalad    PIC X(4).

WORKING-STORAGE SECTION.
77  FileStatusIndicator  PIC X.
    88  EndOfFile VALUE "Y".
PROCEDURE DIVISION.
MAINLINE.
    OPEN INPUT OldFile.
    OPEN OUTPUT NewFile.
    MOVE SPACES TO FileStatusIndicator.
    PERFORM UNTIL EndOfFile
        READ OldFile
            AT END MOVE "Y" TO FileStatusIndicator
            NOT AT END PERFORM WriteNewRecord
        END-READ
    END-PERFORM.
    STOP RUN.
WriteNewRecord.
    MOVE CORRESPONDING OldRecord TO NewRecord.
    MOVE CORRESPONDING OldDateFormat OF OldRecord TO
                       NewDateFormat OF NewRecord.
    MOVE 19 TO CC OF NewDateFormat OF NewRecord.
    WRITE NewRecord.
```

This program reads the records from a file that has dates laid out in the old six-digit format (which contains a two-digit year) and writes them to a file with dates in the new eight-digit format (which contains a four-digit year). The records are identical in every respect — names and all — except for the date field. A `MOVE CORRESPONDING` statement copies every field except the date field (because the two date fields have different names). Another `MOVE`

CORRESPONDING is used to transfer the YY, MM, and DD data. Finally, the constant value 19 is put into the CC field and the new record is written to disk. All that is left to do is delete the old file and start using the new one.

After you convert the file to the new format, you can use it only with programs that you have also converted to the new format. If you don't know exactly which programs use this file, you need to come up with some kind of search method to look through all the program source code to find the ones that reference this file. If you miss just one, your new file could be corrupted by having data in both formats, and it would be unusable. You will have generated garbage. If you don't already have one, I suggest creating a cross-reference of the programs and the files they use.

One more thing. This program assumes that all the dates already in the file are prior to the year 2000. If some program has written data to the file for the years 00 or 01 (meaning 2000 or 2001), you have to add code to solve that problem. To do so, you can use one of the techniques I describe later in this chapter.

Windowing the Year Doesn't Change the File Format

If you find yourself in a situation where you have two-digit year data in a file, and you just have no practical way to change the format of the records in the file, all is not lost. You can use a little trick, known as *windowing,* to automatically map years to the correct century. This technique works only if you know for sure that the dates in the file will never have more than a 100-year spread from oldest to newest. The year that you use to create the dividing line between centuries is called the *pivot year.*

All you do is pick some two-digit year to be the one that determines which century you are in. Say, for example, you can pick 50 as your pivot year. Any year greater than 50 is assumed to be in the 1900s (like 1958, 1978, 1994, and so on) while any year less than the pivot year is in the 2000s (like 2003, 2018, 2027, and so on). If the year is exactly 50, it will be assumed to be 2050. The reason for the 100-year limitation is that no way exists to represent anything before 1951 or after 2050.

This technique won't work for all applications. For example, insurance companies insure items that are more than 100 years old, and some people walking around out there have some really antique birthdays. But this technique can work just fine for many other applications. If you are running a used car lot or keeping track of company statistics, it should be okay.

Here is an example that demonstrates how this technique works. I choose the year 35 as the pivot. In other words, no years in the file come before 1936 or after 2035. The following program shows how the two-digit years are converted to four-digit years using the pivot. The program begins with the two-digit year 70, counts up through 99, back to 00 and up to 70 again. The program prints one line for each year it converts:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Pivot.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  I PIC 9(4) COMP.
01  PivotYear.
    02  CC PIC 99 VALUE 20.
    02  YY PIC 99 VALUE 35.
01  TwoDigitYear.
    02  YY PIC 99.
01  FourDigitYear.
    02  CC PIC 99.
    02  YY PIC 99.
PROCEDURE DIVISION.
MAINLINE.
    MOVE 70 TO YY OF TwoDigitYear.
    PERFORM VARYING I FROM 1 BY 1 UNTIL I > 100
       PERFORM ConvertByPivot
       ADD 1 TO YY OF TwoDigitYear
    END-PERFORM.

ConvertByPivot.
    MOVE CC OF PivotYear TO CC OF FourDigitYear.
    MOVE YY OF TwoDigitYear to YY OF FourDigitYear.
    IF YY OF TwoDigitYear > YY OF PivotYear
        SUBTRACT 1 FROM CC OF FourDigitYear.
    DISPLAY FourDigitYear.
```

This example converts the year dates from 70 through 99 and then continues on from 00 through 70. The following output lists the results of the conversion from two to four digits:

```
1970
1971
1972
1973
 . . .
```

```
1995
1996
1997
1999
2000
2001
2002
2003
2004
 . . .
2031
2032
2033
2034
2035
1936
1937
1938
1939
1940
 . . .
1970
```

This list shows where the transitions occur. Starting with the year 70, all the numbers up through 99 are preceded with 19. Starting with 00, years are preceded with 20. The conversion to 20 continues on until the pivot year 2035 is encountered. Because 2035 is the pivot year, all the conversions of numbers after that begin with 19. The result is that this method can correctly represent any date from January 1, 1936, through December 31, 2035.

This method is probably the best one around for those cases in which you have zillions of records on disk or tape and no practical way to convert them. One great advantage is that you can put this method to use immediately because it doesn't require any kind of data conversion — you can convert the programs one at a time and put them right back into service. Because no data conversion is involved, all the unconverted programs can still run (but, of course, if they are left unconverted, those programs will be really wrong about the dates when the millennium comes).

You can use this technique as a sort of rolling solution. Say, for example, the dates you are tracking never go more than three years into the future. You could use a standard routine that calculates a pivot date that is, say, five years in the future. With each year that passes, your pivot date moves forward. Changing the pivot date has no effect on the results of the calculations (unless some date in the file gets to be more than 100 years behind the pivot date).

Adding a Century Indicator to DD or MM

Here's a sneaky little solution that allows you to use the existing data record format but have the century information embedded right in it. Take a look at our old friend here:

```
01  DateFormat.
    04  YY PIC 99.
    04  MM PIC 99.
    04  DD PIC 99.
```

If this record is included somewhere in the midst of a file, and you just can't convert the file, you can encode the century information in the MM or DD field. You only need one of them. The MM field is a little easier to work with, so I show you how it can be done with the month field.

It is a fact that twelve months are in a year, which imposes a range limit of 01 through 12 on the MM field. This fact means that the first digit of MM must always be either 0 or 1. To encode more information, you can let the values of 0 and 1 represent the first digit of months for years in 1900, and let 2 and 3 represent the first digit of months for years in 2000. This encoding means that any existing data (dates in the 1900s) will not have to be changed. It also means that dates from the year 2000 on will have to be written and read differently — that is, any program that reads or writes the data needs to know how the year data is encoded.

Here's a sample program that demonstrates the trick:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Embedded.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  I PIC 9(2) COMP.
01  SixDigitDate.
    02  YY PIC 99.
    02  MM PIC 99.
    02  DD PIC 99.
01  EightDigitDate.
    02  CC PIC 99.
    02  YY PIC 99.
    02  MM PIC 99.
    02  DD PIC 99.
PROCEDURE DIVISION.
```

```
MAINLINE.
    MOVE 80 TO YY OF SixDigitDate.
    MOVE 15 TO DD OF SixDigitDate.
    MOVE 10 TO MM OF SixDigitDate.
    PERFORM ConvertSixToEight.
    PERFORM ConvertEightToSix.
    PERFORM ShowDates.
    MOVE 30 TO MM OF SixDigitDate.
    PERFORM ConvertSixToEight.
    PERFORM ConvertEightToSix.
    PERFORM ShowDates.
    STOP RUN.

ShowDates.
    DISPLAY MM OF SixDigitDate "/"
            DD OF SixDigitDate "/"
            YY OF SixDigitDate "    "
            MM OF EightDigitDate "/"
            DD OF EightDigitDate "/"
            CC OF EightDigitDate
            YY OF EightDigitDate.
ConvertSixToEight.
    MOVE CORRESPONDING SixDigitDate TO EightDigitDate.
    IF MM OF SixDigitDate > 12
        SUBTRACT 20 FROM MM OF EightDigitDate
        MOVE 20 TO CC OF EightDigitDate
    ELSE
        MOVE 19 TO CC OF EightDigitDate
    END-IF.

