Danny Poo
Derek Kiong
Swarnalatha Ashok

Object-Oriented Programming and Java

Second Edition

Springer

Object-Oriented Programming and Java

Danny Poo • Derek Kiong •
Swarnalatha Ashok

Object-Oriented Programming and Java

Second edition

Springer

Dr Danny Poo
School of Computing
National University of Singapore, Singapore

Dr Derek Kiong
Institute of Systems Science
National University of Singapore, Singapore

Ms Swarnalatha Ashok
Institute of Systems Science
National University of Singapore, Singapore

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Control Number: 2007934261

ISBN-13: 978-1-84628-962-0 e-ISBN-13: 978-1-84628-963-7
First edition © Springer Singapore 1998; 978-981-3083-96-7

Printed on acid-free paper

© Springer-Verlag London Limited 2008

Apart from any fair dealing for the purposes of research or private study, or criticism or review, as permitted under the Copyright, Designs and Patents Act 1988, this publication may only be reproduced, stored or transmitted, in any form or by any means, with the prior permission in writing of the publishers, or in the case of reprographic reproduction in accordance with the terms of licences issued by the Copyright Licensing Agency. Enquiries concerning reproduction outside those terms should be sent to the publishers.

The use of registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant laws and regulations and therefore free for general use.

The publisher makes no representation, express or implied, with regard to the accuracy of the information contained in this book and cannot accept any legal responsibility or liability for any errors or omissions that may be made.

9 8 7 6 5 4 3 2 1

Springer Science+Business Media
springer.com

Table of Contents

Preface to 1st Edition **xiii**
Preface to 2nd Edition **xv**
Overview **xvii**
Acknowledgement **xix**

1 Introduction **1**
1.1 Object-Oriented Programming 1
1.2 Objects and Their Interactions in the Real World 2
1.3 Objects and Their Interactions in Programming 3
1.4 Simulation 3
1.5 Java 4
1.6 Summary 4
1.7 Exercises 5

2 Object, Class, Message and Method **7**
2.1 Objects and Class 7
2.2 Message and Method 9
2.2.1 Message Components 10
2.2.2 Method 10
2.2.3 Client and Server 11
2.3 Creating Objects 12
2.4 Summary 14
2.5 Exercises 14

3 A Quick Tour of Java **17**
3.1 Primitive Types 17
3.2 Object Definition 18
3.2.1 Variable Definitions 18
3.2.2 Methods 19
3.3 Object Instantiation 20
3.4 Object Access and Message Passing 21
3.5 Representational Independence 21
3.6 Overloading 22
3.7 Initialization and Constructors 23

3.8 Expressions, Statements, and Control-flow Mechanisms 24
3.8.1 Operators 24
3.8.2 Expression Statements 30
3.8.3 Control-flow Statements 30
3.9 Blocks 32
3.9.1 Local Declarations 32
3.10 More Control-flow Statements 33
3.11 Arrays 34
3.12 Result Returned by Method 35
3.13 Summary 36
3.14 Exercises 36

4 Implementation in Java 39
4.1 Calculator 39
4.1.1 The clear() Method 40
4.1.2 The display() Method 41
4.1.3 The digit() Method 41
4.1.4 Operator Methods 41
4.2 Code Execution 42
4.3 Simple User Interface 44
4.4 Another Interface for CalculatorEngine 46
4.4.1 Event-Driven Programming 48
4.5 Summary 49
4.6 Exercises 49

5 Classification, Generalization, and Specialization 51
5.1 Classification 51
5.2 Hierarchical Relationship of Classes 53
5.2.1 Superclass and Subclass 53
5.2.2 A Class Hierarchy Diagram 54
5.3 Generalization 55
5.4 Specialization 56
5.5 Organization of Class Hierarchy 56
5.6 Abstract and Concrete Classes 57
5.7 Summary 58
5.8 Exercises 58

6 Inheritance 61
6.1 Common Properties 61
6.2 Inheritance 62
6.3 Implementing Inheritance 64
6.4 Code Reuse 67

6.5 Making Changes in Class Hierarchy 67
6.5.1 Change in Property Definition for All Subclasses 67
6.5.2 Change in Property Definition for Some Subclasses 68
6.5.3 Adding/Deleting a Class 72
6.6 Accessing Inherited Properties 75
6.7 Inheritance Chain 75
6.7.1 Multiple Inheritance 76
6.7.2 Problems Associated with Multiple Inheritance 77
6.7.3 Contract and Implementation Parts 79
6.7.4 Contract and Implementation Inheritance 79
6.8 Interface 80
6.8.1 Multiple Inheritance Using Interface 80
6.8.2 Attributes in an Interface 83
6.8.3 Methods in an Interface 83
6.8.4 Abstract Class and Interface 83
6.8.5 Extending Interface 84
6.8.6 Limitations of Interface for Multiple Inheritance 85
6.9 Summary 88
6.10 Exercises 89

7 Polymorphism 93
7.1 Static Binding 93
7.2 Dynamic Binding 96
7.3 Operation Overloading 97
7.3.1 Same Method Signature 97
7.3.2 Overloading Method Names 98
7.4 Polymorphism 100
7.4.1 Selection of Method 100
7.4.2 Incremental Development 101
7.4.3 Increased Code Readability 102
7.5 Summary 102
7.6 Exercises 102

8 Modularity 103
8.1 Methods and Classes as Program Units 103
8.2 Object and Class Properties 103
8.2.1 Counting Instances 104
8.2.2 Shared Attributes 106
8.2.3 Class Attributes 107

8.2.4 Class Methods 107
8.2.5 Name Aliases 108
8.3 Controlling Visibility 108
8.4 Packages 110
8.4.1 The package Keyword 110
8.4.2 The import Keyword 110
8.5 Encapsulation 111
8.5.1 Bundling and Information Hiding 112
8.5.2 Enhanced Software Maintainability 112
8.5.3 Trade-Off 115
8.6 Summary 116
8.7 Exercises 117

9 Exception Handling 119
9.1 Using Exceptions 119
9.2 Exception Terminology 120
9.3 Constructs and Exception Semantics in Java 120
9.3.1 Defining Exception Objects 121
9.3.2 Defining Exception Handlers 121
9.3.3 Raising Exceptions 122
9.4 A Simple Example 123
9.5 Paradigms for Exception Handling 125
9.5.1 Multiple Handlers 125
9.5.2 Regular Exception Handling 127
9.5.3 Accessing Exception Objects 128
9.5.4 Subconditions 128
9.5.5 Nested Exception Handlers 129
9.5.6 Layered Condition Handling 130
9.6 Code Finalization and Cleaning Up 130
9.6.1 Object Finalization 131
9.6.2 Block Finalization 131
9.7 Summary 132
9.8 Exercises 133

10 Input and Output Operations 135
10.1 An Introduction to the Java API 135
10.2 Reading the Java API Documentation 136
10.3 Basic Input and Output 138
10.4 File Manipulation 141
10.4.1 File Input 142
10.4.2 File Output 143
10.4.3 Printing Using PrintStream 144

10.5 Framework for Code Reuse 145
10.6 DataInputStream and DataOutputStream Byte Stream Class 147
10.7 Character Stream Classes 148
10.8 Tokenizing the Input Using the Scanner Class 150
10.9 Formatting the Output Using the Format String 151
10.10 The File Class 152
10.11 Random Access File Operations 152
10.12 Summary 153
10.13 Exercises 153

11 Networking and Multithreading 155
11.1 The Network Model 155
11.2 Sockets in Java 156
11.2.1 Example Client: Web Page Retriever 157
11.3 Listener Sockets in Java 161
11.3.1 Example Server: Simple Web Server 161
11.3.2 Running the Web Server 164
11.4 Considering Multiple Threads of Execution 165
11.5 Creating Multiple Threads of Execution 166
11.5.1 Thread Creation Using the Thread Class 166
11.5.2 Thread Creation Using the Runnable Interface 168
11.6 Improvement of Web Server Example 168
11.7 Thread Synchronization and Shared Resources 169
11.8 Summary 175
11.9 Exercises 176

12 Generics and Collections Framework 179
12.1 Introduction 179
12.2 Rationale Behind Generics 179
12.2.1 The Problem 180
12.2.2 Run-time Type Identification (RTTI) 182
12.3 Java Generics 183
12.3.1 Generic Class 183
12.3.2 Generic Method 185
12.4 Collections Framework 186
12.4.1 Collections Interface 186
12.4.2 ArrayList Class 187
12.4.3 HashSet Class 190
12.4.4 HashMap Class 194
12.5 Sorting Collections 196

12.5.1 Sort Algorithm 196
12.5.2 Comparator Interface 197
12.6 Searching Collections 198
12.6.1 indexOf and contains Methods 198
12.6.2 binarySearch Method 198
12.7 Summary 199
12.8 Exercises 199

13 Graphical Interfaces and Windows 201
13.1 The AWT Model 201
13.2 Basic AWT Constituents 202
13.2.1 Frames 203
13.2.2 Components 204
13.2.3 Panels 205
13.2.4 Layout in Panels 206
13.2.5 Events 209
13.2.6 Events in JDK 1.1 (and later versions) 212
13.3 Basic Components 214
13.3.1 Label Component 214
13.3.2 Button Component 215
13.3.3 Checkbox Component 215
13.3.4 CheckboxGroup Component 215
13.3.5 TextArea Component 216
13.3.6 Choice Component 217
13.3.7 List Component 218
13.3.8 Menus and Menu Items 219
13.3.9 Dialog Frames 221
13.3.10 File Dialog Frames 223
13.4 Custom Components 224
13.5 Other Kinds of Class Definitions 226
13.5.1 Inner Classes 227
13.5.2 Anonymous Classes 227
13.5.3 Local Classes 228
13.6 Swing Components 230
13.6.1 Transiting from AWT to Swing 231
13.6.2 Model versus View 234
13.7 Summary 235
13.8 Exercises 236

14 Applets and Loaders 237
14.1 Applet Characteristics 237
14.2 Applet Life Cycle 241

14.3 Custom Applets 242
14.4 Images and Audio 243
14.5 Animation in Applets 245
14.6 Efficient Repainting 247
14.7 Applet Parameters 248
14.8 Loading Code Dynamically 250
14.9 Security Restrictions for Untrusted Code 253
14.9.1 Security Policy 255
14.9.2 Keys 256
14.9.3 Permissions 256
14.10 Summary 258
14.11 Exercises 258

15 Java Servlets 259
15.1 Dynamic Web Pages and Servlets 259
15.2 Tomcat Installation 260
15.2.1 Downloading and Installation 260
15.2.2 Configuration 261
15.2.3 Starting and Stopping Tomcat 262
15.3 Sample Servlet 263
15.4 Servlet Characteristics 266
15.5 Servlet Paramters and Headers 266
15.6 Servlet Output 271
15.7 Handling Sessions 271
15.7.1 Session Timeout 274
15.8 Concurrency 274
15.9 Customized Processors 274
15.10 Summary 276
15.11 Exercises 277

16 Object Serialization and Remote Method Invocation 279
16.1 Object Serialization 279
16.2 Components in Object Serialization 281
16.3 Custom Serialization 281
16.3.1 The Externalizable Interface 284
16.4 Distributed Computing with Java 284
16.4.1 RMI and CORBA 285
16.4.2 Java Limitations 285
16.5 An Overview of Java RMI 286
16.6 Using Java RMI 287
16.6.1 Setting Up the Environment on Your Local Machine 287

16.6.2 How RMI Works 287
16.6.3 An RMI Example 288
16.7 RMI System Architecture 289
16.8 Under the Hood 291
16.9 RMI Deployment 293
16.10 Summary 295
16.11 Exercises 295

17 Java Database Connectivity 297
17.1 Introduction 297
17.2 Java Database Connectivity 297
17.3 JDBC Architecture 298
17.4 JDBC Drivers 298
17.4.1 Types of Drivers 299
17.5 JDBC APIs 302
17.5.1 Establishing a Connection 302
17.5.2 Data Manipulation 303
17.6 Data Definition Language (DDL) with JDBC 305
17.6.1 Creating a Table 305
17.6.2 Dropping a Table 306
17.7 Data Manipulation Language (DML) with JDBC 307
17.7.1 Creating (Inserting) Records Using JDBC 307
17.7.2 Deleting Records Using JDBC 307
17.7.3 Retrieving Records Using JDBC 307
17.7.4 Updating Records Using JDBC 309
17.7.5 Updatable Result Sets 310
17.7.6 Prepared Statements 311
17.8 Summary 313
17.9 Exercises 313

Index 315

Preface to 1st Edition

Control abstraction was the message of the first programming revolution seen in high-level programming languages such as Algol and Pascal. The focus of the next revolution was data abstraction, which proposed languages such as Modula and Ada.

The object-oriented revolution began slowly in the 1960s with the programming language Simula, but moved onto more languages such as Smalltalk, Objective-C and C++. Java is almost a hybrid between Smalltalk and C++, and has gained widespread acceptance due to its association with the Internet, its availability to a large user base and reusable libraries for programming in a graphical environment.

