COURSE TECHNOLOGY
CENGAGE Learning
Professional • Technical • Reference

Java™ for COBOL Programmers

Third Edition

- Utilizes COBOL examples to teach object-oriented principles and Java syntax.
- Unlike many books that teach Java to novices, no knowledge of C or C++ is assumed.
- Includes a completely new chapter on Eclipse.
- Provides a companion CD, which includes all the source code for the examples in the book, software downloads, and other helpful tools.

John C. Byrne

JAVA™ FOR COBOL PROGRAMMERS, THIRD EDITION

JOHN C. BYRNE

Charles River Media

A part of Course Technology, Cengage Learning

Australia, Brazil, Japan, Korea, Mexico, Singapore, Spain, United Kingdom, United States

Java for COBOL Programmers,
Third Edition

John C. Byrne

Publisher and General Manager, Course Technology PTR: Stacy L. Hiquet

Associate Director of Marketing: Sarah Panella

Content Project Manager: Jessica McNavich

Marketing Manager: Mark Hughes

Acquisitions Editor: Mitzi Koontz

Development Editor and Technical Reviewer: Arron Ferguson

Project Editor and Copy Editor: Kim Benbow

CRM Editorial Services Coordinator: Jen Blaney

Interior Layout: Jill Flores

Cover Designer: Mike Tanamachi

CD-ROM Producer: Brandon Penticuff

Indexer: Jerilyn Sproston

Proofreader: Ruth Saavedra

© 2009 Course Technology, a part of Cengage Learning.

ALL RIGHTS RESERVED. No part of this work covered by the copyright herein may be reproduced, transmitted, stored, or used in any form or by any means graphic, electronic, or mechanical, including but not limited to photocopying, recording, scanning, digitizing, taping, Web distribution, information networks, or information storage and retrieval systems, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the publisher.

For product information and technology assistance, contact us at **Cengage Learning Customer & Sales Support, 1-800-354-9706**

For permission to use material from this text or product, submit all requests online at **cengage.com/permissions**
Further permissions questions can be emailed to **permissionrequest@cengage.com**

Sun, Sun Microsystems, the Sun logo, Java, JavaBeans, and all trademarks and logos that contain Sun, Solaris, or Java, are trademarks or registered trademarks of Sun Microsystems, Inc. or its subsidiaries in the United States and other countries. Microsoft, Windows, and Internet Explorer are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Apache Tomcat and Tomcat are trademarks of the Apache Software Foundation. Eclipse, Built on Eclipse, and Eclipse Ready are trademarks of Eclipse Foundation, Inc. LegacyJ is the property of LegacyJ Corp. SoftwareMining is a trademark of SoftwareMining, UK.

All other trademarks are the property of their respective owners.

Library of Congress Control Number: 2008931028

ISBN-13: 978-1-59863-565-5

ISBN-10: 1-58450-565-6

eISBN-10: 1-58450-618-0

Course Technology
25 Thomson Place
Boston, MA 02210
USA

Cengage Learning is a leading provider of customized learning solutions with office locations around the globe, including Singapore, the United Kingdom, Australia, Mexico, Brazil, and Japan. Locate your local office at: **international.cengage.com/region**

Cengage Learning products are represented in Canada by Nelson Education, Ltd.

For your lifelong learning solutions, visit **courseptr.com**

Visit our corporate website at **cengage.com**

Printed in the United States of America
1 2 3 4 5 6 7 11 10 09

To the memory of Charles Byrne, whose passion for learning still inspires those who knew him.

About the Author

John C. Byrne is the Vice President of Technology and primary system architect for a high-end enterprise software vendor and has over 20 years experience in the software industry.

Contents

Introduction xvi

Part I Introducing Java 1

1 Objects and Classes 3

The COBOL Subroutine 4

Calling a Subroutine 4

MYSUB COBOL 5

CALLER COBOL 6

CALLER COBOL: CONTROL 7

Terms to Review: Subroutines 9

Objects and Java 10

ErrorMsg Class 11

Caller Class 12

Terms to Review: Objects 13

2 Introducing the Java Development Environment 17

Runtime Interpretation and Java Byte Codes 17

Getting Started with Java's SDK 21

Applets with SDK 24

Classes and Filenames 26

CLASSPATH 27

CODEBASE 28
Packages 28
Inside a Package 29
Name Collisions 31
Packages and Filenames 32
Compressed Packages 33
Applications vs. Applets 34
Reviewing the Samples 34

3 Messages and Methods 37
MYSUB COBOL 38
CALLER COBOL 39
Messages in Java 40
ErrorMsg Class 40
Caller Class 41
Multiple Messages 41
Class ErrorMsg 41
Method Overloading 43
Caller Class 43
Method Overloading in COBOL 44
MYSUB COBOL 44
CALLER COBOL 46
Terms to Review 48
Exercises: Classes, Objects, and Methods 48
Reviewing the Samples 57
HelloWorld: The Application 61
HelloWorld: The Applet 62
ErrorMsg: The Class 63

4 Class Members 65
MYSUB COBOL 66
MYSUB COBOL: ACTION-SWITCH 68
Java Variables 68
ErrorMsg Class 69
Classes, Objects, and Members Review 71
Objects and COBOL 72
Using Objects in Java 76
Java Data Members 77
ErrorMsg Class 78
Caller Class 78
Local Variables 79
Primitive Data Types 80
Arrays 83
Arrays as Parameters 87
Method Members 87
ErrorMsg Class: Static Variable 88
ErrorMsg Class: Static Initializer 88
Constructors 89
Exercises: Class Members 91
Reviewing the Samples 99

5 Inheritance, Interfaces, and Polymorphism 101
Inheritance and Object-Oriented Design 101
Inheritance and Objects 103
Inheriting Methods 103
Caller Class 104
Redefining a Method 105
ErrorMsg Class 105
Caller Class 106

Extending a Method 106
  ErrorMsg Class 106
  Caller Class 108
Why Inheritance? 108
Inheritance, Objects, and COBOL 109
  NEWSUB COBOL 109
  NEWSUB COBOL: MYSUB 110
More COBOL Object-Oriented Design Patterns 112
  CALLER COBOL 114
  MYSUB COBOL 116
Inheritance and Java 119
  ErrorMsg Class 120
  PopupErrorMsg Class 120
  PrintfileErrorMsg Class 120
  TextMessage Class 121
  ErrorMsg Class 121
  PopupErrorMsg Class 122
  PrintfileErrorMsg Class 122
  Consumer Class 122
  TextMessage Class 123
  ErrorMsg Class 123
  ErrorMsg Class: Override 124
  PopupErrorMsg Class 124
  PrintfileErrorMsg Class 124
  Consumer Class 124
  PrintfileErrorMsg Class 125
  Sharing Variables and Methods 126
Hiding Variables and Methods 126
The this Variable 127
Java Interfaces 128
  Writeline Interface 129

PrintfileErrorMsg Class 129
Caller Class 130
Using Interfaces 130
Hiding Methods and Members 131
Polymorphism 132
Exercises 133
Reviewing the Samples 144

Part II Java's Syntax 147

6 Java Syntax 149
COBOL vs. Java Syntax 149
Java Statements 150
Java Comments 152
Java Operators 153
Binary Arithmetic Operations 154
Understanding Reference Variables with COBOL 159
Exercises: Java's Syntax 160
Reviewing the Exercises 165

7 Flow Control 167
Code Block 168
The if Statement 168
The while Statement 172
The do...while Statement 174
The for Statement 175
The switch Statement 176
The break, continue Statements 177
Exercises: Flow Control 181
Reviewing the Exercises 186

8 Strings, StringBuffers, StringBuilders, Numbers, and BigNumbers 189
Strings 189
Comparing Strings 191
Working with Strings 193
Numeric Wrapper Classes 196
StringBuffers 200
BigNumbers 202
Exercises: Strings, StringBuffers, Numbers, and BigNumbers 208
Reviewing the Exercises 216

9 Exceptions, Threads, and Garbage Collectors 219
Exception Class Hierarchy 221
Creating Exceptions 221
Using Exceptions 223
Exception-Processing Suggestions 228
Exception-Processing Summary 229
Threads 229
Inheriting from Thread 230
Implementing Runnable 232
Synchronization 233
Benefits and Cautions 235
Garbage Collection 236
Exercises: Java's Exceptions and Threads 237
Reviewing the Exercises 240

10 I/O in Java 243
Streams vs. Record-Based I/O 244
The File Class 246
InputStream and OutputStream 249
Serialization 252

Readers and Writers 254
RandomAccessFile 256
Exercises 258
Reviewing the Exercises 262

11 Java Collections 263
Collections Background 264
Ordered Collections: Vectors and ArrayLists 266
Keyed Collections: Hashtable and HashMap 270
Other Collections 271
Iterators 272
Ordering and Comparison Functions 273
Exercises: Java Collections 275
Reviewing the Exercises 278

12 Other Java Topics 279
Graphical User Interface Development 279
Properties Files 281
Java Utilities 282
Jar Utility 282
Javadoc Utility 284
Exercises 287
Reviewing the Exercises 289

Part III Introducing Enterprise Java 291

13 Java Database Connectivity 293
How JDBC Works 294
Connecting to the Database 296
Querying a Table 298

Inserting, Updating, and Deleting 300
Configuring the JDBC-ODBC Bridge 301
Exercises 302
Reviewing the Exercises 304

14 Servlets and Java Server Pages 305
Browsers and Web Servers 306
The Servlet as Transaction Processor 308
Servlet Protocol 310
Java Server Pages 313
Getting Started with Servlets and JSPs 316
Exercises 317
Reviewing the Exercises 320

15 Introduction to Enterprise JavaBeans 321
Distributed Computing 322
The Different Kinds of EJBs 323
Container Services 324
The Interfaces and the Implementation Class 325
Accessing the Bean from the Client 327
Exercises 327
Reviewing the Exercises 328

16 Introduction to XML 329
The Basics 330
XML vs. HTML 330
Document Type Definitions (DTDs) 332
DTD Components 333
Elements 334
Attributes 335
Entities 336

XML Declaration 337
A Complete XML Document 338
XML Schemas 339
Authoring XML Documents 339
XML and Java 339
XML and HTML 340
Where to Use XML 341
Electronic Data Interchange (EDI) 342
Online XML or Web Services 343
XML and OAG 344
Other Opportunities 351
Exercises 352
Reviewing the Exercises 354

17 Introducing Eclipse 355
Installing Eclipse 356
Start Using Eclipse 356
Make a New Eclipse Project 358
Run with Eclipse 362
Debug with Eclipse 362
Refactoring with Eclipse 363

Part IV Appendixes 369

Appendix A: About the CD-ROM 371
Exercises 371
Java SDK 371
LegacyJ 372
Eclipse 372
Tomcat 372

SoftwareMining 373
Case Studies 373

Appendix B: Java Information Available Elsewhere 375
Java Resources 375
Java Magazines 375
Java Tools 376
COBOL Information 376

Appendix C: Buzzwords 377
Active Server Pages (ASP) 377
AWT 378
Client/Server 378
Common Object Request Broker Architecture (CORBA) 379
Components 380
Component Object Model (COM) 380
Distributed Component Object Model (DCOM) 381
Enterprise JavaBeans (EJBs) 381
File Transfer Protocol (FTP) 382
Firewall 382
Hypertext Transfer Protocol (HTTP) 383
Integrated Development Environment (IDE) 383
Interface Definition Language (IDL) 383
Internet Inter-ORB Protocol (IIOP) 384
JavaBeans 384
Java Native Interface (JNI) 384
JavaScript 385
Java Server Pages (JSP) 385
JDBC 385
Microsoft Foundation Classes (MFC) 386
Open Database Connectivity (ODBC) 386

Remote Method Invocation (RMI) 386
Remote Procedure Call (RPC) 387
Secure Sockets Layer (SSL) 387
Swing 388
TCP/IP 388
Unified Modeling Language (UML) 389
Uniform Resource Locator (URL) 389
VBScript 390
Web Services 390

Appendix D: Sun Microsystems, Inc. Binary Code License Agreement 391

Index 399

Introduction

It has been more than 10 years since Sun first announced the Java programming language. It is safe to say that few technologies have generated similar excitement, interest, and allegiance. The promise of this new, cross-platform, object-oriented language with its simplified syntax has garnered the support of large numbers of developers, tool vendors, industry experts, and, of course, the occasional consultant.

To its credit, Java has more than lived up to its potential. Today, Java is used to build all types of products and systems, from enterprise class accounting systems to handheld phones and personal computers and everything in between.

While all this excitement is going on, the day-to-day responsibilities of an MIS shop continue. Applications need to be rolled out, existing systems supported, and an occasional enhancement delivered. Many of these systems use legacy tools, such as COBOL, and are fundamental to the operation of the business. The people who support these systems are valuable repositories of business process information, and they know how computer systems are used to meet those business process requirements.

Scores of organizations have decided to replace these legacy systems, including their core business systems, with more modern ones written in Java. Most could probably use a rewrite in any case, and the allure of using new technology, including a new language, will likely overwhelm more than one VP of MIS. Who better to build these new systems than the people who understand how the business works? Does it make sense to assign these essential projects to the person who is writing his or her first business system?

Some of Java's development efficiencies are available in any language, including COBOL. For example, coding styles that make use of subroutines implement important OO design principles. Many large, well-designed COBOL application development environments necessarily define and support good OO design techniques in the form of subroutines, copy members, and coding standards.

However, in procedural language environments like COBOL, the compiler or the runtime system does not directly support the OO developer. A case could be made that OO development environments primarily make the compile and run-

time tools aware of good, efficient coding techniques that have been in use for some time in traditional languages.

A natural language is the tool people use to communicate ideas to each other. Languages have a syntax and a vocabulary of terms that provide structure and organization to ideas. If one person adheres to the rules and provides a linguistic shape and organization to his or her ideas, it is possible to communicate with another person.

Computer languages perform very much the same role. Computer languages define a vocabulary and syntax structure that is suitable for a computer to understand. Programmers learn these rules and convert ideas into code. A programmer's effectiveness is largely determined by how well he or she can communicate with the computer. (Of course, there is the testing, training, documentation, and support parts of the job, but programmers would be better off reading Dr. Spock rather than Dr. Chomsky to gain insights here!)

When people learn a new natural language, they often compare the new terms and syntax with what is already familiar to them. Language learning materials often have glossaries or translations of terms. The new syntax is defined, in part, by comparing it to the student's "native" syntax. In fact, most people who learn a new language think in their native language and mentally translate into the other. Only when they are very comfortable with the new language will they finally think in the new language.

Instead of describing the Java object concepts and syntax in the abstract, or based on references using C or C++ programming languages, *Java for COBOL Programmers, Third Edition* presents various object-oriented concepts first in a COBOL context, and then in the Java syntax that supports this concept. Then, after you are familiar with the underlying concepts, additional Java language rules are defined.

For programmers trying to use Java in a real-world environment, this third edition has been updated to reflect the latest developments in the Java language and programming standards. Advances in the language, such as the new Generics feature, are covered. New alternatives for stream-based input and output processing methods are introduced, along with the latest XML processing options in Java.

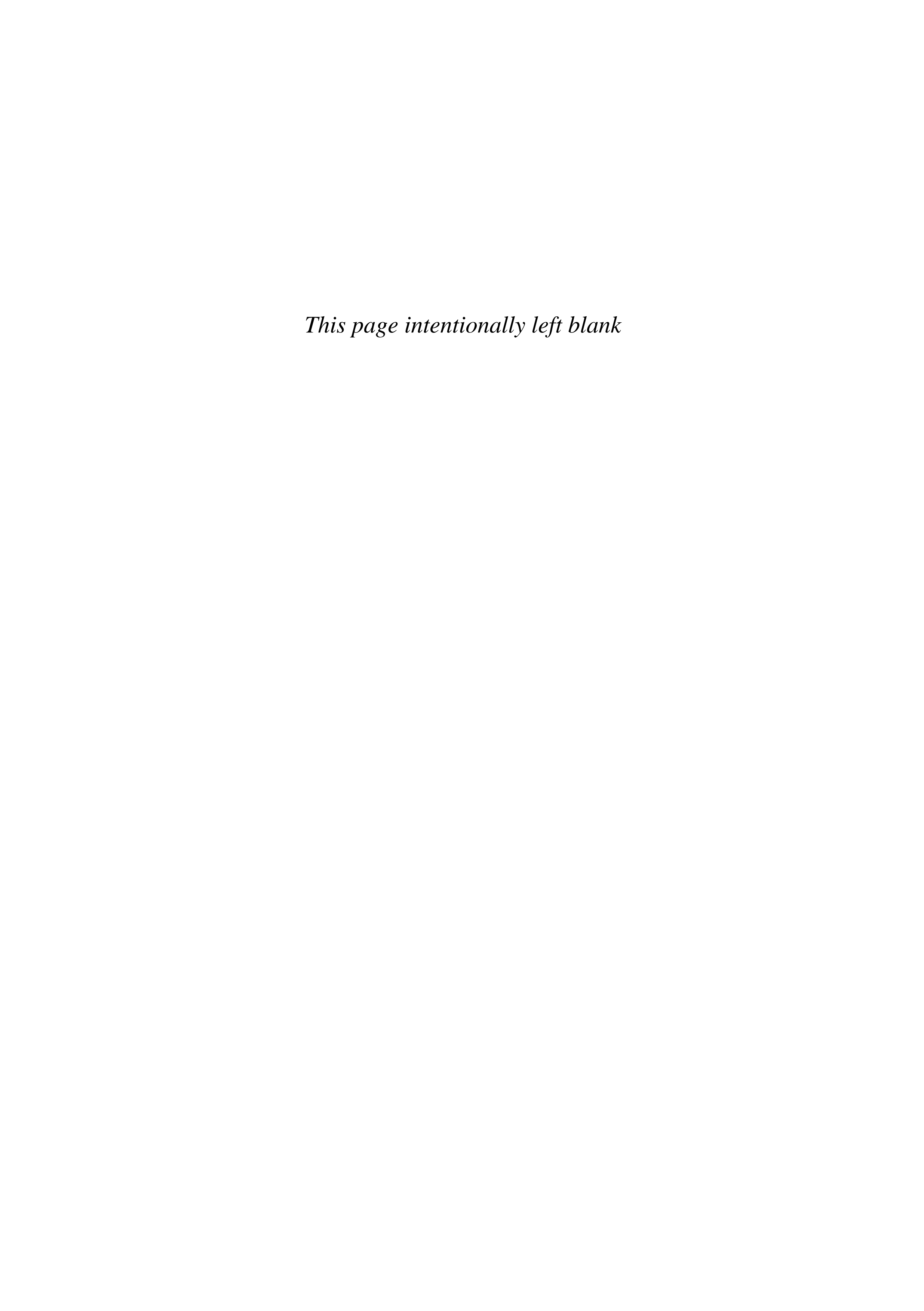
Also new in this edition is a chapter on the Eclipse graphical integrated development environment. This product is presented using a guided, step-by-step progression. You can use the chapter exercises to explore helpful Eclipse features, such as smart editing, debugging, and refactoring.

If you're a COBOL programmer and you've written a subroutine—a subroutine that was used by someone else—then you already understand the most important Java object concepts. Sure, the syntax is different and more powerful, but the principle that one person writes code so others can use it without having to understand all of its details is the core principle of many of the OO design objectives. You still

need to understand the user's needs, and specifications and documentation are still required. You may even be able to adjust end-user requirements based on technical issues, since it is expected that existing components will be reused. In theory, most development projects in an OO environment consist of collecting and shaping end-user requirements and then "assembling" solutions, using as many existing building blocks as possible.

Once you've decided to learn Java, the secret to success is to use the same process as when learning any new thing. Break down the information into manageable pieces, leverage what you already know, pick a good learning environment, and plunge ahead. You already know the hard part (i.e., how to translate business requirements into a computer language). You just need to learn a few new design principles, a new syntax, and some state-of-the-art integrated development environment. This book will help you get started.

Part I Introducing Java



This page intentionally left blank

1 Objects and Classes

In This Chapter

- The COBOL Subroutine
- Calling a Subroutine
- Terms to Review: Subroutines
- Objects and Java
- Terms to Review: Objects

Java's popularity is due to a number of factors. One of the biggest reasons is that it is a popular object-oriented language.

This sounds impressive, but what exactly is an object-oriented language? In fact, what is an object?

Simply put, an *object* is a collection of code organized to perform a function or simply to retain some information on behalf of another program. Objects are created and then used by programs to perform these functions on behalf of the other programs.

Object-oriented (OO) languages, and the object-oriented design approach, contain many ideas already familiar to you. Chapters 1 through 5 will start by describing these concepts, based on the COBOL language. I will then compare Java's definition of objects, and the syntax that supports it, to these concepts. This should help you acquire a good understanding of the basic object-oriented concepts.

The COBOL Subroutine

I'll start with the COBOL subroutine. A COBOL subroutine is a source file that contains COBOL code and implements a logical function. It is organized so that other programs can prepare the appropriate information, call the subroutine, and perform the function. Subroutine parameters are described in the LINKAGE SECTION of the subroutine. The subroutine is able to evaluate or modify passed parameters as part of its algorithm.

The calling program uses a subroutine when it defines and prepares the parameter items for the subroutine, and then calls it. The parameters are passed to the subroutine, using the CALL SUBROUTINE USING statement. After the subroutine completes its function, the calling program can examine the parameter items to see the information returned by the subroutine.

Calling a Subroutine

In Figure 1.1 the calling program (CALLER) prepares a text item as a parameter. It then calls the subroutine (MYSUB), passing this parameter and another parameter.

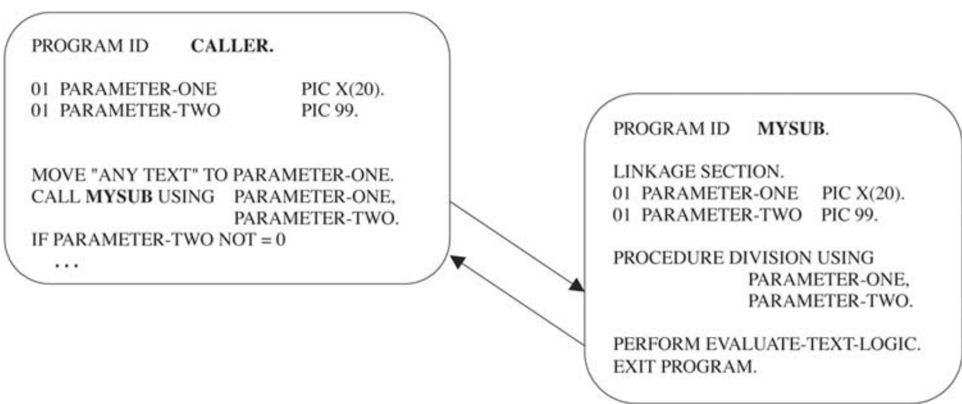


FIGURE 1.1
Calling a subroutine.

The subroutine accepts both of these parameters. Its algorithm is roughly defined as follows: Evaluate the first parameter and set the second, based on some predefined criteria. The details of the evaluation function (that is, the heart of the subroutine's algorithm) are embedded in the subroutine.

The calling program can now evaluate the return parameter, in effect using the evaluation logic of the subroutine. The calling program only needs to know how to call the subroutine and how to evaluate the result of that call. It does not need to know any other details of the subroutine's internal logic.

Objects behave in much the same manner. An object is a collection of code that accepts parameters, implements a function, and returns information to the calling program. Objects, however, differ from a standard COBOL subroutine in a number of ways. One important difference is that objects are created dynamically (at runtime) by a program. They are always associated with, or "tied to," the program that created them. Furthermore, a program can create many objects of the same type, or class.

An object can be understood as a subroutine called with a particular set of linkage items.

MYSUB COBOL

Suppose you've defined a subroutine as follows:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MYSUB.
***********************************************************************
*
* This routine accepts a text item as a parameter and evaluates the
*
* text. If the text is all spaces, MSG-SIZE will be set to 0. Else,
*
* MSG-SIZE will be set to 1.
*
* The text item will also be stored in the passed control structure in
*
* MSG-TEXT.
*
***********************************************************************
*
DATA DIVISION.
WORKING-STORAGE SECTION.
*   These are the subroutine parameter definitions.
LINKAGE SECTION.
01 MYSUB-CONTROL.
     03     MSG-TEXT     PIC X(20).
     03     MSG-SIZE     PIC 9(8).
```

```
01 TEXT-STRING                PIC X(20).
*   This is the interface definition for the subroutine.
PROCEDURE DIVISION USING MYSUB-CONTROL, TEXT-STRING.

MYSUB-INITIAL SECTION.
MYSUB-INITIAL-S.
*   Perform the subroutine's function. Test the passed string for spaces
*   and set MSG-SIZE accordingly.
     IF TEXT-STRING = SPACES
          MOVE 0 TO MSG-SIZE
     ELSE
          MOVE 1 TO MSG-SIZE.
     MOVE TEXT-STRING TO MSG-TEXT.
EXIT-PROGRAM.
     EXIT PROGRAM.
```

CALLER COBOL

Now, suppose you've written a calling program that uses this subroutine:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CALLER.
DATA DIVISION.
WORKING-STORAGE SECTION.
*   Create the parameter definitions.
01 MYSUB-CONTROL.
     03     MSG-TEXT     PIC X(20).
     03     MSG-SIZE     PIC 9(8).
01 TEXT-STRING                PIC X(20).

PROCEDURE DIVISION.

START-PROGRAM SECTION.
START-PROGRAM-S.
*   Prepare the parameters.
     MOVE "ANYTEXT" TO TEXT-STRING.
*   Call the subroutine.
     CALL "MYSUB" USING MYSUB-CONTROL, TEXT-STRING.
*   Evaluate the result.
     IF MSG-SIZE OF MYSUB-CONTROL = 0
        DISPLAY "MSG SIZE equals 0"
     ELSE
```

```
        DISPLAY "MSG SIZE equals," MSG-SIZE.

 EXIT-PROGRAM.
     EXIT PROGRAM.
     STOP RUN.
```

Let's examine these two programs from an object-oriented perspective, using the terminology of the object-oriented design methodology.

You can consider the subroutine MYSUB a class. That is, every time MYSUB is called—even if it is called from different programs—it will behave the same way. Any features, or logic, that MYSUB has will be available to all calling programs. At the same time, some parts of MYSUB are not available to the outside world. For example, any variables in MYSUB's WORKING-STORAGE are private to MYSUB. And the details of MYSUB's logic are not known to any calling programs; only its interface (or LINKAGE SECTION) is published.

You can consider any instance of the parameter item MYSUB-CONTROL in a calling program as an object after MYSUB has been called. That is, after MYSUB has performed its logic (at the request of a calling program), the result of that logic is available in MYSUB-CONTROL.

A calling program can examine or modify the contents of items in MYSUB-CONTROL (MSG-TEXT or MSG-SIZE) and perform some logic based on those contents. These items are called class data members, or *properties* in OO terminology.

Another program in the COBOL run unit can call MYSUB, with its own MYSUB-CONTROL (parameter), and evaluate the result. In this case, CALLER #1's copy of MYSUB-CONTROL will, of course, not be affected by CALLER #2. Each instance of a MYSUB-CONTROL area will now contain unique information. In this case, the unique MYSUB-CONTROL areas are objects. In fact, a single program can manage two separate MYSUB-CONTROL(s) as long as they have unique names.

CALLER COBOL: CONTROL

Suppose you've defined CONTROL areas for two subroutines as follows:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CALLER.
DATA DIVISION.
WORKING-STORAGE SECTION.
*   Create one set of parameter definitions.
01 MYSUB1-CONTROL.
     03     MSG-TEXT     PIC X(20).
     03     MSG-SIZE     PIC 9(8).
*   Create a second set of parameter definitions.
```

```
01 MYSUB2-CONTROL.
     03     MSG-TEXT     PIC X(20).
     03     MSG-SIZE     PIC 9(8).

01 DISPLAY-MESSAGE       PIC X(20).
01 TEXT-STRING           PIC X(20).

PROCEDURE DIVISION.

PROGRAM-START SECTION.
PROGRAM-START-S.
*   Prepare the first set of parameters, and call MYSUB.
     MOVE "ANYTEXT" TO TEXT-STRING.
     CALL "MYSUB" USING MYSUB1-CONTROL, TEXT-STRING.
*   Prepare the second set and call MYSUB.
     MOVE SPACES TO TEXT-STRING.
     CALL "MYSUB" USING MYSUB2-CONTROL, TEXT-STRING.
*   Evaluate the data associated with the first set and then the second
*    set.
     IF MSG-SIZE OF MYSUB1-CONTROL > 0
          MOVE MSG-TEXT OF MYSUB1-CONTROL TO DISPLAY-MESSAGE
     ELSE IF MSG-SIZE OF MYSUB2-CONTROL > 0
          MOVE MSG-TEXT OF MYSUB2-CONTROL TO DISPLAY-MESSAGE.

     DISPLAY "DISPLAY-MESSAGE: ", DISPLAY-MESSAGE.

 EXIT-PROGRAM.
     EXIT PROGRAM.
     STOP RUN.
```

You can consider all instances of MYSUB-CONTROL (after MYSUB has been called) to be class instances, or *objects*. In the example, MYSUB1-CONTROL is one object, and MYSUB2-CONTROL is a second. It is up to the calling program (the consumer of MYSUB) to manage these objects (i.e., the two instances of MYSUBx-CONTROL) as part of the application logic.

For example, if one MYSUBx-CONTROL contains an error message from the database system, and the other MYSUBx-CONTROL contains an error message from the communications system, it is up to the calling application to decide which one to display at the correct time.

```
*   Prepare the database message parameters, and call MYSUB.
     MOVE "Unable to connect to the DataBase." TO TEXT-STRING.
     CALL "MYSUB" USING MYSUB1-CONTROL, TEXT-STRING.
*   Prepare the communications message parameters, and call MYSUB.
     MOVE "Invalid connection request." TO TEXT-STRING.
     CALL "MYSUB" USING MYSUB2-CONTROL, TEXT-STRING.
*   Prepare the generic message parameters, and call MYSUB.
     MOVE "An unknown error has occurred." TO TEXT-STRING.
     CALL "MYSUB" USING MYSUB3-CONTROL, TEXT-STRING.

     ...
```

Later on in this program:

```
*   An error has occurred. The type of error has been recorded in
*   DISPLAY-MSG-TYPE-SW.
*   Evaluate which type of error occurred, and display the correct
error
*   message text item.
     IF DISPLAY-MSG-TYPE-SW = "D"
          MOVE MSG-TEXT OF MYSUB1-CONTROL TO DISPLAY-MESSAGE
     ELSE IF DISPLAY-MSG-TYPE-SW = "C"
          MOVE MSG-TEXT OF MYSUB2-CONTROL TO DISPLAY-MESSAGE
     ELSE
          MOVE MSG-TEXT OF MYSUB3-CONTROL TO DISPLAY-MESSAGE.
     PERFORM DISPLAY-ERROR-MESSAGE.
```

The previous code fragments are examples of how a COBOL program might use three objects. Each of these objects is based on the class MYSUB.

Terms to Review: Subroutines

Here are some of the concepts that have been discussed and how to understand them from a COBOL perspective.

Class: A subroutine is similar to a class. It can perform certain functions when called. The subroutine developer defines these functions. Many calling programs can use this subroutine in order to perform those functions.

Interface: The signature, or parameter specification, for a particular subroutine (or class, in OO terms). In COBOL, a subroutine's signature is the list of items in the subroutine's LINKAGE SECTION. Some items in an interface may be input parameters, and some may be result parameters, or both.

Object: An instance of a class, similar to an instance of the COBOL subroutine's CONTROL area, after the subroutine has been called. You can think of an object as the result of initializing the subroutine or calling it for the first time. This result is stored in the subroutine's CONTROL area.

Class data members: The data items that are associated with the subroutine (or class). Class data members include both the data elements in the subroutine's LINKAGE SECTION and the data elements in the subroutine's WORKING-STORAGE. Class data members are also called *properties* of the class.

Private: Any data elements (or properties) that belong to the class but are not available outside the class. In COBOL, the items in a subroutine's WORKING-STORAGE area are private. (This COBOL allegory is not precise; I will clarify it as I go.)

Public: Any data elements (or properties) that belong to the class but are available outside the class. They are similar to items in a COBOL subroutine's LINKAGE SECTION.

The elementary items in MYSUB-CONTROL (e.g., MSG-TEXT and MSG-SIZE) can be considered data members of the class for the following reasons:

- They are data elements that belong to the class definition. This means that they are only useful as part of the parameter definition for MYSUB. The items in MYSUB-CONTROL will behave correctly (i.e., MSG-SIZE will be set to 0 or 1) only after MYSUB is called.
- They are unique to each instance of the class. More than one instance of MYSUB-CONTROL can be defined and passed as a parameter to MYSUB. The items in any instance of MYSUB-CONTROL will contain information based on the last time MYSUB was called with that instance of MYSUB-CONTROL.
- They can be evaluated and/or set by both the calling program and the subroutine. The data items in the MYSUB-CONTROL define the interface to MYSUB. This means that a calling program can communicate with MYSUB by using the data items defined in MYSUB-CONTROL.

OBJECTS AND JAVA

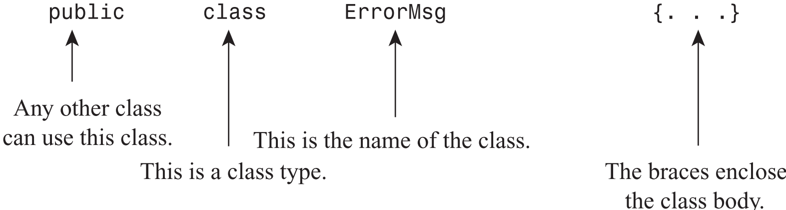
Let's examine what what's been discussed using Java's syntax.

ErrorMsg Class

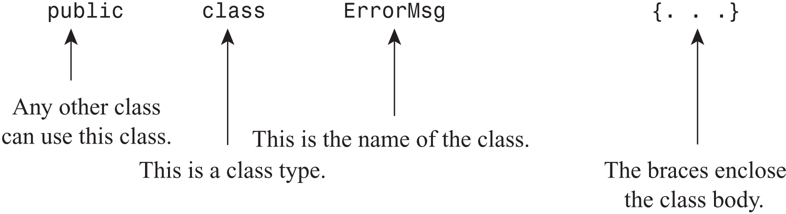
This is the outline of a Java class definition:

```
public class ErrorMsg {
     public String msgText;
     public int  msgSize;
          ...
 // Some logic
          ...
}
```

The first line defines the class.



These next two lines declare the class instance data members, or properties. These are associated with each instance of this class and can be of any valid type. In many ways, they are analogous to the data items in MYSUB-CONTROL.

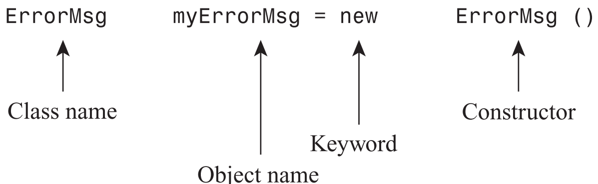


These two statements define the public data members for the class `ErrorMsg`.

The statements could be read this way: "The first data member is a `public` data member, its type is `String`, and its name is `msgText`." "The second data member is a `public` data member, its type is `int`, and its name is `msgSize`."

In order to use (or to call) this class, the consumer of this class (i.e., the caller) creates a new instance of the class with the `new` operation. This is very similar in concept to the COBOL example that defined several unique MYSUBx-CONTROL areas in calling the COBOL program.

Caller Class

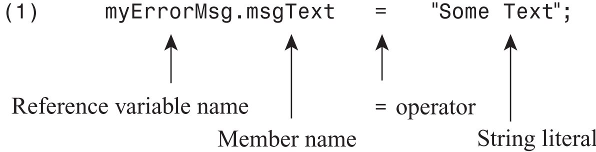


This statement could be read as follows: "Create a `new` object of class `ErrorMsg`, and give it the name `myErrorMsg`."

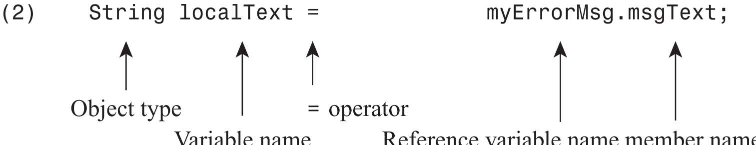
This statement allocates memory for the new object, calls its constructor (more on this later), and returns a reference variable (a kind of pointer) to this new object. In the example, the pointer to the new class is stored in the object reference variable `myErrorMsg`. This reference variable is managed by the calling program in a manner very similar to the way MYSUBx-CONTROL areas are managed in the COBOL program. Note that the class name and the constructor name are the same and that the constructor is called with an empty parameter list, indicating the default constructor.

Later, the program that contains this statement can use the reference variable name `myErrorMsg` to refer to any class data members that belong to this object. The period (.) operator is used to access members of an object. The member name is used to specify which member is being accessed.

For example, statement 1 will assign a string containing `"Some Text"` to the member variable `msgText`. The object that is modified is "pointed to" by the reference variable `myErrorMsg`.



Statement 2 is another example of how `myErrorMsg.msgText` can be used:



Statement 1 assigns the string `"Some Text"` to the data member `msgText`. The object whose data member is being set is, of course, `myErrorMsg`. This object is an object of type `ErrorMsg`. That is, it is an instance of the class `ErrorMsg`. Statement 2 assigns the current string in the data member `msgText` to a local string variable called `localText`.

Terms to Review: Objects

Here are some object-oriented concepts and how to understand them from a Java perspective.

Class: A Java class is a logical grouping of data and methods (methods are conceptually similar to functions) that use the data. In concept, a Java class is similar to a COBOL subroutine: It contains some data elements, it can perform functions when requested, and the subroutine developer defines these functions. Many calling programs can use this class to perform available functions and can manage the data that belong to the class.

Object: An instance of a class. This is similar to an instance of a COBOL subroutine and a unique set of LINKAGE AREA items. You can think of an object as the result of initializing the class or calling it for the first time. However, unlike a COBOL subroutine, many instances of a class can be easily created and managed by the same calling program.

Reference variable: A variable that contains a pointer to an object. After an object is created, the reference variable points to it. A reference variable is used by the calling program to access the data members and functions (that is, the methods) that belong to the object. This is similar to an instance of CALLER's MYSUBx-CONTROL area after the subroutine has been called.

New: The Java operation that creates an instance of the class (i.e., the object). It returns a reference variable that points to the new object.

Constructor: Constructors are invoked when an object is first created. Constructors are similar to methods, but are not considered real methods in Java. For example, constructors cannot be invoked directly.

Data members of the class: The data items or properties that are associated with the class. They include all of the data elements that are defined in the class. These variables are created at the same time each instance of a class is created. They normally belong to each instance of a class and are not shared by unique class instances.

Private: Any data elements (or properties) that belong to the class but are not available outside the class. Private data elements are identified with the keyword `private`. They are similar in this respect to items in a COBOL subroutine's WORKING-STORAGE area, since a calling program cannot directly access these items.

Public: Any data elements (or properties) that belong to the class but are available outside the class. Public data elements are identified with the access keyword `public`. They are similar in this respect to items in a COBOL subroutine's LINKAGE SECTION.

Now let's explore another object-oriented design principle and how it relates to some COBOL concepts.

NOTE

A calling program contains its objects.

Try to visualize for a moment what happens when a COBOL main program calls a subroutine. At runtime, and after the subroutine has been called, both the main program and the subroutine exist in memory. The executing program environment (the COBOL run unit) contains both the main program and the subroutine, as depicted in Figure 1.2.

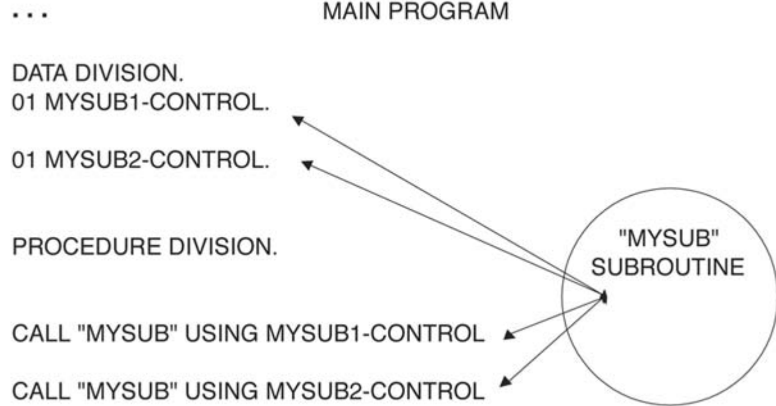


FIGURE 1.2
A COBOL run unit.

The items passed in the USING clause are the parameters to the subroutine. The subroutine can access any of the items that have been passed to it and can access items in its own WORKING-STORAGE area. The MYSUBx-CONTROL areas contain the results of the most recent call to MYSUB.

Note that while the main program can access items in the passed parameters, it cannot access any items in the subroutine's WORKING-STORAGE area. Further, Figure 1.2 shows two separate instances of MYSUBx-CONTROL data items but only one instance of the MYSUB subroutine. This means that all of the items in MYSUB's WORKING-STORAGE area will be shared, regardless of whether MYSUB1-CONTROL or MYSUB2-CONTROL is passed. Because of this limitation, the COBOL program does not behave exactly like an object.

In much the same way, a Java program contains any instances of the classes that it creates. The major difference between a COBOL subroutine and a Java class is that a Java program can contain many instances of its classes. These are called *objects* (see Figure 1.3).

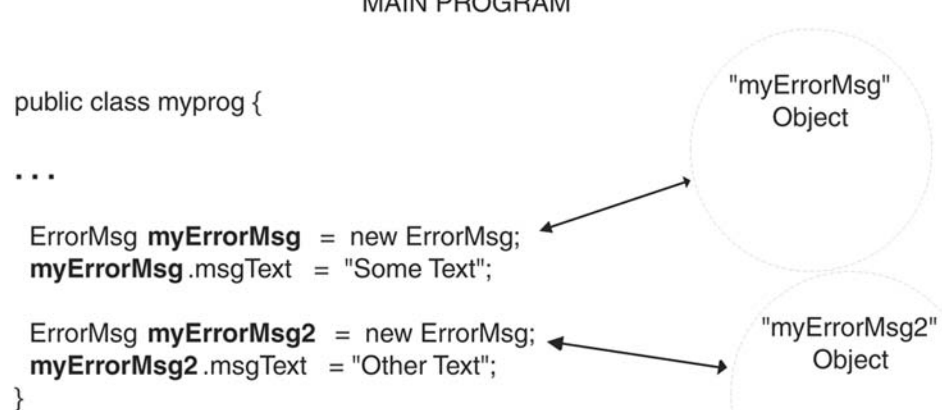


FIGURE 1.3
A Java run unit.

The `new` operation creates a completely new instance of the class in memory, including any private data members (WORKING-STORAGE items) and then loads the class code into memory, if it has not been loaded already. All of the data members defined for that class are created. A reference variable is returned from the `new` operation, and this reference variable, or *handle*, points to the new class instance. These class instances are called *objects*.

Like items in a COBOL routine's LINKAGE SECTION, public data members defined by the class can be accessed by either the main (calling) program or by the object itself. Unlike the COBOL subroutine's WORKING-STORAGE area, internal data members are not shared between instances of these two classes.

An object reference variable (for example, `myErrorMsg`) points to each unique instance of the class. The Java main program uses the reference variable to refer to the data members of a particular class instance in much the same way that COBOL's OF operator works (for example, MSG-TEXT OF MYSUB1-CONTROL). Therefore, the statement

```
myErrorMsg.msgText     = "Some Text";
```

could be compared to the COBOL statement

```
MOVE "Some Text" TO MSG-TEXT OF MY-ERROR-MSG.
```

It's time to interrupt this object-oriented presentation and write some code. Before you write your first program, however, let's take a moment to examine how Java programs are compiled and executed.

2 Introducing the Java Development Environment

In This Chapter

- Runtime Interpretation and Java Byte Codes
- Getting Started with Java's SDK
- Applets with SDK
- Classes and Filenames
- CLASSPATH
- CODEBASE
- Packages
- Inside a Package
- Name Collisions
- Packages and Filenames
- Compressed Packages
- Applications vs. Applets
- Reviewing the Samples

This chapter will introduce Java and the Java virtual machine (JVM or Java VM), the environment in which your Java programs will run.

Runtime Interpretation and Java Byte Codes

Java's designers had a number of primary design objectives. As you have seen, object orientation is one of them. Another is the premise that a program can be compiled on any machine and the output of the compiler simply moved to another machine, where it will execute without changes. This concept is captured in the Java mantra, "Write once, run anywhere."

In an Internet environment, the movement to the execution machine (an end user's PC, for example) is automatically performed by the browser without any special commands by the user. The net result is simplified administration of the applications and immediate access to any Java application for the end user.

To accomplish this, the Java compiler does not create *executable* code—meaning a program that runs natively on a given system. Instead, the Java compiler creates an *intermediate representation* of your program. This representation is somewhere between source code and native machine code. It is called *Java byte codes.* These byte codes are the content that is moved to a computer system at runtime to be executed on that system.

Since the byte codes are not in a format that is native to any system, they can't be executed directly on any system. Instead, a native program interprets the byte codes and performs the application functions expressed in the byte codes. This interpreter is called a *Java virtual machine*, because it creates a virtual system environment in which Java byte codes can execute.

This concept is not new. Interpreted languages such as LISP and SmallTalk have worked this way for years. Some cross-platform COBOL compilers (notably from MicroFocus and AcuCorp) also provide this same type of "instantaneous portability" feature, using a byte code or intermediate version of compiled COBOL programs. Also, Ruby on Rails, ASP.NET (where multiple languages can be used for the output of byte codes), PHP, PERL, Python, and so on provide some form of virtual machine environment. Figure 2.1 shows the compile and execution procedure for a simple Java application.

In theory, the only prerequisite for running a Java program on a system is that an appropriate Java virtual machine be available on that system. As you might suspect, the reality is somewhat different.

Operating system functions and some details of the way an application interacts with the operating system are not exactly the same across all operating systems. Java does a credible job of defining standards that shield most applications from these differences, but it does not cover *all* cases. As a result, some low-level application functions are not defined by Java, or they are left to the individual VM implementation to deliver.

A good example of these issues is the curious fact that Java doesn't define a universal (or "pure Java") notion of a standard output display device. The `System.out.println` statement in Figure 2.1 is not technically 100% pure Java, even though it appears in almost every Java reference manual! It is left up to the Java VM's implementation of the `System.out` object to decide what to do with `System.out.println`.

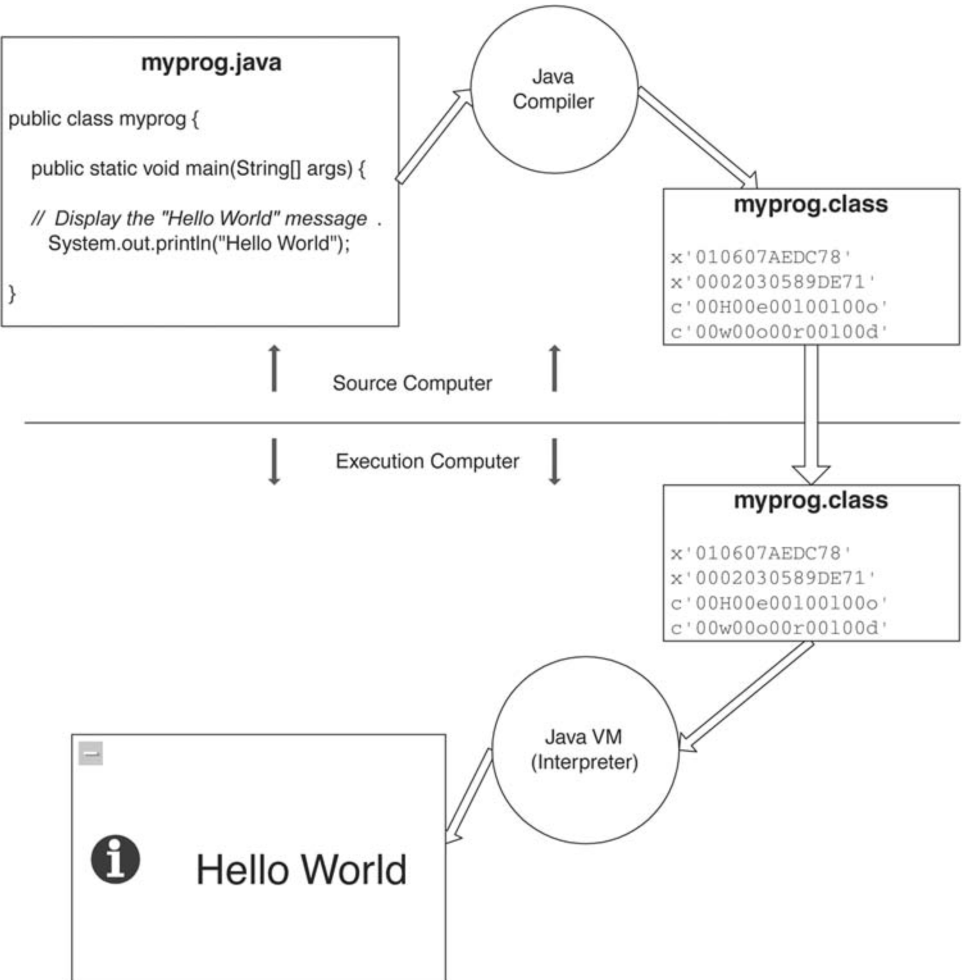


FIGURE 2.1
The Java compile and execution process.

To really complicate matters further, Java VMs are often executed as part of a browser application (such as Firefox or Microsoft's Internet Explorer). Java applications that are embedded in an Internet document (called Java *applets*) execute as part of the browser interface. The browser actually controls some of the details of how the application looks (such as the default font to use).

Java applets, in particular, are most subject to portability problems due to the differences in browsers and the various versions of the Java VMs used by the browsers. The implication is that a software provider should test a product on several different combinations of operating systems/browsers/Java VMs, to make sure the application

works correctly in all of the target execution environments. This has certainly been the case with many interactive Java applications. More importantly, Java applets are restricted to a very stern security model that limits their functionality.

Client-side Java applications running outside of the browser, on the other hand, are surprisingly portable. There are a number of class libraries available that define powerful mechanisms to build graphical user interfaces (GUIs). Sun provides the original Abstract Window Toolkit (AWT) libraries and the newer Java Foundation Classes (JFC)/Swing class graphical libraries, which run on all platforms. Any of these class libraries provide excellent mechanisms to build first-rate GUIs.

Another point to consider is that server-based Java applications are likely to be less susceptible to these variations, and most business logic is server-based. The majority of the differences between Java runtime environments arise from user interface issues. Since server applications don't have their own user interface, they are, by and large, unaffected by most of the runtime environment differences. Server applications will experience some variations in runtime environments, but these variations are fairly technical in nature (such as the mechanism to use for access to the native system), and normally they are easy to isolate.

Enough with the concepts. It's time to write some code. It's a good idea to mix reading about concepts with working on those concepts.

In order to write programs in Java, you will need a few tools. At a minimum, you will need a Java compiler, a JVM to interpret and execute your code, and a text editor. The text editor can be one of your choice, from Notepad on Windows systems to vi in a UNIX environment. The only requirement is that it create vanilla text files without any special formatting characters (such as a word processor might put into a file).

If you are lucky (and smart), you will use an integrated development environment (IDE) from a Java tool vendor. IDEs combine compilers, JVMs, a nice editor, and some type of project management tool into an integrated system. Sun and BEA sell excellent graphical Java development environments. Eclipse, the open source IDE project started by IBM, is used by most Java developers (www.sdtimes.com/content/article.aspx?ArticleID=30020). I will discuss Eclipse in Chapter 17.

The IDE tools take full advantage of a graphical operating system, such as Windows, Linux, and Solaris. These products take most of the grunt work out of the development experience, and they allow you to focus more time on your programming problem and less on tedious chores like managing files.

A popular nongraphical toolset is Sun's Java Software Development Kit (SDK), which is part of its Java platform solution. The SDK includes a free Java compiler and runtime environment and is available from Sun at www.java.sun.com. It is also included on the CD-ROM. You can just download this to your PC and use it. Of course, it doesn't have a graphical editor and is not a visually integrated development environment, but you can't beat the price.

Sun also makes a robust graphical development environment for Java called Java Studio. Sun makes several versions of this tool and distributes the NetBeans edition of the product for free. Finally, the CD contains the Eclipse open source IDE.

Learning how to use a new development environment can be a difficult step in moving from COBOL to Java. These initial exercises are geared more toward helping you become familiar with a new environment than reviewing the concepts already discussed. After you've mastered (or at least come to terms with) the development environment, you'll use these exercises to review the concepts presented. Therefore, you'll begin by concentrating on learning the development environment instead of understanding the code samples. (The code samples are explained and reviewed later.) To help you get started, step-by-step introductions to the SDK for the Java 1.6 platform are presented.

Commands that you are expected to execute (either with a mouse or by typing the command) are identified with the arrowhead marker at the beginning of the command.

Getting Started with Java's SDK

Sun's product has good introductory documentation available for it. An excellent one is on Sun's Java Web site. The following steps will guide you through the process of creating your first Sun-flavored Java application and applet.

First, install the SDK development environment. You can download it from the CD-ROM included with this book or from Sun's Web site. Follow the instructions on the installation Web page to install the software properly.

After it has been installed on your PC, the SDK utilities can be used from an MS-DOS command window. You can start up an MS-DOS window by using this menu path, beginning with the Windows task bar: Start > Run, then type in **CMD**.

In the command window, you can execute the Java compiler (javac.exe), the Java runtime (java.exe), and the applet viewer (appletviewer.exe). All these programs take parameters, such as the name of the Java file to be compiled. If you are using a Windows version before XP, you should also install the DosKey program (type `doskey` in the command window). This program remembers your DOS commands, which you can recall and edit using the cursor keys.

If the system cannot find the java.exe program after installation, you may need to add the directory that contains the SDK executables to your PATH environment variable. This variable controls how Windows finds programs to execute.

In Windows 98, you can use the sysedit function (menu path Start > Run > Sysedit) to edit the AUTOEXEC.BAT file. Assuming you have installed the SDK to the default directory (C:\Program Files\Java\jdk1.6.0_03), add a line to the end of the AUTOEXEC.BAT file that reads as follows: `SET PATH=%PATH%; C:\Progra~1\Java\jdk1.6.0_03\bin`. Changing the AUTOEXEC.BAT file requires that you reboot your PC.

On Windows ME, XP, Vista, 2000, and 2003, use the System icon on the Control Panel to modify the environment variable directly. Add the SDK path (`C:\Program Files\Java\jdk1.6.0_03\bin`) to the end of this variable. The source can be modified with any text editor, including WordPad. Java source files are standard text files, including line-ending sequences (CR, LF on PCs). WordPad is very popular for this task because you can save and compile the Java source without closing down WordPad. Just make sure you save the WordPad files as text files, rather than as document files. Document files contain additional text formatting information that is not appropriate for Java. Freeware and shareware text editors are available on the Web.

Now, it's time to get started.

1. Make a directory to contain your Java files. From the DOS window, you can create a directory on the C: drive by typing the following:

```
➔ cd C:\
➔ mkdir java4cobol
```

2. Change the current directory in your command window to the directory you just created:

```
➔ cd c:\java4cobol
```

3. Using a text file editor, enter this Java source:

```
public class HelloWorld {

     public static void main(String[] args) {
          System.out.println ("Hello World!");

     }
}
```

4. Save the source to a file named C:\Java4cobol\HelloWorld.java.

5. Compile your Java source with the SDK compiler (java.exe) in the DOS command window:

```
➔ javac HelloWorld.java
```

6. Your HelloWorld Java program should compile successfully. If it doesn't, there are three (or more) possible problems:
   - **Cannot find the javac program:** Check your PATH environment variable by using the set command in the DOS window. It should contain the directory where the program javac.exe is (on Windows, normally C:\Program Files\Java\jdk1.6.0_03\bin).
   - **Cannot find HelloWorld.java:** Check the spelling, and make sure you used the correct case. Are you in the directory where the source file was saved (C:\java4cobol)?
   - **The compile reported some errors:** Check your source file. It should be exactly as presented here.
   - Check the Sun Web site for additional suggestions.

7. Now run your program, using the Java runtime that comes with the SDK:

```
➔ java HelloWorld
```

8. If you've done everything correctly, your MS-DOS window should look like Figure 2.2.

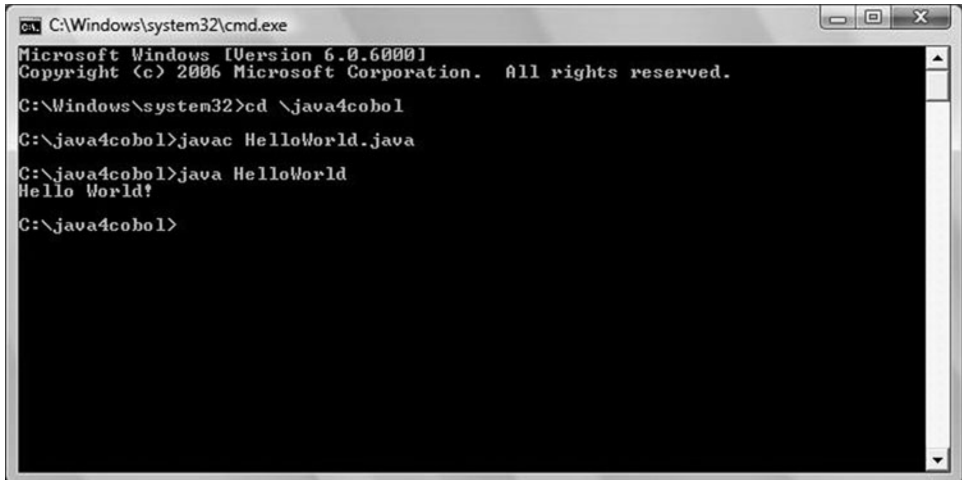


FIGURE 2.2
The Hello World! program.

Congratulations! You should be proud of your first Java program. It is full of potential.

Experiment with the program a little bit in this environment. You can make the changes in the text editor you have chosen. Compile it by typing **`javac HelloWorld.java`** in the command window. After your program is compiled, you can execute it from the MS-DOS command window by typing **`java HelloWorld`**.

- Change the message from "Hello World!" to "Watson come here!"
- What happens if you add a second `System.out.println` statement to your program?

APPLETS WITH SDK

Now you'll write your first Java applet in this environment and run it in the Applet viewer.

1. Using either Windows Explorer or a DOS command, create a new directory (or folder, in Explorer terminology) called "applet" in the java4cobol directory. Your applet files will be stored in this directory.
2. Using a text editor, create a Java source file named HelloWorld.java in this directory. The source file should contain these statements:

```
import java.applet.Applet;
import java.awt.Graphics;
public class HelloWorld extends Applet {
     public void paint(Graphics g)
     {
          g.drawString("Hello Applet World!", 5, 25);
     }

}
```

3. In the DOS command window, change the current directory to your applet directory:

```
➔ cd c:\java4cobol\applet
```

4. Compile your applet using the javac compiler:

```
➔ javac HelloWorld.java
```

Again, your applet should compile successfully at this point. If it doesn't, review the previous suggestions.

Applets are meant to be executed in a hosting environment, often a Web browser. Web browsers display information in HTML documents and are instructed to execute applets by special commands in the HTML document. The HTML command to execute an applet is the `<applet>` tag. This tag instructs the browser to run the applet specified in the applet tag.

You do not need to be an HTML expert for this step, but you will need to create a simple HTML document.

5. Using a text editor, create an HTML document named HelloWorld.html in your applet directory. The file should contain these statements:

```
<html>
  <head>
    <title>HelloWorld</title>
  </head>
  <body>
    <hr>
      <applet
        code=HelloWorld
        width=200
        height=200>

      </applet>
    <hr>
  </body>
</html>
```

   Remember to save this file under the name HelloWorld.html, and to save it as a text file, not as a document.

6. In the DOS command window, execute the APPLETVIEWER program to display your applet. Make sure you are in the applet directory.

```
➔   appletviewer HelloWorld.html
```

   If you've done everything correctly, your MS-DOS window and your applet viewer window should look like Figure 2.3.

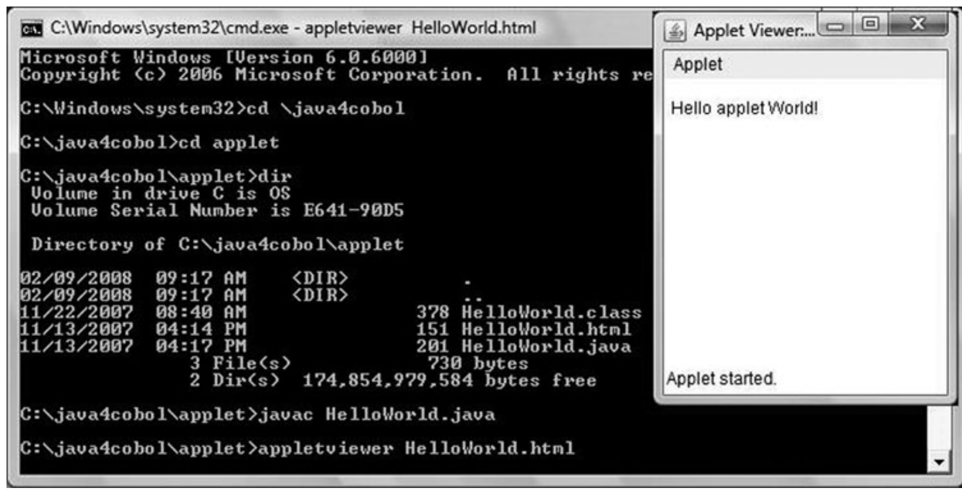


FIGURE 2.3
"Hello World!" in the applet viewer.

Experiment with this applet as well. You can make the changes in the text editor, then recompile and execute the program in the DOS command window. You will need to exit the applet viewer every time you run your program by clicking the X widget at the top right of the applet viewer window. Or, you can use the Reload function in the applet menu.

- Change the message from "Hello World!" to "Watson come here!"
- What happens if you add a second `g.drawString` statement to your program?
- Why do you think only one message showed up? (Hint: `drawString` is a graphically oriented function, whereas `println` is a line-oriented function.)
- What do you think the numbers 5, 25 control on the page?
- What happens if you change the 5 to a 6?
- What happens if you change the 25 to a 35?

Now that you've had some success with your first Java program, let's take a minute to discuss some of the rules of the road that a Java program must observe.

Classes and Filenames

Java source filenames and public Java class names must all have the same name. The compiler automatically creates a class file with the same name as the source file. Consider the first example class, `HelloWorld` (refer to Figure 2.1).

The name of this class (`HelloWorld`), its source filename (HelloWorld.java), and its output file (HelloWorld.class) all have the same name. Because of this, the runtime system can easily find the class file for HelloWorld when it is first needed by looking for a file named HelloWorld.class. Notice that the Java compiler needs the .java extension, but the runtime system assumes the .class extension.

CLASSPATH

But where does the runtime system look for the HelloWorld.class?

In most cases, this is controlled with the CLASSPATH setting. The CLASSPATH setting details a list of directories (or class archives, which can be jar or zip files) on the host operating system that contain classes. The Java runtime searches these directories (in the order they are listed) for your class files as they are required. This is similar to the way the PATH variable is used in Windows to find executable program files. The CLASSPATH variable can be set either as an environment variable (that is, a variable defined to the operating system) or as an optional argument to the runtime instance. The optional argument at runtime is the preferred approach because it doesn't mess up other programs' class paths.

For example, the standalone SDK runtime interpreter searches the c:\java4cobol directory for your classes, given this runtime option:

```
➔ java -cp c:\java4cobol HelloWorld
```

The `-cp` command argument tells the runtime to search in the c:\java4cobol directory for classes. The CLASSPATH argument can be a list of directories as well.

```
➔ java -cp c:\windows\java;c:\java4cobol HelloWorld
```

The CLASSPATH can also include zip or jar files (archive files built by the jar utility) that contain packages bundled by zip or jar utilities. I will say more about this later.

Normally, you will not need to include the standard runtime classes; Java provides a mechanism to find these automatically.

The system environment variable CLASSPATH works in much the same fashion as the CLASSPATH command line argument. It can contain a list of directories in which the runtime system can search for classes. Its use is discouraged, however,

since this variable applies to all Java runtimes on the system. As a result, coordinating the search requirements of multiple Java applications (especially if they are from various vendors) can be difficult.

Most development environments provide a way to assign a CLASSPATH variable to a project. You will need to consult the documentation for the development environment you are using to learn how to do this.

CODEBASE

Instead of the CLASSPATH setting, applets use the CODEBASE reference. This parameter specifies the directory that contains the initial applet, along with any supporting classes it requires. This is the syntax:

```
<applet
CODEBASE = codebaseURL
code=HelloWorld
width=200
height=200>
</applet>
```

As you might suspect, CODEBASEs are defined in terms of a uniform resource locator (URL) on the Web server. For security reasons, Java applet CODEBASE references must be on the same Web server (that is, must have the same domain name) as the HTML page that started the applet.

PACKAGES

A package is a way for the Java developer to group classes that are in some way related. For example, a `GeneralLedger` package would likely contain an `Account` class and a `Ledger` class. These classes might implement the business logic necessary to manage account information or to record a transaction in the `Ledger`.

These classes can define package variables and methods that would be available only to other classes in this package. In addition, public members are automatically made available to classes in the same package. Consequently, classes in a package do not need to import other classes in the same package.

Java defines a default package for classes that do not belong to an explicit package. You've already used the default package in your exercises. Until now, its primary benefit has been that it is a convenient mechanism to access the various example classes. But you could use a Java package to organize the examples.

To do this, you might define a `java4cobol` package. Simply include the package statement as the first uncommented line in your Java source file:

```
package java4cobol;

     public class ErrorMsg  {

          public String msgText;
          public int    msgSize;
               ...
       // Some logic
               ...
     }
```

Now the `ErrorMsg` class is part of the `java4cobol` package. The fully qualified name (FQN) of this class now becomes `java4cobol.ErrorMsg`.

Packages can have many parts to their names. For example, you could define a `java4cobol.message` package, and place all of your message-related classes into this package:

```
package java4cobol.message;

     public class ErrorMsg  {

          public String msgText;
          public int    msgSize;
               ...
       // Some logic
               ...
     }
```

In this case, the complete name of your `ErrorMsg` class would be `java4cobol.message.ErrorMsg`. Notice that `java4cobol` and `java4cobol.message` are different packages. Also, package names should be all lowercase to ensure they will work properly on all operating systems.

Inside a Package

Since all the classes in a package are related, it follows that they can take advantage of each other. For one, the default access control definition for classes, methods, and variables is package. This means that any one of these items that does not have an explicit access control qualifier is visible to all classes in the package.

Another advantage is simpler naming of class names in your code. If a class is part of a package, other classes in the package can refer to it using only its short name instead of its fully qualified name. For example, any class in the `java4cobol.message` package can refer to `ErrorMsg` as simply `ErrorMsg` instead of the more precise `java4cobol.message.ErrorMsg`.

A class from one package can still refer to classes in another package by using the fully qualified name. To continue the example, an `Account` class in the `general_ledger` package could refer to the `ErrorMsg` class by its `package.classname: java4cobol.message.ErrorMsg`.

```
package general_ledger;

    public class Accounts  {

        public String accountID;
        public BigDecimal    accountBalance;
            ...
// Create a new ErrorMsg object. Use the complete package name and class
// name.
         java4cobol.message.ErrorMsg accountNotFound =
                  new java4cobol.message.ErrorMsg ();
         ...
    }
```

However, this is more typing than the average programmer is willing to do. So, Java has defined a way for you to announce which classes your class will use, called the *import* statement, which must follow any *package* statements in your program. It tells the compiler that your class might refer to the classes identified in the import statement. The full class name is described only once in the import statement; afterward, the compiler knows to use the full name whenever it sees the short name.

```
package general_ledger;
import java4cobol.message.ErrorMsg;

    public class Accounts  {

        public String accountID;
        public BigDecimal    accountBalance;
            ...
// Create a new ErrorMsg object. No need to use the full class name or FQN.
        ErrorMsg accountNotFound = new ErrorMsg ();
            ...
    }
```

This type of import statement is officially named a *single-type-import* declaration statement.

For brevity's sake, a class can identify all of the classes in a package by using the asterisk (*) wild card instead of listing each class name. This statement instructs the compiler to import all the classes in `java4cobol.message` as they are needed, including `ErrorMsg`:

```
package general_ledger;
import java4cobol.message.*;
```

This type of import statement is officially named a *type-import-on-demand* declaration statement.

Although the import statement behaves in a manner similar to the COBOL COPY statement, it is really quite different. No code is actually imported, just the public class definitions. Significantly, the import statement tells the compiler to import the class file (the compiler output) and not the Java file (the source file). It is more of an aid to the compiler than it is a technique to copy code from another class.

Now that the `Accounts` class contains this import statement, it can refer to any class in the `java4cobol.message` package using only the class name. You do not need to use the full `package.classname` syntax.

Name Collisions

When using the wild card import statement, it is possible for a class to have the same name in two separate packages, or for your program to create a name that collides with a name in an imported package. In this case, the package name can be used to uniquely identify a class in a package:

```
package general_ledger;
import java4cobol.messages.*;

     public class Accounts  {
// Create a String object with the same name as the class ErrorMsg.
         String ErrorMsg;

         public String accountID;
         public BigDecimal    accountBalance;
                ...
```

```
    // Create a new ErrorMsg object. Specify the full package name.
          java4cobol.messages.ErrorMsg accountNotFound =
                new java4cobol.messages.ErrorMsg ();
                ...
    // Copy the msgText from the ErrorMsg class to the local ErrorMsg String
    // variable.
          ErrorMsg = java4cobol.messages.ErrorMsg.getErrorMsg();
       }
```

PACKAGES AND FILENAMES

Java packages and the directories on your computer's file system are closely related. As a rule, any package name must have a corresponding directory name on the host operating system, and it is case sensitive. This applies to both the development environment and the runtime (deployment) environment.

The name of a package maps to its filename location. In this example, you would expect to find a directory named ..\java4cobol\messages and ..\general_ledger (on UNIX systems, the slashes go the other way: .. /java4cobol/messages and .. /general_ledger). Notice that the introductory periods (..) in the package names (`java4cobol.message`) are replaced with the correct directory separator character (java4cobol\message or java4cobol/message).

The package name does not normally start at the root directory of the host file system. Instead, Java uses the CLASSPATH variable to point to a list of initial directory names, which are then searched for specific package directory names as well as class names.

In the example, if you set CLASSPATH to `c:\windows\java; c:`, the compiler will look in both the C:\windows\java directory and the root directory of the C: drive for a directory named java4cobol\messages. In this directory, it will look for a file named ErrorMsg.class.

It is likely that you will use packages from other vendors, or perhaps you will create packages that some other development or deployment environment will use. Since it is not realistic to coordinate class names or package names in advance, a convention is often used to group packages based on the origin of the package. Many package names are prefixed with the Internet address (domain name) of the provider, but in reverse order. Sometimes the .com portion of the domain name is left out.

For example, if an organization's domain name is mycompany.com, the `java4cobol.message` packages delivered by this company will likely be in a directory named com.mycompany.java4cobol.message, or mycompany.java4cobol.message.

Your source files should use the same conventions, but with a different initial directory so that source files and class files are kept separate. In this way, you can ship your class files and not your source files. As applied to the example, the class files would be in classes\mycompany\java4cobol\messages, and the source files (with the .java extension) in source\mycompany\java4cobol\messages.

Compressed Packages

Finally, the Java compiler and the Java runtime system can read compressed (or zipped) files that might contain many classes. Think of these compressed files as the contents of a directory (including perhaps several subdirectories) packaged into a single file. Microsoft's standard for packaging compressed classes is their cab technology; other vendors use the jar file technology for this purpose. This discussion focuses on the jar file convention, although many of the benefits and techniques would apply equally to cab files.

You can add a jar file to your CLASSPATH like this:

```
➔ java -cp c:\java4cobol.jar HelloWorld
```

In this, the Java runtime program would look in the jar file to find the class `HelloWorld`. Jar files provide several benefits, including these:

- Related classes can be organized and managed in a single archive. This simplifies the administration of the classes.
- A single request (for the jar file) can copy many classes from one system to another. This is especially important in an Internet browser, where a single HTTP request for the jar file will replace several individual HTTP requests (one for each class in the package). The browser issues HTTP requests to copy files from the Web server to the browser. Combining several requests into a single request is a significant performance benefit.
- The classes in the jar file are compressed (using the zip compression format). This can improve download performance and reduces disk space requirements.
- Individual classes can be digitally signed by their author and checked at runtime for authenticity. Only the single file needs to be checked, instead of each component in the file.

APPLICATIONS VS. APPLETS

Java programs run in one of three contexts: as an applet, as an application, or as servlets. I discuss server-side Java programs in more detail toward the end of this book in Part III, "Introducing Enterprise Java."

As the name suggests, an *applet* is an application fragment. It is not a complete application but rather an extension of another application. Most often, the application extended is a browser, such as Firefox or Microsoft's Internet Explorer. In fact, any application can be extended with Java applets. The only requirement is that the application to be extended must provide a Java runtime environment.

Conversely, a Java *application* is a complete application. It can run as a standalone application and independent of any other application. A Java application still needs a runtime interpreter, but this interpreter is a standalone program whose only responsibility is to execute and support Java applications.

A standalone interpreter is often a native program, suitable for executing on the host platform. For example, Sun's standalone interpreter is named java.exe in Windows, and java in every other environment, and it is executed like any other application. Sun provides a version of the Java program for Windows, Linux, and the Solaris operating systems. The only job of the standalone interpreter is to execute the Java byte codes and to perform the functions requested by the application.

REVIEWING THE SAMPLES

I kind of rushed through some of the code and the changes made to it in an effort to explain the development environment issues, so let's take a minute to review, starting with the first application.

- Your first program was a standalone *application*. That is, it only needed a runtime interpreter to execute, which was java.exe.
- The main program (the class that gets the ball rolling) in a Java application must have a specific interface. It must have a public method with this method definition:

```
public static void main(String[] args)
```

- This method accepts a single parameter (named `args` in the example). This parameter must be defined as an array of Strings (`String[]`). The `String` array represents the runtime arguments to the program (as in `jview HelloWorld argument1 argument2`).

- The class that contains this method (`HelloWorld`, in the example), does not need to be any particular name, but it may need to be public, depending on the requirements of the runtime environment. It also must be a *static* class, meaning only one instance of the class can exist at a time.
- The internal name of the class (in the source program HelloWorld.java) and the external name of the class (the HelloWorld.class file built by the compiler) must be the same.
- A Java application run in this manner does not expect to have a GUI environment available to it, but it can. It does expect to be able to write to a logical file, known as *standard out*, and to read from *standard in*. These files are more or less the equivalent of the ACCEPT and DISPLAY verbs that interact with the CRT in COBOL programs.
- For most Windows-based Java interpreters, standard out and standard in are directed toward an MS-DOS command window. Java applications run this way are not graphical applications.
- The statement `System.out.println ("Hello World!");` writes to the console, or standard out. It performs the `println()` method in a built-in object named `System.out`. This method accepts a `String` input parameter and writes it to standard out. Think of it as analogous to this COBOL statement:

```
DISPLAY "Hello World!".
```

- Subsequent calls to `println` write a new line of text to standard out.

The next program you wrote was a Java *applet*.

- An applet expects to be supported by some type of graphical hosting environment.
- Most often, the graphical hosting environment is a browser.
- Some standalone Java runtime environments also provide a built-in graphical hosting environment, or *applet viewer*. For the SDK, this was APPLETVIEWER.EXE. The applet viewer is an excellent tool for testing your applets.
- The main program (the program that gets the ball rolling) with a Java applet is the hosting environment (the browser or the applet viewer). It calls the `paint()` function in your class so that you can display information to the screen. The `paint()` function must have a specific interface. It must have a `public` method with this method definition:

```
public void paint(Graphics g)
```

- This method accepts a single parameter. It is a `Graphics` object (named `g` in the example).

- The class that contains this method (`HelloWorld`, in this example) does not need to be any particular name, but it does need to be public. This class must be of the sort that *extends*, or inherits, from a class named `Applet`.
- The statement `import java.applet.Applet;` at the beginning of the class tells the Java compiler where to find the `Applet` class.
- The internal name of the class (in the source program) and the external name of the class (the HelloWorld.class file built by the compiler) must be the same.
- A Java program run in this manner expects to have a GUI environment available to it. It also expects to be able to write to a logical file known as standard out and to read from standard in.
- Standard out and standard in are not well defined for these types of programs.
- The statement `g.drawString("Hello World!", 5, 25);` writes to the graphical environment. It performs the `drawString` method in the `Graphics` class. This method accepts a `String` input parameter and writes it to the graphical environment. There are additional controls to this method. In the example, I've specified the starting position in the graphical window to place the `String`. Think of it as analogous to this type of COBOL statement, extended with the ability to position the text anywhere on the screen:

```
DISPLAY "Hello World!" AT COLUMN 5 LINE 25
```

- The statement `import java.awt.Graphics;` at the beginning of the class tells the Java compiler where to find the `Graphics` class.
- Subsequent calls to `drawString` write text to the specified place in the graphical window.

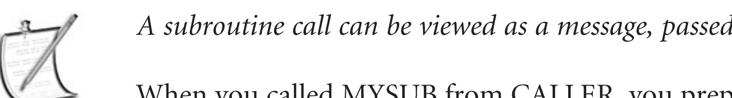
```
g.drawString("Hello Applet World!", 5, 25);
g.drawString("Hello Applet World!", 5, 25);
g.drawString("Hello Applet World!", 5, 25);
g.drawString("Hello Applet World!", 6, 25);
g.drawString("Hello Applet World!", 5, 25);
g.drawString("Hello Applet World!", 5, 35);
```

3 Messages and Methods

In This Chapter

- MYSUB COBOL
- CALLER COBOL
- Messages in Java
- Multiple Messages
- Method Overloading in COBOL
- Terms to Review
- Exercises: Classes, Objects, and Methods
- Reviewing the Samples
- HelloWorld: The Application
- HelloWorld: The Applet
- ErrorMsg: The Class

Let's return to the object-oriented concept discussion. You can extend the COBOL example to explore some other Java concepts.



A subroutine call can be viewed as a message, passed from one object to another.

When you called MYSUB from CALLER, you prepared some information (in TEXT-STRING) and asked MYSUB to evaluate it. In object-oriented (OO) terms, this is sometimes referred to as *message passing*. That is, CALLER passed a particular type of message to MYSUB.

In Java, objects can send messages to other objects. In almost all respects, this is similar to a subroutine call. However, objects typically support more than one type of message. COBOL subroutines can do this, too, but it takes a little planning.

In the example so far, MYSUB can only support one message type (i.e., evaluate TEXT-STRING). Suppose you want MYSUB to support multiple functions that are in some way related. You can do this by extending the MYSUB-CONTROL to include an ACTION-SWITCH. This item is used by the calling program to specify the function requested.

MYSUB COBOL

MYSUB is extended to add an action switch to its interface.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MYSUB.
DATA DIVISION.
WORKING-STORAGE SECTION.
***********************************************************************
* This routine accepts a text item as a parameter and evaluates the    *
* text. If the text is all spaces, MSG-SIZE will be set to 0. Else,    *
* MSG-SIZE will be set to                                               *
* Only if requested, the text item will also be stored                 *
* in the passed control structure.                                      *
***********************************************************************
LINKAGE SECTION.
01 MYSUB-CONTROL.
    03   MYSUB-ACTION-SWITCH     PIC X.
         88 MYSUB-ACTION-EVALUATE          VALUE "E".
         88 MYSUB-ACTION-SET-AND-EVALUATE  VALUE "S".
    03   MSG-TEXT                PIC X(20).
    03   MSG-SIZE                PIC 9(8).

01 TEXT-STRING              PIC X(20).

PROCEDURE DIVISION USING MYSUB-CONTROL, TEXT-STRING.

MYSUB-INITIAL SECTION.
MYSUB-INITIAL-S.
*   Perform the subroutine's function.
     IF TEXT-STRING = SPACES
          MOVE 0 TO MSG-SIZE
     ELSE
          MOVE 1 TO MSG-SIZE.
*   Evaluate if this additional function was requested.
*   If yes, perform it (i.e., save the string in TEXT-STRING).
```

```
          IF MYSUB-ACTION-SET-AND-EVALUATE
               MOVE TEXT-STRING TO MSG-TEXT.

     EXIT-PROGRAM.
          EXIT PROGRAM.
```

Using this interface definition, the CALLER program can ask MYSUB to store the TEXT-STRING in MSG-TEXT. It can also ask MYSUB to evaluate TEXT-STRING in order to determine whether it contains only spaces. The result is stored in MSG-SIZE.

CALLER COBOL

Alternatively, the CALLER can ask MYSUB to simply perform the evaluation and not store the TEXT-STRING in MSG-TEXT. This would be a slight variation on MYSUB's basic function. CALLER can then request either of these two functions by setting ACTION-SW to the appropriate value. Here is an example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CALLER.
DATA DIVISION.
WORKING-STORAGE SECTION.

01 MYSUB-CONTROL.
    03   MYSUB-ACTION-SWITCH          PIC X.
         88 MYSUB-ACTION-EVALUATE             VALUE "E".
         88 MYSUB-ACTION-SET-AND-EVALUATE     VALUE "S".
    03   MSG-TEXT                     PIC X(20).
    03   MSG-SIZE                     PIC 9(8).

01 TEXT-STRING                   PIC X(20).

PROCEDURE DIVISION.
START-PROGRAM SECTION.
START-PROGRAM-S.
*   Set ACTION-SWITCH to the function required (set and evaluate).
     MOVE "S" TO MYSUB-ACTION-SWITCH.
*   Prepare the text parameter to this subroutine
     MOVE "ANYTEXT" TO TEXT-STRING.
*   Call the subroutine to perform this function
     CALL "MYSUB" USING MYSUB-CONTROL, TEXT-STRING.

  DISPLAY "MSG-TEXT: ", MSG-TEXT,
           "MSG-SIZE: ", MSG-SIZE.
```

```
EXIT-PROGRAM.
    EXIT PROGRAM.

    STOP RUN.
```

You may have noticed that you separate MYSUB's parameters into two types: items in the CONTROL-AREA and other parameter items. This is done by design in order to distinguish parameters in the CONTROL-AREA (these are analogous to class data members) and other parameters (analogous to parameters passed to a class's methods, or functions). You will expand on this convention as you go.

MESSAGES IN JAVA

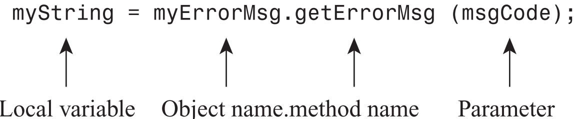
Once again, let's compare what you've done in COBOL to the way you would do it in Java.

ERRORMSG CLASS

Let's define a method in the `ErrorMsg` class.

```
public class ErrorMsg {

     public String msgText;
     public int  msgSize;

     public void setErrorMsg (String inputMsg) {
          ...
 // Some logic
          ...
          ;
     }
}
```

The statement that contains `setErrorMsg` describes a public entry point (i.e., a *method*) to the class `ErrorMsg`. These methods can be called by other classes to perform some function in this class.



The statement could be read this way: “Define a `public` method for class `ErrorMsg`. This method returns no data (`void`) and its name is `setErrorMsg`. This method accepts one parameter of type `String`; the parameter’s name as used by the class is `inputMsg`.”

Caller Class

A consumer (calling) class can use this method in the following way:

```
// Create a new object of type ErrorMsg.
     ErrorMsg     myErrorMsg = new     ErrorMsg ();
     ...
// Call the method in the object called setErrorMsg.
```

```
myErrorMsg.setErrorMsg ("Some Text");
```

Object name.method name   Parameter

This statement calls the object’s method and requests that it perform some behavior based on the passed parameter. Sometimes this is described as sending a *message* to that object.

Multiple Messages

Classes can easily support more than one message definition or even variations on a single message. In fact, it is this easy-to-use message specification that helps distinguish OO languages from their more procedural cousins.

Class ErrorMsg

Let’s look at some examples that show how a class publishes the types of messages it can receive using Java’s syntax.