ConvertEightToSix.
    MOVE CORRESPONDING EightDigitDate TO SixDigitDate
    IF CC OF EightDigitDate > 19
        ADD 20 TO MM OF SixDigitDate
    END-IF
```

The `ConvertSixToEight` paragraph converts a six-digit date field into an eight-digit date field. If the `MM` value is greater than 12, it is taken to be an encoding for the century, so the `CC` value of the eight-digit date is set to 20. Of course, there are really only 12 months, so the program subtracts the magic number 20 from `MM`, which brings it back into the range of 1 through 12. If the `MM` value was already in the range of 1 through 12, it is assumed that the century should be 19.

The ConvertEightToSix paragraph takes a quick peek at the CC value in the eight-digit date and, if it is greater than 19, the magic number of 20 is added to the MM value in the six-digit format.

The output from this program looks like this:

```
10/15/80    10/15/1980
30/15/80    10/15/2080
```

The preceding output shows the same day, 100 years apart. The top line shows both formats of the date in the 1900s. The bottom line shows the two formats for the day in the 2000s. Everything is intuitive to the eye, except for that funny little month number for the six-digit form of the year 2000 dates.

This trick allows the storage of eight-digit dates in a six-digit record. The advantage of this method is that the existing data and the existing records do not have to change. Also, the dates still sort in the right order because all the month numbers in the year 2000 have been translated to the same values. This method also has the advantage of allowing you to start converting programs and testing them with the existing data — they should work just fine with the data already in place in the files. This method does require that all the programs that access the data be altered before any year 2000 dates are allowed in the files. But you need to do that no matter what form of conversion you use.

Using a Single Character for DD or MM

This section describes one of the less-elegant solutions to the year 2000 problem: using a single character for DD or MM. It works, but I can't think of a situation where it is a real improvement over the method I describe in the preceding section of this chapter. Well, it has one advantage — it is able to go past the century 2000 and on into 2100 and beyond. But it is so polecat ugly that nobody would let it live that long.

This solution can work if you have a situation in which six-digit dates are embedded in the files and you can change the data in the files, but you can't change the record size. Change the date record definition from the normal MM, DD, YY to the following format:

```
05  SixCharacterDate.
    08  M    PIC X.
    08  D    PIC X.
    08  YYYY PIC 9999.
```

If that looks odd to you, that's good. It is odd. The trick to make this thing work is to come up with a way to represent the month and the day in a single character each. The month is easy — just use the digits 1 through 9 for January through September, and use A for October, B for November, and C for December. Flushed with success, the next trick is to do the same thing with the day-of-month. The digits 1 through 9 can be used to represent the numbers 1 through 9 and the letters A through V to represent the numbers 10 through 31.

Don't use this technique. Use the form I describe in the previous section. The capability to move on into the twenty-second century is really not that important — besides, if your program lasts that long and has this code in it, everybody will law and think we were all really weird back here.

Don't Get Bitten by the Leap-Year Bug

Every fourth year is a leap year, right? Well, usually. Adding a day to February once every four years works pretty well, except that this adjustment adds just a skosh too much to the calendar. Therefore, once every hundred years, we have to skip having a leap year. However, *that* skipping is just a little too much if we do it *every* 100 years, so once every 400 years, we skip skipping and go ahead and have a leap year. It goes on from here, but 400 years is enough to worry about this soon after lunch.

Is the year 2000 a leap year? Without getting into all the astronomical nitty-gritty of "how many times the earth revolves around the sun" versus "the number of times the earth spins on its axis while traveling around the sun," I can put it simply: Yes, 2000 is a leap year.

Here's how to determine whether a year is a leap year: All years that are evenly divisible by 400 (or years that are evenly divisible by 4 and not divisible by 100) are leap years. For example, the 400-year exceptions (1600, 2000, 2400, and so on) are evenly divisible by 400, and are therefore leap years. Similarly, because a "normal" leap year like 1996 is divisible by 4 but not divisible by 100, it is also a leap year.

Although century years (1700, 1800, 1900, and 2100) skip their leap days, the year 2000 is the 400-year exception that balances the calendar. You can add special code to your program to calculate the leap-year exception for 2000, but don't do it. The fact is, division by 4 and by 100 can find all leap years for 400 years in either direction. Here's a sample program that contains a paragraph that detects a leap year:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Leaper.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  DateFields.
    05  YYYY PIC 9999.
    05  MM   PIC 99.
    05  DD   PIC 99.
77  WORK-YYYY      PIC 9999.
77  WORK-REMAINDER PIC 9999.
PROCEDURE DIVISION.
MAINLINE.
    MOVE "20000101" TO DateFields.
    PERFORM TestYear.
    STOP RUN.
TestYear.
    DIVIDE YYYY BY 400 GIVING WORK-YYYY
        REMAINDER WORK-REMAINDER
    END-DIVIDE.
    IF WORK-REMAINDER IS EQUAL TO ZERO
        DISPLAY YYYY " is a leap year."
    ELSE
        DIVIDE YYYY BY 4 GIVING WORK-YYYY
            REMAINDER WORK-REMAINDER
        END-DIVIDE
        IF WORK-REMAINDER IS EQUAL TO ZERO
            DIVIDE YYYY BY 100 GIVING WORK-YYYY
                REMAINDER WORK-REMAINDER
            END-DIVIDE
            IF WORK-REMAINDER IS EQUAL TO ZERO
                DISPLAY YYYY " is not a leap year."
            ELSE
                DISPLAY YYYY " is a leap year."
            END-IF
        ELSE
            DISPLAY YYYY " is not a leap year."
        END-IF
    END-IF.
```

The date to be tested is put into the `DateFields` record. The `TestYear` paragraph is called on to test for the divisibility of the year. Notice that the value of the year is four digits — this program is correct for all years from 1600 through 2400. Here is the output of this program:

```
2000 is a leap year.
```

Using 99 as a Special YY Is a No-No

It isn't just the year 2000 becoming 00 that will cause all the problems. Occasionally, a situation will arise in which 1999 brings about some confusion. This confusion stems from the COBOL tradition of using all 9s (instead of `HIGH-VALUES` or `LOW-VALUES`) as a special value in a field — you may encounter 99 as the two-digit year meant to indicate "no date."

A numeric field in COBOL allows only the digits 0 through 9, so no matter what combination of digits wind up in the field, they always make up a valid number. Under certain circumstances, however, a programmer wants to be able put "no value" into a field. For example, perhaps you have a field named `AgreedCommission` that holds some agreed-upon commission percentage. The commission can be zero; so if the value is zero, it indicates that the two parties have agreed on that. What do you put in here if the two parties haven't agreed on a commission as of yet? You could have a separate field to indicate whether an agreement has been reached, but it has become a sort of COBOL tradition to use `99` in situations like this. It is obvious that you would never have a commission of 99 percent, right?

The same thing applies to dates. If a record contains an optional date field, quite often you find `YYMMDD` filled with `999999`. It seems like a safe thing to do because no such date exists. The programs are written to check for this special value and process it in some special way.

You can continue to use this sort of encoding, but you need to be careful about a couple of things. The program could have been written so that it just automatically rejects dates of 99 that appear in the `YY` field. The larger problem comes about when you are either converting the data or using a pivot year for date windowing, as I describe earlier in this chapter. It has to be treated as a special case in your date conversion routine, so watch for it.

The Special Form YYDDD

It's not uncommon to store dates in a record that looks like this:

```
01  Expiration.
    04  YY  PIC 9(2).
    04  DDD PIC 9(3).
```

The value of `DDD` is the number of the day during the year. That is, `001` is January 1, `002` is January 2, and so on through the year. The advantage of this format is that it makes doing the calculations that move forward and backward inside a year very easy — all you need to do is add and subtract the number of days. This format is often called a *Julian date*.

The three Julians

Three different Julian calendars exist.

Julius Caesar, a well-known stabbed Roman, established a new calendar based on the solar year instead of a lunar year. Because it came from Julius, it became known as the Julian calendar. It introduced the concept of leap year to adjust for an extra one-fourth day that was in the year. (Unfortunately, his calculations were off just a bit, requiring a new calendar in the 1500s — but that's another story.)

Another Julian calendar, also known as the Julian period, is used mostly by astronomers. In this calendar, a date is simply a number — a count of the number of days beginning from January 1, 4713 BC. Joseph Scaliger, the guy who came up with this calendar, named it after his father — who happened to be named Julius — so that makes two calendars by the name Julian.

The third Julian calendar is the one that is used by programmers. It is a modification of the Julian period calendar. It simply a count of the number of days from the beginning of the year.