Our programming lineage has passed through Pascal, C and C++. As with many other programmers, good run-time checks with automatic memory management and a reusable API made Java a very attractive option. After a half-day on the original Java Whitepaper and the early Java online tutorial, we were sold on the Java bandwagon and already writing code. In another two days' time, we were using the Abstract Windowing Toolkit (AWT) package for graphical applications. In situations where there is no large investment into older languages, we are quite happy to abandon them completely.

Effective programming in Java comes from understanding three key areas – object-oriented concepts, the syntax and semantics of the Java programming language and the Java Application Programming Interface (API). This is our emphasis when we conduct professional courses, and in this book as well.

Much of the material in this book is based on previous courses which we have conducted over the past two years to the industry and the National University of Singapore (NUS). Courses conducted for the industry last about 5 to 7 days, depending on the amount of coaching that participants require. In the Department of Information Systems and Computer Science at NUS, a course on "Object-Oriented Methods" runs over 13 weeks.

As you might have noticed, we have taken to Java as ducks to water. Java has allowed us to think about and specify object behavior. This results in executable code which is merely secondary. What is important is the clean specification of object behavior. Similarly, in getting accustomed to working with objects, we believe that you will enjoy it too.

Preface to 2nd Edition

Since publishing the first edition almost 10 years ago, we have seen Java being used in many high school and university programming courses. Further, many projects now use Java as the implementation language. Similarly, at the Institute of Systems Science, we have seen professional developers warming up to Java for the first time in 1998, to those who use Java in their daily work in 2007.

We have thus updated the material to cover J2EE topics such as JDBC, RMI, Serialization and Java Servlets. We have also added a chapter on Generics as the Java language evolved to allow this elegant feature.

For those who might be embarking on a Java journey now, we wish you a pleasant journey and a well-used road map. Many have taken this journey before and are enjoying the fruits of their learning investment.

Overview

Chapter 1 presents an introduction to the object-oriented world consisting of objects and object communication via the exchange of messages. Object-oriented concepts and terminology used in object-oriented methodology are discussed in chapter 2. Chapter 3 shows how these concepts materialize in the form of Java code and representations. It discusses the basic features and syntax of Java and builds upon the concepts using an incremental Counter example.

Following on from language syntax, chapter 4 demonstrates the standard programming environment using the Java Development Kit (JDK), and how a class definition may be compiled and executed, integrated and reused within other code fragments. The chapter also delves into using the Java Application Programming Interface (API) to demonstrate the ease and productivity gains of code libraries.

Chapter 5 returns to the discussion of objects, in particular, the organization of objects into manageable classes. The concept of class enables a developer to organize a complex problem domain into more manageable components. Grouping objects into classes is an act known as *classification* in object-oriented modeling. When classes are formed, they can be further distinguished into *superclasses* or *subclasses*, according to their similarities or differences in properties. Class hierarchies can then be formed. The creation of superclasses and subclasses is achieved through abstracttion mechanisms known as *generalization* and *specialization* respectively. Classification, generalization and specialization are thus important abstraction mechanisms for organizing objects and managing complexities.

Inheritance is discussed in chapter 6. Common properties of classes can be shared with other classes of objects via the inheritance mechanism. It is through inheritance that software component reuse is possible in object-oriented programming. Software reusability is important because code need not be produced from scratch, thereby increasing the productivity of developers.

Another topic close to the heart of object-oriented programming is *polymorphism*. This topic is concerned with object messaging and how objects of different classes respond to the same message. With polymorphism, objects of different class definition can respond to the same message with the appropriate method. In this way, generic software code can be produced, thus enhancing the maintainability of software systems. Polymorphism is supported by dynamic binding and operation overloading, topics that are central to the discussion in chapter 7.

Enhancing software maintainability is a significant software development objective. A programming technique known as Structured Programming was introduced in the 1980s, promoting modularity as a Software Engineering principle for achieving maintainable software. Modularity is emphasized in object-oriented

programming in the form of method, object, and class definition. *Encapsulation* is the manifestation of modularity in object-oriented programming to the fullest. As will be made clear in chapter 8, encapsulation brings together related properties into class definitions with the structural definition of classes hidden from their users. The purpose of this approach is to hide the implementation detail of objects so that when changes in implementation of objects are called for, users of the objects will not be adversely affected.

Exception Handling is considered in chapter 9. This is especially important in object-oriented programming, as the mechanism for the glue and safety net in code integration and reuse.

The Java API is introduced in chapter 10 and continues with the core classes for input/output, networking, graphical components and applets within Web browsers. Input and output rely on `InputStream` and `OutputStream` classes, as well as `Reader` and `Writer` classes in JDK 1.1.

Chapter 11 introduces network connections via TCP/IP using the `Socket` class, similar to those for input and output in chapter 10, as they share behavior from `InputStream` and `OutputStream`. As multi-processing is typically used with client/server applications, we have also included the multi-threading API in this chapter, together with a skeleton Web server as the working example.

Collection classes with Generics in chapter 12 show how the concepts of modularity and encapsulation work to facilitate code reuse. This chapter not only gives an overview of the classes in the Collections Framework, but this framework is an excellent sample of how reusable code is written.

The AWT model is elaborated with descriptions of its constituents and example usage for Graphical User Interfaces in chapter 13. There are sufficient code examples to build interfaces for most simple applications. We have also incorporated Swing classes for better interactivity.

Applet development relate to graphical interfaces and the issue of dynamic loading of compiled Java bytecodes. This is discussed in chapter 14. Situations where applet behavior differs from Java applications, security measures and implementing a loader over the network are also considered.

Chapter 15 covers Java Servlets. It is related to dynamic code loading and applets in chapter 14, but occurring on the server side. This forms the basis for Java-based Web applications.

Chapter 16 examines Java Object Serialization and Remote Method Invocation. The former may be viewed as a continuation of input and output facilities discussed in chapter 10, but with the focus to implement object persistence. Object Serialization is also used to move objects over a network and forms a key role in implementing Remote Method Invocation (RMI) for distributed applications. A simple client/server application framework using RMI is provided.

Chapter 17 provides an overview of the popular requirement of working with databases – Java Database Connectivity. This topic warrants a whole book, but we limit our discussion to the rationale, perspective and architecture of JDBC together with necessary methods for working with real databases.

Acknowledgement

In revising the text, we are thankful to readers of the first edition who have given encouraging feedback. If not for these folks, we would never have considered this second round.

We also thank our colleagues and bosses at our respective work places who have supported and encouraged this book revision. We also thank the folks at Springer Verlag who felt that a second edition was worthy.

Special thanks are due to Derek's former colleagues at the now defunct Centre for Internet Research (CIR), National University of Singapore, who had worked to use the Internet productively. The lineage of CIR may be traced back to the Technet Unit in the Computer Centre of the National University of Singapore, which was first to provide and promote Internet services in Singapore. The effort saw the spin-off to PacNet in Singapore. In particular, Dr. Thio Hoe Tong, former Director of the Computer Centre and Dr Tan Tin Wee, have supported the Java team even in the early days when we played with the Alpha releases of Java.

Poo, Kiong & Ashok
National University of Singapore
August 2007

1

Introduction

Object-oriented programming has been in practice for many years now. While the fundamental object-oriented concepts were first introduced via the class construct in the Simula programming language in the 1960s, the programming technique was only accepted with the advent of Smalltalk-80 more than a decade later.

Object-oriented programming has come a long way. More and more programs are now designed and developed using this approach. What is object-oriented programming? What makes it attractive as an alternative programming approach? How does it differ from the traditional procedural programming approach? These questions will be discussed in this chapter.

1.1 Object-Oriented Programming

The procedural approach to programming was the de facto approach in the early days of programming. Here, code is modularized based on a system's processes. For instance, in developing a library application system, we would have considered processes such as the checking in an d out of books, making reservations of books, cataloging of books, and so on. Problem solving would involve the analysis of these processes in terms of the procedural tasks carried out and the production of a system whose representation is based on the procedural flow of the processes.

Object-oriented programming, on the other hand, models objects and their interactions in the problem space and the production of a system based on these objects and their interactions.

Since the real-world problem domain is characterized by objects and their interactions, a software application developed using the object-oriented programming approach will result in the production of a computer system that has a closer representation of the real-world problem domain than would be the case if the procedural programming approach is used.

1.2 Objects and Their Interactions in the Real World

Let us consider a real-world situation. There are two persons, Benjamin and his wife, Bambie. They are customers of HomeCare, a company dealing in luxurious furniture. HomeCare sells a variety of sofa sets. Each sofa set is labeled with an identification number and a price tag. After viewing the sofa sets for an hour, Benjamin and Bambie decide to purchase a green leather 5-seater set. They approach Sean, a salesperson, to place their order.

In making his request known to Sean, Benjamin sends a *message* to Sean, "I would like to purchase this green leather, 5-seater set. Can you please have it sent to me by next Wednesday?"

The message that Benjamin has sent to Sean is a `takeOrder` message. It contains information such as the type of sofa set (a green leather, 5-seater set) and the date of delivery (next Wednesday). This information is known as the *parameters* of the `takeOrder` message.

In response to Benjamin's message, Sean replies to Benjamin by returning the result of his request. We can represent the interaction between Benjamin and Sean graphically using Figure 1-1.

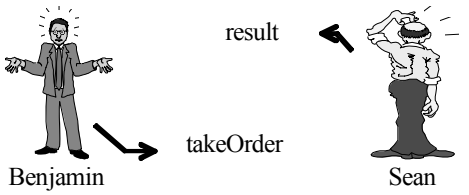


Figure 1-1: Interaction between Benjamin and Sean.

Sean was able to respond to Benjamin's `takeOrder` message because he understood it and had the means to handle Benjamin's request. Although Sean knew how to satisfy Benjamin's request, Benjamin did not. In fact, most of the time, customers do not know how a salesperson has satisfied their orders. All they get from salespersons are replies such as, "I am sorry, madam, we are unable to satisfy your request because the sofa you wanted has been sold," or "Sir, your request has been satisfied. We will deliver the goods on Wednesday between 10 am to 11 am to the address indicated. Thank you for your order."

Sean, as a salesperson at HomeCare, has a responsibility towards Benjamin. He maintains his responsibility by applying a set of operations:

1. He determines if there is sufficient stock to satisfy Benjamin's request.
2. He determines if the requested date for delivery is a suitable date.
3. He instructs the warehouse staff to deliver the goods to Benjamin's address on the requested date, if the above conditions are satisfied.
4. Finally, he informs Benjamin the result of his request.

1.3 Objects and Their Interactions in Programming

The interactions between Benjamin and Sean in the above real-world situation can be represented in object-oriented programming terms. For instance, Benjamin and Sean are *objects* that interact by sending *messages*. Benjamin is thus a *message-sending object,* while Sean is a *message-receiving object*. Alternatively, we can label Benjamin as a sender and Sean as a receiver.

The `takeOrder` request from Benjamin to Sean is an example of a *message*. It may have additional, accompanying information known as *parameters* (or *arguments*) of the message. The fact that Sean responded to Benjamin's message indicates that the message is a valid message. Each valid message corresponds to a *method* that Sean uses to fulfill his responsibility to Benjamin.

An invalid message, on the other hand, is one that the receiver does not have the capability to respond to, that is, the receiver does not have a corresponding method to match the message. For example, if Benjamin had requested a discount on the price, his request would have been rejected because Sean, being a salesperson, would not have the capability (or a corresponding method) to respond to the message.

A method contains a number of operations detailing how Sean is to satisfy the demand Benjamin put on Sean through the request.

Figure 1-2 summarizes the relationships of these terms.

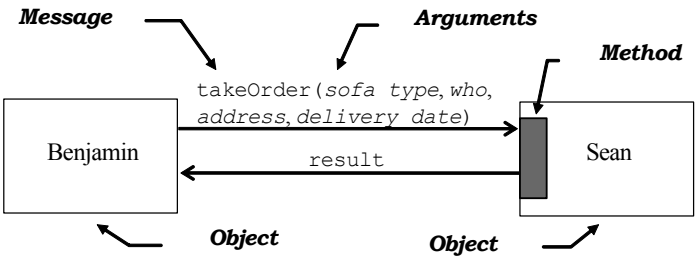


Figure 1-2: Object interactions in object-oriented programming terms.

While Benjamin may know *what* Sean can do through his methods, he may not know *how* Sean does them. This is an important principle of object-oriented programming known as *information hiding:* the sender of a message does not know *how* a receiver is going to satisfy the request in the message.

1.4 Simulation

Based on the above example, it is clear that concepts in object-oriented programming such as *object*, *message* and *method*, do provide a close representation of real-world objects and their interactions. These concepts are thus suitable for simulating actual object interactions in real-world situations.

It is this ability for modeling real-world problems that identified object-oriented programming as being suitable for simulation. The Simula programming language was designed in the early 1970s to provide simulation abilities using object-oriented concepts.

1.5 Java

Java was first introduced in 1995 as a simple and secure object-oriented programming language. It is a unique language in that, being a new language at that time, it was able to attract a lot of interest from the computing community. Within two years after Java was launched, there were an estimated 400,000 Java programmers and over 100 books on Java programming.