```
     public class ErrorMsg {

          public String msgText;
          public int  msgSize;

(1)          public void setErrorMsg (String inputMsg) {
                    ...
          // Some logic
                    ...
// The method is complete. Return to the caller.
                    return;
             }

(2)          public String getErrorMsg () {
                     ...
            // Some logic
                     ...
// The method is complete.
// Return a String variable as the result (or return argument) of this
// method.
                    return (returnMsg);
             }

(3)          public String getErrorMsg (int msgCode) {
                   ...
        // Some logic
                   ...
                   return (returnMsg);
             }
        }
```

The new message definition statements (2 and 3) could be read this way: "Define a `public` method for class `ErrorMsg`. This method returns a `String` data item, and the method's name is `getErrorMsg`. One form of this method accepts no parameters (statement 2), and another form accepts one parameter of type `int` (statement 3)."

The method `getErrorMsg` defines other functions that `ErrorMsg` can support. As such, these are additional components of the *interface* to `ErrorMsg`. These methods can access any of `ErrorMsg`'s data items or methods (both public and private ones). Since the methods are themselves public, any class can call these methods.

Notice that the methods named `getErrorMsg` have different signatures (interface definitions) than the `setErrorMsg` method. They do not have a `String` input parameter, but they do return a `String` variable.

Notice also that the method `getErrorMsg` is defined twice in `ErrorMsg`. The first method specification accepts no parameters, and the second accepts one parameter of type `int`. This parameter's name (as used internally by the method) is `msgCode`.

Method Overloading

Having two variations on the same method is an example of *method overloading*. This is another mechanism to request a specific function in the class `ErrorMsg`. Methods with the same name can be defined to require variable numbers or different types of parameters. Consumer (calling) classes can request any of these public methods. The compiler will examine a method call, including its input parameters and return type, and call the correct public method of the class.

Caller Class

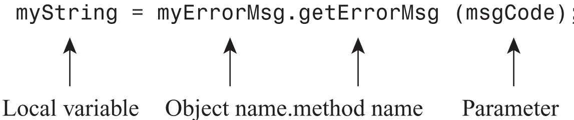
After the `ErrorMsg` class has been properly compiled, a consumer class can use these various methods in the following way:

```
// Define an instance of the ErrorMsg class.
     ErrorMsg    myErrorMsg = new    ErrorMsg ();
// Define a few local variables.
     int msgCode;
     String myString;
     ...
// Get the error message from myErrorMsg; store the result in myString.
```

```
myString = myErrorMsg.getErrorMsg ();
```

Local variable   Object name.method name

```
// Get the error message from myErrorMsg based on the integer value in
// msgCode.
```



METHOD OVERLOADING IN COBOL

There is no direct analogy to method overloading in COBOL, but there is a coding technique that is pretty close. COBOL compilers will generally let you call a subroutine and pass fewer parameters than specified in the USING statement. Suppose you designed a subroutine where some actions performed by the subroutine might require two parameters, and other actions might require three. Calling programs can pass either two or three parameters but would be responsible for passing the correct number of parameters, based on the action requested. It would then be the responsibility of the subroutine to make sure that no missing parameters are accessed by the subroutine during this particular call. Parameters that have been passed can be accessed by the subroutine, but the subroutine designer must be careful not to perform any statement that accesses an item defined in LINKAGE SECTION but not passed by the caller. In fact, some compilers support this technique explicitly by providing a mechanism to detect (at runtime) the number of passed parameters. In some systems, you may have to call an Assembler program to detect the number of parameters.

MYSUB COBOL

This example manages different numbers of parameters. It uses the "GET$NARGS" function as provided by the AcuCorp COBOL compiler.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MYSUB.
************************************************************************
* This routine accepts a text item as a parameter and                  *
* evaluates the text. If the text is all spaces,                       *
* MSG-SIZE will be set to 0. Else, MSG-SIZE                            *
* will be set to 1.                                                    *
* If requested, the text item will also be stored in the               *
* passed control structure.                                            *
* If the text item is not passed, then MSG-TEXT                        *
* will be evaluated instead.                                           *
************************************************************************
DATA DIVISION.
WORKING-STORAGE SECTION.

01 ARGUMENT-COUNT          PIC 9.
01 LOCAL-TEXT              PIC X(20).
```

```
LINKAGE SECTION.
01 MYSUB-CONTROL.
    03   MYSUB-ACTION-SWITCH     PIC X.
         88 MYSUB-ACTION-EVALUATE              VALUE "E".
         88 MYSUB-ACTION-SET-AND-EVALUATE     VALUE "S".
    03   MSG-TEXT                 PIC X(20).
    03   MSG-SIZE                 PIC 9(8).

01 TEXT-STRING              PIC X(20).

PROCEDURE DIVISION USING MYSUB-CONTROL, TEXT-STRING.

MYSUB-INITIAL SECTION.
MYSUB-INITIAL-S.
*   Perform some function to detect the number of arguments.
     PERFORM GET-ARGUMENT-COUNT.
*   Determine whether TEXT-STRING or MSG-TEXT should be evaluated.
Store
  the correct item in LOCAL-TEXT.
     IF ARGUMENT-COUNT = 2
          MOVE TEXT-STRING TO LOCAL-TEXT
     ELSE
          MOVE MSG-TEXT TO LOCAL-TEXT.
*   Now, use LOCAL-TEXT in the subroutine's logic.
    IF LOCAL-TEXT = SPACES
          MOVE 0 TO MSG-SIZE
     ELSE
          MOVE 1 TO MSG-SIZE.

     IF MYSUB-ACTION-SET-AND-EVALUATE
          MOVE LOCAL-TEXT TO MSG-TEXT.

 EXIT-PROGRAM.
     EXIT PROGRAM.

 GET-ARGUMENT-COUNT SECTION.
 GET-ARGUMENT-COUNT-S.
*   Set ARGUMENT-COUNT to the result.
    CALL "GET$NARGS" USING ARGUMENT-COUNT.
```

In this example, there are really two interfaces defined for MYSUB, one with a single parameter and another with two parameters. Calling programs can use either interface, depending on their requirements.

CALLER COBOL

Now, CALLER can pass one or two parameters.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CALLER.
DATA DIVISION.
WORKING-STORAGE SECTION.

01 MYSUB-CONTROL.
    03   MYSUB-ACTION-SWITCH    PIC X.
         88 MYSUB-ACTION-EVALUATE          VALUE "E".
         88 MYSUB-ACTION-SET-AND-EVALUATE  VALUE "S".
    03   MSG-TEXT               PIC X(20).
    03   MSG-SIZE               PIC 9(8).

01 TEXT-STRING             PIC X(20).
...
PROCEDURE DIVISION.
START-PROGRAM SECTION.
START-PROGRAM-S.
*   Set ACTION-SWITCH to the function required (set and evaluate).
     MOVE "S" TO MYSUB-ACTION-SWITCH.
*   Clear the two MYSUB control items, to make sure you see the result.
     MOVE SPACES TO MYSUB-TEXT.
     MOVE ZEROS TO MYSUB-SIZE.
*   Call MYSUB with two parameters.
     MOVE "ANYTEXT" TO TEXT-STRING.
     CALL "MYSUB" USING MYSUB-CONTROL, TEXT-STRING.
     DISPLAY "MSG-TEXT: ", MSG-TEXT,
             " MSG-SIZE: ", MSG-SIZE.
*   Set ACTION-SWITCH to the function required (evaluate only).
     MOVE "E" TO MYSUB-ACTION-SWITCH.
*   Clear the two MYSUB control items, so you are sure to see the *
*   result.
     MOVE SPACES TO MYSUB-TEXT.
     MOVE ZEROS TO MYSUB-SIZE.
*   Call MYSUB, passing only one parameter.
*         The stored string in MSG-TEXT will be evaluated.
     CALL "MYSUB" USING MYSUB-CONTROL.
     DISPLAY "MSG-TEXT: ", MSG-TEXT,
             " MSG-SIZE: ", MSG-SIZE.
*   Try it with some text in MSG-TEXT.
     MOVE "Some Text" TO MYSUB-TEXT.
```

```
           MOVE ZEROS    TO MYSUB-SIZE.
      *   Call MYSUB, passing only one parameter.
      *   The stored string in MSG-TEXT will be evaluated.
           CALL "MYSUB" USING MYSUB-CONTROL.

           DISPLAY "MSG-TEXT: ", MSG-TEXT,
                  " MSG-SIZE: ", MSG-SIZE.

       EXIT-PROGRAM.
           EXIT PROGRAM.
           STOP RUN.
```

I have demonstrated how a developer can support multiple interfaces to a single COBOL subroutine. This avoids the problem of having to write and support two separate yet similar routines; a single routine can perform both roles. However, care must be exercised in both building and using the routine to make sure that the proper interface is being used at the proper time. A calling program can easily cause a runtime error by requesting one type of interface but not passing the proper number of parameters.

Java's compiler and runtime system support this technique more explicitly and with much more functionality. The developer does not need to worry about which interface will be called and which parameter(s) may or may not be available; instead, the compiler automatically takes care of these details.

Remember the distinctions that have made between MYSUB's CONTROL-AREA and the other parameters passed to MYSUB? In these examples, every call to MYSUB must pass the CONTROL-AREA, regardless of the other parameters passed. Only the other parameters are optional.

This is similar to the way Java distinguishes between member data items and passed parameters. Class member items are always available, regardless of the interface used. For example, one method may require a single integer parameter and return a `String` data type, whereas another accepts no parameters and does not return a data type (i.e., its return type is void). However, both of these methods can access any of the class members, both the public and the private ones. The caller, on the other hand, can access only the public class members, either before or after performing the method.

Let's return, for a moment, to the Java example. This call to the method `getErrorMsg` (by the CALLER) will always perform the method in `ErrorMsg` that accepts no parameter and returns a `String` result:

```
myString = myErrorMsg.getErrorMsg ();

(1)                public String getErrorMsg () {...}
```

On the other hand, this call to `getErrorMsg` will always perform the method that accepts one integer parameter and returns a `String` result:

```
myString = myErrorMsg.getErrorMsg (msgCode);

(2)                public String getErrorMsg (int msgCode) {...}
```

The Java compiler will automatically match up the particulars of the method invocation (in CALLER) with the particulars of the method (in `ErrorMsg`), and make sure the proper method is executed.

TERMS TO REVIEW

Let's review some of the object-oriented concepts just discussed and how you can understand them from a Java perspective.

Methods: Methods are the functions that a class can perform. Performing a method is similar to calling a function, or sending a message (to an object).

Method signatures: The method signatures of a class are the view it presents to external programs. A class can have many public methods. Each method can be further qualified by its method name, arguments, return type, and scope (e.g., public, protected, private). This combination of unique method name, argument profile, and return value is sometimes referred to as an interface in OO design terms. Note that the term *interface* is formally reserved for a specific construct in the Java language.

Method overloading: A class can define a method and then create more than one signature for this method. These unique interfaces are distinguished by their input parameters and return type.

EXERCISES: CLASSES, OBJECTS, AND METHODS

It's useful to use the keyboard and—as a good piano teacher might say—let your fingers teach your mind. You'll experiment with some of the sample classes presented so far.

Create an `ErrorMsg` class:

1. Using a text editor, create a Java source file named ErrorMsg.java in the java4cobol directory. The source file should contain these statements:

```
public class ErrorMsg {
// Define some public class instance variables.
          public  String  msgText = " ";
          public  int     msgSize;
// Define a public method.
          public void setErrorMsg (String inputMsg) {
// Modify one of the public variables. Set this variable to the text
// String that was passed as a parameter.
               msgText = inputMsg;
// Return from this method. Since this method has no return value (i.e.,
// it is declared as void), no return statement is necessary.
     }
// Define another public method.
          public String getErrorMsg () {
              String returnMsg;
// Set the local variable returnMsg to the data member msgText.
               returnMsg = msgText;
// Return from this method, and return this String variable.
               return (returnMsg);
          }

}
```

   Remember to save this source as a text file instead of a document file. The file name should be c:\java4cobol\ErrorMsg.java.

2. In the DOS command window, change the current directory to your java4cobol directory:

```
➔ cd c:\java4cobol
```

3. Compile your class, using the javac compiler:

```
➔ javac ErrorMsg.java
```

   Again, your class should compile successfully at this point. If it doesn't, review the suggestions mentioned in the introductory SDK exercises.

4. Using a text editor, modify and compile your HelloWorld.java source file in the java4cobol directory. The source file should contain these statements:

```
public class HelloWorld
{
     public static void main(String args[])
     {
          String tempMsg;
// Our original println statement:
          System.out.println("Hello World!");
// Create a new instance of the ErrorMsg class:
          ErrorMsg myErrorMsg = new ErrorMsg ();
// Get the value of the text item in ErrorMsg, by calling the getErrorMsg
// method.
          tempMsg = myErrorMsg.getErrorMsg ();
// Print the contents of the String returned by this method.
          System.out.println (tempMsg);
// Set the text item in ErrorMsg to some text String, and print its // //
// contents.
          myErrorMsg.setErrorMsg ("Some Text");
          tempMsg = myErrorMsg.getErrorMsg ();
          System.out.println (tempMsg);
// Call the setErrorMsg method again to set ErrorMsg to some other text.
          myErrorMsg.setErrorMsg ("Some New Text");
          tempMsg = myErrorMsg.getErrorMsg ();
          System.out.println (tempMsg);

     }

}
```

5. Now run your program, using the Java runtime that comes with the SDK:

```
➔ java HelloWorld
```

If you've done everything correctly, your MS-DOS window should look like this:

```
C:>javac ErrorMsg.java

C:>javac HelloWorld.java

C:>java HelloWorld
```

```
Hello World!

Some Text
Some New Text

C:\java4cobol>
```

6. Experiment with this program. Add the following lines to the end of your HelloWorld.java source file (i.e., immediately after the last `println` statement):

```
// Create a new instance of the ErrorMsg class.
          ErrorMsg myErrorMsg2 = new ErrorMsg ();
// Set the text item to some text String, and print its contents.
          myErrorMsg2.setErrorMsg ("Some Text for #2");
          tempMsg = myErrorMsg2.getErrorMsg ();
          System.out.println (tempMsg);
// Print the text item in the original object.
          tempMsg = myErrorMsg.getErrorMsg ();
          System.out.println (tempMsg);
```

7. Save the program again. Compile and execute it by performing these statements in the command window:

```
➔ javac HelloWorld.java
➔ java HelloWorld
```

Your DOS window should look like this:

```
C:>javac HelloWorld.java

C:>java HelloWorld
Hello World!

Some Text
Some New Text
Some Text for #2
Some New Text


C:\java4cobol>
```

The `ErrorMsg` class has two methods defined for it: `setErrorMsg` and `getErrorMsg`. You can define a variation on `getErrorMsg`. This alternative, or overloaded method, will convert the returned `String` to all uppercase letters.

1. Using a text editor, edit the Java source file ErrorMsg.java in the java4cobol directory.
2. Add the following lines to the end of your ErrorMsg.java source file (i.e., immediately before the last brace (}) in the class):

```
// Define a variation on the public method getErrorMsg.
          public String getErrorMsg (char caseFlag) {
              String returnMsg;
// Set the local variable returnMsg to the data member msgText.
               returnMsg = msgText;
// Convert to all uppercase, if requested.
               if (caseFlag == 'U')
                    returnMsg = returnMsg.toUpperCase ();
// Return from this method, and return the String variable.
               return (returnMsg);
          }
```

3. Compile the class again in the DOS command window:

```
➔ javac ErrorMsg.java
```

4. Modify HelloWorld to use this new method. Using a text editor, edit the Java source file HelloWorld.java in the java4cobol directory.
5. Add the following lines to the end of your HelloWorld.java source file (i.e., immediately after the last `println` statement):

```
// Call the new variation of the getErrorMsg method.
// This variation will return an all uppercase message.
          tempMsg = myErrorMsg.getErrorMsg ('U');
          System.out.println (tempMsg);
```

6. Compile and execute the program again in the DOS command window:

```
➔ javac HelloWorld.java
➔ java HelloWorld
```

Your output window should contain these lines:

```
C:>javac ErrorMsg.java

C:>javac HelloWorld.java

C:>java HelloWorld
Hello World!

Some Text
Some New Text
Some Text for #2
Some New Text
SOME NEW TEXT


C:\java4cobol>
```

You will now modify the applet version of your `HelloWorld` class so that it also uses the `ErrorMsg` class.

1. Using the text editor, edit the HelloWorld.java file in your java4cobol\ applet directory.
2. Add the required source text to your `HelloWorld` applet class so that it looks like this:

```
import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorld extends Applet
{

     public void paint(Graphics g)
     {
          String tempMsg;

          g.drawString("Hello applet World!", 5, 25);
// Create a new instance of the ErrorMsg class.
          ErrorMsg myErrorMsg = new ErrorMsg ();
// Print the contents of the public data member msgText in our class.
          tempMsg = myErrorMsg.getErrorMsg ();
          g.drawString (tempMsg, 5, 35);
```

```
// Set msgText to some text String, and print its contents.
            myErrorMsg.setErrorMsg ("Some Text");
            tempMsg = myErrorMsg.getErrorMsg ();
            g.drawString (tempMsg, 5, 45);
// Call the setErrorMsg method to set the text to some other text, and
// print its contents.
            myErrorMsg.setErrorMsg ("Some New Text");
            tempMsg = myErrorMsg.getErrorMsg ();
            g.drawString (tempMsg, 5, 55);

     }


}
```

3. Save the source file in text format. The file name should be c:\java4cobol\applet\HelloWorld.java.
4. Compile the program after changing to the applet directory in your DOS window.

```
➔ cd c:\java4cobol\Applet
➔ javac HelloWorld.java
```

What happens? Did you get an error message that looked like this?

```
C:>cd applet

C:>javac HelloWorld.java
HelloWorld.java:13: Class ErrorMsg not found.
           ErrorMsg myErrorMsg = new ErrorMsg ();
           ^
HelloWorld.java:13: Class ErrorMsg not found.
           ErrorMsg myErrorMsg = new ErrorMsg ();
                                     ^
2 errors

C:\java4cobol\applet>
```

What is the proper solution for this error? Should you create a new class called `ErrorMsg` as part of this project? Or should you use the `ErrorMsg` class that you had previously created?

If you chose the first solution, go to the rear of the class. If you chose the second, congratulations! You are well on your way to becoming an object-oriented programmer! The primary objective when you create a class is to have other programs reuse that class. This objective is not met when a class is copied, so make sure to always resist that temptation.

The SDK Java development environment can use a variety of mechanisms to find the proper class files. You will cover them all eventually, but the simplest one to use right now is the CLASSPATH argument. This argument lists directory names the Java compiler should search in order to find any required classes.

5. Tell the Java compiler where the `ErrorMsg` class is by adding the CLASSPATH argument to the java compile statement in your command window:

   ```
   ➔ javac -classpath c:\java4cobol HelloWorld.java
   ```

   It should compile successfully now.
6. Now run your new HelloWorld applet. You will need to set the CLASSPATH argument when you run it as follows:

   ```
   ➔ appletviewer -J-classpath -J.;c:/java4cobol HelloWorld.html
   ```

The syntax for the `classpath` *includes an initial period, followed by a semicolon. This tells the* `appletviewer` *to search in the current directory "." and then in the c:\java4cobol directory for all classes.*

   Experiment with this applet as well. You can make the changes in the text editor and then recompile the applet in the command window.
7. Add the following lines to the end of your HelloWorld.java source file (i.e., immediately after the last `g.drawString` statement).

   ```
   // Create a new instance of the ErrorMsg class.
           ErrorMsg myErrorMsg2 = new ErrorMsg ();
   // Set the text item to some text String, and print its contents.
           myErrorMsg2.setErrorMsg ("Some Text for #2");
           tempMsg = myErrorMsg2.getErrorMsg ();
           g.drawString (tempMsg, 5, 75);
   // Print the text item in the original object.
           tempMsg = myErrorMsg.getErrorMsg ();
           g.drawString (tempMsg, 5, 85);
   ```

8. Save the program again. Compile and execute it by performing these statements in the command window:

```
➔ javac -classpath c:\java4cobol HelloWorld.java
➔ appletviewer -J-classpath -J.;c:/java4cobol HelloWorld.html
```

Your applet window should look like this:

```
Hello applet World!

Some Text
Some New Text

Some Text for #2
Some New Text
```

The `appletviewer` *that comes with SDK does not print null as the way to represent an uninitialized* `String`*; instead, it generates an exception, or error condition. Therefore, the* `ErrorMsg` *class (as defined for the exercises) initializes its data member named* `msgText` *to a single space.*

9. Now, have the applet use the overloaded version of `getErrorMsg`. Add the following lines to the end of yourHelloWorld.java source file (i.e., immediately after the last `g.drawString` statement):

```
// Call the new variation of the getErrorMsg method.
// This variation will return an all uppercase message.
          tempMsg = myErrorMsg.getErrorMsg ('U');
          g.drawString (tempMsg, 5, 95);
```

10. Save the program again. Compile and execute it by performing these statements in the command window:

```
➔ javac -classpath c:\java4cobol HelloWorld.java
➔ appletviewer -J-classpath -J.;c:/java4cobol HelloWorld.html
```

Your applet window should look like this:

```
Hello applet World!

Some Text
Some New Text

Some Text for #2
Some New Text
SOME NEW TEXT
```

Reviewing the Samples

Let's review the class you've created (`ErrorMsg`) and the main program that uses it (HelloWorld). Try to relate the sample source statements to the result (for example, the output) each statement creates. If necessary, rerun the samples or look at the complete source code at the end of this section. Feel free to experiment by yourself.

- `ErrorMsg` is the first example of a reusable class. Your first program, HelloWorld, was a standalone *application*. `ErrorMsg`, on the other hand, is a class that HelloWorld can use. The original HelloWorld program from the first set of exercises was modified to use this class.
- HelloWorld first creates a new instance of `ErrorMsg` with this statement:

```
ErrorMsg myErrorMsg = new ErrorMsg ();
```

- To confirm that an instance has been created, HelloWorld prints out the contents of one of `ErrorMsg`'s public data members. The default value for a `String` data type is `null`. The `ErrorMsg` class as defined in the example in the first path of this chapter does not specify any initial data. Therefore, these statements

```
System.out.println ("HelloWorld!");
tempMsg = myErrorMsg.getErrorMsg ();
System.out.println (tempMsg);
```

would produce

```
Hello World!
null
```

- This `println` statement causes the text `null` to appear in the output window.
- On the other hand, the version of `ErrorMsg` in the exercises initialized the `msgText` data member to one space. Therefore, these statements

```
    System.out.println ("HelloWorld!");
    tempMsg = myErrorMsg.getErrorMsg ();
    System.out.println (tempMsg);
Hello World!
```

cause a single space character to appear in the output window (which is invisible).

- The HelloWorld application then stores some text, in this instance, of `ErrorMsg`. HelloWorld uses the `setErrorMsg` method to store this text.

```
myErrorMsg.setErrorMsg ("Some Text");
```

- HelloWorld next gets the text item from `ErrorMsg` and prints out the contents of this data member. Note that the output for this statement is the text "Some Text."

```
      tempMsg = myErrorMsg.getErrorMsg ();
      System.out.println (tempMsg);
Some Text
```

- Next, HelloWorld modifies the data member and prints out its new contents. In this case, the output for this print statement is the text "Some New Text."

```
      myErrorMsg.setErrorMsg ("Some New Text");
      tempMsg = myErrorMsg.getErrorMsg ();
      System.out.println (tempMsg);
Some New Text
```

- Finally, you modified HelloWorld to create a second instance of `ErrorMsg` and stored a reference to this new instance in the variable `myErrorMsg2`. You then stored the `String` "Some Text for #2" in its data member, using the `setErrorMsg` method:

```
        ErrorMsg myErrorMsg2 = new ErrorMsg ();
        myErrorMsg2.setErrorMsg ("Some Text for #2");
        tempMsg = myErrorMsg2.getErrorMsg ();
        System.out.println (tempMsg);
Some New Text for #2
```

- To show that two unique objects of the same type exist in HelloWorld, you printed out the data contained in both objects. The data associated with `myErrorMsg2` contained "Some Text for #2," and the data associated with `myErrorMsg` contained "Some New Text."

```
     tempMsg = myErrorMsg2.getErrorMsg ();
     System.out.println (tempMsg);
     tempMsg = myErrorMsg.getErrorMsg ();
     System.out.println (tempMsg);
Some New Text for #2
Some New Text
```

- The applet versions of HelloWorld performed much the same as did the application versions. HelloWorld, the applet, created an instance of `ErrorMsg`, stored some text in it, and then printed the text. Instead of printing to standard out, you used the `Graphics` class to print to a graphical window:

```
     System.out.println (tempMsg);          » becomes «
     g.drawString (tempMsg, 5, 35);
Hello applet World!
```

In the SDK applet exercise, this `drawString` statement causes no data to be displayed.

- You were able to simply use the existing version of `ErrorMsg` in the applets. That is, you did not need to create two versions of `ErrorMsg`, one for the application and one for the applet.
- You did need to instruct the compile environment where to look for the `ErrorMsg` class when it was not part of the current directory. You used the CLASSPATH argument (or project setting) to inform the compile environment of the directory that contained ErrorMsg.class.
- In the case of the SDK, you also needed to inform the execution environment (APPLETVIEWER.EXE) where it should look for the `ErrorMsg` class.

- As you did with the application HelloWorld, the applet was extended to contain two instances of `ErrorMsg`, each with its own data members:

```
ErrorMsg myErrorMsg2 = new ErrorMsg ();
myErrorMsg2.setErrorMsg ("Some Text for #2");
```

- To show that two unique data members with the same name exist in HelloWorld (one for each object of type `ErrorMsg`), you displayed both data members. The one associated with `myErrorMsg2` contained "Some Text for #2," and the one associated with `myErrorMsg` contained "Some Text."

```
tempMsg = myErrorMsg2.getErrorMsg ();
g.drawString (tempMsg, 5, 75);
tempMsg = myErrorMsg.getErrorMsg ();
g.drawString (tempMsg, 5, 85);
```

- Finally, you showed how `ErrorMsg` could define two variations on the same method. This is known as *method overloading*. The new variation accepted one character argument and is the argument that was set to the character value `'U'` that represents the case flag to tell the method to convert the return message to all uppercase.

```
// Define a variation on the public method getErrorMsg.
          public String getErrorMsg (char caseFlag) {
               String returnMsg;
// Set the local variable returnMsg to the data member msgText.
               returnMsg = msgText;
//  Define a variation on the public method getErrorMsg.
//  Perform the standard 'getErrorMsg' method.
          public String getErrorMsg (char caseFlag) {

//  Convert to all upper case, if requested.
               if (caseFlag == 'U')
                    return (getErrorMsg().toUpperCase ());
               else

//  Return from this method, without conversion.
                    return (getErrorMsg());
```

HelloWorld can use this new method to request a return message that is all uppercase.

```
          tempMsg = myErrorMsg.getErrorMsg ('U');
          System.out.println (tempMsg);
Hello applet World!

Some Text
Some New Text

Some Text for #2
Some New Text
SOME NEW TEXT
```

HelloWorld: The Application

```
public class HelloWorld {
     public static void main(String args[]) {
          String tempMsg;
// Our original println statement:
          System.out.println("Hello World!");
// Create a new instance of the ErrorMsg class.
     ErrorMsg myErrorMsg = new ErrorMsg ();
// Get the value of the text item in ErrorMsg by calling the
// getErrorMsg method.
          tempMsg = myErrorMsg.getErrorMsg ();
// Print the contents of the String returned by this method.
          System.out.println (tempMsg);
// Set the text item in ErrorMsg to some text String, and print its
// contents:
          myErrorMsg.setErrorMsg ("Some Text");
          tempMsg = myErrorMsg.getErrorMsg ();
          System.out.println (tempMsg);
// Call the setErrorMsg method again to setErrorMsg to some other text.
          myErrorMsg.setErrorMsg ("Some New Text");
          tempMsg = myErrorMsg.getErrorMsg ();
          System.out.println (tempMsg);
```

```
// Create a new instance of the ErrorMsg class.
          ErrorMsg myErrorMsg2 = new ErrorMsg ();
// Set the text item to some text String, and print its contents.
          myErrorMsg2.setErrorMsg ("Some Text for #2");
          tempMsg = myErrorMsg2.getErrorMsg ();
          System.out.println (tempMsg);
// Print the text item in the original object.
          tempMsg = myErrorMsg.getErrorMsg ();
          System.out.println (tempMsg);
// Call the new variation of the getErrorMsg method.
//  This variation will return an all uppercase message.
          tempMsg = myErrorMsg.getErrorMsg ('U');
          System.out.println (tempMsg);

     }
}
```

HelloWorld: The Applet

```
import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorld extends Applet {

     public void paint(Graphics g) {
          String tempMsg;
          g.drawString("Hello applet World!", 5, 25);
// Create a new instance of the ErrorMsg class.
          ErrorMsg myErrorMsg = new ErrorMsg ();
// Print the contents of the public data member msgText in the class.
          tempMsg = myErrorMsg.getErrorMsg ();
          g.drawString (tempMsg, 5, 35);
// Set msgText to some text String, and print its contents.
          myErrorMsg.setErrorMsg ("Some Text");
          tempMsg = myErrorMsg.getErrorMsg ();
          g.drawString (tempMsg, 5, 45);
```

```
// Call the setErrorMsg method to set the text to some other text, and
// print its contents.
          myErrorMsg.setErrorMsg ("Some New Text");
          tempMsg = myErrorMsg.getErrorMsg ();
          g.drawString (tempMsg, 5, 55);

// Create a new instance of the ErrorMsg class.
          ErrorMsg myErrorMsg2 = new ErrorMsg ();
// Set the text item to some text String, and print its contents.
          myErrorMsg2.setErrorMsg ("Some Text for #2");
          tempMsg = myErrorMsg2.getErrorMsg ();
          g.drawString (tempMsg, 5, 75);
// Print the text item in the original object.
          tempMsg = myErrorMsg.getErrorMsg ();
          g.drawString (tempMsg, 5, 85);
// Call the new variation of the getErrorMsg method.
// This variation will return an uppercase message.
          tempMsg = myErrorMsg.getErrorMsg ('U');
          g.drawString (tempMsg, 5, 95);

     }
}
```

ErrorMsg: The Class

```
public class ErrorMsg {
// Define some public class instance variables.
          public String msgText = " ";
          public int  msgSize;
// Define a public method.
          public void setErrorMsg (String inputMsg) {
// Modify one of the public variables. Set this variable to the text
// String that was passed as a parameter.
               msgText = inputMsg;
// Return from this method. Since this method has no return value
// (i.e., it is declared as void), no return statement is necessary.
          }
```

```
// Define another public method.
          public String getErrorMsg () {
              String returnMsg;
// Set the local variable returnMsg to the data member msgText.
               returnMsg = msgText;
// Return from this method, and return this String variable.
               return (returnMsg);
          }
// Define a variation on the public method getErrorMsg.
          public String getErrorMsg (char caseFlag) {
              String returnMsg;
// Set the local variable returnMsg to the data member msgText.
               returnMsg = msgText;
// Convert to all uppercase, if requested.
               if (caseFlag == 'U')
                    returnMsg = returnMsg.toUpperCase ();
// Return from this method, and return the String variable.
               return (returnMsg);
          }

}
```

4 Class Members

In This Chapter

- MYSUB COBOL
- MYSUB COBOL: ACTION-SWITCH
- Java Variables
- Classes, Objects, and Members Review
- Objects and COBOL
- Using Objects in Java
- Java Data Members
- Local Variables
- Primitive Data Types
- Arrays
- Method Members
- Constructors
- Exercises: Class Members
- Reviewing the Samples

I'll continue to extend the examples in order to explore other Java concepts.

The principle of data encapsulation *is similar to the difference between a subroutine's WORKING-STORAGE and its LINKAGE SECTION.*

COBOL specifies that items in a subroutine's WORKING-STORAGE can only be viewed and modified by the subroutine. Conversely, items in a subroutine's LINKAGE SECTION are constructed by the caller, then passed into the subroutine. Therefore, LINKAGE SECTION items are available to both the subroutine and the caller, and the items in a subroutine's WORKING-STORAGE are not.

Suppose you want MYSUB to count the number of times it has been called. In this case, you would define a variable in WORKING-STORAGE and increment it each time MYSUB is called.

Figure 4.1 shows how LINKAGE SECTION items are available to both caller and subroutine, whereas WORKING-STORAGE items in the subroutine are private to the subroutine.

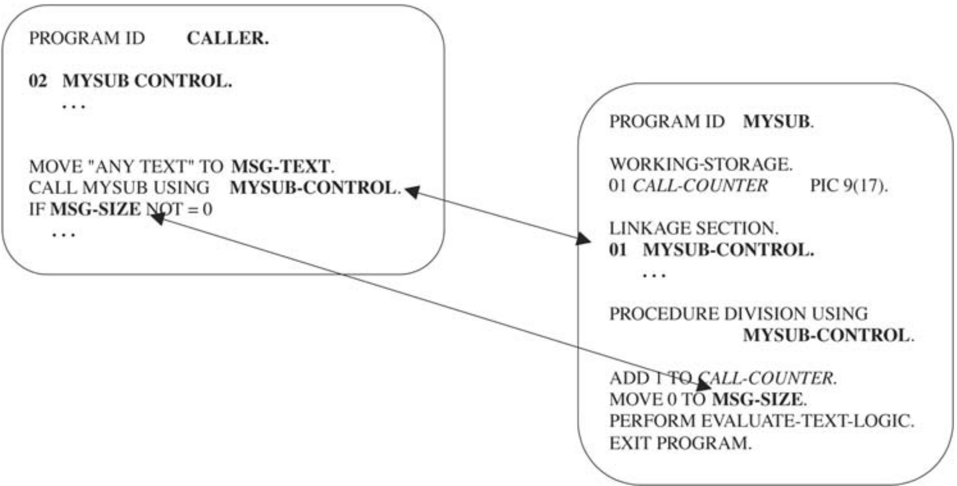


FIGURE 4.1
A subroutine has both LINKAGE AREA and WORKING-STORAGE.

In this way, the subroutine specification defines what is available to external programs and what is private to the subroutine. At the same time, the subroutine is free to define any internal items, based on its own requirements. The calling program is not aware of these details.

MYSUB COBOL

In this subroutine, CALL-COUNTER is a *private* variable and cannot be directly accessed by a caller. As a result, the number of times MYSUB has been called is not discovered by calling programs.

```
PROGRAM-ID. MYSUB.
DATA DIVISION.

WORKING-STORAGE SECTION.
01 CALL-COUNTER                    PIC S9(17) COMP-3 VALUE 0.

LINKAGE SECTION.
01 MYSUB-CONTROL.
```

```
    03   MYSUB-ACTION-SWITCH        PIC X.
         88 MYSUB-ACTION-EVALUATE            VALUE "E".
         88 MYSUB-ACTION-SET-AND-EVALUATE    VALUE "S".
    03   MSG-TEXT                             PIC X(20).
    03   MSG-SIZE                             PIC 9(8).

01 TEXT-STRING                    PIC X(20).

PROCEDURE DIVISION USING MYSUB-CONTROL, TEXT-STRING.

MYSUB-INITIAL SECTION.
MYSUB-INITIAL-S.
*   Increment the counter variable in WORKING-STORAGE SECTION.
     ADD 1 TO CALL-COUNTER.
*   Perform some function to detect the number of arguments.
     PERFORM GET-ARGUMENT-COUNT.
*   Determine whether TEXT-STRING or MSG-TEXT should be
*   evaluated. Store the correct item in LOCAL-TEXT.
     IF ARGUMENT-COUNT = 2
          MOVE TEXT-STRING TO LOCAL-TEXT
     ELSE
          MOVE MSG-TEXT TO LOCAL-TEXT.
*   Now, use LOCAL-TEXT in the subroutine's logic.
     IF LOCAL-TEXT = SPACES
          MOVE 0 TO MSG-SIZE
     ELSE
          MOVE 1 TO MSG-SIZE.

     IF MYSUB-ACTION-SET-AND-EVALUATE
          MOVE LOCAL-TEXT TO MSG-TEXT.

EXIT-PROGRAM.
     EXIT PROGRAM.

GET-ARGUMENT-COUNT SECTION.
GET-ARGUMENT-COUNT-S.
*   Set ARGUMENT-COUNT to the result.
       CALL "GET$NARGS" USING ARGUMENT-COUNT.
```

CALL-COUNTER is available only to MYSUB. Of course, you could define a new ACTION-SWITCH that would allow you to get the current value of CALL-COUNTER as follows in the next section.

MYSUB COBOL: ACTION-SWITCH

If this new ACTION-SWITCH is set, MYSUB returns the value from a private variable.

```
01 MYSUB-CONTROL.
    03   MYSUB-ACTION-SWITCH               PIC X.
         88 MYSUB-ACTION-EVALUATE         VALUE "E".
         88 MYSUB-ACTION-SET-AND-EVALUATE VALUE "S".
         88 MYSUB-ACTION-GET-CALL-COUNTER VALUE "G".
    03   MSG-TEXT                          PIC X(20).
    03   MSG-SIZE                          PIC 9(8).
    03   MYSUB-RETURNED-CALL-COUNTER       S9(17) COMP-3.

     ...

 MYSUB-INITIAL SECTION.
 MYSUB-INITIAL-S.

     ...
*   If requested, return the value in the counter variable.
     IF MYSUB-ACTION-GET-CALL-COUNTER
          MOVE CALL-COUNTER TO MYSUB-RETURNED-CALL-COUNTER.
     ...
```

This type of interface would allow a calling program to ask MYSUB to return the number of times it has been called. An approach such as this one allows the subroutine designer to decide which internal variables to make available and how a calling program should access them. In support of this design, CALL-COUNTER is managed as a private variable and MYSUB-RETURNED-CALL-COUNTER as a *public* variable.

JAVA VARIABLES

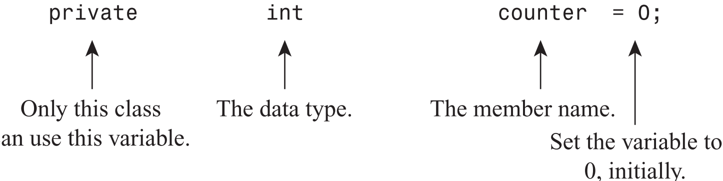
In a similar way, Java variables can be private or public. Public variables can be directly viewed or modified by other classes, whereas private variables cannot. Java also supports *package* variables, which are somewhere in between (that is, they are available to classes in a defined group of classes called a package). Variables that are not declared private or public will default to package variables. Finally, Java also defines another type, *protected*, which has to do with a feature called inheritance, a topic I will discuss later.

ErrorMsg Class

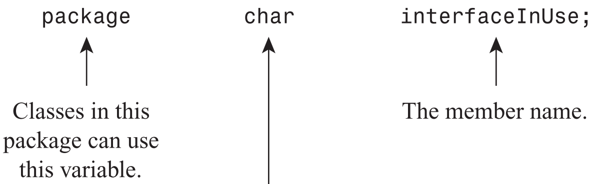
This portion of the `ErrorMsg` class increments the private variable named counter.

```
public class ErrorMsg {

     public String msgText;
     public int    msgSize;
     private int   counter = 0;
     package char  interfaceInUse;

     public void setErrorMsg (String inputMsg) {
        counter = counter + 1;
        interfaceInUse = 'S';
        ...
// Some logic
        ...
        }
  }
```

You can examine the new member definitions and how they are used:

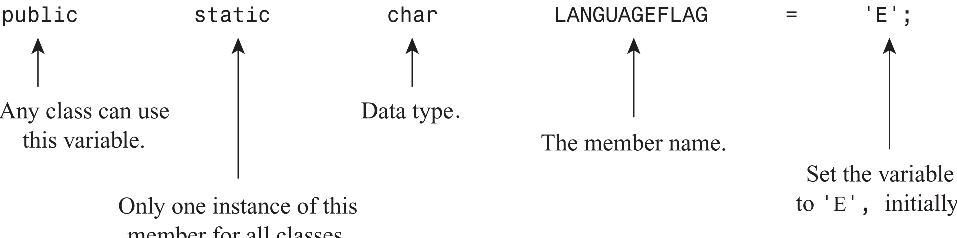


This example also introduces the concept of data variable initialization. Similar to COBOL's syntax, Java variables can be assigned an initial value. Unless a specific value is given, variables are initialized with their default natural values (for example, NULL for object types, and 0 for numeric types). The developer can assign other initial values to variables using the syntax `= value` or by using the `new` directive for objects.



Finally, data members can also be assigned the *static* access control. This control specifies that only one instance of the data member exists, and it is available to all instances of this class in the current run unit. In Java, you call this type of variable a *class* variable. In this sense, these types of data items are like data items in a COBOL subroutine's WORKING-STORAGE. That is, only one instance of this variable exists, and every class of this type can access it.

Static class members can be either public, private, or protected. If a static data item is private, then only classes of this type can access it (like WORKING-STORAGE items in COBOL). If a static data item is public, then every other class can access it as well (like the EXTERNAL access control in COBOL).



Just like data members, Java methods can also be declared as public, private, package, or static. For example, these statements

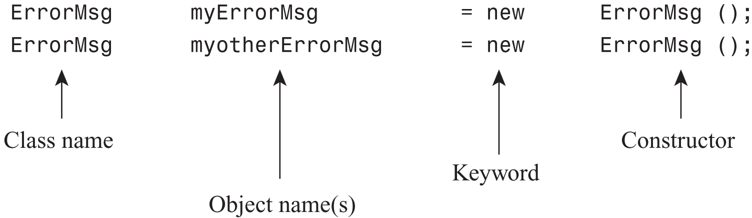
```
public    void setErrorMsg (String inputMsg) () {
private   void countErrorMsg () {
          void checkErrorMsg (String inputMsg) {
public static  void SETLANGUAGE (char languageFlag) {
```

define four methods: one `public`, one `private`, one `package` method, and one that is a `static` method. The default access control is `package`, which is why `checkErrorMsg` does not have a qualifier.

Classes, Objects, and Members Review

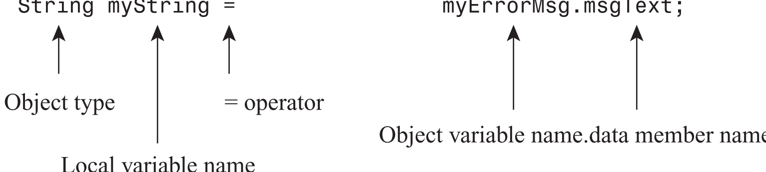
This is a good time to review some important concepts:

- A Java *class* is the blueprint, or specification, for all instances of its type.
- A Java *object* is an instance of a class.
- A calling program (a class itself) first creates a `new` object prior to using it.
- In the process, the calling program stores a *reference* to the object in an *object instance variable* (also known as a reference variable). This reference variable is a *handle*, or a pointer, to the object. It contains identifying information about the object, including information such as the object's storage location (in memory).
- The calling program can create many instances of a class and manages each instance by managing the reference variable.
- *Data members* (variables) and *methods* that are defined by this class are automatically members of this object. They are associated with this instance of the object and are accessed using the object instance variable as a prefix.
- A program can use these object instance (reference) variables to manage more than one instance of a class. For example:

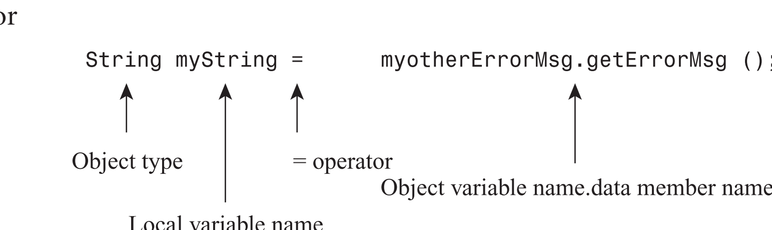


In this example, `myErrorMsg` is one object instance variable, and `myotherErrorMsg` is another. They point to different objects of type `ErrorMsg`.

The object instance variable is used to identify the object that contains the data members or methods:



or



If you understand these terms and followed the descriptions of these concepts, you understand the most important OO concepts (classes, objects, and members). These concepts are the core principles (or building blocks) of any OO language, including Java.

Objects and COBOL

In the COBOL examples, you simulated the concept of object instance variables by creating unique instances of MYSUB-CONTROL:

```
01 MYSUB1-CONTROL.
     03   MYSUB1-ACTION-SWITCH            PIC X.
     03   MSG-SIZE                        PIC 9(8).
          ...

01 MYSUB2-CONTROL.
     03   MYSUB2-ACTION-SWITCH            PIC X.
     03   MSG-SIZE                        PIC 9(8).
          ...
```

The program that contained both of these definitions and also called MYSUB with each one would now have two unique instances of the variables in MYSUBx-CONTROL, reflecting the result of MYSUB's logic:

```
IF MSG-SIZE OF MYSUB1-CONTROL = 0
     ...
ELSE IF MSG-SIZE OF MYSUB2-CONTROL = 0
     ...
END-IF
```

Of course, any other calling program could use the name MYSUB-CONTROL or MYSUB1-CONTROL. These would refer to a different instance of this CONTROL.

This COBOL coding style represents concepts very similar to the Java concepts of classes, objects, and members. That is, a calling program first defines a unique instance of a class, creating an object that can be referenced by name (MYSUB2-CONTROL). This object variable name is used as a prefix to access the member variables for this particular object.

Although very similar, Java's implementation of these concepts is slightly different. One difference in particular probably causes the most trouble for COBOL programmers. That is, all data members (or properties) of a class, whether private or public, are associated with a *class instance* (or *object*). Properties are associated with a particular instance of a class, not with all instances of that class.

To better understand this from a COBOL perspective, imagine that each time you call a subroutine with a new MYSUB-CONTROL, a unique instance of that subroutine is created for you. Further, imagine that the system automatically calls an initialization section in the subroutine (in Java terms, its *constructor*), so that this instance of the subroutine can perform any initializations required. And finally, suppose that every time a new MYSUB-CONTROL is used as a parameter to the MYSUB subroutine, the system would create your own private copy of the subroutine. Therefore, it would appear as if there were separate copies of all items in the subroutine's WORKING-STORAGE (items like CALL-COUNTER), one for each instance of MYSUBx-CONTROL. Actually, it is possible to emulate this behavior quite easily in COBOL.

If your subroutine contains only temporary WORKING-STORAGE items, and uses only items in the LINKAGE SECTION, then it behaves very much like an object. Instead of defining items in WORKING-STORAGE, suppose the subroutine defines a slightly different representation of the items in the LINKAGE SECTION, which includes items known only to the subroutine. In this case, these items can be considered private, compared to the public items known to the caller. Add an INITIALIZE action switch, and you pretty much have a COBOL object. In fact, this is more or less what every OO compiler does internally, regardless of language.

Let's complete the MYSUB example to present this concept. Up to now, you defined only public items in the LINKAGE SECTION and private items in MYSUB's WORKING-STORAGE SECTION. The items in WORKING-STORAGE will be shared across all instances of MYSUBx-CONTROL. Remember how you used this principle to create the COUNTER variable, which counts the number of times MYSUB is called?

You could create an instance of this COUNTER, one that is specific to each MYSUBx-CONTROL. You do this simply by defining a COUNTER item in MYSUBx-CONTROL. If only MYSUB (the subroutine) is aware of this item, then it could be considered a private variable. Finally, you need to define an INITIALIZE action switch so that you can properly initialize this variable.

```
01 MYSUB-CONTROL.
     03  MYSUB-ACTION-SWITCH                PIC X.
         88 MYSUB-ACTION-INITIALIZE        VALUE "I".
         88 MYSUB-ACTION-EVALUATE          VALUE "E".
         88 MYSUB-ACTION-SET-AND-EVALUATE  VALUE "S".
         88 MYSUB-ACTION-GET-CALL-COUNTER  VALUE "G".
     03  MSG-TEXT                      PIC X(20).
     03  MSG-SIZE                      PIC 9(8).
     03  MYSUB-RETURNED-CALL-COUNTER        PIC 9(10).
```

The next item contains the private items in MYSUB-CONTROL:

```
     03  MYSUB-PRIVATE-ITEMS                PIC X(20).
```

The MYSUB subroutine is expanded as follows:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MYSUB.
************************************************************************
* This routine accepts a text item as a parameter and                  *
* evaluates the text. If the text is all spaces,                       *
* MSG-SIZE will be set to 0.                                           *
* If requested, the text item will also be stored in the               *
* passed control structure.                                            *
* If the text item is not passed, then MSG-TEXT                        *
* will be evaluated instead.                                           *
* MYSUB will count the number of times it has been                     *
* with a particular MYSUBx-CONTROL and the number of                   *
* times it has been called using all CONTROLs.                         *
* MYSUB must be called with the INITIALIZE action when                 *
* any new CONTROL is to be used.                                       *
************************************************************************
DATA DIVISION.

WORKING-STORAGE SECTION.
01 CALL-COUNTER                    PIC 9(10) VALUE 0.
01 ARGUMENT-COUNT                  PIC 9.
01 LOCAL-TEXT                      PIC X(20).

LINKAGE SECTION.
*   Below you have a view of MYSUB-CONTROL that is used by the
*   MYSUB subroutine:
01 MYSUB-CONTROL.
```

```
     03  MYSUB-ACTION-SWITCH               PIC X.
         88 MYSUB-ACTION-INITIALIZE       VALUE "I".
         88 MYSUB-ACTION-EVALUATE         VALUE "E".
         88 MYSUB-ACTION-SET-AND-EVALUATE VALUE "S".
         88 MYSUB-ACTION-GET-CALL-COUNTER VALUE "G".
     03  MSG-TEXT                          PIC X(20).
     03  MSG-SIZE                          PIC 9(8).
     03  MYSUB-RETURNED-CALL-COUNTER       PIC 9(10).
     03  MYSUB-PRIVATE-ITEMS               PIC X(20).
*   In the subroutine's definition of MYSUB-CONTROL,
*   PRIVATE-ITEMS is redefined with items known only to the subroutine.
     03   FILLER REDEFINES MYSUB-PRIVATE-ITEMS.
          05   MYSUB-PRIVATE-COUNTER        PIC 9(8).
          05   MYSUB-OTHER-PRIVATE-ITEMS    PIC X(12).

01 TEXT-STRING                              PIC X(20).

PROCEDURE DIVISION USING MYSUB-CONTROL, TEXT-STRING.

MYSUB-INITIAL SECTION.
MYSUB-INITIAL-S.
*   The program then performs some function to detect the number of
*   arguments,
     PERFORM GET-ARGUMENT-COUNT.
     IF ARGUMENT-COUNT = 2
          MOVE TEXT-STRING TO LOCAL-TEXT
     ELSE
          MOVE MSG-TEXT TO LOCAL-TEXT.
*   Increments the Global counter,
     ADD 1 TO CALL-COUNTER.
*   processes the ACTION-SWITCHES,
     IF MYSUB-ACTION-INITIALIZE
*   and initializes the instance counter.
          MOVE 0 TO MYSUB-PRIVATE-COUNTER
*   If requested, the program returns the value in the counter
*   variable.
     ELSE IF MYSUB-ACTION-GET-CALL-COUNTER
          MOVE CALL-COUNTER TO MYSUB-RETURNED-CALL-COUNTER
     ELSE IF MYSUB-ACTION-EVALUATE
*   This is a request to evaluate the text item.
          PERFORM EVALUATE-TEXT-ITEM.
*   The program increments the instance counter.
     ADD 1 TO MYSUB-PRIVATE-COUNTER.
```

```
EXIT-PROGRAM.
     EXIT PROGRAM.

EVALUATE-TEXT-ITEM SECTION.
EVALUATE-TEXT-ITEM-S.
     IF LOCAL-TEXT = SPACES
          MOVE 0 TO MSG-SIZE
     ELSE
          MOVE 1 TO MSG-SIZE.
     MOVE LOCAL-TEXT TO MSG-TEXT.

GET-ARGUMENT-COUNT SECTION.
GET-ARGUMENT-COUNT-S.
*   The program sets ARGUMENT-COUNT to the result.
      CALL "C$NARGS" USING ARGUMENT-COUNT.
```

You have just defined a COBOL subroutine that behaves very much like an object! Two characteristics of the way you've used COBOL make this possible. Some data items in MYSUB-CONTROL are public (i.e., known to the caller), some are private (known only to the subroutine), but *all* data items (in the LINKAGE SECTION) are unique to a given MYSUB-CONTROL. No data associated with a particular MYSUB-CONTROL is stored in the WORKING-STORAGE of the subroutine.

USING OBJECTS IN JAVA

Java naturally associates both data member variables and methods with a particular instance of a class unless the members and methods are defined as static. This means that you should view members of a class as being associated with an instance variable (e.g., `myErrorMsg.msgText`), even though it is defined by the class.

It is this treatment of data members of the class as properties of a particular class, along with the capability of class members to be declared as public, private, or protected, that implements the OO principle of *encapsulation.* A class is said to encapsulate some logic or behavior and to publish only those members that are appropriate for other classes to use. Class members are managed as if they were simply attributes of an instance of the class (i.e., an object). In many respects, a class is just another data type, one with user-defined attributes, and it includes whatever code is necessary to support those attributes. By extension, then, an object is a variable of type class.

This approach promotes the principle of *reuse*, that is, the practice of writing a class (a piece of code) that will be used by other classes. Consumer classes only have the view(s) of the class that the class designer thinks appropriate. Consumer classes then use these classes as if they were simply another type of variable. As I have demonstrated with the COBOL programs, it is certainly possible to write programs that do this in any programming language; but with Java, it is almost impossible to code in any other way.

Up to this point, I've introduced some OO concepts and described how Java supports these concepts. So it's a good time to present a more complete definition of these Java terms and syntax.

JAVA DATA MEMBERS

Data members are variables in the normal understanding of variables in computer languages (for example, data ITEMS in COBOL). They are members of a class in the sense that they are attributes of that class and contain its current state information.

Data members can be qualified with access controls, which define how visible the data members are to other classes. Data members can be either public, private, or package variables. Public members are visible outside the class, and they can be set or evaluated by other classes. Private members are only visible inside the class; other classes cannot view or modify these variables. Package members are visible to all classes in a package (you can think about a package as a sort of directory of related class files). Package is the default access condition.

```
public int    msgSize;
private int   counter = 0;
        char  interfaceInUse;
```

Java also defines another data member access control, *protected*, which is used to support inheritance. I will discuss protected access later, along with inheritance.

All data members are normally associated with an instance of a class (i.e., an object). As I discussed earlier, each new instance of a class contains unique copies of all of its variables, both public and private.

With any rule, there is an exception. (Don't you hate it when you think you finally understand something, and then the instructor throws a curve ball?) *Class variables* are data members that are associated with all instances of a class. These variables persist for the duration of the program and are shared by all instances of this class. In many respects, class variables are similar to items defined in a COBOL

subroutine's WORKING-STORAGE, except that they can be visible to other subroutines. You can think of public class variables as analogous to COBOL EXTERNAL items in a subroutine.

Class member variables can also be qualified with the keyword `Final`, which means that this data member variable cannot be modified by any class. This control is commonly used to declare constants, that is, variables that are only used to set or evaluate other variables. These are very similar to how Level 77 items are used in COBOL.

ErrorMsg Class

Here are some examples of data members in the `ErrorMsg` class that have different access controls:

```
     public class ErrorMsg {
// Public instance variables:
          public String msgText;
          public int  msgSize;
// A private instance variable:
          private int  counter = 0;
// A package instance variable:
            char     interfaceInUse;
// A public static variable:
          public static int     total_counter;
// A final (i.e., read only) variable
          public final     static int NO_TEXT_FOUND = 0;

          public void setErrorMsg (String inputMsg) {
               interfaceInUse = 'S';
               msgText = inputMsg;
               msgSize = msgText.length ();
               counter = counter + 1;
               total_counter = total_counter + 1;
               ...
               ...
          }
     }
```

Caller Class

Some other class would use the `ErrorMsg` class and its variables in this way:

```
// Create an instance of ErrorMsg.
     ErrorMsg      myErrorMsg = new      ErrorMsg ();
// Create an instance of a String variable and put some text in it.
     String inputMsg = "Some input Text";
// Call the function setErrorMsg.
     myErrorMsg.setErrorMsg (inputMsg);
// Evaluate the size of this instance of ErrorMsg using the constant
// NO_TEXT_FOUND.
     if (myErrorMsg.msgSize == myErrorMsg.NO_TEXT_FOUND) {
           ...
     // Some logic
           ...
     }
// Evaluate the number of times ErrorMsg has been called.
     if (myErrorMsg.total_counter == 100) {
           ...
     // Some logic
           ...
     }
// A static class variable can also be identified using only the class
// name.
//  Static variables are not associated with any instance of a class.
     if (ErrorMsg.total_counter == 100) {
           ...
     // Some logic
           ...
     }
```

Local Variables

Java supports a concept of local variables, an idea without a traditional COBOL equivalent. A Java program can define a variable when it is needed, rather than right up front as part of the class definition.

Imagine that COBOL allowed you to insert variable definitions (e.g., 01 MY-ITEM PIC X) right in the middle of the PROCEDURE DIVISION. Furthermore, suppose that variables defined this way were only associated with the paragraph in which they were defined. When the paragraph completed, the variable would go away. Finally, suppose that another MY-ITEM variable could be defined in the same program but in a different paragraph. This instance of MY-ITEM would really be a separate variable, having nothing to do with the original MY-ITEM. You might wonder what kind of confused code that could lead to!

Java allows this sort of "as needed" temporary variable definition. You can just define variables as they are needed. There is no need to define them all up front, as in COBOL. The language does attempt to tighten up some of the more egregious rules defined by C and C++. (Many C and C++ programmers have struggled to debug a program, only to discover that some instance of a program variable *x* was really a different instance of *x*, even though both instances have the same name.)

Temporary local variables do have some benefits, mostly in the area of efficient memory management. A block of code that temporarily needs a variable does not need to define it as an instance variable for the whole class. Instead, the programmer can define it when necessary, as if it were an executable statement. Then the system will delete the variable when the block of code is completed.

Java restricts the scope of the variable to the block of code in which it is defined. Therefore, a program can define counter variables (x appears to be some sort of a standard) as needed, confident that the compiler will detect conflicts in nested blocks of code. Variables that must be shared by multiple blocks of code must be defined as parameters or as class instance variables.

```
if (myErrorMsg.total_counter == 100) {

// Some logic
     ...
// Define a local variable.
     int x;
     ...
     }
// It will be deleted after the }.
```

Finally, values passed as parameters to Java methods are passed *by value.* This means that a copy of the primitive data types (discussed in the following sections) are passed rather than the variable itself. For object reference variables, the reference variable is passed, but it is in fact a reference to the object. As a result, any passed primitive data type values are implicitly local variables. Any changes you make to the objects referenced in passed object reference variables will be reflected in the objects themselves. I'll discuss variable scope in Chapter 7, when I discuss Java's flow control construct.

PRIMITIVE DATA TYPES

As with any language, Java defines some native variable types that can be used to store and represent data. Many languages (especially C) define standard data types, but the implementation details for some types can vary slightly across systems,

creating code and data portability problems. Java attempts to deal with these problems by explicitly prescribing these data types and how they are to be defined and implemented across systems.

Type	Description	No. of Bits	Range Equivalent	COBOL
boolean	True or false	1	N/A	TRUE, FALSE
char	Unicode character	16	x'0000' to 'xFFFF'	N/A
byte	Signed integer	8	–128 to 127	PIC X[1]
short	Signed integer	16	–32768 to 32767	PIC S9(4) BINARY[2]
int	Signed integer	32	–2147483648 to 2147483647	PIC S9(9) BINARY
long	Signed integer	64	–9223372036854775808 to 9223372036854775807	PIC S9(18) COMP–4[3] PIC S9(18) COMP–5
float	IEEE754 number	32	+/–3.40282347E+38 to +/–1.40239846E–45	USAGE IS FLOAT[4]
double	IEEE754 number	64	+/–1.79769313486231570E+308 – +/–4.94065645841246544E–324	USAGE IS DOUBLE[4]

1. *Most COBOL compilers handle single-byte characters as unsigned, that is, with values between 0 and +255.*
2. *Some COBOL compilers define BINARY as COMP–4 or COMP–5. Only values with four digits (–9999 through 9999) are guaranteed to fit in an integer of this size.*
3. *The range of valid values may be limited on some compilers to those that fit in 18 digits.*
4. *Supported by some compilers.*

These next types are not intrinsic Java types, but instead are some of the standard classes that are commonly used in Java programs.

Type	Description	No. of Bits	Range Equivalent	COBOL
String	Sequence of characters	16 (each)	Any set of Unicode characters	PIC X(nnn)[1]
Array	Group of variables	N/A	Any object or data type variable(s)	OCCURS
BigDecimal	Fixed precision number	Variable	Unlimited	PIC S9(n) COMP-3[2]
BigInteger	Fixed precision integer	Variable	Unlimited	PIC S9(n) COMP-3[2]
Object	Object reference variable		An object reference variable for any class type	01 CTL-AREA

1. *COBOL strings are always fixed-length 8-bit characters and bank padded to the defined size of the item. Java strings are variable-length Unicode characters and have an embedded size attribute. I will discuss strings later.*
2. *Sometimes just COMP, COBOL's packed decimal type.*

Values can easily be converted between the various data types, but in many cases, you have to be explicit. A *type cast* is used to indicate to the compiler that you intend to convert from one type to another; otherwise, the compiler might think you are making a mistake!

```
           int     counter = 23;
// 'F' indicates a float constant
          float   bigNumber = 1.23F;     // or 1.23f;
          double  biggerNumber = 123.45;
// Direct assignment from integer to floating point data types is possible.
          bigNumber = counter;
// Direct assignment from floating point to integer data types is not
// possible.
          counter = bigNumber;
// this is a compile error
// Move the floating point number to the integer, using a cast. Of course,
// some precision may be lost, but you've told the compiler you know what
// you're doing.
          counter = (int) bigNumber;
```

```
// Direct assignment from floating point to double is fine, but you need a
// cast to go from double to float:
        biggerNumber = bigNumber;
// OK
        bigNumber = biggerNumber;
// an error
        bigNumber = (float) biggerNumber;
// OK
```

Arrays

Arrays in Java are very similar to arrays in COBOL. Arrays are objects that contain a set of other objects or a set of data elements. All the elements in an array must be of the same type. In addition, the size of an array must be defined when it is created and cannot be dynamically adjusted in size. These are all specifications that should be familiar to the COBOL programmer.

```
      01 ERROR-MESSAGE-ITEMS.
           03 ERROR-MESSAGE OCCURS 10 TIMES  PIC X(80).
// In Java this would be:
      ErrorMsg myErrorMsgs[] = new ErrorMsg[10];
// or
      int errorNumbers[] = new int [10];
```

Note that in the first example, the `myErrorMsgs` array was created, but the `ErrorMsg` objects that it will eventually contain were not created. Instead, the object reference variables inside the array are initialized to `null`. In contrast, an array of data items that contains numeric data items are initialized to 0 by default.

The array brackets can be placed by the type name instead of the variable name. This actually is the more common syntax:

```
int[] errorNumbers = new int [10];
```

Interestingly, it is possible to define an array with no more information than the types of elements it will contain:

```
ErrorMsg[] myErrorMsgs;
```

The array variable `myErrorMsgs` can be subsequently assigned to some other `ErrorMsg` array. This feature can be very useful in managing arguments and return values for methods, although you should try to use the newer *collection* classes for this purpose, which I will discuss in a moment.

Arrays can be created with initial values assigned to its elements using a syntax similar to C's syntax:

```
int errorNumbers[] = {1, 2, 10, 22, 23};
```

As you might suspect, this array will contain five integer values.

Unlike COBOL, in which an array's size is defined only at compile time, the size of a Java array can be defined at runtime:

```
int array_size = 10;
ErrorMsg[] myErrorMsgs = new ErrorMsg[array_size];
```

As in COBOL, an array in Java can be multidimensional:

```
int[][] errorNumbers = new int[10][3];
```

Based on the preceding declaration, `errorNumbers` is an array of 10 integer arrays (each of which is three integers in size).

Every array contains a *length* member, which is the number of elements in the array.

This loop will execute five times because there are five elements in `errorNumbers` and none is equal to 30. I will discuss `for` loops in more detail in Chapter 7, but for now, read the following `for` statement as "Initialize an `int` variable named `x` to 0, perform the next statement until `x < errorNumbersIO.length`, and increment `x` by one after each iteration."

```
int[] errorNumbersIO = {1, 2, 10, 22, 23};
for (int x = 0; x < errorNumbersIO.length; x++) {
     if (errorNumbersIO[x] == 30) {
          break;
     }
}
```

Java checks all array references to prevent out-of-bounds references. Any out-of-bounds reference will generate an `ArrayIndexOutOfBoundsException` exception. (Whew! And you thought COBOL was a bit wordy.)

Although I did introduce arrays as a type of object, Java does not treat arrays simply as standard objects. There is enough special syntax in the language for arrays to view them as a special type of reference variable, slightly different from objects.

For example, the assignment and evaluation statements for arrays that contain primitive data types are very similar to the same statements used by the primitive data type. This syntax is valid for arrays: `if (errorNumbersIO[x] == 30)`. It is quite similar to the syntax used by primitive integer types: `if (errorNumber == 30)`. If arrays were simply objects, this syntax would compare the objects in the arrays, not the numeric values.

Java also provides a set of useful array-specific methods. In order to copy some part of an array into another array, use the `arraycopy()` function in the Java `System` class:

```
System.arraycopy (sourceArray, int sourcePosition,
      destinationArray, int destinationPosition, int
      numberOfEntriesToCopy);
```

Suppose you want to copy the I/O error numbers array and the logical error numbers array into a single array. You could use the `arraycopy()` method to do this:

```
     int[] errorNumbersIO = {1, 2, 10, 22, 23};
     int[] errorNumbersLogical = {101, 102, 108, 122};
     int[] errorNumbersAll = new int[errorNumbersIO.length
                              + errorNumbersLogical.length];
     System.arraycopy (errorNumbersIO, 0,
                             errorNumbersAll, 0, 5);
     System.arraycopy (errorNumbersLogical, 0,
                             errorNumbersAll, 5, 4);
// A temporary array can be defined, and then it can be assigned to any of
// these arrays.
     int[] tempNumbers;
     tempNumbers = errorNumbersIO;
     tempNumbers = errorNumbersLogical;
     tempNumbers = errorNumbersAll;
```

Arrays can be passed into methods and returned from a method as its return value. This is very useful when a method needs to return a set of values instead of just one.

```
     public class ErrorCodes {
// Define a static structure that contains all the I/O error codes.
         static int[] IOCodes = {1, 2, 3, 10, 12, 22, 23, 30};
// A method that returns all the error codes for I/O functions:
         public int[] errorCodesIO {
// Create an array to hold the error codes.
             int[] results = new int[IOCodes.length];
// Copy the array (actually the method System.arraycopy() would be a
// better choice than this loop).
             for(int x = 0; x < IOCodes.length; x++) {
                results[x] = IOCodes[x];
             }
// Return the array that you have just loaded.
             return results;
         }
```

Before I move on to the next topic, let's visit this statement again.

```
ErrorMsg[] myErrorMsgs;
```

What does `myErrorMsgs[0]` contain after this statement?

```
ErrorMsg[] myErrorMsgs;
 myErrorMsgs = getErrorMsgs();
if (someCondition)
     myErrorMsgs = getExtendedErrorMsgs();
```

Since the array has not been initialized via the `new` operator, it does not contain any `ErrorMsgs`. However, I did say it *will* contain reference variables of this object type.

After this statement,

```
myErrorMsgs = new ErrorMsg[10];
```

all the elements in `myErrorMsgs` will contain empty object reference variables, or `null`. Therefore, this statement:

```
if (myErrorMsgs[0] == null)
```

will always evaluate to `true` until `myErrorMsgs[0]` is assigned to an actual reference variable.

```
        myErrorMsgs[0] = new ErrorMsg ();
        if (myErrorMsgs[0] == null) {
    // This section will not be executed, since myErrorMsgs[0] has been
    // assigned to an object.
        }
```

Arrays as Parameters

Parameters passed to Java methods are passed by value, not by reference. In COBOL, parameters are passed by reference to a subroutine, although some compilers support an optional "by value" mechanism.

When arrays are passed to methods as parameters, the method cannot change the array. However, if the array contains objects, then the called method can change the objects contained in the array. Though not exactly the same as a parameter that is passed by reference, this can be a useful mechanism to construct a method that modifies the objects passed to it as an argument.

A way to look at it is that a copy of the reference is made and that copy is passed into the method. They both point to the same thing, but only the copy can be changed to point to something different.

Method Members

Method members are the functions that a class provides. This concept is similar to the COBOL subroutine that provided multiple functions based on an ACTION-SWITCH. Method members are identified by their names (e.g., `setErrorMsg`) and their method signatures (that is, the types and number of parameters). Method member references are distinguished from data member references by an argument definition, which is enclosed in an open parenthesis or closed parentheses for no arguments.

Remember how class data members could be made visible or invisible to other classes? Class method members can also be qualified with access controls. These controls define whether other classes can access methods directly. Class method members can be public, private, or package methods.

Some of Java's method members are not associated with any instance of a class but rather with all instances of a class. These are called *class methods*. Since class methods do not belong to an instance of a class, class methods can access only static variables, not instance variables. Such methods are helpful in managing static variables (for example, resetting a static variable based on some condition).

ErrorMsg Class: Static Variable

The `ErrorMsg` class is enhanced to define a static variable, and a default static method to initialize that variable.

```
      public class ErrorMsg {

           public  String msgText;
           public  int    msgSize;
           private int    counter = 0;
                   char   interfaceInUse;
           public  static int  total_counter;
           public  final   static int   NO_TEXT_FOUND = 0;
   // A static initializer method with no name and no parameters
           static {
                total_counter = 0;
           }
   // A public method with no parameters:
           public void setErrorMsg () {
           }
   // A public method with the same name and one parameter of type String.
   //  This is actually a different method, or interface definition.
           public void setErrorMsg (String inputMsg) {
                interfaceInUse = 'S';
                msgText = inputMsg;
                msgSize = msgText.length ();
   // Call the manageCounters method.
                manageCounters ();
           }
   // A package method named manageCounters.
   //  This method is only visible to other classes in this package.
           void manageCounters () {
                counter = counter + 1;
                total_counter = total_counter + 1;
           }

      }
```

ErrorMsg Class: Static Initializer

Classes can also define specialized *static initializer* code. Like a class variable, this block of code will be executed when the first instance of the class is initiated by a runtime. Often this code will perform special initialization logic for the static class

variables. In fact, only static class variables can be accessed by the initializer code. There are no instance member variables available to this code, since class initializer code is performed only once and before any instances of the class have been created.

```
     public class ErrorMsg {
// Public instance variables:
          public String msgText;
          public int    msgSize;
// A public class variable
          public static int     TOTAL_COUNTER;
// A class initializer code block
          static void   TOTAL_COUNTER (){
               TOTAL_COUNTER = 1;
          }
// A regular instance method
          public  void  setErrorMsg  (String inputMsg) {
               ...
               ...
          }
     }
```

Constructors

Now is a good time to talk about constructors. When an object is first instantiated (with the `new` operator), the JVM will create the new instance of the class and call its *constructor*. This is a special built-in definition that performs any initialization logic that may be required by that class. Constructors are not methods, but they are similar to methods in some respects. This special, perform-one-time-only definition exists in every object, even if the class designer didn't explicitly define it.

As with real methods, a class can have more than one constructor; each one is identified by a unique parameter signature. By default, every class will have at least one constructor, a constructor that accepts no parameters. This default constructor will be generated by the compiler if the class has no constructors defined for it. Sometimes the default constructor is called the "no arguments" constructor.

These constructors are useful for any number of reasons, such as initializing an important member of the object. The programmer defines a constructor by declaring a special definition in the class. This definition must have *exactly* the same name as the class and have no return type.

```
     public class ErrorMsg  {
          public  static  int  total_counter;
// A constructor definition for this class. Note that all constructors
// have the same name as the class. This constructor has no parameters:
          ErrorMsg () {
               total_counter = -1;
          }
// A regular class instance method with no parameters:
          public static void resetErrorMsgCount () {
                total_counter = 0;
          }
```

As in regular method definitions, a class can have more than one constructor definition. They are identified by their signature, or number and types of parameters.

```
     public class ErrorMsg  {
          public  static  int   total_counter;
// A constructor for this class. Note that this constructor has no
// parameters:
          ErrorMsg () {
               total_counter = -1;
          }
// A constructor for this class that is passed one parameter.
          ErrorMsg (String initialMsgText) {
               total_counter = -1;
               msgText = initialMsgText;
          }
```

The consumer class does not explicitly *call* the constructor method. Instead, the `new` operation implicitly calls it. The class consumer identifies which constructor to call by adding parameters to the `new` operation.

```
// Create an instance of ErrorMsg.
// The constructor with no parameters will be called.
     ErrorMsg      myErrorMsg = new      ErrorMsg ();
// Create another instance of ErrorMsg.
// The constructor with one parameters will be called.
      ErrorMsg myotherErrorMsg = new ErrorMsg ("Some Text");
```

A constructor can execute some other constructor in the current class. This is useful if a class has more than one constructor definition but would like to share some code between them. The `this` keyword is used as an object reference variable that points to the current object.

```
// The standard constructor for a class
// It is passed no parameters.
          ErrorMsg () {
              total_counter = -1;
          }
// A constructor for this same class that is passed one parameter
// It will call the standard constructor (to initialize the counter member).
          ErrorMsg (String initialMsgText) {
               this.ErrorMsg       // Perform the standard constructor
               msgText = initialMsgText;
          }
```

Exercises: Class Members

It's time to revisit the example classes and try out all these new ideas.

1. Using a text editor, edit the Java source file ErrorMsg.java in the java4cobol directory. You will add some additional variables to it and examine how a calling program can access these variables. Add the **bolded** lines of code to the beginning of the file so that it looks like this:

```
public class ErrorMsg {
// Define some public class instance variables.
          public String msgText = " ";
          public int  msgSize;
// Define some private class instance variables.
          private int  counter = 0;
          char interfaceInUse;
// Define a public method.
          public void setErrorMsg (String inputMsg) {
// Modify some of the private variables.
              counter = counter + 1;
              interfaceInUse = 'S';
// Modify one of the public variables. Set this variable to the text
// String parameter.
              msgText = inputMsg;
// Set this variable to the length of the text String.
              msgSize = msgText.length ();
```

```
// Return from this method. Since this method has no return value
// (i.e.,
// it is declared as void), no return statement is necessary.
          }

...
```

2. Compile the class in the DOS command window:

```
➔  javac ErrorMsg.java
```

3. Modify HelloWorld so that it uses these new members. With the text editor, edit the Java source file HelloWorld.java in the java4cobol directory. Add the **bolded** lines of code after the third `println` statement so that it looks like this:

```
// Set the text item in ErrorMsg to some text String, and print its
// contents:
          myErrorMsg.setErrorMsg ("Some Text");
          tempMsg = myErrorMsg.getErrorMsg ();
          System.out.println (tempMsg);
// Print the contents of ErrorMsgs String data member directly.
          System.out.println (myErrorMsg.msgText);
```

4. Save the file, and then compile and execute the program in the DOS command window:

```
➔  javac HelloWorld.java
➔  java HelloWorld
```

Your output window should contain these lines:

```
C:>javac ErrorMsg.java

C:>javac HelloWorld.java

C:>java HelloWorld
Hello World!

Some Text
Some Text
Some New Text
...
```

The second "Some Text" line in the output window is the result of your new `println` statement. This statement printed the text in data member `msgText` directly from the object `myErrorMsg`. It can do this because `msgText` has been defined as a public data member in the class `ErrorMsg`. For the sake of brevity, the entire output window is not shown here.

5. Now you will try to access `ErrorMsg`'s other data members from HelloWorld. Add the **bolded** lines of code after your new `println` statement so that it looks like this:

```
// Set the text item in ErrorMsg to some text String, and print its
// contents:
          myErrorMsg.setErrorMsg ("Some Text");
          tempMsg = myErrorMsg.getErrorMsg ();
          System.out.println (tempMsg);
// Print the contents of ErrorMsg's String data member directly.
          System.out.println (myErrorMsg.msgText);
// Try to access the other data member (s) in ErrorMsg.
          System.out.println ("msgSize " + myErrorMsg.msgSize);
          System.out.println ("counter " + myErrorMsg.counter);
```

6. What happens when you try to compile this class (remember to save the file as a text file)?

```
➔  javac HelloWorld.java
```

The compiler knows that private data members cannot be accessed, so don't try it again!

7. Remove the offending statement, and recompile. Now try running the modified program:

```
➔  javac HelloWorld.java
➔  java HelloWorld
```

Your output window should look like this:

```
Hello \World!

Some Text
Some Text
msgSize 9
Some New Text
Some Text for #2
Some New Text
SOME NEW TEXT
```

The data member `msgSize (9)` is now printed in the output window.
What is the conclusion? Public members (`msgSize` and `MsgText`) can be accessed outside the `ErrorMsg` class. However, private members (`counter`) cannot be.

8. Let's build a new method in `ErrorMsg` that will return `counter`. You will also adjust `ErrorMsg` so that `counter` is incremented for each method. Along the way, you will correct the implementation of one of the `setErrorMsg` methods to make sure that the overloaded version of this method (the one with one parameter) reuses the original method.
9. Open the `ErrorMsg` class in the editor. Add the **bolded** lines of code to the first `getErrorMsg` method so that it looks like this:

```
          public String getErrorMsg () {
             String returnMsg;
// Modify some of the private variables.
             counter = counter + 1;
             interfaceInUse = 'G';
// Set the local variable returnMsg to the data member msgText.
              returnMsg = msgText;
// Return from this method, and return the String variable.
              return (returnMsg);
           }
```

10. Add this new method to `ErrorMsg`:

```
// Define a method to return counter.
          public int getCounter () {
// Return from this method with the value of 'counter'.
              return (counter);
          }
```

11. Save this class, and then compile it in the command window:

```
➔  javac ErrorMsg.java
```

12. Now, you'll adjust HelloWorld so that it uses these new members.
Open the HelloWorld class in the editor. Add the **bolded** lines of code after the print of `msgSize` so that it looks like this:

```
// Print the other data members in ErrorMsg.
         System.out.println ("msgSize " + myErrorMsg.msgSize);
         System.out.println ("interface " + myErrorMsg.interfaceInUse);
         System.out.println ("counter " + myErrorMsg.getCounter ());
```

13. Save the source file, then recompile and run the modified program in the command window:

```
➔ javac HelloWorld.java
➔ java HelloWorld
```

Your output window should look like this:

```
Hello World!

Some Text
Some Text
msgSize 9
interface G
counter 3
Some New Text
Some Text for #2
Some New Text
SOME NEW TEXT
```

`msgSize`, `interface`, and `counter` are now printed in the output window.

14. As another experiment, you'll adjust HelloWorld so that it prints out the data members after it calls the overloaded `getErrorMsg` method (that is, the one that accepts a parameter). Add the **bolded** lines of code to the end of the `HelloWorld` class so that it looks like this:

```
// Call the new variation on getErrorMsg.
        tempMsg = myErrorMsg.getErrorMsg ('U');
        System.out.println (tempMsg);
// Print the public variables after performing this overloaded call.
        System.out.println ("msgSize " + myErrorMsg.msgSize);
        System.out.println ("interface " + myErrorMsg.interfaceInUse);
        System.out.println ("counter " + myErrorMsg.getCounter ());
```

15. Save the source file, then recompile and run the modified program in the command window:

```
➔ javac HelloWorld.java
➔ java HelloWorld
```

Your output window should look like this:

```
Hello World!

Some Text
Some Text
msgSize 9
interface G
counter 3
Some New Text
Some Text for #
Some New Text
SOME NEW TEXT
msgSize 13
interface G
counter 7
```

16. For the next experiment, you will examine the behavior of local variables and observe how they are different from class variables. Add the **bolded** lines of code to the `setErrorMsg` method and the first `getErrorMsg` method so that they look like this:

```
// Define a public method.
          public void setErrorMsg (String inputMsg) {
// Define a local variable and increment it.
               int localCounter = 0;
               localCounter = localCounter + 1;
// Modify some of the private variables.
               counter = counter + 1;
               interfaceInUse = 'S';

          . . .
// Define another public method.
          public String getErrorMsg () {
               String returnMsg;
// Define a local variable and increment it.
               int localCounter = 0;
               localCounter = localCounter + 1;
// Modify some of the private variables.
               counter = counter + 1;
               interfaceInUse = 'G';
          }
```

17. Add the **bolded** lines of code to the end of `ErrorMsg`:

```
// Define a method to return localCounter.
          public int getLocalCounter () {
               int localCounter = 0;
               localCounter = localCounter + 1;
// Return from this method with the value of localCounter.
               return (localCounter);
          }
```

18. Save this class, and then compile it in the command window:

```
➔ javac ErrorMsg.java
```

19. Add the **bolded** lines of code to the end of the `HelloWorld` class so that it looks like this:

```
// Call the new variation on getErrorMsg.
          tempMsg = myErrorMsg.getErrorMsg ('U');
          System.out.println (tempMsg);
// Print the public variables after performing this overloaded call.
          System.out.println ("msgSize " + myErrorMsg.msgSize);
          System.out.println ("interface " + myErrorMsg.interfaceInUse);
          System.out.println ("counter " + myErrorMsg.getCounter ());
// Print the localCounter variable.
          System.out.println
             ("localCounter " + myErrorMsg.getLocalCounter ());
```

20. Save the source file, then recompile and run the modified program in the command window:

```
➔ javac HelloWorld.java
➔ java HelloWorld
Hello World!

Some Text
Some Text
msgSize 9
interface G
counter 3
Some New Text
Some Text for #2
Some New Text
SOME NEW TEXT
```

```
msgSize 13
interface G
counter 7
localCounter 1
```

21. For this last experiment, you will work with arrays. Edit the HelloWorld source file and add these Java statements to the end of the previous statement. Enter the `checkIntArray()` method inside the brackets that define the class.

```
// Create an array of five error codes.
       int[] errorNumbersIO = {1, 2, 10, 22, 23};
// Test if any are equal to 30.

       checkIntArray (errorNumbersIO);
     }

// Test if any integers in the passed array are equal to 30.
     static void checkIntArray (int[] intArray) {
          System.out.println ();
          for (int x = 0; x < intArray.length; x++) {
                  if (intArray[x] == 30) {
             System.out.println ("Found '30' at index " + x);
             break;
           }
            else
                  {
                       System.out.println ("error Number "  + x + " = " +
   intArray[x]);
                   }
               }
        }
```

22. Save these modifications as a text file, and then compile the class in the DOS command window. (You may need to add this statement `import java.util.*;` to the beginning of your `HelloWorld` class.)

```
➔  javac HelloWorld.java
➔  java HelloWorld
error Number 0 = 1
error Number 1 = 2
error Number 2 = 10
error Number 3 = 22
error Number 4 = 23
```

23. Add these Java **bolded** statements to the end of the HelloWorld class, but before the checkIntArray() function:

```
// Test if any are equal to 30.
     checkIntArray (errorNumbersIO);
// Set an error code to 30, and call the checkIntArray function again.
    errorNumbersIO[1] = 30;
     checkIntArray (errorNumbersIO);
```

Following are the results when you execute the program:

```
}
error Number 0 = 1
error Number 1 = 2
error Number 2 = 10
error Number 3 = 22
error Number 4 = 23

error Number 0 = 1
Found '30' at index 1
```

Since you are a very clever programmer, try out these adjustments on your own in the HelloWorld applet. For your convenience, the completed applet sample code is included on the CD-ROM.

Reviewing the Samples

Let's review the changes you've made to ErrorMsg and HelloWorld. Try to relate the sample source statements to the result (the output) each statement creates. If necessary, rerun the samples or look at the complete source code for this exercise on the CD-ROM. Feel free to experiment by yourself.