As far as the year 2000 problem is concerned, just about everything that applies to YYMMDD also applies to YYDDD. If you can't expand the size of the record to accommodate two more digits for the century, you can use windowing on the YY portion of the date just like with YYMMDD. You can convert the record and all the files by adding two digits and storing the dates as CCYYDDD, like this:

```
01  Expiration.
    04  CC  PIC 99.
    04  YY  PIC 99.
    04  DDD PIC 9(3).
```

If you cannot change all the records in all the files, you can add a century indicator to DDD like the one for MM (which I describe earlier in this chapter, in the section "Adding a Century Indicator to DD or MM"). The DDD values from 1 through 365 can be for dates in the 1900s, while larger values are for dates in the 2000s. Just add 400 to the DDD value if the year is in the 2000s, which makes the day numbers range from 401 through 765.

Here is an example program that uses this technique to convert between five-digit and seven-digit Julian dates:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Julian.
ENVIRONMENT DIVISION.
DATA DIVISION.
```

```
WORKING-STORAGE SECTION.
77  I PIC 9(2) COMP.
01  FiveDigitJulian.
    02  YY   PIC 99.
    02  DDD  PIC 999.
01  SevenDigitJulian.
    02  CC   PIC 99.
    02  YY   PIC 99.
    02  DDD  PIC 999.
PROCEDURE DIVISION.
MAINLINE.
    MOVE "94271" TO FiveDigitJulian.
    PERFORM ConvertFiveToSeven.
    PERFORM ConvertSevenToFive.
    PERFORM ShowDates.
    MOVE "94671" TO FiveDigitJulian.
    PERFORM ConvertFiveToSeven.
    PERFORM ConvertSevenToFive.
    PERFORM ShowDates.
    STOP RUN.

ShowDates.
    DISPLAY "Five digit date " FiveDigitJulian
            " equals seven digit date " SevenDigitJulian.

ConvertFiveToSeven.
    MOVE CORRESPONDING FiveDigitJulian TO
          SevenDigitJulian.
    IF DDD OF FiveDigitJulian > 400
        SUBTRACT 400 FROM DDD OF SevenDigitJulian
        MOVE 20 TO CC OF SevenDigitJulian
    ELSE
        MOVE 19 TO CC OF SevenDigitJulian
    END-IF.

 ConvertSevenToFive.
    MOVE CORRESPONDING SevenDigitJulian TO FiveDigitJulian
    IF CC OF SevenDigitJulian > 19
        ADD 400 TO DDD OF FiveDigitJulian.
```

The paragraph named `ConvertFiveToSeven` converts a Julian date with only a `YY` year field to a Julian date with both `CC` and `YY` fields. The `ConvertSevenToFive` paragraph converts in the opposite direction.

The Peculiar ACCEPT — A Built-In Oops

The ANSI 85 COBOL standard (the one followed by virtually all COBOL compilers today) has a great big millennium bug squashed right into it. The standard COBOL system date contains a two-digit year and is not capable of representing the year 2000. Right there in the ANSI standards documentation, it specifically uses the year *86* as a two-digit example of the format of the system date. It's as if the idea of planning 14 years into the future (from 1986 to 2000) was just too much and too far. Of course, hindsight is a wonderful thing — it all looks so obvious now, but it wasn't then. I mean, what are your plans for 14 years into the future? How well are you planning for the year 2011?

A PIC is worth a thousand words. Here is some code extracted from the ACCEPT examples in Chapter 12:

```
01  SystemDate.
    02  YY PIC 99.
    02  MM PIC 99.
    02  DD PIC 99.
01  SystemDay.
    02  Year PIC 99.
    02  DayOfYear PIC 999.
      . . .
    ACCEPT SystemDate FROM DATE.
    ACCEPT SystemDay FROM DAY.
```

The COBOL language returns the system date to your program in a definitely politically incorrect format. You can use a couple of solutions. First, look in the documentation of your compiler for a special system date routine of some kind. If your compiler has one that gives you a better date format, use it. If your compiler doesn't have a special system date routine, you can use the following example as a model for writing code that converts the system date to CCYYMMDD format:

```
01  SystemDate.
    02  YY PIC 99.
    02  MM PIC 99.
    02  DD PIC 99.
01  MySystemDate.
    02  CC PIC 99.
    02  YY PIC 99.
    02  MM PIC 99.
    02  DD PIC 99.
  . . .
AcceptMySystemDate.
    ACCEPT SystemDate FROM DATE.
```

```
    MOVE CORRESPONDING SystemDate TO MySystemDate.
    IF YY OF SystemDate > 90
        MOVE 19 TO CC OF MySystemDate
    ELSE
        MOVE 20 TO CC OF MySystemDate
    END-IF.
```

This example uses a pivot year of 90, but you can use any year you like as long as it is in the past. Assuming that the flow of time continues in its present direction and not return us to some date prior to 1990, this routine should work until 2090.

The example first reads the system date into the dreaded YYMMDD format and then, by pivoting on the year 90, determines whether the century should be 19 or 20. This technique will always work because in every case the date will be the current date, not something made up and entered by a user or something.

The same thing will work for the Julian YYDDD format. In fact, the solution is almost identical:

```
01  SystemDay.
    02  YY  PIC 99.
    02  DDD PIC 999.
01  MySystemDay.
    02  CC PIC 99.
    02  YY  PIC 99.
    02  DDD PIC 999.
  . . .
AcceptMySystemDay.
    ACCEPT SystemDay FROM DAY.
    MOVE CORRESPONDING SystemDay TO MySystemDay.
    IF YY OF SystemDay > 90
        MOVE 19 TO CC OF MySystemDay
    ELSE
        MOVE 20 TO CC OF MySystemDay
    END-IF.
```

The Embedded Date

The embedded date is the eleventh of the ten faces of the millennium problem. However, it's pretty rare, so I'll just throw it in as a bonus. The year number can be encoded into serial numbers used for equipment tags and validation numbers during software installation. Exactly how your secret serial number is encoded determines what you need to do and whether you need to do anything at all.

Here's a hypothetical example. Suppose that the serial number is 12 digits long. The year digit is always the fourth digit. The decade digit is the sum of the fifth and eighth digits. Here is some code that sticks the numbers into the serial number:

```
COMPUTE Serial(4) = YearDigit.
IF DecadeDigit IS GREATER THAN 5
    COMPUTE Serial(5) = DecadeDigit - 5
    COMPUTE Serial(8) = 5
ELSE
    COMPUTE Serial(5) = 0
    COMPUTE Serial(8) = DecadeDigit
END-IF.
```

This is not a very fancy encoding, but you get the idea. This code can be used to pull the year back out of the serial number:

```
COMPUTE YearDigit = Serial(4).
COMPUTE DecadeDigit = Serial(5) + Serial(8).
```

The question arises about the year 2000. All you need to know is whether to put 19 or 20 in front of the two-digit year when you pull it out of the serial number. You could encode this information in one character. Exactly how you do this depends on what you are able to do to the serial number format. Can you add a digit to the serial number? Is there a position that is not being used for anything? If there is a digit somewhere that can only be, say, in the range of 1 to 4, how about mapping its values to the range of 5 to 9 for the year 2000?

If you can't change the serial number, about all that you have left is to use the windowing technique I describe earlier in this chapter for unchangeable data files.

Chapter 18

Ten Tasks That Are Really Hard to Do in COBOL

In This Chapter

- Finding the actual size of a COBOL record definition
- Constructing and deconstructing character strings
- Manipulating parts of character strings
- Finding a square root
- Generating random numbers

The creators of COBOL didn't think of everything. Performing certain tasks in COBOL can be, shall I say, cumbersome. Nothing is impossible, though — you can do anything in COBOL that you can do in any other programming language. However, some tasks require a little finagling. Sometimes you know what you want to do, but you wind up with one eye closed, staring at the ceiling for a few minutes, trying to figure out how to do it. Fortunately, these are things you don't need very often.

This chapter presents almost ten example programs, each of which demonstrates a technique for getting something done. I know the chapter is supposed to have ten programs, but, hey, this is the hard part.

This chapter includes COBOL code for determining the actual size of a COBOL record (useful when working with files), constructing character strings (useful for dynamically creating print formats), reading and writing comma-separated data (useful when communicating with PC software), generating random numbers (useful when creating test data), and finding the square root of a number. All of these examples are supplied on the CD that accompanies the book.

Determining the Actual Size of a Record

If you are going to work with COBOL files, the time will come when you need to know the size of a record. Even when working with variable-length records, you need to have numbers for both the smallest and largest records that the file is capable of holding.

In most cases, finding the size of a record is a straightforward matter of counting the characters in each field of the record. The numbers may get large enough that you need to take your shoes off, but if every field is `USAGE DISPLAY`, you can come up with the right number. In other cases, however, a record may include some special data type — something like `COMP` or `BINARY` — that makes calculating the file size really hard. Such cases call for a sneaky trick.

Here's how the trick works. You need a program that fills the entire record with a single character, moves the record to some location that is known to be larger than the record, and then locates the last occurrence of the character. The distance from the beginning of the record to the last character is the size of the record.

To find out the exact size of a record, no matter what compiler you are using and no matter how convoluted the record structure is, just grab a copy of the following program. All you need to do is copy your record into this program, in place of the one named `RecordToBeSized`, and then run the program. The size pops out on a `DISPLAY` statement.

Here's the code for the `RecordSize` program:

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. RecordSize.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
* * * * * * *
*01  RecordToBeSized.
*  Insert your record here. You can replace the name
*  RecordToBeSized with the name of your own record,
*  but you must make the change throughout the program.
* * * * * * *
*
*  The MeasurementBlock must be larger than the
*  RecordToBeSized. This example uses a size of 400 but
*  you will need to expand it for larger records.
*  Be sure you change the number in both places.
 01  MeasurementBlock.
     02  CharacterBlock PIC X(400).
```

```
    02  CharacterArray REDEFINES CharacterBlock.
        03  OneCharacter PIC X OCCURS 400 TIMES.
77  NDX PIC 9(4).
PROCEDURE DIVISION.
MeasureIt.
    MOVE ALL "X" TO RecordToBeSized.
    MOVE SPACES TO MeasurementBlock.
    MOVE RecordToBeSized TO MeasurementBlock.
    PERFORM VARYING NDX FROM 400 BY -1
        UNTIL OneCharacter(NDX) IS EQUAL TO "X".
    DISPLAY "The record size: " NDX.
```