There are a few possible reasons for the phenomenal interest in Java. The year 1995 saw a maturing of Web technologies, and Java's multiplatform capability, which enabled a Java program to execute on any computer, was exceedingly attractive, especially on an open network like the Internet. Java is implemented via part compilation and subsequent execution on an interpreter implemented in software. Java applications are therefore object code portable as long as a Java virtual machine is implemented for the target machine.

The popularity of Java is also ironically due to its similarity with its close rival C++. Java takes the pain out of learning a new language by reusing much of C and C++. At the same time, safe programming practice in Java and language facilities for automatic memory management were benefits that were attractive to programmers on the verge of deserting their C/C++ camps.

In relation to the Internet, Java applets have given rise to a new generation of distributed applications with low software distribution and maintenance costs. As applets are embedded in an HTML document via `<APPLET>` tags, its transmission to the client machine for execution is implicitly handled by the underlying network protocols and thus makes the typical channels of distribution and installation obsolete.

While the object-oriented programming framework promotes reusability of software and code, this very practice has been demonstrated in the rich set of class libraries seen in the Java language. The Java foundation class libraries provide for windowing and graphical user interface programming, network communications, and multimedia facilities. Together, they demonstrate the practical and productive work done in Java.

1.6 Summary

In this chapter, we discussed:

- An overview of object-oriented programming concepts and their applicability for modeling and representing real-world entities and their interactions in the problem-solving process.

- Object-oriented concepts of object, message, and method.
- An overview of the Java programming language and the potential of productive software development

1.7 Exercises

1. Distinguish the programming approach used in procedural programming and object-oriented programming.
2. Discuss the validity of the following statement: The object-oriented programming approach is ideal for simulating real-world problems.
3. Consider the following scenarios and outline the objects and their interactions in terms of messages and arguments:

    (a) a driver driving a car;
    (b) a customer making a cash withdrawal from an automated teller machine (ATM);
    (c) a customer buying a compact disk player from a vendor;
    (d) a traffic policeman directing traffic at a junction;
    (e) a lecturer delivering his/her lecture to a class of students;
    (f) a tutorial discussion between an instructor and students.

2

Object, Class, Message and Method

We had our first introduction to objects, message and method in Chapter 1. Another concept closely associated with the concept of objects is *class*. In object-oriented programming, a *class* is a definition template for structuring and creating objects.

In this chapter, we will discuss the concept of object, message, method and class and how these concepts are used in a computer model.

2.1 Objects and Class

In Chapter 1, we introduced Benjamin. Now, meet Bernie, another customer at HomeCare. As customers of HomeCare, Benjamin and Bernie share some similar information. For example, both have a name, an address, and a budget—information that is relevant when describing customers. This information is known as *object attributes*.

An object attribute definition allows for objects to have independent *attribute values*. For example, Benjamin may have a larger budget and thus a larger budget value (say $2000) than Bernie whose budget may be $1000. Collectively, the values of an object's attributes represent the *state* of the object.

Besides attributes, Benjamin and Bernie also exhibit some behavior typical of a customer. For instance, Benjamin and Bernie execute a *method* when making a purchase. Let us call this method `purchase()`. The method `purchase()` is made up of a set of operations that Benjamin and Bernie would use to send a purchase request to a salesperson.

Structurally, Benjamin and Bernie can be represented as follows:

```
Benjamin as an Object
  Attributes:
    name = "Benjamin"
    address = "1, Robinson Road"
    budget = "2000"
  Methods:
    purchase()   {send a purchase request to a salesperson}
    getBudget()  {return budget}
Bernie as an Object
  Attributes:
    name = "Bernie"
    address = "18, Sophia Road"
    budget = "1000"
  Methods:
    purchase()   {send a purchase request to a salesperson}
    getBudget()  {return budget}
```

`name`, `address` and `budget` are attributes while `purchase()` and `getBudget()` are methods of the two objects. Note that both objects share a common definition of attributes and methods. In fact, all customers of HomeCare share the same set of attribute and method definitions. They all have attributes `name`, `address` and `budget`, and methods `purchase()` and `getBudget()`. In defining these objects, a common definition known as *class* is used.

A class is a definition template for structuring and creating objects with the same attributes and methods. Benjamin and Bernie, being customers of HomeCare, can therefore be defined by a class called Customer as follows:

```
Class Customer
  Attributes:
    name
    address
    budget
  Methods:
    purchase()   {send a purchase request to a salesperson}
    getBudget()  {return budget}
```

One major difference between objects and class is in the way attributes and methods are treated in objects and classes. A class is a definition about objects; the attributes and methods in a class are thus declarations that do not contain values. However, objects are created instances of a class. Each has its own attributes and methods. The values of the set of attributes describe the state of the objects.

Let us now examine the salespersons. Salespersons also have attributes and methods. Sean and Sara are two salespersons at HomeCare. They are thus capable of a behavior typical of a salesperson, for example, taking orders from customers. To fulfill their role as salespersons in a purchase transaction, Sean and Sara perform a method. We shall call this method `takeOrder()`, and represent Sean and Sara as follows:

```
Sean as an Object
  Attributes:
    name = "Sean"
  Methods:
    takeOrder()      {
      check with warehouse on stock availability
      check with warehouse on delivery schedule
      if ok
      then {instruct warehouse to deliver stock(address, date)
            return ok}
      else return not ok
    }

Sara as an Object
  Attributes:
    name = "Sara"
  Methods:
    takeOrder()      {
      check with warehouse on stock availability
      check with warehouse on delivery schedule
      if ok
      then {instruct warehouse to deliver stock(address, date)
            return ok}
      else return not ok
    }
```

Being salespersons, Sean and Sara share similar attributes and methods as expected. Like the customers, their definition can be described by a class called SalesPerson with the following representation:

```
Class SalesPerson
  Attributes:
    name
  Methods:
    takeOrder()      {
      check with warehouse on stock availability
      check with warehouse on delivery schedule
      if ok
      then {instruct warehouse to deliver stock(address, date)
            return ok}
      else return not ok
    }
```

Note that the definition of the SalesPerson class is different from the Customer class since customers and salespersons behave differently—customers make orders and salespersons take orders.

2.2 Message and Method

Objects communicate with one another by sending *messages*. A message is a *method call* from a message-sending object to a message-receiving object. A message-sending object is a *sender* while a message-receiving object is a *receiver*.

An object responds to a message by executing one of its methods. Additional information, known as *arguments*, may accompany a method call. Such parameterization allows for added flexibility in message passing. The set of methods collectively defines the dynamic behavior of an object. An object may have as many methods as required.

2.2.1 Message Components

A message is composed of three components:

- an object identifier that indicates the message receiver,
- a method name (corresponding to a method of the receiver), and
- arguments (additional information required for the execution of the method).

Earlier we saw that Benjamin sent a message to Sean when Benjamin wanted to buy a sofa set. The reasonable location for Benjamin to send the message to Sean is in Benjamin's `purchase()` method as shown below (indicated in bold):

```
Benjamin as an Object
  Attributes:
    name = "Benjamin"
    address = "1, Robinson Road"
    budget = "2000"
  Methods:
    purchase() {
      Sean.takeOrder("Benjamin", "sofa", "1, Robinson Road",
                     "12 November")
    }
    getBudget()      {return budget}
```

The message `Sean.takeOrder(who, stock, address, date)` is interpreted as follows:

- Sean is the receiver of the message;
- `takeOrder` is a method call on Sean;
- `"Benjamin","stock","address","date"` are arguments of the message.

2.2.2 Method

A message is valid if the receiver has a method that corresponds to the method named in the message and the appropriate arguments, if any, are supplied with the message. Only valid messages are executed by the receiver. The `takeOrder()` message is valid because Sean has a corresponding method and the required arguments (`who`, `stock`, `address`, `date`) are supplied with the message.

Sean's `takeOrder()` method is made up of a set of operations (indicated in bold below) as follows:

```
Sean as an Object
  Attributes:
   name = "Sean"
  Methods:
    takeOrder(who, stock, address, date)    {
      check with warehouse on stock availability
      check with warehouse on delivery schedule
      if ok then {
        instruct warehouse to deliver stock to address on date
        return ok
      } else return not ok
    }
```

In the above description, a message is sent from Sean to a Warehouse object to inquire on the order and delivery schedule in Sean's `takeOrder()` method. If both conditions are satisfied, Sean will instruct the Warehouse object to arrange for delivery.

How Sean carries out the method is known only to Sean. Neither Benjamin nor the other customers know how Sean does it. For example, to check on the stock and delivery schedule with the warehouse, Sean may have called the warehouse over the phone or he may have checked the information against a list he had gotten from the warehouse. What Benjamin knows of Sean is that Sean is capable of responding to his request since his message to Sean is acceptable by Sean.

In object-oriented programming, Benjamin and Sean are said to have followed the principle of *information hiding—How* Sean is going to satisfy Benjamin's request *is hidden* from Benjamin. In this way, Sean is free to select whatever way he chooses to satisfy Benjamin's request; he may phone the warehouse or look up the pre-prepared list and vice versa.

2.2.3 Client and Server

By executing a method, a message-receiving object (such as Sean) is said to serve the message-sending object (such as Benjamin). A message-receiving object is thus a *server* to a message-sending object and the message-sending object is thus a *client* of the server.

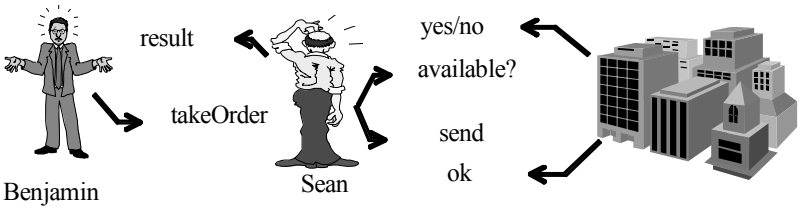


Figure 2-1: Object communication process.

In any object communication, there are at least a client and a server. The client sends a message to request a server to perform a task. The task is fulfilled by a

message-corresponding method of the server. In sending a message to the warehouse, Sean is said to be the client and the warehouse is said to be the server.

Benjamin, Sean, and the warehouse are three objects involved in a communication process. Benjamin is the initiator, with Sean and the warehouse as partners in the communication process. Figure 2-1 depicts a typical communication process amongst objects.

2.3 Creating Objects

In object-oriented programming, objects are created from classes. Instances of Customer objects are created from a Customer class and SalesPerson objects from a SalesPerson class.

Created object instances are individuals with their own state. To illustrate, let us consider the example of counters. A counter is a device that keeps account of the number of times an event has occurred. It has two buttons: an initialize button that resets the counter to 0, and an add button that adds 1 to its present number. Figure 2-2 shows a counter with a number 10.

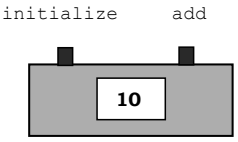


Figure 2-2: A counter.

Structurally, the first counter object can be represented as follows:

```
First Counter Object
  Attributes:
    number = 10
  Methods:
    add()            {number = number + 1}
    initialize()     {number = 0}
    getNumber()      {return number}
```

Figure 2-3 shows two more counters.

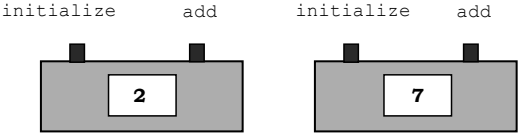


Figure 2-3: Two Additional Counters

Like the first counter, these two counters may be reset to zero and incremented through the `initialize` and `add` buttons respectively, and represented as follows:

```
Second Counter Object
  Attributes:
    number = 2
  Methods:
    add()            {number = number + 1}
    initialize()     {number = 0}
    getNumber()      {return number}

Third Counter Object
  Attributes:
    number = 7
  Methods:
    add()            {number = number + 1}
    initialize()     {number = 0}
    getNumber()      {return number}
```

All the three counters share the same definition of attributes and methods, and like in the previous examples, they can be defined by a class as follows:

```
Class Counter
  Attributes:
    number
  Methods:
    add()         {number = number + 1}
    initialize()  {number = 0}
    getNumber()   {return number}
```

The Counter class has:

- an attribute, `number`;
- an `initialize()` method that causes a counter to reset its `number` to 0.
- an `add()` method that causes a counter to add 1 to its `number`; and
- a `getNumber()` method that returns the current value of the attribute `number`.

Suppose a new object is created from the Counter class. Although the new Counter object would have the same definition of attributes and methods as the previous three counters, its attribute value may not be the same as the other counters. This suggests that the state of the counters may be different from one another.

For the newly created fourth Counter object, it has a state represented by the attribute `number` with a value of `0`, the value at initialization:

```
Fourth Counter Object
  Attributes:
    number = 0
  Methods:
    add()         {number = number + 1}
    initialize()  {number = 0}
    getNumber()   {return number}
```

Note that the attribute value of the fourth Counter object is different from the other three counters.

2.4 Summary

In this chapter, we discussed:

- Objects are defined by classes.
- Objects from the same class share the same definition of attributes and methods.
- Objects from the same class may not have the same attribute values.
- Objects from different classes do not share the same definition of attributes or methods.
- Objects created from the same class share the same definition of attributes and methods but their state may differ.
- A method is a set of operations executed by an object upon the receipt of a message.
- A message has three components: an object identifier, a method name and arguments.
- A message-receiving object is a *server* to a message-sending object known as a *client*.