- The ErrorMsg class was adjusted to include some private and package data members.
- HelloWorld could access the public (msgText and msgSize) and package (interfaceInUse) data members directly but could not access the private (counter) data members.
- You needed to create a new method (getCounter) in order to access this private data member.
- You changed the second version of the method getErrorMsg (in ErrorMsg) so that it called the first version and then performed its custom logic. Notice that

you did not need an object reference variable (like myErrorMsg) for this statement; you can just call the method. In this case, the compiler assumes that you mean this object (that is, the current object).

```
// Perform the standard getErrorMsg method.
                returnMsg = getErrorMsg ();
// Convert to all uppercase, if requested.
                if (caseFlag == 'U')
                    ...
```

- Local variables are automatically created and then destroyed as needed. They are not shared by various blocks of code, even if these blocks of code are members of the same class. The variable localCount was defined, initialized, and incremented in each method. Each method created its own copy of this temporary variable.

Here are a few more important notes:

- The println method can accept one String parameter. If your source combines many different variables with the + operator, these are all converted into Strings, and then combined into a single String, which is passed to println. Therefore,

  ```
  System.out.println ("msgSize " + myErrorMsg.msgSize);
  ```

  results in this output window:

  ```
  msgSize 9
  ```

- The result of a method that returns a value (for example, getCounter returns an integer value) can be used right away in a single statement. There is no need to store the result of the method in a variable, if the result is only to be used for the current statement. Therefore, this statement

  ```
  System.out.println
            ("counter " + myErrorMsg.getCounter ());
  ```

  results in this output window:

  ```
  counter 3
  ```

5 Inheritance, Interfaces, and Polymorphism

In This Chapter

- Inheritance and Object-Oriented Design
- Inheritance and Objects
- Inheriting Methods
- Redefining a Method
- Extending a Method
- Why Inheritance?
- Inheritance, Objects, and COBOL
- More COBOL Object-Oriented Design Patterns
- Inheritance and Java
- Hiding Variables and Methods
- The `this` Variable
- Java Interfaces
- Using Interfaces
- Hiding Methods and Members
- Polymorphism
- Exercises
- Reviewing the Samples

This chapter really begins to get into object-oriented concepts. Understanding these concepts is critical to becoming a sound Java programmer.

Inheritance and Object-Oriented Design

An important benefit of object-oriented (OO) languages like Java is their ability to use the concept of *inheritance*. This technique allows a class to be created that *extends* another class in some fashion. The new class might enhance or specialize

the capabilities of the original class. This approach is a great alternative to the dreaded suggestion, "I'll just copy that program and make a few changes." Before long, the developer will have two versions of the same program to maintain.

When one class inherits from another class, a relationship is defined between the two. The original class is referred to as the *base class* or the *superclass*, and the one that extends it is called the *derived class* or the *subclass*. Subclasses can themselves be inherited by other classes; and these new classes can also be inherited. A typical object-oriented design presents a hierarchy of related base classes and derived classes. As a matter of fact, Java itself is an object-oriented design: Everything is inherited from the Object base class (see Figure 5.1).

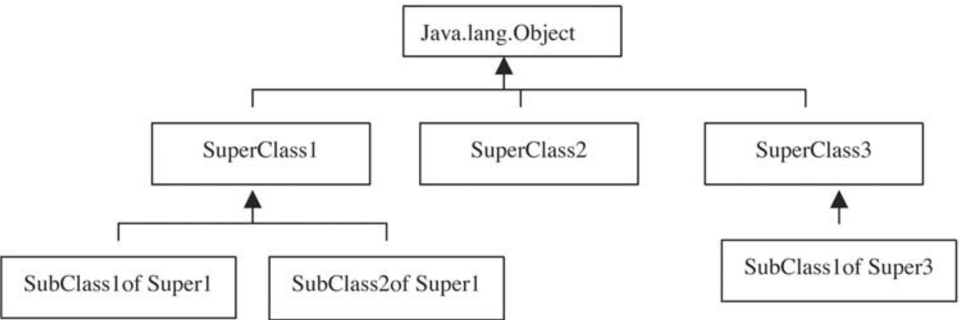


FIGURE 5.1
The subclasses in this diagram inherit from various superclasses. All of these classes ultimately inherit from the Java base class `Object` (`Java.lang.Object`).

To get the greatest benefit out of inheritance, a designer should attempt to organize class structures in advance (a "bottom-up" design). To do this, the designer must analyze application requirements and recognize functions that are similar or are likely to be reused. These must be further examined to *abstract* the common functionality into an organized set of base classes. Then specializations of these classes are designed and constructed to meet the unique application requirements (a "top-down" implementation).

The process of defining the appropriate base classes and the appropriate class hierarchy is one of the most challenging aspects of proper object-oriented design. However, using a well-designed class hierarchy infrastructure is one of the simplest and most efficient processes in object-oriented coding.

An object-oriented design uses inheritance for other benefits as well. It can provide for efficient enhancements to a system. Modifications to a system need be made only to the appropriate class in the class hierarchy; all derived classes will automatically benefit from this enhancement.

Inheritance and Objects

From a conceptual perspective, each subclass, in a sense, *contains* its superclass (i.e., its parent class).

Think about it this way. Each instance of a subclass automatically creates an instance of its *parent* class. Therefore, an object whose class type is a derived class logically has its own identity *and* its parent's identity. Both the base object and the derived object are instantiated (created) when the derived object is instantiated. Afterward, the derived object can call functions in the base object without first creating the base object.

In the same way, a class (or program) that creates and uses a derived class can directly call methods in the superclass without explicitly creating the superclass. This is because all of the superclass's public methods become public methods of the subclass. The subclass can add its own members or modify the behavior of existing methods, but by default, the subclass contains all of the functions of the parent class.

Java actually loads both the base object(s) and the derived object at the same time. It then calls the constructor for each class in hierarchal order, so only one object really exists. But when coding a class inheritance hierarchy, it is helpful to think about a derived class as "containing" its super classes, as you will see in a few pages.

Inheriting Methods

Suppose that the `ErrorMsg` class is derived from a `TextMessage` class, and `TextMessage` contains a public method called `getTranslation`. This method returns a translated version of the message. `ErrorMsg` defines only one public method, named `getErrorMsg`. Even so, `ErrorMsg` appears to have two public methods, `getErrorMsg` and `getTranslation`, as shown in Figure 5.2.

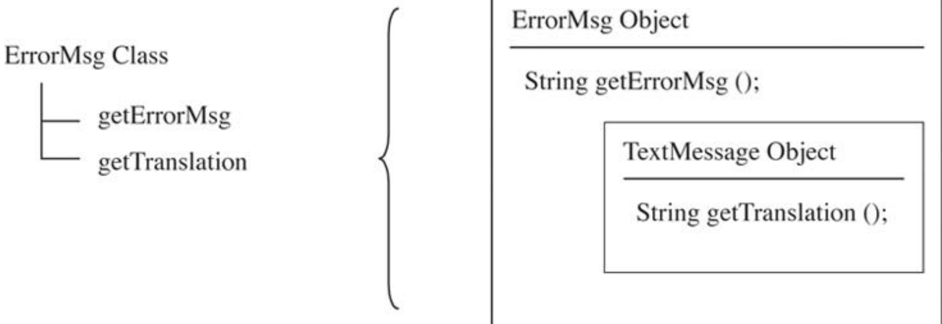


FIGURE 5.2
An object from the `ErrorMsg` class contains any members from its superclass(es). Its public interface is the sum of all of the public methods it and its superclass(es) define.

CALLER CLASS

A class that uses this type of derived class simply needs to create the `ErrorMsg` object. The `TextMessage` object will be automatically created. For example:

```
// Create an instance of ErrorMsg.
// This will automatically create a new instance of ErrorMsg,
// which will contain the members of the base class TextMessage.
     ErrorMsg myErrorMsg = new ErrorMsg ();
// Call the translate method in the base class TextMessage, using the
// ErrorMsg object reference variable.
     String FrenchText = myErrorMsg.getTranslation ();
```

Notice that the caller program creates only the subclass (`ErrorMsg`) and does not need to explicitly create the superclass (`TextMessage`). Yet it still can call the parent class method (`getTranslation`), using the reference variable (`myErrorMsg`) of the subclass. The caller class simply needs to create the subclass; all of the public methods of any inherited base class(es) are instantly available. Likewise, all data members of `TextMessage` (both public and private data members) are created at the same time. Public data members in `TextMessage` can be directly accessed by the caller class.

One of the first things that happens when an object is created is that the system calls every constructor in the base class(es) for this object. This is how the compiler makes sure that every class "contains" its base class. The developer does not have to consciously do this: The compiler and runtime system do it automatically.

Redefining a Method

What if the developer of `ErrorMsg` needs to extend the `getTranslation` function of `TextMessage` in some fashion? In this case, the developer could simply define a function with that name in the class definition for `ErrorMsg`. Now, the `getTranslation` method in `ErrorMsg` will be performed by the caller class and not the `getTranslation` method in `TextMessage`.

ErrorMsg Class

This `ErrorMsg` class defines its own `getTranslation` method. It will be used instead of the `getTranslation` method in `TextMessage` when `ErrorMsg.getTranslation()` is called (see Figure 5.3).

```
// This class extends the TextMessage class.
// Therefore, TextMessage is the superclass, and ErrorMsg is the subclass.
     public class ErrorMsg extends TextMessage {

          public void setErrorMsg (String inputMsg) {
               ...
// Some logic:
               ...
          }
     }
// Define a method named getTranslation. This overrides the method in the
//  base class.
          public String getTranslation () {
               String localString;
               ...
      // Some logic
               ...
               return (localString);
                     }
     }
```

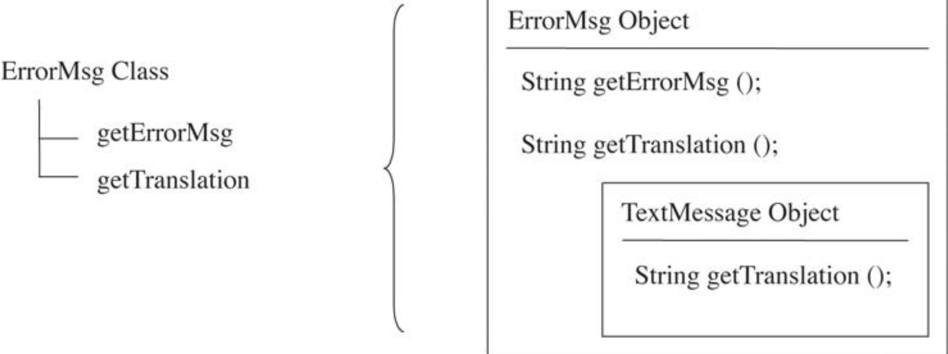


FIGURE 5.3
In this definition of `ErrorMsg`, the derived class (`ErrorMsg`) overrides the `getTranslation` method of its superclass (`TextMessage`). Yet the public interface to `ErrorMsg` remains the same.

CALLER CLASS

Notice that the way the consumer class uses `ErrorMsg` does not change, even though you are performing a new function:

```
     ErrorMsg myErrorMsg = new ErrorMsg ();
// Call the translate method in the subclass ErrorMsg.
     String FrenchText = myErrorMsg.getTranslation ();
```

EXTENDING A METHOD

The `getTranslation` method in the derived class (`ErrorMsg`) could call the original `getTranslation` method in the superclass (i.e., `TextMessage`) if needed. This is often necessary with derived methods in order to perform the original method, plus any specific logic in the derived method. The keyword *super* is the reference variable for the parent object (i.e., the object that was automatically created for this subclass).

ERRORMSG CLASS

This `ErrorMsg` class defines its own `getTranslation` method. This method will perform some specialized logic for the `ErrorMsg` class. It will still use the `getTranslation` method in the base class to get the translated text, and then will convert the translated text to uppercase.

```
public class ErrorMsg extends TextMessage {


        public void setErrorMsg (String inputMsg) {
            ...
    // Some logic
            ...
        }
    }
// Define a method named getTranslation. This method overrides the
// method with the same name in the base class.
        public String getTranslation () {
// Call the base object's method.
            String localString = super.getTranslation ();
// Since this is an error message, change it to all uppercase.
// Perform the toUpperCase function (this is a method that every String has).
            localString = localString.toUpperCase ();
            return (localString);          }
    }
```

The class hierarchy for `ErrorMsg` does not change, even though you are performing a new function (see Figure 5.4).

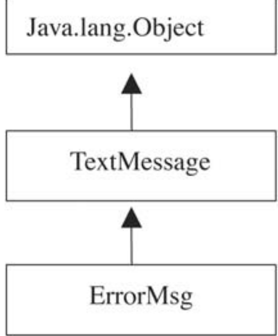


FIGURE 5.4
The class hierarchy for `ErrorMsg`.

CALLER CLASS

Again, the consumer class using `ErrorMsg` does not change, even though you are performing a new function

```
      ErrorMsg    myErrorMsg     = new    ErrorMsg ();
   // Call the translate method in the subclass ErrorMsg.
       String FrenchText = myErrorMsg.getTranslation ();
```

WHY INHERITANCE?

Inheritance is not a concept with a good COBOL equivalent, so you're probably asking, "What's the big deal here? Why would anyone want to do this?" The answer is best explained in the context of design patterns, a theory that has become an important concept in OO circles. (This theory is described in *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson, and Vlissides, published by Addison Wesley Professional in 1994.) The basic premise of design pattern theory is that many development issues can be organized into related groups. If one uses predefined and generally accepted approaches to solving the problems for that group, developers will readily recognize the approach. As a result, the developer will be guided toward the best implementation of a solution for that problem. The quality of the solutions built based on well understood approaches, or patterns, is improved. Further, the time spent on developing the solution is reduced.

A fundamental design pattern is the idea of objects, or reusable blocks of code, that perform a specific task. The theory is that application development problems (or, more precisely, their solutions) are best addressed by breaking down the problem and implementing the most elegant, flexible solution possible. According to object-oriented theory, a solution based on objects will be cheaper and easier to maintain in the long run. An object-oriented analysis attempts to examine a problem using these criteria:

- **What is generic and what is specific about a particular solution?** That is, how can I represent my objects so that the characteristics shared by many objects are isolated from those that are unique to my object?
- **How can I abstract my design so that the things likely to change are isolated from those that are constant?** Since maintenance and customization is the most expensive part of any programming effort, it is important to concentrate the portions with high activity in one place, separate from those with modest activity.

- **How can I build a solution that can adapt to changes that have not yet been anticipated?** What does it mean to build a structure with flexibility in the right places, without compromising reuse? A rigorous design can often be molded to fit new conditions, without necessarily collapsing the original solution.

When you feel up to it, read the *Design Patterns* book. It will provide unique assistance in understanding complex problems and give you valuable insights into building elegantly designed solutions.

Inheritance is another example of a design pattern. Inheritance allows you to organize elementary functionality into your base classes. This functionality is then extended in derived classes. However, all classes that inherit a super-type *will support some form of the basic functionality*. A consumer class can treat all objects of this super-type as if they are the same. Important object-oriented design goals are met, that is, the stable and generic portion of your solution is isolated (in the base class) from the more dynamic and specific portion (in the derived class).

At the same time, these boundaries must be fluid. Derived classes may need to enhance, or possibly supplant, functionality in the base classes, without compromising the structural integrity of the system (i.e., without making things so complicated that they become unmanageable). Since derived classes can override base methods, derived classes can always extend existing method behaviors as needed.

Inheritance, Objects, and COBOL

The simplest way to understand inheritance from a COBOL perspective is to imagine that a subroutine's interface (that is, the items in its LINKAGE SECTION) can be promoted into another subroutine's interface. Imagine that a subroutine (NEWSUB, for example) could pick one other subroutine (it must be a subroutine that NEWSUB calls) and that that called subroutine's LINKAGE SECTION would automatically become part of NEWSUB's LINKAGE SECTION. If NEWSUB did not have a LINKAGE SECTION defined, then one would be automatically created for it.

NEWSUB COBOL

Here is a definition for a shell of a COBOL program named NEWSUB:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. NEWSUB.
DATA DIVISION.
WORKING-STORAGE SECTION.
```

```
PROCEDURE DIVISION.
START-PROGRAM SECTION.
START-PROGRAM-S.

         EXIT PROGRAM.
```

Not much there!

NEWSUB COBOL: MYSUB

Now, this is the definition for NEWSUB after NEWSUB *inherits* from MYSUB:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. NEWSUB.
DATA DIVISION.
WORKING-STORAGE SECTION.

LINKAGE SECTION.

01 PASSED-MYSUB-CONTROL.
    03   MYSUB-ACTION-SWITCH    PIC X.
         88 MYSUB-ACTION-EVALUATE        VALUE "E".
         88 MYSUB-ACTION-SET-AND-EVALUATE VALUE "S".
    03   MSG-TEXT               PIC X(20).
    03   MSG-SIZE               PIC 9(8).
01 PASSED-TEXT-STRING           PIC X(20).

PROCEDURE DIVISION USING PASSED-MYSUB-CONTROL,
                         PASSED-TEXT-STRING.
 START-PROGRAM SECTION.
 START-PROGRAM-S.
*   Check the passed parameters to see if you should just call MYSUB.
     IF MYSUB-ACTION-EVALUATE OF PASSED-MYSUB-CONTROL OR
        MYSUB-ACTION-SET-AND-EVALUATE OF PASSED-MYSUB-CONTROL
          CALL "MYSUB" USING PASSED-MYSUB-CONTROL,
                             PASSED-TEXT-STRING.

     EXIT PROGRAM.
```

As a result of this change, NEWSUB appears to have some of the same properties of MYSUB. A program (perhaps CALLER) can call either MYSUB or NEWSUB, with the same control structures and get the same results.

So far, little of this makes much sense. Why not just have CALLER call MYSUB directly? One answer might be that you need to enhance MYSUB without affecting other programs that already call MYSUB. In this case, NEWSUB could perform some additional new logic and call MYSUB for the old logic.

Suppose you want NEWSUB to always translate the stored text field to all uppercase. Programs that needed this feature plus the standard features of MYSUB could call NEWSUB. NEWSUB would now look like this:

```
NEWSUB COBOL
IDENTIFICATION DIVISION.
PROGRAM-ID. NEWSUB.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 TEMP-TEXT-STRING             PIC X(20).

LINKAGE SECTION.

01 PASSED-MYSUB-CONTROL.
    03   MYSUB-ACTION-SWITCH     PIC X.
         88 MYSUB-ACTION-EVALUATE          VALUE "E".
         88 MYSUB-ACTION-SET-AND-EVALUATE  VALUE "S".
     03     MSG-TEXT             PIC X(20).
     03     MSG-SIZE             PIC 9(8).
01 PASSED-TEXT-STRING           PIC X(20).

PROCEDURE DIVISION USING PASSED-MYSUB-CONTROL,
                         PASSED-TEXT-STRING.
START-PROGRAM SECTION.
START-PROGRAM-S.
*   Check the passed parameters to see if you should just call MYSUB.
     IF MYSUB-ACTION-EVALUATE OF PASSED-MYSUB-CONTROL OR
       MYSUB-ACTION-SET-AND-EVALUATE OF PASSED-MYSUB-CONTROL
          MOVE PASSED-TEXT-STRING TO TEMP-TEXT-STRING
          CALL "MYSUB" USING PASSED-MYSUB-CONTROL,
                             TEMP-TEXT-STRING
*   Convert the stored text after calling MYSUB.
          INSPECT MSG-TEXT CONVERTING
            "abcdefghijklmnopqrstuvwxyz" TO
            "ABCDEFGHIJKLMNOPQRSTUVWXYZ".

       EXIT PROGRAM.
```

As you can see, this approach can meet some basic object-oriented goals. Generic behavior is encapsulated in MYSUB, and more specific behavior is established in NEWSUB.

More COBOL Object-Oriented Design Patterns

There is an even more useful example of inheritance-like behavior in COBOL. As you've seen, a COBOL subroutine is generally a serviceable mechanism to encapsulate some algorithm. Sometimes, however, the restrictions of a COBOL subroutine hinder its use in complex solutions. This is because the subroutine is written as a *black box*, a device with well-defined inputs and equally well-defined outputs.

What happens if your requirements are such that the subroutine fulfills 80 or 90 percent of your requirements but not 100 percent? Normally, you'll have to choose between these solutions:

- Extend the function of the subroutine to meet your requirements.
- Code the specific requirements in your CALLER.

Either solution is fine, provided it is technically possible, but each has some potential deficiencies:

- What if these requirements are too complex or inappropriately unique for inclusion in a subroutine? What if my extensions might break the original design objectives of the subroutine?
- What if the requirements can only be met by modifying the subroutine's logic, not simply its input parameters? For example, it may be necessary to insert additional processing logic into MYSUB's evaluation function. This may not be possible in either NEWSUB or in CALLER.

Faced with these dilemmas, COBOL developers often make the fateful choice to copy the subroutine's logic into another program and rework it to meet the requirements at hand. This is a classic example of how reuse objectives are overwhelmed by the complexities of a particular requirement. There is, however, a potential third solution to subroutine reuse in a complex environment. This solution requires planning in advance, but it can yield substantial benefits.

Some of the object-oriented benefits of inheritance can be achieved if a subroutine returns to its caller before completing a function so that the caller can extend or complete that function.

If a subroutine allows this type of interaction, the caller can modify the subroutine's functions to meet its unique requirements. With this approach, it would not always be necessary to change the subroutine's functions in order to meet new requirements. Think of this type of subroutine not as a black box, but as a box with gloves that extend into the box. These gloves allow you to safely manipulate the internal workings of the box.

Believe it or not, it is possible to implement a form of inheritance with COBOL, even though the language does not directly support this concept. To do so, the interface will include a set of options switches and "traffic-cop" items. The subroutine and the calling program use these to coordinate the program flow. The option switches are set by the calling program to indicate that a particular function is to be extended. The subroutine checks this flag and, if required, returns to the calling program before completing the function. The calling program then checks the traffic-cop item to see if a *mid-function* return has been requested. If so, the caller can perform some additional code to extend the subroutine's function and then return to the subroutine. The subroutine checks the traffic cop and resumes processing the incomplete function.

In Figure 5.5, MYSUB-CONTROL is extended to include the options and traffic-cop items. Using these variables, the calling program can request that any of the functions available with ACTION-SWITCH be extended by the caller. MYSUB returns control to the caller *before* completing the function. The caller then performs any additional logic for that function as required, and then returns to the subroutine.

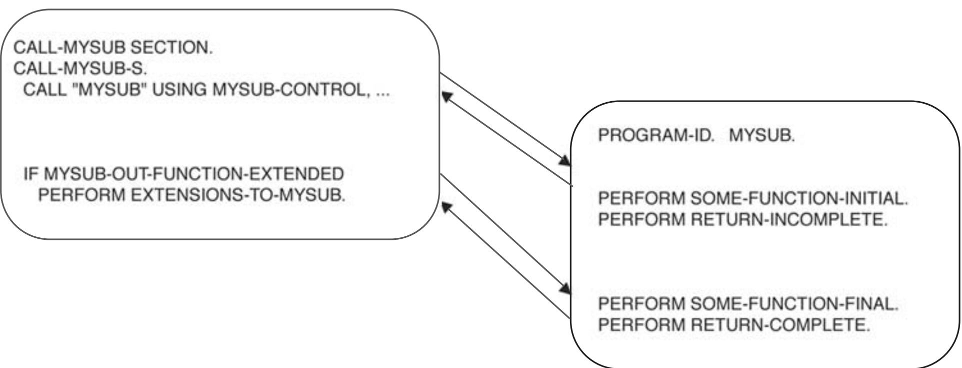


FIGURE 5.5
Extending a COBOL subroutine.

CALLER COBOL

This program will call MYSUB. MYSUB will return to this program before it is completely finished with its algorithm so that this program can perform some additional logic. This program will then call MYSUB again so that MYSUB can complete its logic.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. CALLER.
 DATA DIVISION.
 WORKING-STORAGE SECTION.

01 MYSUB-CONTROL.
     03  MYSUB-ACTION-SWITCH               PIC X.
         88 MYSUB-ACTION-INITIALIZE       VALUE "I".
         88 MYSUB-ACTION-EVALUATE         VALUE "E".
         88 MYSUB-ACTION-SET-AND-EVALUATE VALUE "S".
         88 MYSUB-ACTION-GET-CALL-COUNTER VALUE "G".
     03  MYSUB-ACTION-EXTENDED-SWITCH      PIC X VALUE " ".
         88 MYSUB-ACTION-EXTENDED         VALUE "E".
     03  MYSUB-TRAFFIC-COP-IN              PIC X VALUE " ".
         88 MYSUB-IN-FUNCTION-EXTENDED    VALUE "E".
     03  MYSUB-TRAFFIC-COP-OUT             PIC X VALUE " ".
         88 MYSUB-OUT-FUNCTION-EXTENDED   VALUE "E".
     03  MSG-TEXT                          PIC X(20).
     03  MSG-SIZE                          PIC 9(8).
     03  MYSUB-RETURNED-CALL-COUNTER       PIC 9(10).
     03  MYSUB-PRIVATE-ITEMS               PIC X(20).

 01 TEXT-STRING                            PIC X(20).
 01 TEXT-CHANGED-COUNTER                   PIC 99999 VALUE 0.

PROCEDURE DIVISION.
START-PROGRAM SECTION.
START-PROGRAM-S.
*   Initialize MYSUB.
     SET MYSUB-ACTION-INITIALIZE TO TRUE.
     PERFORM CALL-MYSUB.
*   Prepare a text argument for MYSUB.
     MOVE "ANYTEXT" TO TEXT-STRING.
     SET MYSUB-ACTION-EVALUATE TO TRUE.
*   Request that MYSUB return before completion.
*   MYSUB will reset this switch upon completion.
     MOVE "E" TO MYSUB-ACTION-EXTENDED-SWITCH.
```

```
     PERFORM CALL-MYSUB WITH TEST AFTER UNTIL
     NOT MYSUB-OUT-FUNCTION-EXTENDED.
     DISPLAY "MSG SIZE ", MSG-SIZE,
            " MSG-TEXT ", MSG-TEXT,
            " MSG TEXT changed ", TEXT-CHANGED-COUNTER,
            " times.".
*   Call MYSUB again without changing the TEXT-STRING.
     PERFORM CALL-MYSUB WITH TEST AFTER UNTIL
          NOT MYSUB-OUT-FUNCTION-EXTENDED.
*   Notice that the changed counter did not increment, but the call
*   counter in MYSUB did.
     MOVE "G" TO MYSUB-ACTION-SWITCH.
     MOVE " " TO MYSUB-ACTION-EXTENDED-SWITCH.
     PERFORM CALL-MYSUB.

     DISPLAY "MSG SIZE ", MSG-SIZE,
            " MSG-TEXT ", MSG-TEXT,
            " MYSUB COUNTER ", MYSUB-RETURNED-CALL-COUNTER,
            " MSG TEXT changed ", TEXT-CHANGED-COUNTER,
            " times.".
 EXIT-PROGRAM.
     EXIT PROGRAM.
     STOP RUN.

CALL-MYSUB SECTION.
CALL-MYSUB-S.
     CALL "MYSUB" USING MYSUB-CONTROL, TEXT-STRING.
*   If MYSUB returns before completion, perform some extended
*   functions.
     IF MYSUB-OUT-FUNCTION-EXTENDED
          PERFORM EXTENSIONS-TO-MYSUB.

 EXTENSIONS-TO-MYSUB SECTION.
 EXTENSIONS-TO-MYSUB-S.
*   Count the number of times MYSUB changes MSG-TEXT.
*   If MYSUB is about to change MSG-TEXT, increment a counter.
     IF MSG-TEXT NOT = TEXT-STRING
          ADD 1 TO TEXT-CHANGED-COUNTER.
```

MYSUB COBOL

MYSUB will return to CALLER before it is completely finished so that CALLER can perform some additional logic. When CALLER calls MYSUB again, MYSUB will complete its logic.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MYSUB.
*************************************************************************
* This routine accepts a text item as a parameter and                   *
* evaluates the text. If the text is all spaces,                        *
* MSG-SIZE will be set to 0.                                            *
* If requested, the text item will also be stored in the                *
* passed control structure.                                             *
* If the text item is not passed, then MSG-TEXT                         *
* will be evaluated instead.                                            *
* MYSUB will count the number of times it has been called               *
* with a particular MYSUBx-CONTROL and the number of                    *
* times it has been called using all CONTROLs.                          *
*                                                                       *
* MYSUB must be called with the INITIALIZE action when                  *
* any new CONTROL is to be used.                                        *
*                                                                       *
* MYSUB can return before completing the text evaluation                *
* functions so that the caller can extend this function.                *
*************************************************************************
DATA DIVISION.

WORKING-STORAGE SECTION.
01 CALL-COUNTER          PIC 9(10) VALUE 0.
01 ARGUMENT-COUNT        PIC 9.
01 LOCAL-TEXT            PIC X(20).

LINKAGE SECTION.
*   Next is a view of MYSUB-CONTROL that is used by the
*   MYSUB subroutine:
 01 MYSUB-CONTROL.
     03   MYSUB-ACTION-SWITCH              PIC X.
          88 MYSUB-ACTION-INITIALIZE      VALUE "I".
          88 MYSUB-ACTION-EVALUATE        VALUE "E".
          88 MYSUB-ACTION-SET-AND-EVALUATE VALUE "S".
          88 MYSUB-ACTION-GET-CALL-COUNTER VALUE "G".
     03   MYSUB-ACTION-EXTENDED-SWITCH      PIC X.
          88 MYSUB-ACTION-EXTENDED         VALUE "E".
```

```
     03   MYSUB-TRAFFIC-COP-IN              PIC X.
          88 MYSUB-IN-FUNCTION-EXTENDED    VALUE "E".
     03   MYSUB-TRAFFIC-COP-OUT             PIC X.
          88 MYSUB-OUT-FUNCTION-EXTENDED   VALUE "E".
     03   MSG-TEXT                          PIC X(20).
     03   MSG-SIZE                          PIC 9(8).
     03   MYSUB-RETURNED-CALL-COUNTER       PIC 9(10).
     03   MYSUB-PRIVATE-ITEMS               PIC X(20).
*   In the subroutine's definition of MYSUB-CONTROL,
*   PRIVATE-ITEMS is redefined with items known only to the subroutine.
     03     FILLER REDEFINES MYSUB-PRIVATE-ITEMS.
            05   MYSUB-PRIVATE-COUNTER         PIC 9(8).
            05   MYSUB-OTHER-PRIVATE-ITEMS     PIC X(12).

01 TEXT-STRING          PIC X(20).

PROCEDURE DIVISION USING MYSUB-CONTROL, TEXT-STRING.

MYSUB-INITIAL SECTION.
MYSUB-INITIAL-S.
*   Perform the function to detect the number of arguments.
     PERFORM GET-ARGUMENT-COUNT.
*   Perform the INITIALIZE function if requested.
     IF MYSUB-ACTION-INITIALIZE
          MOVE 0 TO MYSUB-PRIVATE-COUNTER
     ELSE
*   Prepare the text argument and increment the counters, but only if
*    not "continuing" from an incomplete extended function.
     IF NOT MYSUB-IN-FUNCTION-EXTENDED
          IF ARGUMENT-COUNT = 2
               MOVE TEXT-STRING TO LOCAL-TEXT
          ELSE
               MOVE MSG-TEXT TO LOCAL-TEXT
          END-IF
*   Increment the Global counter.
          ADD 1 TO CALL-COUNTER
*   Increment the Instance counter.
          ADD 1 TO MYSUB-PRIVATE-COUNTER.
*   Process the ACTION-SWITCHES.
*   If requested, return the value in the counter variable.
     IF MYSUB-ACTION-GET-CALL-COUNTER
          MOVE CALL-COUNTER TO MYSUB-RETURNED-CALL-COUNTER
```

```
     ELSE IF MYSUB-ACTION-EVALUATE
*   This is a request to evaluate the text item.
          PERFORM EVALUATE-TEXT-ITEM.
*   On normal exits, clear the traffic-cop switches.
     MOVE " " TO MYSUB-TRAFFIC-COP-OUT.
 MYSUB-TRAFFIC-COP-IN.

 EXIT-PROGRAM.
     EXIT PROGRAM.

EVALUATE-TEXT-ITEM SECTION.
EVALUATE-TEXT-ITEM-S.
*   Evaluate the text item, but only if you are not resuming a
*   suspended function.
     IF NOT MYSUB-IN-FUNCTION-EXTENDED
          IF LOCAL-TEXT = SPACES
               MOVE 0 TO MSG-SIZE
          ELSE
               MOVE 1 TO MSG-SIZE
          END-IF
*   If an extended function is requested, return to the caller.
          IF MYSUB-ACTION-EXTENDED
               PERFORM SET-TRAFFIC-COPS-AND-EXIT
          ELSE
*   Else, just move the TEXT in.
               MOVE LOCAL-TEXT TO MSG-TEXT
      ELSE
*   Else, continue the extended function, and move the TEXT in.
          MOVE LOCAL-TEXT TO MSG-TEXT.

SET-TRAFFIC-COPS-AND-EXIT SECTION.
SET-TRAFFIC-COPS-AND-EXIT-S.
    MOVE "E" TO MYSUB-TRAFFIC-COP-OUT.
MYSUB-TRAFFIC-COP-IN.

SET-TRAFFIC-COPS-AND-EXIT-NOW.
    EXIT PROGRAM.

GET-ARGUMENT-COUNT SECTION.
GET-ARGUMENT-COUNT-S.
*   Set ARGUMENT-COUNT to the result.
     CALL "C$NARGS" USING ARGUMENT-COUNT.
```

This technique can be very useful as a mechanism for defining complex subroutines, that is, those whose functions can be extended by calling programs. For example, a set of complex subroutines might serve as a framework for processing multiple items in a list. The items in the list could be rows read in from a database. The list processing framework might provide standard list-management functions, such as searching and positioning. In this case, the basic control logic necessary to process a list is in the subroutine, and the calling program provides whatever specific processing is required.

With the ability to extend the subroutine built right into the subroutine, it is often possible that these types of routines can be used (reused, actually) to meet new requirements. To make this technique useful, it is important that the subroutine be coded so that it returns to the caller at key points, indicating that the function is not complete. The caller can then perform any specialized code required.

However, a word of caution here: The coding technique necessary to support this mechanism in MYSUB is very ugly and is recommended only to the courageous COBOL programmer. Further, if it is appropriate to create a front end to MYSUB (like NEWSUB), the ugliness must be promoted into NEWSUB. In that case, NEWSUB must define and manage an interface (i.e., a LINKAGE SECTION) very similar to MYSUB so as to allow the return from the MYSUB subroutine to percolate back through NEWSUB into CALLER.

In summary, the concept of inheritance, although it is possible to implement in COBOL, can be difficult to implement and requires a very serious commitment to code reuse. In contrast, Java makes these types of designs readily available and directly supported by the language, which also makes it incredibly easy and intuitive.

Inheritance and Java

As I've just noted, Java makes implementation of these concepts much easier, even fundamental. New classes can naturally extend existing classes. Developers need not worry about the program flow and interface specification issues I have discussed, since the compiler handles all of that automatically. Developers are encouraged to reuse existing class definitions to meet new needs—in some cases by modifying the base classes, in others by creating specialized classes that inherit the capabilities of the base classes. In Java, code should never be copied to reproduce or modify an existing class's capabilities.

Java's syntax to define inheritance is the keyword *extends*. In the previous examples, `ErrorMsg` extended the class `TextMessage`. To add an additional level of inheritance, the example now has `ErrorMsg` extended by the classes `PopupErrorMsg` and `PrintfileErrorMsg` (see Figure 5.6).

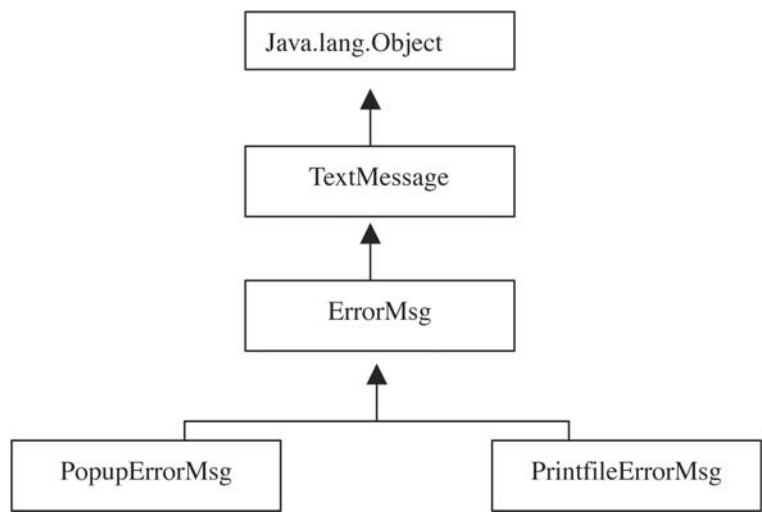


FIGURE 5.6
The class hierarchy for `ErrorMsg`, `Printfile ErrorMsg`, and `PopupErrorMsg`.

ErrorMsg Class

In this set of classes, `ErrorMsg` extends `TextMessage`, while `PopupErrorMsg` and `PrintfileErrorMsg` extend `ErrorMsg`.

```
public class ErrorMsg extends TextMessage {
        ...
        public void setErrorMsg (String inputMsg) {
        ...
        }
    }
```

PopupErrorMsg Class

```
public class PopupErrorMsg extends ErrorMsg {
}
```

PrintfileErrorMsg Class

```
public class PrintfileErrorMsg extends ErrorMsg {
}
```

In other OO languages, a class can simultaneously extend multiple base classes (this is called multiple inheritance*). In Java, a class can extend only one base class. Any Java class that does not explicitly extend another class is, by default, an extension of the* `Java.lang.Object` *base class. So all classes ultimately inherit the class* `Object`.

As previously discussed, any instance variables that belong to the base class are contained in the derived class (assuming they are public). Also, public methods of the base class become public methods of the derived class. Of course, the derived class can define additional instance variables as well as new methods.

A Java class that extends another actually has a sort of dual identity: its parent class identity and its own identity. Therefore, *all* members in the derived class (i.e., both new members defined only in this derived class and members derived from the base class) are referenced in the same way by the calling program.

TextMessage Class

The following code snippets extend (gruesome pun intended) the previous class definitions in order to highlight the Java syntax that supports and uses these concepts.

```
public class TextMessage {

     public String msgText;
     public int  msgSize;
     ...
     public void getTranslation () {
     ...
     }
}
```

ErrorMsg Class

```
public class ErrorMsg extends TextMessage {
     ...
     public void setErrorMsg (String inputMsg) {
     ...
     }
}
```

PopupErrorMsg Class

```
public class PopupErrorMsg extends ErrorMsg {
     public int windowWidth;
     public int windowHeight;
}
```

PrintfileErrorMsg Class

```
public class PrintfileErrorMsg extends ErrorMsg {
     public int linesToSkip;
}
```

Consumer Class

A consumer class can use these derived classes in the following way:

```
// Create new instances of a popup error message and a print file error
//  message.
     PrintfileErrorMsg myPrintMsg = new PrintfileErrorMsg ();
     PopupErrorMsg   myPopupMsg = new PopupErrorMsg ();
     ...
// Capture the length of the popup message myPopupMsg.
// Note that the public member msgSize is defined in the base class
// ErrorMsg.
// Even so, the derived class instance myPopupMsg contains it, and the
// consumer class uses it as if it were part of myPopupMsg.
     int len = myPopupMsg.msgSize;
// Now, compare the size of the message to the size of the popup window.
//  This public data member is defined in the derived class, not the base
//  class.
//  Yet the same ID (myPopupMsg) is used.
     if (len >= myPopupMsg.windowWidth) {
     ...
     }
```

Classes that are related through inheritance can use the member access control *protected*. The keyword *protected* extends the visibility of a member to any derived classes. Using this keyword, a base class can define members that would be appropriate for a derived class to access but would not be appropriate for a consumer class.

TextMessage Class

```
    public class TextMessage {
// This member is available to any class.
         public String msgText;
// This member is available to any class in this package and to all
//  derived classes.
         protected int  msgSize;

         ...

    }
```

ErrorMsg Class

```
    public class ErrorMsg extends TextMessage {
// This member is available only to derived classes
// and not to other classes in the package.
         private protected int  counter = 0;
// This member is available to classes in this package.
               char     msgNumber;
// This member is available only to this class.
         private char     interfaceInUse;

              ...
        }
```

A class's methods can also be extended by a subclass. That is, a derived class can extend the capabilities of its base class's methods. This is called *overriding* the method. A derived class might override a base class's method to provide some specialized version of this method. A consumer class that creates a derived class can, therefore, call derived functions in the same way (i.e., with the same name and the same parameters) that the original (or base) function was called. The Java runtime will figure out which class method to call (in the original class or in the derived class), based on the actual type of the object. Suppose you have a class hierarchy as shown in the following sections.

ErrorMsg Class: Override

```
public class ErrorMsg extends TextMessage {

     ...

     public void setErrorMsg (String inputMsg) {
     ...
     }
}
```

PopupErrorMsg Class

```
public class PopupErrorMsg extends ErrorMsg {
     public int windowWidth;
     public int windowHeight;
}
```

PrintfileErrorMsg Class

```
public class PrintfileErrorMsg extends ErrorMsg {
     public int linesToSkip;
     ...

     public void setErrorMsg (String inputMsg) {
     ...
     }
}
```

The classes `ErrorMsg` and `PrintfileErrorMsg` both define a method called `setErrorMsg`, whereas `PopupErrorMsg` does not define this method. However, a consumer class can call `setErrorMsg` using any of these three object types.

Consumer Class

The `Consumer` class can now call `setErrorMsg`, which will exist in all three of these classes.

```
// Create three objects of various types.
     ErrorMsg          myErrorMsg = new ErrorMsg ();
     PopupErrorMsg     myPopupMsg = new PopupErrorMsg ();
     PrintfileErrorMsg myPrintMsg = new PrintfileErrorMsg ();

// Create an object of type PrintfileErrorMsg, even though its nominal
// type is ErrorMsg.
     ErrorMsg     myAnyMsg  = new PrintfileErrorMsg ();
     ...
// Now, call the setErrorMsg function for each of the objects.
     myErrorMsg.setErrorMsg ("Any Text");
     myPopupMsg.setErrorMsg ("Any Text");
     myPrintfile.setErrorMsg ("Any Text");
     myAnyMsg.setErrorMsg ("Any Text");
```

The first two function calls perform the method `setErrorMsg` as defined in the class `ErrorMsg`. The next two function calls will perform the method as defined in the class `PrintfileErrorMsg`, since these objects belong to a class that has overridden (i.e., defined new versions of) this method.

You may have noticed that the object type of `myAnyMsg` appears to be a little ambiguous. Is it of type `ErrorMsg` or of type `PrintfileErrorMsg`? There are two answers to this question. Since any derived object always contains its base class(es), one answer is that myAnyMsg is of both types. However, since `myAnyMsg` was created using the constructor for the class `PrintfileErrorMsg`, it is a `PrintfileErrorMsg`. And since `PrintfileErrorMsg` contains an override for the method `setErrorMsg`, the `setErrorMsg` method for this object type will be called.

PrintfileErrorMsg Class

Often, a derived class must perform the original function in the base class. In this case, the derived class can use the operator `super` to refer to the function in the base class. Note that the base object is not explicitly created with a `new` statement in the derived class. Instead, it is automatically created at the same time as the derived object.

```
public class PrintfileErrorMsg extends ErrorMsg {
     public int linesToSkip;
     ...

     public void setErrorMsg (String inputMsg) {
```

```
    // Call the setErrorMsg function in the base class ErrorMsg.
                  super.setErrorMsg (inputMsg);
                  if (msgSize != 0) {
                       linesToSkip = 1;
                   }
            }
      }
```

Sharing Variables and Methods

When a class derives from another class, it inherits all of the class data members and methods from its base classes if the scope of those data members is not private. This means that all of the variables and methods defined in the base classes are available in the derived classes. For example, look carefully at the statement in `PrintfileErrorMsg` that sets `linesToSkip`:

```
                  if (msgSize != 0) {
                       linesToSkip = 1;
                  }
```

Where did the variable `msgSize` come from? `PrintfileErrorMsg` did not explicitly define this variable. The only one defined is in `TextMessage`, which `ErrorMsg` inherits.

But remember, `PrintfileErrorMsg` inherits from `ErrorMsg` (which inherits from `TextMessage`). Therefore, `ErrorMsg`, `PrintfileErrorMsg`, and `TextMessage` all share the variables defined in `TextMessage`.

Any of the class instances in this hierarchy can treat `msgSize` as if it is contained in this class instance. At the same time, if any of these class instances modifies `msgSize`, then all of the class instances will see this modification. How does this work? When an instance of `PrintfileErrorMsg` is created, it automatically creates an instance of `ErrorMsg`, which automatically creates an instance of `TextMessage`. The variable `msgSize` is created when `TextMessage` is created, and all of these instances share this single copy of the variable `msgSize`.

Hiding Variables and Methods

A derived class *can* create its own copies of base class variables or methods. To understand this better, let's examine how variable declarations (class data members) are handled.

Variable declaration statements that are at the beginning of a derived class definition and that are named the same as variables in a base class will cause a new variable to be created. These new variables will hide the base class variables from classes that use the derived class (i.e., create an object of this class type) or that derive from it. Identically named variables from the base classes will not be available to consumers of this derived class.

For example, if `ErrorMsg` declared its own version of `msgSize`, then `PrintfileErrorMsg` would only be able to "see" this version of `msgSize`. The copy of `msgSize` declared in `TextMessage` would not be available to `PrintfileErrorMsg`. Only the class that hides a variable can see both versions of the variable. For example, `ErrorMsg` can reference both its version of `msgSize` and the copy of `msgSize` declared in `TextMessage` by using the `super.` reference variable:

```
if (super.msgSize != 0) {
    msgSize = 1;
}
```

The this Variable

Sometimes it is a bit unclear which variable, or method, the programmer intends to reference in a class. This ambiguity is especially true of derived classes, which naturally share names.

To clarify which variable to use, the operator `this` can be used. `this` means "the members or methods associated with this instance of the class." The qualifier `this` is a sort of generic reference variable automatically created for each object. The assignment of `msgSize` in the previous `ErrorMsg` example could be expressed as follows:

```
if (this.msgSize != 0) {
    this.linesToSkip = 1;
}
```

`this.linesToSkip` specifies the `linesToSkip` variable that is part of this instance of the `PrintfileErrorMsg` class (i.e., the object pointed to by `myPrintfile`). `ErrorMsg` could use the `this` syntax to reference the two different copies of `msgSize` it has access to, as follows:

```
if (super.msgSize != 0) {
    this.msgSize = 1;
}
```

Figure 5.7 depicts these relationships among the classes. The diagram is in the Universal Modeling Language (UML) notation for static class definitions. Each box represents a class. The top section of each box contains the class name. The next section describes the public data member names and their types. The last box describes methods explicitly defined in each class. Finally, the arrows represent the class hierarchy. In this case, `PrintfileErrorMsg` and `PopupErrorMsg` inherit from `ErrorMsg`. `ErrorMsg` inherits from `TextMessage`.

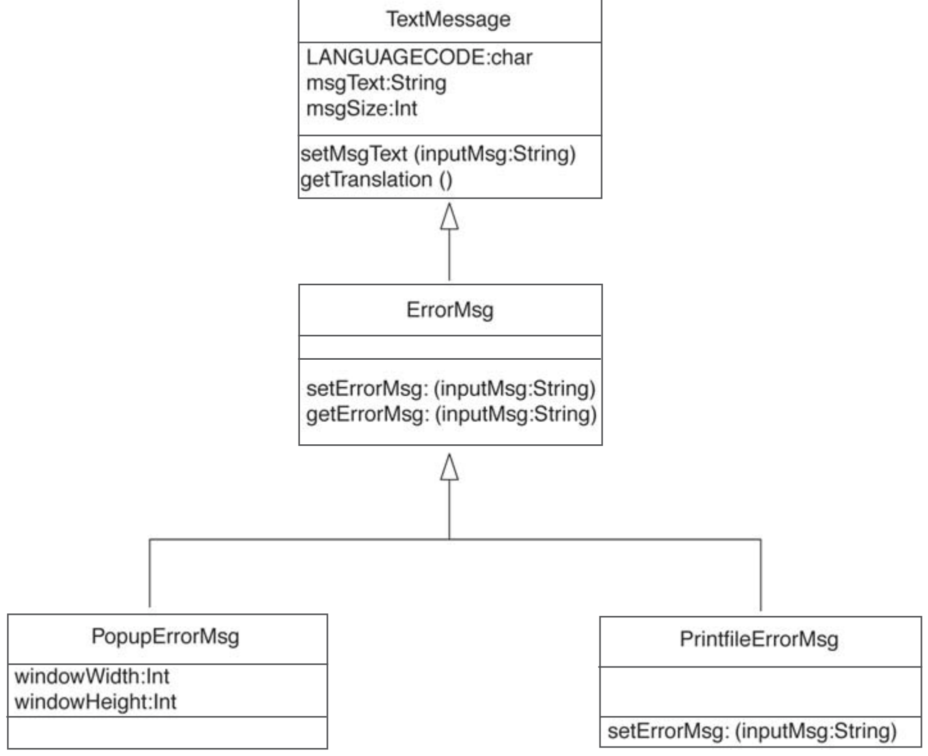


FIGURE 5.7
UML static class notation for `ErrorMsg`, `PrintfileErrorMsg`, and `PopupErrorMsg`.

JAVA INTERFACES

Although a Java class cannot extend more than one base class, it can still implement multiple *interfaces*, or method signatures that have been defined in another class. This is how Java provides some of the benefits of C++'s *multiple inheritance*, which is the ability of a single class to extend multiple base classes.

Java's designers felt that the multiple inheritance model was too complex and unnecessary for most applications, so they arrived at this approach. In Java, `super` refers to only one class (the base class for `this` class). At the same time, a class can implement methods modeled on other (unrelated) interface definitions.

A Java *interface* is like a class but lighter. It does not contain any method logic, and it is not automatically instantiated at runtime. It contains only method signatures (i.e., method names and method parameter types) and no implementation code (i.e., no code block for the method). Therefore, the class that implements an interface will inherit *no instance variables or method code.* Instead, a class that supports an interface promises to implement the methods described in that interface definition.

You can view an interface as a kind of template, or example of the types of methods an implementing class might provide. It is up to the implementing class to actually perform the appropriate logic required to support the interface definition, since there is no base class automatically created by Java for interfaces. The signature descriptions in an interface are a means of organizing and documenting the types of methods that similar classes will support.

Java uses the keyword *implements* to indicate that one class implements the interfaces (method signatures and constants) of an interface definition.

WRITELINE INTERFACE

The following describes an abstract method (and its signature) that an implementing class might need.

```
public interface Writeline {
     public void printLine ();
}
```

PRINTFILEERRORMSG CLASS

The following declares that this class implements the methods from `Writeline`. You will mimic the methods described in this interface in the class `PrintfileErrorMsg`.

```
public class PrintfileErrorMsg extends ErrorMsg
     implements Writeline {

     ...
}
```

CALLER CLASS

Using this interface technique, the class `PrintfileErrorMsg` can now receive a request to print the error message using the `printLine` method. At the same time, `PrintfileErrorMsg` can also support the methods in the base class `ErrorMsg` (such as `setErrorMsg`):

```
// Create an instance of a PrintfileErrorMsg.
       PrintfileErrorMsg myPrintMsg = new PrintfileErrorMsg ();
// Call the derived method in PrintfileErrorMsg.
       myPrintMsg.setErrorMsg ("Any Text");
// Call the interface method in PrintfileErrorMsg.
       myPrintMsg.printLine();
```

One key difference between these two method implementations is that the class `PrintfileErrorMsg` *can* override the `setErrorMsg` method, but if it does not, the method in the base class (`ErrorMsg`) will be executed. However, in order to receive a request to print its error message (the `printLine` method), `PrintfileErrorMsg` *must* implement this method, since there is no real base class method that can be performed.

USING INTERFACES

Interfaces are often supported by helper classes that perform the basic functions of that interface. These supporting classes provide basic implementations of the defined interface. A class that implements a particular interface can simply use these supporting classes as is or implement additional features in addition to the basic features provided.

```
     public class WriteLineImpl {
//  This class provides the implementation defined in the WriteLine
//  Interface.
          public void printLine (String inputMsg) {
              System.out.println (inputMsg);
        }
}

//  PrintfileErrorMsg will use the helper class WriteLineImpl to
//  implement the method defined in the WriteLine Interface.
     public class PrintfileErrorMsg extends ErrorMsg
          implements Writeline {
```

```
//  Create an instance of the helper class.
             WriteLineImpl  writeLineImpl = new WriteLineImpl();
          ...
          public void printLine (String inputMsg) {
//  Use the helper class method to implement this method.
             writeLineImpl. printLine (inputMsg);
            return;
         }
     }
```

This combination of interfaces and supporting classes is often used within a package called a *framework*. A good example of a framework is the *collections* framework first provided with Java 1.2. I will review the collections frameworks in some detail in Chapter 11.

Hiding Methods and Members

Derived classes have to be careful with the use of the *static* keyword when overriding methods in its superclasses. Instance methods (i.e., nonstatic methods) cannot override static methods in the superclass, and static methods cannot override instance methods in the superclass.

If a derived class declares a method as static, then any static methods with the same signature in the superclass(es) will not be readily available to the derived class. The superclass method can be accessed explicitly by using the *super* keyword *or* an explicit reference to the superclass type (e.g., `TextMessage.staticMethod`). If a derived class tries to override an instance method with a static method, the compiler will complain. Finally, a static method in a superclass cannot be overridden with an instance method in the derived class.

On the other hand, a derived class can hide variables in the superclass, using the *static* keyword. If a derived class declares a variable as static, then any instance variables with the same name in the superclass(es) will not be readily available to the derived class. The superclass variables can be accessed explicitly by using the *super* keyword but *not* with an explicit reference to the superclass type (e.g., `TextMessage.staticVariable`). However, a derived class can override an instance method with a static method, and vice versa, without causing the compiler to complain.

POLYMORPHISM

Polymorphism is one of those buzzwords that sounds terrifying at first but, once understood, elicits the question, "Is that all there is?"

You have already seen that invoking a method in a particular object is sometimes referred to as a *message* in OO parlance. Often, more than one class can handle a particular type of message. The actual function performed inside any class depends on the algorithm of the class. Therefore, a particular message (that is, a combination of method name and parameters) can be acted on, or interpreted, differently by different classes. Polymorphism is defined as the ability of different classes to support methods and properties of the same name but with different implementations.

A very common example of polymorphism is the ability of most PC applications to process an open (`FileName`) message. This message will result in very different actions by various applications, but the message, or interface, is the same. An originating application (such as Windows Explorer or an e-mail package) can send this same message to any application, confident that the receiving application will process it correctly.

Classes that are related by inheritance can each accept messages defined in the superclass and are, therefore, polymorphic. Take another look at the previous examples, in which the objects `myErrorMsg`, `myPopupMsg`, `myPrintfile`, and `myAnyMsg` could all receive the same message (`setErrorMsg (Any Text)`). Some of the objects were of the same class, so it would be expected that they could handle this message. Others were of different classes, yet they can all receive this message and process it correctly. This is an example of polymorphism.

Unrelated classes can also be polymorphic—that is, two unrelated classes can potentially accept the same message and respond in unique ways to it.

Suppose the system had a class called `Diagnostic`, and its purpose was to record system events for the purposes of on-site diagnostics. This class may also have a method called `setErrorMsg` that accepted a text `String` parameter. An object of this type could receive the same message (`setErrorMsg (Any Text)`) as the `ErrorMsg` class did. Java interfaces are a great way to define polymorphic behavior.

Polymorphism delivers a number of advantages to the developer. A program can send a particular message to a whole group of classes without considering the differences between those classes. This has the potential for greatly simplifying code, since distinct objects can be treated in some cases as if they are of the same type. In Java, classes that are polymorphic should be either within the same inheritance hierarchy or implement the same interface. An example might be a component-based system that consisted of many transactional objects of various types (customers, invoices, etc.). When the system needed each of the objects to commit their data to the database, a single message (`commit()`) could be sent to all of the objects.

That's all there is to it! Polymorphism is no more than the ability of multiple class types to receive the same message and process it in unique ways.

EXERCISES

It's time to visit the example classes again and try out all these new ideas.

1. Using a text editor, edit a new Java source file and name it TextMessage.java in the java4cobol directory. Enter this code in the class body. Note that the two member variables are the same name as the member variables in `ErrorMsg`. You will remove these variables from `ErrorMsg` in a moment.

```
/
//
// TextMessage
//
//
class TextMessage {
// Define some public class instance variables.
          public String msgText;
          public int  msgSize;
          public static char LANGUAGECODE = 'E';

          public void setMsgText (String inputMsg) {
// Set the msgText and msgSize variables.
                msgText = inputMsg;
// Set this variable to the size of the text String.
                msgSize = msgText.length ();
          }
          public String getTranslation () {
// Perform the translation function for this message.
// In a production environment, the translation for this message might
// be accessed from a database.
// In this sample, you will return the original text
// and "French Text" as a generic translation.
                if (LANGUAGECODE == 'E')
                      return (msgText);
                else
                      return (msgText + " -➔ French Text");
          }
}
```

2. Save this class definition as a text file, then compile the class in the DOS command window:

```
➔ javac TextMessage.java
```

3. Edit the `ErrorMsg` class with the text editor. You will define it as a class that inherits from `TextMessage` and delete some class variables. Add the **bolded** code, and remove the *italicized* code in the beginning of the file, as identified here:

```
public class ErrorMsg extends TextMessage {
// Define some public class instance variables.
          public String msgText;
          public int  msgSize;
// Define some private class instance variables.
          private int     counter = 0;
                  char    interfaceInUse;
// Define a public method.
          public void setErrorMsg (String inputMsg) {
// Define a local variable and increment it.
               int localCounter = 0;
               localCounter = localCounter + 1;
// Modify some of the private variables.
               counter = counter + 1;
               interfaceInUse = 'S';
// Modify one of the public variables. Set this variable to the text
// String that was passed as a parameter.
               msgText = inputMsg;
// Set this variable to the length of the text String.
               msgSize = msgText.length ();

               setMsgText (inputMsg);
          ...
```

4. Save this class definition as a text file, then compile the class in the DOS command window:

```
➔ javac ErrorMsg.java
```

5. Let's simplify the HelloWorld application and add a call to this new method using the text editor. Open the HelloWorld.java source file and remove the lines after this statement (but remember to leave in the two curly braces at the end of the program):

```
// Print the contents of ErrorMsgs String data member directly.
          System.out.println (myErrorMsg.msgText);
```

Add these **bolded** lines:

```
// Get the translation for this method.
          tempMsg = myErrorMsg.getTranslation ();
          System.out.println (tempMsg);
```

6. Save, compile, and rerun the HelloWorld application.

```
➔ javac HelloWorld.java
➔ java HelloWorld
```

The output should look like this:

```
Hello World!

Some Text
Some Text
Some Text
```

7. Edit the HelloWorld application, and set LANGUAGECODE to F. Add this **bolded** line after the statement that creates `ErrorMsg`:

```
// Create a new instance of the ErrorMsg class:
ErrorMsg myErrorMsg = new ErrorMsg ();
TextMessage.LANGUAGECODE = 'F';
```

8. Save, compile, and rerun the HelloWorld application, and observe the output:

```
Hello World!

Some Text
Some Text
Some Text -> French Text
```

9. Create a new instance of `ErrorMsg`, and test how it performs translations. Add these **bolded** lines to the end of HelloWorld:

```
// Create a new instance of the ErrorMsg class.
          ErrorMsg myErrorMsg2 = new ErrorMsg ();
// Set the text item to some text String, and print its contents.
```

```
myErrorMsg2.setErrorMsg ("Some Text for #2");
tempMsg = myErrorMsg2.getErrorMsg ();
System.out.println (tempMsg);
tempMsg = myErrorMsg2.getTranslation ();
System.out.println (tempMsg);
```

10. Save, compile, and rerun the HelloWorld application, and then observe the output. Notice how LANGUAGECODE applies to all instances of `TextMessage`:

```
Hello World!

Some Text
Some Text
Some Text -> French Text
Some Text for #2
Some Text for #2 -> French Text
```

11. Let's create and use the `PrintfileErrorMsg` class. Create a new java source file in the C:\java4cobol directory named PrintfileErrorMsg.java. Insert these lines of text into the class file definition:

```
//
//
// PrintfileErrorMsg
//
//
class PrintfileErrorMsg extends ErrorMsg {

     private static int outputLineSize = 80;
     public int linesToSkip = 0;
     private int charsToSkip = 0;
// Define a setErrorMsg method that establishes the number of characters
// to output in order to center the text from ErrorMsg.
     public void setErrorMsg (String inputMsg) {
          super.setErrorMsg (inputMsg);
          charsToSkip = (outputLineSize - msgSize) / 2;
// Print out this error message.
          printLine ();
     }
// Define a new method that prints this error message to the standard
// output. It will be centered in the output line (based on the size of
// outputLineSize). linesToSkip lines will be skipped first.
```

```
     public void printLine () {
          int i;
// Print out some blank lines.
          for (i=0; i != linesToSkip; i++)
               System.out.println ();
// Print out some blank characters so that the error message text is
//  centered.
          for (i=0; i != charsToSkip; i++)
               System.out.print (' ');
// Print out the error message.
          System.out.println (getErrorMsg ());
     }
}
```

12. Save and then compile this class in the DOS command window:

```
➔ javac PrintfileErrorMsg.java
```

13. Edit the HelloWorld application so that it uses this new class. Using the text editor, open the HelloWorld.java source file and add these **bolded** lines of code to the bottom of the class:

```
// Create a new instance of the PrintfileErrorMsg class.
     PrintfileErrorMsg myErrorMsg3 = new PrintfileErrorMsg ();
     myErrorMsg3.linesToSkip = 2;
// Set the text item to some text String, and print its contents.
     myErrorMsg3.setErrorMsg ("Some Text for #3");
// Print this data member in PrintfileErrorMsg.
     System.out.println ("msgSize for PrintfileErrorMsg = " +
        myErrorMsg3.msgSize);
```

14. Save, compile, and rerun the HelloWorld application. The output should look like this:

```
Hello World!

Some Text
Some Text
SOME TEXT -> FRENCH TEXT
Some Text for #2
SOME TEXT FOR #2 -> FRENCH TEXT
```

```
     Some Text for #3
MsgSize for PrintfileErrorMsg = 16
```

15. Let's create a class data member in `PrintfileErrorMsg` that hides the variable with the same name in `MessageText`. You will experiment with this variable and see how it affects the classes that use `PrintfileErrorMsg`. Using the text editor, open the PrintfileErrorMsg.java file, and add these **bolded** lines (pay particular attention to the `super.` and `this.` identifiers for `msgSize`).

```
//
//
// PrintfileErrorMsg
//
//
class PrintfileErrorMsg extends ErrorMsg {

     private static int outputLineSize = 80;
     public int linesToSkip = 0;
     private int charsToSkip = 0;
// Create a version of this variable that hides the one in TextMessage.
     public int  msgSize;

// Define a setErrorMsg method that establishes the number of
// chars to output in order to center the error msg.
     public void setErrorMsg (String inputMsg) {
          super.setErrorMsg (inputMsg);
          charsToSkip = (outputLineSize – super.msgSize) / 2;
          this.msgSize = super.msgSize + charsToSkip;
          ...
```

At the bottom of the class definition insert the bolded lines as follows:

```
// Print out the error message.
          System.out.println (getErrorMsg ());
// Print out the two msgSize variables.
          System.out.println ("this.msgSize = " + this.msgSize + ",
                    super.msgSize = " + super.msgSize);
```

16. Save and compile this class:

```
➔  javac PrintfileErrorMsg.java
```

17. Rerun the HelloWorld application. The output should look like this:

```
Hello World!

Some Text
Some Text
SOME TEXT -> FRENCH TEXT
Some Text for #2
SOME TEXT FOR #2 -> FRENCH TEXT


        Some Text for #3
this.msgSize = 48, super.msgSize = 16
MsgSize for PrintfileErrorMsg = 48
```

18. Let's create an example of an interface definition and a supporting class and have `PrintfileErrorMsg` use them. Create a new java source file in the C:\java4cobol directory named WriteLineInterface.java. Insert these lines of text into the class file definition:

```
//
// WriteLineInterface
//
//
public interface WriteLineInterface {
// Describe an abstract method (and its signature) that an implementing
// class might need.
     public void printLine ();
}
```

19. Compile this class in the DOS command window:

```
➔  javac WriteLineInterface.java
```

20. Create a new java source file in the C:\java4cobol directory named WriteLine.java. Insert these lines of text into the class file definition:

```
//
//
// WriteLine
//
//
public class WriteLine {
```

```
// Define a STATIC class method that will print out a line.
     public static void printLineWithPosition ( String line,
         int linesToSkip, int charsToSkip) {

         int i;
// Print out some blank lines.
          for (i=0; i != linesToSkip; i++)
                 System.out.println ();
// Print out some blank characters.
          for (i=0; i != charsToSkip; i++)
               System.out.print (' ');
// Print out the error message.
          System.out.println (line);

     }
}
```

21. Compile this class in the DOS command window:

```
➔ javac WriteLine.java
```

22. Edit the `PrintfileErrorMsg` class so that it uses this new interface definition and the supporting class. Add these **bolded** lines of code to the top and to the bottom of the class and remove the logic that you've placed in `WriteLine`. Your `PrintfileErrorMsg` class should look like this:

```
//
//
// PrintfileErrorMsg
//
//
class PrintfileErrorMsg extends ErrorMsg
    implements WriteLineInterface {

     private static int outputLineSize = 80;
     public int linesToSkip = 0;
     private int charsToSkip = 0;
// Create a version of this variable that hides the one in TextMessage.
     public int  msgSize;
// Define a setErrorMsg method that establishes the number of
// chars to output in order to center the error msg.
```

```
     public void setErrorMsg (String inputMsg) {
          super.setErrorMsg (inputMsg);
          charsToSkip = (outputLineSize - super.msgSize) / 2;
          this.msgSize = super.msgSize + charsToSkip;
          printLine ();
     }
// Define a new method that prints this error message to the output.
// It will be centered (based on the size of outputLineSize).
// linesToSkip lines will be skipped first.
     public void printLine () {
          WriteLine.printLineWithPosition (getErrorMsg (),
               linesToSkip, charsToSkip);
// Print out the two msgSize variables.
          System.out.println ("this.msgSize = " +
               this.msgSize + ", super.msgSize = " + super.msgSize);

          }
}
```

23. Save and compile this class:

```
➔ javac PrintfileErrorMsg.java.
```

24. Now that you've made all of these changes, rerun the HelloWorld application. The output should look the same as it did before:

```
Hello World!
null
Some Text
Some Text
SOME TEXT -> FRENCH TEXT
Some Text for #2
SOME TEXT FOR #2 -> FRENCH TEXT


Some Text for #3
this.msgSize = 48, super.msgSize = 16
MsgSize for PrintfileErrorMsg = 48
```

Although the output looks unchanged, the program is in fact very different. You've started to build an infrastructure that will support the ability to print output lines in a standard yet flexible manner. The `WriteLineInterface` definition, and the supporting `WriteLine` class, begin to provide the infrastructure necessary to manage report creation properly.

For example, this infrastructure can be extended to support standard pagination functions when every line is printed. These pagination functions might include the ability to print standard heading information as necessary at the beginning of every page, as well as other page management functions.

For the last exercise, you will experiment with the concept of polymorphism. So far, two classes, `ErrorMsg` and `PrintfileErrorMsg`, contain a method named `setErrorMsg`. The one defined in `PrintfileErrorMsg` is an overridden version of the method defined in `ErrorMsg`. You will create a third version of this method in an unrelated class called `Diagnostic`. You will then have HelloWorld call each of these methods and examine what happens as a result.

25. Create a new java source file in the C:\java4cobol directory named Diagnostic.java. Insert these lines of text into the class file definition:

```
import java.util.*;
//
// Diagnostic
//
//
public class Diagnostic {
// Define a setErrorMsg method. This method will write an error message to
// the system diagnostic output.
     public static void setErrorMsg (String inputMsg) {
// Print a banner.
          System.err.println ("====== A serious error has occurred  ====
            == ");
          Date today = new Date();
          System.err.println (today);
          System.err.println ();
// Print the message.
          System.err.println (inputMsg);
// Print a banner end.
          System.err.println ();
          System.err.println ();
          Thread.dumpStack();
          System.err.println ();
          System.err.println ("====== End of serious error message  ===
            ==");
     }
}
```

26. Save and compile this class:

```
➔  javac Diagnostic.java
```

27. Edit the HelloWorld class so that it uses this new class. Using the text editor, open the HelloWorld.java source file and add these **bolded** lines of code to the bottom of the class:

```
// Print this data member in PrintfileErrorMsg.
     System.out.println ("msgSize for PrintfileErrorMsg = " +
          myErrorMsg3.msgSize);
// Experiment with polymorphism.
     System.out.println ("---------- Experiment with polymorphism --------
         --");
// Create an error message String, and pass it to each of these
// setErrorMsg functions.
     tempMsg = "Display this message";

     System.out.println ();
     System.out.println ("~~~~~~~ setErrorMsg in ErrorMsg does this: ~~~~
        ~~~");
     myErrorMsg.setErrorMsg (tempMsg);

     System.out.println ();
     System.out.println ("~~~~~~~ setErrorMsg in PrintfileErrorMsg does
          this: ~~~~~~~");
     myErrorMsg3.setErrorMsg (tempMsg);

     System.out.println ();
     System.out.println ("~~~~~~~ setErrorMsg in Diagnostic does this : ~~
         ~~~~~");
     Diagnostic.setErrorMsg (tempMsg);
...
```

28. Save, compile, and rerun the HelloWorld application. The output should look like this:

```
---------- Experiment with polymorphism ----------

~~~~~~~ setErrorMsg in ErrorMsg does this: ~~~~~~~

~~~~~~~ setErrorMsg in PrintfileErrorMsg does this: ~~~~~~~
```

```
               Display this message
this.msgSize = 50, super.msgSize = 20

~~~~~~~ setErrorMsg in Diagnostic does this : ~~~~~~~
====== A serious error has occurred  ======
Fri Apr 16 18:08:50 PDT 1999

Display this message


java.lang.Exception: Stack trace
    at java.lang.Thread.dumpStack(Thread.java:983)
    at Diagnostic.setErrorMsg(Diagnostic.java:27)
    at HelloWorld.main(HelloWorld.java:69)

====== End of serious error message  =====
```

REVIEWING THE SAMPLES

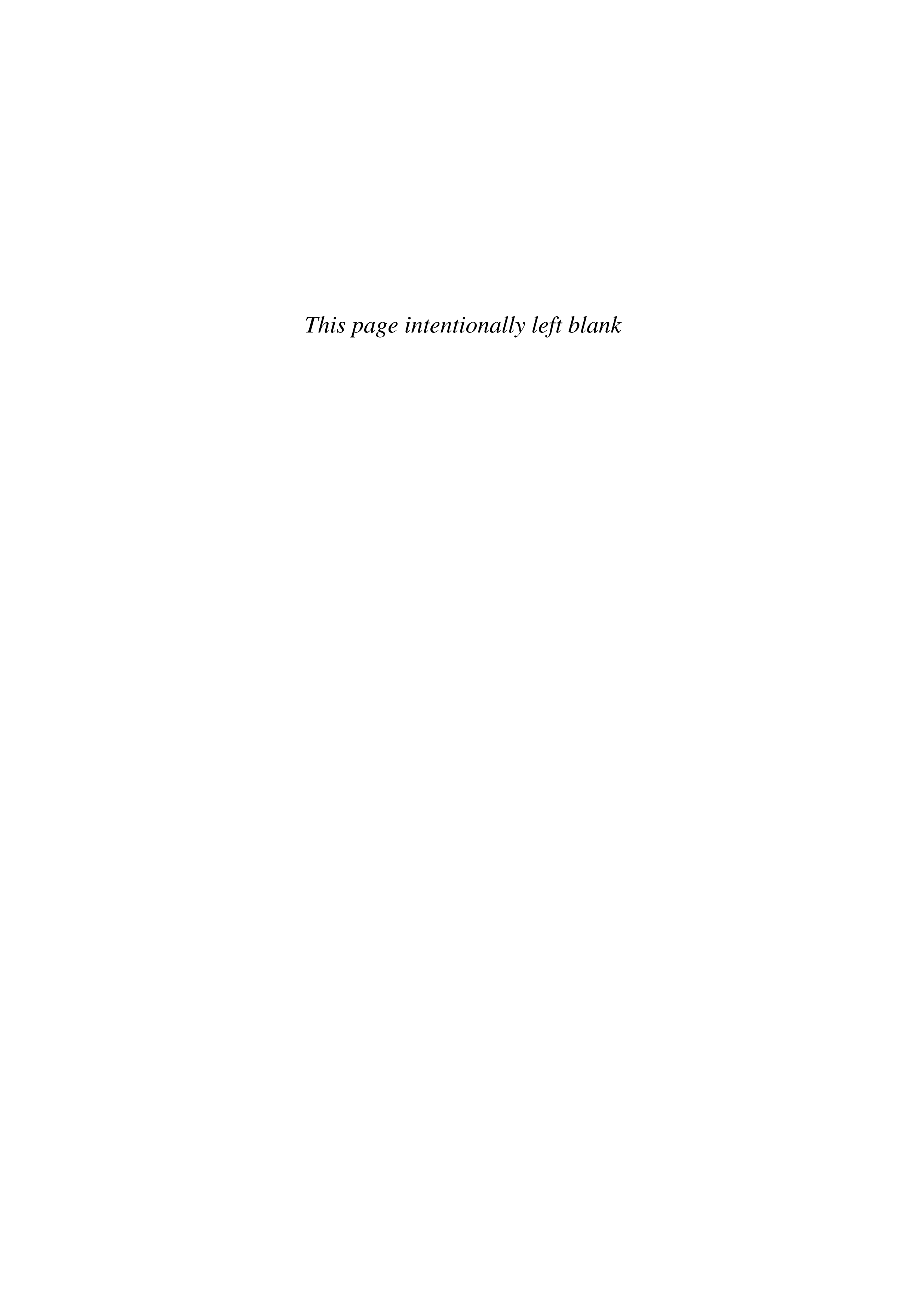
Let's review the changes you've made. Try to relate the sample source statements to the result (for example, the output) each statement creates. If necessary, rerun the samples or look at the complete source code for this exercise on the CD-ROM. Feel free to experiment by yourself.

- You first created a new class called `TextMessage`. You removed some of the data members from `ErrorMsg` and placed them in `TextMessage`.
- Since the data members `msgText` and `msgSize` now belong to `TextMessage`, you defined a `setMsgText` method in `TextMessage` in order to set these variables.
- You placed a new method named `getTranslation` in `TextMessage`. This method examined the static variable named LANGUAGECODE.
- The `ErrorMsg` class was adjusted to inherit from this new class. Since `ErrorMsg` inherits from `TextMessage`, the public members and methods in `TextMessage` are automatically available to any class that creates an `ErrorMsg` class. In the example, HelloWorld creates an instance of `ErrorMsg` and can access these variables as if they are part of `ErrorMsg`. `ErrorMsg` and `TextMessage` share these variables.
- When you set the static variable LANGUAGECODE to F, all instances of `TextMessage` simulated a translation into French. Instances of `ErrorMsg` (which are inherited from `TextMessage`) also exhibit this behavior.
- You created a new class `PrintfileErrorMsg`. It inherits from `ErrorMsg`.

- The statement `import java.util` at the beginning of the class tells the compiler to introduce the definitions for the `java.util` classes into the `Diagnostic` class. You needed to do this to use the Java `Date` class in the code.
- You introduced a data member named `msgSize` in the class `PrintfileErrorMsg`. This variable hid the similarly named variable in the class `TextMessage` from HelloWorld, that is, when HelloWorld performed the following statement:

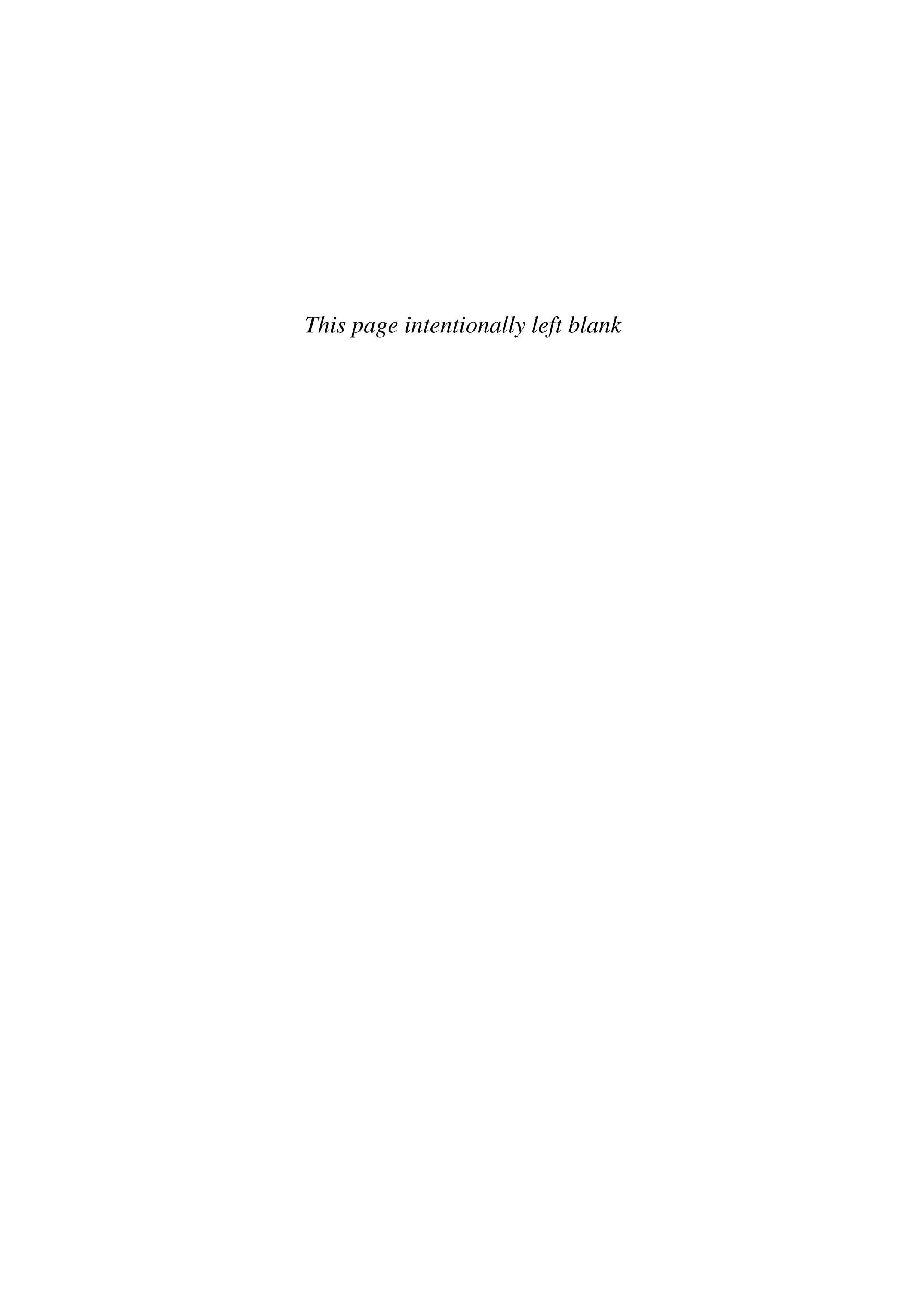
```
// Print this data member in PrintfileErrorMsg.
     System.out.println ("msgSize for PrintfileErrorMsg = " +
          myErrorMsg3.msgSize);
```

- Before the change, the variable in `TextMessage` was printed. After the change, the variable in `PrintfileErrorMsg` was printed.
- You constructed an interface definition (`WriteLineInterface`) and defined a supporting class (`WriteLine`). You then used this class to simplify the logic in `PrintfileErrorMsg`. In a real system, any other class could also use the `WriteLine` infrastructure in order to centrally manage printing functions.
- You demonstrated how polymorphism can be used. You sent the same message (`setErrorMsg (String)`) to instances of three different classes (`ErrorMsg`, `PrintfileErrorMsg`, and `Diagnostic`). Each of these classes respond to the message in a unique way:
  - `ErrorMsg` simply stored the `String` and did not print any text.
  - `PrintfileErrorMsg` printed the message after skipping some lines and centering the `String` in the print line. `PrintfileErrorMsg` also printed the values in the two `msgSize` variables.
  - `Diagnostic` printed the message and then performed a standard Java function to print the current call stack. This function would be very useful in a production system for recording contextual information appropriate for "postmortem" analysis of application or system failures.
- You used a new object in the `System` class named `err`. This object is similar to the `System.out` object you have been using all along, except it is designed to write error messages instead of standard, or informative messages. Both objects will write to the display device by default.



This page intentionally left blank

Part II Java's Syntax


This page intentionally left blank

6 Java Syntax

In This Chapter

- COBOL vs. Java Syntax
- Java Statements
- Java Comments
- Java Operators
- Binary Arithmetic Operations
- Understanding Reference Variables with COBOL
- Exercises: Java's Syntax
- Reviewing the Exercises

Until now, I have concentrated on the object-oriented concepts instead of the Java syntax. I have already introduced some of Java's syntax by way of previous examples. This section will present a more complete definition of Java's syntax.

Java's syntax was deliberately based on C++, which was in turn based on C. Therefore, COBOL programmers sometimes have a harder time understanding Java's syntax than do C and C++ programmers. Harder is not the same as impossible, however, so I expect you to follow along.

COBOL vs. Java Syntax

COBOL's syntax has the benefit of being very simple and somewhat like English. As a result, COBOL programs tend to be longer than programs written in other languages but are often more readable. Significantly, COBOL programmers tend to be pretty good typists.

In contrast, Java's syntax is much more concise. For example, a single statement can contain many expressions. Another good example of this contrast is the assignment function. In COBOL, the syntax is the wordy (but very clear) MOVE xxx TO yyy. In Java, it is simply yyy = xxx.

Both languages are not, strictly speaking, line oriented. They share the concept of a statement that may span more than one line. Most statements in COBOL can end with a period (.). In Java, all statements *must* end in a semicolon (;).

COBOL uses verbs such as IF, ELSE, and END-IF, both to test a condition and to group statements that should be performed together as a result of that condition. Java uses the first two verbs (`if` and `else`) to test conditions but uses the curly braces (`{}`) to group conditional statements.

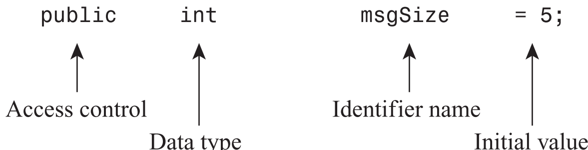
Finally, Java's syntax encourages the liberal use of objects in a program, whereas performing COBOL subroutine calls and using the results properly can be a little confusing. As a result, Java is much better suited to code-reuse strategies.

But let's not get ahead of ourselves. Instead, I'll start at the very beginning.

JAVA STATEMENTS

A Java *statement* is the equivalent of a *sentence* in COBOL. A statement is the smallest complete building block in a Java program. A statement can define a variable, perform some logic, or define an interface. Every statement must end in a semicolon.

The simplest statement type is the sort that defines a variable. The syntax for defining a variable is as follows:



I've already reviewed Java's access control options and the data types that are available. I've also discussed how variables can be assigned initial values when they are first instantiated.

An *identifier* is any named program component (such as variables, class names, object names, or class members). Similar to a COBOL name, an identifier can be made up from any alphanumeric characters but must start with an alphabetic character. There are a few differences, some as part of the language definition and some that are simply conventions.

- Java compilers are case sensitive. The identifier `MsgSize` is not the same as `msgSize`.
- The dash (-) character is not valid in a variable name in many contexts and, therefore, is never used in a Java identifier name. By convention, the underscore (_) character is sometimes used to separate the parts of an identifier name; mixed case can be used for the same purpose.

- By convention, most identifiers are mixed case, and instance identifiers (variables, objects, members, and so on) begin with an initial lowercase. User-defined types, such as classes, often begin with an initial uppercase. Static class variables and members are normally all uppercase.
- Valid names cannot start with a numeric character.

Here are some examples of valid identifier names, and what they represent, by convention (based on Sun's coding conventions found at http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html):

```
msgSize              // an instance identifier
m_ErrorMsg           // an instance identifier, probably an object identifier
ErrorMsg             // a user-defined type, such as a class name
NO_TEXT_FOUND        // a static CLASS variable
```

Here are some invalid identifier names:

```
1instanceVariable            // cannot start with nonalpha
another instance variable    // embedded spaces
COBOL-STYLE                  // embedded dashes
public                       // reserved word
```

A Java executable statement performs some part of a class's algorithm (it is similar to a sentence in COBOL's procedure division). These statements can contain multiple expressions, all of which will be performed as part of the statement.

This is a simple Java expression that declares a `String` type object:

```
String inputMsg
```

Placing a semicolon at the end makes it a complete Java statement:

```
String inputMsg;
```

This is a Java expression that performs the method `getErrorMsg()` in the object `myErrorMsg` and passes no parameters:

```
myErrorMsg.getErrorMsg ()
```

This expression can be combined with an assignment expression in order to return the result into `inputMsg`:

```
inputMsg = myErrorMsg.getErrorMsg ()
```

Placing a semicolon at the end makes it a complete Java statement:

```
inputMsg = myErrorMsg.getErrorMsg ();
```

This is a Java statement that compares `inputMsg` to another `String` and performs some logic. Since `inputMsg` is a `String`, it has a method called `equals()`, which accepts a `String` parameter and returns a boolean `true` or `false`. The `if` construct tests the boolean result of this method and performs the statements in the braces if the result is true:

```
if (inputMsg.equals ("Any Text")) {
    ...
}
```

Expressions can be combined into more complex Java statements:

```
String inputMsg = myErrorMsg.getErrorMsg ();
```

or this statement:

```
if (myErrorMsg.msgText.equals ("Any Text")) {
    ...
}
```

Java Comments

Java programmers are expected to place comments in the code. Imagine that! To support this, Java allows for the following comment styles:

Style	Format	Comments
C comments	/* . . . */	Can span multiple lines
C++ comments	//	Stops at end of line, avoids some common errors in C
Javadoc comments	/** . . . */	Used to autogenerate external documentation

In the examples so far, you have used the C++ style comments, since they are line oriented, like COBOL comments:

```
// This member is available to any class.
     public String msgText;
```

The C style comments are also sometimes used, especially for multiline comments, but here is a word of caution: This style can cause code to be inadvertently commented out.