Here's how the program works. The first two `MOVE` statements in the `MeasureIt` paragraph fill `RecordToBeSized` and `MeasurementBlock` with `X` characters and spaces, respectively. The third `MOVE` statement copies the contents of `RecordToBeSized` into `MeasurementBlock`. In other words, `MeasurementBlock` now contains exactly the same number of `X` characters as `RecordToBeSized`, with spaces filling out the rest of the `MeasurementBlock`.

The `PERFORM..UNTIL` statement sets the variable `NDX` equal to the index of the last character in the `MeasurementBlock` and then works through each preceding character in `MeasurementBlock` until it encounters an `X`. (In effect, the clause `FROM 400 BY -1` tells the `PERFORM..UNTIL` statement to start at the end of the `MeasurementBlock` and work toward the beginning.) Because the `PERFORM..UNTIL` statement starts at the end of the `MeasurementBlock` and works toward the beginning, the first `X` character it encounters is actually the last `X` character in the `MeasurementBlock`. Consequently, the index of this entry in `MeasurementBlock` matches the record size of `RecordToBeSized`.

After you find the record size, you jot down the number and tuck this little program back into your bag of tricks. I told you it was sneaky.

Arranging Data into Columns

COBOL has a talent for laying out fixed-size print lines containing fixed-position fields. All you have to do is define a record containing the fields in the positions you want them and then `MOVE` data into the fields, and you have a formatted line ready for printing. If you want lots of different print line formats, you just define lots of different records — one record for every print line format.

If you want to `MOVE` fields into position by columns instead of using a record format to define the position of each field, you want to do something COBOL was not designed to do. There is a way to do it, but it's a little weird.

A couple of characteristics of MOVE make this formatting technique possible. First, you can move any displayable data into a record and second, whenever you move displayable data into a record and the data doesn't fill the entire record, the balance of the record is filled with blanks. These facts make it possible to write a program that has a really strange technique for moving data into a specific column of a print line:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ColColCol.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  OutLine.
 02  COL-1.
  03  FILLER PIC X.
  03  COL-2.
   04  FILLER PIC X.
   04  COL-3.
    05  FILLER PIC X.
    05  COL-4.
     06  FILLER PIC X.
     06  COL-5.
      07  FILLER PIC X.
      07  COL-6.
       08  FILLER PIC X.
       08  COL-7.
        09  FILLER PIC X.
        09  COL-8.
         10  FILLER PIC X.
         10  COL-9.
          11  FILLER PIC X.
          11  COL-10.
           12  FILLER PIC X.
           12  COL-11.
            13  FILLER PIC X.
            13  COL-12.
             14  FILLER PIC X.
             14  COL-13.
              15  FILLER PIC X.
              15  COL-14.
               16  FILLER PIC X.
               16  COL-15.
                17  FILLER PIC X.
                17  COL-16.
                 18  FILLER PIC X.
                 18  COL-17.
                  19  FILLER PIC X.
```

```
                    19  COL-18.
                     20  FILLER PIC X.
                     20  COL-19.
                      21  FILLER PIC X.
                      21  COL-20.
PROCEDURE DIVISION.
Mainline.
    MOVE "Goober" TO COL-1.
    MOVE "Peas" TO COL-15.
    DISPLAY Outline.
    MOVE SPACES TO OutLine.
    MOVE "Goober" TO COL-3.
    MOVE "Peas" TO COL-12.
    DISPLAY Outline.
    MOVE SPACES TO OutLine.
    MOVE "Goober" TO COL-6.
    MOVE "Peas" TO COL-10.
    DISPLAY Outline.
    STOP RUN.
```

If you think the program itself looks weird, you should meet the guy who thought it up. As you can see from the code, putting text into a particular position on a line of output is a simple matter of moving that text into the appropriate column in `OutLine`. If you move more than one thing into the print line, be sure you do it in left-to-right order (every time you poke something into a column, everything to its right is overwritten). The output from this program looks like this:

```
Goober         Peas
  Goober   Peas
     GoobPeas
```

You can use this flexible format to combine data and descriptive information into any sort of print line you want. For example, the following code fragment shows how to place more than one field into the print line:

```
77  Maximum PIC 9(2) VALUE 43.
77  Minimum PIC 9(2) VALUE 17.
PROCEDURE DIVISION.
Mainline.
    MOVE "Max:" TO COL-1.
    MOVE Maximum TO COL-6.
    MOVE "Min:" TO COL-9.
    MOVE Minimum TO COL-14.
    DISPLAY Outline.
```

The four MOVE statements construct a line by combining descriptive text with data. Notice that the code in this example starts with the smaller column numbers — this is necessary because every MOVE statement replaces every character to its right. The output of this example looks like this:

```
Max: 43 Min: 17
```

Extracting Part of a Text String

If you have a character string that has more stuff in it than you need, you can use the example I show here to extract just the little section that you want. For example, you can pull the zip code out of an address or the area code out of a phone number. You tell this program which character to start with and which character to end with, and it goes in there and brings 'em back alive:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Substring.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  Extractor.
    02  ExtractedString PIC X(50).
    02  FILLER REDEFINES ExtractedString.
        03  ToChar PIC X OCCURS 50 TIMES
                INDEXED BY ToIndex.
    02  MasterString PIC X(200).
    02  FILLER REDEFINES MasterString.
        03  FromChar PIC X OCCURS 200 TIMES
                INDEXED BY FromIndex.
    02  StartIndex PIC 9(4) COMP.
    02  EndIndex PIC 9(4) COMP.
PROCEDURE DIVISION.
Mainline.
    MOVE "No milk if you don't have a cow in the barn"
        TO MasterString.
    DISPLAY MasterString.
    MOVE 16 TO StartIndex.
    MOVE 31 TO EndIndex.
    PERFORM ExtractSubstring.
    DISPLAY ExtractedString.
```

```
    STOP RUN.

ExtractSubstring.
    MOVE SPACES TO ExtractedString.
    SET ToIndex TO 1.
    PERFORM VARYING FromIndex FROM StartIndex BY 1
            UNTIL FromIndex > EndIndex
        MOVE FromChar(FromIndex) TO ToChar(ToIndex)
        SET ToIndex UP BY 1
    END-PERFORM.
```

First, you create a workspace like the one named `Extractor` in the example. It uses a combination of `REDEFINES` (I describe them in Chapter 4) and `OCCURS` (see Chapter 7) to build a work area to be used for string extraction.

Next, set up the string and specify which characters you wish to have extracted. You move the string you want a piece of into the `MasterString`, and then you specify the starting and ending characters by putting values into `StartIndex` and `EndIndex`.