2.5 Exercises

1. Distinguish the terms “Object” and “Class”.
2. Consider the scenario of buying flowers from a florist. Outline the objects in such a transaction together with the messages exchanged.
3. Given a class definition Rectangle below, describe the structure of any 3 instances of Rectangle.

```
class Rectangle {
  Attributes:
    length
    width
  Methods:
    getLength() { return length }
    getWidth()  { return width }
    draw()      { ... }
}
```

4. How would you implement the concept of class and method in a non-object-oriented programming language such as COBOL, Pascal or C?
5. Define using the following structure a class definition for cars. A car generally has abilities to start, move forward, move backward,

stop and off. A car can also return to its relative location. The starting location is a value 0.

```
class Car {
  Attributes:
    ...
  Methods:
    ...
}
```

6. Distinguish between a client and a server.
7. A client communicates with a server by sending a _____ to the server. The _____ is a call on a _____ of the server.

3

A Quick Tour of Java

Earlier, we introduced key object-oriented concepts such as objects, methods and classes and how these may be ultimately used in a computer model. In this chapter, we see how the Java programming language is used to construct our object model of the problem domain. This approach is advantageous in that it allows our model to operate or "come alive" under computer control.

3.1 Primitive Types

The Java programming language allows the creation of objects that will ultimately participate in message communication. We have seen that objects may have diverse behavior and that it is more convenient to specify objects via classification, that is, class constructs.

Before examining class definitions for user-specified objects, we should be mindful that Java also provides primitive values from which other (bigger) objects may be described in terms of and constructed from. For example, a complex number may be seen as being comprised of two numbers representing the real and imaginary parts.

The primitive types `byte`, `short`, `int` and `long` defined in the Java language allow for the representation of discrete integer values of widths 8, 16, 32, and 64 bits, respectively. These in turn correspond to the representation of numeric ranges –128 to 127, –32768 to 32767, –2147483648 to 2147483647, and –9223372036854775808 to 9223372036854775807, respectively.

The primitive types `float` and `double` allow for the representation of single and double precision floating-point real values with representational widths of 32 and 64 bits, respectively. The adopted IEEE 754 standard includes both positive and negative sign-magnitude numbers, both positive and negative zeros and infinities, and unique not-a-number representations.

Values of type `float` are of the form $s \cdot m \cdot 2^e$, where s is either +1 or –1, m is a positive integer less than 2^{24}, and e is an integer between –149 and 104. Similarly, values of type `double` have the similar form $s \cdot m \cdot 2^e$, but m is a positive integer less than 2^{53}, and e is an integer between –1075 and 970.

Finally, the primitive types `char` and `boolean` allow for 16-bit multi-byte characters and `false`/`true` boolean values, respectively.

3.2 Object Definition

Building upon the primitive values supported by the language proper, other entities to be manipulated are user-designed objects which are defined via class constructs. A class construct in Java consists of the `class` keyword followed by the class name and braces `{ }` which delimit the declaration of attributes and methods for its instances. The Counter class introduced in Chapter 2 would have the following form in Java:

```
class Counter {
  attribute and method declarations
}
```

Object attributes are, in turn, either nested component objects or primitive types used to represent the object. An *instance method* manipulates the object by altering its attribute values. The `number` attribute and `add()` method in the Counter class below are representative of an object's *state* and *operation*, respectively:

```
class Counter {
  int number;
  void add() {
    number = number +1;
  }
}
```

The `number` attribute is also known as an *instance variable* because it occurs in every object or instance of the Counter class. This further implies that an attribute in one instance is independent from that in another instance. In the same vein, a method manipulates object attributes in the same instance. This occurs when a method is invoked and the corresponding code in its body is executed. In our recent example, invoking the `add()` method of an object will increment the corresponding `number` attribute.

3.2.1 Variable Definitions

Variable definitions in Java take the form below, where the type name `T` precedes the variable name `v`:

```
T v;
```

Typing in a programming language allows the values for a variable to be anticipated. As such, appropriate storage may be set aside for these values.

There is another subtle advantage of typing in programming languages: the values associated with the variable also imply what operations are valid and applicable. For example, multiplication and division applies to numeric values but not to character values. Thus, the language compiler may flag multiplication and division of character values as erroneous.

All variables in Java are typed, allowing the compiler to verify during compilation that operations on the object associated with the variable are legitimate.

3.2.2 Methods

A method definition that occurs in a class construct is made up of two distinct portions: the method signature header and its implementation code body surrounded by the braces `{ ... }`.

The method signature portion, such as void `add()` in the Counter example, has the generic form below, where `m` is the method name, `T` its return type, with `Rn` and `pn` being parameter types and names, respectively (`n` being the number of parameters):

```
T m(R1 p1, R2 p2, ... Rn pn)
```

We have seen that a method named `m()` is invoked to correspond to a message *m* sent to the object. Consequently, the object may return a result to the message sender. The type of this value is denoted by `T`. If no result needs be returned, the keyword `void` is used instead.

The formal parameters p1, p2…pn contain the additional values sent together with the message. They have corresponding types R1, R2…Rn, and are used by the compiler to verify that the correct parameters are supplied for each method invocation. Any number of formal parameters may be specified, but the number of actual parameters in a message must match those originally specified in the method signature.

The implementation of a method consists of a block of statements surrounded by `{ }`. Often, such methods would modify the object's attributes. In the case of the `add()` method of our Counter example, it increments the variable `number`. A block consists of declarations of any local variable, expressions and control-flow constructs. These will be discussed in detail in following sections.

In the slightly expanded version of the Counter class below, an extra `initialize()` method has been added to re-initialize the Counter value so that counting can be easily restarted. This allows instances to respond to the additional *initialize* message.

```
class Counter {
  int number;
  void add() {
    number = number+1;
  }
```

```
  void initialize() {
    number = 0;
  }
}
```

If the number of times a counter is restarted is significant, we can introduce another attribute `reused` to maintain this information. Correspondingly, this attribute is incremented in the block of the `initialize()` method:

```
class Counter {
  int number = 0;
  int reused = 0;
  void add(){
    number = number+1;
  }
  void initialize(){
    number = 0;
    reused = reused+1;
  }
}
```

The previous example of the class Counter definition shows that an object may have as many attributes and methods as required to effectively model the object. In the most recent definition, objects of the class Counter have two attributes (`number` and `reused`, both with an initial value of 0 when created) and two methods (`add()` and `initialize()`).

3.3 Object Instantiation

A class construct provides a description for objects of that class, and serves as a template for objects to be created. However, no instances of the class is crested, except by calling the object allocator function `new()`. The expression `new Counter()` returns a newly created instance of the Counter class. However, in order that this new object may be referred to, it is assigned to an appropriate variable. Assuming the variable `carpark` in the fragment below, a new Counter object may be created via `new Counter()`, and then assigned to the former:

```
Counter carpark;
...
carpark = new Counter();
```

Henceforth, the newly created object may be referred to via the variable `carpark`. Where more Counter objects are needed, the object allocator function `new()` may be repeatedly invoked, and the resultant objects assigned to other variables such as `entrance` and `exitDoor`:

```
Counter entrance, exitDoor;
...
entrance = new Counter();
exitDoor = new Counter();
```

3.4 Object Access and Message Passing

Since the attributes and methods of an object are considered its characteristics, these are accessed via the qualification operator "." with respect to an object proper. Thus, the counts of the various Counters `carpark`, `entrance` and `exitDoor` are `carpark.number`, `entrance.number` and `exitDoor.number`, respectively. The total number from these counters is:

```
carpark.number + entrance.number + exitDoor.number
```

Similarly, the `initialize()` method of Counters `carpark`, `entrance` and `exitDoor` may be invoked via:

```
carpark.initialize();
entrance.initialize();
exitDoor.initialize();
```

3.5 Representational Independence

While accessing object attributes directly is permissible, it is not ideal because it couples implementation code to the current object representation. As such, any changes in object representation propagates to dependent code, resulting in high software maintenance cost.

A common object-oriented programming practice is *information hiding*—to make object representations inaccessible to clients so that modifications in (server) object representations do not propagate excessively. This decoupling of dependencies reduces software maintenance cost.

Limiting access to object representations in Java is mainly achieved by the two main constraint specifiers `private` and `public`. The former limits access of the following entity to within the class construct, while the latter makes it accessible to any client code.

```
class Counter {
  private int number = 0;
  private int reused = 0;
  public void add() {
    number = number+1;
  }
  public void initialize() {
    number = 0;
    reused = reused+1;
  }
}
```

Since constraint specifiers in the above class definition hides the internal representation of Counter objects, the resultant attributes are no longer accessible, and useless for interrogation. In this case, accessor methods `getNumber()` and `getReused()` are required, as outlined in the following code fragment. They provide

access to internal details, but without dependency overheads. Representation independence is maintained by confining access to private attributes to within the class construct. This topic is further discussed in Chapter 8.

```
class Counter {
  private int number = 0;
  private int reused = 0;
  public void add() {
    number = number+1;
  }
  public void initialize() {
    number = 0;
    reused = reused+1;
  }
  public int getNumber() { return number; }
  public int getReused() { return reused; }
}
```

3.6 Overloading

Attribute names of a class may be the same as those in another class since they are accessed independently. An attribute *x* in a class does not necessarily have any semantic bearing with another as they have different scopes, and does not preclude using the same attribute there.

Within a Java class construct, methods may share the same name as long as they may be distinguished either by:

- the number of parameters, or
- different parameter types.

This criterion requires a message with associated parameters to be uniquely matched with the intended method definition.

If we had wanted a Counter to be incremented other than by 1, we could define another `add()` method that takes an integer parameter instead:

```
class Counter {
  private int number = 0;
  private int reused = 0;
  public void add() {
    number = number+1;
  }
  public void add(int x) {
    number = number+x;
  }
  public void initialize() {
    number = 0;
    reused = reused+1;
  }
  public int getNumber() { return number; }
  public int getReused() { return reused; }
}
```

If `carpark` had been assigned an instance of Counter, `carpark.add()` would invoke the first method to increment by 1, while `carpark.add(2)` would invoke the new one just defined.

3.7 Initialization and Constructors

Currently, object creation and initialization are seen as distinct operations. The abstraction in object-oriented programming languages often allows these two operations to be combined implicitly. As such, constructors may be seen as unique methods invoked implicitly when an object instance is created. Implicit initialization relieves the programmer from performing this important function, but more importantly prevents uninitialized objects as a result of absent-minded programmers. Carefully designed constructors allow for object invariants to be maintained regardless of how they were created.

Apart from having the same name as the class, and not having a return result type, a constructor is not different from a method. It has similar syntax for its parameters and implementation body.

In place of attribute initialization, our next Counter example uses a constructor method. This offers additional functionality compared with the former approach.

```
class Counter {
  private int number, reused;
  public void add() {
    number = number+1;
  }
  public void initialize() {
    number = 0;
    reused = reused+1;
  }
  public int getNumber() { return number; }
  public int getReused() { return reused; }
  Counter() { number = 0; reused = 0; }
}
```

While this change is not significant in our trivial example, constructors allow more flexibility such as the execution of arbitrary expressions and statements when compared with static attribute initializers. As with methods, constructors may also be overloaded. This provides for varied ways for objects to be created and initialized.

The additional new overloaded constructors in the new class definition below allows for various initial values for `number` and `reused` other than just 0:

```
class Counter {
  private int number, reused;
  public void add() {
    number = number+1;
  }
  public void initialize() {
    number = 0;
    reused = reused+1;
  }
```

```
  public int getNumber() { return number; }
  public int getReused() { return reused; }
  Counter() { number = 0; reused = 0; }
  Counter(int x) { number = x; reused = 0; }
  Counter(int x, int y) { number = x; reused = y; }
  Counter(float z) { number = (int) z; reused = 0; }
}
```

3.8 Expressions, Statements, and Control-flow Mechanisms

We saw earlier that a method definition consists of the method signature and its implementation body. As an object responds to messages by executing code in the method body to affect changes in its state, assignment is a very common operation.

```
v = E;
```

Assignment consists of a left-hand variable that will contains or "holds" the value specified via the right-hand expression. It may be a literal value such as `3`, a variable that holds the intended value such as `number`, or an operator with appropriate operands, such as `x+4`, or even `r.f` or `y*p(5)`. In the same way that `+` is an operator, `.` and `()` are also operators. The last expression involves nested expressions: the result of `p(5)` is used in multiplication.

3.8.1 Operators

We first examine the operators in Java.