```
/* This member is available to any class.
Its access modifier is public */
     public String msgText;
/* All these lines are comment lines.
     public String msgText;
  Let's hope your editing environment will make this obvious */
```

I will discuss Javadoc comments in Chapter 12.

Examples in this book generally use the two forward slashes (//) style of comments declaration.

Java Operators

Java provides for the usual arithmetic assignment operators. Most of these are similar to the operators available in COBOL's COMPUTE statement.

*	Multiplication
/	Division
%	Modulo
+	Addition
–	Subtraction
=	Assignment

As you would expect, there are precedence standards, and parentheses can be used to override or clarify any precedence conventions.

```
     x = 2 + 7 * 3;          // x = 23
     x = (2 + 7) * 3;        // x = 27
```

The order of precedence is important, but if you have to think about it, you are either writing or reading code that will be hard to maintain. Rather than rely on the order of precedence conventions, always make your intentions explicit by using parentheses.

Unlike COBOL, Java always uses arithmetic symbols instead of words for its math operations. For example, the assignment operator is an equals sign (=) instead of MOVE, or x – y instead of COBOL's SUBTRACT Y FROM X. Math operations in Java always use a syntax similar to COBOL's COMPUTE verb.

BINARY ARITHMETIC OPERATIONS

Binary operations (that is, operations involving two operators) come in several types: arithmetic, conditional, relational, assignment, and bitwise. You will explore the various arithmetic operations here.

Integer operations always create a result type of `int`, unless one of the operands is of type `long`, in which case the result type is `long`. If x in the preceding example had been defined as a `short`, then you would have had to cast the result into a `short` in order to avoid a compiler error.

```
short x;
x = (short) (2 + 7 * 3);          // x = 23
```

Likewise, floating-point operations always create a result of type `double` unless both operands are of type `float`. Remember that floating-point constants by default are of type `double`. Further, the compiler can automatically convert either an integer or a float into a double, but converting a double into either of these requires a cast. Similarly, integers will automatically convert to floats, but floats won't convert to integers unless you cast them.

```
int i;
float f;
double d;

d = f + 1.2;                    // OK
f = f + 1.2;                    // not good, cast required
f = (float) (f + 1.2);          // OK
f = f + 1.2F;                   // also OK
i = (int) f;                    // OK, but possible precision loss
f = (float) (d + 1.2);          // OK, but possible precision loss
d = f + i;                      // OK
```

These conventions for managing numbers are easy to remember if you observe this logic:

- For integer math, the default type is `int`.
- For floating-point math, the default type is `double`.
- Smaller types can automatically be converted into larger types.
- Larger types must be cast into smaller types. There is the possibility of data loss.
- The size order is `double`, `float`, `int`, `short`, `byte`.

Java coders often use a shortcut (assignment with operator) to perform simple math operations on a single variable:

```
x = x + 7;
x += 7;                     // the same statement
```

This coding style can take a little getting used to, since one of the source operands (the second x) that would be expected in an algebraic expression is missing. The addition operator (+, in the example) is normally placed to the left of the = sign in this type of complex assignment expression. Each of the math operators (*, /, +, and –) can be combined with an assignment expression (e.g., *=, /=, +=, and –=).

Like C and C++, Java provides for increment and decrement operators (`++` and `--`). These provide a convenient mechanism to increment or decrement a variable.

```
x = 7;
x++;                    // x = 8
x--;                    // x = 7
```

These operators come in two types: the postfix (x++) and the prefix (++x). The difference shows up when the auto increment operators are used in an expression. The prefix operator increments or decrements before presenting the value, and the postfix increments or decrements after presenting the value.

```
x = 7;
y = 2 * x++;          // y = 14, and x = 8
x = 7;
y = 2 * ++x;          // y = 16, and x = 8
```

As you can see from the examples, there are subtle differences between postfix and prefix increment operators, especially when used as part of an expression. A number of Java language authorities recommend that these operators not be used as part of an expression, but only as standalone statements in order to avoid confusion on this matter.

```
x = 7;
x++;                        // An increment statement with no ambiguity
y = 2 * x;                  // y = 16, and x = 8
```

Table 6.1 lists the operators available in the Java language in the order in which they are evaluated by the compiler.

TABLE 1.2 RESPONSIBILITIES OF CSIRT DURING THE INCIDENT RESPONSE PROCESS

Operator	Description	COBOL Equivalent
.	Member selection	
[ ]	Array subscript	( )
( )	Expression grouping or	( ) as used by expression grouping IF function call ((A = B) AND (X = Y))
++, - -	Autoincrement/decrement	PERFORM VARYING
*, /, %	Multiply, divide, remainder	COMPUTE or MULTIPLY, DIVIDE GIVING
+, -,	Addition, subtraction	COMPUTE or ADD, SUBTRACT GIVING
<<, >>, >>>	Bitwise shift operations*	
<=, <, >=, >	Less than or equal to, less than, greater than or equal to, greater than	NOT >, <, NOT <, >
= =, !=	Equality test, inequality test	= , NOT =
&, |, ^, ~	Bitwise AND, OR, XOR, NOT	
&&, ||, !	Logical AND, OR, NOT	AND, OR, NOT
=	Assignment	MOVE
*=, /=, %=, +=, -=, &=	Complex assignment	COMPUTE

**Java includes a number of bitwise operations and makes some important distinctions between these operations and boolean logical operations. Since COBOL does not have bitwise operations and most business operations do not require them, the reader is encouraged to explore this concept in other sources.*

The special case of assignment and equality as applied to object reference variables warrants attention. An object reference variable is just like any other variable in many ways. But remember that it contains a reference to the object and does not contain the object's values. Therefore, although two different object reference variables can point to different objects with equal values, they are not the *same* object. The only way that two object reference variables can be equal is if they point to the same object.

Some examples may help clarify this. Recall that the `String` data type is not really an intrinsic type, but rather, is a class. Therefore, `String` variable names are actually object reference variables. Consider this example:

```
String s1 = new String ("Some text");
```

Often, the following syntax is used instead of the previous declaration:

```
String s1 = "Some text";
```

Suppose the program also includes this statement:

```
String s2 = "Some other text";
```

Clearly, `s1` and `s2` are not equivalent, so this test fails, as you would expect.

```
if (s1 == s2) {
      ...                // This code would not execute
}
```

In addition, the following test will also fail, since `s1` and `s2` do not contain equivalent text:

```
if (s1.equals (s2)) {
      ...                // This code will not execute
}
```

Now, suppose you have two `Strings` that contain the same text.

```
String s1 = new String ("Some text");
String s2 = new String ("Some text");
```

```
if (s1 == s2) {
                                // This code will not execute
                                // even though the text contained in these
                                // two Strings are equal. s1 and s2 will
                                // point to different String objects
}
```

Yet the following test always succeeds since the `equals()` method in the `String` class compares the value of the `String` object with the passed `String`:

```
if (s1.equals (s2)) {
     ...                // This code will execute
}
```

What is the bottom line? Always use the `equals()` or `compareTo()` methods when you want to compare two `Strings`. Never assume that the `String` reference variables will compare appropriately, based on the text values in the `String`.

Get ready for the real weirdness. Object reference variables are variables and, therefore, can be assigned values. Normally, this happened only with the `new` keyword, as in the following:

```
String s1 = new String ("Some text");
String s2 = new String ("Some other text");
```

But it is perfectly legal to set object reference `s2` equal to `s1`:

```
s2 = s1;
```

Now, both `s2` and `s1` point to the same object. (As a matter of fact, the object formerly pointed to by `s2` could be unreferenced and will likely be garbage collected or deleted by the system at some point.) This time, the equality test, when applied to the values of the `Strings`, will succeed even though the initial object pointed to by `s2` was different from `s1`:

```
if (s1.equals (s2)) {
            ...         // This code would execute
}
```

The same is true of this test, which compares the object references:

```
if (s1 == s2) {
```

```
                ...         // This code would execute since s1 and s2
                ...         // point to the same object.
        }
```

The only way to modify the text associated with `Strings s1` or `s2` is to point these reference variables to different `String` objects. This is normally done by performing some method that returns a `String` object and assigning the reference variable to that `String`. `String` objects themselves are immutable. That means `String` objects themselves cannot be changed; only the reference variables that point to a `String` can be changed. For example, if you need another `String` value in `s1`, you must create a new `String`, and point `s1` to it.

```
     String s1 = new String ("Some text");
     String s2 = new String ("Some other text");
// Assign s2 to a new String object.
// This object is the result of the trim() method, which removes leading
//  and trailing spaces.
// String s1 contains no leading or trailing spaces, so the String object
//  returned from trim() will contain the same text as String s1.
     s2 = s1.trim();
     if (s1.equals (s2){
// This code would execute since s1 and s2 contain the same text.
     }
```

Understanding Reference Variables with COBOL

It may be useful to revisit the COBOL example to help explain this concept. Recall that the CALLER program defined two instances of MYSUB-CONTROL, as follows:

```
01 MYSUB1-CONTROL.
      03   MYSUB1-ACTION-SWITCH              PIC X.
           88 MYSUB1-ACTION-INITIALIZE        VALUE "I".
           88 MYSUB1-ACTION-EVALUATE          VALUE "E".
           88 MYSUB1-ACTION-SET-AND-EVALUATE VALUE "S".
           88 MYSUB1-ACTION-GET-CALL-COUNTER VALUE "G".
      03   MSG-TEXT                          PIC X(20).
      03   MSG-SIZE                          PIC 9(8).
      03   MYSUB1-RETURNED-CALL-COUNTER      PIC 9(10).
      03   MYSUB1-PRIVATE-ITEMS              PIC X(20).
```

```
01 MYSUB2-CONTROL.
      03  MYSUB2-ACTION-SWITCH             PIC X.
          88 MYSUB2-ACTION-INITIALIZE       VALUE "I".
          88 MYSUB2-ACTION-EVALUATE         VALUE "E".
          88 MYSUB2-ACTION-SET-AND-EVALUATE VALUE "S".
          88 MYSUB2-ACTION-GET-CALL-COUNTER VALUE "G".
      03  MSG-TEXT                         PIC X(20).
      03  MSG-SIZE                         PIC 9(8).
      03  MYSUB2-RETURNED-CALL-COUNTER     PIC 9(10).
      03  MYSUB2-PRIVATE-ITEMS             PIC X(20).
```

Suppose that the MSG-TEXT items in both MYSUBx-CONTROL areas contained the same text ("Some Text," for example). In this case, this COBOL code would display a message:

```
IF MSG-TEXT OF MYSUB1-CONTROL = MSG-TEXT OF MYSUB2-CONTROL
     DISPLAY "MSG-TEXT's are equal".
```

However, this COBOL code may or may not display a message, depending on the value of the other items in the MYSUBx-CONTROL areas:

```
IF MYSUB1-CONTROL = MYSUB2-CONTROL
     DISPLAY "MYSUBx-CONTROL's are equal".
```

When you compare object reference variables (including a `String` object reference variable) with the == Java operator, it is as if you are comparing two CONTROL areas in COBOL. In order to compare the MSG-TEXT items in the CONTROL areas, you must explicitly compare just those items. Likewise, in order to compare text in a Java `String`, you must use the `equals()` method and not compare the object reference variables.

EXERCISES: JAVA'S SYNTAX

It's time to visit the example classes again and try out all these new ideas.

1. Edit the HelloWorld.java source file in your java4cobol directory with a text editor. You'll start by deleting the code that experimented with inheritance. Remove the lines after this statement (but remember to leave in the two curly braces at the end of the program):

```
// Print the contents of ErrorMsg's String data member directly.
        System.out.println (myErrorMsg.msgText);
```

2. Add these Java expressions at the end of the previous statement (before the last two curly braces):

```
// Experiment with Java statements.
        String testMsg
        myErrorMsg.getErrorMsg ()
```

3. Save these modifications as a text file, and then compile the class in the DOS command window:

```
➔ javac HelloWorld.java
```

   You should get an error message indicating that a semicolon is missing.

   Add a semicolon to the end of each of these expressions. Compile this class again. It should now compile successfully.

4. These two statements simply define a `String` variable, and execute the `getErrorMsg()` method in the `ErrorMsg` class. As written, they are not very useful, since you end up with an empty `String` variable, and the result of this effort is lost. Add this additional **bolded** Java code to the statements, making them more useful:

```
// Experiment with Java statements.
        String testMsg;
        testMsg = myErrorMsg.getErrorMsg ();
```

   Compile this class again.

5. Combine these two statements into one, as follows:

```
// Experiment with Java statements.
        String testMsg = myErrorMsg.getErrorMsg ();
```

   Compile this class again.

6. Next, you'll adjust the original statement and make it more complex. Although you are using the `if` statement in this code (and I won't explore it in detail until the next chapter), you should be able to follow it fairly easily. Add this additional **bolded** Java code:

```
// Experiment with Java statements.
          String testMsg = myErrorMsg.getErrorMsg ();
          if (myErrorMsg.getErrorMsg ().equals (testMsg)) {
             System.out.println ("testMsg = ErrorMsg text");
          }
```

7. Save, compile, and rerun the HelloWorld application.

```
➔ javac HelloWorld.java
➔ java HelloWorld
```

The output should look like this:

```
...
testMsg = ErrorMsg text
```

8. Next, add this additional **bolded** Java code:

```
// Experiment with Java statements.
          String testMsg = myErrorMsg.getErrorMsg ();
          if (myErrorMsg.getErrorMsg ().equals (testMsg)) {
// Define a temporary integer variable.
        int i = 5;
          System.out.println ("testMsg = ErrorMsg text");
              System.out.println ("i = " + i);
           }
```

9. Compile and rerun the HelloWorld application. The output should look like this:

```
...
testMsg = ErrorMsg text
i = 5
```

10. Next, you'll explore how parentheses can modify the results of a statement. Using a text editor, add these **bolded** lines to the end of your HelloWorld.java source file.

```
          int x, y, z;
          x = 3;
          y = 4;

          z = x + 1 * 2;
          System.out.println ("z = " + z);
```

Compile and rerun the HelloWorld application. The output should look like this:

```
...
testMsg = ErrorMsg.text
i = 5
z = 5
```

11. Add the parentheses as indicated here.

```
      z = (x + 1) * 2;
      System.out.println ("z = " + z);
```

Compile and rerun the HelloWorld application. As you would expect, the value of z is different. The output should look like this:

```
testMsg = ErrorMsg text
i = 5
z = 8
```

12. You'll explore how to use the Java operators to control evaluations. Add these **bold** lines to the end of your HelloWorld java source file:

```
// Experiment with operators
          if (((x == y) || (z < x)) && ((z != y) || (x >= 1))) {
               System.out.println ("Condition is true");
          }
          else {
               System.out.println ("Condition is not true");
          }
```

Compile and rerun the HelloWorld application. The output should look like this:

```
...
testMsg = ErrorMsg text
i = 5
z = 8
Condition is not true
```

Compare the previous Java statements to this COBOL sentence:

```
IF  (((X = Y) OR (Z < X)) AND
    ((Z NOT = Y) OR (X NOT < 1)))
    DISPLAY "Condition is true"
ELSE
    DISPLAY "Condition is not true".
```

Notice the liberal use of parentheses in the Java example and the use of curly braces around the `if` and `else` code blocks, even though these constructs may not be strictly required.

What Java operator can you reverse in order to make the "Condition is true" message appear?

13. I'll experiment a bit with additional data types. Add these **bolded** lines to the end of your HelloWorld class:

```
// Experiment with additional data types.
// These data types are automatically converted to the double data type
// during the comparison.
        double d = 4;
        float f = 4;
        short s = 4;
        if ((d == y) && (d == f) && (d == s)) {
             System.out.println ("Condition is true");
        }
        else {
             System.out.println ("Condition is not true");
        }
```

Compile and rerun the HelloWorld application. The output should look like this:

```
...
testMsg = ErrorMsg text
i = 5
z = 8
Condition is not true
Condition is true
```

14. Try a test involving overflow of integer values. Add these **bolded** lines to the end of your HelloWorld class:

```
// Experiment with overflow.
        d *= 536870912;
        y *= 536870912;
        if (d == y) {
            System.out.println ("Condition is true");
        }
        else {
            System.out.println ("Condition is not true. d = " + d
                + " y = " + y);
        }
```

Compile and rerun the HelloWorld application. The output should look like this:

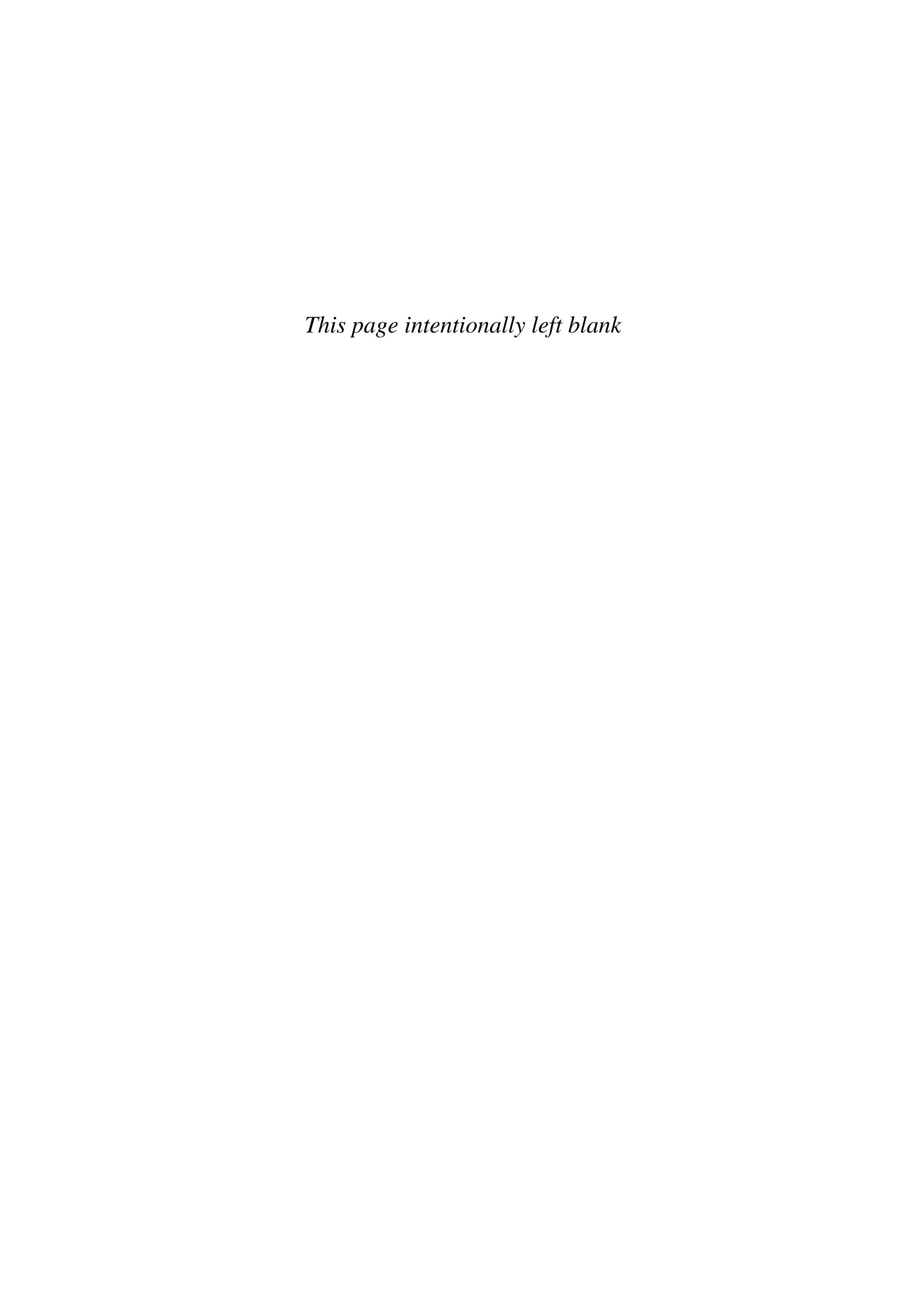
```
...
testMsg = ErrorMsg text
i = 5
z = 8
Condition is not true
Condition is true
Condition is not true. d = 2.147483648E9 y = -2147483648
```

Java specifies that integer multiplication will not cause an error condition (exception) even if overflow occurs, so be careful with large integer numbers. Notice the way `d` is printed. Java double numbers are stored in IEEE 754 format, and this is a `String` representation of this format.

Reviewing the Exercises

Let's review the samples you've created. Try to relate the sample source statements to the result (that is, the output) each statement creates. If necessary, rerun the samples, or look at the complete source code for this exercise on the CD-ROM. Feel free to experiment by yourself.

- Java statements must end with a semicolon. A single statement can contain several Java expressions.
- Java uses parentheses to group expressions in much the same way that COBOL does. Java's AND, OR, and NOT operators (&&, ||, !) work in much the same way as do their COBOL counterparts.


This page intentionally left blank

7 Flow Control

In This Chapter

- Code Block
- The `if` Statement
- The `while` Statement
- The `do...while` Statement
- The `for` Statement
- The `switch` Statement
- The `break, continue` Statements
- Exercises: Flow Control
- Reviewing the Exercises

As with any language, Java defines a set of expressions and rules. The programmer uses the expressions and follows the rules to define the sequence of steps to be performed by the program. In this chapter, I'll examine the way a Java program manages flow control.

Java borrows heavily from C++, which has borrowed heavily from C. Nowhere is this relationship more evident than in Java's syntactic definitions, especially the program flow control operators.

Code Block

A block of code is a set of Java statements grouped by curly braces {}. Blocks can also have other blocks nested inside them.

```
        {this is a statement in a block of code;
         this is another statement in the same block of code;

              {this is a statement in a block nested in the first;
               this is another statement in a nested block;
               }
    // This is the end of the nested code block.

        }
    // This is the end of the first code block.
```

COBOL organizes statements through the use of verbs such as `if`, `else`, and `end...if`. The end-of-sentence character (period) also defines the end of any conditional group of statements. In Java, multiple statements that are executed as a result of some flow control expression (such as `if` and `else`) must be grouped into a code block using the braces.

However, the addition of another statement to the code block would require that these two statements be grouped with braces. Therefore, it is good practice to place braces around single-statement code blocks, especially if they are the result of some conditional expression.

By convention, the statements in a code block are indented to the same column. This helps to visually organize the statement groups and improves readability. (Sun defines a set of code conventions at http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html.)

In addition, it is good programming practice to build your code blocks (that is, create the `if` and `else` sections, including the braces) before adding the statements inside the code block. This allows you to concentrate on the two steps independently and, therefore, reduces the number of compile errors and runtime bugs created by syntax mistakes.

The if Statement

```
    Structure: if (condition) { block }
        else {block} ;
```

This basic control flow statement will evaluate `condition`, and if `true`, the first block of code will execute; if `false`, the second block will be performed. As is the case in COBOL, the second code block (the `else` condition) is optional.

```
if (inputMsg.equals ("Any Text")) {
     ...
}
else {
     ...
}
```

`condition` can be any Java expression that returns a boolean. The result of that expression is evaluated.

```
if (myErrorMsg.msgText.equals ("Any Text")) {
     ...
}
```

`condition` must be a *boolean* expression, that is, one that evaluates to either true or false. In the example, the `equals()` method returns a true or false boolean, so it is a good candidate for inclusion in an `if` statement.

The expression `myErrorMsg.msgText.equals` *contains two member operators (that is, two periods). This implies that two members (a data member and a method member in this case) will be accessed.*

Evaluate the expression this way: The data member `msgText` *in the object* `myErrorMsg` *is a* `String` *variable. As a result, it contains a method named* `equals()`*. This method accepts a* `String` *parameter and returns a boolean result of true if the passed* `String` *parameter contains the same text as is contained in this* `String`*. The boolean result is evaluated by the* `if` *operator.*

Multiple expressions can be grouped, and the result of such grouping will be evaluated. The logical operators AND (&&), OR(||), and NOT (!) group boolean expressions in the same way their text equivalents do in COBOL. For example, this statement:

```
if (myErrorMsg.msgText.equals ("Any Text")  ||
   (myErrorMsg.msgText.equals ("Some Text")) {
     ...                // The IF code block
}
```

will perform the code block if `myErrorMsg` contains either "`Any Text`" or "`Some Text.`" The following is a syntactically equivalent COBOL statement:

```
IF (MY-ERROR-MSG = "Any Text") OR
   (MY-ERROR-MSG = "Some Text")
...                   // The IF code block
END-IF.
```

Java will "short-circuit," or stop evaluating `condition`, as soon as an unambiguous result is available. Therefore, it is possible that some of the expressions in `condition` will not be executed. In the example, if `myErrorMsg` contains "`Any Text`", the `equals()` method will be performed only once.

The order of evaluation of expressions can be controlled using parentheses (). Just as in COBOL, you should liberally use them to clarify your intentions, even if the compiler does figure it out correctly.

```
     if (myErrorMsg.msgText.equals ("Any Text") &&
          myErrorMsg.msgSize == 8 ||
          myErrorMsg.msgText.equals ("Some Text") {
// This code block will be performed only when the text equals "Any Text"
// since the length of "Some Text" is 9 (that is, it contains 9
// characters).
         ...
}
```

But the intentions of the developer are unclear, or perhaps there is a bug. The proper code is more likely to be as follows:

```
     if ((myErrorMsg.msgText.equals ("Any Text") &&
           myErrorMsg.msgSize == 8) ||
          myErrorMsg.msgText.equals ("Some Text")) {
// This code block will be performed for both "Any Text" and "Some Text."
          ...
}
```

This is a good time to talk some more about local variable scope. Local variables are variables that have been defined somewhere in a Java code block:

```
     if (inputMsg.equals ("Any Text")) {
         ...
// This next statement defines a local variable inputMsgSize and sets it to
// msgSize.
```

```
        int inputMsgSize = myErrorMsg.msgSize;
        ...
      }
    else {
        ...
    }
```

The variable `inputMsgSize` is valid only in the code block in which it has been defined. In the example, that would be the first code block (that is, the statements executed if `inputMsg` is equal to `"Any Text"`). Note that these statements are bounded by the first pair of braces.

This variable is also valid in any *inner* code blocks. Therefore, a statement that sets `inputMsgSize` to –1 in an inner code block is valid:

```
    if (inputMsg.equals ("Any Text")) {
        ...
// The next statement defines a local variable inputMsgSize and sets it to
// msgSize.
        int inputMsgSize = myErrorMsg.msgSize;
        if (inputMsgSize == 0) {
// The next statement resets the local variable inputMsgSize.
            inputMsgSize = -1;
            ...
          }
        ...
     }
     else {
```

Outside the braces for the first code block (for example, in the `else` condition code block), the variable `inputMsg` is not valid, and references to it would cause a compile time error:

```
    if (inputMsg.equals ("Any Text")) {
    ...
// The next statement defines a local variable inputMsgSize and sets it to
// msgSize.
         int inputMsgSize = myErrorMsg.msgSize;
         ...
    }
    else {
// The next statement refers to inputMsgSize outside its code block
// and would cause a compile time error.
```

```
          if (inputMsgSize == 0) {
          }
          ...
     }
//  The next statement refers to inputMsgSize outside the whole if statement
//  and would be invalid as well.
     if (inputMsgSize == 0) {
     ...
     }
```

Note that local variables cannot be defined *inside* a code block and later evaluated *outside* it. This sort of result state variable must be defined outside the code block if it is to be used outside the code block.

If an inner code block were to attempt to define a new local variable with the same name as an existing variable (`inputMsgSize`, in the example), then the Java compiler would detect that as a name collision and report an error. This is, by the way, a departure from some C and C++ compilers, where local variables can be named the same as existing variables—a source of more than a few bugs.

Finally, a local variable named `inputMsgSize` could be defined in some other code block and used inside that code block. This often happens with temporary variables and counters, such as the commonly used x. Java's tight scoping rules should allow the compiler to catch most instances of inappropriate use (and reuse) of local variables.

THE WHILE STATEMENT

```
Structure:    while (condition) { block } ;
```

This basic loop control statement will evaluate `condition` and, if true, will perform the block of code. Let's hope that some statement in the block of code will eventually cause `condition` to not be true, or else you would have an endless loop!

```
// Assume errorMsgs is an array of ErrorMsg objects that has previously
//  been created
// ARRAY_SIZE is the maximum number of elements allowed in ErrorMsg.
 //  Examine each ErrorMsg object to find the first with a size equal to 0.
//  Define x outside the while code block.
     int x = 0;
     while (x < ARRAY_SIZE) {
```

```
            int inputMsgSize = errorMsgs[x].msgSize;
            if (inputMsgSize == 0) {
// Exit the loop immediately. x will point to this element.
                    break;      // The break statement causes the loop to
                                // exit. The statement after the loop's
                                // ending brace will be the next statement
                                // executed.
           }
           x++;         // Increment the loop variable.
           continue;    // Continue the loop. This statement is not required
                        // but is shown here to show how continue might be
                        // used. The continue statement causes the loop to
                        // proceed with the next iteration.
      }
// The loop has completed.
// At this point either you have found an element that contains a 0 size,
//  or you have examined all of the objects in ErrorMsgs. Test x to see if
//  you tested all of the items in ErrorMsgs (that is, is x equal to
//  ARRAY_SIZE?).
      if (x != ARRAY_SIZE) {
// You have an ErrorMsg with a 0 size !
           ...
      }
      else {
// You have none. The array was exhausted
           ...
      }
```

As in any programming language, Java loops are often a mechanism to manipulate an array of similar items. I discussed arrays in Chapter 4, and I will discuss the more powerful collection processing in detail in Chapter 11, but a word of review about arrays and array processing in Java seems appropriate at this time.

In COBOL, subscripts for arrays (that is, items that OCCUR x TIMES) start with 1. That means the first item in an array is referenced as ITEM (1), and the last item is referenced as ITEM (x).

For example, the statement:

```
01 MY-ITEMS    PIC X(2) OCCURS 10 TIMES.
```

defines an array called MY-ITEMS. Each item in the array is 2 bytes, and there are 10 items in the array. Any subscript in the range of 1 through 10 is valid:

```
IF (MY-ITEMS(1) = "AA")
    OR (MY-ITEMS(10) = "XX")
```

However, subscripts outside this range are not valid and will generate either a compile error or a runtime error. Both of these statements will generate an error:

```
IF (MY-ITEMS(0) = "AA")
    OR (MY-ITEMS(11) = "XX")
```

In contrast, Java follows the conventions of C and defines the first item in an array as item 0. Therefore, many loops in Java start with a loop variable equal to 0 and end when that variable is equal to the number of items in the array. This means that item[1] is actually the second item in the array, and item[ARRAY_SIZE] is not a valid reference. Instead of the parentheses subscript identifiers that COBOL uses, subscripts in Java are identified with brackets [ ].

The do...while Statement

```
Structure:   do { block } while (condition) ;
```

This loop control statement is very similar to the basic `while` statement, except that `block` will always be executed at least once. `condition` is evaluated after the code block is performed, and if true, the code block is reiterated.

```
// Assume errorMsgs is an array of ErrorMsg objects that has previously
// been created.
// ARRAY_SIZE is the maximum number of elements allowed in ErrorMsg.
// Examine each ErrorMsg object to find the first with a size equal to 0.
// Define x and inputMsgSize outside the while code block.
     int inputMsgSize = 0;
     int x = 0;

     do {
           inputMsgSize = errorMsgs[x].msgSize;
           if (inputMsgSize == 0) {
// Exit the loop immediately. x will point to this element.
               break;
```

```
        }
        else {
            x++;
        }
    }
    while (x < ARRAY_SIZE);
```

In this example, `inputMsgSize` will be set at least once. Since it has been defined outside the `do...while` loop, it will be valid outside the loop and will always contain some value as set in the loop.

The for Statement

```
Structure:   for (expression1; (condition); expression2) { block }
```

This loop control statement is commonly used to support iteration. The first expression is the counter initialization and is performed only at the start of the loop. The condition is tested to determine if the loop should continue, and if true, the code block is performed. The second expression is the expression that changes the counter. It is performed after the code block is completed, but before the next iteration of the loop. The code block will be performed iteratively as long as `condition` is true. After every iteration of the code block, and if `condition` is true, the counter increment expression (`expression2`) will be performed.

```
    for (int x = 0; x < ARRAY_SIZE ; x++) {
          int inputMsgSize = errorMsgs[x].msgSize;
          if (inputMsgSize == 0) {
// Exit the loop immediately. x will point to this element.
              break;
          }
    }
```

Note that the sample is not of much use, since both `x` and `inputMsgSize` have been defined inside the loop and are, therefore, not available outside it (`x` was actually implicitly defined in the `for` statement). If you tried to use either `x` or `inputMsgSize` outside the loop, you would create a compiler error.

It would be necessary to define these variables before the loop if you want to use them after it is complete.

```
// Define x and inputMsgSize outside the while code block.
     int inputMsgSize = 0;
     int x = 0;

     for (x = 0; x < ARRAY_SIZE ; x++) {
           inputMsgSize = errorMsgs[x].msgSize;
           ...   // as before
     }
```

You may have noticed the similarities between the `for` loop and the `while` loop. The `for` loop is a combination of the most common structures used in a `while` loop. For example, this is a common `while` loop structure.

```
initializationExpressionA;
while (testExpressionB) {
     performCode;
     incrementExpressionC;
}
```

This is combined in a `for` loop as follows:

```
for (initializationExpressionA; testExpressionB;
     incrementExpressionC;) {
     performCode;
}
```

The switch Statement

```
Structure:  switch (expression) {

case (statement1):
        code block1 ;
        break;

case (statement2):
        code block2 ;
        break;

default:
        code block3 ;
        }
```

This flow control statement is commonly used to perform various functions based on the value in a variable. It is similar to COBOL's EVALUATE verb. `expression` is evaluated and then compared for equality to each of the `case` statements. If `expression` is equal to the `case` statement, the code block for that statement is performed.

```
int inputMsgSize = 0;
int x = 0;

for (x = 0; x < ARRAY_SIZE ; x++) {
      switch ( errorMsgs[x].msgSize)
      {
      case 0:
         errorMsgs[x].setErrorMsg("Default Text");
         break;
     case 1:
     case 2:
     case 3:
        errorMsgs[x].setErrorMsg("Text < 4 chars");
        break;
     default:

     }
}
```

This statement does have a number of limitations. You can evaluate only certain primitive types (`char`, `byte`, `short`, `int`) and enumerated types. You cannot use ranges of values, as is possible with Level 88s in COBOL. And there are a few surprising little side effects, depending on whether the `break` statement is performed.

Normally, each code block is coded with a `break` in order to exit the `switch` loop. If a break is not defined, then upon completion of the case code block, the evaluation continues with the next case condition. This may or may not be what you intended, so you should always place breaks in `case` statements; and if you want evaluation to continue, make sure that you document it.

THE BREAK, CONTINUE STATEMENTS

I've introduced the statements `break` and `continue` by example rather than with a formal definition, so let's address these statements. These are statements that manage flow control in all the loop structures (`while`, `for`, `switch`, and so forth). The

`break` statement causes the current loop to exit or complete immediately. The `continue` statement causes the current loop to be reiterated from its beginning, preserving the current values. Consider the following examples:

```
      int x;
      int inputMsgSize = 0;
      for (x = 0; x < ARRAY_SIZE ; x++) {
            inputMsgSize = errorMsgs[x].msgSize;
            if (inputMsgSize == 0) {
// Exit the loop immediately. x will point to this element.
                break;
           }
      }
// Evaluate inputMsgSize after the loop.
      if (inputMsgSize == 0) {
            ...
      }
```

The `break` statement exits the current `for` loop at once. Variables modified by the loop will maintain their current values and may be available after the break statement is executed, depending on how the variables were defined. In the example, `inputMsgSize` will be set to 0 if the break was performed.

In contrast, consider this example:

```
      for (x = 0; x < ARRAY_SIZE ; x++) {
           int inputMsgSize = errorMsgs[x].msgSize;
           if (inputMsgSize == 0) {
// Continue with the next item in the loop.
               continue;
            }
// Translate the error message using the getTranslation method.
// Note that the String parameter passed to the setErrorMsg() method is the
// result of the getTranslation() method.
          errorMsgs[x].setErrorMsg ((errorMsgs[x].getTranslation()));
       }
```

The `continue` statement causes the loop to continue with the next iteration. In the example, error messages with sizes equal to 0 will not be translated. But the loop will continue until all the error messages are examined (and translated, if they contain text).

The `break` statement and the `continue` statement affect only the current loop. If loops are nested, and a break is performed, then program flow will continue with the next statement after the current loop. Conversely, `continue` will cause the current loop to be reiterated.

```
for (x = 0; x < ARRAY_SIZE ; x++) {
      int inputMsgSize = errorMsgs[x].msgSize;
      if (inputMsgSize == 0) {
// Find the next item with some text. Then move its text into this
item's text.
        for (y = x + 1; y < ARRAY_SIZE ; y++) {
            inputMsgSize = errorMsgs[y].msgSize;
            if (inputMsgSize != 0) {
               errorMsgs[x].setErrorMsg
                 (errorMsgs[y].getErrorMsg);
               break;
             }
          }
       }
// The break statement causes this statement to be processed as the
next statement.
// Translate the error message.
      errorMsgs[x].setErrorMsg ((errorMsgs[x].getTranslation()));
  }
```

In this example, the `break` statement causes the inner `for` loop to be exited. The outer loop will continue.

What happens if no error messages contain text? You would attempt to perform the translate method without any text! Let's hope the method is robust enough to deal with this condition, but suppose it isn't? The best way to code for this situation would be to break out of the outer and inner loops as soon as you discover that there is no text to translate.

Java provides a labeled `break` statement to help with this requirement. This statement allows the programmer to specify which loop should be exited. It's as close as Java gets to a `goto` statement. `goto` is not a valid Java word, but it is reserved (that is, it is not valid as a user-defined name).

Without starting any religious arguments, it is fair to say that there are situations where an explicit statement to exit a loop is a superior construct than complex and hard-to-maintain `if...break` and `if...continue` statements. Once the decision is made to exit a particular code block, it is better if the code clearly states that intention.

You can extend the example as follows:

```
// First, define a label at the beginning of your loop.
Translate_loop:

     for (x = 0; x < ARRAY_SIZE ; x++) {
           int inputMsgSize = errorMsgs[x].msgSize;
           if (inputMsgSize == 0) {
// Find the next item with some text. Then move its text into this
// item's text.
               for (y = x + 1; y < ARRAY_SIZE ; y++) {
                   inputMsgSize = errorMsgs[y].msgSize;
                   if (inputMsgSize != 0) {
                      errorMsgs[x].setErrorMsg
                        (errorMsgs[y].getErrorMsg);
                      break;
                   }
                   else {
                      if (y == (ARRAY_SIZE - 1)) {
// You are on the last item, without finding text. Exit the translate
// loop entirely.
// Do not process any more items.
                              break Translate_loop;
                          }
                       }
                  }
            }
// Translate the error message.
          errorMsgs[x].setErrorMsg
                ((errorMsgs[x].getTranslation()));
  }
// This statement will be processed as the next statement after
// break Translate_loop; is executed.
// It is also performed as the next statement after the loop exits
// normally.
     System.out.println ("The loop has completed");
```

Table 7.1 summarizes the Java flow control operators.

TABLE 7.1 JAVA FLOW CONTROL OPERATORS

Construct	Description	COBOL Equivalent
if condition…else…	Evaluate condition and perform either the first or the second code block	IF condition …ELSE …
for condition…	Iteratively perform the next code block until condition is not true	PERFORM paragraph VARYING
while condition…	Perform the next code block, until condition is not true	PERFORM paragraph UNTIL
do…while	Perform the next code block, evaluate condition, and if true, perform code block again	PERFORM paragraph UNTIL condition
switch (i) case…	Perform the appropriate code block depending on the value of (i)	EVALUATE
case…		
break	Exit the current loop	EXIT
continue	Reiterate the current loop from the beginning	
labeled break	Exit the loop with this name. The statement past the labeled loop will be the next statement executed.	GO TO

Exercises: Flow Control

Time to visit the example classes again and try out all these new ideas.

1. Edit the HelloWorld.java source file in your java4cobol directory with a text editor. Remove the lines after the `if` statement where you tested the value of `textMsg`.
2. First, you need to adjust the `if` code block in order to explore the scope of local variables. After the `if` statement where you tested the value of `textMsg`, add these additional **bolded** Java statements:

```
// Experiment with Java statements.
          String testMsg = myErrorMsg.getErrorMsg ();
          if (myErrorMsg.getErrorMsg ().equals (testMsg)) {
// Define a temporary integer variable.
            int i = 5;
            System.out.println ("testMsg = text in ErrorMsg");
            System.out.println ("i = " + i);
          }
          else {
               System.out.println ("i = " + i);
          }
          System.out.println (“i = “ + i);
```

Attempt to compile this class. You should get an error message indicating that the compiler does not know the definition of the variable `i`. Delete the first `println` statement, then the other statement. Is either valid? What does this tell you about the scope of local variable `i`? Can it be accessed outside the code block where it was created (as defined by a pair of matching braces {})? Where would you need to place the definition of `i` in order to compile the previous Java statements? Go ahead and try it.

The code samples presented till now have all used the // style of comment identification. You will experiment with other styles of comment identification.

3. Comment out the lines you just added:

```
// Experiment with Java statements.
          String testMsg = myErrorMsg.getErrorMsg ();
          if (myErrorMsg.getErrorMsg ().equals (testMsg)) {
// Define a temporary integer variable.
               int i = 5;
               System.out.println ("testMsg = text in ErrorMsg");
               System.out.println ("i = " + i);
          }
/* Comment out the next few lines.
          else {
               System.out.println ("i = " + i);
          }
          System.out.println ("i = " + i);
Comment out the next few lines. */
```

Recompile this program.

4. Let's work with one of the more common Java constructs, the `for` loop. As mentioned earlier, this is analogous to the PERFORM COBOL verb. Add these lines to the end of your HelloWorld class:

```
// Experiment with for and case.
     int even = 0, odd = 0, other = 0, total = 0, i;
     for (i = 0; i < 9; i++) {

          total++;
          switch (i) {

          case 1:
          case 3:
          case 5:
          case 7:
               odd ++;
               break;

          case 2:
          case 4:
          case 6:
          case 8:
               even ++;
               break;

          default:
               other ++;
               break;
          }
     }
     System.out.println ("Odd, Even, Other, and Total = " + odd + ", " +
          even + ", " + other + ", " + total);
```

Compile and rerun the HelloWorld application. The output should look like this:

```
...
Odd, Even, Other, and Total = 4, 4, 1, 9
```

Look closely at the `for` loop definition and the output line. How many times did the `for` loop execute? What do you think the range of values for `i` were as the loop executed?

Remove the `break` statement in each of the `case` sections. What happens? Why did this happen? Hint: Refer to the introductory discussion on the `case` statement.

5. Let's test how the `continue` statement works in a `for` loop. Add these **bolded** lines to your HelloWorld class:

```
// Experiment with for and case.
     int even = 0, odd = 0, other = 0, total = 0, i;
     for (i = 0; i < 9; i++) {

          total++;
          if (i > -1 ) {
               continue;
          }
          switch (i) {
```

Compile and rerun the HelloWorld application. The output should look like this:

```
...
Odd, Even, Other, and Total = 0, 0, 0, 9
```

Look closely at the `for` loop definition and the output line. How many times did the `for` loop execute? Why do you think that no variables except `total` were incremented?

6. Let's create a `for` loop inside the original one. You will also observe the effect of a `break` statement in a `for` loop. Insert comment markers for the *italicized* lines, and add these **bolded** lines to your HelloWorld class:

```
// Experiment with for and case.
     int even = 0, odd = 0, other = 0, total = 0, i;
     for (i = 0; i < 9; i++) {

          total++;
//       if (i > -1 ) {
//            continue;
//       }
         switch (i) {

          case 1:
          case 3:
          case 5:
```

```
            case 7:
                odd ++;
                for (int i2 = 0; i2 < 10; i2++) {
                odd++;
                break;
                }
                break;
           case 2:
           case 4:
           case 6:
           case 8:
                even ++;
                break;
           default:
                other ++;
                break;
           }
      }
      System.out.println ("Odd, Even, Other, and Total = " + odd + ", " +
           even + ", " + other + ", " + total);
```

Compile and rerun the HelloWorld application. The output should look like this:

```
...
Odd, Even, Other, and Total = 8, 4, 1, 9
```

Look closely at the `for` loop definition and the output line. How many times did the outer `for` loop execute? Why do you think that the variable named `odd` was only incremented once for each execution of the inner loop, even though the inner loop definition says it should be performed 10 times?

7. For the last exercise, you'll define a labeled `break`. This statement allows you to go to the end of a code block from inside the code block. Add these **bolded** lines to your HelloWorld class:

```
// Experiment with for and case.
      int even = 0, odd = 0, other = 0, total = 0, i;
outerloop:
      for (i = 0; i < 9; i++) {

           total++;
           switch (i) {
```

```
                case 1:
                case 3:
                case 5:
                case 7:
                    odd ++;
                    for (int i2 = 0; i2 < 10; i2++) {
                        odd++;
                        break outerloop;
                    }
                    break;
                case 2:
                case 4:
                case 6:
                case 8:
                    even ++;
                    break;

                default:
                    other ++;
                    break;
                }
            }
            System.out.println ("Odd, Even, Other, and Total = " + odd + ", " +
                even + ", " + other + ", " + total);
```

Compile and rerun the HelloWorld application. The output should look like this:

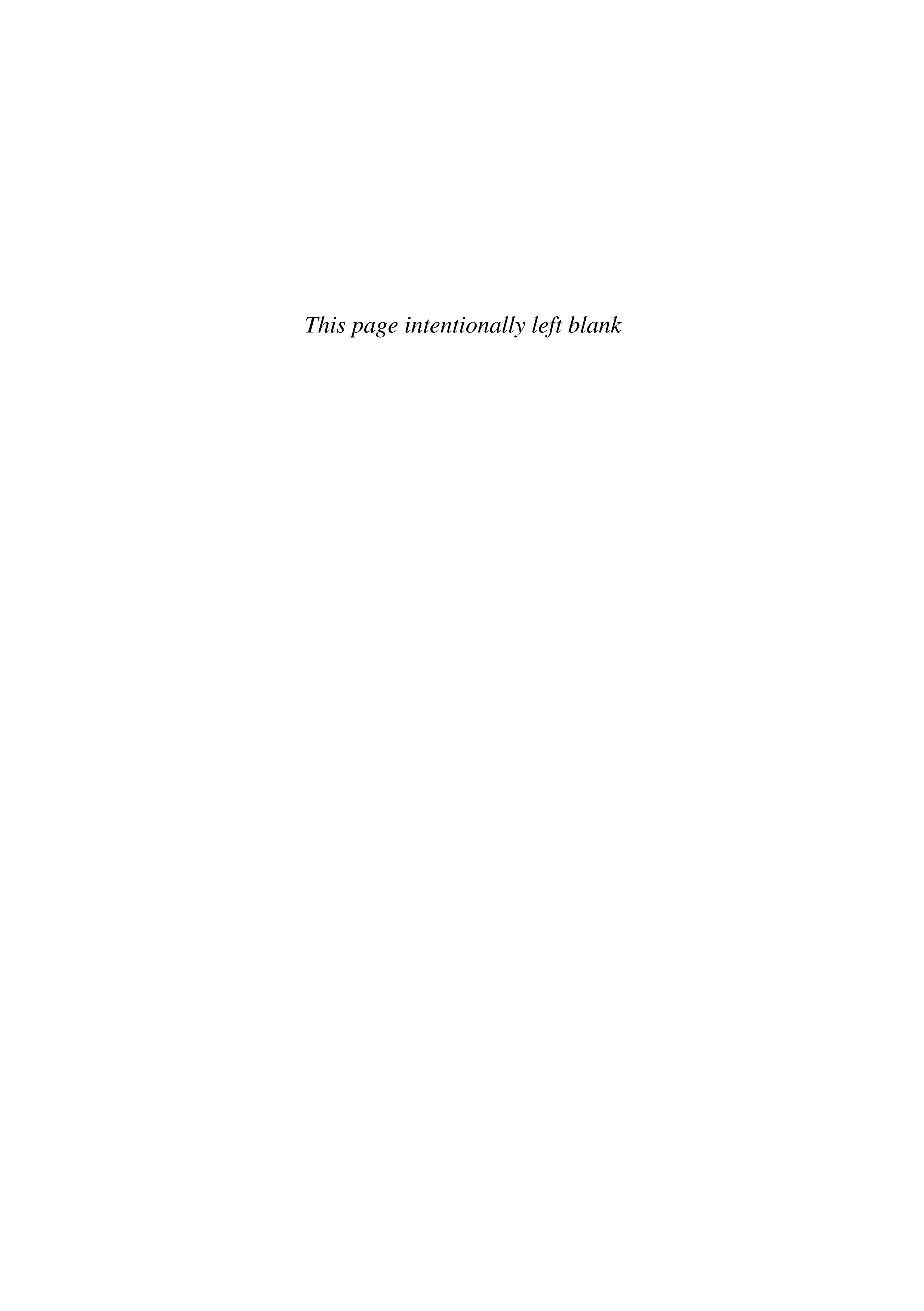
```
...
Odd, Even, Other, and Total = 2, 0, 1, 2
```

Look closely at the labeled `for` loop definition and the output line. How many times did the outer `for` loop execute this time? Why do you think that the variable named `Even` was never incremented in the loop?

Reviewing the Exercises

Let's review the samples you've created. Try to relate the sample source statements to the result (that is, the output) each statement creates. If necessary, rerun the samples, or look at the complete source code for this exercise on the CD-ROM. Feel free to experiment by yourself.

- The `switch` and `case` statements are similar to COBOL's EVALUATE verb.
- Java's `for` loop is similar to COBOL's PERFORM UNTIL construct. The `continue` statement causes the current iteration of this loop to terminate and the next iteration to start. The `break` statement causes the current code block to be exited immediately.
- Finally, you constructed a labeled `for` loop. After the loop was performed only two times, you exited the labeled loop abruptly, causing the statement after the loop to be executed. In fact, you exited the outer loop from inside an inner loop.


This page intentionally left blank

8 Strings, StringBuffers, StringBuilders, Numbers, and BigNumbers

In This Chapter

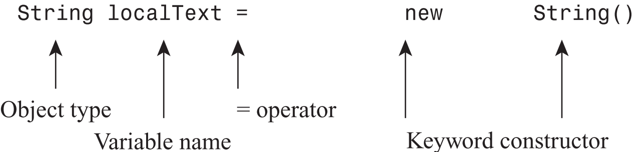
- Strings
- Comparing Strings
- Working with Strings
- Numeric Wrapper Classes
- StringBuffers
- BigNumbers
- Exercises: Strings, StringBuffers, Numbers, and BigNumbers
- Reviewing the Exercises

This chapter explores text manipulations and number-related classes and methods.

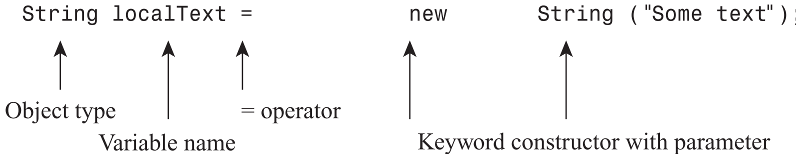
Strings

Java provides a `String` class that developers can use to hold and manipulate character strings (that is, a sequence of chars). The COBOL developer may recognize `Strings` as the equivalent of the PIC X(x) definition, and they are generally similar. However, Java's native `String` handling functions are far superior to COBOL's notoriously weak character manipulation support. Further, `Strings` are real objects, and so some differences naturally exist. Java `Strings` contain 16-bit Unicode charac-

ters, which can represent any foreign character within the Basic Multilingual Plane (BMP). However. Unicode 5.1 now contains over 100,000 characters, and so Java only supports the most commonly used character sets. This is in contrast to COBOL characters, which are only 8-bit, based on a particular code page (typically ASCII, EBCDIC, or UTF-8). The developer can create a `String` object using the standard Java syntax. You should be familiar with this construct, since you have used it several times already.

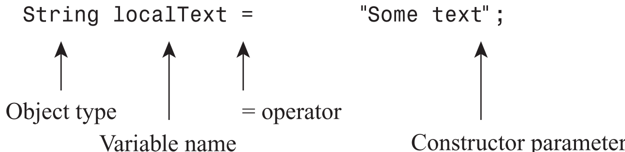


`String` provides an alternative constructor that accepts a `String` parameter.



This statement will set the `localText String` reference equal to a `String` that contains "Some Text."

This complete syntax is often compressed into the following form:



Notice that the compressed syntax is very similar to the syntax used to define primitive data types:

```
int localInt =     1;
```

Despite the similarity in syntax, there is a major difference in the examples. `localInt` is a primitive data type, not an object. In contrast, `localText` is an object, since all `Strings` are objects.

Let's review the following statement:

```
String localText =       "Some text";
```

First, a `String` object containing the text "Some text" was created by the compiler. Then, the object reference variable `localText` was created and adjusted to reference that object. The compiler handles all this automatically because it treats `String` variables in a special way. Whenever the compiler sees a double quoted constant, it will automatically make sure that a `String` object has been created.

The `String` *class, part of the Java runtime, actually maintains a pool of* `Strings` *with unique values. So the first time the runtime sees a literal string or a string-valued constant expression, the* `String` *class will check an internal pool of previously created* `Strings`*. If one is found, then that* `String` *will be used. Otherwise, a new one will be created and added to the internal pool. In either case, the compiler will make sure a* `String` *object is created for your program when you use the previously mentioned syntax.*

This runtime optimization helps conserve memory utilization and improve performance for commonly used strings. However, in most cases you, as a Java developer, should not be concerned with, or code in a manner that is dependant on, these details, since the `String` *class encapsulates this optimization in proper OO fashion.*

Comparing Strings

The base `Object` class contains many useful methods. Since the `String` class inherits from `Object`, every `String` object you create will contain these methods as well. One example is a method called `equals()` that will compare two `Strings` for equality:

Return type	Method name	Parameter signature
boolean	equals	(Strings)

A Java statement can use this method to compare the text value in a `String` object (such as `localText`) to some other `String` object:

```
if (localText.equals ("Some other text")) {
   ...
}
```

You may wonder why I didn't use the more familiar equality operator (==) to evaluate `localText`. This is because `Strings` are objects and not intrinsic data types. Therefore, the equality operation would compare the object reference variables and not the text contained in those objects.

```
if (localText == "Some other text") {
     ...
}
```

This is a valid statement and looks natural, but it is really one you will hardly ever use. It can be read the following way: "Compare the object reference variable `localText` for equality to the (implicit) object reference variable that contains 'Some other text.' Return true if they are exactly equal."

Since object reference variables contain the memory location of the object, this statement will be true only if both variables point to exactly the same location in memory! This will rarely be true, so the preceding statement will almost always fail. It is clearly not the way to test `String` variables or any other types of objects for equal text values.

`Strings` have a number of special characteristics assigned to them by Java. One is the fact that they are considered *immutable*, meaning that they cannot change.

"Wait a minute!" you may be saying. "We saw some examples that changed the value of `String` variables."

Actually, there were no such examples. In reality, new `Strings` are created when required, and the `String` variable was changed to point to the new object. Consider the original `String` definition statement:

```
String localText =       "Some text";
```

A subsequent statement can cause `localText` to point to another `String`:

```
localText =       "Some other text";
```

Follow what really happens when this statement is executed:

1. A new `String` object containing the text "Some other text" is created when required by the runtime.
2. The `String` reference variable `localText` is changed to point to this object instead of the original object.
3. The original `String` object ("Some text") is marked as unused and may eventually be deleted.

Working with Strings

`Strings` can easily be concatenated using the + operator:

```
String localText =      ("Some text ");
localText += "/ Some other text";
```

After execution of the second statement, `localText` would point to a (new) `String` containing "Some text / Some other text."

The + operator will automatically convert any data type into a `String` if necessary.

```
String localText =      ("Some text ");
int x = 4;
localText += "/ "
localText += x;
```

After execution of the second statement, `localText` would point to a (new) `String` containing "Some text / 4."

As a COBOL programmer, one would think that to add a number to a `String` might be considered a nonsensical operation, yielding unpredictable results. What would happen in COBOL if you said the following:

```
ADD 4 TO "Some Text".
```

Some sort of compiler error, you hope! But in Java, this type of statement is perfectly legal. The numeric item automatically gets converted into a `String`.

Java accomplishes this by assigning special processing to the + operator in some cases, such as when the destination is a `String` variable. In this case, the + operator becomes a *concatenate* operator. If any of the source operands are not `Strings`, they will automatically be converted into `Strings` by the compiler, and then concatenated. In the example, the value 4 was originally created as an integer, and then it was converted into the character `String 4`. This `String` was then added to the text in `localString`, creating a new `String` object.

This scenario is an example of a concept called operator overloading. *Some OO languages allow operators such as + to be overloaded, or redefined by the developer. In this way, special processing can be performed based on the class type. Although this is a very powerful tool, it can easily lead to programs that are hard for anyone but the original developer to understand. For example, a developer might look at the statement x – y, and expect a straightforward algebraic function. But if the – operator has been overridden when applied to y, then the developer may ask, "What does x – y actually mean in this case, and why is it so different from a – b in the previous statement?" Java does not allow operators to be overloaded by the developer, although the compiler uses the concept itself in this case.*

The conversion of objects and data types into `Strings` is accomplished by the `toString()` method. Every Java object and data type has a method of this name. The + operator simply calls this method and concatenates the `String` returned by the method with the current `String`. (This is actually a good demonstration of the power of polymorphism.)

Java's class definition for `Strings` includes quite a few very useful methods. Some are described in Table 8.1, along with the (roughly) equivalent COBOL statements.

TABLE 8.1 STRING MANIPULATION IN JAVA AND COBOL

Method	Description	COBOL Equivalent
char charAt (int index)	Return the character at position index in String.	Reference modification: **STRING1**(3:1)
int compareTo (String string2)	Compare two Strings; return –1, 0, or 1 depending on relative position in Unicode order.	IF **STRING1** <,=,> **STRING2**
String concat (String string2)	Return a new String where all string2's characters have been appended to string1's.	STRING **STRING1**, **STRING2** INTO **STRING3** DELIMITED BY SPACES (or SIZE)
boolean endsWith (String string2)	Test if String ends in string2.	IF **STRING1**(CTR:SIZE) = **STRING2**
boolean equals (object anyObject)	Test if the text in String equals the text in object.	IF **STRING1** = **STRING2**[1]

continued

Method	Description	COBOL Equivalent
boolean equalsIgnoreCase (String string2)	Test if the text in String equals the text in string2, regardless of case.	MOVE **STRING1** TO **NEW-STRING** INSPECT CONVERTING **NEW- STRING2** IF **STRING2** = **NEW-STRING**
int indexOf (String string2)	Test if the text in String contains the text in string2. Return its first location or –1.	INSPECT **STRING1** TALLYING CTR FOR CHARACTERS BEFORE INITIAL **STRING2**
int lastIndexof (String string2)	Test if the text in String contains the text in string2. Return its last location or –1.	INSPECT **STRING1** TALLYING COUNTER FOR ALL **STRING2**
int lastIndexof (String string2, int fromIndex)	Test if the text in String contains the text in string2 starting at position fromIndex. Return its last location or –1.	INSPECT **STRING1**(CTR:SIZE) TALLYING **COUNTER** FOR ALL **STRING2**
int length() defined at compile time.	Return the length of String.	Items are always fixed length.
String replace (char old Char, char newChar)	Return a new String where all occurrences of oldChar have been replaced with newChar.	INSPECT **STRING1** REPLACING ALL **OLD-CHAR** BY **NEW-CHAR**
boolean startsWith (String string2)	Test if the text in String begins with the text in string2.	IF **STRING1**(1:SIZE)=**STRING2**
String substring (int beginIndex)	Return a new String with the characters starting at beginIndex to the end of string.	MOVE **STRING1**(POS:SIZE-REMAINING) TO **STRING2**
String substring (int beginIndex,int endIndex)	Return a new String with the characters starting at beginIndex to endIndex.	MOVE **STRING1**(POS:SIZE) TO **STRING²**
String toLowerCase ()	Return a new String with all the characters in String converted to lowercase.	MOVE **STRING1 TO NEW-STRING** INSPECT CONVERTING **NEW-STRING**

continued

Method	Description	COBOL Equivalent
String toUpperCase ()	Return a new String with all the characters in String converted to uppercase.	MOVE **STRING1 TO NEW-STRING** INSPECT CONVERTING **NEW-STRING**
String trim ()	Return a new String with the characters in String, except leading and trailing spaces.	STRING **STRING1** INTO **STRING2** DELIMITED BY SPACES
String valueOf (datatype arg)	Return a String representation of arg.	MOVE **NUMBER** TO **NUMBER-AS-Z9**[3]

1. *COBOL automatically adds spaces to the end of strings of unequal length. Java* `Strings` *have an implied length (including all spaces) and are not equal unless their sizes are also equal.*
2. *Multilanguage case conversion is not possible with this approach. Library packages are recommended.*
3. *There is no COBOL analogy for a string representation of true or false, but numeric types can be converted into characters.*

NUMERIC WRAPPER CLASSES

In the same way that numeric items can be converted into `Strings`, `Strings` sometimes need to be turned into numbers. However, since primitive data types are not objects, they do not have methods and members. Where can the Java language designers place the necessary methods?

To provide a mechanism to represent primitive types as objects, Java defines special classes that wrap the basic numeric data types into objects. These classes are very useful when objects (rather than intrinsic data types) are required. These classes have familiar names (`Integer`, `Long`, `Float`, `Double`, `Byte`), and all are inherited from the parent `Number` class. All sorts of useful methods are contained in these classes.

One example is the `parseInt()` method, which the developer can use to convert a `String` into a numeric item. This method accepts a `String` input parameter and returns a numeric item of type `int`.

This method implements the Java construct called exception processing. This means that a program can call the method, but some exception may occur before the method completes. The calling program can use the `try` *and* `catch` *code blocks to properly handle possible exceptions. Exceptions are discussed in more detail in Chapter 9.*

```
String inputMsg = " 003 ";
int x = 0;
// Try to convert String to integer. Catch any NumberFormat exception errors.
     try  { x = Integer.parseInt (inputMsg.trim());
     }
     catch (NumberFormatException e) {
                         ...
          x = 0;
     }
```

This code sample first removes any trailing or leading spaces from `inputMsg` (using its `trim()` method), and then passes that result `String` to the `parseInt()` method in the `Integer` class. (Notice that no `Integer` object was created before `parseInt()` was called; this method is a static method, and so can be accessed without first creating an `Integer` object.) If any nonnumeric characters are discovered in `inputMsg`, the `NumberFormatException` exception will be thrown. It is best if this code block catches the exception and performs any appropriate action (such as setting x to zero).

The `Double` numeric class wrapper does not have a similar method. Instead, you have to use the `doubleValue()` method to return a numeric of type `double` and then pass the returned `String` to one of the constructors for the `Double` class. The (rather gruesome) code looks like this:

```
String inputMsg = " 003 ";
double d = 0;
// Try to convert String to a double using the doubleValue() String method.
// Catch any NumberFormat exception errors.
     try  { d = new Double (inputMsg.trim()).doubleValue();
     }
     catch (NumberFormatException e) {
          ...
          d = 0;
     }
```

Java does provide a slightly more useful and consistent mechanism to convert `Strings` into numerics: the `DecimalFormat` class and its `parse()` method. This method accepts a `String` parameter and returns an abstract class of type `Number` (that is, either a `Long` or a `Double` object). The `Number` class, in turn, does implement the `doubleValue()` method, so you can always get a numeric double value from it if you would like:

```
String inputMsg = "1,003.4";
double d = 0;
// Try to convert String to a double. Catch any NumberFormat exception errors.
try {
      d = new DecimalFormat().parse(inputMsg.trim()).doubleValue();
    }
    catch (NumberFormatException e) {
        ...
        d = 0;
    }
```

The `parse()` method for this class will process numeric `String` input items, even if they contain thousands of separator characters, such as the comma (,). Still, this is clearly more work than one would expect to simply convert a `String` to a numeric value!

Java 1.5 introduces the `Scanner` class that contains a number of helpful string manipulation utilities. For example, the `nextInt()` method is a convenient mechanism to convert a `String` to an integer.

```
String inputMsg = "1003";
Scanner s = new Scanner(inputMsg);
int x = 0;
x = s.nextInt();
```

Primitive data items can be placed into a `Double` wrapper object quite easily by using the proper constructor:

```
double d = 2.4;
Double dd = new Double (d);
```

And, of course, the `Double` wrapper class can return a primitive data type as well:

```
Double dd = new Double ("222.3");
double d = dd.doubleValue ();
```

As mentioned earlier, the wrapper classes contain other useful methods, including `compareTo()`, `toString()`, and so on. These methods can be used to perform some functions directly on wrapper objects, without having to always convert the objects to primitive data types first (see Table 8.2).

TABLE 8.2 COMPARISON AND CONVERSION OF DOUBLE, FLOAT, INTEGER, LONG, AND SHORT WRAPPER OBJECTS

Method	Description	COBOL Equivalent
int compareTo (wrapper arg)	arg is numerically compared to this wrapper object. Each wrapper class supports a numeric comparison for objects of the same type as this wrapper.	IF **NUMBER1** = **NUMBER2**
int compareTo (object arg)	arg is compared as an object to this wrapper object.	
double doubleValue ()	Returns a primitive numeric data type.	data type.
String toString ()	Returns a String representation of this wrapper object.	MOVE **NUMBER** TO **NUMBER-AS-Z9**

Java 1.5 introduces the concept of *autoboxing* and *unboxing*. These are conversion functions that are automatically implemented by the compiler. When the compiler detects that a primitive data type needs to be converted to a wrapper class, it will generate the code to do so (autoboxing). When a primitive data type needs to be extracted from a wrapper class (unboxing), the compiler will insert that code as well.

The following code fragment creates a new `Integer` object (`ii`), then adds 12 to the value in that object. Prior to Java 1.5, this code fragment would have created a compiler error. With Java 1.5, the compiler automatically creates the code necessary to unbox the `int` value, adds 12 to it, then automatically boxes the new value into an `Integer`.

```
Integer ii = new Integer (23);
ii = ii + 12;
```

Although the compiler will perform these transitions for you, you should not use the autoboxing feature as a substitute for remembering the distinction between classes and primitives. If you really want to add a value to an `int`, you should do that, and not simply add the value to an `Integer`. Since there are many more steps to add to the value in an `Integer`, it is much slower compared to adding to a primitive `int`, even if the code looks very similar.

StringBuffers

Java provides a `StringBuffer` class, which is a class similar to `String`, except that objects of this class type can be modified. Developers use objects of this type instead of `Strings` when the text contained in the object must be modified frequently. `StringBuffers` have a size attribute and are automatically increased in size as necessary. The only real downside to `StringBuffers` is that they do not perform as well in some cases as `Strings` do, and you cannot use the + operator for concatenation.

`StringBuffers` can be converted into `Strings` using the `toString()` method. Conversely, any data type can be easily added to a `StringBuffer` using the `append()` method. In this case, the same operator overloading conventions defined by the `String` class are used.

Some particularly useful `StringBuffer` methods are listed in Table 8.3. Unless otherwise noted, the return type is a `StringBuffer`, and this `StringBuffer` is modified by the method.

TABLE 8.3 STRING MANIPULATION IN JAVA AND COBOL

Method	Description	COBOL Equivalent
append (datatype arg)	Arg is converted into a String and added to the end of StringBuffer.	MOVE NUMBER TO NUMBER-AS-Z9 STRING NUMBER-AS-Z9 TO STRING1
char charAt (int index)	Returns the character at position index in the StringBuffer.	MOVE **STRING2**(POS:1) TO **CHARACTER**
setCharAt (int index, char character)	The character at position index is replaced with character.	MOVE **CHARACTER** TO **STRING2**(POS:1)
insert (int offset, datatype arg)	Arg is converted first to a String and then placed in StringBuffer starting at position offset.	MOVE **NUMBER** TO **NUMBER-AS-Z9** MOVE **STRING1-REDEFINES-NUMBER- AS-Z9** TO **STRING2**(POS:SIZE)
int length ()	Return the length of StringBuffer.	Items are always fixed length, defined at compile time.
replace (int offset, int length, String str)	Replace the characters in StringBuffer starting at offset for the number of chars in length with the characters in string.	MOVE **STRING1**(POS:LEN) TO **STRING2**

continued

Method	Description	COBOL Equivalent
setLength (int length)	Set the length of StringBuffer to length. Any new character positions will be set to null.	Items are always fixed length, defined at compile time.
String subString (int offset)	Return a String with all the characters in stringBuffer starting at offset.	MOVE **STRING1**(POS:LEN) TO **STRING2**
String subString (int offset, int length)	Return a String with all the characters in StringBuffer starting at offset for the number of chars in length.	MOVE **STRING1**(POS:LEN) TO **STRING2**
String toString	Return a String with all of the characters in StringBuffer.*	MOVE **STRING1** TO **STRING2**

* *A new* `String` *is not created at first. Instead, the new* `String` *simply points to the same place in memory where the text for* `StringBuffer` *is stored. If and when* `StringBuffer` *is eventually changed,* `StringBuffer` *will allocate a new place in memory to hold the text.*

Here are examples of how `StringBuffers` might be used:

```
// Build an array of Strings.
// This array will contain 6 String objects.
     String[] inputWords = {"These", "are", "words", "in", "a", "sentence"};
// Build a sentence.
     String sentence = makeSentence (inputWords);
     System.out.println (sentence);

     public String makeSentence(String[]sentenceWords) {
           StringBuffer sent = new StringBuffer();
           for (int i = 0; i < sentenceWords.length; i++) {
// Build up the sentence. Place each word in the StringBuffer followed by a
// space.
// The StringBuffer will be automatically resized for each append.
              sent.append(sentenceWords[i]);
              sent.append(" ");
          }
// Return the sentence (as a String).
          return (sent.toString());
      }
```

BigNumbers

Although Java does not define a primitive data type similar to COBOL's familiar packed decimal, Java defines two classes that are similar. These support math functions for numbers of arbitrary (and user-defined) precision. When you use these classes instead of the primitive float and double intrinsic data types, there is no requirement to work around the precision problems of traditional floating-point arithmetic. The `BigInteger` class supports arbitrary precision integer arithmetic, and `BigDecimal` supports arbitrary precision arithmetic with scale (that is, decimal positions). With Java 1.5, `BigDecimal` also supports fixed precision floating-point computation.

Both `BigInteger` and `BigDecimal` have constructors with a `String` as a parameter. This is often how a `BigNumber` is initialized:

```
BigInteger customerNumber =
     new BigInteger ("12345678901234567890123456789O");
BigDecimal currencyValue =
     new BigDecimal ("4.56789012345678901234 5");
BigDecimal customerOrder =
     new BigDecimal (customerNumber, 10);
```

In the example, `currencyValue` will have a scale attribute (that is, the number of digits to the right of the decimal) of 11. `customerOrder` would contain the numeric value in `customerNumber`, with a scale of 10. You can also use the static method `valueOf()` to set a `BigNumber` to a value:

```
BigInteger currencyValue = BigInteger.valueOf(123);
BigDecimal currencyValue = BigDecimal.valueOf(123, scale);s
```

These classes do come with a cost, however. Math functions using these classes are considerably slower than math functions that use the primitive data types. Another inconvenience: You have to use the math functions the classes provide rather than the native Java operators (+, –, *, /, and so forth).

This is another example of how operator overloading could make Java applications simpler to read and write. If Java had overloaded the math operators when applied to `BigNumbers`*, this syntax could be used:*

```
     customerBalanceYen = customerBalance * currencyValue;
instead of this syntax:
     customerBalanceYen = customerBalance.multiply (currencyValue);
```

As it is, the math functions provided with the `BigNumber` *classes are the only way to add, subtract, multiply, and divide* `BigNumbers`*.*

The `BigDecimal` constructors and math functions also support an optional `MathContext` parameter. This class allows you to establish a particular set of `BigDecimal` attributes (scale, rounding mode) and use that set each time you need a new `BigDecimal`.

```
MathContext currencyScale = new MathContext (123, RoundingMode.HALF_DOWN);
BigDecimal currencyValue = new BigDecimal (123, currencyScale);
BigDecimal currencyRate = new BigDecimal (1.3, currencyScale);
BigDecimal convertedCurrencyValue =
    currencyValue.multiply (currencyRate, currencyScale);
```

Table 8.4 shows the most commonly used constructors for `BigDecimals` and `BigIntegers`.

TABLE 8.4 COMMON CONSTRUCTORS FOR BIGDECIMAL AND BIGINTEGER

Constructor	Description	COBOL Equivalent
BigDecimal (BigInteger val)	Converts a BigInteger into a BigDecimal	MOVE **INTEGER1** TO **PACKED-NUM1**
BigDecimal (BigInteger unScaledVal, int scale)	Converts a BigInteger into a BigDecimal with the scale requested	MOVE **INTEGER1** TO **PACKED-NUM1**
BigDecimal (double val)	Converts a double into a BigDecimal	MOVE **DOUBLE-NUM1** TO **PACKED-NUM1***
BigDecimal (String val)	Converts a String representation of a number into a BigDecimal	MOVE WITH CONVERSION **STRING1** TO **PACKED-NUM1**

continued

Constructor	Description	COBOL Equivalent
BigInteger (String val)	Converts the decimal String representation of a number into a BigInteger	MOVE WITH CONVERSION STRING1 TO INTEGER-NUM1
BigInteger (byte[ ] val)	Converts a byte array that contains the 2's complement binary representation of a BigInteger into a BigInteger	MOVE **GROUP-ITEM1** TO **INTEGER-NUM-GROUP-ITEM1**

* *Some COBOL compilers do support floating-point numerics.*

Let's compare some COBOL statements that use packed numbers to their Java equivalents that use `BigDecimals`.

```
01 DECIMAL-ITEMS USAGE IS COMP-3.
    03 DOLLAR-AMOUNT              PIC S9(7)V(2)          VALUE "1234567.12".
    03 CURRENCY-RATE              PIC S9(3)V9(5)         VALUE "123.12345".
    03 EXCHANGE-AMOUNT            PIC S9(10)V9(4).
     MULTIPLY DOLLAR-AMOUNT BY CURRENCY-RATE GIVING
        EXCHANGE-AMOUNT.
```

Note that there is the real possibility that the result of this multiplication may not fit in EXCHANGE-AMOUNT, and the value that is placed in this variable will be an approximation of the actual result. By default, COBOL will truncate the intermediate result to fit the target variable.

The most straightforward way to handle this situation is to define a larger target variable.

```
01 DECIMAL-ITEMS USAGE IS COMP-3.
    03 DOLLAR-AMOUNT              PIC S9(7)V(2)          VALUE "1234567.12".
    03 CURRENCY-RATE              PIC S9(3)V9(5)         VALUE "123.12345".
    03 EXCHANGE-AMOUNT            PIC S9(10)V9(7).
     MULTIPLY DOLLAR-AMOUNT BY CURRENCY-RATE GIVING
     EXCHANGE-AMOUNT.
```

In this example, EXCHANGE-AMOUNT will always be able to contain the result. The advantage of fixed-decimal arithmetic is that the developer can always control the precision of the result by managing the size of the result variable(s).

But at some point you will run into limits. Either the compiler will specify an upper limit on the number of decimal positions available (traditionally 17 digits in COBOL) or a limit is defined in the external representation of the number (for printing or for storage in a database, for example).

In these cases, result values must be *cast* into smaller variables. The COBOL developer can control the precision of the result of this casting by defining the precision of the variables. The developer also has some ability to specify how rounding should be handled.

```
01 DECIMAL-ITEMS USAGE IS COMP-3.
    03 DOLLAR-AMOUNT              PIC S9(7)V(2)          VALUE "1234567.12".
    03 CURRENCY-RATE              PIC S9(3)V9(5)         VALUE "123.12345".
    03 EXCHANGE-AMOUNT            PIC S9(10)V9(4).
     MULTIPLY DOLLAR-AMOUNT BY CURRENCY-RATE GIVING
     EXCHANGE-AMOUNT.
     MULTIPLY DOLLAR-AMOUNT BY CURRENCY-RATE GIVING
     EXCHANGE-AMOUNT ROUNDED.
```

In the first example, any digits beyond the 10–3 position will be truncated from the (intermediate) result before being moved into EXCHANGE-AMOUNT. In the second example, the digit in the 10–4 position will be evaluated. If greater than 4, the previous digit will be incremented by 1. The remaining digits will be truncated.

Java allows for similar types of fixed-precision math functions with its `BigDecimal` object type and to a lesser extent its `BigInteger` object type. Both types are examples of arbitrary precision data types. This means they are automatically defined to be the correct size and can, in theory, hold any result. Further, the developer can specify (even at runtime) any number of decimal positions (that is, the scale) in a `BigDecimal` number. As an added bonus, Java provides eight different rounding options, allowing the developer to have complete control when moving intermediate results into smaller target variables (for external storage or for printing, for example).

```
BigDecimal dollarAmount = new BigDecimal ("12345678901.12");
BigDecimal currencyRate = new BigDecimal ("123.12345");
BigDecimal exchangeAmount = new BigDecimal;
exchangeAmount = dollarAmount.multiply (currencyRate);
exchangeAmount = dollarAmount.multiply (currencyRate,
                                        ROUND_HALF_UP);
```

The static integers shown in Table 8.5 can be passed into some of the math functions of `BigDecimals` to control their rounding functions.

TABLE 8.5 ROUNDING OPTIONS IN JAVA AND COBOL

Mode	Description	COBOL Equivalent
ROUND_CEILING	Round toward positive infinity.	
ROUND_DOWN	Round toward zero (and away from 10).	
ROUND_FLOOR	Round toward negative infinity.	
ROUND_HALF_DOWN	Round toward "nearest neighbor" (i.e., toward 0 or 10). If both neighbors are equidistant, round down.	
ROUND_HALF_EVEN	Round toward the "nearest neighbor." If both are equidistant, round toward the even neighbor.*	
ROUND_HALF_UP	Round toward "nearest neighbor." If both are equidistant, round up.	ROUNDED
ROUND_UNNECESSARY	Asserts that the requested operation has an exact result, and no rounding is necessary.	
ROUND_UP	Rounds away from zero (toward 10).	

* *This mode should, in theory, round up as often as it rounds down in case of equidistant neighbors, thereby reducing one source of rounding error.*

Table 8.6 shows the most common math functions involving `BigDecimals`.

TABLE 8.6 COMMON MATH FUNCTIONS IN JAVA AND COBOL

Method	Description	COBOL Equivalent
int compareTo (BigDecimal val)	Compares this BigDecimal with another BigDecimal	IF **PACKED-NUM1** = **PACKED-NUM2**
BigDecimal add (BigDecimal val)	Returns a BigDecimal whose value is (this + val), and whose scale is the greater of this.scale() and val.scale()	ADD **PACKED-NUM1** TO **PACKED-NUM2** GIVING **PACKED-NUM3**
BigDecimal subtract (BigDecimal val)	Returns a BigDecimal whose value is (this − val), and whose scale is the greater of this.scale() and val.scale()	SUBTRACT **PACKED-NUM1** FROM **PACKED-NUM2** GIVING **PACKED-NUM3**
BigDecimal multiply (BigDecimal val)	Returns a BigDecimal whose value is (this * val), and whose scale is the sum of this.scale() and val.scale()	MULTIPLY **PACKED-NUM1** BY **PACKED-NUM2** GIVING **PACKED-NUM3**
BigDecimal divide (BigDecimal val, int roundingMode)	Returns a BigDecimal whose value is (this / val), and whose scale is the same as this.scale(). Rounding behavior is controlled by the value of roundingMode	DIVIDE **PACKED-NUM1** BY **PACKED-NUM2** GIVING **PACKED-NUM3** ROUNDED
BigDecimal divide (BigDecimal val, int scale, int roundingMode)	Returns a BigDecimal whose value is (this / val), and whose scale is as specified by scale	DIVIDE **PACKED-NUM1** BY **PACKED-NUM2** GIVING **PACKED-NUM3** ROUNDED
Lint scale (BigDecimal val)	Returns the scale of this BigDecimal	
BigDecimal setScale (int scale, int roundingMode)	Returns a BigDecimal whose scale is as specified. Truncated digits are evaluated based on the roundingMode	MOVE **PACKED-NUM1** INTO **PACKED-NUM2**

EXERCISES: STRINGS, STRINGBUFFERS, NUMBERS, AND BIGNUMBERS

Time to visit the example classes again and try out all these new ideas.

1. Edit the HelloWorld.java source file in your java4cobol directory with a text editor. You'll start by deleting the code that experimented with flow control statements. Remove the lines after this statement (but remember to leave in the two curly braces at the end of the program):

```
// Print the contents of ErrorMsg's String data member directly.
          System.out.println (myErrorMsg.msgText);
```

2. Add these Java expressions at the end of the previous statement (before the last two curly braces). In this set of examples, a view of the output line is inserted after the `println()` statement so that the concept explored by the example is easier to follow.