Then you `PERFORM ExtractSubstring`, and you're done — your extracted substring is in `ExtractedString`.

This example first displays the output of the entire string, and then displays the extracted substring. The entire string looks like this:

```
No milk if you don't have a cow in the barn
```

The characters from 16 through 31 (the extracted substring) look like this:

```
don't have a cow
```

The `ExtractSubstring` paragraph uses the index `FromIndex` to retrieve characters from the input string, and it uses the index `ToIndex` to put characters into the output string. The value of `ToIndex` is initialized to 1 because you want to start at the beginning of the output string. The `PERFORM VARYING` statement starts the `FromIndex` at `StartIndex` (the position of the first character to be extracted) and counts in through `EndIndex` (the position of the last character to be extracted). Each time through the loop, one character is moved from the `FromChar` array (which is a `REDEFINES` of the input string) to the `ToChar` array (which is a `REDEFINES` of the output string).

Combining Text Strings

COBOL does a really swell job with fixed-length fields. You can create pages and pages of rows and columns of data, all neatly aligned vertically and horizontally — even diagonally if you are the kind of person who likes that sort of thing. Getting COBOL to handle variable-length fields is not as easy, but it can be done.

This program demonstrates how you can take text from two variable-length fields — in this case, first names and last names — and combine the two strings to produce some coherent output. The operation of pasting two strings back-to-back this way is called *concatenation*. The program concatenates the names in last-name-first format. For improved readability, the program inserts a comma and a space as part of each concatenated string. The input is the first name and last name in separate fields, like this:

```
Sally Jo  Hickson
Tracy     Clark
J. R.     Creampuff
```

Under normal circumstances, the input data would be from a file, but this example just uses input coded right into the program:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Concat.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  Builder.
    02  ResultString PIC X(200).
    02  FILLER REDEFINES ResultString.
        03  ToChar PIC X OCCURS 200 TIMES
                INDEXED BY ToIndex.
    02  NewString PIC X(50).
    02  FILLER REDEFINES NewString.
        03  FromChar PIC X OCCURS 50 TIMES
                INDEXED BY FromIndex, LastIndex.
    02  BlankCount PIC 9(2) COMP.
01  InputData.
    02  FirstName PIC X(10).
    02  LastName PIC X(10).
PROCEDURE DIVISION.
Mainline.
    MOVE "Sally Jo  Hickson" TO InputData.
    PERFORM BuildString.
    DISPLAY ResultString.
    MOVE "Tracy     Clark" TO InputData.
```

```
      PERFORM BuildString.
      DISPLAY ResultString.
      MOVE "J. R.     Creampuff" TO InputData.
      PERFORM BuildString.
      DISPLAY ResultString.
      STOP RUN.

 BuildString.
      SET ToIndex TO 1.
      MOVE SPACES TO ResultString.
      MOVE LastName TO NewString.
      MOVE ZERO TO BlankCount.
      PERFORM Concatenate.
MOVE "," TO NewString.
      PERFORM Concatenate.
      MOVE FirstName TO NewString.
      MOVE 1 TO BlankCount.
      PERFORM Concatenate.

 Concatenate.
      PERFORM UNTIL BlankCount = ZERO
          SET ToIndex UP BY 1
          SUBTRACT 1 FROM BlankCount
      END-PERFORM.
      PERFORM VARYING LastIndex FROM 50 BY -1 UNTIL
              FromChar(LastIndex) NOT EQUAL SPACE
                    OR
              LastIndex = 1
      END-PERFORM.
      PERFORM VARYING FromIndex FROM 1 BY 1 UNTIL
              FromIndex > LastIndex
                    OR
              ToIndex >= 200
          MOVE FromChar(FromIndex) TO ToChar(ToIndex)
          SET ToIndex UP BY 1
      END-PERFORM.
```

The output from this program looks like this:

```
Hickson, Sally Jo
Clark, Tracy
Creampuff, J. R.
```

All of the work of string concatenation is performed inside the record named `Builder`. Each input string piece is placed in `NewString` and the final output string winds up in `ResultString`. The `REDEFINES` serves the purpose of letting program get one character at a time. The `BlankCount` holds the number of blanks that the `Concatenate` paragraph inserts before appending the characters (such as the space following the comma).

The `Mainline` of the program simulates input by moving a first and last name into the `InputData` record. It then calls `BuildString` and displays the result.

The `BuildString` paragraph controls the formatting of the output string. This paragraph starts off by setting the `ToIndex` to `1` (so that output starts with the first character). Three parts are being concatenated: the last name, the comma, and the first name. Each one is inserted in the same way — the string is moved into `NewString` and the `Concatenate` paragraph is performed. The only variation in the process is when the `BlankCount` is set to insert a single blank in front of the first name.

The `Concatenate` paragraph does the detail work of appending the input string onto the end of the output string. The index value `ToIndex` determines the location of each character in the output string, so `ToIndex` is first adjusted to skip over any requested spaces. Next, the value of `LastIndex` is set to the position of the last nonblank character in the input data because the program needs to know how many characters are to be moved. Finally, the characters are moved one at a time from the input to the output strings, taking care to check the value of `ToIndex` and make sure it doesn't run past the end of `ResultString`.

Writing Comma-Delimited Text

Over the years, a de facto standard has arisen for formatting data records to be transferred from one program to another on personal computers. The process of transferring data is commonly known as *exporting* and *importing* data. Under this de facto standard for sharing data, a program that exports data writes each data record as a single line of text, with commas separating the individual fields in each record. Humans can easily understand this comma-delimited format, but COBOL needs your help in exporting and importing a comma-delimited file. Here is a little program that writes the contents of a record as comma-delimited text:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. StrungOut.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
```

```
01  Builder.
    02  ResultString PIC X(200).
    02  FILLER REDEFINES ResultString.
        03  ToChar PIC X OCCURS 200 TIMES
                INDEXED BY ToIndex.
    02  NewString PIC X(50).
    02  FILLER REDEFINES NewString.
        03  FromChar PIC X OCCURS 50 TIMES
                INDEXED BY FromIndex, LastIndex.
01  InputData.
    02  FirstName  PIC X(10).
    02  LastName   PIC X(10).
    02  ScrimmageDate.
        03  CC     PIC 99.
        03  YY     PIC 99.
        03  MM     PIC 99.
        03  DD     PIC 99.
    02  Title      PIC X(5).
    02  Job        PIC X(8).
PROCEDURE DIVISION.
Mainline.
    MOVE "Sally Jo  Hickson   20020821hick barefoot"
        TO InputData.
    PERFORM BuildString.
    DISPLAY ResultString.
    STOP RUN.

BuildString.
    SET ToIndex TO 1.
    MOVE SPACES TO ResultString.
    MOVE LastName TO NewString.
    PERFORM Append.
    MOVE FirstName TO NewString.
    PERFORM Append.
    MOVE ScrimmageDate TO NewString.
    PERFORM Append.
    MOVE Title TO NewString.
    PERFORM Append.
    MOVE Job TO NewString.
    PERFORM Append.

Append.
    IF ToIndex > 1
        MOVE "," TO ToChar(ToIndex)
        SET ToIndex UP BY 1
```

(continued)

(continued)

```
        END-IF.
        PERFORM VARYING LastIndex FROM 50 BY -1 UNTIL
                FromChar(LastIndex) NOT EQUAL SPACE
                      OR
                LastIndex = 1
        END-PERFORM.
        PERFORM VARYING FromIndex FROM 1 BY 1 UNTIL
                FromIndex > LastIndex
                      OR
                ToIndex >= 200
            MOVE FromChar(FromIndex) TO ToChar(ToIndex)
            SET ToIndex UP BY 1
        END-PERFORM.
```

The mechanics of this process are very similar to those I describe in the section "Combining Text Strings," earlier in this chapter. That program and this one both have the job of snuggling one string up against another. The `Mainline` paragraph initializes the `InputData` record and calls `BuildString` to do the conversion into a single string.

The `BuildString` paragraph moves each field of the record into the `NewString` work area and performs `Append` to have the `NewString` appended to the `ResultString`. The `Append` paragraph puts a comma on the output line, but only if some data already appears on the output line. `Append` then determines the index of the last character to be moved (by starting at the end of the character array and moving backward until a nonblank is found). The characters are then moved, one at a time, to the `ResultString`. The index `ToIndex` remains in position in case more data is to come.

The final export string looks like this:

```
Hickson,Sally Jo,20020821,hick,barefoot
```

Of course, if you can write data like this, you need to be able to read it. That's the subject of the following section.