(a) Arithmetic Operators

The arithmetic operators in Java include the common addition "+", subtraction "−", multiplication "*" and division "/" operations.

```
int a = 13;
int v = 7;

a+v // returns result 20
a−v // returns result 6
a*v // returns result 91
a/v // returns result 1
```

These operators apply to numeric operands, and return the type of operands. When operands are mixed, the widest is used to prevent unexpected truncation.

```
float b = 13;
int w = 7;

b+w // returns result 20.0
b−w // returns result 6.0
```

```
b*w // returns result 91.0
b/w // returns result 1.8571428
```

The "%" operator returns the remainder of integer division.

```
int a = 13;
int v = 3;

a/v // returns result 4
a%v // returns result 1
```

When used as a unary operator, "–" negates the numeric operand.

```
int a = 13;

-a // returns result -13
```

(b) Logical Operators

The logical operators in Java include the standard *and* "&&", *or* "||" and *not* "!". Each operator returns a boolean result:

&& returns true if both operands are true.

x	y	x && y
false	false	false
false	true	false
true	false	false
true	true	true

|| returns true if at least one operand is true.

x	y	x || y
false	false	false
false	true	true
true	false	true
true	true	true

! returns true if the single operand is false.

x	! x
false	true
true	false

(c) Relational Operators

The *equality* "==" and *inequality* "!=" operators in Java operate on all values and objects to return a boolean result.

```
int h = 4;
int j = 4;
int k = 6;
Counter m = new Counter();
Counter n = new Counter();

h == j // returns true, h and j have the same value
h == k // returns false
k == k // returns true

m == n // false, m and n are different objects even if they have the
       //  same constituents
n == n // true
```

The following relational operators in Java operate on numeric values.

<	less than
>	greater than
<=	less than or equal
>=	greater than or equal

```
int h = 4;
int j = 4;
int k = 6;

h < k  // returns true
h < j  // returns false
h <= j // returns true
```

(d) Bitwise Operators

The following bitwise operators in Java operate on corresponding bits of `byte`, `short`, `int` and `long` values.

&	bitwise "and"
^	bitwise exclusive "or"
|	bitwise inclusive "or"
~	bitwise one's complement
>> <<	right and left bitwise shift
>>>	right bitwise unsigned shift

```
int a = 15; //   binary    000001111
int v = 34; //   binary    000100010

a & v       // returns result 2    000000010
a ^ v       // returns result 45   000101101
a | v       // returns result 47   000101111
~a          // returns result -16
            //   11111111111111111111111111110000

v >> 3      // returns result 4    000000100
v << 3      // returns result 272  100010000

~a >> 3     // returns -2
            //   11111111111111111111111111111110
~a >>> 3    // returns 536870910
            //   01111111111111111111111111111110
```

(e) Assignment

Having seen the basic means of providing new values to variables, assignment "=" is more correctly viewed as an *operator* rather than a statement. In addition to assigning the right-side value to the left-side variable, it also returns the value assigned. As such, the assignment expression may appear in the context of an enclosing expression. (In the example below, the result of the assignment operator to variable `a` is not used.)

```
int a, b;                    implies    int a, b;
a = b = 2;                              a = (b = 2);
```

The code fragments above assign `2` to `b` and the result of `2` is assigned to `a`. This is because unlike the common arithmetic operators which associates from left-to-right, the assignment operator associates from right-to-left. This is highlighted in the next section on operator precedence.

In general, an assignment operator results in a side-effect since it changes the value of the variable being assigned. Other related assignment operators have the special form "*op*=", where *op* is a typical operator.

`x` *op*`= f;` has the same meaning as `x = x` *op* `f;`

In the above equivalent form, *op* may be operators such as +, -, *, /, %, >>, <<, &, ^ or |. Thus +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, and |= are also valid assignment operators.[1]

The other two operators related to assignment are auto-increment "++" and auto-decrement "--". Since they are used in either postfix and prefix forms, four scenarios are as illustrated in the code fragments below.

The postfix form (such as `f++`) returns the result before the increment/decrement operation, whereas the prefix form (such as `++f`) returns the results after the increment/decrement operation. In the code fragments below, `f` is either incremented or decremented. However, `g` is either assigned a "pre" or a "post" value depending on whether the prefix or postfix forms are used, respctively.

```
int f, g;

f = 6;
g = f++;
// g has 6
// f has 7
```

```
int f, g;

f = 6;
g = ++f;
// g has 7
// f has 7
```

```
int f, g;

f = 6;
g = f--;
// g has 6
// f has 5
```

```
int f, g;

f = 6;
g = --f;
// g has 5
// f has 5
```

(f) Conditional Expression

The conditional expression operator `? :` returns one of two values depending on the boolean condition. For example, the expression *A*`?`*B*`:`*C* returns the value of *B* if *A* is true, else the value of *C* is returned.

(g) Typecast

The typecast operator `(`*type*`)`*E* performs 2 basic functions at run-time depending on the source expression *E*. For a numeric type (such as `int`, `float`, or `double`), it converts the value to another of the specified type.

[1] There is a subtle difference between `x[i++] += 4` and `x[i++] = x[i++] + 4`, in that `i++` is evaluated once in the former but twice in the latter.

```
int k = 5;
double d = 4.16;

k = (int) d*k;
```

The resultant expression of `d*k` is double value, and the typecast operation converts it to an integer. For variable, the operator confirms that the object referenced is compatible with the specified class.

```
Object x = new Counter();
Counter c;

c = x;                    // illegal since not all Objects are Counters
c = (Counter) x;          // legitimate because x is at run-time
                          //     verified to reference a Counter
```

(h) Precedence and Associativity

Since most operators have operands that can be (nested) expressions, *operator precedence* and *associativity rules* are necessary to define the evaluation order. For example in evaluating "`a+b*c`", "`b*c`" is evaluated before its result is added to `a` because multiplication "`*`" has higher precedence than addition "`+`".

Table 3.1 summarizes the operators discussed so far.

The operators at the top of the table have higher precedence than those at the bottom. It is as though precedence pulls operands, so that operators with a higher precedence are evaluated before those with lower precedence. All binary operations are left-associative, except assignment operators which associate right-to-left.

Table 3-1: Operator precedence.

Operator
`[] . (`*params*`) E++ E--`
unary operators: `-E !E ~E ++E --E`
`new (`*type*`)E`
`* / %`
`+ -`
`>> << >>>`
`< > <= >=`
`== !=`
`&`
`^`
`|`
`&&`
`||`
`? :`
`= += -= *= /= %= >>= <<= &= ^= |=`

Precedence allows the expression "a>>b+c" to be unambiguously interpreted as "a>>(b+c)", and not "(a>>b)+c". Similarly, "!m&&n" is interpreted as "(!m)&&n" and not "!(m&&n)".

Associativity rules come into effect when equal precedence levels do not help in resolving evaluation order. Due to associativity rules (which is left-to-right for "/" and "*", that is, evaluating the left operator and then right), "a/b*c" is interpreted as "(a/b)*c" and not "a/(b*c)".

Similarly, due to right-to-left associativity, "~y++" is interpreted as "~(y++)" instead of "(~y)++".

3.8.2 Expression Statements

The simplest and most common statements in Java are expression statements, which consist of an assignment expression, method invocation or instance creation followed by a semicolon. The following are expression statements:

```
int a, b;
T c;

a = 1;          // assignment expressions
a++;
c = new T();

new T();        // instance creation

c.m();          // method invocation
```

3.8.3 Control-flow Statements

Apart from the simple statements, there are control-flow statements that affect the execution order of statements. These statements are commonly grouped under conditional and iterative statements.

(a) Conditional Statements

Conditional statements allow for conditions to be attached to a statement as to whether it will be executed. The most basic form is the `if` statement. In the code fragment below, the statement *S* is executed only if the boolean condition *E* evaluates to `true`:

```
if (E)
   S;
```

A slight variation is the `if-else` statement which allows for an *either-or* choice. If the boolean condition *E* evaluates to is `true` and *S* is executed, then *R* would not. If *S* is not executed, then *R* would be:

```
if (E)
   S;
else
   R;
```

Apart from one-way and two-way branches in flow-control, the `switch` statement allows for multi-way selection:

```
switch (E) {
  c1: S1;
      break;
  c2: S2;
      break;
  c3: S3;
  c4: S4;
      break;
  default: Sd;
}
```

Generally, the `switch` statement allows for the execution of a choice of statements depending on the expression *E*: *S1* when *E* evaluates to the constant *c1*, *S2* when *c2*, …, etc., the mappings given by each switch limb. The `break` statement causes flow-control to leave the switch statement immediately.

In the case of the execution of statement *S3* when *E* evaluates to *c3*, the absence of a `break` statement causes execution to continue to *S4* instead. The `default` limb is used when the evaluated value of `E` does not match any constant values in the limbs.

(b) Iterative Statements

Iterative statements allow for constituent statements to be executed repeatedly. In the most basic way, the body of the `while`-statement below is repeatedly executed when the boolean condition *E* is true. The loop terminates when *E* is `false`, after which execution proceeds to the next statement:

```
while (E)
   S;
```

The `while`-statement is known as a *pretest* loop since the constituent *S* is only executed if the condition *E* evaluates to is `true`. Thus, if *E* was false in the first instance, the statement *S* is never executed.

On the other hand, the `do-while` statement is a *posttest* loop. *R* is first executed and subsequently while the boolean expression *F* evaluates to `true`, *R* is executed again. Again, the loop terminates when *F* evaluates to `false`. Thus, this control flow construct will execute *R* at least once.

```
do {
   R;
} while (F);
```

3.9 Blocks

A block, indicated by `{ }`, may occur at any location where a statement is valid. It is considered the sequential construct as the group of statements it surrounds is treated as a single statement or processing unit.

Thus, while the various control-flow constructs merely show a single statement as the constituent body, a block may be used where such constructs should contain more than one statement. For example, *factorial* may be computed by the following `while`-statement:

```
f = 1;
while (k > 1) {
  f = f*k;
  k--;
}
```

Blocks allow for control-flow constructs to be nested within bigger constructs. In the code fragment below, the nested `if-else` statement allows for the number of even numbers and sum of odd numbers in the range to be computed.

```
even = 0; sumOfOdd = 0;
f = 1;
while (k > 1) {
  f = f*k;
  if (k % 2 == 0)
    even++;
  else
    sumOfOdd = sumOfOdd + k;
  k--;
}
```

3.9.1 Local Declarations

In treating a statement sequence as a single statement, a block may also be thought of as a sub-machine which fulfills a specific task. As such, the scope of local variable declarations is the rest of the block in which the declaration occurs. This allows declarations and associated references to be localized, thereby aiding maintainability.

```
while (k > 1) {
  f = f*k;
  if (k % 2 == 0) {
    double d = 4.5;
    ....
    even++;
  } else {
      long d = 23546;
      ...
      sumOfOdd = sumOfOdd + k;
  }
  k--;
}
```

The code fragment above is legitimate because both local declarations of `d` have been confined to their respective nested blocks – `d` is a double in the first block, while `d` is a long in the second.

With instance variables, a local declaration has the same form below, where `T` is the declared type of variable `v`.

```
T v;
```

For notational convenience, declarations may have two variations:

- a list of variables with the same type separated by commas; and
- an initial value may be provided via an expression following an assignment operator

```
T v, w = n(), z;
```

3.10 More Control-flow Statements

Three other control-flow statements are commonly used: `for`-statement, `break` and `continue`. The `for`-statement is often used as a counter-controlled loop to iterate through a loop a fixed number of times, even though it does not have explicit mechanisms for counter control. For example, the earlier *factorial* example could be re-coded as follows:

```
for (f = 1; k > 1; k--)
  f = f*k;
```

The generic form of the `for`-statement

```
for (Q; R; S)
  T;
```

is often easier thought of as a transformed `while`-statement, where *Q*, *T*, and *S* are the initializer, conditional and re-initializer expressions:

```
Q;
while (R) {
  T;
  S;
}
```

The `break`-statement was encountered when the `switch`-statement was discussed. Its more generic function is to transfer control out of the innermost `switch`, `while`, `do` or `for`-statement. This is why using the `break`-statement ensures that only the statements associated with the case-limb are executed.

For the situation with the `while`, `do` or `for`-statements, the `break`-statement allows for a quick exit from the iteration. In many situations, its use can result in a simpler program structure. For example, the following two code fragments have similar effects.

```
finished = false;
while (E && !finished) {
  S;
  if (F) {
    U;
    finished = true;
  }
  if (!finished)
    T;
}
```

```
while (E) {
  S;
  if (F) {
    U;
    break;
  }
  T;
}
```

Finally, the `continue`-statement transfers control to the beginning of the innermost iterative loop so as to reevaluate the boolean condition for the next iteration. Unlike, the `break`-statement, control-flow does not exit from the loop. As such, the following two code fragments have similar effects.

```
while (E) {
  S;
  if (F) {
    U;
    continue;
  }
  T;
}
```

```
skip = false;
while (E) {
  S;
  if (F) {
    U;
    skip = true;
  }
  if (!skip)
    T;
  else
    skip = false;
}
```

While the differences between program structures in the above examples may seem mild for the `break` and `continue` statements to be useful, it is more pronounced for program structures that are deeply nested.

3.11 Arrays

Just as objects are created dynamically (i.e., it happens at run-time during program execution), arrays in Java are similarly created. The size of an array need not be specified or computed during compilation.

An array is thus declared using the subscript operator, but without indication of the upper bound:

```
Counter gates[];
```

An array is created via the `new` operator, but with the array size within square brackets:

```
gates = new Counter[8];
```

The array size associated with `gates` is 8, but this does not imply eight Counter objects. Instead, it is important to understand a Counter array as being similar with multiple variables, that is, it is an object that does not further contain Counter objects but merely references eight potential Counter objects.