```
// Experiment with Java Strings.
// Construct a new String and initialize it, using its
// constructor.
// Point the object reference variable englishCaps to this new
// String.
          String englishCaps = " ABCDEFGHIJKLMNOPQRSTUVWXYZ ";
// Get the size of the String using the length() method.
          System.out.println ("length() = " + englishCaps.length
());
...
// length() = 28 in this case
// Remove the leading and trailing spaces from the String.
// Point the reference variable englishCaps to this new String
          englishCaps = englishCaps.trim ();
// Get the new size of the String using the length() method.
          System.out.println ("length() = " +
                englishCaps.length ());
...
// length() = 26 in this case
// Find the position of the character M.
          System.out.println ("indexOf(M) = " +
                englishCaps.indexOf("M"));
// Get the character at position 12.
```

```
// Since all Java indexes start with 0, index 12 is actually the
// 13th position.
// This method returns a char data type, which is converted
// into a String by the println() method.
    System.out.println ("charAt(12) = " +
          englishCaps.charAt(12));
...
indexOf(M) = 12
charAt(12) = M
// Get the substring at position 12 through 17.
          System.out.println ("substring(12, 17) = " +
                englishCaps.substring (12, 17));
...
substring(12, 17) = MNOPQ
// Make a new String with the lowercase representation of this
// String.
          String englishLower = englishCaps.toLowerCase ();
          System.out.println ("englishLower = " +
                englishLower);
...
englishLower = abcdefghijklmnopqrstuvwxyz
// Compare the two English Strings using two different String
// methods.
          System.out.println ("compareTo = " +
                englishCaps.compareTo (englishLower));
          System.out.println ("equalsIgnoreCase = " +
                englishCaps.equalsIgnoreCase (englishLower));
...
compareTo = -32
equalsIgnoreCase = true
// Create a duplicate of englishCaps.
// Compare it to englishCaps using two String comparison
// methods.
 String temp = englishCaps;
          System.out.println ("compareTo = " +
                englishCaps.compareTo (temp));
          System.out.println ("equals = " +
                englishCaps.equals (temp));
...
compareTo = 0
equals = true
// Compare the two String reference variables.
```

```
          System.out.println ("compare reference variables = "
                + (englishCaps == temp));
...
compare reference variables = true
// Convert temp to lowercase.
// Compare temp to englishLower using the String method
// equals().
 temp = englishCaps.toLowerCase ();
          System.out.println ("equals = " +
                englishLower.equals (temp));
// Then compare the two String reference variables.
// The result of this comparison is difficult to predict, but
// will normally be false.
          System.out.println ("compare reference variables = "
             + (englishLower == temp));
...
equals = true
compare reference variables = false
```

3. Save these modifications as a text file, and then compile the class in the DOS command window:

```
➔  javac HelloWorld.java
Hello World!

Some Text
Some Text
length() = 28
length() = 26
indexOf(M) = 12
charAt(12) = M
substring(12, 17) = MNOPQ
englishLower = abcdefghijklmnopqrstuvwxyz
compareTo = -32
equalsIgnoreCase = true
compareTo = 0
equals = true
compare reference variables = true
equals = true
compare reference variables = false
```

Examine the various string methods you've used and the results they have returned. Do they make sense to you? Experiment with some other string methods on your own (perhaps the `replace()` method).

4. Now you'll experiment with the numeric wrapper classes. Add this **bolded** Java statement to the beginning of your HelloWorld source:

```
import java.text.*;
//
//
// HelloWorld Application
//
//
```

5. Add these Java statements at the end of the previous statement (before the last two curly braces):

```
// Experiment with the numeric wrapper classes.
// Convert from String to several numeric types.
// You will use the static methods in the wrapper classes.
     int i;
     long l;
     double d, d2;
     String inputMsg = "003";
// Try to convert String to integer. Catch any NumberFormat
// exception errors.
     try    {
            i = Integer.parseInt (inputMsg.trim());
     }
     catch (NumberFormatException e) {
            i = 0;
     }
// Try to convert String to a long. Catch any NumberFormat
// exception errors.
     try    {
            l = Long.parseLong (inputMsg.trim());
     }
     catch (NumberFormatException e) {
            l = 0;
     }
// Try to convert String to a double. Catch any NumberFormat
// exception errors.
     try    {
            d = new Double (inputMsg.trim()).doubleValue();
     }
     catch (NumberFormatException e) {
            d = 0;
     }
```

```
// Use the DecimalFormat Static method to convert a complex
// String to a double.
     try    {
            d2 = new DecimalFormat().parse("3.0").doubleValue();
     }
     catch (ParseException e) {
            d2 = 0;
     }
     System.out.println ("Integer value = " + i);
     System.out.println ("Long value = " + l);
     System.out.println ("Double value = " + d);
     System.out.println ("Second Double value = " + d2);
```

6. Compile and run this class. The output should look like this:

```
...
compare reference variables = true
equals = true
compare reference variables = false
Integer value = 3
Long value = 3
Double value = 3.0
Second Double value = 3.0
```

7. Let's do some math with the wrapper classes. Add these Java statements to the end of the previous statement (before the last two curly braces):

```
// Math and the numeric wrapper classes
// Make two objects of type Double.
// Try out both Constructors.
     Double dd = new Double ("3.0");
     Double dd2 = new Double (dd.doubleValue());
// Perform an addition function.
// Since the wrapper classes do not have any math function
// themselves,
// you have to convert them to primitive data types (by using
// the method
// doubleValue()) and then use the result as a constructor
// parameter to a
// new wrapper class.
     d = dd.doubleValue ();
     d2 = dd2.doubleValue ();
     d += d2;
     dd = new Double (d);
```

```
     System.out.println ("dd after addition = " + dd);
// Or combine the previous expressions into one statement:
     dd = new Double (dd.doubleValue () + dd2.doubleValue());
     System.out.println ("dd after addition = " + dd);
// Compare two Double objects.
// Note that the wrapper classes do have comparison methods
// (compare(), and
// equals()), so you don't have to extract the value into a
// primitive data type.
     if (dd.equals (dd2)) {
           System.out.println ("dd equals dd2");
     }
     else {
           System.out.println ("dd does not equal dd2");
     }
// Let's use the autoboxing features,
// Let the compiler perform any conversions
// necessary to add 11 to dd
     dd = dd + 11;
     System.out.println ("dd after addition with autoboxing = "
+ dd);
```

8. Compile and run this class. The output should look like this:

```
nd Double value = 3.0
dd after addition = 6.0
dd after addition = 9.0
dd does not equal dd2
dd after addition with autoboxing = 20.0
```

9. Next you'll experiment with the `StringBuffer` class. Add these Java statements to the end of the previous statement (before the last two curly braces):

```
// Experiment with StringBuffers.
// Convert this String array to a StringBuffer.
     String[] inputWords = {"These", "are", "words", "in", "a",
"sentence"};
// Build a sentence.
     StringBuffer sent = new StringBuffer();
     for (i = 0; i < inputWords.length; i++) {
// Build up the sentence. Place each word in the StringBuffer
// followed by a
// space. The StringBuffer will be automatically resized for
// each append.
```

```
                sent.append(inputWords[i]);
                sent.append(" ");
        }
        System.out.println (sent.toString());
```

10. Compile and run this class. The output should look like this:

```
...
These are words in a sentence
```

11. Now it's time to experiment with the `BigDecimal` and the `BigInteger` classes. Add these **bolded** Java statements to the beginning of your HelloWorld source:

```
import java.text.*;
import java.math.*;
import java.util.*;


//
//
// HelloWorld Application
//
//
```

12. Add these Java statements to the end of the previous statement (before the last two curly braces):

```
// Experiment with BigNumbers
// Construct some numbers to work with:
     BigDecimal accountBalance1 = new BigDecimal ("12345.678");
     BigDecimal accountBalance2 = new BigDecimal ("876.54321");
     BigDecimal accountBalance3 = new BigDecimal ("1");
     BigDecimal transactionAmount = new BigDecimal ("100.00");
// Add the transaction amount to the account balance.
     accountBalance1 = accountBalance1.add (transactionAmount);
     System.out.println ("accountBalance1 = " +
           accountBalance1);
// Compute the discounted amount of the transaction, and place
// into a new
// variable with the same scale as the transactionAmount
// variable.
```

```
     BigDecimal discountRate = new BigDecimal (".85321");
     BigDecimal discountedTransaction =
          transactionAmount.multiply (discountRate);
     System.out.println ("discountedTransaction = " +
          discountedTransaction);
// Subtract the discounted transaction evenly from accounts 2
// and 3.
     BigDecimal discountDistribution =
           discountedTransaction.divide (BigDecimal.valueOf(2),
// Make sure the result is the same scale as accountBalance.
           accountBalance1.scale(), BigDecimal.ROUND_HALF_EVEN);
     System.out.println ("discountDistribution = " +
           discountDistribution);
// No need to worry about scale or lost precision here;
// the scale of the result will be the greater of the two
// BigDecimal
// operands.
     accountBalance2 =
          accountBalance2.subtract (discountDistribution);
     accountBalance3 =
          accountBalance3.subtract (discountDistribution);
     System.out.println ("accountBalance2 = " +
          accountBalance2);
     System.out.println ("accountBalance3 = " +
          accountBalance3);
// Now you have to set the number's scale so that you can fit it
// into a
// print column. You want two decimal positions to be printed.
     BigDecimal printAccountBalance =
           accountBalance1.setScale (2,
           BigDecimal.ROUND_HALF_UP);
     System.out.println ("printAccountBalance = " +
           printAccountBalance);
```

13. Compile and run this class. Your output window should look like this:

```
...
These are words in a sentence
accountBalance1 = 12445.678
discountedTransaction = 85.3210000
discountDistribution = 42.660
accountBalance2 = 833.88321
accountBalance3 = -41.660
printAccountBalance = 12445.68
```

14. Let's use the `Scanner` class to convert a `String` to an `int`. Add these Java statements to the end of the previous statement (before the last two braces):

```
// Let's use the Scanner class to convert a String to an int
   inputMsg = "1003";
   int x = 0;
   Scanner s = new Scanner(inputMsg);
   x = s.nextInt();
   System.out.println("int from Scanner = " + x);
```

15. Compile and run this class. Your output window should look like this:

```
...
int from Scanner = 1003
```

Reviewing the Exercises

Let's review the samples you've created. Try to relate the sample source statements to the result (for example, the output) each statement creates. If necessary, rerun the samples, or look at the complete source code for this exercise on the CD-ROM. Feel free to experiment by yourself.

- Java strings are similar to items defined as PIC X in COBOL. Java strings contain characters and have an explicit length attribute.
- Java's `String` class contains many useful methods for manipulating string items. At the same time, Java's compiler sometimes treats string variables similarly to intrinsic data types.
- These Java statements:

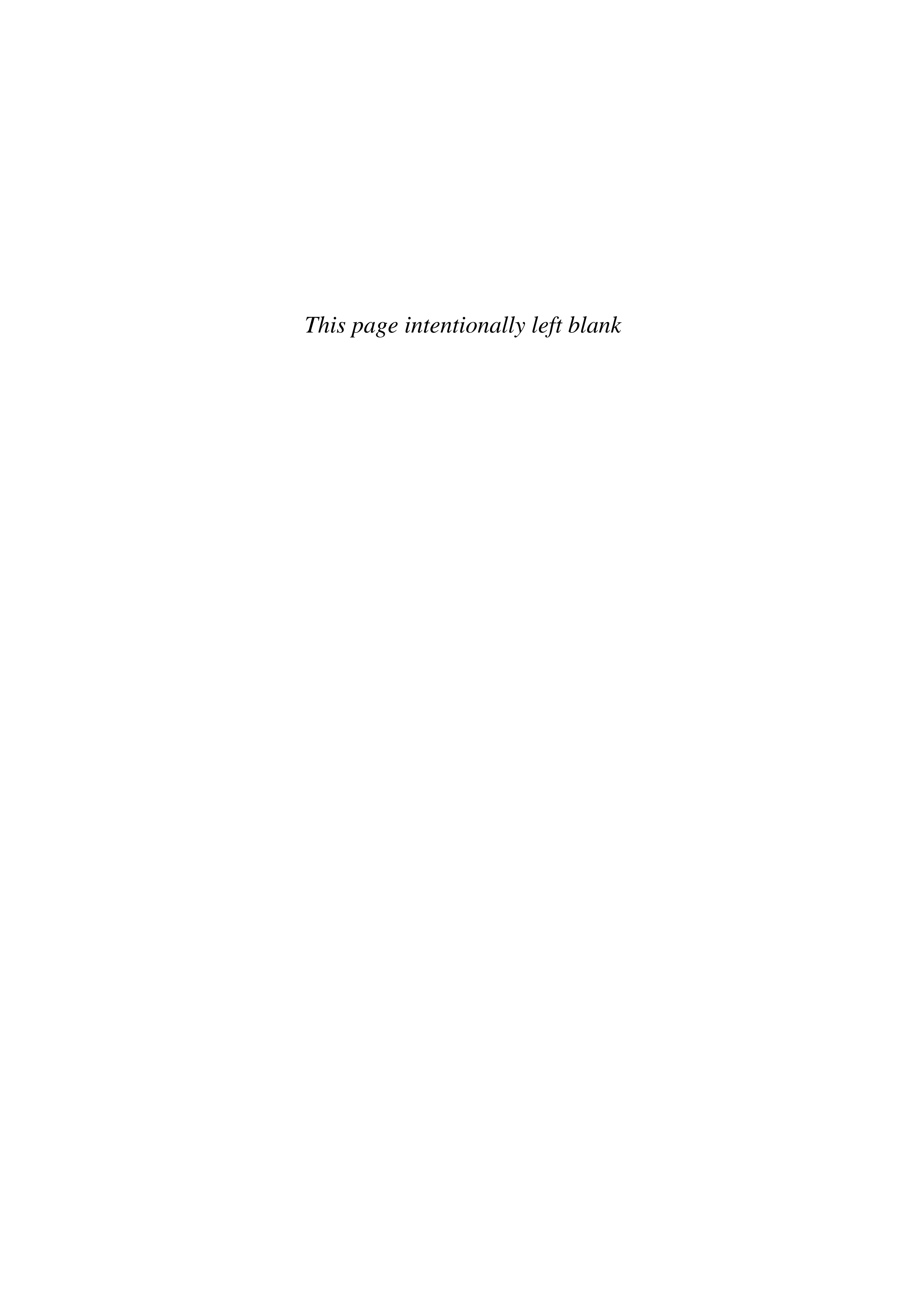
```
System.out.println ("compareTo = " +
     englishCaps.compareTo (temp));
System.out.println ("equals = " +
     englishCaps.equals (temp));
```

  call the `compareTo()` and then the `equals()` methods in the object `englishCaps`. In both cases, the called method is passed a string parameter (`temp`). The result of this method is printed using the `println()` method.

- Any Java type (including the primitive types such as integers, and boolean `true` or `false`) can be converted into a string. When the target of a math function (+ in this case) is a string, Java will automatically call the `toString()` method. These features are why the previous statements cause this output:

```
...
compareTo = 0
equals = true
```

- Java automatically converted the results of these methods (an integer result in one case, and a boolean result in the other) into strings suitable for the `println()` method.
- Java's numeric wrapper classes can contain numeric data types, such as `double` and `integer`. These wrapper classes have specific uses (generally when an object data type is required) but are not well suited for use as a general-purpose data store.
- Java's `BigNumber` classes (`BigInteger` and `BigDecimal`) are the appropriate choice when large, fixed-precision numbers are required. These classes provide the developer with unlimited precision capabilities and explicit control over scale and rounding. However, since they do not perform as well as the primitive data types and cannot use the standard Java math operators, developers must determine on a case-by-case basis if these data types should be used or if the primitive data type should be used.



This page intentionally left blank

9 Exceptions, Threads, and Garbage Collectors

In This Chapter

- Exception Class Hierarchy
- Creating Exceptions
- Using Exceptions
- Exception-Processing Suggestions
- Exception-Processing Summary
- Threads
- Inheriting from Thread
- Implementing Runnable
- Synchronization
- Benefits and Cautions
- Garbage Collection
- Exercises: Java's Exceptions and Threads
- Reviewing the Exercises

Proper error handling is a feature found in every well-designed system. A good system not only checks for and reports the obvious errors, but it also plans for and properly manages the unexpected errors. The general rules for good error handling in any language might be stated this way:

- Make sure to check for all possible error conditions.
- If your program can handle the error properly (for example, ask the operator for correction or assume a default processing condition), then do so.
- If your program cannot handle the error, then report the error to the calling program as an error condition as soon as possible.

- Include as much contextual information as possible (or practical) in the error. This likely means that the code that detected the error must provide this information.
- Serious processing errors, or unmanaged errors, should be detected as quickly as possible. In general, these types of errors should not allow processing to continue.
- Errors that cause abnormal termination should be logged to a file with complete contextual information for convenient postmortem analysis.

COBOL is very good at managing simple error conditions. For example, most COBOL programs validate the end user's input and report back any invalid input to the end user. However, processing-type errors (file I/O errors, for example) can be much more cumbersome. If you have been involved with any significant COBOL project, you know already how difficult good error handling can be in some cases. Let us explore the following example.

Suppose your system includes a currency rate calculation subroutine, and your program uses that subroutine to perform currency rate conversions. What should happen if you call the subroutine and that subroutine cannot read the record in the database required for proper currency rate conversion? Clearly, the answer depends on several factors, including whether your application is a batch or an interactive one, and whether or not your application can continue without the rate.

It is a complex task to design and document a currency conversion subroutine interface that supports each of these technical scenarios in addition to the business interface requirements. The technical interface requirements (such as batch versus online mode, calling program error processing capabilities, potential error conditions of the subroutine, and error details in message form) will likely overwhelm, or at least confuse, the business application interface requirements.

Java has the advantage of being designed more than 30 years after COBOL, and so its architects have come up with a built-in solution for this type of requirement. This solution is, naturally enough, based on the use of objects. Java defines a special object type called `Exception`. Classes can describe which exception objects they will create, and which ones they can handle. Moreover, the compiler checks class definitions to make sure that your classes contain appropriate logic for any declared exceptions.

One important design objective of Java exceptions is that standard or business logic should be separated from exception-processing logic as much as possible. At the same time, exceptions, when they do happen, should be dealt with as soon as possible and by the class that is closest to the problem.

EXCEPTION CLASS HIERARCHY

The basic class Java provides to assist in error processing is the `Exception` class. The Java `Exception` class inherits from the `Throwable` class. The `Throwable` class is also the superclass for the `Error` class used by the Java runtime processor to report serious runtime errors (you should not normally have occasion to use the `Error` class directly).

The `Exception` class comes with many prebuilt subclasses. Two subclasses of particular interest are `RuntimeException` and `IOException`. As a general rule, `RuntimeExceptions` are created as a result of logic errors in your program. Most often, effort should be spent in correcting and preventing this type of error instead of handling it. Most Java applications attempt only to manage the `IOException` class of errors, or some subclass (that is, specialization) of this error type. Figure 9.1 shows the class hierarchy for the standard `Exception` classes.

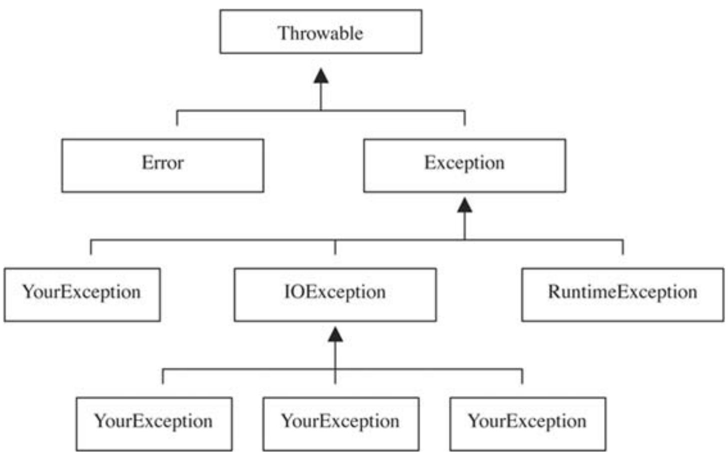


FIGURE 9.1
Exception class hierarchy.

CREATING EXCEPTIONS

A Java class can use any of the predefined `Exception` objects. These are listed in the SDK documentation set on the CD-ROM and are also available at www.java.sun.com.

If the standard exception definitions do not meet your requirements, you can create new ones. Simply extend either the `Exception` or (preferably) the `IOException` class.

Sun follows certain conventions for naming `Exception` classes, and you should follow this convention as well. In particular, classes that inherit from `Exception` or

RuntimeException should end with Exception. Also, quite a number of standard Exception classes are available on the Internet, so always be sure to check if an Exception class exists before making your own.

This exception will be used when no database is available for the application to connect to:

```
class NoDatabaseAvailableException extends IOException {
      ...
}
```

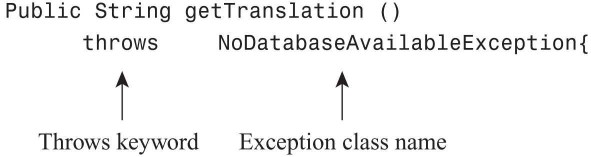
By convention, Exception classes define a constructor that accepts a String parameter. This allows the class that created the exception to place some contextual information in the Exception. In addition, Exception classes normally support a method named getMessage(). This method returns a string that contains the contextual information and perhaps more detailed information about the error condition.

```
class NoDatabaseAvailableException extends IOException {

      public NoDatabaseAvailableException (String message) {
             super (message);
      }
...

      public String getMessage () {
             return "Unable to access the database for" +
             super.getMessage ();
      }

}
```

Your new exception is now ready for use.

You can envision an Exception object as an optional return value from a method. You already know that methods can return values as defined in the method interface. In addition, a method can create an Exception object and return this object instead of the normal value.

A Java method that may create an exception must publish that property as part of its interface definition. The syntax for this is as follows:



A class defined in this manner may return a `NoDatabaseAvailableException` instead of the standard `String` return value. Suppose you want the `getTranslation()` method to return with an `Exception` if a translation is requested and no database is available to retrieve the translation. In addition, you would like to return with the current language code and the text of the original message. Based on these requirements, the conversion class throws an exception with the following syntax:

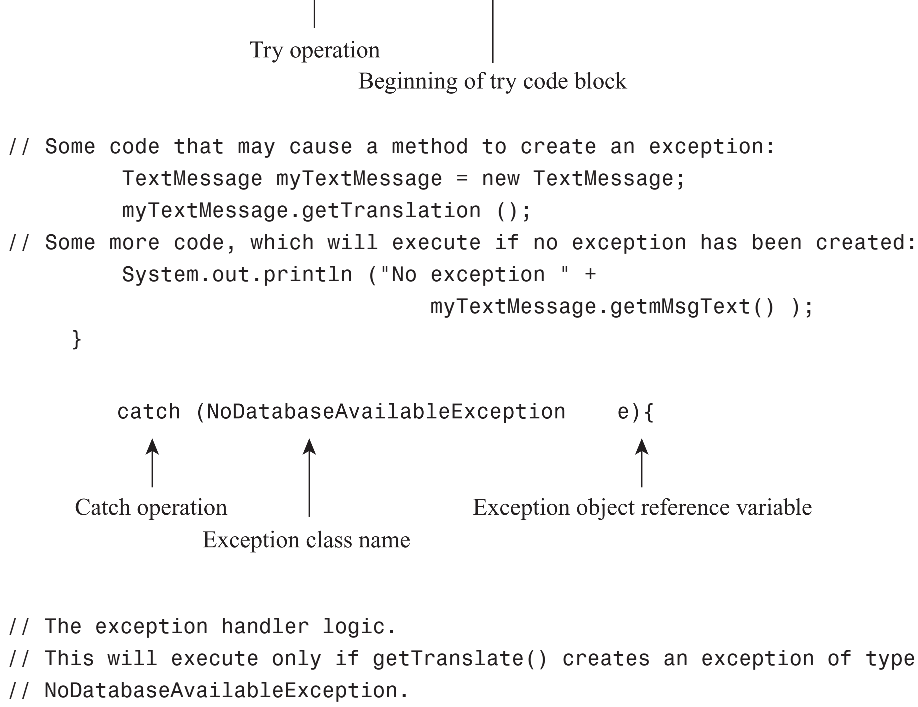
```
if (database.connect () = null) {
      throw new NoDatabaseAvailableException ("LANGUAGECODE = " +
         LANGUAGECODE + " text = " + msgText);
}
```

When the Java runtime processes the `throw` statement, the method exits immediately to the calling class. Any statements after the `throw` statement are not executed. This is roughly analogous to the following COBOL syntax. (It is assumed that the required data elements are defined in the subroutine's CONTROL item, and that these items begin with the prefix SUB-.)

```
IF DATABASE-NOT-CONNECTED
        STRING "LANGUAGECODE = ", SUB-LANGUAGE-CODE,
        " text = ", SUB-MESSAGE-TEXT DELIMITED BY SIZE INTO
        SUB-ERROR-MSG
        SET SUB-NO-DATABASE-AVAILABLE TO TRUE
        PERFORM EXIT-PROGRAM.
...
 EXIT-PROGRAM.
       EXIT PROGRAM.
```

Using Exceptions

Now, for the last piece of the puzzle. If a method can throw an exception, then any classes that call this method must handle the potential exception(s) in some fashion. A class handles an exception from a method by using the `try...catch` construct:




```
try            {
// Some code that may cause a method to create an exception:
         TextMessage myTextMessage = new TextMessage;
         myTextMessage.getTranslation ();
// Some more code, which will execute if no exception has been created:
         System.out.println ("No exception " +
                                 myTextMessage.getmMsgText() );
     }

     catch (NoDatabaseAvailableException    e){

// The exception handler logic.
// This will execute only if getTranslate() creates an exception of type
// NoDatabaseAvailableException.
          System.out.println (e.getMessage() );
    }
```

Statements inside the `try` code block will stop executing as soon as an exception is created. If no exceptions are created, or if your class has caught and handled the exception, your class will continue with the statement after the `try...catch` code block.

Your class can handle multiple exceptions that may be created by statements in the `try` code block simply by adding an additional `catch` code block after some other `catch` code block.

```
     try            {
          ...
     }
     catch (NoDatabaseAvailableException    e){
          ...
     }

     catch (IOException    e){
// The exception handler logic for all other I/O conditions.
```

```
            System.out.println (e.getMessage() );

        }
```

Notice that the more specific exception is caught first, then the generic exception. Be sure to execute the logic for the more specific class (the subclass) before the more generic superclass. If you reverse the order, your more specific logic won't be executed.

Once in a while, you may need to catch an exception and perform some logic based on the exception, but still not completely manage the primary reason for the exception condition. For example, suppose that your `TextMessage` really needs to be translated before the application can continue. In this case, you should certainly write the exception message to a file (to the standard output device in the example), but in addition, you may want to inform the end user. To meet these requirements, you can *rethrow* the exception and allow the calling class to complete the error-handling logic.

```
        catch (NoDatabaseAvailableException    e){
    // The exception handler logic.
    // This will execute only if getTranslate() creates an exception of type
    // NoDatabaseAvailableException.
             System.out.println (e.getMessage() );
    // Pass the exception to the calling program.
             throw e;
        }
```

The last `Exception` structure is the `finally` operator. This structure provides a mechanism to define statements that should execute even if an exception occurs. Statements in a `finally` code block will execute after the `try` block, even if an exception is caught. So, you can place logic that is always required in the `finally` block, and it will always be executed.

The `finally` block is often used to make sure external resources, such as a database connection or an open file handle, are in a properly managed condition at all times.

For example, the `getTranslate()` function in `TextMessage` reads a database to get the translated error text. Suppose that in order to read the database, you need to get a database connection from a pool of database connections. If you have any problems reading the database in `getTranslate()`, you can throw the `NoDatabaseAvailableException`. However, you still have to manage the connection and return it to the pool.

```
getTranslate (String message) {
     try           {
           // Get a database connection from the connectionManager pool.
           dbConnection = connectionManager.getConnection();
           ...
           // use dbConnection to read the database.
           // This can cause an Exception to be thrown
           messageFound = dbConnection.read();
           ...
     }
     catch (IOException    e){
           // Build a NoDatabaseAvailableException.
           // Add some context information, such as the message
           // you are trying to translate, the language code
           // and the text from the current exception.
           NoDatabaseAvailableException   noDB = new
NoDatabaseAvailableException
                                     (message + languageCode + e.toString());
           throw .noDB;
     }

     finally {
// This statement will always be executed.
           //  Make sure the dbConnection is returned to the
           //  connectionManager pool.
           connectionManager.releaseConnection(dbConnection);
     }
}
```

This `finally` code block will always be performed. It holds for the following conditions:

- The `try` block completes normally.
- The `try` block explicitly throws an exception that is caught in a related `catch` block.
- The `try` block explicitly throws an exception that is not caught in a related `catch` block.
- The `try` block executes a return statement.
- The `try` block causes an exception to be thrown in some other class, which is caught in a related `catch` block.
- The `try` block causes an exception to be thrown in some other class, which is not caught in a related `catch` block.

This approach gives the developer complete control over exception processing, yet it still provides excellent default exception-processing mechanisms. A class can demand that control be returned to it whenever exceptions happen through the use of the `try...catch...finally` structure. The runtime will first perform the `try` function, and then perform the `catch` function for matching exceptions. In any event, the `finally` function will always be performed. Even if an exception is thrown by the method, the runtime will execute the `finally` code block before it passes control back to any calling class.

A class may not be able to handle the exceptions created by classes that it uses. Or the class may only partially handle the exception. In these cases, the class must pass the exception along to its caller, which must in turn either handle it or pass the exception along as well.

The syntax for defining which exceptions will be passed along is also `throws`. That is, if a class B uses a class C, and C throws an exception, the consumer class B does not necessarily need to handle the error. It can simply define that exception in its own exception list. At runtime, the exception created by C will be automatically passed to the caller of B, as if B created that exception.

```
// This class does not handle the NoDatabaseAvailableException exception.
// Instead, it is listed as a possible exception in one of its methods.
     public class TextMessageConsumer
          public void someMethod ()
          throws NoDatabaseAvailableException {
          ...
// Some code that may cause a class to create an exception.
// Note that these statements are not in a try...catch code block.
     TextMessage myTextMessage = new TextMessage;
     myTextMessage.getTranslation ();
```

In this example, the `TextMessageConsumer` class does not explicitly handle the `NoDatabaseAvailableException` in a `try...catch` code block. If this exception were to occur, the caller of `TextMessageConsumer.someMethod()` would receive the exception, as if the method `TextMessageConsumer.someMethod()` had created it.

To review, here is the Java specification for exception processing. Suppose that class C throws some exception. Any Java class B that creates an instance of class C must explicitly handle the exception, either by catching it or by throwing it to the class that created B. The compiler checks this for you. As a result, you know at compile time that a class you are using may create an error condition, and your class must either handle the error or pass control along to a class that can.

Exception-Processing Suggestions

Exceptions are a very handy way to describe error conditions and ensure that they are properly managed. As is always true, too much of a good thing is not necessarily a better thing.

One concern is that exception processing is much more expensive than is simple testing of a value. It is generally better (performance-wise) to test for an error condition if you can, instead of creating and catching an exception. As a rule, exceptions should be reserved for unusual error conditions and not as a programming technique to test for anticipated conditions.

For example, the `getTranslate()` method can throw a `NoDatabaseAvailableException` if no database is available. However, if you simply cannot find the translated text in the database, you should not treat that condition as an exception. The method `getTranslate()` should handle this condition in some fashion, perhaps by logging the condition to an audit file and returning the original text as the translated text.

Another concern is that a class that throws an exception forces its subclasses and its consumer classes to handle the exception. A class designer should define and throw exceptions only when it makes sense for a class consumer to be aware of the error condition. Of course, if you define a class that uses other classes, and these other classes throw exceptions, you may have to throw them in the classes you build if your class cannot handle the exception internally. This situation only highlights the potential problems created by the aggressive use of exceptions.

Another related suggestion: Make sure that your base classes that will throw exceptions are defined that way early in the development process. It is extremely frustrating to define a new exception in a low-level class, and then have to edit and recompile all the classes that use this class. You will need to recompile because every class that calls a method that can throw an exception must either handle the exception or throw it.

Make sure that your class is in a proper state if you throw or catch an exception. Since a `throw` statement effectively acts as a `goto` statement and abruptly interrupts the normal sequence of statements in your class, it is possible that some variable or object in your object will be in an invalid state the next time your object is accessed. If necessary, place appropriate code in the `finally` code block to ensure that your object is in good shape for the next call.

Finally, group your exception processing into larger `try...catch` code blocks. For example, don't insert a `try...catch` block around every statement that might create an exception. If a section of code might create several exceptions, it is very likely that the exception-handling process is the same for each exception. If this is the case, group the statements into a single, larger `try...catch` code block.

Another important consideration is defining which exception types to worry about. Many standard Java statements can actually throw a `RuntimeException`. As a matter of standard practice, you should ignore those types of exceptions in your code. If they happen, something pretty serious is amiss, and your program will terminate with helpful context information, such as the stack trace and what line in your source code likely caused the error.

However, if you build or use a class that throws a type of `IOException`, then those types should be managed in some manner by your program.

Exception-Processing Summary

As you can see, Java's implementation of exception processing helps complex systems manage error conditions in a standard and predictable manner. The requirements of good error processing (as defined in the beginning of the chapter) are encapsulated in the language definition. Classes can define both their return values and their error conditions. Consumer classes must explicitly define how they wish to handle these error conditions, either with `try...catch` code blocks or by rethrowing the exception. The `finally` statement ensures that appropriate state-management functions will be executed, even if exceptions occur. Best of all, the compiler understands this definition and checks to make sure you are following the exception-processing rules.

Threads

All modern computer operating systems are multitasking systems. A multitasking system is one that can perform several tasks simultaneously. The operating system task manager shares computer resources (disk, memory, I/O paths, but especially the CPU) between the various jobs running on the system. Robust operating systems are preemptive multitasking systems, meaning that the task manager will make sure that no single task hogs a resource and causes other tasks to wait.

Historically, an operating system manages its jobs at the process level. Operating system processes are self-contained execution units (as defined by the operating system). The operating system creates, schedules, and provides system resources to processes. Most programming languages (including COBOL and Java) construct a program that will execute as a process in the host operating system. Inside that process, there will be memory space managed by the operating system (kernel space), and memory space that is specific to the process (user space).

Inside a traditional process, there is only one thread of execution, that is, only one statement can be performed at a time. When a statement is complete, the next statement executes. The developer can control the sequence of statements, but there is no way for the developer to have two statements in a program execute at the same time.

In contrast, Java defines a multithreaded execution model. This model allows any number of instruction streams or lines of execution inside a single process to execute simultaneously. That is, your program can have many threads of execution accomplishing work at the same time. A thread is any asynchronous subprocess inside your main program process.

Java threads allow you to construct applications that do not wait for a function to complete before executing another function. Perhaps your program contains an `Account` object, which reads transactions from a database based on an end user's request. At the same time that this function is executing, you may need to perform an `interrupt()` method on a user interface class, which checks to see if the user wants to cancel the request. Without simultaneous threads, you would have to wait for the `Account` object to complete its work before you could check the user interface class. As a result, the user would not be able to cancel the request until the request was complete!

Inheriting from Thread

Java provides two mechanisms to create new threads. The simplest mechanism is to inherit from the `Thread` class. This class contains methods that need to be overridden in order to make your `Thread` class offer functionality. You need to override the `run()` method and place your execution logic in here. The statements in the `run()` method will execute simultaneously with other statements in your program. The following statements define a class with a `run()` method that will print out a message:

```
        public class MsgThread extends Thread {
             public MsgThread () {
                  super ();
             }

             public void run () {
                       for (;;) {
   // Create an infinite loop.
                       System.out.println ("Inside a thread");
                       }
                  }
             }
```

In some other class, you can create several instances of these classes.

```
MsgThread run1 = new MsgThread ();
MsgThread run2 = new MsgThread ();
```

The threads will not execute yet. In order to get the logic in the `Thread` classes going, you have to call their `run()` method. You do this (indirectly) by calling its `start()` method.

```
run1.start();
run2.start();
```

Now both of these threads will execute simultaneously (and endlessly). What if you want to stop an executing thread before it completes? Generally, you should define a *stop processing* variable and have your code periodically check that variable. For example, suppose you've started a thread, and then want to stop that thread and log this event.

```
    MsgThread run1 = new MsgThread ();
    run1.start();
//  Perform some other logic, then stop the thread.
  ...
    run1.stopThread();

    System.out.println ("Thread cancelled" + run1);
```

Next, you define a method named `stopThread` in `MessageThread`. This method sets the flag `timeToStop`, which will cause the thread to stop. Notice that `timeToStop` is defined as `volatile`. This tells the runtime to disable certain optimizations for this variable, ensuring that when this flag is set to `true` in `stopThread()`, that value is seen right away in the `run()` loop.

```
public class MsgThread extends Thread {
     private volatile Boolean timeToStop = false;
     public MsgThread () {
          super ();
     }

     public void stopThread () {
          timeToStop = true;
    }
```

Finally, you change the `run()` loop in `MsgThread` to check the value of `timeToStop`. You exit the `run` method and stop the thread when that value is true.

```
          public void run () {
  // Create a loop that runs till timeToStop.
               while (!timeToStop) {
                    System.out.println ("Inside a thread");
               }
               System.out.println ("Exit a thread");
          }
     }
```

Implementing Runnable

One obvious limitation of the first thread-management technique is the fact that your class has to inherit from the `Thread` class. What if you need to build a threaded class, but it must inherit from some other class? In order to support this requirement, Java provides another mechanism to create in-process threads.

The second approach is to have your class implement the `Runnable` interface. The class that you want to run as a thread needs to define the `run()` method from this interface. You also should define a `stopThread()` method that can be used to stop your class.

```
     public class MsgThreadRunnable
          implements Runnable {
          private volatile Boolean timeToStop = false;
          public void stop Thread() {
               timeToStop = true;
         }

        public void run () {
  // Create a loop that runs till timeToStop.
               while (!timeToStop) {
                    System.out.println ("Inside a thread");
               }
               System.out.println ("Exit a thread");
          }
     }
```

As a last step, you have to create a new `MsgThreadRunnable` object and pass your object to the `Thread` constructor. You then use the `Thread` object's `start()` method to start your thread as follows:

```
      MsgThreadRunnable msg1 = new  MsgThreadRunnable ();
      MsgThreadRunnable msg2 = new  MsgThreadRunnable ();
      Thread run1 = new Thread (msg1);
      Thread run2 = new Thread (msg2);
      run1.start();
      run2.start();
//  Perform some other logic, then stop the threads.
  ...
      msg1.stopThread();
      msg2.stopThread();
```

Synchronization

Java threads execute inside a single process (in this case the JVM) and, therefore, share resources within the JVM. Threads in a process normally share the same class instances and address space. Therefore, all object data members are shared between the various threads.

Since Java threads share resources inside the process, the Java program itself has to manage resource utilization conflicts. For example, it would ordinarily be inappropriate for a function to increment a variable while some other function resets that variable to zero.

The solution for this problem is to *synchronize* your methods or objects. When you synchronize a method, it waits for all other currently executing instances of this object's method to complete. When you synchronize on an object, Java makes sure that only one thread at a time modifies that object.

Java's synchronization coordinates access at the object level (either on a `java.lang.Object` or on the `java.lang.Class` object in the case of class methods). This means that two distinct object instances of the same class are not coordinated. Static members can be synchronized, but they do not block access to synchronized object instances. To protect a critical method against concurrent access, you can use the `synchronized` keyword:

```
public class ErrorMsg {
     public synchronized String getErrorMsg () {
     }
}
```

In this case, only one thread at a time in a process can call the `getErrorMsg()` method in a particular instance of the `ErrorMsg` class. If other threads attempt to call `getErrorMsg()` for a particular instance of `ErrorMsg`, the second thread will wait.

Another way to coordinate activities among threads is to define a synchronization block of code. This is a block of code that has been explicitly marked to synchronize on any object or static class member.

```
        public class ErrorMsg {
             DatabaseConnection myDB = new DatabaseConnection ()

             public String getErrorMsg () {
                  if (ErrorMsgnotRead) {
                       synchronized (myDB.CONNECTION_SYNCH) {

   // Read the database, and make sure only one instance of DatabaseConnection
   // is doing it at a time. The CONNECTION_SYNCH static class member is
   // used to synchronize all instances of DatabaseConnection.
                       ...
                       }
                  }
             }
        }
```

Of course, it is up to the developer to make sure that all relevant access to this object is also synchronized. Sometimes this technique is referred to as manual synchronization.

Java 1.5 adds some additional thread management tools in the concurrency utility library. One popular tool is the *semaphore*. Using a semaphore, a developer can control access to any arbitrary block of code or resource. Think of a semaphore as a type of check out/check in control applied to a defined set of permits. A thread can ask for a permit (for example, permission to use a resource) using `acquire()`. The thread will block until a permit is available. When a thread is done with the permit, it returns it to the permission pool using `release()`. When a permission pool has only one permit available, a semaphore can be used as a mutual exclusion locking mechanism.

For example, a semaphore can be used to synchronize access to the output stream (typically the console). A thread can ask for access to the console by calling the `acquire()` or `acquireUninterruptibly()` method of an agreed-upon semaphore object. As long as all threads use this same object, only one thread at a time will get control to that object. The others will wait until the thread with control calls the `release()` method.

Benefits and Cautions

Java's ability to create and execute multiple threads allows the developer to build responsive, powerful applications. In particular, applications that perform significant work but still need to respond to user input are good candidates for a multi-threaded design. Another good nominee is an application that needs to respond to end-user input from multiple interactive dialog boxes (the search utility in a word processor, for example).

Other possibilities include applications that contain functions that can be processed in parallel. In today's environment, production systems often have multiple CPUs. Breaking up your task into multiple functions that can run simultaneously allows the system to assign your work to multiple CPUs. The result will likely be more work done in less time.

Some parallel functions are obvious (perhaps you have a data migration application, and it can be structured to extract data from multiple tables simultaneously). Other times the opportunities require a little more thought (how can I break up a single table extract function into multiple steps?). In any event, the effective use of threads can result in more efficient, more responsive applications.

Finally, some environments effectively define the threading model for you. For example, if you are using a JEE application server (discussed in Chapter 15), it is likely that the code you write will execute in a multi-threaded environment.

Still, it is important not to treat threads like a shiny new tool that should be used with every project. As with any good thing, it is possible to get carried away with using threads.

- **The use of multiple threads in an application can be a very difficult programming model.** It is up to you (the developer) to anticipate and prevent resource conflicts. In many cases, you even need to manage (schedule, start, stop, and synchronize) threads so that they execute when required. Furthermore, an application that contains a shared resource problem (for example, one thread updates a counter, while another thread decrements it) is a very difficult application to debug. In fact, the debugger can affect the symptoms of the problem you are analyzing, since it runs in a thread of its own.
- **Thread implementations vary significantly across operating systems and virtual machines.** Some VMs use the native operating system thread mechanism, whereas others create their own thread management logic in the VM. In the latter case, multi-threaded performance is likely to be less than what you would expect.
- **Don't recreate an operating system using threads.** It is unlikely that an average developer can write process-management and context-switching logic that is more effective than the process-management and context-switching logic built into the operating system. The major advantage of threads is that the developer

has more intimate knowledge of the application implementation. Therefore, the developer can identify which parts of the application can be shared between threads and which should be isolated into distinct threads. But if your application is actually a collection of individual functions that have little in common, you are probably better off implementing this application in separate processes for each function and allowing the operating system to manage them for you.

- **Make sure the algorithm behind the application is multithreaded.** It doesn't do a lot of good to create a thread and then immediately wait for it to complete. For example, if your application needs to collect all the required transactions from the database before it can analyze them, you should probably not create the transaction collection function as a thread. On the other hand, if you can start to analyze the transactions as they are collected, it might be appropriate to implement the collection function as a separate thread from the analysis function.

Garbage Collection

At any construction site, there are builders who build the structures as well as garbage collectors. These people (or systems) move around the site and remove any unused materials and equipment. They are important contributors to the efficient and effective operation of the site.

Java's runtime environment works pretty much the same way. Constructors create new objects, and a built-in garbage collection system removes these objects when they are no longer needed. Java determines whether an object is needed using a complex (and ever-improving) algorithm, but the most important attribute of an object (at least from the garbage collector's perspective) is whether there are any current references to this object. If an object exists in a Java runtime environment but is not referenced by any other object, it will likely be garbage collected, that is, removed from the memory of the currently executing program.

Java's runtime system includes a garbage collector thread. This low-priority system thread runs periodically, scanning the modifiable portion of memory, in order to detect any objects that are not currently referenced. These are marked for deletion (actually they are identified as not referenced by the scanning process). These objects are subsequently deleted by the garbage-collection process, and the memory associated with them is made available for other objects.

When an object is deleted, it may be appropriate for that object to first clean up certain resources before it is deleted. For example, an object that contains a database connection should probably close that connection as the last thing it does before it is deleted.

An object that needs to clean up resources as it is deleted can declare a `finalize()` method (in reality it needs to override the `finalize()` method in the base `java.lang.Object` class). The garbage collector will call this method before it deletes the object.

By default, there is no guarantee that the `finalize` method will be called at all! The Java runtime process could exit without first destroying this object. The only guarantee is that if the object is about to be garbage collected, then this method will be called first. There is no guarantee that this object will be garbage collected before the Java runtime process exits. There is also no guarantee on which thread will call `finalize()`.

In the database connection example, the runtime process may terminate before the object's `finalize()` method is called and the database connection is properly closed. The database engine will likely detect this event as an application failure and will roll back any incomplete transactions.

Keep this behavior in mind, and code `finalize` methods defensively. That is, you should only place process-level housekeeping or clean up code (such as memory management functions) in `finalize` methods. Do not assume that any code in a `finalize` method will always or ever be performed.

You can also force the garbage collector to run by explicitly calling the `System.gc()` method.

And as a last issue to be aware of, `finalize()` methods do not automatically call `finalize()` methods in a class's superclass. This is unlike the behavior of constructors, where the superclass's constructor is automatically called. Nested `finalize()` calls must be explicitly coded, as shown here:

```
protected void finalize() throws Throwable {
  super.finalize();
}
```

Exercises: Java's Exceptions and Threads

In this exercise, you are going to create some threads that simply display some messages, sleep, and eventually complete execution.

1. Using a text editor, create a file named MsgThread.java. Add these lines:

```
public class MsgThread extends Thread {
     private volatile Boolean timeToStop = false;
     public MsgThread () {
```

```
                    super ();
               }

               public void stopThread () {
                    timeToStop = true;
              }

               public void run () {
                    while (!timeToStop) {
// Create a loop that runs till timeToStop.
                         System.out.println ("Inside thread " +
                           Thread.currentThread().getName());
                    }
                    System.out.println ("This thread is stopping " +
Thread.currentThread().getName());
               }
          }
```

2. Compile the class in the DOS command window:

```
➔  javac MsgThread.java
```

3. Using a text editor, create a file named MsgThreadRunnable.java. Add these lines:

```
          public class MsgThreadRunnable
               implements Runnable {
               private volatile Boolean timeToStop = false;
               public void stopThread () {
                   timeToStop = true;
                }

             public void run () {
// Create a loop that runs till timeToStop.
                    while (!timeToStop) {
                         System.out.println ("Inside thread " +
                           Thread.currentThread().getName());
                    }
                    System.out.println ("This thread is stopping " +
Thread.currentThread().getName());
               }
          }
```

4. Compile the class in the DOS command window:

   ```
   ➔ javac MsgThreadRunnable.java
   ```

5. Edit the HelloWorld.java source file in your java4cobol directory with a text editor. Delete the code from the `main()` method but not the `main()` method itself.
6. Next, add the following code to create and then use `MsgThread` and `MsgThreadRunnable`.

```
import java.util.*;
//
//
// HelloWorld
//
//

public class HelloWorld
{
         public static void main(String args[]) {

//  Create an object that inherits from Thread,
//   and then start that Thread.
              MsgThread run1 = new MsgThread ();
              run1.start();

//  Create an object that implements Runnable,
//   and then start the execution of that class.
              MsgThreadRunnable msg1 = new MsgThreadRunnable ();
         Thread run2 = new Thread (msg1);
         run2.start();

//  Wait a few milliseconds, then stop those threads.
         try {
             Thread.currentThread().sleep(100);
         }
         catch (InterruptedException e) {
         }

//  Use the Thread object in this case.
         run1.stopThread();
```

```
//  Use the MsgThreadRunnable object in this case.
         msg1.stopThread();
         System.out.println ("Thread 1 cancelled" + run1);
         System.out.println ("Thread 2 cancelled" + run2);

     }

}
```

7. After compiling the program, you can execute it. Your results should look like this:

```
Inside thread Thread-1
Inside thread Thread-0
Inside thread Thread-1
Inside thread Thread-1
Inside thread Thread-0
...
```

This will be followed by multiple displays of the thread execution messages. It will end with the completion messages:

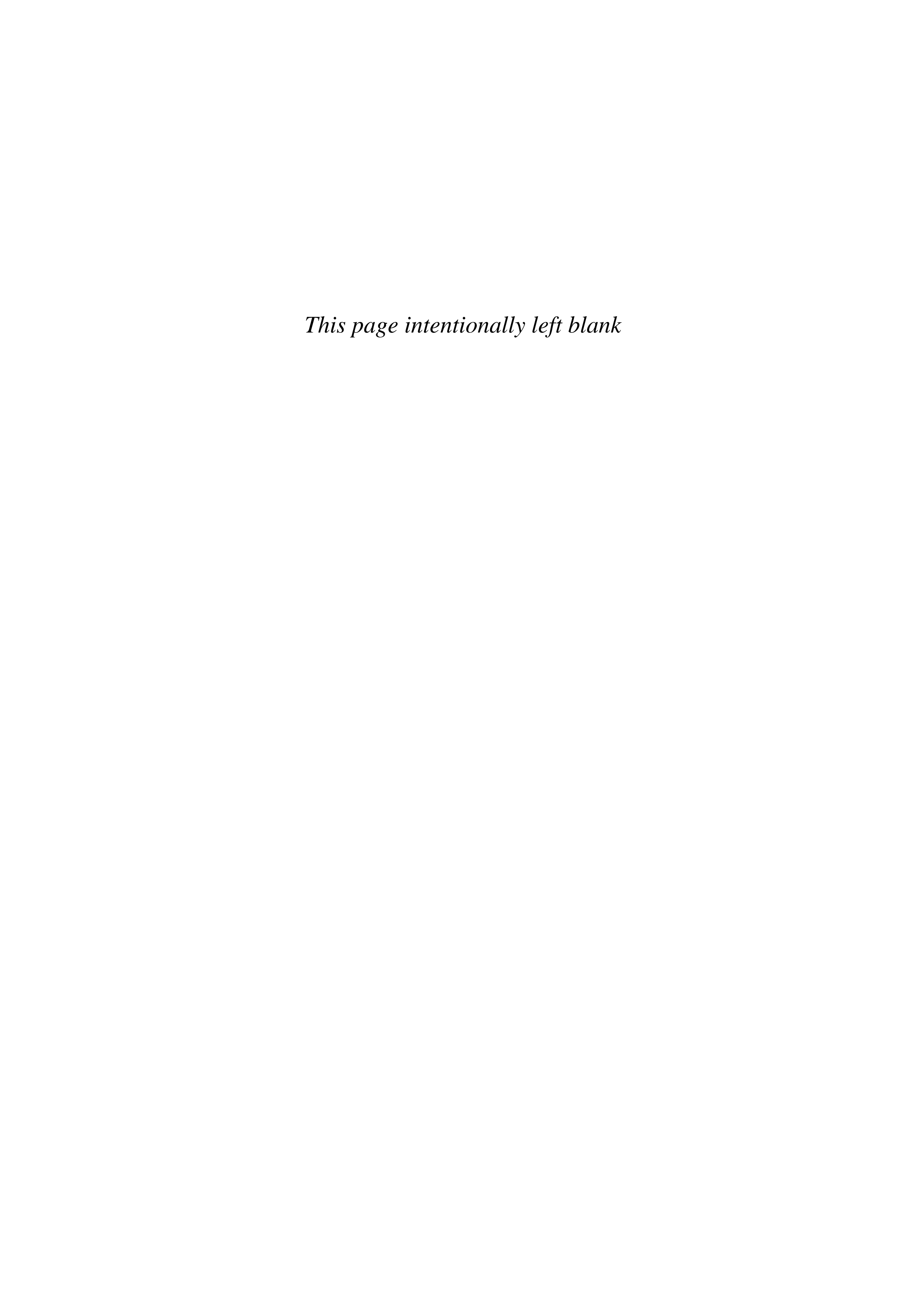
```
This thread is stopping Thread-0
Thread 1 cancelledThread[Thread-0,5,main]
Thread 2 cancelledThread[Thread-1,5,main]
This thread is stopping Thread-1
```

Reviewing the Exercises

Let's review the samples you've created. Feel free to experiment by yourself.

- Threads can be created either by extending `Thread` or implementing the `Runnable` interface. In the example, you used a class (`MsgThread`) that inherits from `Thread` and another class (`MsgThreadRunnable`) that implements the `Runnable` interface.
- The useful work in these two classes is performed in the `run()` method. In the examples, you simply write to `System.out`, but in practice, a thread will perform some useful task, such as printing a report or waiting for a user response.
- The threads are stopped when the main class (HelloWorld) calls the `stopThread()` method.

- The static method `currentThread()` always refers to the current thread at whatever point it is called. Each thread has a name assigned by the VM, but that name can be overridden programmatically.
- The `sleep()` method causes a thread to stop running and yields control to other threads that might be able to execute.
- The `sleep()` method can throw an `InterruptedException`, so code that includes it must be wrapped in a `try...catch` block.



This page intentionally left blank

10 I/O in Java

In This Chapter

- Streams vs. Record-Based I/O
- The File Class
- InputStream and OutputStream
- Serialization
- Readers and Writers
- RandomAccessFile
- Exercises
- Reviewing the Exercises

Input and output usually play a big role in COBOL batch programs, especially in legacy systems. I/O operations are considerably different in Java than they are in COBOL, yet it is possible to do most of the same things in Java as you would in COBOL. If you find yourself working on a Java enterprise application, you will be doing very little traditional file I/O because such systems are usually built utilizing database technologies. Most Java applications use database technologies even when they are single-user systems. Nevertheless, I/O plays a much broader role in Java than simple file operations, and an understanding of how it works in Java is essential to becoming a complete Java programmer.

STREAMS VS. RECORD-BASED I/O

Input and output operations in Java are implemented in the java.io package, or in the newer java.nio package. Input and output operations are considerably different in Java than they are in COBOL. Any experience in C/C++ translates easily into Java. But if you don't have any exposure to C/C++, don't worry.

COBOL I/O is usually file- and record-based. You relate the file to its storage medium through the SELECT and ASSIGN statements in the INPUT-OUTPUT SECTION of the ENVIRONMENT DIVISION. You define the record in the FILE SECTION of the DATA DIVISION. You write code to OPEN the file, READ or WRITE its records, and CLOSE the file when you are finished processing it. For example:

```
SELECT ACCOUNT-FILE ASSIGN TO ACCOUNTS
ORGANIZATION IS SEQUENTIAL.

FD  ACCOUNT-FILE.
01    ACCOUNT-RECORD.
         05  ACCOUNT-NUMBER               PIC 9(10).
         05  ACCOUNT-NAME                 PIC X(25).

         OPEN INPUT ACCOUNT-FILE.
         READ ACCOUNT-FILE
              AT END GO TO CLOSE-FILE.
         CLOSE ACCOUNT-FILE.
```

In many environments in which COBOL runs, the operating system maintains information about the file and its characteristics. At runtime, the operating system relates the external name of the file to an actual file on physical media. When the program attempts to open the file, COBOL and I/O subsystems of the operating system compare the definition of the file, as presented in the program with the actual file. If the actual file in the example had 100-byte records instead of the 35-byte record you defined, you would get an error and the program would abort. Java and the Java VM (JVM) do not perform this type of cross-checking between the characteristics of the file as defined to the operating system and the program.

Input and output in Java is stream-based. A stream is simply a flow of bytes. The flow may be to or from a file on disk, a network connection, an array of bytes in memory, from standard in/standard out, or even another program.

Stream-based I/O has the advantage that the same methods of reading and writing data can be used with independence from (even ignorance of) whatever is creating or using the data that offers application programmers a convenient abstraction. For example, you have been using the `System.out.println()` method extensively for displaying information on the console. The "out" in `System.out` is

really a PrintStream (more about PrintStreams later). The System class also has a method setOut(PrintStream out). You could direct all of the output going to the console to a file simply by creating a new PrintStream from a FileOutputStream and setting the System.out to use the new PrintStream. In other words, System.out is simply a PrintStream that, by default, goes to the console. When you code System.out.println(), however, the line will be output to wherever the PrintStream is directed, whether it be a console, a file, or even another program that might be written to capture console output.

Stream-based I/O has the disadvantage that it has no concept of a record apart from the code that is processing the data.

Let's assume for the moment that you wish to process the same Account file with Java that you processed in the preceding example with COBOL. You have taken the file from wherever it existed and brought it to the local machine with the name account.dat. For simplicity, let's assume that the data is stored as ASCII and no carriage return or line feeds have been added to the file. The file still consists of the fixed 35-byte records. To process this file with Java, you would use the following code:

```
// Create a stream related to a disk file.
FileInputStream instrm = new FileInputStream("account.dat");
// Allocate a byte array to hold each record.
byte [] accountRecord = new byte[35];
// Read the first record.
int bytesRead = instrm.read(accountRecord);
// Loop until end of file when bytesRead = –1.
while (bytesRead != -1) {
// Process each record and read the next record.
String accountNumber = new String(accountRecord, 0, 10);
String accountName = new String(accountRecord, 10, 25);
bytesRead = instrm.read(accountRecord);
}

// Close the stream and file.
instrm.close();
```

You opened the stream implicitly when you instantiated FileInputStream. Neither Java nor the operating system checked to make certain account.dat is a sequential file of fixed-length, with 35-byte records. If the file contained 34-byte records, you would get no error. The read() method would dutifully return 35 bytes of data from the file, and you would process the file incorrectly unless you incorporated additional error-checking logic into the program. Additionally, since

Java has no concept of a record, you have to break the record apart into its component fields. The `String` constructor that converts portions of a byte array into a string illustrates one method of doing this.

THE FILE CLASS

The `File` class is an abstraction of an entry in the file system of the underlying operating system. A `File` instance can represent either a file or a directory. The `File` has many useful methods for inquiring about and manipulating the file system.

Here are some valid file names using the various conventions:

```
C:\documents\mybook\chapterone.doc (Windows)
/home/account/documents/mybook/chapterone.doc (Unix)
\\mymachine\myshared_drive\documents\mybook\chapterone.doc (UNC under Windows)
```

Java and the JVM often will deal with these differences as transparently as possible. With most JVMs, you can instantiate a file called ./documents/mybook/chapterone.doc on a Windows system or a UNIX system and have the same behavior. If you need to be able to run on different platforms, the more correct way of doing this would be as follows:

```
String myFilename = File.separator + "documents" + File.separator +
  "mybook" +  File.separator + "chapterone.doc";
```

`File.separator` is a static `String` variable in the `File` class that will be set correctly for the operating system in which the JVM is running. The JVM on Windows will set the `File.separator` to a backslash (\); the JVM on UNIX will set it to a forward slash (/). So `myFileName` would equate to \documents\mybook\chapterone.doc on Windows and /documents/mybook/chapterone.doc on UNIX. The `File.separatorChar` is a static `char` value that acts the same way.

You have four ways to create a new `File` instance.

```
File myFile = new File(String pathName);
File myFile = new File(File parent, String child);
File myFile = new File(String parent, String child);
File myFile = new File(URI uri);
```

The first way takes a string that defines the file or directory as a full or relative pathname. The second and third ways require a parent and a child name. The parent is the directory name where the child resides. The child may be either a file or a

directory itself. The fourth constructor accepts a URI type file specification. The URI specification encompasses additional ways to specify a file beyond the tradition file system syntax. (See http://java.sun.com/javase/6/docs/api/java/net/URI.html for more details.)

You can create a file instance using either full or relative pathnames. If the current directory were \documents, you could instantiate a file referencing the mybooks subdirectory in several ways:

```
// Remember the double slash is required to represent a single slash.
File myDirectory = new File("\\documents\\mybooks");
File myDirectory = new File(".\\mybooks");
File myDirectory = new File("mybooks");
```

Once the file is instantiated, you can extract information about the file using various `get` methods.

Method	**Description**
getName()	Returns the name of the file or directory without path information.
getPath()	Returns the relative or full path, depending upon how the File was instantiated.
getParent()	Returns the name of the directory that "owns" the file or directory.
getAbsolutePath()	Returns a nonrelative and absolute pathname that is resolved against the current drive/directory.
getCanonicalPath()	Returns a nonrelative and absolute pathname that is unique, but highly system dependent.

The `getPath()` and `getParent()` methods return information differently, depending upon how the `File` instance was constructed. The `getName()`, `getAbsolutePath()`, and `getCanonicalPath()` methods, however, return the same information no matter how the `File` instance is constructed. For example, if you construct the mybooks directory like this:

```
File myDirectory1 = new File("\\documents\\mybooks");
File myDirectory2 = new File("mybooks");
```

```
getName()  returns "mybooks" for myDirectory1 and myDirectory2.
getPath()  returns "\documents\mybooks" for myDirectory1 and "mybooks"
for myDirectory2
getParent()       returns "\documents" for myDirectory1and null for
myDirectory2
```

you may create a `File` instance even when the file or directory it represents does not exist. None of the constructors throw an `IOException` or `FileNotFoundException` if the file or directory does not exist (they will throw a `NullPointerException` if the `pathName` or child is null). This capability is needed to invoke the various methods to create directories and files. If you need to know whether the directory or file exists, `File` provides the `exists()` method that returns true if the entity exists and false if it does not. The `isDirectory()` method returns true if the `File` entity is a directory and exists; otherwise, it returns false. The `isFile()` method returns true if the entity is a file (not a directory) and exists; otherwise, it returns false.

For both files and directories, the `File` class provides access to the characteristics of the file:

Method	Description
length()	Returns the size of the file or directory
lastModified()	Returns the time stamp of the file or directory
isHidden()	Tests to see if file or directory is hidden
canRead()	Tests to see if you have read access to the file or directory
canWrite()	Tests to see if you have write access to the file or directory

The `File` class provides several directory-related functions.

Method	Description
list()	Returns the files and directories in the directory as a String array
listFiles()	Returns the files and directories in the directory as a File array
mkdir()	Create the directory
mkdirs()	Create the tree of directories
delete()	Delete the directory

The `File` class provides access to several file-related functions.

Method	Description
createNewFile()	Returns true if the file creation is successful
createTempFile()	Creates a file in the temporary directory and returns a File if successful
delete()	Delete the file
deleteOnExit()	Delete the file when the VM shuts down

InputStream and OutputStream

`InputStream` and `OutputStream` are the parent classes for all of the streams that are used to read and write byte streams. Both classes are abstract classes. If you remember what abstract classes are from earlier chapters, you will recall that an abstract class cannot be instantiated. That means you will never be using `InputStream` or `OutputStream` directly and will always be using classes that inherit from them.

`InputStream` defines three basic methods of reading data:

Method	Description
read()	Abstract method that reads a single byte and returns its value as an integer
read(byte [] data)	Reads bytes to fill the byte array and returns the number of bytes read
read(byte [] data, int offset, int len)	Reads bytes to fill the byte array at the selected point and returns the number of bytes read

The classes that extend `InputStream` will override these methods to handle the underlying storage or transport media. A `FileInputStream` will read data from disk. An `InputStream` obtained from a network connection will read data from a TCP/IP connection. The logic for performing the reading remains essentially the same because in both cases the logic is using the `InputStream` methods for reading and writing.

Method	Description
write(int b)	Abstract method that writes a single byte
write (byte [] data)	Writes all the bytes from the byte array
write(byte [] data, int offset, int len)	Writes selected bytes from the byte array

`OutputStream` parallels `InputStream` in having three basic methods for writing data:

To illustrate the usage of `InputStreams` and `OutputStreams`, I am going to extend the earlier example and create a simple file-copy program. A simple file-copy program in COBOL would look something like this:

```
FD  INPUT-FILE.
01    INPUT-RECORD.
         05  FILLER                                            PIC X(100).

FD  OUTPUT-FILE.
01    OUTPUT-RECORD.
         05  FILLER                                            PIC X(100).

         OPEN INPUT INPUT-FILE.
         OPEN OUTPUT OUTPUT-FILE.
```

```
READ-ANOTHER-RECORD
         READ INPUT-FILE
              AT END GO TO CLOSE-FILES
         END-READ.
         MOVE INPUT-RECORD TO OUTPUT-RECORD.
         WRITE OUTPUT-RECORD.
         GO TO READ-ANOTHER-RECORD.

CLOSE-FILES.
          CLOSE INPUT-FILE.
         CLOSE OUTPUT-FILE.
```

To perform similar processing with Java, use the following code:

```
// Create streams related to a disk file.
FileInputStream instrm = new FileInputStream("in.dat");
FileOutputStream outstrm = new FileOutputStream("out.dat");
// Allocate a byte array to hold each record.
byte [] inRecord = new byte[100];
// Read the first record.
int bytesRead = instrm.read(inRecord);
// Loop until end of file when bytesRead = –1.
while (bytesRead != -1) {
// Write the record and read the next record.
outstrm.write(inRecord);
bytesRead = instrm.read(inRecord);
}

// Close the streams and files.
instrm.close();
outstrm.close();
```

Several problems could occur in processing the file. Errors could obviously occur on opening either the input or the output file. Another, subtler type of problem is the `write()` method chosen. The `write()` method will always write the 100-byte array. As long as the input file contains 100-byte records and the file length is an exact multiple of 100, the program would work fine. If that is not the case, the final `read()` could read less than 100 bytes and the final `write()` could write extra bytes at the end of the file. To prevent this problem, the write could be changed as follows:

```
// Write only the bytes read.
outstrm.write(inRecord, 0, bytesRead);
```

Serialization

Java provides an ability to transform any object into a stream of bytes and save the stream of bytes to the file system, thus making the object persistent. Then, at a later time, a program could read back the byte stream and transform the stream of bytes back into an object. You can think of this mechanism as somewhat like teleportation in science fiction. You create an object in a VM, set various values in it, transform it into bytes, and send it someplace else. Upon arrival at the other location, another VM reassembles the bytes back into the original object with all of the values properly initialized to their "teleported" values.

The two VMs may be two actual VMs communicating in real time over a network. Serialization is a key mechanism that lies behind Remote Method Invocation (RMI). RMI permits a program running in one VM to invoke a method in a program running on a different VM, one perhaps a continent away accessed through a network.

The two VMs may be two executions of the VM on the same physical machine. The serialization, in this case, may be to a file system. A program executes and accepts data from a user. The user decides to exit the program. The program saves the data entered by the user to disk by serializing the Java objects that contain the data. Later the user starts the program, and the program reads the saved data from disk, serializes it back into the original objects, and the program can pick up where it left off in interacting with the user.

Although you might think serialization would be difficult for you as a programmer to implement, it is actually quite simple. To make any class serializable only requires that you add the `Serializable` interface tag to the class. In other words, if you have a class defined like this:

```
public class MyClass
{
// code
}
```

About the only thing required to make the class serializable is to do this:

```
public class MyClass implements Serializable
{
// code
}
```

`Serializable` is a Java interface without any method declarations. Simply adding `implements Serializable` makes the objects of that class able to be serialized—almost. Serialization does require that all of the attributes or data elements of the class also be serializable. This stands to reason: Java cannot serialize the object if it cannot serialize all the components of the class. Almost all the Java API classes are already serializable, and all Java primitives are serializable. Unless `MyClass` contains classes you have defined that are not serializable, it can be serialized.

Serialization is extremely powerful. Not only is the object itself serialized, but all of the objects it contains are serialized. This continues in an iterative fashion unless some of the lowest levels of the data structure are serialized. The same applies also to the objects contained in one of the Java `Collections` classes that will be discussed in more detail in the next chapter

Occasionally, you may have a reason not to serialize a data element in a class. Suppose you have a large array of data that you use as a work area during processing. You may have no need to save the data. Perhaps the data can be easily recreated or has significance only during certain processes. You would not want to write the data to disk or transmit it over the network. The `transient` keyword allows you to specify that a data element (primitive, array, or object) should not be serialized.

```
public class MyClass
{
transient String [] myBigArray;
// code
}
```

To accomplish serialization, you need to use `ObjectInputStream` and `ObjectOutputStream`.

```
int i = 55;
String myValue = "some string data";
MyClass appData = new MyClass();
// Create the output streams.
FileOutputStream ostream = new FileOutputStream ("save.dat");
ObjectOutputStream oostream= new ObjectInputStream(ostream);
// Serialize the data.
oostream.writeInt(i);
oostream.writeObject(myValue);
oostream.writeObject(appData);
// Close the streams.
oostream.close();
ostream.close();
// Create the input streams.
```

```
FileInputStream istream = new FileInputStream("save.dat ");
ObjectInputStream oistream = new ObjectInputStream(istream);
// Read the data back, making sure the order is same as it was written.
i = oistream.readInt();
myValue = (String) oistream.readObject();
appData = (MyClass) oistream.readObject();
// Close the streams.
istream.close();
oistream.close();
```

Many streams have constructors that allow them to be instantiated with other streams of the same type. In this case, both `ObjectInputStream` and `ObjectOutputStream` were instantiated with corresponding file streams. If you are reading or writing large amounts of data, you will want to create buffered streams in this manner.

READERS AND WRITERS

The `InputStream` and `OutputStream` classes are for reading and writing byte streams. As discussed in Chapter 8, Java stores strings and characters in a double-byte Unicode format. Text files, as stored and manipulated by most editors, treat characters as single bytes of data. The alert reader can see problems in all of this. How does Java transform single bytes into double bytes? What is the extra byte for in the Unicode format?

In most cases, we in the Western world never have problems with any of this. We use an extended ASCII character set with character values ranging from 0 to 255 (some of the characters are control characters or are unprintable, or both). In Unicode, the extra byte represents the character set, and the value of the extended ASCII character set is 0. To transform the ASCII letter A, hexadecimal value 0x41, into double-byte format, you simply place a byte of value zero in front of it, 0x0041. In transforming the A back to single-byte format, you remove the leading zero.

What would happen, however, if someone created a file on a system using a different character set and shipped it to us? The `Reader` and `Writer` classes are the parent classes for resolving these problems. `Readers` and `Writers` are used for reading and writing text files. Unless you deal with communications from countries using non-European languages, you will probably never have to use the character set conversion features that the `Reader` and `Writer` classes support. As a knowledgeable programmer, you need to know that the support is present if you need it. Otherwise, you can take advantage of the API that facilitates ordinary text input and output.

Reading a text file involves instantiating a `FileReader`. In the following example, you'll make use of `BufferedReader` for its `readLine()` method. The `readLine()` method expects a carriage return, a line feed, or carriage return-line feed pair to indicate the termination of a line.

```
// Create reader related to a disk file.
FileReader fileReader = new FileReader("myfile.txt");
BufferedReader bufReader = new BufferedReader(fileReader);
// Read a line.
String line = bufReader.readLine();
// Check for end of file when line equals null. If not, print line and
// read another.
while (line != null) {
     System.out.println(line);
     line = bufReader.readLine();
}
bufReader.close();
fileReader.close();
```

If you needed to handle differences in character sets, the code would be slightly more complex. Let's say the file came from China.

```
// Define the encoding for Chinese.
String encoding = "zh_CN";
// Create streams related to a disk file.
FileInputStream instrm = new FileInputStream("myfile.txt");
InputStreamReader instrmReader = new InputStreamReader (instrm, encoding);
```

At this point, you could read the file and each line would return a string with the correct Unicode encoding. Quite likely `println()` would not be able to display the correct characters to the console. However, if the machine where the program was running had support for the Chinese character set, a display of the strings in a graphical environment would show the Chinese characters correctly.

Writing a text file is just as simple as reading one. As you might expect, for every `Reader` there's a `Writer`.

```
// Create writers related to a disk file.
FileWriter fileWriter = new FileWriter(("myfile.txt");
BufferedWriter bufWriter = new BufferedWriter(fileWriter);
// Write two lines each, followed by a new line.
bufWriter.write("line one"]);
bufWriter.newLine();
```

```
bufWriter.write("line two"]);
bufWriter.newLine();
// Close everything.
bufWriter.close();
fileWriter.close();
```

RandomAccessFile

The RandomAccessFile class is quite different from the other java.io classes for performing input and output operations. RandomAccessFile does not extend any of the reader/writer classes or any of the stream classes. It does not extend any other of the java.io classes. Also, in contrast to the other classes of java.io, it is not specialized for input or output but is able to do both through its implementation of the DataInput and DataOutput interfaces.

You can use RandomAccessFile to obtain functionality quite close to the COBOL functionality of the relative record file. Unlike COBOL, but like the stream and reader/writer classes, the concept of a logical record is completely lacking outside the Java code you create to read and write the data in the file. You might best think of a RandomAccessFile as a stream that you can read or write from any location in the file. (However, about the closest you can get to this notion of a record would be to set the data array to the size of the record.)

Relative file I/O is not supported by all COBOL compilers and can have different implementations on different platforms. Generally, the logic to read a relative file randomly is something like this:

```
SELECT CUSTOMER -FILE ASSIGN TO CUSTOMER
     ORGANIZATION IS RELATIVE
     ACCESS MODE IS RANDOM
     RELATIVE KEY IS WW-CUSTOMER-NUMBER.

FD  CUSTOMER -FILE.
01    CUSTOMER -RECORD.
         05  CUSTOMER-NAME            PIC X(25).
         05  CUSTOMER-ADDRESS         PIC X(30) OCCURS 5 TIMES.

01   WW-CUSTOMER- NUMBER              PIC 9(10) VALUE ZEROES.
```

```
            OPEN INPUT CUSTOMER -FILE.
            MOVE 300 TO WW-CUSTOMER-NUMBER.
            READ CUSTOMER -FILE.
.            CLOSE CUSTOMER -FILE.
```

The preceding program reads the 300th record of the customer file.

`RandomAccessFile` instances are constructed by passing two parameters. The first parameter is either a string representing a file pathname or a `File` instance. The second parameter is also a `String` parameter and is used to indicate the manner in which the file is to be opened. If the second parameter is `r`, it means that the file is to be opened in read-only mode and the write methods of the class will fail with `IOExceptions`. If the second parameter is `rw`, the file is opened in read-write mode. In addition, if the file does not exist, it will be created. (There are also `rwd` and `rws` parameters, which are asynchronous versions of these modes.) Once you have opened the file with the constructor, you can process it either sequentially or randomly. To process it sequentially, you simply use the various read and write methods available from the implemented interfaces to write bytes, strings, or any of the Java primitive variable types.

To process a file randomly, you need to understand the concept of a file pointer. A file pointer is simply the current byte position in the file from the beginning of the file. When you open the file, you are at its start and the current position is 0. Every reading or writing operation you perform resets this position. If you read 5 bytes, the position is 4. If you then write 2 bytes, the position is 6. The first byte of the file is byte 0. If you lose track of where you are positioned in the file, you can call the `getFilePointer()` method to tell you the location.

You use the `seek()` method to perform random access. The `seek()` method takes a long number as its argument. It repositions the file pointer without performing any reading or writing. You can leap from the beginning of the file to the last byte of the file simply by repositioning the file pointer with the `seek()` method.

To simulate relative record input/output, you add the concept of a record. As long as the records are fixed length, you can easily translate from a record pointer to a file pointer by subtracting one from the record number (if you consider the first record number one, not zero) and multiplying by the record length.

Putting it all together, this is how you would process the customer file with Java:

```
// Set the record length and number field.
int recLen = 175;
int recNum = 300;
// Open the file.
RandomAccessFile file = new RandomAccessFile("test.fil", "r");
// Seek to the 300th record.
file.seek((recNum - 1) + recLen);
```

```
// Allocate the bytes for the record and read it,
byte [] record = new byte[recLen];
file.read(record);
// Close the file.
file.close();
```

EXERCISES

Now it's time to try out what you've learned with some simple examples.

1. Edit the HelloWorld Java program you've been working with and remove the code from the `main()` method. You will need to import the java.io package in order to compile and run this example.

```
import java.io.*;
```

2. First, let's create a `File` object referring to a file called myfile.dat in the current working directory.

```
// Create a File object.
File file = new File("myfile.dat");
```

3. You'll check for the existence of this file. It won't exist the first time you run the program.

```
// Check for existence of file.
if (file.exists())
      System.out.println("File exists");
else
     System.out.println("File does not exist");
```

4. You're going to write raw bytes to this file. You will have to bracket this code with a `try...catch` block because you could get an exception.

```
try
{
// Create output stream to write to file.
     FileOutputStream ostrm = new FileOutputStream(file);//
Write bytes to file from a String.
     for (int i = 0; i < 20; i++)
     {
```

```
                String text = "This is text " + i;
                byte [] bytes = text.getBytes();
                ostrm.write(bytes);
            }
        // Close the output stream.
            ostrm.close();
        }
        catch (Exception e)
        {
            System.out.println("You had a output problem:" + e);
        }
```

5. Finally, you're going to check for the existence of the file again. If the code executes properly, this time the file will exist.

```
// Check for existence of file.
if (file.exists())
     System.out.println("File exists");
else
     System.out.println("File does not exist");
```

6. Let's compile this program and execute it. The output should look like this:

```
File does not exist
File exists
```

7. Let's enhance the program to read the data back that you write out to the file. After the code you added, let's create an input stream. Once again, you have to bracket the code with a `try...catch` block.

```
    try
    {
// Create input stream to read file.
        FileInputStream istrm = new FileInputStream(file);
// Allocate bytes to hold data.
     byte [] bytes = new byte[14];
     int bytesRead =      istrm.read(bytes);
     while (bytesRead != -1)
     {
// Convert bytes to string and read some more.
         System.out.println(new String(bytes));
         bytesRead = istrm.read(bytes);
     }
```

```
// Close the input stream.
     istrm.close();
}
catch (Exception e)
{
     System.out.println("You had an input problem:" + e);
}
```

8. Let's compile this program and execute it. The output should look like this:

```
File exists
File exists
This is text 0
This is text 1
This is text 2
This is text 3
This is text 4
This is text 5
This is text 6
This is text 7
This is text 8
This is text 9
This is text 1
0This is text
11This is text
 12This is tex
t 13This is te
xt 14This is t
ext 15This is
text 16This is
 text 17This i
s text 18This
is text 19his
```

Notice that the `read` method doesn't have any concept of lines or what constitutes a logical unit of data. It is simply reading raw bytes in blocks of 14, so the strings of text begin to become offset when you try to read strings of 15 bytes.

9. Let's enhance the program to write the data in fixed lengths so that when you read it back in again, there is no offset. After the statement that creates the output stream, add these additional statements.

```
// Use a DecimalFormat object to make sure 2 digits
// are used when you print the value.
DecimalFormat form = new DecimalFormat();
form.setMinimumIntegerDigits(2);
```

Then replace the statement that creates the output text with this statement, which uses the `DecimalFormat` object:

```
//  String text = "This is text " + i;
String text = "This is text " + form.format(i);
            Finally, change the size of the read buffer:
//   byte [] bytes = new byte[14];
byte [] bytes = new byte[15];
```

10. Let's compile this program and execute it. The output should look like this:

```
File exists
File exists
This is text 00
This is text 01
This is text 02
This is text 03
This is text 04
This is text 05
This is text 06
This is text 07
This is text 08
This is text 09
This is text 10
This is text 11
This is text 12
This is text 13
This is text 14
This is text 15
This is text 16
This is text 17
This is text 18
This is text 19
```

Notice that output is now formatted based on fixed length "rows."

Reviewing the Exercises

In this chapter, you performed some simple operations using the `File` classes, including instantiating the class and checking for the existence of the file in the file system itself.

You also did some simple input and output using the `File`-based input and output stream classes.

You should remember that I/O in Java is stream-based and not always related to the file system. In fact, once the stream is instantiated, operations on the stream are completely ignorant of the underlying storage and transport mechanisms. The transport could be in memory, across a network, or through the operating system to disk.

The `Reader` and `Writer` classes I discussed are more typically used for text output and input. You might want to modify the program to use the readers and writers I've discussed.

11 Java Collections

In This Chapter

- Collections Background
- Ordered Collections: Vectors and ArrayLists
- Keyed Collections: Hashtable and HashMap
- Other Collections
- Iterators
- Ordering and Comparison Functions
- Exercises: Java Collections
- Reviewing the Exercises

Collections provide the ability to manage groups of objects. A collection can be ordered or unordered, contain duplicates or not, depending upon the particular implementation chosen. Collections can also contain key-value pairs to provide a facility for maintaining in memory an index of objects.

What's wonderful about collections for the COBOL programmer is the automatic memory management that lies behind the group of objects. Unlike COBOL arrays, space does not need to be allocated in advance for a fixed number of objects in this collection. You can simply define the collection and add objects to it as needed, without any concern for running out of space. The only limitations are those of the physical memory of the machine or operating system.

Collections Background

Before Java 1.2, the java.util package had two primary classes that implemented functionality similar to the functionality of the Java Collections framework: `Vector` and `Hashtable`. A `Vector` may be thought of as an array that can expand or contract dynamically as objects are added to it or taken from it. A `Hashtable` may be thought of as a group of objects, each of which is paired with or indexed by a key.

If you ever work with programs written before Java 1.2, you will probably find extensive usage of `Vectors` and `Hashtables`. `Vectors` are indispensable in situations in which you needed to maintain in memory a list of objects but could not predict the exact number of objects you would have.

Java 1.2 created a much more powerful framework for handling groups of objects. The Collections framework is an extensible assortment of supporting classes and interfaces. `Vector` and `Hashtable` still exist and operate much the way they used to in prior versions of Java. Code written with `Vector` and `Hashtable` in Java 1.1 or before will compile and work the same in Java 1.2.

`Vector` and `Hashtable` stand out from the rest of the Collections framework because they are legacy classes. The methods for adding and retrieving objects from them are different from those for the rest of the framework. They are also different in another big way: The methods for manipulating the objects in `Vector` and `Hashtable` are synchronized. This means that you can add and remove objects from them from multiple threads at the same time, and the class maintains its internal consistency. That sounds good, but it can be costly; synchronized code carries a heavy performance penalty. If you are writing a single-thread application or are manipulating the objects in the `Vector` or `Hashtable` from a single thread, you don't need to add that overhead.

To a large extent, the newer `ArrayList` collection class mimics the functionality of `Vector`, and `HashMap` mimics the functionality of `Hashtable`. These collection classes do not use synchronized methods, so they perform much better when synchronization isn't required.

One useful way to understand collections is to compare them to traditional COBOL files. COBOL programs can traverse a file in a particular order, looking for specific records, and then modify or perhaps delete those records. In order to examine or modify any particular record, the program must first read it into a record structure. Standard functions exist to read, update, delete, and reorder (sort) the file. Further, the file has a determinate number of records, and the program can detect when it has reached the end of the file.

All COBOL programmers know how to process files. Data stored in files is readily available to a COBOL program. If you need to build a program that

processes this file, all you need to know is where the file is stored and its record layout. The COBOL developer can concentrate on the business problem to be solved and efficiently use the syntax appropriate to process a COBOL file.

The preceding descriptions are (more or less) the characteristics of Java collections. It is also helpful to note that collections have the following differences:

- Java collections are stored in memory, not on disk (although you can certainly extend the basic collection class to provide for disk storage).
- Multiple users (that is, programs) can simultaneously access files, whereas collections are contained in a single runtime instance.
- Java collections can be passed as parameters to functions and can be returned as the return item from a method. This would be similar to passing the file selection and file definition to a subroutine, allowing the subroutine to read and update the file. (Actually, some COBOL compilers do allow this!)
- Java collections store references to objects, not the objects themselves. When you add an object to a collection, you are simply adding a reference to the object. You are not cloning the object and creating a copy of it.

A collection is a set of related objects or elements. One collection may be ordered in some fashion, and others may be unordered. A collection type might allow duplicates, whereas another implementation might not.

The basic collection interface supports a few *bulk operations*, or methods that can be used to move many elements at one time. As an added bonus, the contents of a collection can be easily moved into arrays and out of arrays.

These are some of the interface definitions in Java's Collection framework:

Set: An interface for implementing classes that support groups of elements that cannot contain duplicates. Elements in a `Set` are not ordered in any particular fashion.

List: An interface for a class that supports groups of ordered elements. Elements in a `List` can contain duplicates. Elements in a `List` are often accessed using an index (or subscript) in a manner very similar to arrays.

Map: An interface for a class that maps key elements to values (i.e., other objects). It is roughly analogous to an indexed (ISAM) file in COBOL, where a key is mapped to a record (a file that is contained in memory).

SortedSet: An extension of the `Set` interface. Elements in a SortedSet collection are automatically ordered in some fashion, either through the natural ordering of the elements or because an explicit `Comparator` object provides for an ordering algorithm.

Queue: A collection that is designed to manage objects that are to be "processed" in some fashion. Typically the objects in a Queue are processed in first-in-first-out (FIFO) order, although there are other orders available.

The preceding interfaces are just definitions; it is up to an actual implementation to construct a class that supports these definitions. The Java Collections frameworks provide some very useful basic collection implementations. These include some Set collection types, a few List collection types, and two Map collection types, as well as a number of Queue types.

These collection implementations can, in many cases, be used as is by your application.

Sets	HashSet and TreeSet
Lists	ArrayList and LinkedList
Maps	HashMap and TreeMap
Queues	PriorityQueue and ConcurrentLinkedQueue and BlockingQueue

ORDERED COLLECTIONS: VECTORS AND ARRAYLISTS

The most commonly used ordered List is the `ArrayList` or its older cousin, the `Vector`. Both classes support a growable array of objects that can be accessed with an integer index.

The `Vectors` and `ArrayLists` are Java constructs that are similar to Java arrays in many ways, but they implement additional capabilities.

- They are explicitly implemented as a class in the java.util package.
- They can grow and shrink dynamically as required.
- They can hold only Objects (that is, `Vectors` and `ArrayLists` cannot contain primitive numeric types).

A program can create a `Vector` or `ArrayList` using this syntax:

```
Vector errorMsgsVector = new Vector ();
ArrayList errorMsgsList = new ArrayList ();
```

The following syntax, introduced with Java 1.5, uses the concepts of Generics to tell the compiler what type of object you want to store in your collections. The

compiler will evaluate the type of each object that is placed into the collection and report any invalid object types as compile-time errors.