The `STRING` verb in COBOL just isn't quite flexible enough to handle this type of string. For one thing, it can't accept embedded blanks inside input fields, so a name like "Sally Jo" just won't work. You can set it to accept embedded blanks, but then it accepts all the trailing blanks in every field and scatters your data out over several acres.

Reading Comma-Delimited Text

The program in this section is an example of reading data from a comma-delimited list and putting it into the fields of a record. Think of this program as the inverse of the program I describe in the previous section:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. StrungIn.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  Builder.
    02  ExtractedString PIC X(50).
    02  FILLER REDEFINES ExtractedString.
        03  ToChar PIC X OCCURS 50 TIMES
                INDEXED BY ToIndex.
    02  IncomingString PIC X(200).
    02  FILLER REDEFINES IncomingString.
        03  FromChar PIC X OCCURS 200 TIMES
                INDEXED BY FromIndex,LastIndex.
01  InputData.
    02  FirstName  PIC X(10).
    02  LastName   PIC X(10).
    02  ScrimmageDate.
        03  CC     PIC 99.
        03  YY     PIC 99.
        03  MM     PIC 99.
        03  DD     PIC 99.
    02  Title      PIC X(5).
    02  Job        PIC X(8).
PROCEDURE DIVISION.
Mainline.
    MOVE "Sally Jo,Hickson,20020821,hick,barefoot"
        TO IncomingString.
    PERFORM BreakDown.
    DISPLAY InputData.
    STOP RUN.

BreakDown.
    PERFORM VARYING LastIndex FROM 200 BY -1
        UNTIL FromChar(LastIndex) IS NOT EQUAL TO SPACE
    END-PERFORM.
    MOVE SPACES TO InputData.
```

(continued)

(continued)

```
       SET FromIndex TO 1.
       PERFORM Extract.
       MOVE ExtractedString TO LastName.
       PERFORM Extract.
       MOVE ExtractedString TO FirstName.
       PERFORM Extract.
       MOVE ExtractedString TO ScrimmageDate.
       PERFORM Extract.
       MOVE ExtractedString TO Title.
       PERFORM Extract.
       MOVE ExtractedString TO Job.

   Extract.
       SET ToIndex TO 1.
       MOVE SPACES TO ExtractedString.
       PERFORM UNTIL FromChar(FromIndex) = ","
                           OR
                     FromIndex > LastIndex
           MOVE FromChar(FromIndex) TO ToChar(ToIndex)
           SET FromIndex UP BY 1
           SET ToIndex UP BY 1
       END-PERFORM.
       IF FromIndex < LastIndex
           SET FromIndex UP BY 1.
```

The `Mainline` of the program moves a comma-delimited line of data into `IncomingString`. The paragraph `BreakDown` is performed to split `IncomingString` into its component parts in the `InputData` record, and then the record is displayed. The output looks like this:

```
Hickson   Sally Jo  20020821hick barefoot
```

The `BreakDown` paragraph first sets the value of `LastIndex` to the last nonblank character of the input string. This step is important because the incoming string is variable length and the program needs to know where the actual data ends. The `Extract` paragraph is called a number of times (once for each incoming field) and the extracted data is then moved into its final resting place in the `InputData` record.

The `Extract` paragraph — starting from the current `FromIndex` position in the `FromChar` array — moves the characters, one by one, into the `ToChar` array. This process continues until a comma is found or the `FromIndex` reaches the `LastIndex`, indicating the end of data. Finally, the value of `FromIndex` is increased by one to move past the comma.

Converting Between Upper- and Lowercase

This handy routine converts all alphabetic characters in a field to either uppercase or lowercase:

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. CaseWorker.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  CaseWorkArea.
     02  WorkString PIC X(100).
     02  FILLER REDEFINES WorkString.
         04  Work PIC X OCCURS 100 TIMES INDEXED BY W.
     02  LowerCaseLetters PIC X(26) VALUE
         "abcdefghijklmnopqrstuvwxyz".
     02  FILLER REDEFINES LowerCaseLetters.
         04  LoLet PIC X OCCURS 26 TIMES.
     02  UpperCaseLetters PIC X(26) VALUE
         "ABCDEFGHIJKLMNOPQRSTUVWXYZ".
     02  FILLER REDEFINES UpperCaseLetters.
         04  UpLet PIC X OCCURS 26 TIMES.
     02  Lndx PIC 9(4) COMP.
 PROCEDURE DIVISION.
 Mainline.
     MOVE "Starting with MIXED case, You can
-         " switch UP and down."
         TO WorkString.
     DISPLAY WorkString.
     PERFORM ToUpperCase.
     DISPLAY WorkString.
     PERFORM ToLowerCase.
     DISPLAY WorkString.
     STOP RUN.

 ToUpperCase.
     PERFORM VARYING W FROM 1 BY 1 UNTIL W > 100
         IF Work(W) IS NOT EQUAL TO SPACE
             PERFORM VARYING Lndx FROM 1 BY 1
                     UNTIL Lndx > 26
                 IF Work(W) = LoLet(Lndx)
                     MOVE UpLet(Lndx) TO Work(W)
                 END-IF
```

(continued)

(continued)

```
                END-PERFORM
            END-IF
        END-PERFORM.

    ToLowerCase.
        PERFORM VARYING W FROM 1 BY 1 UNTIL W > 100
            IF Work(W) IS NOT EQUAL TO SPACE
                PERFORM VARYING Lndx FROM 1 BY 1
                        UNTIL Lndx > 26
                    IF Work(W) = UpLet(Lndx)
                        MOVE LoLet(Lndx) TO Work(W)
                    END-IF
                END-PERFORM
            END-IF
        END-PERFORM.
```

The record `CaseWorkArea` holds the string to be converted in `WorkString`. The string can be a mixture of upper- and lowercase letters, along with any other characters — the only ones that are affected by the conversion are the letters of the alphabet.

The two paragraphs `ToUpperCase` and `ToLowerCase` are alike — they just go in opposite directions. The outer `PERFORM` loop moves the index `W` one by one through all the characters that may need conversion. To save a bit of time, because the `SPACE` character occurs more often than any other, it is skipped. An inner `PERFORM` loop compares each character against the list of characters of the wrong case and, if one is found, the case is switched by having the character replaced by the corresponding character in the alphabet of the opposite case.

This example program first converts a string of mixed case into all uppercase and then converts it to all lowercase. The output looks like this:

```
Starting with MIXED case, You can switch UP and down.
STARTING WITH MIXED CASE, YOU CAN SWITCH UP AND DOWN.
starting with mixed case, you can switch up and down.
```

Finding a Square Root

COBOL doesn't have a square root verb, but you won't miss it. COBOL actually has something better — an exponentiation operator. Raising a number to the one-half power is the same as taking its square root.

You can use the exponentiation operator to raise any number to any power. For example, the following statement finds the square of 800:

```
COMPUTE EightHundredSquared = 800 ** 2.
```

You are not limited to just squares. You can do cubes, like this:

```
COMPUTE FrammisCubed = Frammis ** 3.
```

In fact, you can raise any number to any power you want. You can even raise values to something halfway between squared and cubed, like this:

```
COMPUTE GooPowered = Goo ** 2.5.
```

It's this capability to raise something to a fractional power that allows COBOL to find a square root. The laws of algebra state that taking the square root of a number is the same as raising it to the one-half power.

Here's a program that prompts you to enter a number and then displays the square root of that number:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Sqrt.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  ResponseNumber PIC 9(10) VALUE 1.
01  ShowNumber PIC ZZZZZZZZ9.
01  SquareRoot PIC ZZZZZ9.999.
PROCEDURE DIVISION.
Mainline.
    PERFORM UNTIL ResponseNumber IS EQUAL TO ZERO
        DISPLAY "Enter Number (or 0 to quit): "
            WITH NO ADVANCING
        ACCEPT ResponseNumber
        IF ResponseNumber IS NOT EQUAL TO ZERO
            PERFORM DoCalculations
        END-IF
    END-PERFORM.
    STOP RUN.