Thus, individual array elements must be created explicitly:

```
Counter gates[];
gates = new Counter[8];
for (int i=0; i<8; i++);
  gates[i] = new Counter();
```

3.12 Result Returned by Method

Now that we have examined the statement constructs in Java, we return to see how a method may return a result to its sender. A method in a class definition has the following general form, with the `return`-statement returning control-flow back to the message sender:

```
T foo(gT g, hT h ...)
{
   // local definitions

   // statements

   return v;
}
```

The value returned `v`, must be of the type `T` as indicated in the method signature. If the sender does not require any results, the keyword `void` should be used as the return type. In this case, the returning expression `v` would be omitted in the `return` statement. The `return`-statement need not be the last statement in the block as implied in the previous example. In a non-trivial structure, multiple `return`-statements might be used as in the next example, but the programmer must evaluate if the situation improves program structure and readability.

```
T foo(gT g, hT h ...)
{
  for (E; F; G)
    if (H)
      return v;
    else if (J) {
      b;
      return w;
    }
  return x;
}
```

3.13 Summary

In this chapter, we discussed:

- Primitive types in Java.
- `class` constructs.
- Definition of instance variables and methods.
- Object instantiation.
- Message passing and expressions.
- Statements and control-flow mechanisms.

Generally, these constructs are representative of what Java offers. The `class` construct is key in Java because it allows for objects to be defined to model the problem domain. Below that, variables and methods are defined, which correspond to data and code. Code abstraction result in hierarchical statement blocks (with optional local variables) and control flow mechanisms. Figure 3-1 illustrates this hierarchy.

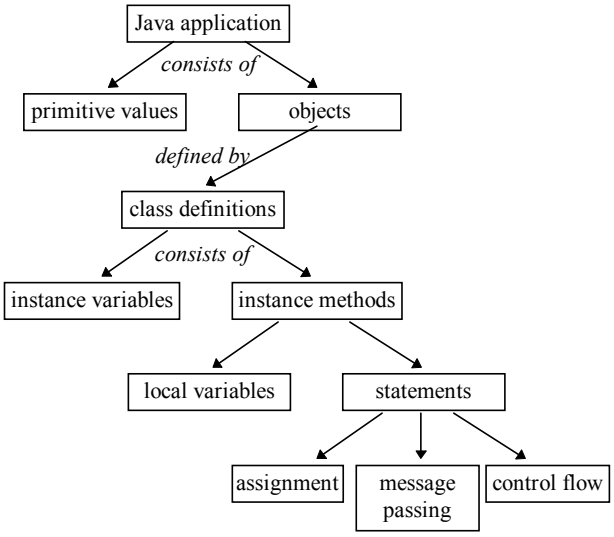


Figure 3-1: Hierarchical relationships of Java constructs.

3.14 Exercises

1. Which of the following are valid variable names in Java?

   _object
   object–oriented

object_oriented
object.oriented
$java
java
integer
string
Int
933
fm90.5
1fm

2. Define a Square class with the length of its side as an instance variable. Include an appropriate constructor method and methods to enlarge an instance as well as compute its area.
3. Using the Square class in Question 2, create ten randomly sized squares and find the sum of their areas. (The method `Math.random()` returns a random number between 0.0 and 1.0 each time it is invoked.)
4. Add the functionality for a Square object to draw itself via ASCII characters. For example, a Square of length 4 can be drawn as:

```
****
*  *
*  *
****
```

or:

```
XXXX
X++X
X++X
XXXX
```

The `System.out.print()` and `System.out.println()` methods may be useful.

5. Find a number with nine digits $d_1d_2d_3,\ldots, d_9$ such that the sub-string number $d_1, \ldots, d_n$ is divisible by n, $1<=n<=9$. Note that each of the digits may be used once.

4

Implementation in Java

In Chapter 3, we demonstrated how object-oriented concepts can be implemented via notations in the Java programming language. For validation purposes, these concepts allow objects in our system to be operational. We now proceed to see how they are practically applied to example problems as typical programs.

4.1 Calculator

We first consider how a simple calculator with the basic four arithmetic functions, as illustrated in Figure 4-1, may be implemented. Most generic machines allow adding 11 to 13 to be accomplished via the buttons [1][3][+][1][1][=].

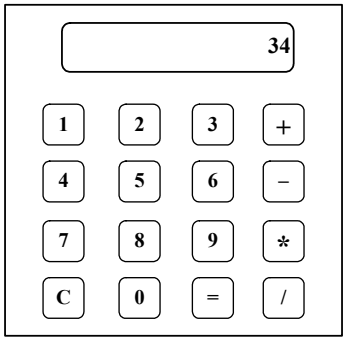


Figure 4-1: Four-function calculator.

For a simple implementation, we will initially not concern ourselves with the nonessentials of external looks (e.g., keypad layout or casing specifics as in real calculators), but instead concentrate on the core calculating engine. Enhancements involving mechanisms for the user interface may be subsequently considered. This is consistent with the software engineering principle of *abstraction*.

Conceptually, the core calculator engine may be viewed as being comprised of registers to hold values, together with built-in operators such as addition and subtraction to manipulate such values. Having operations involving two operands, the calculator needs at least two registers.

Four arithmetic operations imply at least four operators. A *compute* operator is required to get its execution started, together with a *clear* operator to prepare the registers for new computations. These correspond to the equal and clear keys on a calculator. Lastly, the digits form the basic input mechanism for numeric operands. We would assume that the *display* operator will retrieve a value for display on the calculator panel.

Object initialization may be easily accomplished via constructor methods. In this case, a `CalculatorEngine` object is initialized via invoking the *clear* operator. The resultant Java code skeleton for `CalculatorEngine` objects with this basic representation is shown in Listing 4-1. Note that all operators easily correspond to instance methods, as well as to buttons on the face on a conventional calculator. Code represented by ellipses will be elaborated in due course.

```
class CalculatorEngine {
  int value;
  int keep;       // two calculator registers
  void add()         { ... }
  void subtract()    { ... }
  void multiply()    { ... }
  void divide()      { ... }
  void compute()     { ... }
  void clear()       { ... }
  void digit(int x)  { ... }
  int display()      { ... }
  CalculatorEngine() { clear(); }
}
```

Listing 4-1: CalculatorEngine skeleton.

4.1.1 The `clear()` Method

The `clear()` method initializes the CalculatorEngine object to the state for a calculator to begin the key sequence for next calculation. It is thus intuitive to set the variables `value` and `keep` to `0`.

```
void clear() {
  value = 0;
  keep = 0;
}
```

4.1.2 The `display()` Method

To implement the `display()` method to provide computation results, we must first clarify the purposes of the instance variables `value` and `keep`. The former is updated as a result of new inputs from numeric keys or the result of an operator, and thus is used to update the display area.

As expressions are keyed in using an infix notation (e.g., 1 3 + 1 1 =), the first operand must be stashed away before it is overwritten by the entry of the second operand. The `keep` instance variable serves this purpose.

```
int display() {
 return(value);
}
```

4.1.3 The `digit()` Method

The `digit()` method accumulates digits keyed in via the numeric keypad. A new digit shifts existing digits in the `value` instance variable one place to the left. This manipulation is accomplished by multiplication by 10 and followed by addition of the last digit.

```
void digit(int x) {
  value = value*10 + x;
}
```

While this method stands out amongst the other methods as it expects an integer parameter to indicate which numeric key was pushed, it can be circumvented by using wrapper methods such as `zero()`, `one()`, `two()`, `three()`, ...`nine()`.

```
void one() {
  digit(1);
}

void two() {
  digit(2);
}
...
```

4.1.4 Operator Methods

The infix mode of the *add*, *subtract*, *multiply* and *divide* operators requires that the specified operation be stashed away to be applied **after** input of the second operand. For this purpose, we define another instance variable `toDo` which records the action to be associated with the next compute operation.

```
char toDo;
void add() {
  keep = value;   // keep first operand
  value = 0;      // initialise and get ready for second operand
  toDo = '+';     // this is what we should do later
}
```

```
void subtract() {
  keep = value;   // keep first operand
  value = 0;      // initialise and get ready for second operand
  toDo = '-';     // this is what we should do later
}
```

Since all the binary operations have the same form, it is again natural to adopt abstraction techniques to relocate common code in a `binaryOperation()` method:

```
void binaryOperation(char op) {
  keep = value;   // keep first operand
  value = 0;      // initialize and get ready for second operand
  toDo = op;
}

void add()      { binaryOperation('+');  }
void subtract() { binaryOperation('-');  }
void multiply() { binaryOperation('*');  }
void divide()   { binaryOperation('/');  }
```

Lastly, we conclude with the compute operation which provides the answer to applying the operator in `toDo` on the operands `value` and `keep`.

```
void compute(){
  if (toDo == '+')
    value = keep + value;
  else if (toDo == '-')
    value = keep - value;
  else if (toDo == '*')
    value = keep * value;
  else if (toDo == '/')
    value = keep / value;
  keep = 0;
}
```

4.2 Code Execution

In the previous chapter, we learned that the `new` operator creates an object instance of the class that it is applied to. Thus,

```
CalculatorEngine c = new CalculatorEngine();
```

creates an instance and associates it with the variable `c`. Subsequently, the code sequence

```
c.digit(1);
c.digit(3);
c.add();
c.digit(1);
c.digit(1);
c.compute();
```

computes the value of the expression 13 + 11. For verification purposes, the Java API (Application Program Interface) method `System.out.println()` may be used to produce output on the screen:

```
System.out.println(c.display());
```

There is, however, a slight snag: the `CalculatorEngine` object instance is the only object in existence, yet which object would send it messages to compute expressions? Or even more fundamental, at the very commencement of program execution when no objects existed, how was the first object created?

Java solves this issue through the introduction of *class methods*, which are invoked with respect to the class they are associated with rather than object instances. More specifically, the body of the static method named `main()` is the first code sequence to be executed. As such, the previous code sequence must be brought into `main()` and rearranged as follows:

```
public static void main(String arg[]) {
  CalculatorEngine c = new CalculatorEngine();
  c.digit(1);
  c.digit(3);
  c.add();
  c.digit(1);
  c.digit(1);
  c.compute();
  System.out.println(c.display());
}
```

The various code fragments may be brought together within a class construct in the file `CalculatorEngine.java` as shown in Listing 4-2.

```
class CalculatorEngine {
  int value;
  int keep;       // two calculator registers
  char toDo;

  void binaryOperation(char op)  {
    keep = value;   // keep first operand
    value = 0;      // initialize and get ready for second operand
    toDo = op;
  }

  void add()      { binaryOperation('+'); }
  void subtract() { binaryOperation('-'); }
  void multiply() { binaryOperation('*'); }
  void divide()   { binaryOperation('/'); }

  void compute() {
    if (toDo == '+')
      value = keep + value;
    else if (toDo == '-')
      value = keep - value;
    else if (toDo == '*')
      value = keep * value;
    else if (toDo == '/')
      value = keep / value;
    keep = 0;
  }
```

```
  void clear() {
    value = 0;
    keep = 0;
  }

  void digit(int x)  {

    value = value*10 + x;
  }

  int display() {
    return(value);
  }

  CalculatorEngine() { clear(); }

  public static void main(String arg[]) {
    CalculatorEngine c = new CalculatorEngine();
    c.digit(1);
    c.digit(3);
    c.add();
    c.digit(1);
    c.digit(1);
    c.compute();
    System.out.println(c.display());
  }
}
```

Listing 4-2: `CalculatorEngine` class.

With the Java Development Kit (JDK) appropriately installed, it may be compiled via:

```
$ javac CalculatorEngine.java
```

where `CalculatorEngine.java` is the name of the file containing the Java source and **`$`** is the system's command line prompt. Similarly, execution of the resultant Java byte code may proceed via:

```
$ java CalculatorEngine
```

4.3 Simple User Interface

While the code in `static void main()` does execute to show the behavior of a `CalculatorEngine` object instance, it is an absolutely clumsy situation. Each evaluation of a new arithmetic expression requires editing code and recompilation. Ideally, we should be compiling the source once, but inputting different expressions for evaluation.

It is common to have an user interface object to work cooperatively with the `CalculatorEngine`. This separation of concerns allow for the `CalculatorEngine` to

be independent of interface issues. We will initially consider a line-mode user interface and subsequently enhance it for a windowing environment.

To this end, a `CalculatorInterface` object fulfills this role. It has the role of a middleman that does not work too much, but instead accepts input and passes it onto the `CalculatorEngine`. Similarly, feedback from the `CalculatorEngine` is collected and becomes output for the `CalculatorInterface` object.