```
Vector< ErrorMsg > errorMsgsVector = new Vector< ErrorMsg > ();
ArrayList< ErrorMsg > errorMsgsList = new ArrayList< ErrorMsg > ();
```

Both `Vector` and `ArrayList` have constructors that pass an integer for the capacity of the collection. The capacity value is merely the initial capacity and, if the constructor without it is used, a default value for initial capacity is used.

`Vectors` and `ArrayLists` have a size attribute. *Size* is the number of elements currently in the `Vector`. The sample `Vector` so far has a size of zero (no elements in the `Vector`). The values of this attribute can be accessed with the `size()` methods.

To add elements to the `Vector`, use the `addElement()` method. To add elements to the `ArrayList`, use the `add()` method. This places a new element in the `Vector` in the next available position and increases the size of the `Vector` by 1.

```
// Create some objects.
     ErrorMsg myErrorMsg = new ErrorMsg ();
     ErrorMsg myotherErrorMsg = new ErrorMsg ("Some Text");
     ErrorMsg mythirdErrorMsg = new ErrorMsg ("Third One");
// Place them in the Vector.
     errorMsgsVector.addElement (myErrorMsg);
     errorMsgsVector.addElement (myotherErrorMsg);
     errorMsgsVector.addElement (mythirdErrorMsg);
// Place them in the ArrayList.
     errorMsgsList.add(myErrorMsg);
     errorMsgsList.add(myotherErrorMsg);
     errorMsgsList.add(mythirdErrorMsg);
```

Now the `Vector` and `ArrayList` both have a size of three.

A `Vector` and `ArrayList` will automatically expand when their capacities are exceeded. If you were to add 11 items to the `Vector`, then the `Vector` would resize itself (and probably relocate itself in memory). The default behavior is to double in size whenever the current capacity is exceeded. Defining an increment value when the `Vector` is first created will control how the `Vector` is expanded:

```
     Vector errorMsgs = new Vector (10, 20);
```

This `Vector` will have an initial capacity of 10 and will increment by 20 element positions whenever its current capacity is exceeded (that is, first 10, then 30, then 50 elements).

A performance penalty is incurred when a `Vector`'s capacity is exceeded. So under some circumstances it might be advantageous to allocate capacity for a `Vector`, especially one that will hold large numbers of elements (more than 50 or so). The default for a `Vector` is to allocate space for 10 elements and to increment by 10.

Actually, using `Vectors` and `ArrayLists` in your program is a little more complex than it would seem. This is because you have to use the proper method when you add, change, or retrieve objects from a `Vector`. Furthermore, `Vectors` and `ArrayLists` store only objects, not specific class types. Subsequently, any object retrieved from them must be cast back into the correct class before it can be properly used, if you do not store the same Generic type definition specified when the collection was created.

Let's contrast the statements necessary to access elements in an array with the way you would manage elements in an `ArrayList`:

```
// Define an array of five error messages.
     ErrorMsg[] errorMsgs = new ErrorMsg[5];
// Add some elements to the array.
     errorMsgs[0] = myErrorMsg;
     errorMsgs[1] = myotherErrorMsg;
// Modify the first element in the array.
     errorMsgs[0] = mythirdErrorMsg;
// Retrieve the first element in the array.
     myErrorMsg = errorMsgs[0];
// Create a potential problem because errorMsgs may not be large enough to
// contain all of the error messages in ErrorMsgIO.
     for (int x = 0; x < errorMsgIO.length; x++) {
          errorMsgs[x] = errorMsgIO[x]
               ...
     }
// Retrieve the number of elements in errorMsgs.
     int y = errorMsgs.length;
```

This is how you could code `ErrorMsgs` as an `ArrayList`:

```
// Define an ArrayList that will contain ErrorMsgs.
     ArrayList<ErrorMsg> errorMsgs = new ArrayList<ErrorMsg> ();
// Add some elements to the list.
     errorMsgs.add (myErrorMsg);
     errorMsgs.add (myotherErrorMsg);
// Modify the first element in the list.
     errorMsgs.set (0, mythirdErrorMsg);
// Retrieve the first element from the list.
     myErrorMsg = errorMsgs.get (0);
```

```
// This loop will never be a problem because ErrorMsgs will grow as required
// to contain all the error messages in ErrorMsgIO.
      for (int x = 0; x < errorMsgIO.length; x++) {
          errorMsgs.add (errorMsgIO[x]);
                  ...
      }
// Retrieve the number of elements in errorMsgs.
      int y = errorMsgs.size();
```

Since `Vectors` and `ArrayLists` can contain only objects, and not primitive data types, numeric items cannot be directly placed into `Vectors`. Instead, numbers must be cast first into their object wrappers before they can be placed into a `Vector` array.

```
      // Add an Integer element to an ArrayList.
      ArrayList<Integer> myArrayList = new ArrayList< Integer > ();
      int x = 5;
   Integer  myInteger = new Integer (x);
   myArrayList.add (myInteger);
```

The autoboxing feature introduced with Java 1.5 can create these object wrappers for you automatically. However, under the covers, the compiler is still creating the integer object.

```
      // Use Autoboxing to add an Integer element to the ArrayList.
      int x = 5;
   myArrayList.add (x);
```

Another difficulty with integer objects is that they are immutable, just like string objects. That is, once an integer object is created, there is no way to change its value. Instead, a new integer must be created to contain a new value. As a result, groups of numeric values are often managed with Java arrays (which can contain primitive data types), instead of with `Vectors` or `ArrayLists`.

You can shrink a `Vector` or `ArrayList` in order to conserve memory. The `trimToSize()` method will truncate the `Vector`'s capacity to the current size. You should perform this function only if you are sure the `Vector` is not likely to grow, since any additions to the `Vector` will immediately cause it to be reallocated.

Following is a table showing the most commonly used `ArrayList` methods with their corresponding `Vector` methods.

ArrayList Method	Corresponding Vector Method
ArrayList()	Vector()
add()	addElement()
get()	elementAt()
set()	setElementAt()
remove()	remove()
size()	size()

KEYED COLLECTIONS: HASHTABLE AND HASHMAP

The most commonly used keyed list is the `HashMap` or its older cousin, the `Hashtable`. Java 1.5 introduces a high-performance implementation of `HashMap`: `ConcurrentHashMap`. All classes support a growable array of objects that can be accessed with a key.

These classes are used for managing objects and providing direct access to the objects. Like `ArrayLists` and `Vectors`, you can only place objects into them. When using Generics, the compiler will ensure that the object returned is the type specified for the collection. If you need to, you can cast the returned object to the correct type in order to be able to use it.

From a COBOL perspective, maps can be viewed as similar to the indexed file type. The index of the file is analogous to the keys in a Map collection, and the record area is analogous to the values in a Map collection. To be more precise, an indexed sequential (ISAM) file type is very similar to the TreeMap collection, and an indexed file accessed via a hashed key is very similar to the HashMap collection.

A program can create a `Hashtable` or `HashMap` using this syntax:

```
Hashtable errorMsgsTable = new Hashtable ();
HashMap errorMsgsMap = new HashMap ();
```

Like `Vector` and `ArrayList`, `Hashtable` and `HashMap` have constructors that pass an integer for the capacity of the collection and a size attribute. Also, you should specify the object types that your collection will contain using the `<ObjectType, ObjectType>` qualifier.

```
Hashtable<String, ErrorMsg> errorMsgsTable = new Hashtable<String, ErrorMsg>
(20);
HashMap<String, ErrorMsg> errorMsgsMap = new HashMap<String, ErrorMsg> (50);
```

To add elements to the `Hashtable` or `HashMap`, use the `put()` method. The `put()` method takes two arguments, both of which are objects. The first argument is the key and the second is the value or the object you wish to retrieve with the key.

```
// Create some objects.
     ErrorMsg myErrorMsg = new ErrorMsg ();
     ErrorMsg myotherErrorMsg = new ErrorMsg ("Some Text");
     ErrorMsg mythirdErrorMsg = new ErrorMsg ("Third One");
// Place them in the HashMap.
     errorMsgsMap.put ("error one", myErrorMsg);
     errorMsgsMap.put ("error two",  myotherErrorMsg);
     errorMsgsMap put ("error three", mythirdErrorMsg);
// Place them in the Hashtable.
     errorMsgsTable.put ("error one",  myErrorMsg);
     errorMsgsTable.put ( "error two", myotherErrorMsg);
     errorMsgsTable.put ( "error three", mythirdErrorMsg);
```

Now the `Hashtable` and `HashMap` have a size of three. You retrieve the values from the `Hashtable` or `HashMap` using the `get()` method.

```
// Get them from the HashMap.
     ErrorMsg myErrorMsg  = (ErrorMsg ) errorMsgsMap.get ("error one");
     myErrorMsg  =  errorMsgsMap.get ("error two");
     myErrorMsg  =  errorMsgsMap.get ("error three");
// Get them from the Hashtable.
     myErrorMsg  =  errorMsgsTable.get ("error one");
     myErrorMsg  =  errorMsgsTable.get ( "error two");
     myErrorMsg  =  errorMsgsTable.get ( "error three");
```

Other Collections

Java provides some other collection types that are quite useful in particular situations.

HashSet: A general-purpose yet efficient implementation of the basic `Set` interface. Elements in a `HashSet` are not ordered in any particular way, but iteration (that is, sequential access) to these elements is optimized compared to `TreeSets`.

TreeSet: A basic implementation of the `SortedSet` interface. Elements in a `TreeSet` will be ordered based on its comparator. The default comparator orders elements based on their natural order, but you can override this behavior with your own comparator.

A word of caution about the Set collection type. The `Set` interface defines good semantics that allow a program to add and remove elements from the collection, but there is no reliable way to manage elements that have changed while they are part of a set. This is because the modification is assigned to the element itself, not the collection (that is, the collection doesn't see the modification). The proper sequence in this case is to get the element, remove it from the set, modify it, and then add it into the set. To extend this programming model to the COBOL file processing analogy, you would need to read in the record, modify it, delete the record, and then write the record.

AbstractSequentialList: The basic implementation of the `List` interface, appropriate for sequential access to the members of the list. This class is provided in order to simplify the task of a developer who wishes to implement a custom list collection, one that supports sequential access (for example, a `LinkedList`).

LinkedList: The `LinkedList` implementation of the `List` interface. This uses many of the methods provided by the `AbstractSequentialList` class. It also adds methods to conveniently add, delete, and retrieve elements from the beginning or end of a list. `LinkedLists` are most appropriate when you want to optimize write performance, compared to read performance.

ITERATORS

Iterators are objects that support sequential access to the elements in a collection. The `iterators()` method of `Vector` and `ArrayList` returns an `Iterator` object.

```
    Iterator iterator =  errorMsgsVector. iterator ();
    while (iterator.hasNext()) {
            ErrorMsg myErrorMsg  =  (ErrorMsg ) iterator.next ();
}
```

To obtain an iterator for a `HashMap` or `Hashtable`, you must obtain a reference to the iterator for either the values or the keys. Once you have the one you want, you can then use it to iterate either of them.

```
Iterator iterator =  errorMsgsMap.values().iterator ();
while (iterator.hasNext()) {
        ErrorMsg myErrorMsg  =  (ErrorMsg ) iterator.next ();
}
```

Java 1.5 modified the processing associated with the `for` loop in order to simplify the use of collections. Essentially, if you are examining the contents of a collection, you can use the `{ for : each }` syntax to examine the contents of a collection in sequential order, without explicitly defining or managing an iterator.

```
for (ErrorMsg  myErrorMsg : errorMsgsVector)
        System.out.println  ("Error message: " +  myErrorMsg );
```

Read this statement as "for each `ErrorMsg myErrorMsg` in `errorMsgsVector`." Notice that this statement places the current value (that is, the value pointed to by the implied Iterator) into the variable `myErrorMsg`. This syntax also uses Generics to simplify and clarify the statement.

Ordering and Comparison Functions

The `Collections` class contains several static methods that provide useful functions on collections, such as sorting and searching. These powerful (and polymorphic) functions demonstrate some of the benefits provided by collections. They are standard mechanisms available to any developer who needs to manage a group of related items. Some examples of the collection algorithms provided with Java are as follows.

sort: Organize a list based on the natural order of the objects in the list, or based on a comparator (a user-defined ordering method). Sort always orders a list in ascending order.

reverse: The same function as `sort()`, but the elements are organized in descending order.

fill: Populate a list with copies of the same element. Existing elements in the list will be overwritten, so this method is very useful when you want to reinitialize a list.

replaceAll: Replaces all instances of a particular value with another.

copy: Copy elements from a list into another list. The target list must be large enough to hold all the elements, but if the target list is larger, then the extra elements will not be affected.

swap: Swaps some of the elements in a list.

binarySearch: Examine a list to find a particular element. If the element is contained in the list, then its index position will be returned. Otherwise, the negative value of the position it can be inserted in is returned.

min and max: Examine a collection to find the element with the lowest or highest value. As with the sort algorithm, the natural ordering of the elements can be used, or a user-defined comparator can be used to compare one element to another.

Ordering: Elements in a list-type collection are always ordered in some fashion. By default, this is the natural order of the objects. Java defines an interface (`Comparable`) that defines how objects of a given class should be compared. Classes that implement this interface use the `compareTo()` method to compare two elements in a collection. Many of the original Java classes have been retrofitted to support this method.

A specific collection can also define an ordering method unique to that collection type. A user-defined implementation of the `Comparator` interface, with its `compare()` and `equal()` methods, can be passed to the `Collection.sort()` function. In this case, the user-defined compare function will be used to evaluate elements during any sort operations.

Suppose you have a class named `Pupil`, and it has two properties, `name` and `grade`. When you place a `Pupil` object into a collection, you may want to make sure they are sorted by name. To do this, you can implement the `Comparator` interface in `Pupil`.

```
public class Pupil implements Comparator
{
     public String name;
     public String grade;
     public int compareTo (Object o) {
          Pupil p = (Pupil) o;
           //  Compare the name property of this Pupil,
          //  with another, and return the result.
           return name.compareTo (p.name);
     }
  ...
```

The SortedSet implementation extends the basic Set implementation. This collection type orders its elements in some fashion but does not allow duplicate elements. The default `Comparator` is used to compare two elements (and therefore provide ordering), but you can write a specialized `Comparator` unique to your requirements.

The existence of a `Comparator` implies that all elements in a SortedSet collection must be comparable with each other. To do this, either the `compareTo` method of the element (that is, `element1.compareTo (element2)`) or the `compare()` method of the `Comparator` (that is, `Comparator.compare()`) will be used.

Exercises: Java Collections

It's time to visit the example classes and try out all these new ideas. You'll start with the Set collection type and then visit the List and the Map type.

1. Create a new class called `Pupil` that looks like the following class. You will use this class for the exercises with `ArrayList` and `HashMap`.

```
public class Pupil
{
     protected String name;
     protected String grade;

     public Pupil(String name, String grade)      {
          this.name = name;
          this.grade = grade;
     }

     public String getName()      {
          return name;
     }

     public String getGrade()      {
          return grade;
     }

     public void setName(String name)      {
          this.name = name;
     }

     public void setGrade(String grade)      {
          this.grade = grade;
     }
}
```

2. Create an `ArrayList` and three instances of `Pupil`.

```
// Create an ArrayList.
                      ArrayList<Pupil> list = new ArrayList<Pupil> ();

// Create three instances of Pupil.
          Pupil pupilOne = new Pupil("Susan", "A");
          Pupil pupilTwo = new Pupil("Tom", "B");
          Pupil pupilThree = new Pupil("Mary", "C");
```

3. Let's add the instance of `Pupil` to the `ArrayList`.

```
// Add the three instances of Pupil to the ArrayList.
          list.add(pupilOne);
          list.add(pupilTwo);
          list.add(pupilThree);
```

4. Let's create a `HashMap`.

```
// Create a HashMap.
     HashMap<String, Pupil> map = new HashMap<String, Pupil> ();
```

5. Let's retrieve the objects from the `ArrayList`. As you do so, you are going to display their names and grades, and then you are going to add them to the `HashMap` using their names as a key.

```
          for (int i = 0; i < list.size(); i++)
          {
// Retrieve the pupils from the list, print their names.
               Pupil pupil = list.get(i);
              System.out.println(pupil.getName() + " earned a "
+ pupil.getGrade());
// Add them to the HashMap.
               map.put(pupil.getName(), pupil);
          }
```

6. You are now going to retrieve Tom's record and change his grade.

```
// Retrieves Tom from the HashMap and changes his grade.
          Pupil pupilTom = map.get("Tom");
          pupilTom.setGrade("A");
```

7. You are now going to redisplay the list.

```
// Redisplay the list.
          for (int i = 0; i < list.size(); i++)
          {
               Pupil pupil = list.get(i);
               System.out.println(pupil.getName() + " earned a "
+
                    pupil.getGrade());
           }
```

8. Compile the program and execute it. Your output should look like this:

```
Susan earned a A
Tom earned a B
Mary earned a C
Susan earned a A
Tom earned a A
Mary earned a C
```

9. You are now going to redisplay the list using the `for...each` syntax.

```
// Redisplay the list using the simpler for : each syntax
          for (Pupil pupil : list)
          {
               System.out.println(pupil.getName() + " earned a "
+
                  pupil.getGrade());
           }
```

10. Compile the program and execute it. Your output should look like this:

```
Susan earned a A
Tom earned a B
Mary earned a C
Susan earned a A
Tom earned a A
Mary earned a C
Susan earned a A
Tom earned a A
Mary earned a C
```

Reviewing the Exercises

ArrayList and HashMap (and the older legacy classes, Vector and Hashtable) are used quite commonly in Java development. These classes and the other collection classes provide an easy-to-use facility for managing objects in memory. Collection objects, like ArrayList and HashMap, really contain references to objects. ArrayList references objects in an ordered fashion, whereas HashMap references objects in the keyed manner.

An important thing to realize from this exercise is that both collection objects, the ArrayList and the HashMap, contain just the references to the Pupil objects. They do not contain copies of the objects. So when you retrieve the Tom Pupil object from the HashMap and change its grade, you are changing the one and only copy of this object, as it exists in memory. When you display the list the second time, Tom's grade is now an A because the ArrayList is referencing the same object that you retrieved from the HashMap.

12 Other Java Topics

In This Chapter

- Graphical User Interface Development
- Properties Files
- Java Utilities
- Exercises
- Reviewing the Exercises

This chapter explores a variety of miscellaneous topics a Java programmer needs to understand. Included are a brief overview of graphical user interface development and a discussion of properties files and Java utilities.

Graphical User Interface Development

Graphical user interface (GUI) development is a big topic, but I will touch on it only briefly. The basic GUI functionality for Java is contained in the Abstract Window Toolkit (AWT), released with the earliest versions of Java. AWT is a standard interface that Java applications (or applets) can use to create and interact with

GUIs. The AWT interfaces talk directly to the underlying graphical operating system interfaces. Since AWT is ported to every Java environment, any Java application that uses AWT can display GUIs in any Java environment, although the look and feel of the application may vary.

So far it sounds good, but because of several problems, AWT has been regarded by many as the weakest part of the original Java environment. The primary problem is that AWT was designed to the lowest common denominator of graphical environments. Important capabilities (e.g., tabbed views, spinners, complex layouts) required by graphical applications are not available in AWT.

Sun improved on AWT with the newer Java Foundation Classes (JFC), released as a patch to the Java Development Kit (JDK) 1.1. These APIs are grouped in a package named for the code name for the internal development project at Sun (Swing). The Swing package is meant to replace the AWT. Most applications that have Java UI's use Swing instead of AWT.

Applications that use Swing have access to more control over the user interface. For example, an application can now manage borders around graphical components, and components do not all need to be rectangular. Swing brings with it a more robust event model and widgets not available in AWT, such as table and tree control. Swing is not dependant on the native components and, therefore, the look and feel of Swing applications are independent of the platform.

For development of new Java applications, Swing and JFC are clearly the best choices available. Yet the learning curve is relatively steep if you have no experience in GUI development. Following are some of the basic concepts and knowledge you need to get started.

The javax.swing and javax.swing.event packages contain most of the classes you will need, but you will also use java.awt and java.awt.event packages extensively because Swing is built on or uses AWT.

The JFrame class is used to create the main frame window of an application. You can add a menu bar to it using JMenuBar or add a tool bar using JToolBar. You can populate the frame with buttons, combo boxes, text fields, and other GUI elements. A layout manager is generally used to control the appearance. Various layout managers are defined in java.awt.

Figure 12.1 describes how these various components combine to create a Swing user interface.

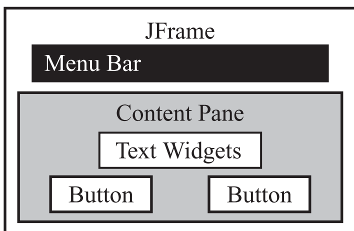


FIGURE 12.1
Swing user interface.

The JFC is an event-driven model. Once you create the main window and populate it with GUI elements, the program operates by reacting to the various events for which you have registered interest. The way you show interest is to implement listener interfaces and attach the listener to GUI elements. For example, you might code an `ActionListener` and attach it to a button. When the button is clicked, the `actionPerformed()` method in your listener is called with information about the event. To understand how to do all this, you should go to the Java Sun Web site and find the tutorials on constructing GUIs.

Properties Files

In many environments where COBOL is used, important runtime information exists outside the program. Most of the time, the actual file names that are the target of execution and other parameters to control execution are defined in job control language and passed to COBOL by the operating system. This allows the same program to be used with different files, such as test and production files, or with various execution options, without having to recompile the program.

One alternative for a Java program is to use command-line arguments. This can become quite cumbersome if the number of arguments becomes extensive. Not only do you have difficulty editing the command line, you also have the problem of parsing and interpreting the arguments on the program side.

A better alternative is a properties file. In a Java environment, a properties file is often used to supply this runtime information. A *properties file* is simply a text file that consists of name-value pairs separated by the equals (=) operator. It is much like a `Hashtable` written in text and, in fact, the `Properties` class that you use to read and write the properties file extends `Hashtable`. Here are the contents of a simple properties file:

```
# my property file containing runtime file names
inputfile=testin.fil
outputfile=testout.fil
```

The file has two properties: `inputfile` and `outputfile`. The line with the `#` is simply a comment. To load the properties, you write code like this:

```
Properties props = new Properties();
props.load(new FileInputStream("myproperties.prop"));
```

Once the properties are loaded, you can access them like this:

```
// returns "testin.fil"
String inputFilename = props.getProperty("inputfile");
// returns "testout.fil"
String outputtFilename = props.getProperty("outputfile");
```

Using this technique, the program could go into production and use production files simply by editing the properties file.

JAVA UTILITIES

Java provides a number of utilities to assist in your development processes. I'll cover a few here.

JAR UTILITY

The Java Archive utility is used for bundling classes, images, and other resources that compose a Java project. The Java Archive utility was discussed in Chapter 2 from the standpoint of a user of a jar file. Here I'm going to discuss how to create one.

Internally, a jar file is much like a zip file. It contains entries for files and directories, and it contains the compressed content of the files. Packaging applications in jar files simplifies distribution and reduces the network load when classes are passed over the Internet or through a network.

There are two primary differences between a jar file and a zip file: A jar file can be signed with a digital signature to provide security and authentication, and a jar file contains a manifest that carries information about the content of the file and how it is to be handled.

The jar utility program comes with the SDK. It is the tool used to create and manage jar files. Like the rest of the SDK, it must be run from a DOS command window. The simplest format for using the jar program is as follows:

```
➔ jar        -cf      ajarfile        *.class
```

This command copies all the classes in the current directory to the jar file named `ajarfile`. If any subdirectories exist in the current directory, that directory is processed recursively (that is, any class files in a subdirectory are included in the jar file, along with the subdirectory name). In addition, a manifest is created and included in the jar file. It describes the contents of the jar file.

The following command updates an existing jar file with all the class files in the current directory:

```
➔ jar -uf     ajarfile  *.class
```

The next command lists the contents of the jar file `ajarfile`:

```
➔ jar -tf     ajarfile
```

or with more details:

```
➔ jar-tvf    ajarfile
```

After you've created a jar file, you can use it by treating it like a subdirectory in your CLASSPATH directory. The CLASSPATH variable lists directories from which any of these three class container types may exit:

- The directory location of actual class files.
- The initial directory for a package (that is, further subdirectories that may contain class files).
- The directory location where a jar file exists (which contains class files, perhaps in subdirectories inside the jar file).

For example, the classes in a jar file named ajarfile.jar can be included in the CLASSPATH option when running a Java program, as follows:

```
java –cp c:\java4cobolm,c:\java4cobol\ajarfile HelloWorld
```

In the case of an applet, a jar file is identified with the `archive=` parameter of the `applet` tag, as follows:

```
<applet
CODEBASE = codebaseURL
code=HelloWorld
archive="ajarfile"
width=200
height=200>
</applet>
```

As with regular class files, an applet's jar filename is relative to the CODEBASE directory.

Javadoc Utility

Javadoc is a utility that allows the programmer to document the code in the source and then easily generate the documentation in a readable format. Javadoc goes well beyond the comment feature in COBOL, although it also can serve the same purpose. The difference with Javadoc is that the utility has the ability to process specially formatted comments in Java source code and produce external documentation, including a complete class structure showing inheritance and other relationships between the classes.

Javadoc is a utility provided in the standard Java SDK that parses source code and, by default, automatically generates HTML output that documents the Java code. The executable code for Javadoc is found in the bin directory in which the SDK is installed. In reality, the Javadoc executable is a starter application that creates a Java VM and invokes a doclet to do the work of generating the output documentation.

You run Javadoc by executing the program at the command line and passing a directory or list of directories to the utility. This action produces HTML pages that link together to document and index the classes found in the directories. It does not automatically copy the various images referenced in the HTML to the proper location, so you will need to do that manually.

Standard Javadoc comments start with `/**` and end with `*/`. Note the double asterisks at the start that distinguish the Javadoc comment from an ordinary multiline Java comment. Comments can be plain text or can include HTML code. Here's how a comment might look embedded in Java code:

```
/**  The purpose of this comment is to help in the automatic generation of
documents. A programmer can insert (well-written) descriptions of classes
and members. These will be extracted and placed into a published document
in HTML format.
*/
```

Javadoc comments are recognized when placed immediately before class, interface, constructor, method, or field declarations. Documentation comments placed in the body of a method, however useful they may be, are ignored.

Even if you do not include any comments, Javadoc will produce a listing of classes and methods. By default, it lists public and protected methods and members and excludes the private ones. This can be changed with command-line options.

Javadoc comments can also be marked with special markup tags, which provide for additional formatting and cross-reference control for the generated document. The following markup tags are used to document the interface of public methods:

```
@param
@return
@throws
```

The following markup tags are used to document packages or classes that contain public methods:

```
@version
```

It is a good idea to place Javadoc comments (with at least these tags) at the beginning of your classes and methods. Java programmers will expect to see them, and several useful tools are available that will process this type of program comment specification. The output of the Javadoc tool included with the SDK is especially nice looking and useful.

The advantage of in-program documentation is that, since the code and the documentation are in the same file, the chances that the documentation will match the code are improved. In addition, the public class interface definitions will always be accurate because the tools automatically generate this information from the Java source code.

One disadvantage is that you still must rely on the programmer to type in the comments and specifications. There is no guarantee that the comments actually do match the code. Another problem is that Java editing environments are not WYSIWYG graphical editors. Generating nice-looking end-user documentation with Javadoc and HTML markups takes a lot of work, even if the end users are developers and so, presumably, have lower expectations.

Most people find it useful to include comments before method definitions and for the class as a whole. The comments may contain special tags that Javadoc understands and can use to customize the output. These tags follow the comments.

The following shows a lightly commented source code fragment:

```
/**
* This method returns a string describing DocTest.
** @return a string description of DocTest
*/

public String getDescription()
{
...
}
```

When Javadoc creates the documentation for the class, it includes the description "This method returns a string describing DocTest." as the description for the method `getDescription()` and adds a return label in bold with the description "a string description of DocTest" after it.

The leading asterisk characters on each line are not necessary. If you omit the leading asterisk on a line, however, all leading white space is removed. Without leading asterisks, the indents are lost in the generated documents. Here are some common tags and their uses:

@deprecated	Indicates that this section of the API is deprecated
@author	Indicates the author name (requires the author command-line flag)
@param	Used to describe a method parameter
@return	Used to describe a return value
@throws or @exception	Synonymous tags used to list exceptions thrown
@version	Place your versions number here

The default doclet (called *standard*) by default produces HTML documentation. Most people do not go beyond usage of the standard doclet, but it is possible to override the standard doclet with a custom doclet. In fact, several other doclets have been written. Sun has a list of experimental and third-party doclets at its Web site.

EXERCISES

Even though you haven't explored in depth how to create a GUI, you are going to walk through a simple example. To start, copy the file SimpleFrame.java from the Chapter12 subdirectory in the Exercises directory on the CD-ROM to your java4cobol directory.

ON THE DVD

1. Compile the `SimpleFrame` class. `SimpleFrame` takes care of the work of creating a frame window and various listeners so that the window will respond properly to messages. It also provides some utility methods for creating buttons and menu items. You won't be using a menu in this exercise, but you will be creating a button.
2. Create a new Java file called MyFrame.java. Add the following import statements to the file.

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
```

3. Write the code to create the `MyFrame` class. Notice that `MyFrame` must extend `SimpleFrame` for this exercise to compile correctly.

```
public class MyFrame extends SimpleFrame
{
}
```

4. Add a constructor. This constructor invokes the constructor in SimpleFrame. The text "My Frame" will become the title on the window.

```
public MyFrame()
{
    super("My Frame");
}
```

5. Override the `buildGUI()` method in `SimpleFrame`. This method is automatically invoked in the `SimpleFrame` constructor.

```
public void buildGUI()
{
// Call the createButton method in SimpleFrame.
    JButton button = createButton("Click me", "Click me");
```

```
// Tell the frame how to lay out components you add.
    getContentPane().setLayout(new FlowLayout());
// Add the button to the frame.
    getContentPane().add(button);
}
```

6. Override the `actionPerformed()` method in `SimpleFrame`. `SimpleFrame` and, by extension, `MyFrame` are listeners for action events. The `create-Button()` method added the `ActionListener` in the button creation process, so any action taken on the button (that is, clicking it) automatically calls this method. If you had multiple buttons, the action command would allow you to distinguish which button had been clicked.

```
public void actionPerformed(ActionEvent e)
{
// Get the action command.
    String command = e.getActionCommand();
// Pop up a message when the button is clicked.
    if (command.equals("Click me"))
        informationMessage("I've been clicked", "Click Message");
}
```

7. Finally, code a main method that can be executed.

```
static public void main(String [] args)
{
    new MyFrame();
}
```

8. Compile `MyFrame` and execute it. A window entitled “My Frame” should pop up with a single button centered in the upper part of the window. When you click the button, the message “I’ve been clicked” should appear.
9. The file SimpleFrame.java has had Javadoc comments added to it. As a final task, go to the java4cobol directory and key in the following command:

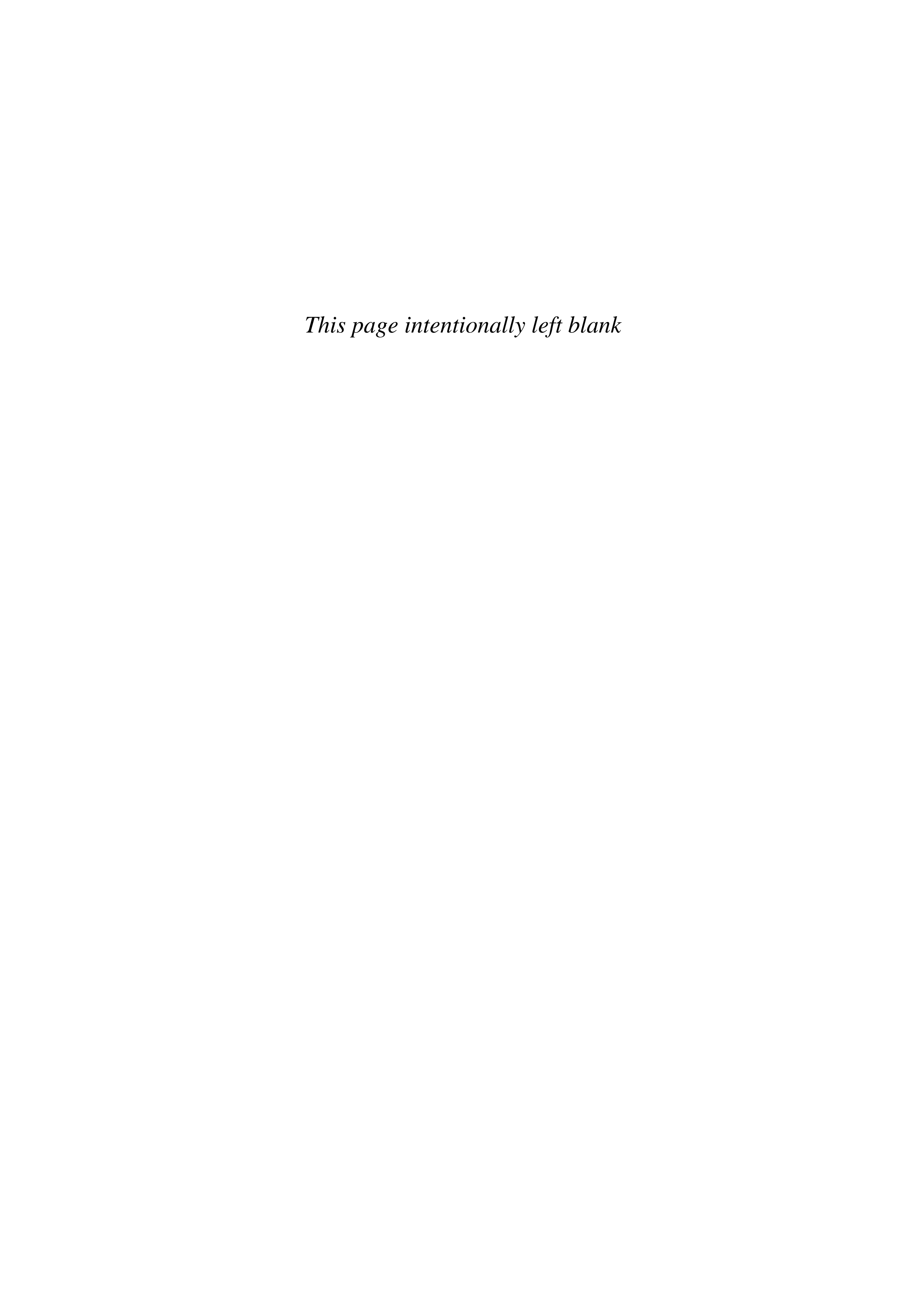
```
➔javadoc SimpleFrame.java
```

If your system paths are set up correctly, the Javadoc should run and create HTML that documents the `SimpleFrame` class. With your browser, open the index.html file that was created in the java4cobol directory and examine the output.

Reviewing the Exercises

In this exercise, you created a simple window by extending a class called `SimpleFrame`. In order to understand more fully what is required to develop a fully functional GUI application, look at the code in `SimpleFrame` and use it as a basis for building more complex applications.

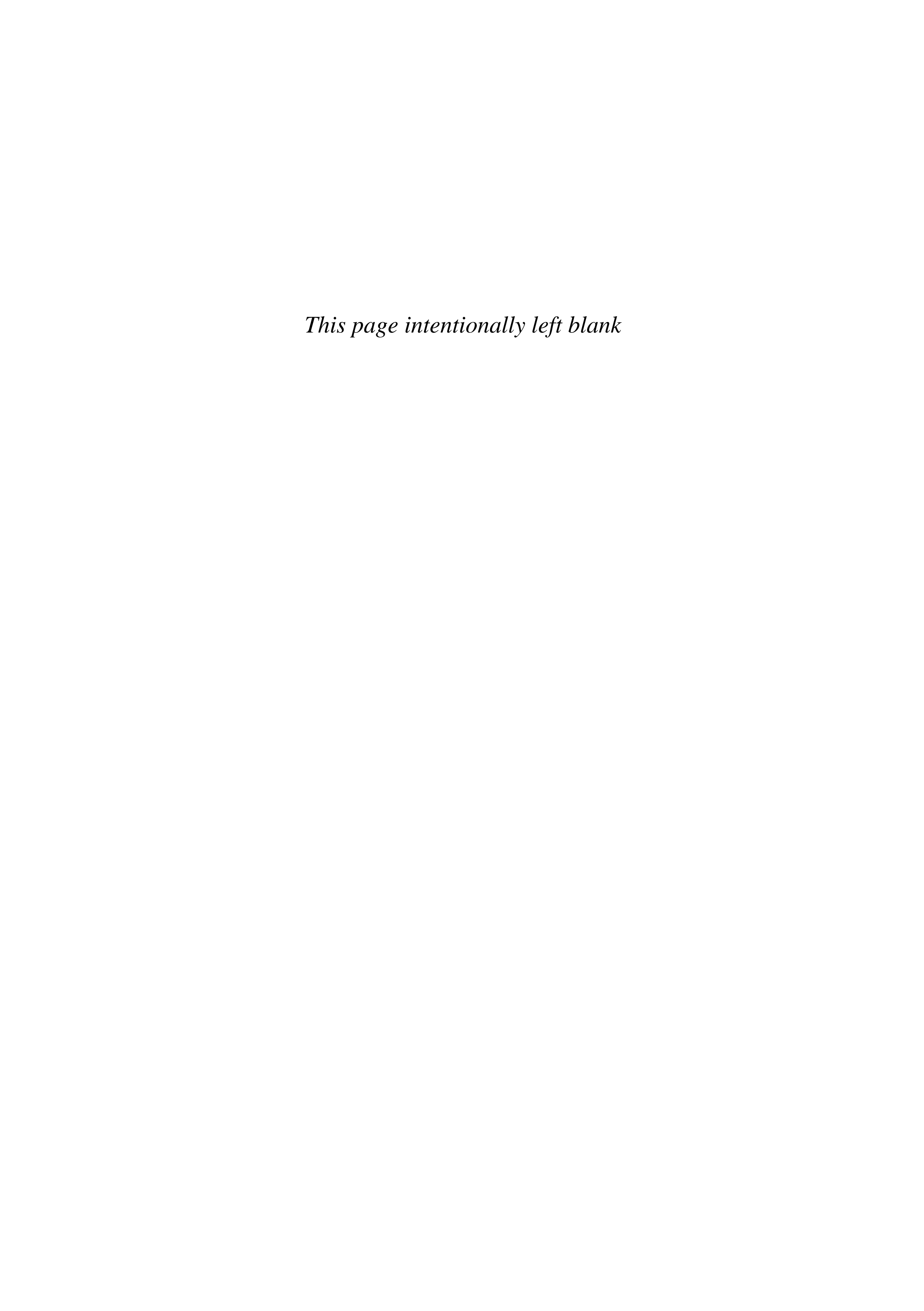
You also generated the Javadoc that was already supplied in the `SimpleFrame` source. You might want to add Javadoc comments to your `MyFrame` source and generate the Javadoc for that class.



This page intentionally left blank

Part III

Introducing Enterprise Java


This page intentionally left blank

13 Java Database Connectivity

In This Chapter

- How JDBC Works
- Connecting to the Database
- Querying a Table
- Inserting, Updating, and Deleting
- Configuring the JDBC-ODBC Bridge
- Exercises
- Reviewing the Exercises

JDBC is a call-level interface built with Java for performing operations on relational databases. It resembles the Open Database Connectivity (ODBC) standard created by Microsoft, but it is generally easier to use. The range of operations it can perform is extremely broad: queries, inserts, updates, deletes, and stored procedure calls. It also has an API that allows access to database metadata, which includes capabilities for obtaining information about tables and columns.

Although JDBC is included in the Java Standard Edition SDK, I am going to discuss it as a part of Enterprise Java because, in most cases, Enterprise Java requires access to relational database systems. Much enterprise technology simplifies database access and provides pooling mechanisms to minimize the overhead and cost of establishing and releasing database connection.

How JDBC Works

Most business applications today need to store information in relational databases. Business applications written in Java are no exception. The standard Java database access technology is JDBC. JDBC is essentially a variation on ODBC with some Java bindings. It is designed as a Java API to Structured Query Language (SQL) data.

JDBC is just a specification; it is up to the various database vendors or third parties to create JDBC drivers for a particular database. In some cases, a JDBC-ODBC bridge can be used, but likely not for production purposes.

JDBC provides a fairly complete set of SQL-friendly data access mechanisms, such as scrollable result sets, absolute and relative positioning (in the result set), access to stored procedures, and data type conversions. Most SQL-92 (Entry Level 2) statements are supported in JDBC.

In order to access relational databases through Java and JDBC, we need a JDBC driver that can perform whatever communications and translations are required to act upon the database. A driver is a set of Java classes (usually in the form of a jar file) provided by the vendor that adopts the standard behaviors required for all JDBC drivers. These standard behaviors are defined in the various Java interfaces that comprise the JDBC API. The API, in addition, defines certain optional behaviors that a driver may choose to support. The API provides the ability to query the capabilities of the driver once it is loaded.

How the driver goes about its business behind the scenes is completely up to the driver. JDBC drivers may communicate from Java to native code, which then communicates with the database.

Other drivers use Java to communicate with the database using middleware or through direct communication with the database. These do not require that any native code be installed on the client machine.

Type 1 and Type 2 JDBC drivers fall into the first group of drivers. They require native code on the client machine. A Type 1 driver is a JDBC-ODBC bridge. It translates the JDBC calls into ODBC calls and uses ODBC to access the database. A Type 2 driver converts JDBC calls into the native API for the database. This requires that binary code from the database vendor be installed and configured properly on the client.

Type 3 and Type 4 JDBC drivers fall into the second group of drivers. They do not require native code on the client machine. They require only the Java packages and classes that support the driver. A Type 3 JDBC driver relies on a middleware server to handle its communications with the database. A Type 4 JDBC driver communicates directly with the database from Java without any intermediary software other than the communication protocol itself. Figure 13.1 illustrates how JDBC works with the various driver types.

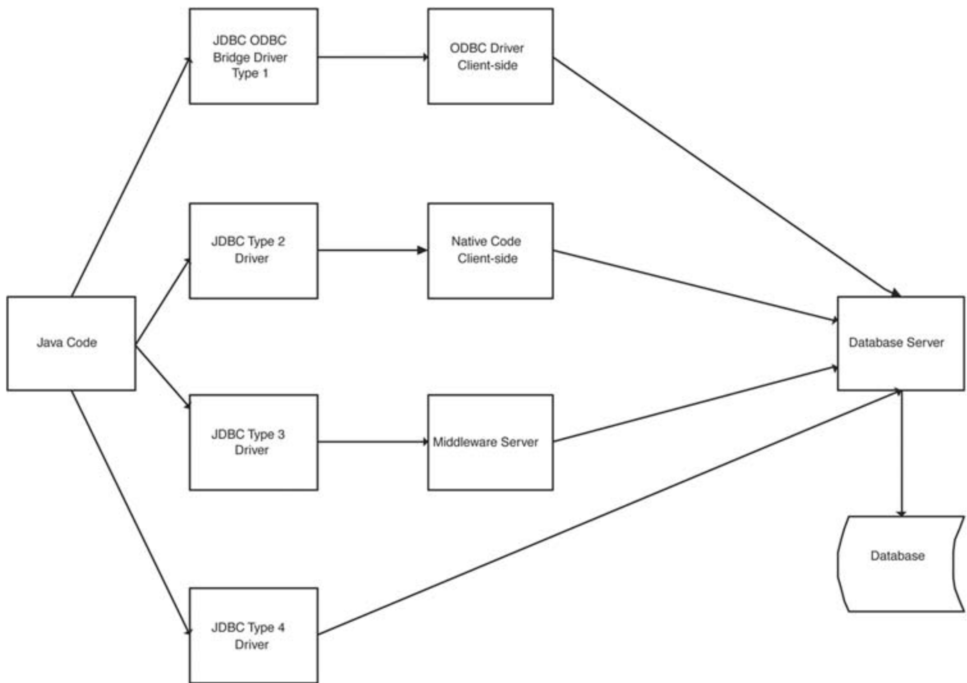


FIGURE 13.1
How JDBC works.

Sometimes you have a choice about which type of driver to use. You may also have a choice about vendors. Clearly Type 3 and Type 4 drivers are easier to deploy to a large number of client machines, since they do not require any code other than the driver itself on the machines. A Type 1 driver is useful for learning how JDBC works, including simple testing and prototyping. For production, a Type 1 driver is the worst choice and should be considered only when no other option is available. Not only does it require ODBC software correctly configured on the machine, it also has the overhead of ODBC on top of whatever overhead JDBC carries. Beyond this, generalizations become more difficult. A vendor's Type 2 driver may perform dramatically better than its Type 4 driver. If it is not required to be deployed to many client machines or if its performance advantages outweigh the deployment difficulties, the Type 2 driver could be the better choice. A Type 3 driver with its middleware server might be the best choice in a heterogeneous environment with many different databases, particularly if it performs well. With a Type 3 driver, one driver could be deployed to all client machines to provide access to all the databases in the enterprise.

Connecting to the Database

The typical JDBC sequence required to access the database is slightly involved but quite straightforward after some practice. You create a connection to the database; you perform some actions, such as querying or updating; then you free the resources and the connections.

The first step is to create a connection to the database. To do this you need to have the JDBC driver class in your CLASSPATH. Most often the driver will be in a jar file supplied by the JDBC developer or vendor. To establish the connection, first you must load the driver. Then you can use the `DriverManager` class to establish the actual connection.

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection dbConnection =
DriverManager.getConnection("jdbc:odbc:mydata");
```

Although I have not discussed the class named `Class`, don't be confused by the `Class.forName()` that is used to accomplish the driver load. Every VM has a class loader that performs the task of locating the actual code on disk, performing required integrity and security checks on it, and loading it into memory. As a complex program executes using many different classes, the driver-loading process takes place transparently behind the scenes. The `forName()` method is one way of loading a class explicitly in code (similar to an import statement that is resolved at runtime). If the class loader is unable to find the class, it will throw a `ClassNotFoundException`; otherwise, the class loader locates the class in the CLASSPATH and performs the same tasks it would for any other class. In the preceding code, you are loading the Sun JDBC-ODBC Bridge Driver explicitly by passing the correct class name to the `forName()` method.

After the driver is loaded, you can obtain a connection to a specific database that the driver supports by using the `DriverManager`. When the driver is loaded, it interacts with the `DriverManager` class to register itself. This registration process allows the `DriverManager` to map the URL string that is passed from the `getConnection()` method to a driver. The driver interprets the URL string and establishes the connection to the actual database.

The required conventions for the URL string are different for each driver, so you must consult the driver documentation for the exact parameters needed to establish the connection. In the example (`"jdbc:odbc:mydata"`), you are using JDBC to connect to an ODBC data source called "mydata." The JDBC-ODBC Bridge Driver requires the information in this format to establish the connection. If you are unfamiliar with ODBC and data sources, don't worry. I will cover the topic more thoroughly later in this chapter.

The `Connection` class returned by the driver is a factory class that generates many of the other classes required to perform database operations with JDBC. In particular, the `Connection` class provides the facilities for processing SQL statements, either by executing the statement directly or by returning an instance of a class implementing `Statement` or one of the `Statement` subinterfaces.

A factory class *is a popular design pattern. A factory class creates an object for you. The factory pattern provides more control than the standard* `new` *operator. For example, you might want to globally control each instance of a* `Connection` *object so that when the system is rebooting, you can attempt to close all connections. If all* `Connection` *objects are created by a single factory class, then that class will know about each* `Connection` *that was created. Therefore, it can manage them properly when the system is rebooting.*

The `Connection` class also controls the various aspects of database transactions. By default, new connections are in *auto-commit mode*. This means that each statement that modifies the database is automatically committed if it succeeds. Unless your requirements are relatively simple, auto-commit mode is not what you want. If you require the data in multiple tables to be synchronized with each other, you will want to turn auto-commit off, like this:

```
dbConnection.setAutoCommit(false);
```

Once auto-commit is off, you can perform transaction logic as you update the database. Transaction logic, for example, enables you to roll back (or reverse) the changes to one or more tables when the attempted update to another fails. The tradeoff is that you must become very explicit about committing and rolling back the updates. This is handled completely by the `Connection` class. As you might expect, the methods are called `commit()` and `rollback()`.

```
// Post the changes since the last commit or rollback.
dbConnection.commit();
// Reverse the changes since the last commit.
dbConnection.rollback();
```

I will discuss the methods in `Connection` that deal with statement processing later. Some other methods in the `Connection` class are shown in the following table:

Method	Description
getMetaData()	Returns a MetaData object that can be used for obtaining table names and column information.
getWarnings()	Returns a SQLWarning object containing supplemental information about database interactions.
clearWarnings()	Clears the warnings so that future calls to getWarnings() returns null until another warning occurs.

Unless you are writing a traditional client-server application, you might (seldom) actually need to manage database connections. Enterprise Java and server-side Java typically utilize a connection-pooling technology to manage database connections. A connection pool is a cluster of connections that can be reused as necessary to fulfill the needs of applications. Since the cost of establishing connections and maintaining large numbers of connections is significant, a pool reduces this overhead. The `Connection` *returned from the pool-management technology still behaves the same way as the* `Connection` *you would obtain directly from JDBC code.*

Querying a Table

After connecting to the database, you will have created a `Connection` object that will allow you to perform actions upon the database. One of the most common actions is the query.

Java defines a `Statement` interface that you can use to process a variety of SQL statements. You must first create an object of that type, based on your `Connection` object.

```
Statement sqlStmt = dbConnection.createStatement();
```

The next step is to execute an SQL statement and place the results in a `ResultSet`. A `ResultSet` object is one that will contain the rows that are returned for your SQL statement. For example, this statement will select posting account IDs and currency IDs from the `posting_accounts` table and place the results in the `ResultSet` object named `accounts`:

```
    ResultSet rs = sqlStmt.executeQuery(
"SELECT POSTING_ACCOUNT_ID, CURRENCY_ID FROM POSTING_ACCOUNTS WHERE
   COMPANY_ID = 'CC1'");
```

The `ResultSet` class contains a `next()` method. This method positions the cursor (the current position in the result set) in the next available row. When the result set is exhausted, `next()` returns false. This method is similar to the READ NEXT statement in COBOL.

```
while (rs.next()) {}
```

You cannot retrieve any data from the `ResultSet` until you have done the first `next()`. The `next()` method, however, does not actually return any data. To get data into your own variables, you will need to call the proper `getXXX()` method where `XXX` is the specific data type to return (e.g., `getFloat()`, `getLong()`, `getInt()`, etc.). JDBC is a strongly typed interface, meaning that each `get()` method is specific to the data type referenced. There is a separate `getXXX()` method for strings, another for floats, and another for ints.

The statements to retrieve information from the `ResultSet` named `accounts` might look as follows. (Of course, each iteration of the `next` loop as written will overwrite the results of the previous iteration, but I want to keep the example simple.)

```
int postingAccountID;
String currencyID;

while (rs.next()) {
       postingAccountID = acounts.getInt("POSTING_ACCOUNT_ID");
       currencyID = acounts.getString("CURRENCY_ID");
}
```

In the preceding example, you are retrieving data from the result set by passing a column name as a string. You can also retrieve data by passing a column number. Column numbers begin with one rather than zero, so the following code would be equivalent.

```
while (rs.next()) {
        postingAccountID = acounts.getInt (1);
        currencyID = acounts.getString(2);
    }
```

Inserting, Updating, and Deleting

Performing update operations on a database is quite similar to performing queries. A statement is passed as a string to the `Connection` object, and a `Statement` object is returned. The statement can then be executed. The only difference is that the method that executes an update is `executeUpdate()`, whereas the statement to execute the query was `executeQuery()`. The `executeUpdate()` method is used for inserting, updating, or deleting, that is, for all operations that modify the database.

Since updating is so similar to querying from a Java perspective (even though the SQL is quite different), I'll use this opportunity to introduce two new concepts: the prepared statement and parameter handling.

For the database manager to process SQL, two steps must occur. First, the database manager must parse the SQL and decide on an optimization strategy for execution. Second, it must carry out the execution.

If you intend to use the same SQL statement repeatedly, you would like to avoid the overhead of the first step since in every case the parsing and the strategy determination will arrive at the same result. The prepared statement allows you to do this. A *prepared statement* is a precompiled SQL statement with the statement parsed and the optimization strategy already determined. By supplying placeholders in the SQL statement for the values that need to change from execution to execution, you can execute the same statement repeatedly with different values and avoid the overhead of compiling the statement more than once.

Look at the following update statement:

```
String updateSQL = "UPDATE POSTING_ACCOUNTS SET ACCOUNT_BAL =  ?
WHERE COMPANY_ID =  ?";
```

The question marks represent placeholders for parameters that you will supply prior to execution. The parameters are numbered left to right in the statement, beginning with 1. Parameter 1 represents the value that will be used to set the value of ACCOUNT_BAL. Parameter 2 is the value for completing the equal condition in the WHERE clause. You create a `PreparedStatement` by calling the `prepareStatement()` method of `Connection`.

```
PreparedStatement sqlStmt =  dbConnection .prepareStatement(updateSQL);
```

Once you have prepared the statement, you then set the values of your parameters.

```
sqlStmt.setDouble(1, 1000.00);
sqlStmt.setString(2, "C001");
```

Finally, you can execute the update itself.

```
sqlStmt.executeUpdate();
```

You could execute the same statement repeatedly with different values simply by setting the parameters to new values and calling the `executeUpdate()` method. Closing the statement would release the statement; and the statement could not be executed again without going through the preparation process. This same logic can be applied to inserting or deleting rows from a table.

Configuring the JDBC-ODBC Bridge

The JDBC-ODBC Bridge may be an option for you if you do not have a JDBC driver and database and you want to experiment with JDBC. As mentioned previously, the bridge is not desirable for production situations. It carries high overhead and is not fully supported.

The bridge is useful for prototyping and educational purposes if you are developing on a Windows platform. The driver itself comes with the Java SDK. If you have any database with an ODBC driver, you can access it from Java code using the JDBC-ODBC Bridge.

To use the JDBC-ODBC Bridge, follow these steps:

1. Open up the Control Panel from the Settings menu selection on the Start menu.
2. Find the ODBC Data Sources Administrator icon and double-click it. This may be located in the Administrative Tools folder. You should see a list of data sources that have been configured.
3. If you need to add a new data source, click the Add button. You should see a list of ODBC drivers that are installed on your system.
4. Select a driver that supports the database you wish to access through Java. For example, select Microsoft Access Driver.
5. The next dialog box varies by driver. You should supply a data source name, which is the logical name by which the database will be known. If you would like to use the code in the example class without any changes, name the data source “mydatasource.” You may also supply a description of the data source if you want. The Microsoft Access Setup requires you to select a database or to create a new one. For this exercise, select the file named mydatabase.mdb.
6. Once you have completed the setup, the new data source you configured should appear in the master list of data sources.

7. To access the database using Java, enter the following code:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection dbConnection =
driverManager.getConnection("jdbc:odbc:mydatasource");
```

where `mydatasource` is the name of the data source you configured in the ODBC Data Sources Administrator.

EXERCISES

The Chapter 13 subdirectory in the Exercises folder on the CD-ROM contains a Java file called JDBCTest. Included in the same directory is a Microsoft Access database called mydatabase.mdb. In order to use this database and run the examples without modifications, you will need to use Microsoft Access. You will also need to configure an ODBC interface to Access as previously described. A simplified Microsoft Access database comes with Microsoft Excel.

This database has a table called Contacts, which contains some records you can use for this exercise if you do not have access to another database. If you do have access to another database, you can modify the program to retrieve the rows from a different database and table.

1. The program as written uses the JDBC-ODBC Bridge Driver. To use a different driver, you will need to change this line to indicate the correct class name for the driver.

```
private static String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
```

2. The program accesses a datasource called mydatasource. If you want to use the sample database in the directory, you will need to set up a datasource in ODBC that points to the mydatabase.mdb file. Follow the instructions in the previous section. If you use a different database, you may need to change this URL.

```
// Needs an ODBC datasource with name mydatasource to be set up.
private static String url = "jdbc:odbc:mydatasource";
```

3. The program uses the `getConnection()` method in the sample program to get the connection. It then creates a statement.

```
Connection connection = getConnection();
Statement stmt = connection.createStatement();
```

4. You query the Contacts table, retrieving all rows and all columns.

```
// Get everything from Contacts table in mydatabase.mdb.
ResultSet rs = stmt.executeQuery("select * from contacts");
```

5. You iterate through the result set, display the results, and then free all the resources.

```
// Iterate result set and print rows.
while (rs.next())
{
    System.out.println("name=" + rs.getString(1));
    System.out.println("phone num=" + rs.getString(2));
    System.out.println("email=" + rs.getString(3));
    System.out.println("note=" + rs.getString(4));
}
// Close down.
rs.close();
stmt.close();
closeConnection();
```

6. The output for the sample table looks like this:

```
name=Susan Cobb
phone num=777-555-1212
company=Sun
email=scobb@sun.com
note=Not a real person
name=Bill Smith
phone num=121-333-3433
company=Empire Bank
email=bsmith@empire.com
note=Not a real person
name=John Smith
phone num=444-343-3333
company=XYZ Corp
email=jsmith@xyz.com
note=Not a real person
```

Reviewing the Exercises

JDBC provides a robust and relatively simple interface for accessing most industry standard relational databases. The basic logic involved in database access is as follows:

```
Load the driver specific to the database.
Establish a connection to the database.
Perform queries and updates using Statements.
Examine and process the results of queries, or commit the updates.
Free the statement resources and free the connection to the database.
```

14 Servlets and Java Server Pages

In This Chapter

- Browsers and Web Servers
- The Servlet as Transaction Processor
- Servlet Protocol
- Java Server Pages
- Getting Started with Servlets and JSPs
- Exercises
- Reviewing the Exercises

For a variety of reasons, many enterprises are using the Internet as a model for new application development. The major reason is ease of deployment. Every machine with a browser and connection to the server has immediate access to the application. No messy software has to be installed and kept up to date on each machine. The browser provides a feature-rich environment for sophisticated and user-friendly applications. As more and more people become Internet savvy, education and training costs are dramatically reduced.

Java plays a key role in this new paradigm. Java running on the server provides a robust and high-performance environment for developing and running applications. Although servlets and Java Server Pages (JSPs) can be used for both presentation and business logic, good design principles usually require a separation of the two. Servlets and JSPs in most designs are used for presentation logic. Business logic in the full-blown Java Enterprise Edition (JEE) environment is usually handled by Enterprise JavaBeans, the subject of the next chapter.

Browsers and Web Servers

To do Web development, you need to understand how Web servers work and how they interact with their clients, the browsers. Once this understanding is in place, you will readily see how servlets and JSPs fit into the picture.

The simplest requirement of a Web server is to serve static HTML pages and images. In doing this activity, Web servers are really file servers. The files they serve are text files containing HTML and their associated images and resources. The clients they serve are usually browsers. The protocol for the communication is Hypertext Transfer Protocol (HTTP). Modern Web servers, of course, can do much more than simply serve files, but let's stay with the simplest communication for the moment.

The client, usually a browser, opens a connection to the server. After it gets a connection, it sends a request that is formatted correctly for HTTP. The server responds according to what was requested, in a form that is formatted correctly for the protocol. The connection is closed.

Each request from the client and response by the server is an autonomous communication. Each runs independently from all that went before it and from all that came after it. The transaction is stateless. No persistent connection exists between the client and the Web server. No "memory" of other transactions is built into the protocol. The protocol was designed this way for speed.

Now you'll dissect a simple transaction to a deeper level. Let's pretend you have installed a Web server on your machine and have placed a simple HTML document called resume.html in the root directory that the Web server uses to serve files. This directory is configurable for each Web server. The machine is running at IP address 90.121.111.5.

First, the client starts a request of the Web server by opening a TCP/IP connection to the Web server.

The address of the connection is expressed as a URL. The URL is the standard addressing system used to locate resources on the Internet. The syntax for a URL is [protocol]:[resource]. The protocol is the access scheme or method. Common protocols are http, ftp, gopher, and WAIS.

The part after the colon is interpreted according to the access scheme. In general, two slashes after the colon introduce a hostname or hostname:port. The hostname can be an IP address, such as 90.121.111.5, or a name, such as www.yahoo.com. If it is a name, the name must be translated to a real IP address. A domain name system (DNS) accomplishes this task. The port number is the socket through which the server is listening for connections. In most cases, the port number can be omitted because each protocol has its default port. The browser will attempt to establish the connection on the default port if the port number is absent in the URL. As long as the server is listening on the default port, the connection will be made. The standard port for the HTTP is port 80.

For HTTP, the next part is a pathname. In this simple example, this maps to the pathname of a file on the server. The file can contain any type of data. Browsers know what to do with HTML and GIF and JPEG images. Browsers may pass the file to an external viewer if the appropriate plug-in for the file type has been installed. This is what happens, for example, when you view a PDF file in the Acrobat viewer. Once you have it installed, it is invoked to display the contents of the file. A pound sign (#) following the pathname indicates a particular position on the HTML page.

The URL, after the pathname, may also have a query string preceded by a question mark (?). A query string is used when you are running a program on the Web server, instead of serving a page. So let's stay with the simple example for a moment.

You type the following into your browser's address window: **http://90.121.111.5/resume.html**. The browser tries to open the connection to the Web server running on port 80 at 90.121.111.5.

After the connection is opened, the browser sends the request in correctly formatted HTTP. In the example, the request is for a file called resume.html. The browser looks for and finds the file resume.html in its root directory, where it serves files. The Web server reads the file and creates a response. Included in the response is the output type that is being returned to the browser. A Multipurpose Internet Media Extension (MIME) type placed in the response provides information about

the type of data the browser can expect. In this case, the data type is text/html. This information is included in the header portion of the response. Web servers map file extensions to MIME types. An image file might have the MIME type image/gif.

How the file is rendered by the browser depends on the browser, but only certain types are interpreted directly by most browsers. These include HTML and images in GIF or JPEG format. A file whose type is not recognized directly by the browser may be passed to an external "viewer" application, such as a sound player.

The browser receives the response, reads the HTML, and formats the display. The transaction is complete and the connection between the browser and the server is closed.

As mentioned earlier, the URL after the pathname may have a query string. The query string consists of one or more parameters followed by their values. Each parameter consists of a parameter name followed by an equals sign and the parameter value. Parameters are distinguished from each other by an ampersand (&).

For example, the query `string ?x=5&y=7&z=1+2+3` has three parameters: x, y, and z. The parameter x has the value of 5; y has the value of 7. The parameter z is an array with values 1, 2, and 3.

The types of characters that can be passed in a query string are restricted to alphanumeric characters, certain reserved characters (:/?#"<>%+), and a few other characters ($-_.&+). If you need to pass anything else, it must be encoded. Characters are encoded as a "%" followed by two hexadecimal digits that represent the value of the character.

The browser request can be in one of two forms, GET and POST. A GET request is simply a request for data as identified in the URL. A POST request includes additional information in the request, in the message body. For example, if you enter information in a Web search portal and click the Search button, your search terms are sent in the message body that is included in the POST request to the search Web site.

THE SERVLET AS TRANSACTION PROCESSOR

To oversimplify things a little, a *servlet* is a Java program invoked by a Web server to respond to a request from a browser. A servlet responds to requests from the browser and builds HTML output. It is transaction-based. The request arrives; the servlet processes it and builds the response. The Web server routes the response

back to the browser. The transaction is over. No persistent connection exists between the Web server and the servlet or between the servlet and the browser.

Since a servlet is a Java program, it must run in a Java VM. The VM it runs in is a specialized environment called a *servlet engine*. The role of the servlet engine is to provide the infrastructure and framework for all the servlets that are required for a Web site. The servlet engine loads the servlet, initializes it, invokes it, and routes the generated response back to the Web server.

The servlet and the servlet engine are intimately integrated with the Web server. This integration can take several forms:

1. If the Web server is written in Java, it probably has a servlet engine integrated directly into it.
2. If the Web server is written in some other language, it will need to have an extension to connect the Web server with the servlet engine. In this case, the servlet runs outside the Web server, and the Web server communicates with it through some protocol, usually TCP/IP. Most servlet engines supply extensions or connectors to most of the popular Web servers.
3. Some servlet engines can themselves operate as Web servers. Functionally, this isn't different from being a Web server written in Java with an integrated servlet engine. The difference is that the Web server part of the servlet is usually not highly optimized. The servlet engine works fine as a Web server for development, testing, and low-transaction-volume situations, but it would be better deployed in a high-transaction-volume situation as an extension to a Web server written in native platform code.

Servlet Protocol

A servlet is invoked as a URL, much like a static page or image file is invoked. In the configuration between the Web server and the servlet engine, virtual mappings are created. For example, the Web server may be configured to pass everything with the directory name of servlet to the servlet engine. The servlet engine then maps the name following the directory name to the actual servlet class that is to process the request. For example, the request for http://www.mywebsite.com/servlet/Start goes to the Web server. The Web server examines the request and sees the directory name servlet. It knows it must pass the request to the servlet engine. The servlet engine examines the request and sees the name Start after the virtual directory name. In its configuration, for example, it knows it must pass the request to "com.mywebsite.servlets.WelcomeServlet." This path is illustrated in Figure 14.1.

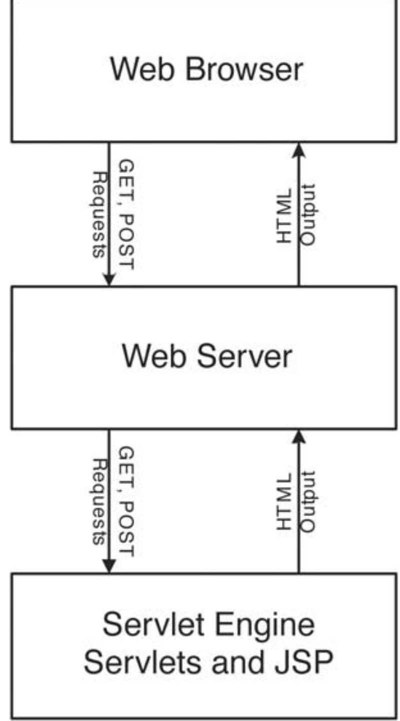


FIGURE 14.1
How the browser interacts with the Web server.

Servlets primarily receive HTML GET and POST requests that originate from browser forms. GET requests pass their parameters on the command line, for example, `http://www.mywebsite.com/servlet/Start?parm1=1&parm2=2`. POST requests pass their parameters in a data stream. In either case, the parameters are passed as name-value pairs and are encoded.

To create a servlet, you extend `HttpServlet` and override the `service()` or the `doGet()/doPut()` methods. You will need to import javax.servlet and javax.servlet.http packages to have this compile. These packages are found in the JEE SDK. The following is a simple servlet:

```
public class WelcomeServlet extends HttpServlet
{
public void doGet(HttpServletRequest req, HttpServletResponse res) throws
   IOException, ServletException {
   res.setContentType("text/html");
          PrintWriter out = res.getWriter();
          out.println("<html>");
          out.println("<body>");
          out.println("<head>");
          out.println("<title>" + "Welcome Page" + "</title>");
          out.println("</head>");
          out.println("<body>");
          out.println("<h1> Welcome Page </h1><hr>“);
          out.println("Welcome to the web site");
          out.println("</body>");
          out.println("</html>");
}
}
```

This servlet is very simple. It sets the content type in the response header so that the browser knows what type of data (that is, the MIME type) will follow. It then gets the `PrintWriter` associated with the `HttpResponse` object and uses the writer to write the HTML to the browser. The Web server is ultimately responsible for delivering the stream of HTML to the browser.

The preceding code would work fine if your servlet received only GET requests. The `doGet()` method would not be called for a POST request. You would need to override `doPost()` to receive the POST request.

The servlet engine initializes the servlet once when it is first loaded. The `init()` method can receive runtime parameters just like applets and do any one-time initialization required for the servlet. Once they are loaded, servlets stay loaded unless the servlet engine decides to shut them down. Servlets are handed requests from a thread pool, which is both good and bad. The good part is that the startup overhead for new requests is minimal. The bad part is that you need to be concerned about trashing your internal data if two service requests start executing simultaneously.

Remember that only one copy of your servlet is in the memory of the VM. If browser number 1's request comes to the servlet and you set a value in your class instance during the processing of that request, that value will still be set when browser number 2's request is handed to your servlet. In reality, the situation is even more complicated. Depending upon the thread-switching mechanisms in place in the VM, request number 2 might actually change the value of your variables even while request number 1 is still being processed. You might wonder how anyone can ever code in this environment. Although several solutions exist to the problem of reentrancy, the most common solution and the easiest is to create a session.

The servlet engine has the ability to create a `Session` object that is associated with the session between the browser and the Web server. A session is a time-sensitive association between an instance of a browser and the Web site. This `Session` object can exist across the interactions between the browser and the Web server. Typical logic is to create a `Session` object when someone first enters a site. The `HttpRequest` object has a method that returns the session if it exists or creates it if it doesn't exist.

```
HttpSession session = req.getSession();
```

Once you have a session, you can put variables and objects in it:

```
String userId = "auser";
session.setAttribute("userId ", userId);
```

You can also retrieve values from it:

```
String userId = (String) session.getAttribute("userId ");
```

Variables can be simple or complex. You could, for example, create a class that contains all the variables that need to exist across interactions between the browser and the Web server and store an instance of the class in the session when the user first enters the Web site. For every interaction after that, you retrieve the instance associated with the session and use it for the logic. Since each user has a unique session and a unique instance of the class containing your variables, you can act on all the variables in the instance without worrying about trashing the variables associated with a different user.

If you have experience in programming CICS or IMS in an IBM environment or similar technologies on other platforms, you may notice considerable similarity between a servlet and a mainframe TP program. In CICS, each execution of the program is a transaction unto itself. No permanent connection exists between the user at the terminal and the program. CICS makes use of the COMMAREA to store data between executions, just as a servlet uses the `Session` *object.*

Java Server Pages

Java Server Pages (JSPs) came about with the recognition that large Web projects often required many different types of expertise. Most Web designers are not programmers, and most programmers are not good Web designers. In addition, placing HTML code inside of Java print statements is tedious (although some packages have been developed to simplify this), and it requires recompilation every time the appearance of the Web page needs to change. Imagine having to recompile to change background colors or add an icon.

A JSP typically looks more like HTML than Java, although it is possible to embed a significant amount of Java code into the page. Following is the code for an enhanced Welcome page you might display after a user has logged into the site and you've stored the user ID in the `Session` object.

```
<HTML><HEAD>
<TITLE>Enhanced Welcome Page</ TITLE >
</HEAD>
<BODY>
     <%
     try
     {
            String userId =  session.getAttribute("userId");
     %>
     <h1>Enhanced Welcome Page</H1>
     <%
out.println("Welcome to the Web site, " + userId);
     }
     catch(Exception e)
     {
          out.println("Failure:" + e);
     }
%>
</BODY></HTML>
```

The tags `<%` and `%>` are special tags that tell the servlet engine (the servlet engine processes JSPs, too; you'll see why in a moment) that this is Java code, not HTML. The `session` and `out` variables are automatically allocated and initialized in a JSP, although `session` could be `null` if, in fact, no session exists.

When it runs, this JSP obtains the user ID from the session and writes the phrase "Welcome to the Web site" followed by the user ID after a heading that says "Enhanced Welcome Page."

Although JSPs appear to be extensions of HTML, in reality JSPs are servlets in disguise. When the Web server invokes the servlet engine and the servlet engine detects that the file name has a .jsp extension, it knows the page is a JSP. The servlet engine finds the page that contains the text above and parses it. As the engine is parsing, it converts the page into the source code of a servlet. The static HTML elements are converted to `out.println()` method calls containing the HTML. The Java code is simply copied into the servlet source code. It wraps the generated code

with the required import statements, class definition statement, braces, and whatever else is required to create a servlet. It then compiles the servlet. If that is successful, it then executes the servlet.

Once the servlet has been successfully compiled, the parsing and compilation process will not occur again unless the JSP document has been changed. The servlet engine looks at the time stamp on the JSP and the time stamp on the compiled servlet class file to decide if it needs to recompile the JSP.

Although it is possible to design complex systems without JSPs or without servlets, many systems are designed using both servlets and JSPs. The complex business logic is coded as servlets. This logic doesn't change often, and you would not want to clutter the presentation with its complexities. Once the business logic is executed, the servlet usually stores objects in the `HttpRequest` or `HttpSession` object and, using a `RequestDispatcher`, passes control to a JSP. The JSP retrieves the object and uses its values to build the display dynamically.

Although a complete discussion of JSPs is beyond the scope of this book, here are a few of the other JSP tags:

Tag	**Description**
<%= expression %>	Evaluates the expression as a string and outputs it.
<jsp:forward>	Forwards a request to an HTML file, JSP, or servlet.
<jsp:useBean>	Instantiates or references a bean with a specific name and scope.
<jsp:getProperty>	Inserts the value of a bean property into the response.
<jsp:include>	Includes a static resource or the result from another Web component.
<jsp:setProperty>	Sets a bean property value or values.

Getting Started with Servlets and JSPs

If you choose to install the Tomcat Web server bundled with the CD-ROM that accompanies this book, you already have a complete environment for testing servlets and JSPs. Tomcat, the servlet engine from the Apache Project, is also integrated with Eclipse; you can actually debug your servlets from inside the Eclipse IDE.

Or, you can download Tomcat from the Apache Web site (http://tomcat.apache.org/) and install it. For this exercise, you can assume that Tomcat is installed in C:\apache-tomcat. If you have installed Tomcat in some other directory, then substitute your directory in place of apache-tomcat.

Once you install Tomcat and start it, you will need to place your compiled servlet class files and JSPs in the proper directories for them to be recognized by the servlet engine. For this exercise, you will copy the compiled class file to the examples directory (C:\apache-tomcat\webapps\examples\servlets).

Follow the instructions in the installation and configuration information on how to invoke your servlet or JSP. In particular, you will need to make sure that there is a JAVA_HOME environment variable that points to the location where you have the JDK installed. On most Windows systems, this means you will have to use the path Computer > Properties > Environment Variables to set a new environment variable. On Windows Vista, the path is Computer > Properties > Advanced Properties > Environment Variables. Select the option to add a new environment variable. Name the variable "JAVA_HOME," and set the value to the location of the Java SDK (C:\Progra~1\Java\jdk1.6.0_03). Click the OK buttons until this change is accepted.

Double-clicking on (running) the startup.bat file in the apache-tomcat\bin directory should now start Tomcat. If Tomcat does not start, visit the Tomcat FAQ pages at tomcat.apache.org. You can stop Tomcat by running the shutdown.bat file.

You will need to specify a port number if the servlet engine is not running on port 80. For example, invoking myjsppage.jsp if the servlet engine is running on port 8080 might require you to enter an address like this into your browser:

```
http://localhost:8080/myjsppage.jsp
```

This address references a Web server running on port 8080 on your local machine. By default, Tomcat uses port 8080, although you can change this at your convenience.

After you install Tomcat or another Web server/servlet engine, make sure the examples supplied with the servlet engine work before you proceed to the exercises.

To make sure Tomcat is installed and working properly, type this URL in your browser:

```
http://localhost:8080
```

You should see the introductory Tomcat page in your browser. Again, if you do not, check the FAQ section of the Apache Tomcat Web site.

Next, make sure the Apache servlet environment is working properly. Type this URL into your browser:

```
http://localhost:8080/examples/servlets/index.html
```

You should see a page that demonstrates several pre-packaged servlet pages.

Finally, make sure the Apache JSP environment is working properly. Type this URL into your browser:

```
http://localhost:8080/examples/jsp/index.html
```

You should see a page that demonstrates several pre-packaged JSP pages.

Exercises

In this example exercise, you are going to code a simple servlet that creates a session the first time it is invoked and displays a simple page. Every time you refresh the page, you are going to run the servlet and increment a counter that counts the number of times you have run the servlet and displays this result.

1. Create a CounterServlet.java file with the editor.
2. In order to compile a servlet, you will need to import the following packages.

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
```

3. `CounterServlet` will extend `HttpServlet` to create the shell for the class, like this:

```
public class CounterServlet extends HttpServlet
{
}
```

4. Next, you need to create code for the `doGet()` method:

```
public void doGet(HttpServletRequest req, HttpServletResponse res)
          throws IOException, ServletException
{
}
```

5. You are now ready to code the body of the `doGet()` method. This will be the code that executes every time the servlet is invoked:

```
public void doGet(HttpServletRequest req, HttpServletResponse res)
          throws IOException, ServletException
{
// You must set the content type before writing output.
     res.setContentType("text/html");
// This creates a session if one doesn't exist or retrieves it if it does
// exist.
     HttpSession session = req.getSession();
// Sessions are like HashMaps and only contain objects not primitives.
     Integer count = (Integer) session.getAttribute("count");
// Count will be null the first time you invoke the servlet because it is a
// new session.
     if (count == null)
          count = new Integer(1);
     else
          count = new Integer(count.intValue() + 1);
// This stores the object back into the session.
     session.setAttribute("count", count);
// This is writing the HTML back to the browser.
     PrintWriter out = res.getWriter();
```

```
        out.println("<html>");
        out.println("<body>");
        out.println("<head>");
        out.println("<title>" + "Counter Page" + "</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Counter Page</h1><hr>");
        out.println("Welcome to the Web site<p>");
   out.println("You have been here " + count.intValue() + " time(s)<p>");
        out.println("</body>");
        out.println("</html>");
   }
```

6. Compile the servlet. You will need to include the jar files that contain the servlet classes in your compile command. Do this by adding the `extdirs` command to the compile command.

```
javac -extdirs C:\apache-tomcat\lib CounterServlet.java
```

7. Once you have successfully compiled `CounterServlet`, you will need to copy the java class to the location used by the servlet engine to deploy the servlet. If you are using Tomcat, you will need to copy the class file to the servlet classes directory (C:\apache-tomcat\webapps\examples\WEB-INF\classes).

   You will also need to modify the Tomcat configuration file (C:\apache-tomcat\webapps\examples\WEB-INF\web.xml) and add the proper references to `CounterServlet`. These references tell Tomcat how to convert the names in the URL to a particular servlet. For your convenience, a modified web.xml file (suitable for the Tomcat version supplied on the CD) is provided in the Examples directory for this chapter. Copy that file to C:\apache-tomcat\webapps\examples\WEB-INF\web.xml. You may also need to stop, and then restart Apache so that it loads this new configuration file.

   If you are using another servlet engine, you will have to consult the documentation about how to deploy the servlet. In most cases, you can deploy a simple servlet by dropping the class file into the correct directory. If the documentation is hard to understand, just keep dropping it into different directories until you find the one that works.

   In order to invoke the servlet, you will need to start your browser and key into the address line the correct URL to connect to the servlet engine and invoke the servlet. A typical line might be something like this (although the port number may be different with your servlet engine):

```
http://localhost:8080/examples/servlets/servlet/CounterServlet
```

8. When the servlet runs, the result should look like this in the browser:

```
Counter Page
Welcome to the Web site
You have been here 1 time(s)s
```

9. Click Refresh several times and the number 1 should increment by 1 every time you refresh the page.

Reviewing the Exercises

Let's review the simple servlet you've created. This servlet created a session the first time it was invoked and stored an integer object into the session. Every time the servlet was invoked again, the integer object was retrieved, the count was incremented, and an updated integer object was placed back into the session.

You could have created an int class variable and incremented its value every time you invoked the servlet without creating a session. The result with a single user would have been indistinguishable from the result you achieved using an integer object stored in the session. This approach would have failed to give the correct result if there were two users because it would show the combined count for both users rather than the user-specific count that the `CounterServlet` tracked for you.

15 Introduction to Enterprise JavaBeans

In This Chapter

- Distributed Computing
- The Different Kinds of EJBs
- Container Services
- The Interfaces and the Implementation Class
- Accessing the Bean from the Client
- Exercises
- Reviewing the Exercises

Enterprise JavaBeans (EJBs) technology is one aspect of the JEE specification that also includes specifications for servlets, messaging, Java Server Pages, and Java Messaging and other components.

EJBs appear to be more difficult than they really should be. The apparent complexity of the infrastructure of EJBs carries with it a big tradeoff. The JEE architecture handles much of the scaling and integrity issues required for robust enterprise code. This includes database connection pooling and transaction services, so the bottom line is that the complexity is worth the tradeoff for these types of enterprise applications. Simpler applications may be well served by the use of simpler architectures, such as the use of only Web server servlets for your application.

Distributed Computing

Enterprise JavaBeans bring the benefits of Java-built components to distributed, transaction-oriented systems. An EJB can be instantiated (that is, created) on a remote system and then respond to message requests from an application on a local system. Furthermore, EJBs can contain and manage transaction definitions. EJBs automatically participate in a transactional model as defined by your application. (In a transactional model, either all the work in a transaction is guaranteed to be intact, or committed to a database, or none of it is.)

Unfortunately, EJBs and ordinary JavaBeans share the same name. This has undoubtedly been the source of endless confusion among inexperienced developers, recruiters, and nontechnical managers. The fact is that EJBs and ordinary JavaBeans have very little to do with each other. Their only real thread of commonality is that both EJBs and ordinary JavaBeans are attempts to move software development in a more componentized direction.

JavaBeans are Java components defined in a way that makes them easy to use. In most functional respects, the JavaBeans specification is similar to the ActiveX specification, except what is specific to Java. JavaBeans are simply regular Java classes that employ a certain convention for getting and setting their internal values or properties. When a JavaBean implements the JavaBean API, the class is able to interact with other Java classes in a simple, standard fashion. For example, the GUI design tools that are a part of many integrated development environments expect you to code JavaBeans.

In contrast, EJBs are business logic components. They are designed to provide back-end business logic, particularly for thin client applications. They operate through interfaces and are largely ignorant of what is happening between the human and client application. Best of all, EJBs don't have to manage the complex system plumbing that is normally associated with remote component management and transaction integrity control. All these system-level issues (e.g., component invocation, life-cycle management, security, and transaction semantics) are handled by the EJB infrastructure. A major design objective of the EJB spec is to allow developers to focus on writing code that solves the business problem(s) at hand rather than the technical issues surrounding the management of remote distributed component services.

The EJB specification is primarily an effort to standardize remote component-management techniques available to Java components. It also attempts to simplify and improve existing techniques, such as Common Object Request Broker Architecture (CORBA) and Distributed Component Object Model (DCOM). EJB is very Java-centric; it fits naturally into the Java language.

EJBs running in EJB containers (the supporting software that wraps around the EJBs) represent the best possible environment for developing the middle tier for robust, scalable applications. EJBs are written in Java. They are independent of the platform and the operating system. They are able to scale horizontally (inside a single system or across multiple systems) and vertically (using multiple systems for individual deployment tiers).

The Different Kinds of EJBs

EJBs come in three flavors: `SessionBeans`, `EntityBeans`, and `MessageBeans`. `SessionBeans` are intended for storing session-related data or may be used as a simple remote API. `EntityBeans` are representations of persistent objects usually in databases. `MessageBeans` are activated in response to messages and are intended to provide the logic for processing the message.

`SessionBeans` can be either stateless or stateful. A *stateless* `SessionBean` is simply a remote API. It has no knowledge of you before you invoke its methods and it keeps no memory of you after you invoke it. In the client, you create references to access it, you call its methods passing arguments, and you get back a result. This type of EJB might be used for a complex calculation or, perhaps, to perform a very simple database operation that does not need any transaction logic, such as accessing commonly reused data or values.

For a *stateful* `SessionBean`, the word *session* is truly appropriate. When you create a reference to a stateful `SessionBean` in a client, a parallel copy of the `SessionBean` is created in the EJB container. One client, one bean. As you interact with the `SessionBean`, you can change its internal variables and the `SessionBeans` maintains this state from one call to another. Although stateful `SessionBean` could be used to represent database objects, they might more commonly be used to represent temporary working copies of data that might or might not be eventually committed to more persistent storage. For example, a stateful `SessionBean` might be created to represent a shopping cart. The application might store and remove items in the cart. At some point, the application might discard the cart or save it to the database, possibly using an `EntityBean` to do so.

An `EntityBean` is intended to be a representation of a persistent object that is usually stored in a database. If you like, you can code the logic to retrieve and store the object (bean-managed persistence), or you can let the EJB container do the work for you (container-managed persistence).

`EntityBeans` extend the `EntityBean` parent class. Unless you are using container-managed persistence, an `EntityBean` will require additional methods for creating new entities (which you can think of as rows in a database), saving and

loading the entities, and looking them up by various keys and arguments. All the methods for creating and looking them up (various *find* methods) are defined in the home interface and have their parallel versions in the implementation class. If you are using container-managed persistence, much of this work is done for you.

A `MessageBean` is a message listener that consumes messages from a queue or a durable subscription. The messages may be sent by any JEE component—from an application client, another enterprise bean, or a Web component—or from an application or system that does not use JEE technology.

CONTAINER SERVICES

EJBs can define the following control options at the class or the method level. By default, the class settings are applied to each method, but an individual method can define its own setting.

Security: What rights are required by the client in order to create this class or perform its methods? EJBs can use the new security model built into Java 2 to define and control access rights.

Transaction: What transaction level is supported or required by this bean? Here are the possibilities:

- `TX_BEAN_MANAGED`
- `TX_SUPPORTS`
- `TX_NOT_SUPPORTED`
- `TX_REQUIRED`
- `TX_REQUIRES_NEW`
- `TX_MANDATORY`

These settings indicate the valid transaction contexts for the bean. (It may be necessary for the client to first establish a new transaction context in order to meet the transaction requirements of the bean.)

Isolation level: What JDBC isolation level does the client require when a read is performed? The possibilities are TRANSACTION_READ_UNCOMMITTED, TRANSACTION_READ_COMMITTED, TRANSACTION_REPEATABLE_ READ, and TRANSACTION_SERIALIZABLE.

The container manages the transaction context and client/server connection on behalf of the client. The client does not need to be directly involved in details of the transaction semantics, such as begin transaction, commit, or rollback. Instead, the container creates a simplified client view of the EJB for the client, and the client can access the EJB as if it were a simple remote object. If the client needs to, however, it can adjust the default transaction behavior of the EJB.

What's more, the EJB specification requires that the EJB server container provides extensive supporting software. This includes database connection pooling and transaction services. This code shouldn't have to be rewritten for every application. Once it is right, it will be right for everybody. You don't have to reinvent the wheel.

The Interfaces and the Implementation Class

Actually developing an EJB (of whatever type) requires that the developer code two interfaces and the implementation class. You might expect some relationship to exist between the interfaces and the class. There is a relationship, but it is not the usual relationship between classes and interfaces. The implementation class does not implement (in the Java sense) either of the two interfaces you have to code. You will grow accustomed eventually to the somewhat nonstandard relationship between the interfaces and the class. Initially the relationship is confusing. Keep in mind that the interfaces are for the client—the application that will be using the EJB—and things will be clearer.

The first interface you need to code is the remote interface. Believe it or not, this is what your implementation class will implement in reality, although it does not need to implement the interface in the Java sense of the word. The remote interface is intended for the client. It is a description of the business methods that are accessible to the client once the EJB is created.