DoCalculations.
    COMPUTE SquareRoot = ResponseNumber ** (1.0/2.0).
    DISPLAY SquareRoot.
```

Each time you enter a nonzero value, the program displays the square root of that number and then asks for another number. Typical input and output looks like this:

```
Enter Number (or 0 to quit): 2
     1.414
Enter Number (or 0 to quit): 2000
    44.721
Enter Number (or 0 to quit): 144
    12.000
Enter Number (or 0 to quit): 0
```

Generating Random Numbers

Random numbers have several legitimate uses. For one thing, you can use them as test data that you can feed as input to a program to see how it works when the real-world data gets thrown at it. It's just not possible to try every conceivable combination of data to validate a program, but you can use random data to *field test* your software. Another big use for random numbers is in games — but, for some reason, not a lot of COBOL games exist. I don't exactly know why.

Before you use the random number program that I present in this section, take a look at the manual for your compiler to see if you already have one. If you don't find one there, try looking for one in the set of utility programs available on your machine. If you still don't find one, go ahead and use this one.

COBOL is not capable of generating truly random numbers. The best COBOL can do is to generate a *pseudo-random* sequence of numbers — these are numbers that appear to be random but really aren't. You give a pseudo-random number generator a starting value (called the *seed*) that determines the sequence of numbers that the program generates. They are pseudo-random because the same seed always produces the same sequence of numbers. The following example program uses the system clock for the seed to prevent the same sequence from being started again and again:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SemiRandom.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
```

```
01  RandomWork.
    02  BigNumber PIC 9(16).
    02  FILLER REDEFINES BigNumber.
        03  HighEnd PIC 9(8).
        03  LowEnd PIC 9(8).
    02  FILLER REDEFINES BigNumber.
        03  FILLER PIC 9(4).
        03  RandomNumber PIC 9(6).
        03  FILLER PIC 9(6).
    02  FILLER REDEFINES BigNumber.
        03  OneDigit PIC 9 OCCURS 16 TIMES
                INDEXED BY DigitIndex.
77  I PIC 9(2).
PROCEDURE DIVISION.
Mainline.
    PERFORM FirstRandom.
    PERFORM VARYING I FROM 1 BY 1 UNTIL I > 10
        PERFORM NextRandom
        DISPLAY RandomNumber
    END-PERFORM.
    STOP RUN.

FirstRandom.
    ACCEPT LowEnd FROM TIME.
    ACCEPT HighEnd FROM TIME.
    PERFORM NextRandom.

NextRandom.
    MULTIPLY BigNumber BY BigNumber.
    PERFORM VARYING DigitIndex FROM 1 BY 1
            UNTIL DigitIndex > 16
        ADD 1 TO OneDigit(DigitIndex)
    END-PERFORM.
```

Sixteen digits are used in generating the random number, but only six of the digits are actually used for the value in this example. Performing the `FirstRandom` paragraph sets up the initial value of the 16 digits from the system clock and then performs `NextRandom` to juggle them around a bit.

Each time a new random number is desired, the paragraph `NextRandom` is performed. It first squares the number to semi-randomly change all the digits. Then it does something that looks sort of weird. It adds 1 to each digit. The reason it adds 1 is because multiplication is used to modify the digits and if two or three zeroes happen to get into the number, they will replicate themselves. By adding one to all the digits, this zero-propagation problem is eliminated.

Every time this program is run, it generates a different set of numbers. Here is the output from a typical run:

```
987909
028350
427386
668506
997121
905177
708248
516641
296469
757610
```

Appendix

About the CD

Here's what you can find on the *COBOL For Dummies* CD-ROM:

- Demo versions of Acucobol compilers for Windows 3.1, 95, and NT
- A set of COBOL interpreters from Deskware for AIX, Linux, SunOS, Solaris, and Windows 95/NT
- A complete COBOL development system from Fujitsu
- A demo version of the Micro Focus NetExpress development environment
- Source code for the COBOL programs from Chapters 17 and 18
- A bonus appendix that shows you how to write COBOL programs that generate reports with headers, footers, running totals, and subtotals
- Another bonus appendix full of diagrams to help you remember the syntax of COBOL's verbs

System Requirements

The CD includes software and documentation for use in various operating environments. Specific system requirements depend on which portions of the CD you plan to use. To install and use Windows software from the CD-ROM, make sure that your system meets the following minimum requirements:

- A PC with a 25 MHz or faster 486 processor
- Microsoft Windows 3.1, Windows 95, or Windows NT
- At least 8MB of RAM installed on your computer. For best performance, I recommend at least 16MB of RAM.
- At least 50MB of hard drive space available to install a compiler
- A CD-ROM drive
- A VGA graphics display monitor

If you need more information on the basics, check out *PCs For Dummies,* 4th Edition, by Dan Gookin; *Windows 95 For Dummies* by Andy Rathbone; or *Windows 3.11 For Dummies,* 3rd Edition, by Andy Rathbone (all published by IDG Books Worldwide, Inc.).

What You'll Find

The following sections describe the software and the sample files on the CD.

Acucobol

The CD includes a fully functional demo version of ACUCOBOL-GT, a single-pass COBOL compiler that you execute from a command line to produce a program that you can then run by using the ACUCOBOL-GT runtime system. For detailed installation and operation instructions, check out the Word document QWKSTRT.DOC in the ACUCOBOL directory. For more information about Acucobol, visit its Web site: `www.acucobol.com`.

Deskware

Deskware is a freeware COBOL interpreter that runs on Linux, Windows 95, Windows NT, Solaris, SunOS, and AIX. The DESKWARE directory on the CD contains a subdirectory for each of the five platforms, and each subdirectory contains an executable program named COBOL (or, in the case of Windows, COBOL.EXE), which is the COBOL interpreter.

You can either copy the interpreter to your system (no installation procedure is required), or you can run it directly from the CD. To run an example program, just enter this command:

```
COBOL program name
```

To run the interpreter in interactive mode, which allows you to enter your COBOL code directly and have it executed, just enter this command:

```
COBOL -i
```

After the program starts running, just type `help` to get a list of the options. Here's Deskware's Web address: `www.deskware.com`.

Fujitsu COBOL

Note: In order to install the software from Fujitsu, please use the following serial number: 99-03317-70168.

The CD contains lots of software from Fujitsu, including compilers and other development tools for several platforms. Here's a quick rundown on the various subdirectories you can find in the FUJITSU directory on the CD:

- **COBOL16:** 16-bit COBOL development tools for Windows 3.1.
- **COBOL32:** 32-bit COBOL development tools for Windows 95 and NT.
- **HP10:** A compiler, a runtime system, and other development tools for the HP-UX 10 operating environment.
- **SOL:** Fujitsu COBOL tools for Sun Solaris users.

Here's the address for the Fujitsu Web site: `www.adtools.com`.

Micro Focus NetExpress

The NETXPRES directory on the CD includes a demo version of the Micro Focus NetExpress COBOL development environment, which includes a compiler, an editor, a debugger, and various other programming and project management tools for Windows 95/NT. Here's the address for the Micro Focus Web site: `www.microfocus.com/`.

Code examples from the book

The directory named EXAMPLES contains source code for the example programs from Chapters 17 and 18. These examples are not fully functional utilities, but they do contain code that could be useful to you in a larger program that you are writing.

Within the EXAMPLES directory, two subdirectories contain the same source code examples — one for Windows text and one for UNIX text. To use the source code, just copy it into the directory of your choice and use your editor to make changes or to extract the portions you want to use.

Bonus appendixes

Included on the CD are two bonus appendixes: "Printing Reports" and "Diagrams of the COBOL Verbs." To read these bonus appendixes, you need to install the Adobe Acrobat Reader, a free program that lets you open, read, and print Portable Document Format (PDF) files. The following section describes the steps for installing software from the CD.

You use Acrobat Reader to display and print the bonus appendixes. As with most Windows programs, you open the files that contain the bonus appendixes — D:\PRINTRPT.PDF and D:\VERBS.PDF — by using Acrobat Reader's File⇨Open command.