The implementation of `CalculatorInterface` consists of an initializing phase where a `CalculatorEngine` object is bound to an `CalculatorInterface` object, and an execution phase which performs the necessary dispatching. These are implemented by the constructor and `run()` methods of `CalculatorInterface` respectively, as illustrated in Listing 4-3.

```
import java.io.*;

class CalculatorInput {
  BufferedReader stream;
  CalculatorEngine engine;

  CalculatorInput(CalculatorEngine e) {
    InputStreamReader input = new InputStreamReader(System.in);
    stream = new BufferedReader(input) ;
    engine = e;
  }

  void run() throws Exception {
    for (;;) {
      System.out.print("[" + engine.display() + "]");
      String m = stream.readLine();
      if (m == null) break;
      if (m.length() > 0) {
        char c = m.charAt(0);
        if (c == '+') engine.add();
        else if (c == '-') engine.subtract();
        else if (c == '*') engine.multiply();
        else if (c == '/') engine.divide();
        else if (c >= '0' && c <= '9') engine.digit(c - '0');
        else if (c == '=') engine.compute();
        else if (c == 'c' || c == 'C') engine.clear();
      }
    }
  }

  public static void main(String arg[]) throws Exception {
    CalculatorEngine e = new CalculatorEngine();
    CalculatorInput x = new CalculatorInput(e) ;
    x.run();
  }
}
```

Listing 4-3: `CalculatorInput` class.

While the code for `CalculatorInterface` relies on facilities for exception and input/output handling that have not been described yet, these may initially be ignored. Nevertheless, the code serves two immediate purposes here:

- It demonstrates the context of a *test harness* and how it is easily constructed to aid incremental development.
- It shows the synergistic cooperation of two objects with distinct concerns in an object-oriented design environment.

Until these topics are discussed in Chapters 9 and 10, it is not harmful that at present, they be taken by faith. The "`throws Exception`" signature suffix allows for Java exceptions to be for the moment ignored. It is useful for modular and secure programming methodology. It also suffices that the `BufferedReader` class facilitates input, and that the `readLine()` method allows an input line to be read.

The new user interface class may be compiled via

```
$ javac CalculatorInput.java
```

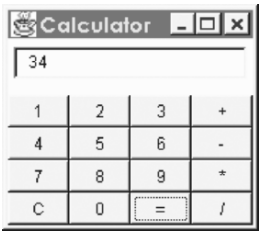
This provides the added flexibility of arbitrary computations via keyboard input sequences. The calculator display is indicated within square brackets "`[ ]`":

```
$ java CalculatorInput
[0]1
[1]3
[13]+
[0]1
[1]1
[11]=
[24]
```

4.4 Another Interface for CalculatorEngine

The separation of concerns between `CalculatorEngine` and `CalculatorInterface` allows for the former to be reused in different environments. To show the ease of code reusability when a neat modular structure is adopted, another user-interface framework to work with `CalculatorEngine` is introduced in this section.

Similar to `CalculatorInput`, `CalculatorFrame` provides an environment for a `CalculatorEngine` object to execute. The major difference is that `CalculatorFrame` caters for a windowing environment, and gives the illusion that the calculator "hides" behind the frame.



Windowing facilities in Java will be discussed in Chapter 13. However, this graphical calculator example is still appropriate since its objective is to show the benefits of modular code and reusable API libraries in Java. Code in the constructor method sets up a calculator frame with buttons and a display at appropriate locations. Using a graphical user interface in this instance is fairly straightforward since mouse clicks on calculator buttons are mapped to `actionPerformed()` method. As such, code that performs the necessary dispatching to `CalculatorEngine` shown in Listing 4-4 is similar to that in the `run()` method in `CalculatorInput`.

```
import java.awt.*;
import java.awt.event.*;

class CalculatorFrame extends Frame implements ActionListener {

   CalculatorEngine engine;
   TextField display;

   WindowListener listener = new WindowAdapter() {
     public void windowClosing(WindowEvent e)  { System.exit(0); }
   };
   CalculatorFrame(CalculatorEngine e) {
     super("Calculator");
     Panel top, bottom; Button b;

     engine = e;
     top = new Panel();
     top.add(display = new TextField(20));
     bottom = new Panel();
     bottom.setLayout(new GridLayout(4,4));
     bottom.add(b = new Button("1")); b.addActionListener(this);
     bottom.add(b = new Button("2")); b.addActionListener(this);
     bottom.add(b = new Button("3")); b.addActionListener(this);
     bottom.add(b = new Button("+")); b.addActionListener(this);
     bottom.add(b = new Button("4")); b.addActionListener(this);
     bottom.add(b = new Button("5")); b.addActionListener(this);
     bottom.add(b = new Button("6")); b.addActionListener(this);
     bottom.add(b = new Button("-")); b.addActionListener(this);
     bottom.add(b = new Button("7")); b.addActionListener(this);
     bottom.add(b = new Button("8")); b.addActionListener(this);
     bottom.add(b = new Button("9")); b.addActionListener(this);
     bottom.add(b = new Button("*")); b.addActionListener(this);
     bottom.add(b = new Button("C")); b.addActionListener(this);
     bottom.add(b = new Button("0")); b.addActionListener(this);
     bottom.add(b = new Button("=")); b.addActionListener(this);
     bottom.add(b = new Button("/")); b.addActionListener(this);
     setLayout(new BorderLayout());
     add("North", top);
     add("South", bottom) ;
     addWindowListener(listener) ;
     setSize(180, 160) ;
     show();
   }
   public void actionPerformed(ActionEvent e) {
     char c = e.getActionCommand().charAt(0);
     if (c == '+') engine.add();
     else if (c == '-') engine.subtract();
     else if (c == '*') engine.multiply();
```

```
      else if (c == '/') engine.divide();
      else if (c >= '0' && c <= '9') engine.digit(c - '0');
      else if (c == '=') engine.compute();
      else if (c == 'C') engine.clear();
      display.setText(new Integer(engine.display()).toString());
    }
    public static void main(String arg[]) {
      new CalculatorFrame(new CalculatorEngine());
    }
}
```

Listing 4-4: `CalculatorFrame` class.

4.4.1 Event-Driven Programming

While much code may be presently skipped, it is of great encouragement to readers that the API libraries allow for windowing applications to be developed with minimal user-code. Much code occur in the constructor and `actionPerformed()` methods, which sets up the calculator buttons and respond to mouse clicks.

The code in the `CalculatorFrame` class looks somewhat strange because it is not completely procedural in its specification. In particular, while the body of the `actionPerformed()` method resembles that in `run()` in `CalculatorInput`, the former is not explicitly invoked from within the class, such as from `static void main()` (as was the case for `CalculatorInput`) .

Procedural programming is the paradigm where actions are specified in a step-by-step sequence such as a baking recipe. Within each Java method, code is specified procedurally and the execution order may be easily determined.

In event-driven programming, code fragments are instead associated with events and invoked when these events occur. In a typical graphical environment with windowing facilities, events correspond to mouse movements, mouse clicks and keystrokes from the keyboard. It is impossible to determine or plan in advance what course of actions users might take to accomplish a certain task. Instead, we associate code fragments with significant events so that their side-effects will be appropriate response to such external events.

In an object-oriented system, methods are convenient units of code, and are used to receive stimuli from external devices. In Java, an `ActionListener` keeps a lookout for events involving mouse clicks. This is relevant to our `CalculatorFrame`, and we in turn implement the `actionPerformed()` method as a trigger point for mouse clicks on calculator buttons.

Thus, for each push as a calculator button, `actionPerformed()` is invoked, and it uses the `getActionCommand()` to identify which button. The framework for windowing using the AWT API will be further elaborated in Chapter 13.

Similarly, `WindowAdaptor` is used to monitor events involving windows, and `windowClosing()` is the corresponding method which is invoked when the user clicks to close the calculator window. Execution is terminated via `System.exit(0)`.

4.5 Summary

This chapter demonstrates Java syntax and semantics covered earlier in Chapter 3. The calculator case study example shows how:

- a class construct may be developed and compiled for execution by the virtual machine;
- execution commences with the `static void main()` method;
- objects are instantiated and appropriately coordinated for cooperative message passing to model;
- the `BufferedReader` class is used for input;
- the AWT package is used for GUI programming using frames and involving event-handling.

4.6 Exercises

1. The operators for `CalculatorEngine` are binary and require two operands. How would unary operators that require one operand be incorporated? Modify the `CalculatorEngine` class to add the following capabilities.
    - `squareOf`
    - `factorial`
2. Choose two interface characters most apt for `squareOf` and `factorial` and incorporate the additional capability into the `CalculatorInput` class.
3. Rearrange the layout of the calculator panel in the `CalculatorFrame` class to accommodate the new capabilities, and modify the appropriate event-handlers to take advantage of these functions.
4. The `CalculatorFrame` class produces the result of 247 corresponding to the key input sequence [1][3][+][1][1][=][7]. Explain the reason for this observation, and suggest how it may be corrected.

5

Classification, Generalization, and Specialization

Objects with similar definitions have been grouped and defined into classes. The act of identifying and categorizing similar objects into classes is known as *classification* in object-oriented modeling. In this chapter, we will examine how objects are grouped into classes and how the relationships of classes can be organized into a class hierarchy using abstraction mechanisms *generalization* and *specialization*. In addition, we will discuss the concept of *superclass* and *subclass* as a prelude to discussing generalization and specialization.

5.1 Classification

In Table 5-1, there are 18 instances of animal. Each entry has a name and a short description. Some of the animals share common information. We will group the animals based on their commonality with one another.

We begin with Mighty, Flipper, Willy, Janet, Jeremy, Bunny, and Smudge. These objects are grouped together into the Mammal category because they share some common information typical of a mammal:

- their young are born alive;
- they are warm-blooded;
- they breathe through their lungs; and
- their bodies are covered with hair.

Table 5-1: A list of objects.

Object	What is it?	Object	What is it?
Angel	Fish	Mighty	Elephant
Bunny	Rabbit	Smudge	Cat
Janet	Female person	Jaws	Shark
Jeremy	Male person	Swift	Eagle
Flipper	Dolphin	Willy	Whale
Heather	Hen	Parry	Parrot
Wise	Owl	Sally	Snake
Kermit	Frog	Lily	Lizard
Beatle	Bug	Ben	Bee

Similarly, Parry, Heather, Wise, and Swift are grouped into a Bird category because they share common information typical of a bird:

- they have a beak;
- they have two legs;
- they have two wings;
- their wings and body are covered with feathers;
- they can fly;
- they lay eggs; and
- they are warm-blooded.

In a like manner, we group:

- Angel and Jaws into a Fish category;
- Sally and Lily into a Reptile category;
- Beatle and Ben into an Insect category; and
- Kermit into an Amphibian category.

Figure 5-1 shows the six categories of animal we have produced so far. In object-oriented modeling, the act of categorizing objects is known as *classification*. The categories formed are known as *classes*.

From a modeling perspective, classes form meaningful abstractions for organizing information; any reference to the classes would indirectly refer to the objects in the class.

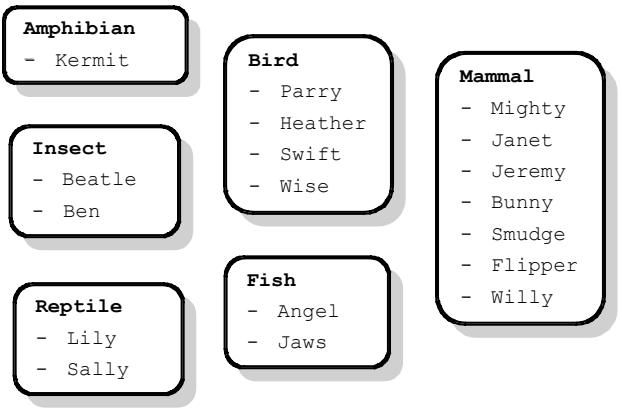


Figure 5-1: Categories of animal.

5.2 Hierarchical Relationship of Classes

Classes formed can be organized in a hierarchical manner. Depending on the position of a class in the hierarchy, it may be known as a superclass or a subclass of a class. Let us examine the notion of superclass and subclass here.

5.2.1 Superclass and Subclass

In Chapter 2, we introduced Sean and Sara as salespersons of HomeCare. We also mentioned that Sean and Sara are objects of the SalesPerson class. Let us introduce two more employees, Simon and Sandy. Specifically, Simon and Sandy are managers with properties that are slightly different from Sean and Sara. We will classify Simon and Sandy as objects of a different class: Manager. All four persons are employees of HomeCare and objects of another class: Employee. The relationships of these classes and objects are illustrated in Figure 5-2.

Note that Sean and Sara are shown as instances of the SalesPerson class while Simon and Sandy are instances of the Manager class. The enclosing boundary of the Employee class over these objects indicates that the objects are also instances of the Employee class.

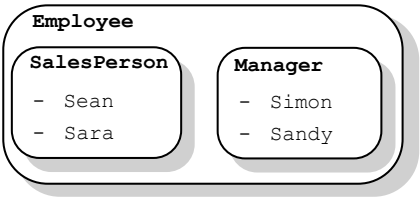


Figure 5-2: Employee, SalesPerson and Manager classes.

Sean and Sara therefore belong to two classes: the Employee class and SalesPerson class. Likewise, Simon and Sandy belong to the Employee and Manager class. This implies that the information about Sean and Sara as employees is also true of them as salespersons. We can thus refer to Sean as an employee or a salesperson.