```
public interface MyEJBInterface extends EJBObject
{
        public String getMyEJBValue() throws RemoteException;
 }
```

The second interface you code is the home interface. The home interface is what the client will use to create the reference to the EJB. In fact, the home interface returns the remote interface but triggers in the EJB container the entire supporting infrastructure required to create or instantiate the implementation class.

The home interface may contain multiple creation methods based on key lookup or creation. It can also contain methods to create multiple instances of beans based on a selection criterion that reflects range.

```
public interface MyEJBHome extends EJBHome
{
        MyEJBInterface create() throws RemoteException, CreateException;
}
```

The implementation class itself is really the meat of the business logic you are providing—it is the real EJB. This class will either extend `SessionBean`, `EntityBean`, or `MessageBean`, depending on the type of EJB you are creating.

The implementation class will contain the methods that you defined in the remote interface class. This class must also contain the methods required to support the entire interaction with the EJB container.

```
public class MyEJB implements SessionBean
{
        public String getMyEJBValue()  { return "some value"; }
        public void ejbCreate() {}
        public void ejbRemove() {}
        public void ejbActivate() {}
        public void ejbPassivate() {}
        public void setSessionContext(SessionContext sc) {}
}
```

For `SessionBeans`, these methods include the various life-cycle methods: `create`, `remove`, `activate`, `passivate`, and `setContext`. In stateless `SessionBeans`, you can implement trivial versions of these methods because essentially you are simply creating a remote API. That is what is shown in this example. Once you go beyond stateless `SessionBeans`, you will need to do more. For one thing, you will probably need multiple creation methods. For each `create()` method you defined in the home interface (such as `create()`, `create(String key)`, and `create(int numberKey)`), you will need a corresponding `ejbCreate` (such as `ejbCreate()`, `ejbCreate(String key)`, and `ejbCreate(int numberKey)`) in the implementation class. The `passivate` and `activate` methods are associated with saving and restoring the state of the bean if the container needs to swap it to disk.

Once you've coded the bean and its interfaces, you are ready to deploy. Deployment is specific to the application server that is in use. Basically, you will need to tell the server where the code is to implement the bean and define its characteristics: session stateless, session stateful, entity bean-managed, or entity container-managed. Depending on the type of bean, you will need to define additional things.

Deployment will generate the code that the client needs to access the EJB. During this process, you will give a name to the EJB. This name will be the name used to actually locate the EJB.

Accessing the Bean from the Client

The Java Naming and Directory Interface (JNDI) provides the facilities for finding the remote server on which the EJB runs and for obtaining a reference to it. The `InitialContext` is the starting point for this lookup process. The `lookup()` method itself obtains the reference which is, in fact, a reference to the home interface. The home interface is used for actually creating the EJB and obtaining an interface to access its methods.

```
        Context initial = new InitialContext();
        java.lang.Object objref = initial.lookup("MyEJBName");
        MyEJBHome home = (MyEJBHome)
        PortableRemoteObject.narrow(objref,MyEJBHome .class);
        MyEJBInterface myEJB = home.create();
// Now you can use the EJB.
        String value = myEJB.getMyEJBValue();
```

Exercises

To do any exercises with EJBs, you will need a JEE-compliant application server. You can download the JEE SDK from the Java Sun Web site. It contains everything you will need to develop and test EJBs but is not a robust implementation that could be used in a production environment. The Java Sun Web site at http://java.sun.com/JEE/tutorial/index.html contains an excellent beginner's tutorial.

Following are high-level steps to code, deploy, and use a simple stateless `SessionBean`. The Java Sun tutorial provides details on how to do each of these steps.

1. Download the JEE SDK from Java Sun and install the software.
2. Code and compile the `ConverterBean` example found in the Getting Started Section of the tutorial.
3. Code the ConverterClient application that accesses the `ConverterBean`.
4. Start the JEE Server.
5. Deploy the `ConverterBean`.

6. Execute the ConvertClient application. If you have all the pieces in place, the application should access the `ConverterBean` class instance running in the application server, invoke the currency conversion methods in the sample, and display the results of the conversion.

REVIEWING THE EXERCISES

Although this simple example is relatively trivial, in the real world, EJBs are used for complex business logic and especially for database interactions. The objective of this simple example was to expose you to the complex infrastructure required to develop in the JEE environment. Try to understand how the various interfaces and pieces of code interact in order to implement EJBs.

16 Introduction to XML

In This Chapter

- The Basics
- XML vs. HTML
- Document Type Definitions (DTDs)
- DTD Components
- XML Declaration
- A Complete XML Document
- XML Schemas
- Authoring XML Documents
- XML and Java
- XML and HTML
- Where to Use XML
- Electronic Data Interchange (EDI)
- Online XML or Web Services
- XML and OAG
- Other Opportunities
- Exercises
- Reviewing the Exercises

XML is another example of a computer language technology that has taken the industry by storm. The only language to match Java's buzz, and to acquire more instantaneous and ardent supporters, is XML.

Just what is this language, and what does it mean to the business application developer? After reading the latest article extolling the virtues of XML, a developer can easily be left with the impression that XML can be the answer to all of your worries, from uncommunicative business partners to the common cold. But what is the reality? What are the sorts of problems that XML can help the business application developer solve?

THE BASICS

XML is an acronym for Extensible Markup Language. It is simply a language specification for documents that describe and contain data. XML's designers have attempted to combine the simplicity and the ubiquity of HTML with the rich descriptive capabilities of Standard Generalized Markup Language (SGML). HTML and XML are, in fact, both SGML document types.

An XML document is a text file that conforms to the XML language specification. It contains data in a structured format and descriptive information about the data. The primary role of an XML document is to present data generated by one application (or system) to another. Consequently, XML documents are well suited as general-purpose data repositories and data transport containers, as well as common structures such as configuration files.

XML VS. HTML

XML provides more control than HTML primarily by allowing a document to describe its own tags (similar to a data type or, significantly, a record type). This capability allows a document to organize its data in a structured format. An XML document can also contain enough metadata (information about the data) so that any application can reliably parse the document and extract the data from the document.

In contrast, HTML is designed to describe documents in a format suitable for end-user viewing in a graphical browser. HTML documents do not contain information about the meaning of the data, nor are they structured in a way that makes it easy for a program to analyze. Therefore, an application may have a difficult time extracting relevant data from an HTML document.

A relatively simple example makes this point. Here is a portion of an HTML page that might be generated by an Internet book retailer. It informs an Internet browser how to represent the current contents of the shopping cart page to a potential purchaser.

```
<td bgcolor="#FFFFFF" width="51%">
<a href="../81332713233407">
      <em>Debt of Honor</em></a>
<br>
Tom Clancy;
Paperback</b>
<font size=2 face="Verdana, Helvetica, Courier" color=#000000>
<NOBR>Price: <font color=#990>$6.99</font></b></NOBR><br>
```

```
</td>
<td bgcolor="#FFFFFF" width="51%">
<a href="../81332713233407">
      <em>The Hunt for Red October</em></a>
<br>
Tom Clancy;
Hardcover</b>
<font size=2 face="Verdana, Helvetica, Courier" color=#000000>
<NOBR>Price: <font color=#990>$18.99</font></b></NOBR><br>
</td>
```

An Internet browser has no trouble understanding how to format and display this information in a page to the end user. While viewing the page in a browser, the end user has no trouble understanding what the data means (the shopping cart contains two books, one for $6.99 and the other for $18.99).

However, what if you want a program to parse this document and to extract the item number and other information, including its price, from the HTML document? In theory, you could use a trial-and-error design approach to build a parser for this particular document. Perhaps you could fine-tune this algorithm so that it can process the shopping cart HTML page and extract the price of the book:

- Look for the string `NOBR>Price:`.
- Skip past the font declaration (`<font ..>`).
- The characters before the next font declaration contain the price.
- Ignore the currency symbol in the price character string.
- Convert the price string into a numeric price variable.

However, you would have no guarantee that the parser would work if the vendor made even minor changes to the Web site, or that the parser wouldn't be confused by similar pages. More important, you would have no guarantee that the parser would work with another vendor's HTML pages. Furthermore, important contextual information is hard to decipher. For example, what is the identifier (the order ID) for this shopping cart? Is it contained in the `href` identifier?

An XML document, on the other hand, contains information in a format that can be readily parsed by an application. An XML document might express a shopping cart using this type of syntax:

```
<Order orderNumber="81332713233407">
   <LineItem>
     <Title>Debt of Honor</Title>
     <Author>Tom Clancy</Author>
     <BookType>Paperback</BookType>
```

```
      <Price>$6.99</Price>
    </LineItem>
    <LineItem>
      <Title>The Hunt for Red October</Title>
      <Author>Tom Clancy</Author>
      <BookType>Hardcover</BookType>
      <Price>$18.99</Price>
    </LineItem>
</Order>
```

Clearly, this syntax is simpler to parse with a program and will produce more predictable results. An application can process and validate information from this document with confidence.

Notice that XML uses the `begin tag...end tag` construct in a manner similar to HTML. XML data is contained inside user-defined elements. For example, `<Title>` is the beginning of an element, and `</Title>` is the end of the element. Every XML document must conform to these and other requirements in order to be classified as well formed, or syntactically correct.

Elements can be nested as child elements inside of other elements. Observe how the `LineItem` element in the example contains each of these elements: `Title`, `Author`, `BookType`, and `Price`.

Document Type Definitions (DTDs)

An XML document not only contains information in a predictable format, it can also describe the organization of the information it contains. The beginning of an XML document may contain a document type definition (DTD), or a reference to an external DTD. This section of the XML document defines the structure of the document's contents. It identifies the elements that are allowed in this document and the relationships between the various elements in the document. An XML parser can use this information to make sure that the document is not only well-formed (that is, it conforms to the generic XML syntax rules mentioned previously), but that it is also valid (that is, conforms to the ordering, grouping, and cardinality rules specified in a DTD).

For example, the XML shopping cart document could contain this partial DTD:

```
<!ELEMENT Order (LineItem)+>
<!ATTLIST Order orderNumber CDATA #REQUIRED>
<!ELEMENT LineItem (Title, Author+, BookType, Price)>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Author (#PCDATA)>
```

```
<!ELEMENT BookType (#PCDATA)>
<!ELEMENT Price (#PCDATA)>
<!ENTITY HARDCOVER "Hardcover">
<!ENTITY PAPERBACK "Paperback">
```

The first statement in this DTD describes an XML element whose name is `Order`. According to this DTD, an `Order` consists of one or more `LineItem` elements (represented by the `(LineItem)+` expression in the example). The `Order` must be further described using the attribute named `orderNumber`. An `orderNumber` attribute can contain any valid character sequence (as specified by the CDATA type). It is a required attribute, as specified by the #REQUIRED flag.

A `LineItem`, in turn, consists of a `Title`, one or more `Authors` (as indicated by the command `Author+`), a `BookType`, and a `Price`, all in the specified sequence given. These subelements are all user-defined data types, which can contain any character sequence (#PCDATA).

The last section of the DTD declares two internal entities (similar to macros) named HARDCOVER and PAPERBACK. These internal entities are assigned the String values "`Hardcover`" and "`Paperback`", respectively.

A DTD not only provides the structural information required to properly parse the document, it allows the parser to examine the document for completeness and document integrity. It is similar in nature (although limited in features) to the Structured Query Languages' (SQL's) Data Definition Language (DDL) statements. If a DTD specifies that certain elements are required, then these elements must be present in the document. Conversely, only properly identified elements are allowed in an XML document.

An input parser application can quickly scan XML documents to validate that they match their required structure, independent of any data or content validation. An XML document can be independently checked to make sure it is both well formed (it conforms to proper generic XML syntax) and valid (it conforms to its DTD).

An XML document can either contain its DTD (that is, as the initial part of the document) or simply refer to it as an external file. Either a URL or a local file name can be referenced as the DTD repository. External file references work much the same way that COPY statements work in COBOL. External file references are preferred, since their use conserves bandwidth and provides a standard location for the most recent specification.

DTD Components

A DTD describes these optional XML document components in any combination: elements, attributes, and entities.

Elements

An element is the user-defined atomic data unit. An XML document is a collection of elements. An element can contain text, other elements, or even a combination of the two. The syntax for an element in an XML document is `<Name> content </Name>`. An element is roughly analogous to a data item in a COBOL program.

These are four of the elements from the XML sample:

```
<Title>Debt of Honor</Title>
<Author>Tom Clancy</Author>
<BookType>&PAPERBACK</BookType>
<Price>$6.99</Price>
```

The sample also contained a `LineItem` element, which was comprised solely of other elements:

```
<LineItem>
     <Title>Debt of Honor</Title>
     <Author>Tom Clancy</Author>
     <BookType>&PAPERBACK</BookType>
     <Price>$6.99</Price>
</LineItem>
```

The `LineItem` element consists of other elements. A group item in a COBOL program is similar, since it consists of other data items.

A document's DTD declares all the elements a document can contain, the cardinality rules, whether or not elements have attributes, and the order and arrangement of those elements. A DTD is roughly equivalent to the WORKING-STORAGE area in a COBOL program, except that a DTD can contain much more descriptive information about its elements.

In the sample, an element named `Title` is defined in the DTD with the following construct:

```
<!ELEMENT Title (#PCDATA)>
```

The `Title` element can now exist in the body of the sample document as follows:

```
<Title>Debt of Honor</Title>
```

The symbols used to specify cardinality rules for particular elements are shown in the following table:

Symbol	Rule
No symbol	Exactly one element is allowed.
+	One or more elements are allowed.
?	Zero or one elements are allowed.
*	Zero or one or more elements are allowed.

ATTRIBUTES

An element can also be described in more detail with information that varies with each instance of the element in the document. For example, suppose you would like the `Order` element to contain an attribute named `orderNumber`. In any particular `Order` element, this attribute would hold the unique ID that identified this `Order`.

Attributes are defined and used in much the same manner as elements. The DTD declares which attributes exist for an element. Particular elements in the document body can then be qualified with these attributes.

In the sample, an element named `Order` is declared in the DTD. It consists of one or more `LineItems`. An attribute for `Order` named `orderNumber` is defined as well. It can contain any valid string and is required for each instance of `Order`, as specified in this partial DTD:

```
<!ELEMENT Order (LineItem)+>
<!ATTLIST Order orderNumber CDATA #REQUIRED>
```

An `Order` element can now exist in the body of the sample document as follows:

```
<Order orderNumber="81332713233407">
   <LineItem>
      ...
   </LineItem>
</Order>
```

Notice that attributes are entered inside the initial element tag. Attribute values must be inside quotes, and the first quote must be preceded by an equals sign, after the attribute name.

```
 (<Order ..>).
```

Common data types for attributes are shown in the following table:

Type	Contents
CDATA	Character data.
Enumerated	A list of possible values, one of which is chosen, e.g., (Hardcover | Paperback).
ID	A named attribute. Must be unique in the DTD.
IDREF	A reference to a named ID.

Entities

An XML document can define its own constants or entities. These are named storage units (portions of valid XML content), defined and used by a document. An entity can contain character strings, markup commands, or even references to external documents. Here are two entities as declared in the sample DTD:

```
<!ENTITY HARDCOVER "Hardcover">
<!ENTITY PAPERBACK "Paperback">
```

After they are declared, entities can be used in any appropriate place in the XML document. In the sample document, you can use the entity (by name, with an ampersand as a prefix and a semicolon as a suffix) in place of the text represented by the entity. Consequently, both of these constructs are valid:

```
<BookType>&PAPERBACK;</BookType>
<BookType>"Paperback"</BookType>
```

In many ways, entities are similar to Level 88 items in a COBOL program. Entities are often used for XML content that is frequently reproduced in the document (an internal entity), or that is defined externally, and may vary with each instance of the document type (an external entity). External entities can be defined as public, that is, widely used (PUBLIC), or they can be more private, intended to be used by a small set of XML authors (SYSTEM).

```
<!ENTITY name SYSTEM "URI">
<!ENTITY name PUBLIC "public_ID" "URI">
```

DTD's also provide *parameter* entities, which can also be internal or external. Parameter entities define some notation sequence in a single place, and then let you use that definition repeatedly, instead of having to retype the notation sequence each time.

This example defines a parameter entity named `requiredAttribute`, and then uses it as part of an attribute definition.

```
<!ENTITY % requiredAttribute "CDATA #REQUIRED">
<!ATTLIST Order orderNumber %requiredAttribute>
```

It is the same as the following:

```
<!ATTLIST Order orderNumber CDATA #REQUIRED>
```

Parameter entities can only be used in the DTD itself, and not in the XML document.

XML Declaration

An XML document begins by identifying itself as an XML document. The following represents a typical introduction to an XML document:

```
<?xml version="1.0" standalone="yes" encoding="UTF-8"?>
<!DOCTYPE Order [
     ... <! -- DTD specifications -->
]
```

The first expression declares the XML version to which the document conforms. The document is a version 1.0 document.

The document is also a standalone XML document, without references to other documents. Since it is a standalone document, it must contain its DTD, if one exists. It is possible for a simple standalone XML document to have no DTD.

Finally, the first line states that the document can contain only 8-bit UTF-8 bytes. In UTF-8, a "character" can be represented in a single byte for the traditional ASCII characters, and non-ASCII characters can be represented in a sequence of multiple bytes.

An XML document can contain 16-bit characters, similar to Unicode in Java. If this document were a 16-bit document, the required syntax would be `encoding="UTF-16"`. However, almost every tool and every platform supports the UTF-8 coding scheme; support for any other scheme is quite limited. So be sure to do careful research before using any coding scheme other than UTF-8.

The next line in the type declaration identifies the document type as `Order`. An `Order` comprises all the elements and attributes specified. These directives will be included inside the braces ([]).

The exclamation point followed by two dashes is a tag that represents a comment. The text between <!-- and the closing --> will be ignored by an XML parsing program.

A Complete XML Document

The following sample XML document brings all these concepts together. It contains a document type declaration, an internal DTD specification, and then the actual data contents of the document:

```
<?xml version="1.0" standalone="yes" encoding="UTF-8"?>
<!---      A Book order in XML format    -->
<!DOCTYPE Order [
<!ELEMENT Order (LineItem)+>
<!ATTLIST Order orderNumber CDATA #REQUIRED>
<!ELEMENT LineItem (Title, Author+, BookType, Price)>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Author (#PCDATA)>
<!ELEMENT BookType (#PCDATA)>
<!ELEMENT Price (#PCDATA)>
<!ENTITY HARDCOVER "Hardcover">
<!ENTITY PAPERBACK "Paperback">
]>
<Order orderNumber="81332713233407">
   <LineItem>
     <Title>Debt of Honor</Title>
     <Author>Tom Clancy</Author>
     <BookType>&PAPERBACK;</BookType>
     <Price>$6.99</Price>
   </LineItem>
   <LineItem>
     <Title>The Hunt for Red October</Title>
     <Author>Tom Clancy</Author>
     <BookType>&HARDCOVER;</BookType>
     <Price>$18.99</Price>
   </LineItem>
</Order>
```

XML Schemas

DTD's are the original XML specification language. However, there are a number of limitations with this standard. As a result, newer, more expressive standards have emerged. The most common is XML Schema, or XML Schema Definition (XSD). This standard offers several improvements over DTD.

- XSDs are themselves in XML format, and so they can be parsed by XML parsers.
- XSDs allow for *namespaces*, or the specification for where a particular (tag) name is valid. This also allows a single name to be valid in more than one place.
- XSDs allow for more validation rules than DTDs.

Authoring XML Documents

In practice, you will not normally need to create XML documents by hand using only a text editor. One possibility is that you will use an XML authoring tool. An even more likely scenario is that you will write an application that generates valid XML documents based on data from an existing system.

There are several good XML editing tools from a variety of sources, and some are freely available. The XML site, www.xmls.com, contains an excellent list of standalone XML editors, integrated DTD editors, and other very useful information concerning XML software.

XML and Java

Any programming language can create or process an XML document. Java is no exception. XML is not a specification of a programming language, nor is it a programming language interface. Rather, it describes how an application can represent data for the benefit of another application. As such, XML is very programming-language neutral. For example, there are no predefined XML data types other than quoted character strings.

Starting with Java 1.4, Sun provides implementations of the Java API for XML Processing (JAXP) interface. This interface provides a means for Java applications to read, write, parse, and transform XML documents. Individual implementations of the JAXP interface are available, in addition to the version provided with the Java SDK.

JAXP provides three techniques to process XML documents: DOM, SAX, and StAX.

- **Document Object Model (DOM):** This is the most convenient interface. When you use this interface, the entire XML document is loaded into memory, and the elements in the document are loaded into a hierarchy of individual objects. Your program may then navigate through the hierarchy to read or write the object that represents the XML tag you are interested in.

  This interface is the most straightforward and the most intuitive. It is a quite powerful mechanism to navigate complex XML documents. However, this convenience comes at a cost. Since the entire XML document resides in memory, it is very slow and very memory intensive.
- **Simple API for XML parsing (SAX):** This interface is much faster and uses less memory than the DOM interface. XML tags are read or written in sections as a stream of bytes. As an XML tag is identified in the stream, the SAX parser invokes *callbacks*, or predefined methods in your program. In these methods, you can collect and store the XML content that you are interested in.

  This method, while faster and more scalable, does have its own costs in the form of increased programming complexity. You more or less have to write low-level custom code for each XML tag that you need to process. And in many cases, you still need to process the entire document.
- **Streaming API for XML (StAX):** This latest interface combines some of the simplicity of the DOM with the speed of the SAX interface. When using this interface, your program gets an iterator and requests sections of the XML document. You can then decide to process that information, skip to the next section, or just decide to stop processing. Unlike SAX, where the parser controls your program, with StAX, your program is in control of the parser.

In many situations, you will probably find that the DOM parser is most appropriate for modest-sized XML documents whose elements must be read or updated randomly many times. A configuration file is a good example of this type of access. For larger XML files that are processed most often in sequential order, the StAX interface is superior. A batch interface from an external system is a good example this type of XML document.

XML AND HTML

Unlike HTML, XML does not define presentation attributes such as `</font>`. An XML document designer is free to define and use such tags, but they do not currently have a generally understood meaning.

The XHTML standard combines the structure of XML with the graphical representation capabilities of HTML. XHTML is a language definition very similar to HTML, but it has more rigorous rules. For example, XHTML tags and attributes in

XHTML documents must be all lowercase. And XHTML requires that all tags must be closed. In HTML, this isn't always necessary. There are other differences, most of which are meant to bring the HTML standard into compliance with stricter language and parsing rules of XML.

Where to Use XML

XML is important to the business application developer because it finally offers a standard language and syntax suitable for intersystem data transfer. Every business system needs to interact with at least one other system in some fashion. The other system can be an external system at a business partner. Or the other system might be an internal system (perhaps a data warehouse) that needs information from another internal system (for example, the production order entry system).

Before the widespread use of XML, the standard technique used to transfer information between two business systems has been to write a set of interface programs. System one would create an interface file with the required information. Or system one might write the information to interface tables in a database. The target system will read that information, validate it, and update the target system database. In rare cases, the source or target system will access the other system's database directly, but this approach requires simultaneous access to both systems and developers who understand both systems.

Interface systems often require serious planning and attention. Carefully predefined and documented interface format specifications are necessary so that each system's developers can understand what is required. The normal process flows in both systems need to be accounted for. Most important, the data transformation requirements are embedded in the interface system logic.

As business requirements and the systems that support these requirements change, interface system requirements may change as well. Implementing these changes will require careful coordination and integrated testing plans, even if only two systems are involved. When multiple target systems or their interfaces need to be adjusted for a single-source system, modifications to the existing integration process can become unmanageable. If multiple business partners' systems are involved, one can only hope that their interface specifications are adaptable and up to date and that the original developers are still around.

XML promises to improve this process significantly. As general purpose, self-describing data repositories, XML files are readily accessible by multiple systems. Intricate coordination and data mapping designs are not as crucial, since the XML files describe themselves. Systems need only to create XML documents based on an agreeable DTD or to input data from an XML file based on their own requirements. Modifications to the source or the target system do not need to be so closely

coordinated. Often, new data tags (information) in an XML document can be ignored by a processing system if that system has no need for the data. The new data will not, by itself, obscure the data required by the processing system.

ELECTRONIC DATA INTERCHANGE (EDI)

EDI-based processes are prime candidates for improvements with XML. EDI is a standard definition (defined by ANSI or EDIFACT) of acceptable document structures for various types of business transactions. Document structures are defined for purchase orders, acknowledgments, invoices, and other transaction types.

Systems use EDI when they either generate a document in EDI format or receive and subsequently process an EDI document. EDI documents are intended to be transmitted from one business partner to another. In most cases, some data translation and reformatting is necessary so that EDI documents can be properly used by target systems.

Organizations called value-added networks (VANs) provide EDI-related services, such as secure and reliable document transmission services, business partner connections (that is, electronic access to your business partners), and document management (such as store and forward, translation, and logging). Examples of VANs are GE Information Systems (GEISCO), Sterling Commerce, and Harbinger.

To date, EDI standards have proved useful as well-defined data representation formats. However, their rigid structure (along with other issues, such as high implementation and transaction costs) has restricted the acceptance of EDI standards in many situations. Typically, EDI is employed as a data transfer mechanism between business partners only if the number of documents exchanged justifies the effort and the expense and both organizations have technically savvy IT organizations.

The XML language, combined with widely available Internet technologies, promises to radically change the way business documents are transferred between organizations. Organizations are all interconnected today over the Internet. SSL (secure socket layer), encrypted digital signatures, and other technologies provide adequate transactional security for most situations. These technologies, allied with XML's ability to combine flexible document content with well-understood document structure, enable organizations to confidently, effectively, and efficiently execute electronic business processes with a wide variety of business partners.

XML and EDI do not necessarily directly compete as potential solutions. An XML file can contain data that conforms to EDI content specifications, even if it does not conform to EDI's document structure specifications. All that is required is a suitable DTD. In fact, several initiatives are underway to do just that. EDI-based interface

systems can easily wrap their documents in XML for external representation to business partners. The business partners can either process the documents directly or first convert the documents into a format suitable for a legacy EDI system.

Online XML or Web Services

The use of XML is not restricted to batch operations. Real-time interfaces to systems necessarily define data structures. Often, these structures must be self-describing for a variety of reasons.

For example, suppose you have a real-time interface that performs some service on behalf of a client application. This service must support several client applications simultaneously. Suppose further that the interface specification needs to include additional data elements in order to support some new requirement. Not all client systems can be updated simultaneously; in fact, it is likely that at least some older client systems need to be supported for a year or more.

One traditional solution for this requirement is to include a version identifier in the interface structure. The real-time service module can examine the version of a particular message and respond to either the new or the old type of message. (In this situation, you are well advised to normalize either message type into some standard internal structure rather than actually leaving the original code as is. Otherwise, there will be two versions of the service module to maintain.)

An XML-based real-time interface specification is a perfect choice for this situation. Each client service request that is based on XML will be self-describing. The service module can extract the data that is actually in a particular message, perhaps providing defaults for data that is not present in the message. The service module can respond with an XML-based return message containing either the new or the old data set. A client application can extract the information it requires from the response. An even more sophisticated solution would allow the client application to pass in an initial XSD or DTD at runtime. This XSD would define the structure of the request and response method that the client can accept.

These advantages work for the client application as well. Client applications that use XML-based messages when talking to services can talk to new and old services simultaneously. The structure of the messages is not tied to the various versions of the services, only the message content is. This greatly simplifies the coding required to support multiple versions of service modules talking to multiple versions of clients.

XML is also an excellent infrastructure for interactive messages between systems. For example, an online procurement system might send an XML-based catalog query to a supplier's system. The supplier's system could reply with catalog

items described in XML, including prices. The procurement system could simply display the results in a browser or store the results in a database for later access.

The Web services standard encapsulates these advantages. Web services are essentially XML documents embedded in HTTP requests and responses. Web services combine the expressiveness and flexibility of XML with the omnipresence of the HTTP transport mechanism to create a powerful, adaptable, and standard intersystem communication technology.

XML AND OAG

The Open Application Group (OAG) is an organization whose mission is to simplify integration processes between business systems. OAG frequently addresses integration issues as they apply to enterprise resource planning (ERP), accounting, and human resource packaged application systems.

OAG has defined a set of XML DTDs and is promoting these document definitions as standards for business documents, such as purchase orders and invoices. They have also defined intersystem document types, such as journal transactions to a general ledger system. Business applications that support these document definitions can reliably integrate with other systems, even when the two systems come from distinct vendors.

For example, an accounts payable system from one vendor can generate an XML document that conforms to OAG's DTD specification for journals. An OAG-compliant general ledger from another system can accept that document as input and post the transactions in an appropriate manner.

The following is a sample journal DTD from OAG. It has been simplified slightly from its complete representation. The complete OAG DTD specification is included on the CD-ROM.

ON THE DVD

The sample begins with a pair of base DTDs named domains.dtd and fields.dtd. These define the basic data type entity (STRDOM) and the field-level entity names that will be used in the OAG documents.

```
<!-- String Data: Generic Data Domains -->
<!ENTITY % STRDOM "(#PCDATA)">
<!ELEMENT ACCTPERIOD   %STRDOM;>
<!ELEMENT ACCTTYPE     %STRDOM;>
<!ELEMENT ACCTYEAR     %STRDOM;>
<!ELEMENT BUSNAREA     %STRDOM;>
<!ELEMENT COSTCENTER   %STRDOM;>
<!ELEMENT CURRENCY     %STRDOM;>
<!ELEMENT DRCR         %STRDOM;>
```

```
<!ELEMENT DEPARTMENT   %STRDOM;>
<!ELEMENT GLENTITYS    %STRDOM;>
<!ELEMENT GLNOMACCT    %STRDOM;>
<!ELEMENT NUMOFDEC     %STRDOM;>
<!ELEMENT ORIGREF      %STRDOM;>
<!ELEMENT SIGN         %STRDOM;>
<!ELEMENT USERID       %STRDOM;>
<!ELEMENT USERAREA     %STRDOM;>
<!ELEMENT VALUE        %STRDOM;>
<!ELEMENT VALUECLASS   %STRDOM;>
<!ELEMENT VERB         %STRDOM;>
```

Next, the sample contains a higher-level reusable DTD named segments.dtd. Of particular interest in the segment DTD are these elements:

AMOUNT: A general-purpose element that will contain an amount data item. An AMOUNT is a collection of the VALUE, NUMOFDEC, SIGN, CURRENCY, and DRCR elements. It has a required attribute named `qualifier`. This attribute can contain either of the types defined by the SEG_AMOUNT_QUALIFIERS entity or the type defined by a generic entity named SEG_AMOUNT_ QUALIFIERS_EXTENSION. AMOUNT also has an attribute named type. This attribute can contain either of the values in the SEG_AMOUNT_TYPES entity or the value defined by a generic entity named SEG_AMOUNT_TYPES_ EXTENSION.

DATETIME: A general-purpose element that will contain a date-time data item. A DATETIME is a collection of the YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, SUBSECOND, and TIMEZONE elements. It has a required attribute named `qualifier`. This attribute can contain either of the types defined by the SEG_DATETIME_QUALIFIERS entity or the type defined by a generic entity named SEG_DATETIME_QUALIFIERS_EXTENSION. DATETIME also has an attribute named type. This attribute can contain either of the values in the SEG_DATETIME_TYPES entity or the value defined by a generic entity named SEG_DATETIME_TYPES_EXTENSION.

```
<!-- From oagis_segments.dtd -->
<!-- AMOUNT -->
<!ENTITY % SEG_AMOUNT_QUALIFIERS_EXTENSION "OTHER">
<!ENTITY % SEG_AMOUNT_QUALIFIERS
    "(ACTUAL | APPRVORD | AVAILABLE | BUDGET | COMMISSION |
      DISCNT | DOCUMENT | EXTENDED | ITEM | OPENITEM | ORDER |
```

```
        ORDLIMIT | TAX | TAXBASE | TOTLIMIT |
        %SEG_AMOUNT_QUALIFIERS_EXTENSION;)">
<!ENTITY % SEG_AMOUNT_TYPES_EXTENSION "OTHER">
<!ENTITY % SEG_AMOUNT_TYPES
    "(T | F | %SEG_AMOUNT_TYPES_EXTENSION;)">
<!ELEMENT AMOUNT (VALUE, NUMOFDEC, SIGN, CURRENCY, DRCR)>
<!ATTLIST AMOUNT
    qualifier %SEG_AMOUNT_QUALIFIERS; #REQUIRED
    type %SEG_AMOUNT_TYPES; #REQUIRED
     index CDATA #IMPLIED>
<!ENTITY % AMOUNT.ACTUAL.F    "AMOUNT">
<!ENTITY % AMOUNT.ACTUAL.T    "AMOUNT">
<!ENTITY % AMOUNT.DOCUMENT.T  "AMOUNT">
<!-- DATETIME -->
<!ENTITY % SEG_DATETIME_QUALIFIERS_EXTENSION "OTHER">
<!ENTITY % SEG_DATETIME_QUALIFIERS
    "(ACCOUNTING | AVAILABLE | CREATION | DELIVACT | DELIVSCHED |
      DISCNT | DOCUMENT | DUE | EFFECTIVE | EXECFINISH | EXECSTART |
       EXPIRATION | FORECASTF | FORECASTS | FROM | INVOICE | LABORFINSH
|
       LABORSTART | LASTUSED | LOADING | MATCHING | NEEDDELV | OPFINISH
|
       OPSTART | PAYEND | PROMDELV | PROMSHIP | PYMTTERM | REPORTNGFN |
       REPORTNGST | REQUIRED | RESORCDWNF | RESORCDWNS | SETUPFINSH |
       SETUPSTART | SHIP | TEARDOWNF | TEARDOWNS | TO |
       %SEG_DATETIME_QUALIFIERS_EXTENSION;)">
<!ENTITY % SEG_DATETIME_TYPES_EXTENSION "OTHER">
<!ENTITY % SEG_DATETIME_TYPES
    "(T | F | %SEG_DATETIME_TYPES_EXTENSION;)">
<!ELEMENT DATETIME (YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, SUBSECOND,
    TIMEZONE)>
<!ATTLIST DATETIME
    qualifier %SEG_DATETIME_QUALIFIERS; #REQUIRED
    type %SEG_DATETIME_TYPES; #IMPLIED
     index CDATA #IMPLIED>
<!ENTITY % DATETIME.ACCOUNTING  "DATETIME">
<!ENTITY % DATETIME.DOCUMENT    "DATETIME">
<!ENTITY % DATETIME.PAYEND      "DATETIME">
<!-- BSR -->
<!ELEMENT BSR (VERB, NOUN, REVISION)>
<!-- SENDER -->
<!ELEMENT SENDER (LOGICALID, COMPONENT, TASK, REFERENCEID,
CONFIRMATION,
```

```
    LANGUAGE, CODEPAGE, AUTHID)>
<!-- CNTROLAREA -->
<!ELEMENT CNTROLAREA (BSR, SENDER, DATETIME)>
<!-- From oagis_segments.dtd -->
```

After the base DTDs, you have a DTD for a specific document type. The post journal DTD begins by describing the general structure of this document in a comment. According to the comment, a `post_journal` document consists of JOURNALs, which in turn, consists of one or more sets of JEHEADER and JELINEs.

```
<!-- From 001_post_journal_004.dtd -->
<!--
     Structure Overview
     POST_JOURNAL (JEHEADER, JELINE+)
          JEHEADER ()
          JELINE ()
     Notes
-->
<!-- ==================================================== -->
```

The following syntax effectively copies the statements in the file named oagis_domains.dtd:

```
<!ENTITY % DOMAINS SYSTEM "oagis_domains.dtd">
%DOMAINS;
<!ENTITY % FIELDS SYSTEM "oagis_fields.dtd">
%FIELDS;
<!ENTITY % SEGMENTS SYSTEM "oagis_segments.dtd">
%SEGMENTS;
<!-- ==================================================== -->
```

Now, the DTD describes the valid content (that is, the elements and attributes) of POST_JOURNAL_004, a sort of container for journals:

```
<!ELEMENT POST_JOURNAL_004 (CNTROLAREA, DATAAREA+)>
    <!ATTLIST VERB value CDATA #FIXED "POST">
    <!ATTLIST NOUN value CDATA #FIXED "JOURNAL">
    <!ATTLIST REVISION value CDATA #FIXED "004">


     <!ELEMENT DATAAREA (POST_JOURNAL)>
```

Next, the DTD describes the valid content (that is, the elements) of POST_JOURNAL (the actual journal information). A POST_JOURNAL consists of a JEHEADER (journal header) and one or more JELINE (journal line) elements:

```
<!ELEMENT POST_JOURNAL (JEHEADER, JELINE+)>
```

This DTD segment describes the valid content (that is, the elements) of JEHEADER (the journal header information). Some of the fields in JEHEADER are optional (as identified by a ?). Elements that are entity names, such as AMOUNT.DOCUMENT.T, are enclosed in parentheses.

```
<!ELEMENT JEHEADER (
(%AMOUNT.DOCUMENT.T;)?,
(%DATETIME.DOCUMENT;)?,
(%DATETIME.PAYEND;)?,
GLENTITYS, ORIGREF, DESCRIPTN?, DOCTYPE?, JEID?,
LEDGER?, USERID?, USERAREA?)>
```

And finally, the DTD describes the valid content (that is, the elements) of JELINE (the journal line information):

```
<!ELEMENT JELINE (
(%AMOUNT.ACTUAL.T;),
(%AMOUNT.ACTUAL.F;)?,
GLNOMACCT, BUSNAREA?, COSTCENTER?, DEPARTMENT?,
DESCRIPTN?,
...
((%DATETIME.ACCOUNTING;) | (ACCTPERIOD, ACCTYEAR)),
USERAREA?)>


<!-- From 001_post_journal_004.dtd -->
```

A sample XML document based on the DTD you've defined would look like this:1

```
<?xml version="1.0" standalone="no"?>
<!--
    $Revision: 6.0.1 $
    $Date: 31 October 1998 $
    Open Applications Group Sample XML Data
    Copyright 1998, All Rights Reserved
```

```
    $Name: 001_post_journal_004.xml $
-->
<!DOCTYPE POST_JOURNAL_004 SYSTEM "001_post_journal_004.dtd">

<POST_JOURNAL_004>
    <CNTROLAREA>
          <BSR>
               <VERB>POST</VERB>
               <NOUN>JOURNAL</NOUN>
            <REVISION>004</REVISION>
          </BSR>
          <SENDER>
               <LOGICALID>XX141HG09</LOGICALID>
               <COMPONENT>INVENTORY</COMPONENT>
               <TASK>RECEIPT</TASK>
               <REFERENCEID>95129945823449</REFERENCEID>
               <CONFIRMATION>1</CONFIRMATION>
               <LANGUAGE>EN</LANGUAGE>
            <CODEPAGE>test</CODEPAGE>
               <AUTHID>JOE DOE</AUTHID>
          </SENDER>
        <DATETIME qualifier = "CREATION" >
               <YEAR>1999</YEAR>
               <MONTH>12</MONTH>
               <DAY>31</DAY>
               <HOUR>23</HOUR>
               <MINUTE>59</MINUTE>
               <SECOND>45</SECOND>
               <SUBSECOND>0000</SUBSECOND>
               <TIMEZONE>-0500</TIMEZONE>
          </DATETIME>
    </CNTROLAREA>
     <DATAAREA>
          <POST_JOURNAL>
             <JEHEADER>
                <AMOUNT qualifier = "DOCUMENT" type = "T">
                         <VALUE>2340500</VALUE>
                         <NUMOFDEC>2</NUMOFDEC>
                         <SIGN>+</SIGN>
                         <CURRENCY>USD</CURRENCY>
                         <DRCR>D</DRCR>
                </AMOUNT>
                <DATETIME qualifier = "DOCUMENT" type = "T">
                         <YEAR>1995</YEAR>
```

```
                <MONTH>12</MONTH>
                <DAY>31</DAY>
                <HOUR>23</HOUR>
                <MINUTE>59</MINUTE>
                <SECOND>45</SECOND>
                <SUBSECOND>0000</SUBSECOND>
                <TIMEZONE>-0500</TIMEZONE>
           </DATETIME>
        <DATETIME qualifier = "PAYEND">
           <YEAR>1998</YEAR>
           <MONTH>01</MONTH>
           <DAY>02</DAY>
           <HOUR>12</HOUR>
           <MINUTE>00</MINUTE>
           <SECOND>00</SECOND>
                <SUBSECOND>0000</SUBSECOND>
                <TIMEZONE>-0500</TIMEZONE>
           </DATETIME>
           <GLENTITYS>CORPHEADQUARTER</GLENTITYS>
           <ORIGREF>RCPT#12550699</ORIGREF>
           <DESCRIPTN>INVENTORY RECEIVED FROM GLOBAL MANUFAC
           TURING</DESCRIPTN>
           <USERID>KURTC</USERID>
    </JEHEADER>
      <JELINE>
           <AMOUNT qualifier = "ACTUAL" type = "T" >
                <VALUE>2340500</VALUE>
                <NUMOFDEC>2</NUMOFDEC>
                <SIGN>+</SIGN>
                <CURRENCY>USD</CURRENCY>
                <DRCR>D</DRCR>
           </AMOUNT>
                 <AMOUNT qualifier = "ACTUAL" type = "F" >
                     <VALUE>001001001</VALUE>
                <NUMOFDEC>2</NUMOFDEC>
                <SIGN>+</SIGN>
                <CURRENCY>USD</CURRENCY>
                <DRCR>D</DRCR>
           </AMOUNT>
           <GLNOMACCT>2310</GLNOMACCT>
         <BUSNAREA>INVENTORY</BUSNAREA>
              <COSTCENTER>CC123</COSTCENTER>
              <DEPARTMENT>DEPT001ABC</DEPARTMENT>
           <DESCRIPTN>INVENTORY</DESCRIPTN>
```

```
                        <DATETIME qualifier = "ACCOUNTING" >
                             <YEAR>1996</YEAR>
                             <MONTH>01</MONTH>
                             <DAY>02</DAY>
                             <HOUR>12</HOUR>
                             <MINUTE>09</MINUTE>
                             <SECOND>45</SECOND>
                             <SUBSECOND>0000</SUBSECOND>
                             <TIMEZONE>-0500</TIMEZONE>
                        </DATETIME>
                     </JELINE>
                     <JELINE>
                              <AMOUNT qualifier = "ACTUAL" type = "T" >
                             <VALUE>2340500</VALUE>
                             <NUMOFDEC>2</NUMOFDEC>
                             <SIGN>-</SIGN>
                             <CURRENCY>USD</CURRENCY>
                             <DRCR>C</DRCR>
                        </AMOUNT>
                        <GLNOMACCT>6940</GLNOMACCT>
                        <DESCRIPTN>ACCOUNTS PAYABLE</DESCRIPTN>
                        <ACCTPERIOD>03</ACCTPERIOD>
                              <ACCTYEAR>1999</ACCTYEAR>
                   </JELINE>
              </POST_JOURNAL>
         </DATAAREA>
    </POST_JOURNAL_004>
```

Other Opportunities

Because XML data is so much more accessible than data stored in proprietary formats, whole new application functions are possible. For example:

- **XML data repositories:** Information publishers can produce information in XML format. XML documents can be stored for later access by a variety of applications, not just by the applications that generated the information. Ideally, traditional reports will no longer exist, but will be replaced by XML documents. These documents will be presented to (and manipulated by) end users with XML-capable viewers. These viewers will provide many more analytical capabilities than are available with simple text searching of an ASCII file. There are also XML-based databases available.

- **Information transfers that have not been cost effective until now:** Custom-built or EDI-based information transfer systems have traditionally been used for high-value or high-volume requirements. Other information transfers have not been addressed, although organizations would clearly benefit from the ability to transfer information between a wide variety of business partners.

  A classic example of information transfer is catalog content management. Buyers would like access to the most current catalog information, but the cost and complexity of accepting and processing catalog information from a variety of suppliers is generally too high. XML-based catalogs are a mechanism by which a supplier can publish a catalog, confident that buyers will be able to process it. There is also the XBRL (or Extensible Business Reporting Language) standard for business reporting.
- **Intelligent searching:** XML-based searching guarantees better results. Not only can content be searched (as is the case today), but content can be cross-checked with meaning (structure). An XML search for "mark price" will not return lists of preprinted pricing labels when what you want is information about Mark Price, the basketball player. Not only can data and meaning be searched together, it is possible to search based on meaning only. An XML-enabled Web site can effectively publish which documents (or Web pages) contain information about certain data types.
- **Intelligent agents:** Applications can be built that will browse an environment (the Internet, your local systems, or perhaps just the hard drive on your PC) and detect items of interest. If you are interested in eventually buying at auction a 500-MHz PC when the price drops below $500, an agent can scan an auction site's XML-based auction status catalog for this information and create an e-mail for you when a system meets your target price.

EXERCISES

ON THE DVD

In the Chapter 16 subdirectory of the Exercises directory on the CD-ROM is a simple Java program that illustrates how a Java XML parser works.

1. Copy the XMLDocAnalyzer.java to your java4cobol directory and compile it.
2. Copy the library.xml to your java4cobol directory.
3. Now run the program using the library.xml file as the file to analyze by typing the following command:

```
➔    java XMLDocAnalyzer library.xml
```

Your output should look like this:

```
1---> library ELEMENT_NODE
2------> #text TEXT_NODE
2------> fiction ELEMENT_NODE
3---------> #text TEXT_NODE
3---------> book ELEMENT_NODE
4------------> #text TEXT_NODE
4------------> title ELEMENT_NODE
5---------------> #text TEXT_NODE The Shining
4------------> #text TEXT_NODE
4------------> author ELEMENT_NODE
5---------------> #text TEXT_NODE Stephen King
4------------> #text TEXT_NODE
3---------> #text TEXT_NODE
3---------> book ELEMENT_NODE
4------------> #text TEXT_NODE
4------------> title ELEMENT_NODE
5---------------> #text TEXT_NODE One Flew Over the Cuckoo's Next
4------------> #text TEXT_NODE
4------------> author ELEMENT_NODE
5---------------> #text TEXT_NODE Ken Kesey
4------------> #text TEXT_NODE
3---------> #text TEXT_NODE
2------> #text TEXT_NODE
2------> non-fiction ELEMENT_NODE
3---------> #text TEXT_NODE
3---------> book ELEMENT_NODE
4------------> #text TEXT_NODE
4------------> title ELEMENT_NODE
5---------------> #text TEXT_NODE Java for COBOL Programmers
4------------> #text TEXT_NODE
4------------> author ELEMENT_NODE
5---------------> #text TEXT_NODE John C. Byrne
4------------> #text TEXT_NODE
4------------> author ELEMENT_NODE
5---------------> #text TEXT_NODE Jim Cross
4------------> #text TEXT_NODE
3---------> #text TEXT_NODE
3---------> book ELEMENT_NODE
4------------> #text TEXT_NODE
4------------> title ELEMENT_NODE
5---------------> #text TEXT_NODE The Sibley Guide to Birds
4------------> #text TEXT_NODE
```

```
4------------> author ELEMENT_NODE
5---------------> #text TEXT_NODE David Sibley
4------------> #text TEXT_NODE
3---------> #text TEXT_NODE
2------> #text TEXT_NODE
```

4. Examine the contents of the library.xml file and the source code for XML-DocAnalyzer to understand how the parser operates.

Reviewing the Exercises

The XML document you parsed is a simple document without a schema or DTD that predefines its content. The parser read the document into memory and created a tree-like structure of nodes within nodes. At the top of the structure is the root node. The Java programs simply requested from the parser the information contained in the node structure of the document.

1. Portions of this XML document are copyright by Open Applications Group. All rights reserved.

17 Introducing Eclipse

In This Chapter

- Installing Eclipse
- Start Using Eclipse
- Run with Eclipse
- Debug with Eclipse
- Refactoring with Eclipse

Eclipse is a modern, extensible, open source Integrated Development Environment (IDE). This means it is the latest technology for developers, can be customized by and for developers, and best of all, it is free for you to use. Some surveys (for example, at www.sdtimes.com/content/article.aspx?ArticleID=30020) suggest that more than 60 percent of Java developers use Eclipse as their primary development tool.

Eclipse comes with many important and powerful features, features that you would expect in a modern IDE. These include a file and project manager, a syntax sensitive editor, support for code refactoring, interfaces to source control systems, such as CVS and Subversion, and many others. Eclipse also provides an interface for plug-ins, which are extensions to the product that anyone can build. Many plug-ins are available, some for free and others for a fee.

The Eclipse development environment is not just for Java. It is an IDE that can be used for any programming language. There is even a plug-in that supports development for COBOL projects.

Installing Eclipse

You can get Eclipse from either the companion CD-ROM (Eclipse/eclipse-java-europa-winter-win32.zip for the Windows version) or directly from the Eclipse Web site at www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/europa/winter/eclipse-java-europa-winter-win32.zip. For developers using Windows, Eclipse is delivered as a zip file, which is a compressed bundle of files. You will need to unpack the contents of that bundle into a directory on your computer in order to install Eclipse, such as C:\java.

If you are not sure how to unpack a zip file, here are some suggestions:

- Double-click on the Eclipse zip file.
- Your computer should open a new window that displays the list of files in the bundle.
- Next, select the option to Extract all the files.
- Choose any place on your computer to copy the Eclipse files.
- If your computer does not know how to open a zip file, you will need to install a copy of a program that can open zip files. Try using either Winzip (www.winzip.com/downwz.htm), PowerArchiver from www.powerarchiver.com/download, or 7zip from www.7-zip.org. After installing one of these tools, your computer will be able to work with zip files. Double-click on the Eclipse zip file, and proceed as previously described.

Start Using Eclipse

At the end of the install process, Eclipse will start up automatically. You can also start Eclipse by double-clicking on the file named Eclipse.exe in the directory where you've unzipped the files (for example, C:\java\Eclipse\Eclipse.exe). After Eclipse starts, you should see the introductory screen shown in Figure 17.1.

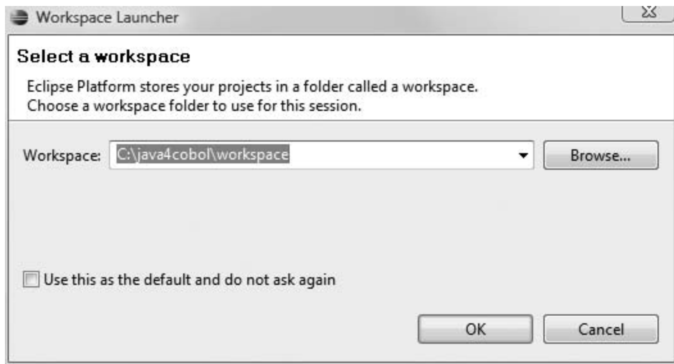


FIGURE 17.1
Workspace Launcher.

Enter the location on your computer to store your Eclipse project. Try using C:\java4cobol\workspace. Next, you should see the Eclipse home page, as shown in Figure 17.2.

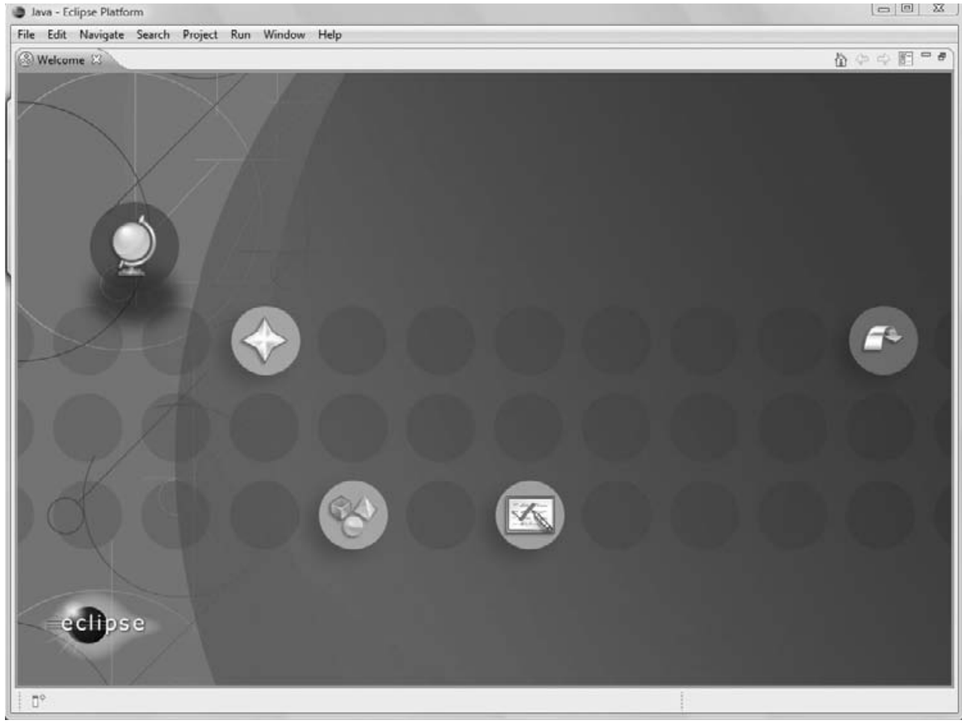


FIGURE 17.2
Eclipse home page.

The icons in the center take you to various built-in information available in Eclipse. You should spend some time reviewing the Tutorials section to get more information on how to use Eclipse and look at the code samples to get insights into how to write high-quality Java code.

To start using Eclipse, select the Workbench icon on the right. You will see the default Workbench view for Eclipse, as shown in Figure 17.3.

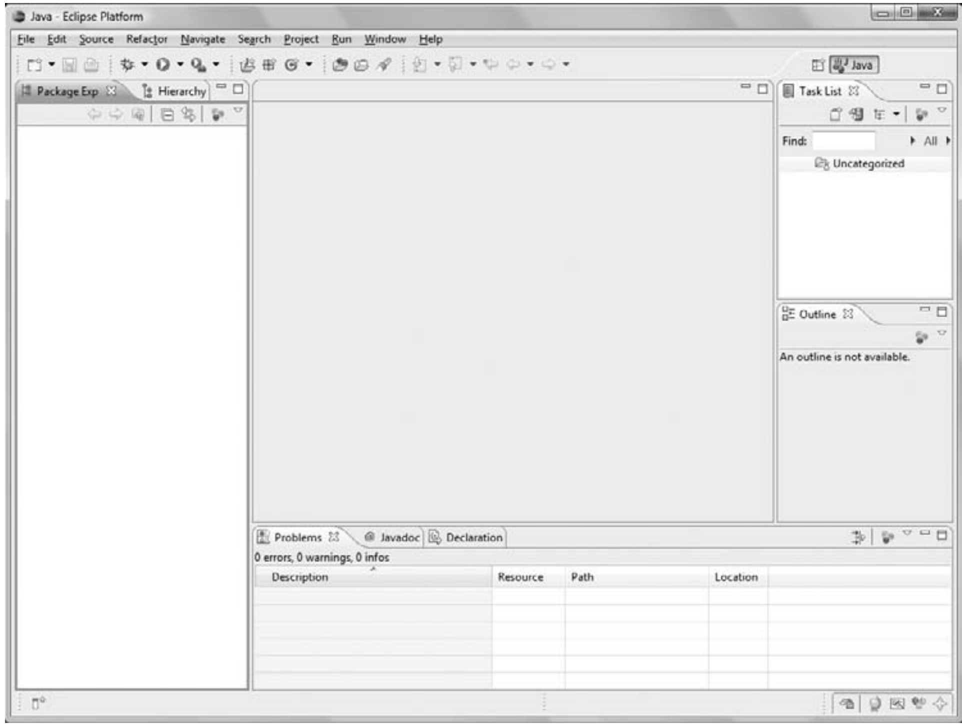


FIGURE 17.3
Eclipse Workbench view.

You will spend most of your time in the Workbench view. You can edit, compile, and run your Java code without switching to any other window or view.

MAKE A NEW ECLIPSE PROJECT

Let's create a new project. Select the File menu path on the top-left corner, and then select the New > Java Project menu path to get to the dialog shown in Figure 17.4.

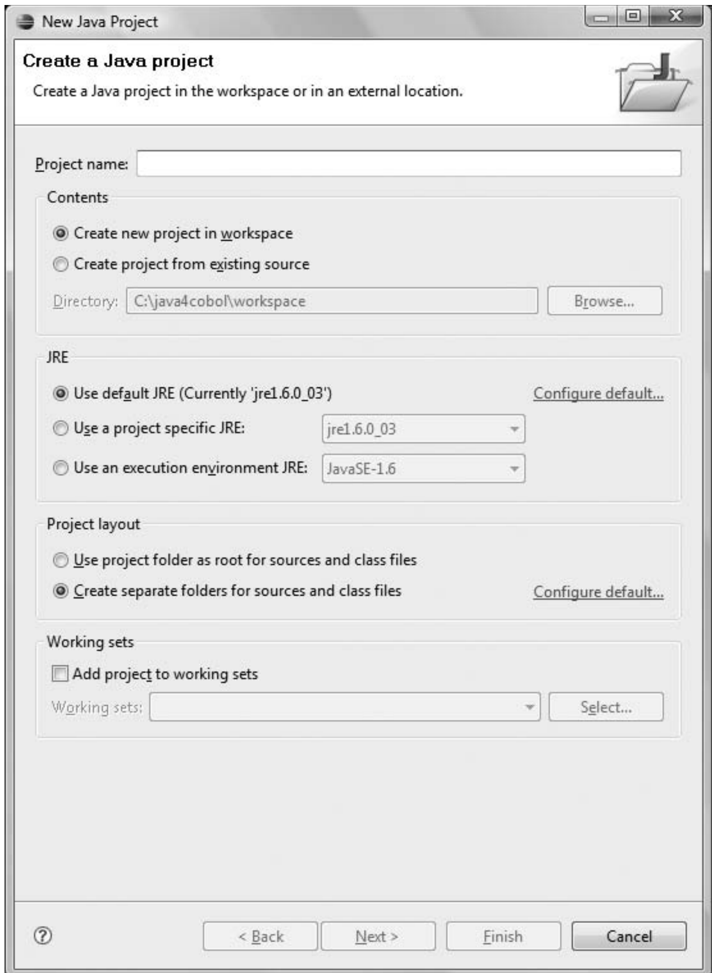


FIGURE 17.4
Create a new Java project.

For your first project, you will use the examples from Chapter 8. Type **Chapter 8** in the Project Name field and click Finish. Eclipse will create the project files it needs.

Next, you need to add the existing Java classes to the project. Right-click on your new project in the Package Explorer window, and select Import. You can import into Eclipse from several places; you will import from the filesystem. Choose the General import filter type, and from the list that is displayed, select the FileSystem type. Then click the Next button, highlighted in Figure 17.5.

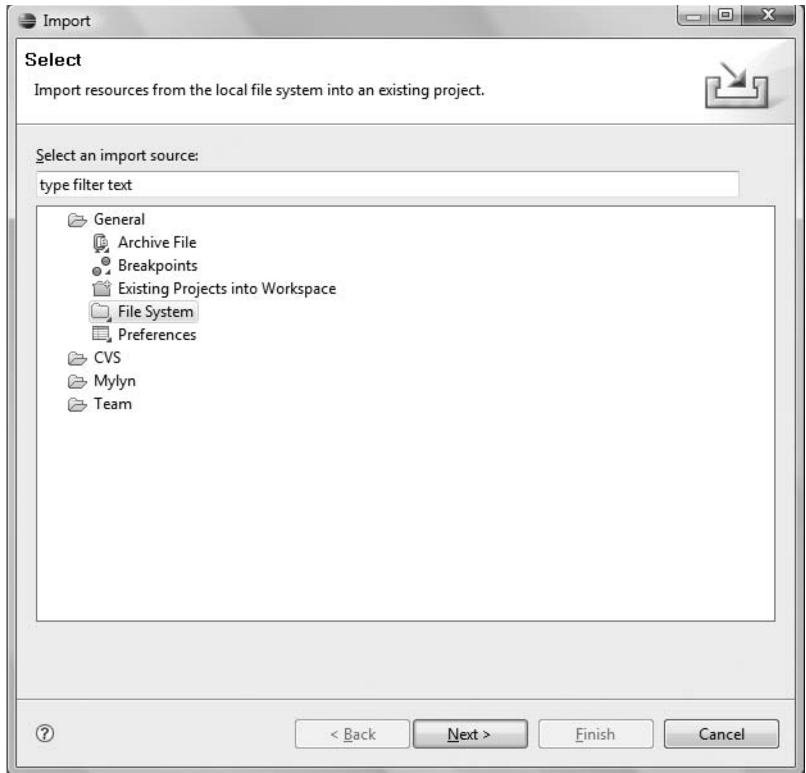


FIGURE 17.5
Import Java files.

In the next dialog, enter the directory name where you placed the Java files for Chapter 8. If you want to import the files directly from the companion CD-ROM, the name will be similar to E:\ Exercises\Chapter08\java4cobol, where E: should be replaced with the drive letter for your CD drive. Enter the directory name where the Java files are located, and press the Tab key.

Click on the selection box for the java4cobol directory name in the center of the screen, and then click the Filter Types button. In the next dialog, you can select what file types to import. Select the *.java option, then click OK.

Finally, make sure the target for the import step (that is, the Into Folder field) is the Chapter8\src directory.

If you've done everything correctly, you should see the java files selected, as shown in Figure 17.6.

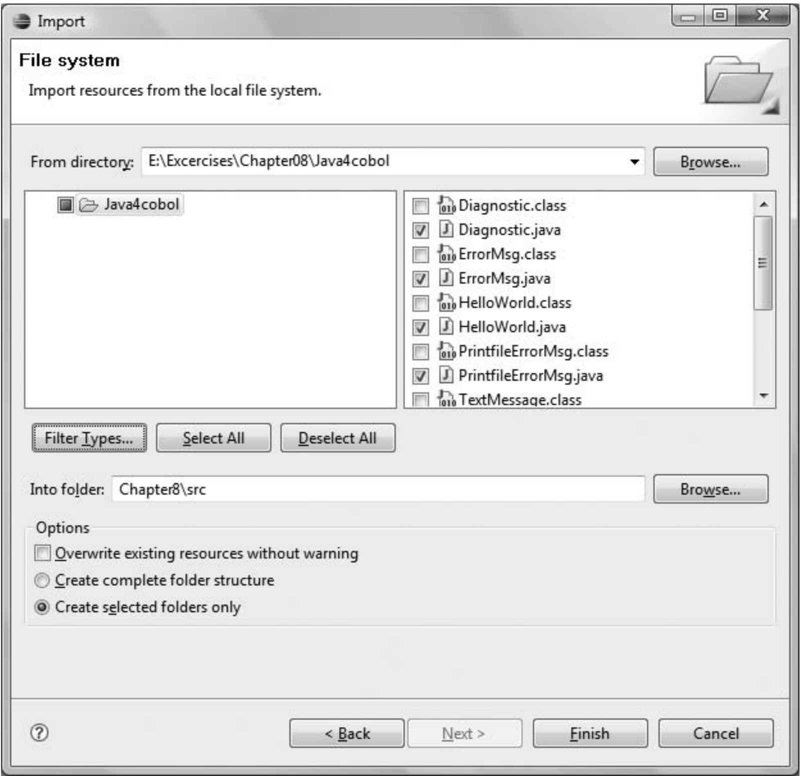


FIGURE 17.6
Import from file system.

Click the Finish button. Eclipse will import these java files into the src directory in your project.

Click on the src subdirectory in the Package Explorer in Eclipse, and then the Default Package subdirectory. You should see the java source files from Chapter 8 in the Explorer. Try clicking on one of the Java files. You should see the Java code in the edit window in the center of the screen. Notice how Eclipse color codes the different parts of the Java code to make it easier for you to read and understand the code.

You can have Eclipse show you the line numbers of your source statements. Move your cursor to the left-most section of the source edit window. There should be a blue bar on the side of that window. Click the right mouse button, and select the Show Line Numbers option from that dialog.

Eclipse has already compiled the classes for you. Check the bin directory in your project workspace. If you've followed the directory name suggestions, that directory would be C:\java4cobol\workspace\Chapter8\bin. The bin directory contains the newly compiled class files.

Run with Eclipse

Now it is time to execute your project with Eclipse. Choose the Run menu choice at the top of the Eclipse screen. Then choose the first item on that menu, which is Run. If Eclipse should ask you what type of project you have, select Java Application. Eclipse will now execute the HelloWorld class.

At the bottom of the Eclipse screen, in the console window, you should see the last part of the output from the Chapter 8 exercises. Use the scroll bar for that window to see the beginning of the output.

Debug with Eclipse

Next, you will use the Eclipse debugger. Double-click on the HelloWorld Java source file from the Package Explorer. It will open up in the edit window. Then click on the line that says `System.out.println("Hello World!");`.

Move your cursor to the left-most section of the edit window. There should be a blue bar on the side of that window. Click the right mouse button, and select the Toggle Breakpoint option. A small blue ball should appear on the line, indicating a breakpoint has been set for that line, as shown in Figure 17.7.

Choose the Run menu choice at the top of the Eclipse screen. Then choose the second item on that menu, Debug. Eclispe will ask you to confirm that you want to switch to the debug perspective. Choose Yes. Eclipse will now execute the HelloWorld class in debug mode.

Eclipse has many features in debug mode. The most common feature, and the one you will use first, allows you to watch as you step through your program. At this point, Eclipse has started the HelloWorld program and is stopped at the line with the breakpoint. Try pressing the F6 button on your keyboard several times, until you see the second statement with `System.out.println`. Notice how Eclipse steps through the Java code. After the `println` statement is executed, notice how the console window at the bottom is updated with the output from `println`.

Eclipse lets you easily see the current value of objects. Position your mouse over the `tempMsg` variable. Notice how Eclipse shows you the contents of `tempMsg`.

The most common actions in debug mode are available in the Run menu choice at the top of the Eclipse screen. Select that menu to see the actions available. Choose the Resume option to let the program continue to completion.

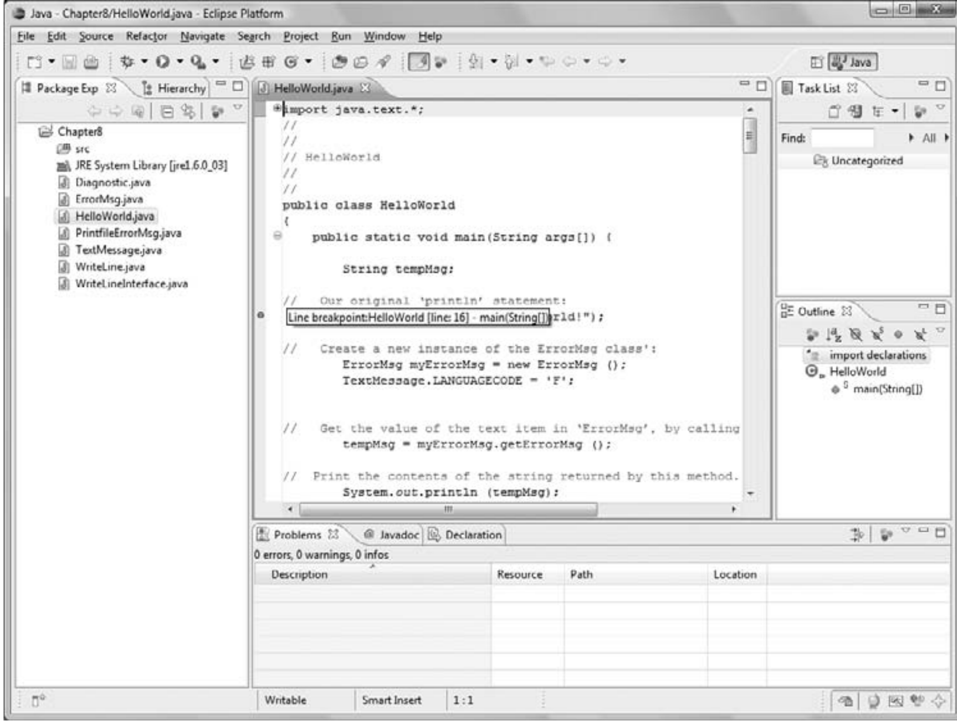


FIGURE 17.7
Set a breakpoint.

Refactoring with Eclipse

Eclipse improves the productivity of developers in many ways. One important feature is the ability to reorganize, or *refactor* your Java code. Often during a Java development project, you will want to reorganize your class names or decide that an alternative packaging plan is an important improvement.

If you were to make these changes by hand, you would need to find, edit, and compile each Java file that was impacted by the change. If part of your changes included repackaging, then you would need to create new directories, and move files into the proper directory. Eclipse can make all of these changes automatically for you.

Let's start by importing the exercises from Chapter 2. These exercises are relatively simple, but actually contained an interesting challenge in the package definition of the classes.

Switch back to the Java perspective, by selecting the Java perspective icon at the top right of the screen. Create a new Java project in Eclipse and call it "Chapter2." You should see your new project listed in the Package Explorer Window.

Then, import the files from the Chapter 2 exercises, using the same steps used to import the files from the Chapter 8 exercises. In the Filter Types dialog in the Import function, add *.html and *.java to your choices so that you get the java source and the HelloWorld.html file, which you will need to run the applet. Make sure to point the target of the import to Chapter2\src.

After Eclipse has imported the Java and HTML files, you will notice a red X box in the Package Explorer. Follow the red boxes to find the problem, by opening each subdirectory with red X boxes in the project, until you get to the applet\HelloWorld Java file. Double-click on that file.

You should see the applet\HelloWorld Java source opened in the code window, and a red X box on the first line, as shown in Figure 17.8.

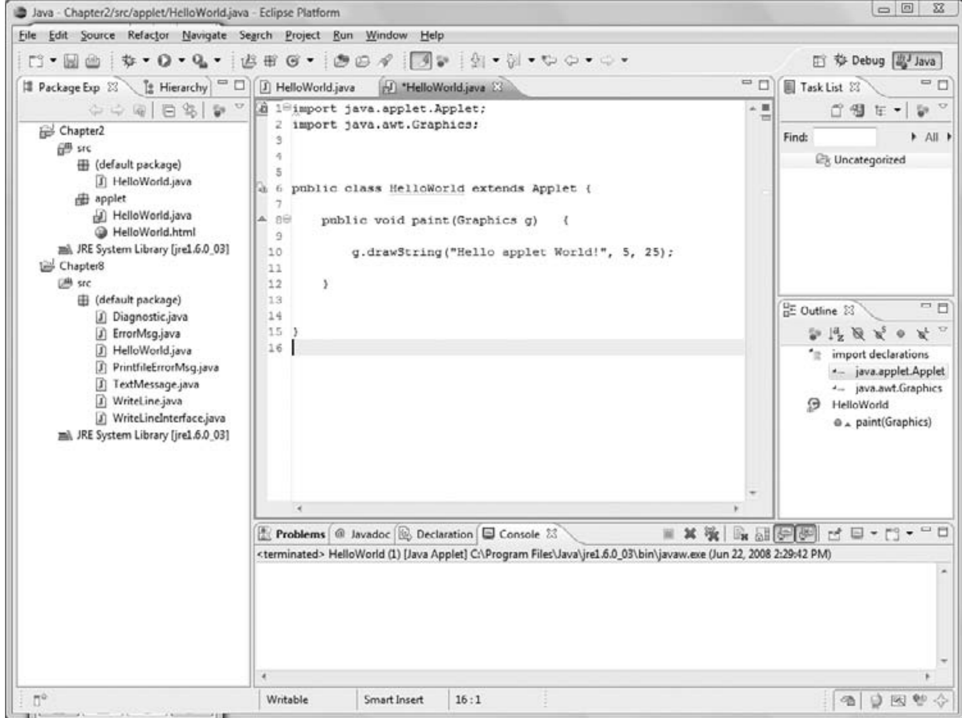


FIGURE 17.8
Applet in Eclipse with error.

Click on the red X box in the code window. Eclipse will offer a suggestion to fix the problem by adding the package declaration "applet" to the file. Double-click the suggestion, and Eclipse will add the line `package applet;` at the top of the

HelloWorld Java source file. Save this change by pressing CTRL-S or by using the File > Save menu path. You should see the red X boxes disappear as Eclipse recompiles the `applet.HelloWorld` class.

Now you can run the HelloWorld applet. Select the Run > Run As > Java Applet menu choice. You should see the HelloWorld applet window pop up, with the "Hello applet World!" message in the dialog. Exit the window, and try changing the message to "Hello applet World in Eclipse!"

For the final exercise, you will deal with a more complicated example of refactoring, using the exercises from Chapter 3.

Create a new Java project in Eclipse and call it "Chapter3." Then, import the files from the Chapter 3 exercises on the CD-ROM. In the Filter Types dialog in the Import function, add *.html and *.java to your choices so that you get the java source and the HelloWorld.html file, which you will need to run the applet. Make sure to point the target of the import to Chapter3\src.

You should see the familiar red X box in the Package Explorer for Chapter3. Before you fix that problem, you want to organize this project a little differently. You want to move the two classes in the default package to the package named java4cobol.

In the Package Explorer, open the Chapter3\src\default package directory. You should see two Java classes listed, `HelloWorld` and `ErrorMsg`. Using the Shift key and the left mouse button, select both of those classes. Then, using the right mouse button, choose Refactor from the menu list. Finally, select the Move option from the refactor choices, as shown in Figure 17.9.

On the next dialog, titled Move, click the Create Package button. Enter **java4cobol** as the package name in the next dialog, and click Finish. Then click OK in the Move dialog. You may see a dialog indicating some potential matches have been found. If you do, click the Continue button.

In the Package Explorer, you can see that Eclipse has moved the two selected classes from the default package to the package named java4cobol. Open up that new package, and double-click on the `ErrorMsg` class. You can see the new package defintion inserted by Eclipse into the `ErrorMsg` java source file. Eclipse has also moved the java files source from the default package location to the java4cobol directory. You can check this by looking at the directory C:\java4cobol\workspace\Chapter3\src\java4cobol on the file system.

Next, you will fix the two problems in the applet\HelloWorld class. Open the source for Chapter3\applet\HelloWorld in the source edit window by finding that class in the Package Explorer and double-clicking on it.

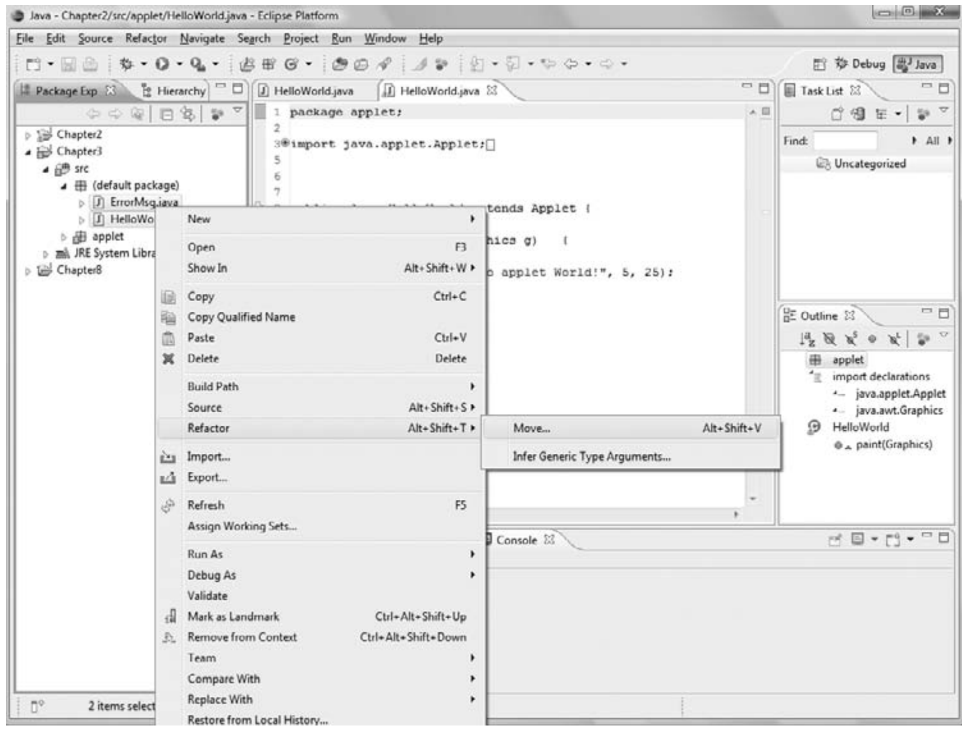


FIGURE 17.9
Refactor in Eclipse.

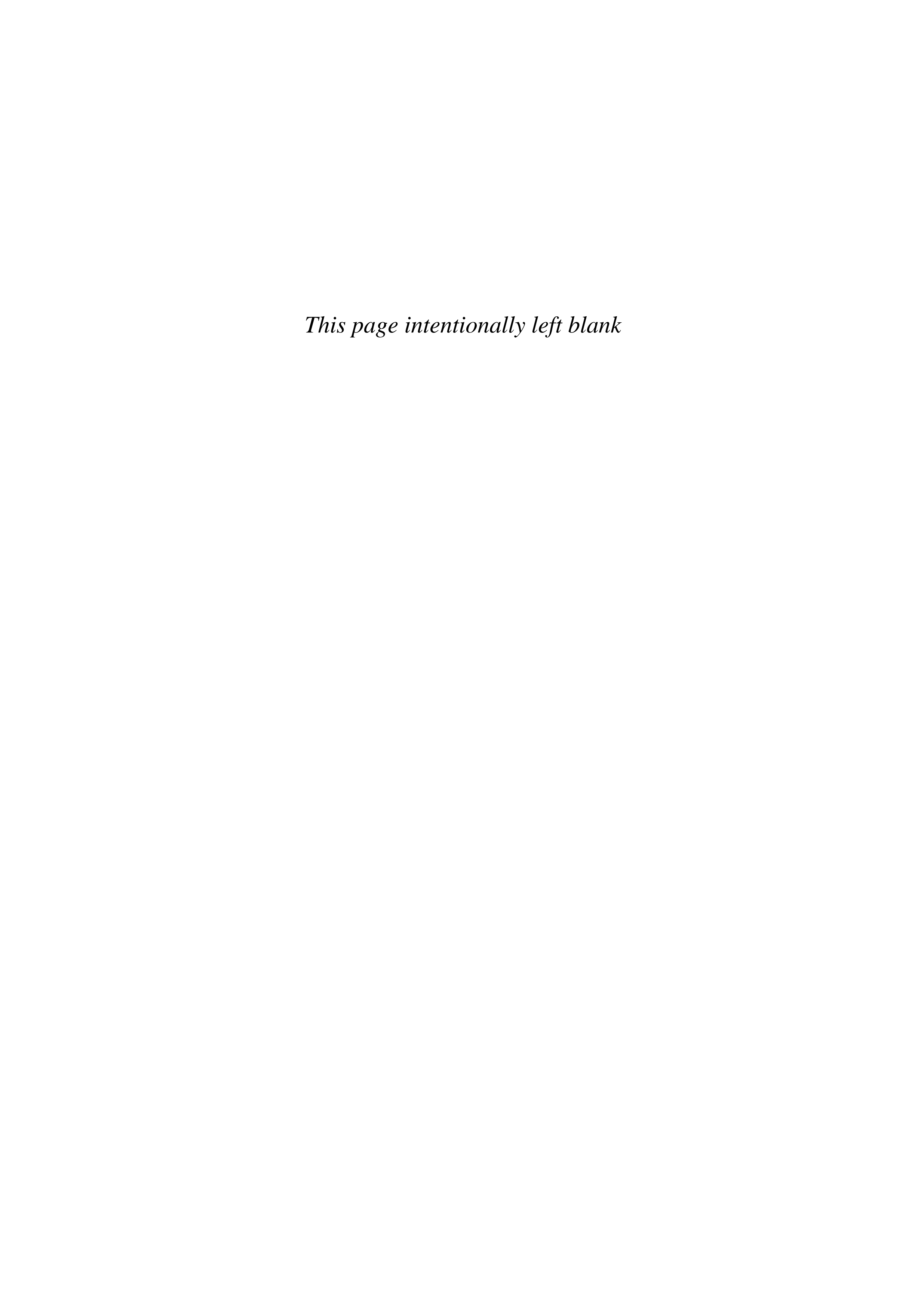
Add the `package applet` statement to the first line of that class, or use Eclipe to help you do that, just as you did in the previous exercise. This will fix the first problem.

To fix the second problem, you will need to import the `ErrorMsg` class from the java4cobol package you just created. After the two import statements, add a new one: `import java4cobol.ErrorMsg;`. The beginning of the Java source for applet\HelloWorld should now look like this:

```
package applet;

import java.applet.Applet;
import java.awt.Graphics;
import java4cobol.ErrorMsg;

public class HelloWorld extends Applet {
...
```

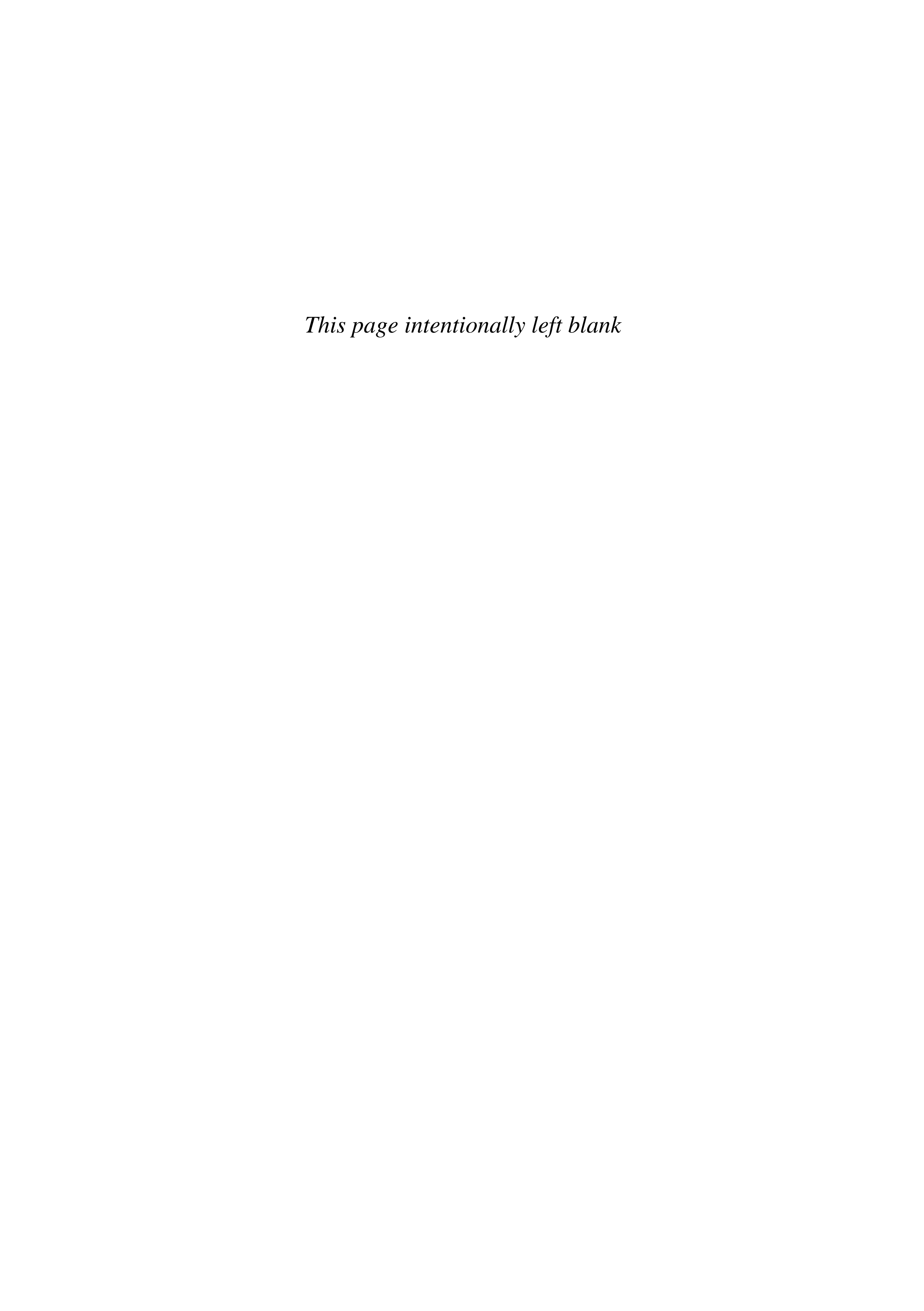
Now, when you save the source file, the red X boxes should all disappear. And if you run that program using Run > Run As > Java Applet, you should see the correct text for this exercise in the applet dialog.

Eclipse is arguably one of the more important recent devlopments in the world of Java programming. In your future career as a Java developer, time spent learning how to use Eclipse effectively is time well spent.



This page intentionally left blank

Part IV Appendixes


This page intentionally left blank

Appendix A
About the CD-ROM

In addition to the code samples, source files, and Sun's SDK, there are other folders on the CD-ROM that hold one or more executable program files that will install various programs. Descriptions of the contents of each folder and instructions for their use follow.

Exercises

The CD-ROM contains all the code samples in the book organized by chapter. You can and should use these samples as reference materials as you perform the exercises, but I wouldn't recommend just copying and compiling them. You will get more out of the exercises if you follow the step-by-step instructions in the book. No pain, no gain.

Java SDK

Java 2 SDK Version 1.6
Sun Microsystems, Inc.
www.java.sun.com

The Sun SDK Version 1.6 is necessary to fully utilize this book. You may either download the SDK from Sun's Web site or install the SDK using this CD-ROM. Run the executable in the Java JDK folder to install the SDK. You may place the SDK in any directory. The default is C:\Program Files\Java\jdk1.6.0_03.

LegacyJ

PERCOBOL Compiler
LegacyJ
www.legacyj.com

This is an evaluation version of LegacyJ's PERCOBOL compiler, a Java/COBOL integration product. Using PERCOBOL, developers can write code in COBOL and deploy in Java. In addition to the PERCOBOL demo, LegacyJ has included both COBOL and Java sample programs for you to review.

To install this program, run the program setupwin.exe in the LegacyJ directory. Review and answer the license information, then select a directory for loading the software.

This evaluation demonstration version can only be used from the command line; for example, you can run the PERCOBOL conversion program by typing `percobol` in the command window. This information and demonstration software are updated periodically.

Eclipse

Eclipse Open Source IDE
Eclipse Foundation Inc.
www.eclipse.org

Eclipse is a popular, robust, open source GUI development environment for Java. The CD-ROM contains the Windows installation program in the Eclipse directory.

Tomcat

Open Source Web server
Apache Software Foundation
http://tomcat.apache.org/

Tomcat is a robust, open source Web server that implements the latest Java Servlet and Java Server Pages technologies. Tomcat is often used by Java developers for both their development work and in production. The CD-ROM contains the Windows installation program in the Tomcat directory.

SoftwareMining

CORECT
SoftwareMining, Ltd.
www.softwaremining.com/

This is another COBOL-to-Java conversion tool. To install this program, run the program sm-corect-v81-setup in the Corect directory.

SoftwareMining can assist with these larger COBOL-to-Java projects by providing a two-step modernization approach:

- **Step 1:** Translation of COBOL code to maintainable Java code. The new code is legible to both the retrained COBOL programmers and to any new Java programmers joining the team. Testing of the new system is undertaken by comparing the results produced by the Java system to that produced by the original COBOL system. SoftwareMining also assists with data-migration activities to achieve migration of all test and production data.
- **Step 2:** Having successfully translated and tested the new Java application, developers can then start any smaller enhancements. These typically include the following: screen redesigns, application of service-oriented architecture, refactoring, and moving code between different classes.

SoftwareMining's Automated Translation route provides substantial benefits:

- The translated system adheres to high programming standards allowing future maintenance and enhancements in Java
- The project implementation requires only a small team, reduces the margin for error, and enables completion within budget

The following document outlines the technical architecture of the generated code: www.softwaremining.com/download/SM-TechnicalOverview.pdf.

Case Studies

A major Canadian bank recently used the CORECT Toolkit to translate their IBM/Unisys COBOL application to Java. A manual rewrite would have achieved translation of one program per month. SoftwareMining's Automatic Translation achieved one translated/tested program per day, making it 20 times faster than the manual rewrite with significantly reduced project risks.

In another project, a U.S. state tax office migrated their IBM COBOL application to Java in order to bring the system up to date with new technology and to reduce maintenance costs. A systems integrator used SoftwareMining's CORECT

Tool to translate the COBOL application used by 250 people. On completion of the testing phase, a second enhancement phase was started to redesign screens, apply service-oriented architecture, and make functional enhancements. The translation of the 5 million lines involved only eight Java developers and testers and four users performing user acceptance testing. The project was successfully completed and delivered within 14 months.

For further information please visit www.softwaremining.com.

Appendix B Java Information Available Elsewhere

This is a listing of a few of the Java resources available on the Internet. There are several good introductory tutorials, as well as resources for in-depth information. As with any list, this one is surely incomplete, and feedback is welcome.

Java Resources

http://java.sun.com. The original source for Java information. Lots of documentation, downloads, tutorials, and other useful information.

www.ibm.com/developer/java. IBM's Java site is loaded with information, sample code, FAQs, training material, and so on.

www.javacrawl.com. An automated crawler for the latest in Java happenings.

www.programmingtutorials.com/java.aspx. A good listing of Java tutorials, from the basic to the complex.

www.theserverside.com. Information about J2EE development.

Java Magazines

www.javareport.com. An informed and informative Java magazine. Absent most of the religious fervor found in some Java publications, it focuses on the current state of the art in Java technologies.

www.java.sys-con.com. A Java magazine for developers.

www.javaworld.com. A complete Java resource, including reviews, code samples, and good writers. A superior online presence as well.

Java Tools

www.javashareware.com. A complete Java site that contains Java projects of all types, FAQs, answers to questions from experts, and other great resources. The focus is on promoting Java with shared and useful resources.

www.java-source.net. Another good source for free Java tools, information, and code.

COBOL Information

www.acucobol.com. Acucorp's home page. Cross-platform COBOL design and deployment tools.

www.microfocus.com. The home page for MicroFocus COBOL compiler.

www-306.ibm.com/software/awdtools/cobol. IBM's COBOL site.

www.flexus.com. The home page for Flexus, a COBOL tool vendor. This site also contains a good collection of links to other COBOL sites.

www.infogoal.com/cbd/cbdtol.htm. An exhaustive listing of COBOL tools and information, both online and offline.

Appendix C
Buzzwords

Active Server Pages (ASP)

Active Server Pages are Microsoft's version of Java Server Pages or *vice versa*, depending upon your perspective.

The widespread acceptance of Internet standards and business processes that use the Web has increased the number of deployment environments available to a business developer. One such environment is the Web server itself. Normally, a Web server simply sends static HTML documents and other file types to a Web browser. A business process requires much more dynamic content, as information is collected from business systems and presented to the end user.

The original technique used to extend a Web server is a Common Gateway Interface (CGI). A browser can request a CGI program instead of a standard HTML page. The Web server will execute the CGI program or script at the request of the Web browser. The CGI program can in turn reply with HTML data to the Web server; this HTML data is passed along to the Web browser.

However, CGI suffered from poor performance and a lack of frameworks to support it. To solve these and other problems, Microsoft has defined a proprietary extensibility model for their Web server called Active Server Pages (ASPs), or more recently, ASP.Net. ASPs allow the developer to program, or script, Web server processing in much the same way that a Web browser can be scripted. Using ASPs, a developer can integrate data from a database with static HTML code and perform program processing logic. The output of an ASP on the Web server is often standard HTML, which is sent to the Web browser for display.

The ASP environment combines standard components with a choice of scripting languages (VBScript and JavaScript) and an object model that defines the current Web page. Using these resources, the developer can control the content, presentation, and sequence of any part of the Web page as it is created. The result is a rich, dynamic Web page construction environment.

AWT

The basic GUI functionality for Java is contained in the abstract window toolkit (AWT), released with the earliest versions of Java. AWT is a standard interface that Java applications (or applets) can use to create and interact with GUIs. The AWT interfaces talk directly to the underlying graphical operating system interfaces. Since AWT is ported to every Java environment, any Java application that uses AWT can display GUIs in any Java environment.

However, AWT has been generally regarded as the weakest part of the original Java environment. The primary problem is that AWT is too low level for most application developers and does not do a good job of hiding all the differences between graphical hosting systems. At the same time, important capabilities required by graphical applications are not available in AWT. Sun has improved on AWT with the JFC/Swing classes released in the more recent versions of Java.

Client/Server

Client/server systems and architecture have changed the way most new computer systems have been built over the last decade and a half. Client/server systems have effectively replaced the traditional mainframe-hosted model as the most popular environment for new business systems in many organizations.

A client/server system is just that, a system with two primary components: a client (typically, a PC) and a server. The client passes messages to the server, and the server performs the requested function and responds with a result. The simplest client/server architecture is a two-tiered relational database model, with a relational database as the server. The application logic is on the client and passes SQL requests to the server. The server responds with data from the database. More complex models involve separate application logic layers, which serve as glue to connect the client to the database server.

The server in a client/server system does not need to be a single physical machine or even a single component. The server often consists of an application server component and a database component. In high-volume situations, there can be more than one application server and more than one database server.

Common Object Request Broker Architecture (CORBA)

I've discussed how you can create and use objects as part of your program. In some cases, you may need to create and use objects that exist on another system. Objects can be created and accessed on another system by using an object request broker (ORB). An ORB provides services that allow you to create objects on a remote system and to call that object's public functions.

The OMG has defined a specification for a standard ORB, the Common Object Request Broker architecture (CORBA). This detailed specification describes which services a standard ORB should provide and how they should be implemented. A number of vendors (such as BEA, HP, Imprise, IBM, and Sun) have implemented products that conform to the CORBA specifications. These products support a wide variety of operating systems and development languages.

CORBA products have been successfully used in a number of large-scale OO projects. They provide good, and efficient, distributed object-management services. However, CORBA products have not had widespread acceptance for a number of reasons:

- **Complexity:** CORBA products and the CORBA interfaces can be difficult to master. Effective use of CORBA requires a serious commitment to learning a complex technology.
- **Language and development environment affinity:** One of CORBA's assets is the fact that it has been designed as a language-neutral and platform-neutral technology. This is also one of its problems. The developer has to step outside the application programming model and address the CORBA model. Only recently have some popular language-specific development environments wrapped CORBA interfaces in a way that feels natural to the application developer.
- **Availability and cost:** As a rule, development projects that do not have special requirements for deployment will be more successful at project rollout time. If your project requires a CORBA product to be installed at each client system, then you will likely have to manage that task as part of your product installation. You may have to charge for it as well.
- **Interoperability:** The CORBA spec does not include a working reference model, so minor differences are likely to exist in any actual implementation. In addition, vendors are free to extend their implementations with specific features. As a result, most CORBA products from different vendors do not work well with each other.

Components

Components are collections of reusable code, organized so that some other application can use them. Components share many of the same characteristics and design goals as objects (code reuse, public interfaces, and so on), but have a different emphasis. The most important aspect about a component is not how it is built, but rather how it is to be used.

Components can be created in any language and can publish an interface that has little to do with how it is constructed internally. Components are often collections of objects of various types; as such, they can represent a fairly complex service. One example of a large, commonly used component is a relational database system.

Objects can be represented as components, and a component can consist of one or more objects. However, an object will naturally publish its interface based on its class definition and the definitions of the classes it inherits. In addition, unless your development environment supports multiple inheritance (Java does not), it is difficult to publish an OO interface that reflects the results of a coalition of several objects.

Therefore, building a component and using a component architecture is more than just publishing the interfaces of important objects in your Java system. It involves thinking about how client applications might use these objects (or combinations of these objects) and how you should best publish interfaces. In effect, the component designer is building services to be used by other applications.

Component Object Model (COM)

Microsoft's standard for client-side component access is COM. This is a standard architecture that components can use to publish their interfaces (interactively, at design time, or at runtime). Component consumers can access, modify, and call these components. COM components publish the data items they contain as properties and the functions they support as methods. COM components are language neutral, meaning they can be written in any language and used by any language. However, COM technology is limited in practice to the Windows platforms.

Microsoft development tools can easily wrap your code in COM or COM+ components. All Microsoft environments (even environments that are not, strictly speaking, development environments, such as Excel) can make use of COM components. COM is optimized for performance on a single PC. Applications can use COM without significant performance penalty as long as both component consumer (the client) and the component are on the same PC.

Distributed Component Object Model (DCOM)

Microsoft's standard for distributed object services is its distributed component object model (DCOM). This is both a specification and an implementation. Furthermore, it is technology that is embedded in every relevant software Microsoft product, from compilers to transaction servers to database engines. It is now even part of the Windows operating system.

The good and bad news about DCOM is that it is very Microsoft specific. Most of the challenges listed in the discussion of CORBA are not problems with DCOM, as long as you use only Microsoft, or Microsoft-centric products. If you need to use non-Microsoft products, or non-Microsoft platforms, then the availability of DCOM on that platform will be questionable.

DCOM has the reputation of not performing as well as its competitor, CORBA, although it is hard to find real-world studies that confirm this complaint. On the other hand, DCOM is very easy to use. Since DCOM is based on the ActiveX component standard, most Microsoft compilers can easily create DCOM components, and nearly every Microsoft tool can use DCOM components.

Enterprise JavaBeans (EJBs)

EJBs bring the benefits of Java-built components to distributed, transaction-oriented systems. An EJB can be instantiated (created) on a remote system, and then respond to message requests from an application executing on a local system. Best of all, EJBs don't have to manage the complex system plumbing that is normally associated with remote-component management and transaction integrity control. All these system-level issues (component invocation, life-cycle management, security, and transaction semantics) are handled by the EJB infrastructure. A major design objective of the EJB spec is to allow developers to focus on writing code that solves the business problem(s) at hand rather than the technical issues surrounding the management of remote distributed component services.

The EJB specification is primarily an effort to standardize remote component-management techniques available to Java components. It also attempts to simplify and improve existing techniques, such as CORBA and DCOM. EJB is very Java-centric; it fits naturally into the Java language. It is only modestly more difficult to build and use EJB components than it is to build and use regular objects in Java.

The major problems with EJB are these:

- **EJB is a Java-only technology:** If your project requires transaction and remote object-management services for other languages, then DCOM and CORBA are the only choices.

- **EJB is a specification, although there is a working reference model:** It is possible that some vendors' EJB management products may not work well with each other.
- **EJB is not built into any operating system:** Therefore, Java-built projects that use EJB may have to distribute the EJB components in addition to the Java components.

File Transfer Protocol (FTP)

The popularity of the Internet is due, in no small part, to the adoption of standard functions by a wide variety of tools. FTP is one of those functions.

FTP is an efficient protocol that can be used to transfer files from one system to another. It includes commands to move binary and text files, one at a time, or in groups. It also includes commands to list the contents of a directory on either the local system or the remote system. Browsers, Web servers, and other Internet tools all support FTP. Often, large file downloads performed over the Internet use FTP as the protocol.

Firewall

Web servers are designed to be accessible to the public. An Internet mall, or Internet shopping site, wants to place few restrictions on who can visit the site or when they can visit. A "corporate presence" Web site wants to post information about the company for all to see.

Internet systems, because they are so public, are subject to a variety of attacks from hackers. Hackers can be either just curious or malicious "crackers." Recent high-profile attacks on government sites have highlighted the potential risks involved in placing important systems in so public a venue.

A firewall system is meant to protect public Web sites from attack. It is placed between the external Internet and the system it is meant to protect. All traffic meant for the protected system is first passed through the firewall, where it is inspected and perhaps rejected. Only approved traffic is allowed to reach the protected Web site.

A firewall prevents most of the obvious attacks by restricting traffic to and from a Web server to approved message types. For example, a firewall might allow only http message requests directed to a specific port (a type of input channel) on a particular Web server. Other requests directed to that Web server (for example, requests to copy files or to change passwords) will be rejected.

Hypertext Transfer Protocol (HTTP)

HTTP is another important Internet standard. This is a simple mechanism for a client application (a browser, for example) to request a file from a Web server. Typically, Web pages are accessed using the HTTP protocol.

To use HTTP, a browser needs only to access the file using a public address called a URL (universal resource locator). The HTTP request is automatically directed to the Web server identified in the URL, and the required file is transferred by the Web server to the browser. The browser can then read the information contained in the file, and format a user interface based on the commands in the file.

A Web page ordinarily consists of many files. All the graphics files in a Web page are normally stored on the Web server as separate files. A Web browser actually makes many HTTP requests to get all the files necessary to display a Web page. Only after all the files referenced in the Web page have been downloaded and displayed will the entire Web page be complete.

Integrated Development Environment (IDE)

Java applications are often developed using an IDE. These tools help the Java developer edit and manage source code, compile applications, and debug problems. Modern IDEs are graphical applications with lots of productivity features, and often include interfaces to source version control systems.

Interface Definition Language (IDL)

IDL is both a standard and a technique. IDL files are external descriptions of the interfaces to a component or an object. CORBA includes a standard IDL specification that is used to describe any object that will be managed by an ORB. The IDL Language for Web Services is WSDL.

Development tools often use IDL files to store information about the interface to a component in a readily accessible manner. Some tools allow the developer to use a wizard application to quickly create an IDL based on some existing class. Other tools use IDLs as the repository of information about available components.

Internet Inter-ORB Protocol (IIOP)

IIOP is a standard mechanism to pass messages between applications (specifically, between an ORB server and the client that uses the ORB) over the Internet. For example, IIOP can also be used as a protocol by which Web servers accept and respond to ORB message requests from Internet clients. An ORB client will typically be a browser, but any Internet client is acceptable, including another Web server. Most of the popular general-purpose Web servers support IIOP, as do some special-purpose servers built primarily to service IIOP requests. IIOP is specifically mandated as the Internet message protocol for CORBA.

JavaBeans

A JavaBean is a self-contained Java component. It is a self-describing, reusable software unit intended to be used by another application. A Java bean is a Java class that implements serializable, has a public default constructor, all instance variables are private in scope, and getters and setters for all instance variables. However, modern Java beans do much more than simply provide a means to set and get properties.

A Bean is designed so that a development environment can (visually) publish the Bean's interface and allow the developer to easily manipulate the interface. A Java Bean supports Java functions that allow a development tool to query the component (*introspection*). Developers can query the Bean to get additional meta data about its properties (*annotation*). Beans support events (predefined messages) that can cause the Bean to be stored and subsequently reused (*persistence*). The Java Bean standard is roughly equivalent to Microsoft's COM component standard, except that COM is a language-neutral, platform-specific standard, whereas JavaBeans are language specific and platform neutral.

Java Native Interface (JNI)

The Java native interface (JNI) is the standard mechanism for Java applications to access functions written in other programming languages, including C and C++. Java applications can also use JNI to access features specific to the host operating system. JNI is supported by all the VMs provided by vendors and on all Java platforms.

JavaScript

JavaScript is an interpretive scripting language developed by Netscape. It is intended as a generic (that is, cross-platform) scripting language able to extend the standard functionality of Web servers and Web browsers. It doesn't really have anything to do with Java per se, other than some syntax similarities and its cross-platform promise.

In many situations, a developer needs a programming language in order to implement certain types of application capabilities. Static HTML pages are not always enough to describe an application with rich functional requirements. JavaScript can be used to create programming logic that extends either the Web server or the HTML pages on the client with application logic. When used on the Web server, JavaScript interacts with the data on the Web page and other sources (perhaps data from a database) and generates HTML statements dynamically. Web pages built in this way are interpreted in the standard fashion by a Web browser. On the client, JavaScript interacts with the browser directly and instructs the Web browser to perform functions beyond those available in standard HTML.

Java Server Pages (JSP)

Part of the Java 2 Enterprise Edition basket is a technology called Java server pages (JSPs). This is a mechanism to extend a Web server with JavaScript and/or components written in Java (servlets). This product is meant to compete directly with Microsoft's Active Server Pages (ASP) technology, as a cross-platform development environment for dynamic Web pages that can operate with any Web server.

Using JSP, a developer can build Web pages at runtime based on data from a Java component, static HTML data, or a JavaScript. The result of a JSP is often standard HTML, which is sent to the client by the Web server.

JDBC

The standard Java database access technique is JDBC. JDBC is essentially a variation on ODBC with some Java bindings. It is designed as a Java API to SQL data. JDBC is just a specification; it is up to the various database vendors or third parties to create JDBC drivers for a particular database. In some cases, a JDBC-ODBC bridge can be used, but likely not for production purposes.

JDBC provides a fairly complete set of SQL-friendly data access mechanisms, such as scrollable result sets, absolute and relative positioning (in the result set), access to stored procedures, and data type conversions. Most SQL-92 (Entry Level 2) statements are supported in JDBC.

Microsoft Foundation Classes (MFC)

Low-level programming in Microsoft Windows can be a very complex and tedious task. Windows comes with hundreds of APIs, most of which are designed to display and manage the GUI. Each new release of Windows comes with a new batch of APIs that may or may not be useful to your project but will certainly clutter up the API reference manual.

Microsoft has attempted to simplify this complexity with its Microsoft Foundation Class (MFC) libraries. The set of libraries is designed to let the C++ programmer easily access the most commonly used UI components, without sacrificing power or flexibility. Furthermore, the MFC programmer can create specializations of the base MFC classes for his or her own purposes. Most sophisticated Windows applications today use MFC classes and the MFC programming environment.

Open Database Connectivity (ODBC)

ODBC is a standard data access technique developed by the SQL Access Group in the early 1990s. This is a specification for a standard call level interface that supports access to data sources. Various vendors, including all the major database vendors, have developed ODBC drivers that can be used to access information stored in their databases. ODBC drivers are now shipped as a standard component of Windows systems.

Although used primarily with a SQL database, ODBC is sometimes used to access information from other sources, including text files, spreadsheets, and ISAM files. ODBC drivers also are available on other platforms besides Windows, such as UNIX and Macintosh.

Remote Method Invocation (RMI)

Sun has defined a standard technique that Java applications can use to create objects on remote systems, and then call their methods. This standard, called RMI, is similar

in principle to traditional RPC function calls (see the "Remote Procedure Call" section), except that it is tailored for Java. Most Java runtimes (but not all) contain the libraries necessary to support RMI. RMI is an essential building block for EJBs.

Remote Procedure Call (RPC)

Remote procedure call (RPC) is a generic name for a protocol that allows an application on one system to call a function on another system. The best way to understand RPC is to start with a normal procedure call. Suppose that program A calls program B with parameters and then waits for the reply. When program B is complete, it returns to program A, perhaps with return parameters. In an RPC environment, program B can be on another system.

In most RPC implementations, the caller must perform a small amount of specialized logic to initiate the called program on the remote system before calling the remote program. After initialization, an RPC implementation makes the remote program appear as if it were local, except for the performance penalty incurred by remote function calls as compared to a local call.

Secure Sockets Layer (SSL)

SSL is a transmission protocol designed by Netscape that employs encryption techniques to provide secure transmissions over the Internet. Other service-based utilities, such as HTTP and FTP, can employ SSL to secure the messages they send over the Internet.

SSL sits between these service utilities and the Internet's basic message-passing protocol (TCP/IP). For example, an HTTP server that uses SSL (such as https://www.domain…) talks to the SSL layer, which encrypts the message and then sends it out over TCP/IP. On the other end of the message pipe, the browser reads the message from the SSL, which gets the encrypted message from TCP/IP, decrypts it, and returns it to the browser. SSL hides the message content sent between partner applications from public scrutiny.

Swing

The basic GUI functionality for Java is contained in the abstract window toolkit (AWT). However, the AWT has been generally regarded as the weakest part of the original Java environment. Sun has improved on AWT with the newer Java Foundation Classes (JFCs), originally released as a patch to SDK 1.1. These APIs are grouped in a package named after the internal development project at Sun (Swing). The Swing package is meant to replace the AWT.

Applications that use Swing have access to more control over the user interface. For example, an application can now manage borders around Swing components, and Swing components do not need to be rectangular. Swing applications can interact better with native applications.

On the downside, the Swing components are fairly large. A Swing application must download the Swing components to the client if they are not already loaded.

TCP/IP

The standard transport protocol for the Internet is transmission control protocol/Internet protocol or TCP/IP. This low-level protocol defines the way messages must be packaged, delivered, and accessed by systems on a network. Other service-based utilities, such as HTTP and FTP, employ TCP/IP as the wire-level protocol to perform their functions.

TCP/IP's most popular feature is the ability to create a virtual *pipe* between a client and a server application. The client and server can use this pipe to send messages to each other, simply by reading and writing to the pipe, as if it were a file. These messages are tagged as belonging to a specific client and server application. Other client applications and other server applications will not see these messages, even if they are on the same machine. Web browsers and Web servers use this capability when they send and process HTTP requests, for example.

By virtue of these and other features, TCP/IP facilitates the transfer of information between applications over a public network, such as the Internet.

Unified Modeling Language (UML)

The unified modeling language is a definition of a software modeling language. Software modeling techniques employ rigorous analysis and documentation methodologies to capture a representation of how a system is built and how the system performs its processes. UML combines features from existing software-modeling practices. At the same time, UML is intended to be simpler, or more approachable, than some of the previous modeling techniques.

The UML is a rich language that can represent all sorts of system processes, from simple data entry functions to complex real-time distributed systems.

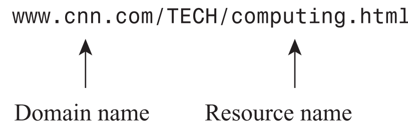
Uniform Resource Locator (URL)

URL is a standard convention that identifies a unique location for a file, or other resource, in a manner that is unambiguous. Any application, such as a browser, can use a URL name to locate a file. You are likely familiar with URLs already. The name of your favorite Web site is identified with a URL (www.cnn.com, for example).

The first part of a URL is the protocol used (e.g., HTTP or FTP). The second part of a URL identifies the *domain name* that contains the resource. A domain name is a publicly registered ID of a server on the Internet. Domain names are assigned by government-affiliated organizations to an organization or individual. Domain name servers on the Internet resolve individual domain names to a specific identifier called an IP address. The format of an IP address is nnn.nnn.nnn.nnn, in which each n is replaced with a number.

A domain is either a unique physical server or a logical location on a server. It can even be a collection of servers (a server farm), grouped to appear as one.

The remainder of the URL name (that is, the characters after the domain name) point to a file, a directory, or some other resource located on the server.



VBScript

Microsoft has developed its own Web scripting language called VBScript. VBScript is similar in many ways to JavaScript. Like JavaScript, VBScript provides an environment that allows the developer to perform programming logic dynamically (at runtime). VBScript can be executed on either a Web client or a Web server. As the name suggests, VBScript is similar in syntax to Visual Basic (VB). VBScript and JavaScript can usually be combined on the same Web page.

Web Services

A Web service is simply an application that can be addressed as a URL and programmatically returns information to clients who want to use it. Clients don't need to know how a service is implemented because it is component based.

Like components, Web services represent black-box functionality that can be reused without worrying about how the service is implemented. Web services provide contracts that describe the services provided. Developers can assemble applications using a combination of remote services, local services, and custom code.

Appendix D
Sun Microsystems, Inc. Binary Code License Agreement

for the JAVA SE DEVELOPMENT KIT (JDK), VERSION 6

SUN MICROSYSTEMS, INC. ("SUN") IS WILLING TO LICENSE THE SOFTWARE IDENTIFIED BELOW TO YOU ONLY UPON THE CONDITION THAT YOU ACCEPT ALL OF THE TERMS CONTAINED IN THIS BINARY CODE LICENSE AGREEMENT AND SUPPLEMENTAL LICENSE TERMS (COLLECTIVELY "AGREEMENT"). PLEASE READ THE AGREEMENT CAREFULLY. BY DOWNLOADING OR INSTALLING THIS SOFTWARE, YOU ACCEPT THE TERMS OF THE AGREEMENT. INDICATE ACCEPTANCE BY SELECTING THE "ACCEPT" BUTTON AT THE BOTTOM OF THE AGREEMENT. IF YOU ARE NOT WILLING TO BE BOUND BY ALL THE TERMS, SELECT THE "DECLINE" BUTTON AT THE BOTTOM OF THE AGREEMENT AND THE DOWNLOAD OR INSTALL PROCESS WILL NOT CONTINUE.

1. DEFINITIONS. "Software" means the identified above in binary form, any other machine readable materials (including, but not limited to, libraries, source files, header files, and data files), any updates or error corrections provided by Sun, and any user manuals, programming guides and other documentation provided to you by Sun under this Agreement. "General Purpose Desktop Computers and Servers" means computers, including desktop, laptop and tablet computers, or servers, used for general computing functions under end user control (such as but not specifically limited to email, general purpose Internet browsing, and office suite productivity tools). The use of Software in systems and solutions that provide dedicated functionality (other than as mentioned above) or designed for use in embedded or function-specific software applications, for example but not limited to: Software embedded in or bundled with industrial control systems,

wireless mobile telephones, wireless handheld devices, kiosks, TV/STB, Bluray Disc devices, telematics and network control switching equipment, printers and storage management systems, and other related systems are excluded from this definition and not licensed under this Agreement. "Programs" means Java technology applets and applications intended to run on the Java Platform Standard Edition (Java SE) platform on Java-enabled General Purpose Desktop Computers and Servers.

2. LICENSE TO USE. Subject to the terms and conditions of this Agreement, including, but not limited to the Java Technology Restrictions of the Supplemental License Terms, Sun grants you a non-exclusive, non-transferable, limited license without license fees to reproduce and use internally Software complete and unmodified for the sole purpose of running Programs. Additional licenses for developers and/or publishers are granted in the Supplemental License Terms.
3. RESTRICTIONS. Software is confidential and copyrighted. Title to Software and all associated intellectual property rights is retained by Sun and/or its licensors. Unless enforcement is prohibited by applicable law, you may not modify, decompile, or reverse engineer Software. You acknowledge that Licensed Software is not designed or intended for use in the design, construction, operation or maintenance of any nuclear facility. Sun Microsystems, Inc. disclaims any express or implied warranty of fitness for such uses. No right, title or interest in or to any trademark, service mark, logo or trade name of Sun or its licensors is granted under this Agreement. Additional restrictions for developers and/or publishers licenses are set forth in the Supplemental License Terms.
4. LIMITED WARRANTY. Sun warrants to you that for a period of ninety (90) days from the date of purchase, as evidenced by a copy of the receipt, the media on which Software is furnished (if any) will be free of defects in materials and workmanship under normal use. Except for the foregoing, Software is provided "AS IS". Your exclusive remedy and Sun's entire liability under this limited warranty will be at Sun's option to replace Software media or refund the fee paid for Software. Any implied warranties on the Software are limited to 90 days. Some states do not allow limitations on duration of an implied warranty, so the above may not apply to you. This limited warranty gives you specific legal rights. You may have others, which vary from state to state.
5. DISCLAIMER OF WARRANTY. UNLESS SPECIFIED IN THIS AGREEMENT, ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR

NON-INFRINGEMENT ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT THESE DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

6. LIMITATION OF LIABILITY. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO THE USE OF OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. In no event will Sun's liability to you, whether in contract, tort (including negligence), or otherwise, exceed the amount paid by you for Software under this Agreement. The foregoing limitations will apply even if the above stated warranty fails of its essential purpose. Some states do not allow the exclusion of incidental or consequential damages, so some of the terms above may not be applicable to you.
7. TERMINATION. This Agreement is effective until terminated. You may terminate this Agreement at any time by destroying all copies of Software. This Agreement will terminate immediately without notice from Sun if you fail to comply with any provision of this Agreement. Either party may terminate this Agreement immediately should any Software become, or in either party's opinion be likely to become, the subject of a claim of infringement of any intellectual property right. Upon Termination, you must destroy all copies of Software.
8. EXPORT REGULATIONS. All Software and technical data delivered under this Agreement are subject to US export control laws and may be subject to export or import regulations in other countries. You agree to comply strictly with all such laws and regulations and acknowledge that you have the responsibility to obtain such licenses to export, re-export, or import as may be required after delivery to you.
9. TRADEMARKS AND LOGOS. You acknowledge and agree as between you and Sun that Sun owns the SUN, SOLARIS, JAVA, JINI, FORTE, and iPLANET trademarks and all SUN, SOLARIS, JAVA, JINI, FORTE, and iPLANET-related trademarks, service marks, logos and other brand designations ("Sun Marks"), and you agree to comply with the Sun Trademark and Logo Usage Requirements currently located at http://www.sun.com/policies/trademarks. Any use you make of the Sun Marks inures to Sun's benefit.
10. U.S. GOVERNMENT RESTRICTED RIGHTS. If Software is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in Software and accompanying documentation will be only as set

forth in this Agreement; this is in accordance with 48 CFR 227.7201 through 227.7202-4 (for Department of Defense (DOD) acquisitions) and with 48 CFR 2.101 and 12.212 (for non-DOD acquisitions).

11. GOVERNING LAW. Any action related to this Agreement will be governed by California law and controlling U.S. federal law. No choice of law rules of any jurisdiction will apply.
12. SEVERABILITY. If any provision of this Agreement is held to be unenforceable, this Agreement will remain in effect with the provision omitted, unless omission would frustrate the intent of the parties, in which case this Agreement will immediately terminate.
13. INTEGRATION. This Agreement is the entire agreement between you and Sun relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification of this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

SUPPLEMENTAL LICENSE TERMS

These Supplemental License Terms add to or modify the terms of the Binary Code License Agreement. Capitalized terms not defined in these Supplemental Terms shall have the same meanings ascribed to them in the Binary Code License Agreement . These Supplemental Terms shall supersede any inconsistent or conflicting terms in the Binary Code License Agreement, or in any license contained within the Software.

A. Software Internal Use and Development License Grant. Subject to the terms and conditions of this Agreement and restrictions and exceptions set forth in the Software "README" file incorporated herein by reference, including, but not limited to the Java Technology Restrictions of these Supplemental Terms, Sun grants you a non-exclusive, non-transferable, limited license without fees to reproduce internally and use internally the Software complete and unmodified for the purpose of designing, developing, and testing your Programs.

B. License to Distribute Software. Subject to the terms and conditions of this Agreement and restrictions and exceptions set forth in the Software README file, including, but not limited to the Java Technology Restrictions of these Supplemental Terms, Sun grants you a non-exclusive, non-transferable, limited license without fees to reproduce and distribute the Software, provided that (i) you distribute the Software complete and unmodified and only bundled as part of, and for the sole purpose of running, your Programs, (ii) the Programs add significant and primary functionality to the Software, (iii) you do not distribute additional software intended to replace any component(s) of the Software, (iv) you do not remove or alter any proprietary legends or notices contained in the Software, (v) you only distribute the Software subject to a license agreement that protects Sun's interests consistent with the terms contained in this Agreement, and (vi) you agree to defend and indemnify Sun and its licensors from and against any damages, costs, liabilities, settlement amounts and/or expenses (including attorneys' fees) incurred in connection with any claim, lawsuit or action by any third party that arises or results from the use or distribution of any and all Programs and/or Software.

C. License to Distribute Redistributables. Subject to the terms and conditions of this Agreement and restrictions and exceptions set forth in the Software README file, including but not limited to the Java Technology Restrictions of these Supplemental Terms, Sun grants you a non-exclusive, non-transferable, limited license without fees to reproduce and distribute those files specifically identified as redistributable in the Software "README" file ("Redistributables") provided that: (i) you distribute the Redistributables complete and unmodified, and only bundled as part of Programs, (ii) the Programs add significant and primary functionality to the Redistributables, (iii) you do not distribute additional software intended to supersede any component(s) of the Redistributables (unless otherwise specified in the applicable README file), (iv) you do not remove or alter any proprietary legends or notices contained in or on the Redistributables, (v) you only distribute the Redistributables pursuant to a license agreement that protects Sun's interests consistent with the terms contained in the Agreement, (vi) you agree to defend and indemnify Sun and its licensors from and against any damages, costs, liabilities, settlement amounts and/or expenses (including attorneys' fees) incurred in connection with any claim, lawsuit or action by any third party that arises or results from the use or distribution of any and all Programs and/or Software.

D. Java Technology Restrictions. You may not create, modify, or change the behavior of, or authorize your licensees to create, modify, or change the behavior of, classes, interfaces, or subpackages that are in any way identified as "java", "javax", "sun" or similar convention as specified by Sun in any naming convention designation.

E. Distribution by Publishers. This section pertains to your distribution of the Software with your printed book or magazine (as those terms are commonly used in the industry) relating to Java technology ("Publication"). Subject to and conditioned upon your compliance with the restrictions and obligations contained in the Agreement, in addition to the license granted in Paragraph 1 above, Sun hereby grants to you a non-exclusive, nontransferable limited right to reproduce complete and unmodified copies of the Software on electronic media (the "Media") for the sole purpose of inclusion and distribution with your Publication(s), subject to the following terms: (i) You may not distribute the Software on a stand-alone basis; it must be distributed with your Publication(s); (ii) You are responsible for downloading the Software from the applicable Sun web site; (iii) You must refer to the Software as JavaTM SE Development Kit 6; (iv) The Software must be reproduced in its entirety and without any modification whatsoever (including, without limitation, the Binary Code License and Supplemental License Terms accompanying the Software and proprietary rights notices contained in the Software); (v) The Media label shall include the following information: Copyright 2006, Sun Microsystems, Inc. All rights reserved. Use is subject to license terms. Sun, Sun Microsystems, the Sun logo, Solaris, Java, the Java Coffee Cup logo, J2SE, and all trademarks and logos based on Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. This information must be placed on the Media label in such a manner as to only apply to the Sun Software; (vi) You must clearly identify the Software as Sun's product on the Media holder or Media label, and you may not state or imply that Sun is responsible for any third-party software contained on the Media; (vii) You may not include any third party software on the Media which is intended to be a replacement or substitute for the Software; (viii) You shall indemnify Sun for all damages arising from your failure to comply with the requirements of this Agreement. In addition, you shall defend, at your expense, any and all claims brought against Sun by third parties, and shall pay all damages awarded by a court of competent jurisdiction, or such settlement amount negotiated by you, arising out of or in connection with your use, reproduction or distribution of the Software and/or the Publication. Your obligation to provide indemnification under this section shall arise provided that Sun: (a) provides you prompt notice of the claim; (b) gives

you sole control of the defense and settlement of the claim; (c) provides you, at your expense, with all available information, assistance and authority to defend; and (d) has not compromised or settled such claim without your prior written consent; and (ix) You shall provide Sun with a written notice for each Publication; such notice shall include the following information: (1) title of Publication, (2) author(s), (3) date of Publication, and (4) ISBN or ISSN numbers. Such notice shall be sent to Sun Microsystems, Inc., 4150 Network Circle, M/S USCA12-110, Santa Clara, California 95054, U.S.A , Attention: Contracts Administration.

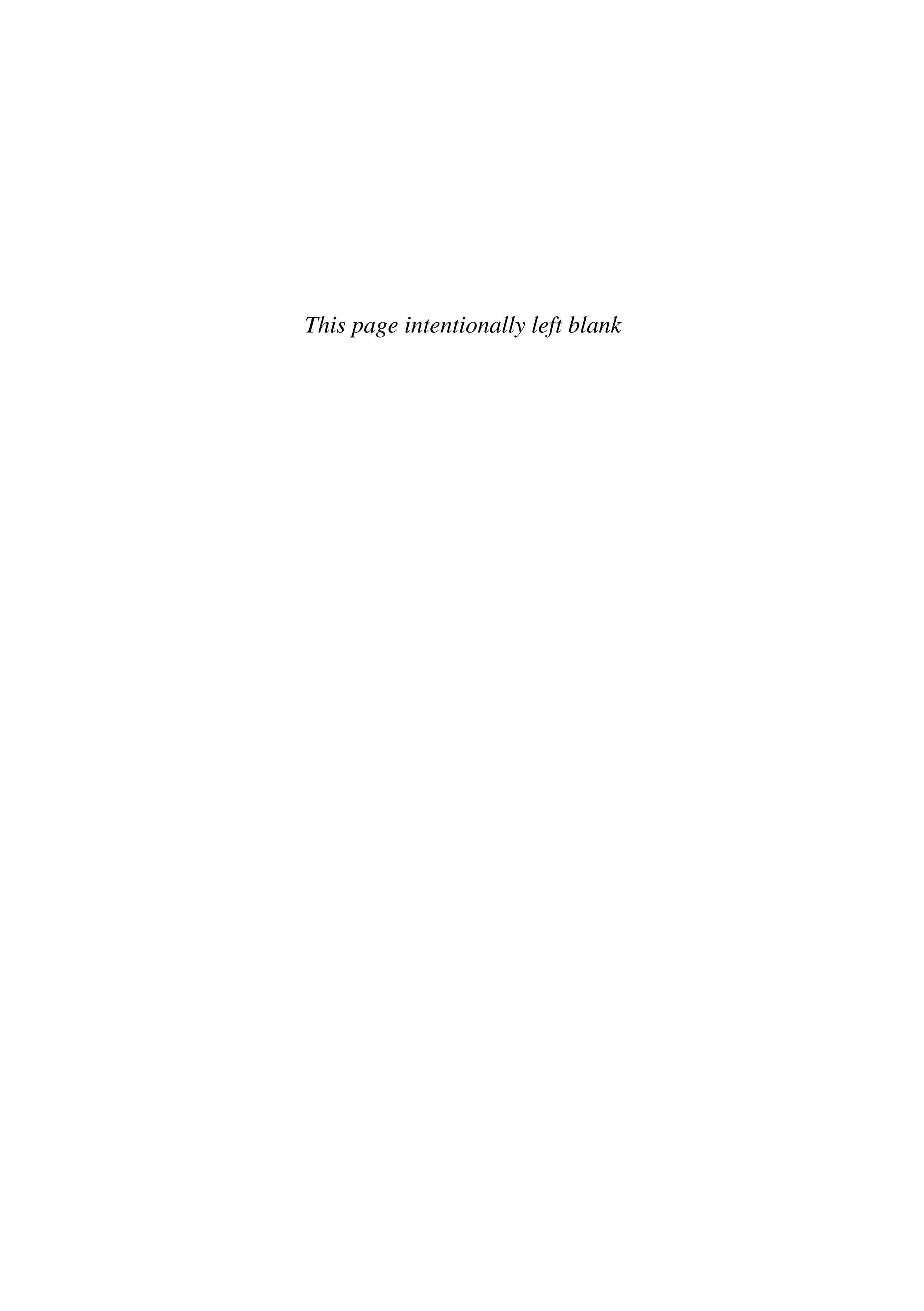
F. Source Code. Software may contain source code that, unless expressly licensed for other purposes, is provided solely for reference purposes pursuant to the terms of this Agreement. Source code may not be redistributed unless expressly provided for in this Agreement.

G. Third Party Code. Additional copyright notices and license terms applicable to portions of the Software are set forth in the THIRDPARTYLICENSEREADME.txt file. In addition to any terms and conditions of any third party opensource/freeware license identified in the THIRDPARTYLICENSEREADME.txt file, the disclaimer of warranty and limitation of liability provisions in paragraphs 5 and 6 of the Binary Code License Agreement shall apply to all Software in this distribution.

H. Termination for Infringement. Either party may terminate this Agreement immediately should any Software become, or in either party's opinion be likely to become, the subject of a claim of infringement of any intellectual property right.

I. Installation and Auto-Update. The Software's installation and auto-update processes transmit a limited amount of data to Sun (or its service provider) about those specific processes to help Sun understand and optimize them. Sun does not associate the data with personally identifiable information. You can find more information about the data Sun collects at http://java.com/data/.

For inquiries please contact: Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A.


This page intentionally left blank

Index

SYMBOLS

*** (asterisk) wild card,** 31
// (two forward slashes), 153
[ ] (braces), 337
_ (underscore character), 150
{ } (curly braces), 167
+ (concatenate) operator, 192, 193
++ (increment) operator, Java, 155
++x (prefix) operator, Java, 155
<applet> tag, 25
- (dash) character, 150, 155
. (period), 150
; (semicolon), 150–152

A

Abstract Window Toolkit (AWT), 20, 279–280
AbstractSequentialList, 272
acquire( ) method, threads, 234
acquireUninterruptibly( ) method, threads, 234
ACTION-SWITCH, COBOL
  adding multiple message types with, 38–39
  implementing inheritance, 113–118
  Java objects in COBOL, 73–76
  method members vs., 87
  private variables similar to, 14
  returning value from private variables, 68
ActiveX specification, 322
add( ) method, ArrayLists, 267
addElement( ) method, Vectors, 267
AMOUNT element, OAG standard, 345
applet viewer (appletviewer.exe)
  executing Java's SDK, 21
  running applets in, 25–26
  testing applets with, 35
<applet> tag, 25
applets
  applications vs., 34
  CODEBASE references, 28
  creating with Java's SDK, 24–26
  defined, 19
  HelloWorld, 62–63
  portability problems of, 19–20
  summary of, 35
  using AWT for GUI interaction, 279–280
applications
  applets vs., 34
  HelloWorld, 61–62
  using AWT for GUI interaction, 279–280
arithmetic operations, Java, 154–159
  exercises, 164–165
  floating-point operations, 154–155
  increment and decrement operators, 155
  integer operations, 154–155
  list of, 156
  object reference variables, 157–159
  postfix and prefix operators, 155
  shortcut coding style, 155
arraycopy( ) function, 85
arraycopy( ) method, 85
ArrayIndexOutOfBoundsException exception, 84
ArrayList collection class
  exercises, 275–278
  overview of, 266–270
  Vectors vs., 264
arrays
  exercises, 98–99
  Java syntax, 173–174
  overview of, 83–87
  as parameters, 87
ASCII character set, 254

ASSIGN statement, 244
assignment function, Java vs. COBOL, 149
asterisk (*) wild card, import statement, 31
attributes, DTD, 335–336
authoring, XML documents, 339
autoboxing, 199, 269
auto-commit, Connection class, 297
AUTOEXEC.BAT file, 22
AWT (Abstract Window Toolkit), 20, 279–280

B

base classes. *see* also superclasses (parent classes)
- bottom-up design, 102
- hiding variables and methods, 126–127
- Java inheritance concepts, 121–126
- multiple inheritance and, 121
- overview of, 102
- using inheritance as design pattern, 109

Basic Multilingual Plane (BMP), 190
begin tag...end tag construct, XML, 332
BigDecimals, 202–204, 214–215, 217
BigIntegers, 202–204, 214–215, 217
BigNumbers, 202–208, 214–216, 217
binary arithmetic operators, Java, 154–159
binarySearch( ) function, collections, 274
bitwise operations, Java, 156
black box, COBOL, 112
BMP (Basic Multilingual Plane), 190
boolean expressions, 169–170
bottom-up design, inheritance, 102
braces ({ }), 167
brackets ([ ]), 337
break statement
- defined, 181
- exiting switch loop with, 177
- flow control exercises, 184–186
- overview of, 177–180

breakpoint, Eclipse debugger, 362–363
browsers
- supporting applets, 35
- Web servers and, 308–310

C

C comments, Java syntax, 152–153
C++ comments, Java syntax, 152–153
cab files, 33
CALL COUNTER, COBOL, 66–68
CALL SUBROUTINE USING, COBOL, 4–9
callbacks, SAX parser, 340
CALLER COBOL
- calling subroutine, 4–9
- implementing inheritance, 114–115
- message passing, 37, 39–40
- method overloading, 46–48

calling program
- accessing public data members, 16
- Caller class, 12–13
- calling subroutine, 4–9
- causing runtime errors, 47
- COBOL subroutines and, 4
- containing its objects, 14
- creating new object prior to using it, 71
- error processing capabilities, 196, 219–220, 225
- inheritance in COBOL, 113, 119
- method overloading in COBOL, 44–45
- and reference variables, 71, 73

cardinality rules, DTD elements, 334–335
case sensitivity
- identifiers, 151
- Java compilers, 150

case statements
- EVALUATE verb vs., 187
- flow control exercise, 184
- overview of, 176–177

catalog content management, XML, 352
CICS, 314
class libraries, building GUIs with, 20, 279–281, 286–289
class members, 65–100
- arrays, 83–87
- classes, objects and members, 71–72
- constructors, 89–91
- data members, 77–79
- exercises, 91–99
- local variables, 79–80
- method members, 87–89
- MYSUB COBOL, 66–68
- objects and COBOL, 72–76
- overview of, 65–66
- primitive data types, 80–83
- reviewing samples, 99–100
- using objects in Java, 76–77
- variables, 68–70

class methods, 87

class variables
defined, 77
exercises, 96–98
overview of, 70
classes
from COBOL perspective, 9
data members, 7, 10
defined in Java, 71–72
exercises in, 49–57
grouping into packages. *see* packages
inheritance hierarchy, 102–103
Java interfaces vs., 129
from Java perspective, 13, 15
CLASSPATH directory
adding jar file to, 33, 283
exercises, 55
overview of, 27–28
client
accessing EJB from, 327
using online XML, 343
CLOSE file, 244
COBOL
collections vs. files in, 264–265
error management, 220
Java BigDecimals vs., 204–205
Java flow control operators vs., 181
Java I/O vs., 244–245
Java operators vs., 153–154
Java Strings vs., 189–190, 194–196
Java syntax vs., 149–150
objects and, 109–112
statements, 150
understanding reference variables, 154–159
COBOL subroutines. *see also* MYSUB COBOL
behaving as object, 72–76
calling, 4–9
Java class vs., 13, 15
objects vs., 5
overview of, 4
terms to review, 9–10
code block, 168–172
CODEBASE references, 28
collections, 263–278
AbstractSequentialList, 272
background, 264–266
defining, 263
exercises, 275–277
HashSet, 271
iterators, 272–273
keyed, Hashtable and HashMap, 270–271
LinkedList, 272
managing arguments and returning values for methods, 84
ordered, Vector and ArrayList, 266–270
ordering and comparison functions, 273–275
reviewing exercises, 277–278
TreeSet, 272
commands, executable, 21
COMMAREA, CICS, 314
comments
flow control exercises, 182
Javadoc, 152–153, 284–286
Comparator interface, collections, 274–275
compareTo( ) method, 216, 274–275
comparison functions, collections, 273–275
compile process, Java, 18–19
compressed packages, 33
concatenate operator (+), 192, 193
ConcurrentHashMap collection class, 270
Connection class, 297–298
constructors
BigDecimal and BigInteger, 203–204
Hashtable and HashMap, 270–271
from Java perspective, 14
overview of, 89–91
Vectors and ArrayLists, 267
consumer (calling) class
extending methods, 107–108
inheritance and, 122–125
method overloading, 43
redefining methods, 106
sending messages to objects, 41
treating super-type objects, 109
using derived classes, 122
containers, EJB, 323–325
continue statement
defined, 181
flow control exercise, 184
overview of, 177–180
CONTROL-AREA, COBOL, 40, 47
copy( ) function, collections, 273
COPY statement, COBOL, 31
CounterServlet.java file, 317–320
currentThread( ) method, 238–239, 241

D

dash (-) character, 150, 155
data encapsulation, COBOL, 65
data members, Java, 14, 77–79
data repositories, XML, 351
data types, primitive, 80–83
databases. *see* JDBC (Java Database Connectivity)
DATETIME element, OAG standard, 345–347
debugging, with Eclipse, 362–363
DecimalFormat class, 197–198, 261
declarations, XML, 337–338
deletes, database, 300–301
derived classes
  clarifying variable for this operator, 127–128
  defined, 102
  extending inherited methods, 106–108
  hiding methods and members, 131
  hiding variables and methods, 126–127
  inheritance and, 103, 121–126
  inheriting methods, 103–104
design patterns
  COBOL object-oriented, 112–119
  inheritance and, 108–109
***Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma, Helm, Johnson, and Vlissides),** 108–109
directories
  creating for applets, 24
  creating for Java files, 22
  File class functions, 248–249
  Java packages related to, 32–33
distributed computing, EJBs, 322–323
DNS (domain name system), 32, 308
Document Object Model (DOM) interface, 340
document type definitions. *see* DTDs (document type definitions), XML
doGet( ) method, HttpServlet, 312, 318
DOM (Document Object Model) interface, 340
domain name system (DNS), 32, 308
doskey command, 21
Double numeric class wrapper, 197–198
doubleValue( ) method, 197–198
do...while statement, 174–175, 181
drawstring( ) method, Graphics class
  adding to applet, 26, 55–56, 59
  overview of, 36
DriverManager, 296
drivers, JDBC
  connecting to database, 296–298
  defined, 294
  types of, 294–295
DTDs (document type definitions), XML
  attributes, 335–336
  complete documents, 338–339
  documents and, 332–333
  elements, 334
  entities, 336–337
  OAG specifications for, 344–351

E

Eclipse, 355–367
  debugging with, 362–363
  getting started with, 356–358
  installing, 356
  making new project, 358–361
  overview of, 355
  refactoring with, 363–367
  runtime, 362
EDI (Electronic Data Interchange), XML and, 342–343
EJBs (Enterprise JavaBeans), 321–328
  accessing from client, 327
  container services, 324–325
  distributed computing and, 322–323
  exercises, 327–328
  interfaces and implementation class, 325–327
  types of, 323–324
Electronic Data Interchange (EDI), XML and, 342–343
elements
  adding to Hashtable and HashMap, 271
  adding to Vectors or ArrayLists, 267–268
  DTD, 334–335
  XML nested, 332
encapsulation, 76
Enterprise Java
  Eclipse. *see* Eclipse
  Enterprise JavaBeans. *see* EJBs (Enterprise JavaBeans)
  JDBC. *see* JDBC (Java Database Connectivity)
  XML. *see* XML (Extensible Markup Language)
enterprise resource planning (ERP), 343
entities, DTD, 336–337
EntityBeans, EJBs, 323–324, 326–327

equals( ) method, 191–192, 216
ERP (enterprise resource planning), 343
Error class, 221
ErrorMsg class
  arrays, 83–87
  constructors, 90–91
  defining methods in, 40–41
  exercises, 49–61
  exercises, class members, 91–94, 96–97, 99
  exercises, polymorphism, 142–143, 145
  exercises, variables, 133–138, 140, 144
  Hashtable and HashMap, 271
  HelloWorld applet, 62–63
  HelloWorld application, 61–62
  hiding variables and methods, 127
  inheritance, 103–108
  inheritance and Java, 120–126
  iterators, 272–273
  Java data members, 78–79
  Java variables, 69–71
  method members, 88–89
  method overloading, 43, 47–48
  multiple messages, 41–43
  objects and Java, 11–13
  overview of, 63–64
  packages, 29–32
  refactoring in Eclipse, 366–367
  reviewing samples, 57–61
  the `this` variable, 127–128
  Vector and ArrayList, 267–268
errors
  exception handling. *see* exceptions
  refactoring with Eclipse, 363–367
  EVALUATE verb, COBOL, 177, 18**1, 187**
Exception class, 221–222
exception processing, 196
exceptions, 219–229
  creating, 221–223
  Exception class hierarchy, 221
  exercises, 237–241
  overview of, 219–220
  processing suggestions, 228–229
  using, 223–227
EXCHANGE-AMOUNT, COBOL, 204–205
executable code, 18
executeUpdate( ) method, 301
execution process, Java, 18–19
exists( ) method, File class, 248
extends keyword, Java inheritance, 119–121
Extensible Business Reporting Language (XBRL), 352
Extensible Markup Language. *see* XML
external entities, DTD, 336

F

factory class, 297
File class, 246–249, 258–261
FILE SECTION, DATA DIVISION, 244
FileInputStream class, 250
filenames
  packages and, 32–33
  source, 26–27, 33
FileReader class, 255
File.separator variable, 246
fill( ) function, collections, 273
Final keyword, 78
finalize( ) methods, garbage collection, 237
finally statement, 224–226, 229
floating-point operations, Java, 154–155
flow control, Java, 167–187
  break statement, 177–180
  code block, 168
  continue statement, 177–180
  do...while statement, 174–175
  exercises, 181–186
  if statement, 168–172
  list of operators, 181
  review, 186–187
  for statement, 175–176
  switch statement, 176–177
  while statement, 172–174
for loop, collections, 273
for statement
  defined, 181
  flow control exercises, 183–185
  overview of, 175–176
forName( ) method, 296
fully qualified names, 30

G

garbage collection, 236–237
Generics, 266–267
get( ) methods, File class, 247–248
GET requests, 310, 312
"GET$NARGS" function, 44–45
getConnection( ) method, database, 296, 302
getCounter( ) method, 100

getErrorMsg( ) method
adding to HelloWorld, 52–54
synchronization, 234
using overloaded version of, 56–58, 95
getFilePointer( ) method, RandomAccessFile, 257
getMessage( ) method, Exception class, 222
graphical user interfaces. *see* GUIs (Graphical user interfaces)
Graphics class, 36
GUIs (graphical user interfaces)
building with class libraries, 20
development of, 279–281
exercise in creating, 286–289

H

handle, reference pointer, 71
HashMap collection class
exercises, 275–278
Hashtable vs., 264
overview of, 270–271
HashSet, 271
Hashtable collection class, 264, 270–271
HelloWorld applet
exercises in, 53–56
overview of, 62–63
review, 59–60
HelloWorld application
ErrorMsg class in, 57–58
overview of, 61–62
HelloWorld program, 22–24, 57
HelloWorld.html, 25
HelloWorld.java (example exercises)
applets, 24
BigNumbers, 214–215
class members, 92–93
flow control, 181–186
I/O in Java, 258–261
Java syntax, 160–165
messages, 49–57
numeric wrapper classes, 210–213
string methods, 210
threads, 239–240
hiding methods
and members, 131
and variables, 126–127
home interface, EJB, 325–326
home page, Eclipse, 357
hostname, 308
HTML (Hypertext Markup Language)
executing applets, 25
Web servers and, 308–310
XML vs., 330–332, 340–341
HTTP (Hypertext Transfer Protocol)
compressed packages and, 33
Web servers using, 308–310
HttpServlet, 312, 318
Hypertext Markup Language. *see* HTML

I

IDE (integrated development environment), 20, 355
identifiers, Java, 150–152
if statement
defined, 181
flow control exercises, 181–182
overview of, 168–172
immutable strings, 192
implementation class, EJB, 325–327
implements keyword, 129–130
import statements, 30–31
importing, Java files to Eclipse project, 358–360
IMS, 314
increment operator (++)**, 155**
indexed sequential (**ISAM**) **file type,** 270
information transfer systems, XML, 352
inheritance, 101–126
clarifying variable for this operator, 127–128
class definitions, 121–122
COBOL object-oriented design patterns, 112–119
COBOL perspective on, 109–112
as design pattern, 108–109
Exception class and, 221
exercises, 133–144
extending class methods, 123–124
extending methods, 106–108
with extends keyword, 119–121
hiding methods and members, 131
hiding variables and methods, 126–127
inheriting methods, 103–106
methods, 123–126
object-oriented design and, 101–102
objects and, 103
polymorphism and, 132
reviewing samples, 144–145
sharing variables and methods, 126
string methods, 191–192

from threads, 230–232
using protected keyword, 122–123
initialization
data variable, 69–70
servlet, 312–313
inner code blocks, 171–172
inputfile property, properties file, 282
INPUT-OUTPUT SECTION, ENVIRONMENT DIVISION, 244
InputStream class, 249–251, 259–260
inserts, database, 300–301
inside packages, 29–31
installing, Eclipse, 356
integer operations, Java, 154–155, 269
integrated development environment (IDE), 20, 355
intelligent agents, XML, 352
interfaces
from COBOL perspective, 9
EJB, 325–327
Java, 128–131
Java Collection framework, 265–266
internal entities, DTD, 336
I/O in Java, 243–262
exercises, 258–262
File class, 246–249
InputStream and OutputStream classes, 249–251
overview of, 243
RandomAccessFile class, 256–258
Reader and Writer classes, 254–256
serialization, 252–254
streams vs. record-based I/O, 244–246
IOException class
creating Exception objects with, 221
defined, 221
handling exceptions, 228
naming conventions, 221–222
ISAM (indexed sequential) file type, 270
isDirectory( ) method, File class, 248
isFile( ) method, File class, 248
isolation level, EJBs, 324
iteration, 175–176
iterators, 272–273

J

jar files
compressed package technology, 33
containing servlet classes, 319
creating, 282–284
Java. *see also* Enterprise Java
applets. *see* applets
BigNumbers. *see* BigNumbers
collections. *see* collections
data members, 77–79
exceptions. *see* exceptions
flow control. *see* flow control, Java
GUI development, 279–281
inheritance and, 119–126
interfaces, 128–131
I/O. *see* I/O in Java
numbers. *see* numbers
properties files, 281–282
StringBuffers. *see* StringBuffers
strings. *see* strings
threads. *see* threads
utilities, 282–286
variables, 68–70
XML and, 339–340
Java API For XML Processing (JAXP), 339–341
Java Archive utility, 282–284
Java byte codes, 18
Java compiler (javac.exe)
compiling applet, 24–25
compiling Java source, 23
executing for SDK, 21
writing programs, 20
Java Database Connectivity. *see* JDBC (Java Database Connectivity)
Java Development Environment, 17–36
applications vs. applets, 34
classes and filenames, 26–28
CLASSPATH variable, 27–28
CODEBASE references, 28
compressed packages, 33
getting started with SDK, 21–24
inside packages, 29–31
name collisions, 31–32
packages, 28–29
packages and filenames, 32–33
review of code samples, 34–36
runtime interpretation and Java byte codes, 17–21
writing applet with SDK, 24–26
java. exe (Java runtime), 21, 23
Java Foundation Classes (JFC), 20, 280–281
Java Naming and Directory Interface (JNDI), 327
Java run unit, 16
Java runtime (java. exe), 21, 23

Java Server Pages. *see* JSPs
Java Studio, 21
Java syntax, 149–165
binary arithmetic operators, 154–159
COBOL vs., 149–150
comments, 152–153
exercises, 160–165
flow control. *see* flow control, Java
operators, 153–154
reference variables, understanding with COBOL, 159–160
statements, 150–152
Java virtual machines (JVMs), 18–20
JAVA_HOME environment variable, 316
JavaBeans, 322
Javadoc
comments, 152–153
utility, 284–286
JAXP (Java API For XML Processing), 339–341
JDBC (Java Database Connectivity), 293–304
configuring JDBC-ODBC bridge, 301–302
connecting to, 296–298
exercises, 302–304
how it works, 294–295
inserting, updating and deleting, 300–301
overview of, 293
querying tables, 298–299
JDBC-ODBC bridge, 294–295, 301–304
JEE SDK, download and tutorial, 327–328
JFC (Java Foundation Classes), 20, 280–281
JFrame class, 280–281
JNDI (Java Naming and Directory Interface), 327
JSPs (Java Server Pages), 314–320
design principles for, 307
exercises, 317–320
getting started with servlets and, 316–317
overview of, 314–316
JVM (Java virtual machine), 18–20

K

keyed collections, 270–271

L

LANGUAGECODE static variable, 133, 135–136, 144
length numbers, arrays, 84
Level 88 items, COBOL, 336
LINKAGE SECTION, COBOL
data encapsulation and, 65–66
describing subroutine parameters in, 4
implementing inheritance, 109–112, 116–119
terms to review, 9–10
LinkedList, 272
List interface, collections
AbstractSequentialList implementation, 272
defined, 265
LinkedList implementation, 272
local variables
exercises, 96–98
overview of, 79–80
review, 100
scope, 170–172
logical operators, 169–170
lookup() method, JDB, 327

M

Map interface, collections, 265, 270
markup tags, Javadoc comments, 285–286
math functions, 202–204, 207
MathContext parameter, Big Decimals, 203
max( ) function, collections, 274
member data items, 47
members
hiding methods and, 131
Java, 71–72
memory
collections stored in, 265
shrinking Vectors or ArrayList to conserve, 269
message passing, 37
MessageBeans, EJBs, 324, 326–327
messages, 37–64
in COBOL, 38–40
defined, 132
ErrorMsg class, 63–64
exercises, 57–61
exercises in classes, objects and methods, 49–57
HelloWorld applet, 62–63
HelloWorld application, 61–62
in Java, 40–41
method overloading in COBOL, 44–48
multiple, 41–43
overview of, 37–38
terms, 48
method members, 87–89

method overloading
in COBOL, 44–48
exercise in, 56–58
in Java, 41–43, 48
review, 60–61
method signatures, 48
methods
ArrayList, 270
array-specific, 85–86
Connection class, 298
constructors vs., 89–91
ErrorMsg class, 40–41
extending inherited, 106–108
File class, 246–249
hiding members and, 131
hiding variables and, 126–127
inheriting, 103–104, 123–126
InputStream class, 250
Java, 48
Java string, 191–192, 194–196
numeric wrapper classes, 196–199
overriding, 123, 131
package, 70, 87–89
private, 70
public, 70
redefining inherited, 105–106
static, 70
StringBuffer, 200–201
Vector, 270
MIME (Multipurpose Internet Media Extension) type, 309
min( ) function, collections, 274
MS-DOS window, 21
MsgThread.java file, 237–238
MsgThreadRunnable.java file, 238–239
multiple inheritance, 121, 128–129
Multipurpose Internet Media Extension (MIME) type, 309
myErrorMsg class, 12–13, 58–60
MyFrame.java, 287–289
MYSUB COBOL
implementing inheritance, 116–119
message passing, 37–40
method overloading, 44–48
private variables, 66–68
MYSUB subroutine, 5–9
MYSUB-CONTROL
all instances as objects in, 8–9
behaving as object, 72–76
class data members in, 10
ErrorMsg class and, 11
inheritance, 110–112, 113–119
object-oriented language and, 7

N

name collisions, 31–32
naming conventions
classes inside packages, 29–31
Exception classes, 221–222
internal and external classes, 35
name collisions, 31–32
packages and files, 32–33
public Java classes, 26–27
source files, 26–27
New Java Project dialog, Eclipse, 358–359
NEWSUB COBOL, 109–119
next() method, ResultSet, 299
NoDatabaseAvailableException, 223, 228
Number class, 197–198
numbers
BigNumbers, 202–208
exercises, 210–217
numeric wrapper classes, 196–199
placing into Vector arrays, 269
rounding options in Java and COBOL, 206
numeric wrapper classes, 196–199, 210–213, 217

O

OAG (Open Application Group), XML, 344–351
object instance variables, 71, 72–76
object reference variables, Java, 157–160
object wrappers, 269
ObjectInputStream class, 253–254
object-oriented language. *see* OO (object-oriented language)
ObjectOutputStream class, 253–254
objects, 3–16
and COBOL, 5–10, 72–76
defined, 71–72
functionality of, 3
inheritance and, 103
Java and, 12–16, 76–77
message passing, 37

ODBC (Open Database Connectivity)
exercises, 302–304
JDBC as variation on, 293–294
JDBC-ODBC bridge, 294–295, 301–302
online references
authoring XML documents, 339
Eclipse, 20, 356
Java Software Development Kit, 20, 327
opening zip files, 356
predefined Exception objects, 221
Sun's coding conventions, 151, 167
Tomcat, 316
URI type file specification, 247
online XML, 343–344
OO (object-oriented language)
classes, objects and members, 71–72
COBOL and, 5–10, 112–119
inheritance and, 101–102
from Java perspective, 3, 13–14
message passing, 37
method overloading, 48
method signatures, 48
methods, 48
Open Application Group (OAG), XML, 344–351
Open Database Connectivity. *see* ODBC (Open Database Connectivity)
OPEN file, 244
operator overloading, 193, 203
operators, Java
binary arithmetic, 154–159
exercises, 163–165
flow control, 181
syntax, 153–154
options switches, 113
Order document type, 337
ordered collections, 266–270
ordering functions, collections, 273–275
outputfile property, properties file, 282
OutputStream class, 249–251, 258–260
overriding methods, 123, 131

P

Package Explorer, Eclipse
creating Java project, 359, 361
debugging with, 362
refactoring with, 363–367
packages, 28–33
compressed, 33
and filenames, 32–33
inside, 29–31
methods, 70, 87–89
name collisions, 31–32
overview of, 28–29
variables, 68
paint( ) function, 35
parameter entities, DTD, 336–337
parameters
arrays as, 87
calling subroutine, 4–9
passed, 47
parent classes. *see* superclasses (parent classes)
parentheses
controlling evaluation of expressions, 170
Java syntax, 153–154, 162–165
parse( ) method, DecimalFormat class, 197–198
parseInt( ) method, 196–197
passed parameters, Java, 47
pathname, Web servers, 309
PERFORM UNTIL, COBOL, 181, 187
performance, and Vectors, 267–268
period (.), 150
PIC X, COBOL, 189, 216
polymorphism, 132–133
PopupErrorMsg sample, 120, 122, 124–125, 128
port number, 308, 317
POST requests, 310, 312
precedence conventions, operators, 153–154
prefix operator (++x), 155
prepared statements, 300
prepareStatement( ) method, Connection, 300
primitive data types
autoboxing and unboxing, 199
Java Strings syntax vs., 190
overview of, 80–83
PrintFileErrorMsg sample
creating and using, 136–138
hiding variables and methods, 127, 138–139
inheritance, 119–120, 122, 124–126
review, 144–145
the this variable, 127–128
using Java interfaces, 129–130, 139–141
printIn( ) method, 216
printIn statement
ErrorMsg class, 57–58
exercises, class members, 92–93
review, 100

PrintStream, 245
private elements or properties, COBOL, 10
private elements or properties, Java, 14
private methods, 70, 87–89
private variables
COBOL, 66–68
Java, 68–70, 77, 94
properties, OO terminology, 7
properties files, Java, 281–282
protected keyword, Java, 122–123
protected variables, Java, 68, 77
public elements or properties, COBOL, 10
public elements or properties, Java, 14
public methods, 70, 87–89
public variables
COBOL, 68
Java, 68–70, 77, 94
put() method, Hashtable and HashMap, 271

Q
query strings, URL, 309–310
querying tables, 298–299
Queue interface, collections, 266

R
RandomAccessFile class, 256–258
READ file, 244
READ NEXT statement, COBOL, 299
reading data
InputStream methods, 250
Reader class, 254–256, 262
readLine( ) method, BufferedReader, 255
record-based I/O, 244–246
refactoring, with Eclipse, 363–367
reference variables
arrays as, 85
assignment and equality in, 157–159
defined, 71
understanding with COBOL, 159–160
release( ) method, threads, 234
Reload function, applet menu, 26
remote interface, coding EJB, 325
Remote Method Invocation (RMI), 252–254
replaceAll( ) function, collections, 273
ResultSet objects, querying tables, 298–299
reuse, Java support for, 77
reverse( ) function, collections, 273
RMI (Remote Method Invocation), 252–254
rounding functions, BigDecimals, 205–206
run( ) method
creating new thread, 230–232
exercises, 238–240
implementing Runnable interface, 232–233
Runnable interface, 232–233, 238–240
runtime
Eclipse, 362
garbage collection at, 236–237
Java byte codes at, 17–21
optimizing strings at, 191
RuntimeException class, 221–222, 228

S
s class. *see* derived classes
SAX (Simple API for XML) parsing interface, 340
Scanner class, 198, 216
schemas, XML, 339
scope, local variable, 170–172
SDK (Software Development Kit)
creating applets with, 24–26
development of, 20
getting started with, 21–24
jar utility program, 282–284
searches, XML-based, 352
security, EJB, 324
seek( ) method, RandomAccessFile, 257
SELECT statement, 244
semaphore tool, 234
semicolon (;), 150–152
Serializable interface, 252–254
serialization, 252–254
servlet engine
overview of, 310
processing JSPs, 315
servlet protocol, 311–314
servlets, 307–320
browsers and Web servers, 308–310
design principles for, 307
exercises, 317–320
getting started with, 316–317
Java Server Pages and, 314–316
protocol, 311–314
as transaction processor, 310
Session objects, creating, 313
SessionBeans, EJBs, 323, 326–327
Set interface, collections
caution about, 272

defined, 265
HashSet implementation, 271
setErrorMsg method
inheritance, 120–125
inheritance exercises, 134, 136–138, 140–145
message exercises, 58–60
polymorphism, 132
setMsgText method, 133–134, 144
setOut( ) method, PrintStream, 245
SGML (Standard Generalized Markup Language), 330
sharing, methods and variables, 126
signature, subroutine, 9
Simple API for XML (SAX) parsing interface, 340
SimpleFrame class, 287–289
single-type-import declaration statement, 31
16-bit Unicode characters, Java Strings, 190
size, Vectors and ArrayLists, 267–268
slashes (//), 153
sleep() method, threads, 239, 241
Software Development Kit. *see* SDK (Software Development Kit)
sort( ) function, collections, 273
SortedSet interface, collections
Comparator interface and, 275
defined, 265
TreeSet implementation of, 272
source filenames, 26–27, 33
SQL
JDBC support for, 294
querying tables using, 298–299
src directory, Eclipse, 360
standalone interpreters, 34
standard doclet, Javadoc, 286
Standard Generalized Markup Language (SGML), 330
standard in, 35–36
standard out, 35–36
start( ) method, 231, 233
stateful SessionBeans, EJBs, 323
stateless SessionBeans, EJBs, 323
statements, Java
COBOL vs., 150
in code blocks, 167
exercises, 161–163
flow control. *see* flow control, Java
syntax, 150–152
static class, 35, 70
static initializer code, 88–89
static keyword, 131
static methods, 70
StAX (Streaming API for XML) interface, 340
stop processing variables, 231
stopThread( ) method, 231–233, 238–240
stream-based IO, 244–246
Streaming API for XML(StAX) interface, 340
StringBuffers, 200–201, 213–214
strings, 202–208
comparing, 191–192
exercises, 210, 216
overview of, 189–191
StringBuffers, 200–201
working with, 193–196
subclasses
defined, 102
Exception class hierarchy, 221
inheritance and, 31–32, 103–104
subroutines
calling, 4–9
COBOL, 4
reusing in complex environments, 112–119
terminology, 9–10
Sun, integrated development environments, 20–21
super keyword, 131
superclasses (parent classes)
hiding methods and members, 131
inheritance and, 103
inheritance as design pattern and, 109
inherited methods, 103–104
inherited methods, extending, 106–108
inherited methods, redefining, 105–106
overview of, 102
swap( ) function, collections, 273
Swing class graphical libraries, 20, 280–281
switch statement, 176–177, 181
synchronization, 233–234, 264
synchronized keyword, 233–234
sysedit function, 22
System icon, Control Panel, 22
System.out.printIn( ) method
displaying information on console with, 244–245
I/O exercises, 259–260
Java compile and execution, 18–19
overview of, 35

T

tables, querying database, 298–299
tags, JSP, 316
templates, Java interfaces as, 129
text editors
  creating HelloWorld.java, 24
  getting started with SDK, 22
  writing programs in Java with, 20
TextMessage class sample
  creating, 133–134
  extending methods, 107
  hiding methods and members, 131
  hiding variables and methods, 127
  inheritance, 119–124, 126, 134–136
  inheriting methods, 103–104
  redefining methods, 105
  review, 144–145
  sharing variables and methods, 126
TEXT-STRING, 39
this keyword, 90, 127–128
Thread class, 230–232
threads, 229–241
  benefits and cautions, 235–236
  exercises, 237–241
  garbage collection, 236–237
  implementing Runnable, 232–233
  inheriting from, 230–232
  overview of, 229–230
  synchronization, 233–234
Throwable class, 221
timeToStop( ) method, threads, 231, 238–240
Toggle Breakpoint option, Eclipse, 362
Tomcat Web server, 316–317, 319
top-down implementation, inheritance, 102
toString( ) method, 193, 200
transaction contexts, EJBs, 324
transaction logic, updating database, 297
transaction processors, servlets as, 310
TreeSet, 272
try...catch code blocks, 223–226, 228–229
type cast, 82–83
type-import-on-demand declaration statement, 31
Types 1 - 4 JDBC drivers, 294–295

U

UML (Universal Modeling Language), static classes, 128
unboxing, 199
underscore character (_), 150
Unicode format
  comparing Java strings, 194
  double-byte, 254
  Java strings, 82, 189–190
  XML document characters similar to, 337
Universal Modeling Language (UML), static classes, 128
updates, database, 300–301
URI type file specification, 247
URL (Uniform Resource Locator)
  CODEBASEs defined by, 28
  overview of, 308–309
UTF-8 coding scheme, XML declarations, 337
utilities, Java, 282–286

V

value-added networks (VANs), EDI services, 342
VANs (value-added networks), EDI services, 342
variables
  clarifying for this operator, 127–128
  defined, 71
  exercises, 131–145
  hiding, 126–127
  Java, 68–70
  reference, 154–159
  sharing methods and, 126
  stop processing, 231
V**ector collection class,** 264, 266–270
virtual machine environment, 18

W

Web servers
  browsers and, 308–310
  servlet engines operating as, 310
  servlet protocol, 311–314
  Tomcat, 316–317
Web services, XML, 343–344
while statement
  defined, 181
  do...while statement vs., 174
  overview of, 172–174
  for statement vs., 176
Windows 98, Me, XP, Vista, 2000 and 2003, 22
WordPad, 22
WORKING-STORAGE area, COBOL
  class variables vs., 70
  data encapsulation and, 65–66

DTD comparable to, 334
implementing inheritance, 116–119
private items in, 10
Workspace Launcher, Eclipse, 357
write( ) method, OutputStream, 251
WRITE file, 244
writing data
OutputStream methods, 250–251
Writer class, 254–256, 262

X

x++ (postfix) operator, Java, 155
XBRL (Extensible Business Reporting Language), 352
XHTML, 340–341
XML (Extensible Markup Language), 329–354
authoring XML documents, 339
basics of, 330
complete XML documents, 338
declaration, 337–338
document type definitions, 332–333
DTD components, 333–337
Electronic Data Interchange, 342–343
exercises, 352–354
HTML vs., 330–332, 340–341
Java and, 339–340
OAG specifications for, 344–351
possible new application functions, 351–352
schemas, 339
Web services or online, 343–344
where to use, 341–342
XML documents
authoring, 339
complete, 338
defined, 330
XMLDocAnalyzer.java, 352–354
XSD (XML Schema Definition), 339

Z

zip files
compressed packages, 33
jar files vs., 282
unpacking, 356

License Agreement/Notice of Limited Warranty

By opening the sealed disc container in this book, you agree to the following terms and conditions. If, upon reading the following license agreement and notice of limited warranty, you cannot agree to the terms and conditions set forth, return the unused book with unopened disc to the place where you purchased it for a refund.

License:

The enclosed software is copyrighted by the copyright holder(s) indicated on the software disc. You are licensed to copy the software onto a single computer for use by a single user and to a backup disc. You may not reproduce, make copies, or distribute copies or rent or lease the software in whole or in part, except with written permission of the copyright holder(s). You may transfer the enclosed disc only together with this license, and only if you destroy all other copies of the software and the transferee agrees to the terms of the license. You may not decompile, reverse assemble, or reverse engineer the software.

Notice of Limited Warranty:

The enclosed disc is warranted by Course Technology to be free of physical defects in materials and workmanship for a period of sixty (60) days from end user's purchase of the book/disc combination. During the sixty-day term of the limited warranty, Course Technology will provide a replacement disc upon the return of a defective disc.

Limited Liability:

THE SOLE REMEDY FOR BREACH OF THIS LIMITED WARRANTY SHALL CONSIST ENTIRELY OF REPLACEMENT OF THE DEFECTIVE DISC. IN NO EVENT SHALL COURSE TECHNOLOGY OR THE AUTHOR BE LIABLE FOR ANY OTHER DAMAGES, INCLUDING LOSS OR CORRUPTION OF DATA, CHANGES IN THE FUNCTIONAL CHARACTERISTICS OF THE HARDWARE OR OPERATING SYSTEM, DELETERIOUS INTERACTION WITH OTHER SOFTWARE, OR ANY OTHER SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES THAT MAY ARISE, EVEN IF COURSE TECHNOLOGY AND/OR THE AUTHOR HAS PREVIOUSLY BEEN NOTIFIED THAT THE POSSIBILITY OF SUCH DAMAGES EXISTS.

Disclaimer of Warranties:

COURSE TECHNOLOGY AND THE AUTHOR SPECIFICALLY DISCLAIM ANY AND ALL OTHER WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING WARRANTIES OF MERCHANTABILITY, SUITABILITY TO A PARTICULAR TASK OR PURPOSE, OR FREEDOM FROM ERRORS. SOME STATES DO NOT ALLOW FOR EXCLUSION OF IMPLIED WARRANTIES OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THESE LIMITATIONS MIGHT NOT APPLY TO YOU.

Other:

This Agreement is governed by the laws of the State of Massachusetts without regard to choice of law principles. The United Convention of Contracts for the International Sale of Goods is specifically disclaimed. This Agreement constitutes the entire agreement between you and Course Technology regarding use of the software.