Installing the Software from the CD

Complete these steps to install the Acucobol, Fujitsu, or Micro Focus software, and the Adobe Acrobat Reader on your Windows system:

1. **Insert the CD into your CD-ROM drive and close the drive door.**
2. **Windows 3.x (that is, 3.1 or 3.11) users: From Program Manager, choose File⇨Run and then click the Browse button.**

   Windows 95/NT users: Click the Start button, choose Run, and then click the Browse button.
3. **Select the file you want to install and then click OK (substitute your CD-ROM drive letter if different from D:)**

To Install	*Select*
Acucobol on Windows 3.*x*	D:\ACUCOBOL\GTEVAL16.EXE
Acucobol on Windows 95/NT	D:\ACUCOBOL\GTEVAL32.EXE
Fujitsu on Windows 3.*x*	D:\FUJITSU\COBOL16\SETUP.EXE
Fujitsu on Windows 95/NT	D:\FUJITSU\COBOL32\SETUP.EXE
NetExpress on Windows 95/NT	D:\NETXPRES\SETUP.EXE
Adobe Acrobat Reader on Windows 3.1	D:\READER\AR16E30.EXE
Adobe Acrobat Reader on Windows 95	D:\READER\AR32E30.EXE

4. **Click the OK button and follow the Setup instructions.**

   To install the Fujitsu compiler, you need to enter the following serial number: 103-2001 1699-03317-70168. You should see the first part of this number in the install window, so you just need to enter the following digits: 99-03317-70168.
5. **After Setup is complete, restart your computer and start Windows.**

The CD contains compilers and utility software that have been tested and should load and run properly. But things go wrong. If you get error messages like `Not enough memory` or `Setup cannot continue`, try one or more of these methods and then try using the software again:

- Turn off any antivirus software that you have on your computer.
- Close all running programs.
- Have your local computer store add more RAM to your computer.

If you still have trouble with the CD, please call IDG Books Worldwide Customer Service: 800-762-2974 (outside the U.S.: 317-596-5261).

INDEX

(continued)

• B •

• C •

• D •

(continued)

• E •

• F •

• G •

• H •

• I •

• J •

• N •

• O •

(continued)

• *P* •

• S •

• X •

• Y •

• Z •

IDG Books Worldwide, Inc., End-User License Agreement

READ THIS. You should carefully read these terms and conditions before opening the software packet(s) included with this book ("Book"). This is a license agreement ("Agreement") between you and IDG Books Worldwide, Inc. ("IDGB"). By opening the accompanying software packet(s), you acknowledge that you have read and accept the following terms and conditions. If you do not agree and do not want to be bound by such terms and conditions, promptly return the Book and the unopened software packet(s) to the place you obtained them for a full refund.

1. **License Grant.** IDGB grants to you (either an individual or entity) a nonexclusive license to use one copy of the enclosed software program(s) (collectively, the "Software") solely for your own personal or business purposes on a single computer (whether a standard computer or a workstation component of a multiuser network). The Software is in use on a computer when it is loaded into temporary memory (RAM) or installed into permanent memory (hard disk, CD-ROM, or other storage device). IDGB reserves all rights not expressly granted herein.

2. **Ownership.** IDGB is the owner of all right, title, and interest, including copyright, in and to the compilation of the Software recorded on the disk(s) or CD-ROM ("Software Media"). Copyright to the individual programs recorded on the Software Media is owned by the author or other authorized copyright owner of each program. Ownership of the Software and all proprietary rights relating thereto remain with IDGB and its licensers.

3. **Restrictions on Use and Transfer.**

   (a) You may only (i) make one copy of the Software for backup or archival purposes, or (ii) transfer the Software to a single hard disk, provided that you keep the original for backup or archival purposes. You may not (i) rent or lease the Software, (ii) copy or reproduce the Software through a LAN or other network system or through any computer subscriber system or bulletin-board system, or (iii) modify, adapt, or create derivative works based on the Software.

   (b) You may not reverse engineer, decompile, or disassemble the Software. You may transfer the Software and user documentation on a permanent basis, provided that the transferee agrees to accept the terms and conditions of this Agreement and you retain no copies. If the Software is an update or has been updated, any transfer must include the most recent update and all prior versions.

4. **Restrictions on Use of Individual Programs.** You must follow the individual requirements and restrictions detailed for each individual program in the "About the CD" appendix of this Book. These limitations are also contained in the individual license agreements recorded on the Software Media. These limitations may include a requirement that after using the program for a specified period of time, the user must pay a registration fee or discontinue use. By opening the Software packet(s), you will be agreeing to abide by the licenses and restrictions for these individual programs that are detailed in the "About the CD" appendix and on the Software Media. None of the material on this Software Media or listed in this Book may ever be redistributed, in original or modified form, for commercial purposes.

5. **Limited Warranty.**

   (a) IDGB warrants that the Software and Software Media are free from defects in materials and workmanship under normal use for a period of sixty (60) days from the date of purchase of this Book. If IDGB receives notification within the warranty period of defects in materials or workmanship, IDGB will replace the defective Software Media.

   (b) **IDGB AND THE AUTHOR OF THE BOOK DISCLAIM ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, WITH RESPECT TO THE SOFTWARE, THE PROGRAMS, THE SOURCE CODE CONTAINED THEREIN, AND/OR THE TECHNIQUES DESCRIBED IN THIS BOOK. IDGB DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE SOFTWARE WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE SOFTWARE WILL BE ERROR FREE.**

   (c) This limited warranty gives you specific legal rights, and you may have other rights that vary from jurisdiction to jurisdiction.

6. **Remedies.**

   (a) IDGB's entire liability and your exclusive remedy for defects in materials and workmanship shall be limited to replacement of the Software Media, which may be returned to IDGB with a copy of your receipt at the following address: Software Media Fulfillment Department, Attn.: *COBOL For Dummies,* IDG Books Worldwide, Inc., 7260 Shadeland Station, Ste. 100, Indianapolis, IN 46256, or call 800-762-2974. Please allow three to four weeks for delivery. This Limited Warranty is void if failure of the Software Media has resulted from accident, abuse, or misapplication. Any replacement Software Media will be warranted for the remainder of the original warranty period or thirty (30) days, whichever is longer.

   (b) In no event shall IDGB or the author be liable for any damages whatsoever (including without limitation damages for loss of business profits, business interruption, loss of business information, or any other pecuniary loss) arising from the use of or inability to use the Book or the Software, even if IDGB has been advised of the possibility of such damages.

   (c) Because some jurisdictions do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitation or exclusion may not apply to you.

7. **U.S. Government Restricted Rights.** Use, duplication, or disclosure of the Software by the U.S. Government is subject to restrictions stated in paragraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause of DFARS 252.227-7013, and in subparagraphs (a) through (d) of the Commercial Computer–Restricted Rights clause at FAR 52.227-19, and in similar clauses in the NASA FAR supplement, when applicable.

8. **General.** This Agreement constitutes the entire understanding of the parties and revokes and supersedes all prior agreements, oral or written, between them and may not be modified or amended except in a writing signed by both parties hereto that specifically refers to this Agreement. This Agreement shall take precedence over any other documents that may be in conflict herewith. If any one or more provisions contained in this Agreement are held by any court or tribunal to be invalid, illegal, or otherwise unenforceable, each and every other provision shall remain in full force and effect.

Installation Instructions

The *COBOL For Dummies* CD-ROM contains sample programs, COBOL interpreters and compilers, and other COBOL development tools that you can install and use. Here's a quick overview of the CD's contents:

- Completely functional, demo versions of Acucobol compilers for Windows 3.1 and Windows 95/NT
- COBOL interpreters from Deskware for AIX, Linux, SunOS, Solaris, and Windows 95/NT
- A complete COBOL development system from Fujitsu, including compilers for Windows, HP-UX, and Sun
- A fully functional, timed demo version of the NetExpress COBOL development environment from Micro Focus
- Complete source code for the example COBOL programs from Chapters 17 and 18
- A bonus appendix that shows you how to write COBOL programs that generate reports with such features as headers, footers, running totals, and subtotals
- Another bonus appendix full of diagrams to help you remember the syntax of COBOL's verbs

For instructions on installing the sample programs and the software from the CD-ROM, see the "About the CD" appendix in this book.

IDG BOOKS WORLDWIDE BOOK REGISTRATION

We want to hear from you!

Visit **http://my2cents.dummies.com** to register this book and tell us how you liked it!

- Get entered in our monthly prize giveaway.
- Give us feedback about this book — tell us what you like best, what you like least, or maybe what you'd like to ask the author and us to change!
- Let us know any other *...For Dummies*® topics that interest you.

Your feedback helps us determine what books to publish, tells us what coverage to add as we revise our books, and lets us know whether we're meeting your needs as a *...For Dummies* reader. You're our most valuable resource, and what you have to say is important to us!

Not on the Web yet? It's easy to get started with *Dummies 101*®*: The Internet For Windows*® *95* or *The Internet For Dummies*®, 4th Edition, at local retailers everywhere.

Or let us know what you think by sending us a letter at the following address:

...For Dummies Book Registration
Dummies Press
7260 Shadeland Station, Suite 100
Indianapolis, IN 46256-3945
Fax 317-596-5498

BUSINESS AND GENERAL REFERENCE BOOK SERIES FROM IDG

COMPUTER BOOK SERIES FROM IDG