Which class Sean is referred to is a matter of generality. When Sean is referred to as an employee, we are being general about who he is but when he is referred to as a salesperson, specific information about his role and employment is specified. For example, Sean takes orders and earns a commission for each sale since he is a salesperson but this does not apply to Sandy who is a manager, despite the fact that both are employees of HomeCare. Similarly, when we speak of an object as an employee, we are being general and its differences with objects of other classes are ignored.

The Employee class is said to be a *generalized* class of SalesPerson and Manager. Conversely, SalesPerson and Manager are said to be *specialized* classes of the Employee class. Generalized and specialized classes can be organized into a class hierarchy with the generalized classes placed toward the top of the hierarchy and the specialized classes toward the bottom of the hierarchy.

A specialized class is known as a *subclass* of a class while the generalized class is known as a *superclass* of a subclass in object-oriented terms. For example, SalesPerson is a subclass of the Employee class which is also the superclass of SalesPerson.

5.2.2 A Class Hierarchy Diagram

The hierarchical relationships among classes can be seen in a class hierarchy diagram in Figure 5-3. A box in the diagram represents a class while a triangle denotes the hierarchical relationship between classes with a superclass positioned at the top. Subclasses are placed toward the bottom of a class hierarchy diagram.

A class can be a superclass to a class or a subclass of another class or both depending on its position in the hierarchy. For example in Figure 5-3:

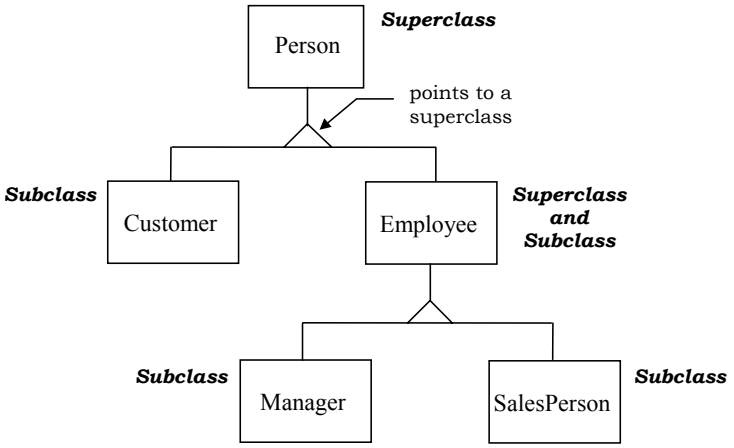


Figure 5-3: A class hierarchy diagram.

- Manager and SalesPerson are subclasses of the Employee class.
- Employee is a subclass of the Person class.
- Person is a superclass of Customer and Employee class.
- Employee is thus a superclass (to Manager and SalesPerson) and a subclass (of Person) in the hierarchy.

5.3 Generalization

Generalization is the act of capturing similarities between classes and defining the similarities in a new generalized class; the classes then become subclasses of the generalized class. For example, the Mammal, Fish, Bird, Reptile, and Amphibian classes introduced earlier, are similar in that all objects from these classes have a backbone. Based on this similarity, we can refer to them via a new superclass, say Animal-with-Backbone. Hence, we can refer to Kermit (an object of the Amphibian class) as an object of the Animal-with-Backbone class too. Similarly, the Insect class can be generalized into an Animal-without-Backbone class since objects from the Insert class are without a backbone. Figure 5-4 summarizes the relationships of these classes.

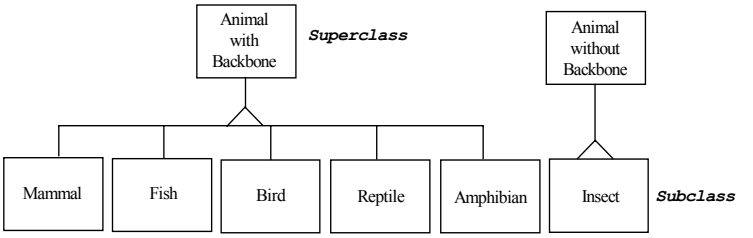


Figure 5-4: Generalizing classes.

The Animal-with-Backbone class and Animal-without-Backbone class can be further generalized by considering the similarities of objects from these two classes. Let us call the generalized class, Animal. The generalization of properties of Animal-with-Backbone class and Animal-without-Backbone class into the Animal class is shown in Figure 5-5.

The Animal class, being the topmost class in the class hierarchy, is thus the most general class of the entire Animal class hierarchy. This means that Swift, which is an object of the Bird class, is also an object of the Animal-with-Backbone class and Animal class, for example. When we refer to Swift as an object of the Animal class, we are being general about it and we would be ignoring specific information about Swift as a bird in this reference.

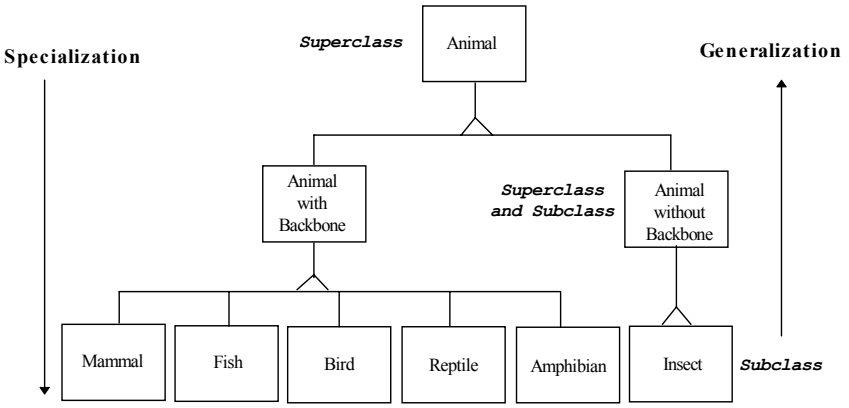


Figure 5-5: A class hierarchy for animals.

5.4 Specialization

In contrast, *specialization* is the act of capturing differences among objects in a class and creating new distinct subclasses with the differences. In this way, we are specializing information about objects of the superclass into subclasses. For example, in creating Animal-with-Backbone class and Animal-without-Backbone class from the Animal class, we are distinguishing information about objects with a backbone from others without a backbone into Animal-with-Backbone and Animal-without-Backbone classes. Eventually, only objects with a backbone would be classified into the Animal-with-Backbone class and the others into the Animal-without-Backbone class.

Similarly, objects from the Animal-with-Backbone class can be further classified into the Mammal, Fish, Bird, Reptile, or Amphibian classes depending on their properties definition.

5.5 Organization of Class Hierarchy

Classes in a class hierarchy diagram are organized in such a way that generalized classes are placed toward the top of the hierarchy.

As we traverse higher into a class hierarchy, the classes become more general in definition and more objects can be classified into them. As we traverse lower into the class hierarchy, the subclasses become more specialized in definition and fewer objects can be classified into them.

5.6 Abstract and Concrete Classes

There are classes in a class hierarchy that are so general that there is really no intention to create objects from them. Such classes are meant to contain common attributes or methods of subclasses for reuse purpose. These classes are known as *abstract classes* in object-oriented modeling. For example, the Animal-with-Backbone class has been included into the class hierarchy to contain properties similar to objects in classes Mammal, Fish, Bird, Reptile, and Amphibian. Similarly, the Animal-without-Backbone class abstract general information about insects and finally, the Animal class generalizes all common properties of classes in its definition.

While classes Animal, Animal-with-Backbone and Animal-without-Backbone abstract the common properties of objects, object instances are actually created from the lowest level subclasses. Such classes from which objects are instantiated are known as *concrete classes*. Thus, Mammal, Fish, Bird, Reptile, Amphibian and Insect are concrete classes for the above class hierarchy on animals.

Abstract classes are implemented in Java using the `abstract` keyword as follows:

```
abstract class Animal {
...
}

abstract class Animal-with-Backbone extends Animal {
...
}

abstract class Animal-without-Backbone extends Animal {
...
}
```

and concrete classes are defined in the usual way:

```
class Mammal extends Animal-with-Backbone {
 Mammal(String name) {}
  ...
}

class Fish extends Animal-with-Backbone {
 Fish() {}
 ...
}

class Mammal extends Animal-with-Backbone {
 Mammal(String name) {}
  ...
}

class Fish extends Animal-with-Backbone {
 Fish() {}
 ...
}
```

```
class Amphibian extends Animal-with-Backbone {
 Amphibian() {}
 ...
}

class Insect extends Animal-without-Backbone {
 Insect() {}
 ...
}
```

Intuitively,

```
Animal a = new Animal();
```

is not valid while

```
Mammal j = new Mammal("John");
```

is valid.

5.7 Summary

The following concepts were discussed in this chapter:

- *Classification*—categorizing objects into a class.
- A *subclass* is a specialized class of a superclass, and a *superclass* is a generalized class of a subclass.
- *Generalization*—the act of capturing similarities between classes and defining the similarities in a new generalized class; the classes then become subclasses of the generalized class.
- *Specialization*—the act of capturing differences among objects in a class and creating new distinct subclasses with the differences.
- *Abstract class*—a class from which no object instances will be created.
- *Concrete class*—a class from which object instances will be created.

5.8 Exercises

1. In your own words, describe *generalization* and *specialization*.
2. Information about some objects is given below. Classify the objects into these classes: Bird, Insect, Fish, and Four-Legged Animal.
3. Create generalized classes for the classes in Question 2 and produce a class hierarchy.
4. Indicate in your class hierarchy for Question 3 abstract classes and concrete classes.

Object	What is it?	Data	Methods
Aaron	Ant	a, b, d, f	X(),Z()
Beatle	Bug	a, b, d, f	X(),Z()
Smudge	Dog	a, b, d, l	X(),Z()
Swift	Eagle	a, b, c, e	X(),Y()
Herman	Hawk	a, b, c, e	X(),Y()
Oscar	Orange	u	N()
Rosie	Rose	v	O()
Tora	Tiger	a, b, d, l	X(),Z()
Goldie	Goldfish	a, b, d, g	X(),Z()
John	Male Person	a, b, d, h, i	X(),Z()
Jack	Jaguar	a, b, d, l	X(),Z()
Angel	Goldfish	a, b, d, g	X(),Z()

6

Inheritance

In Chapter 5, we discussed generalization/specialization as an abstraction mechanism for modeling classes and their hierarchical relationships with one another. We also introduced superclasses as generalized classes of subclasses.

In this chapter, we will discuss *inheritance* as a mechanism for propagating properties (attributes and methods) of superclasses to subclasses. The properties then form part of the subclasses' definition. From an implementation standpoint, inheritance encourages software reuse. The impact of inheritance on software development will also be discussed.

6.1 Common Properties

A class hierarchy on persons and employees was earlier introduced in Chapter 5. This hierarchy is reproduced in Figure 6-1 but with attribute and method definitions added onto it.

We can make the following observations about the class diagram:

- attribute `name` and method `getName()` are common in all three classes;
- attribute `employee number` and method `getEmployeeNumber()` are common in Employee and SalesPerson class;
- attribute `commission` is specific to SalesPerson class and does not appear in other classes.

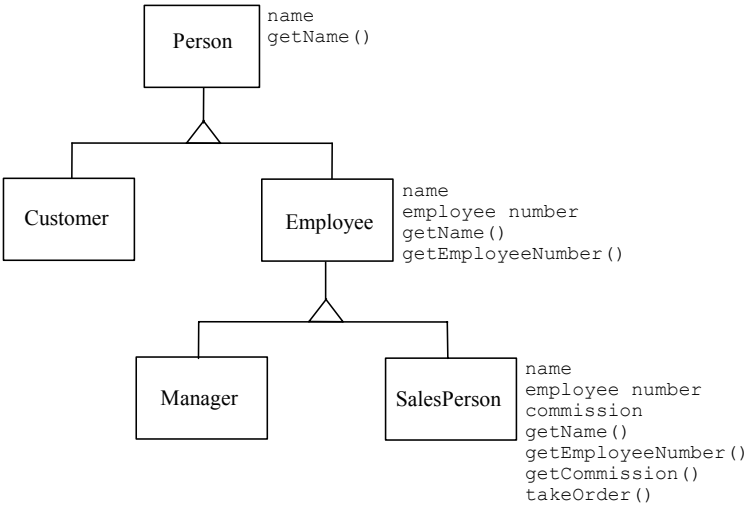


Figure 6-1: Common properties in classes.

6.2 Inheritance

From a software reuse standpoint, generalized properties defined in superclasses should be made available to subclasses without having to declare them explicitly in the subclasses. In object-oriented programming, such reuse is possible via inheritance.

Inheritance is the ability of a subclass to take on the general properties of superclasses in the inheritance chain. The properties then form part of the subclass' definition. Inheritance enables superclasses' properties to be propagated downward to the subclasses in a class hierarchy, and makes the properties available as part of the subclasses' definition. These properties are said to be *inherited* (or taken on) by the subclasses.

Using inheritance, the SalesPerson class of Figure 6-1 can now be defined by a combination of properties from:

- the Employee class;
- the Person class; and
- its own specific attribute and method definition.

Figure 6-2 shows the modified class hierarchy for persons (with inherited properties highlighted in bold). The Person class has the following definition:

```
Class Person {
  Attributes :
    name
  Methods :
    getName()     {return name}
}
```

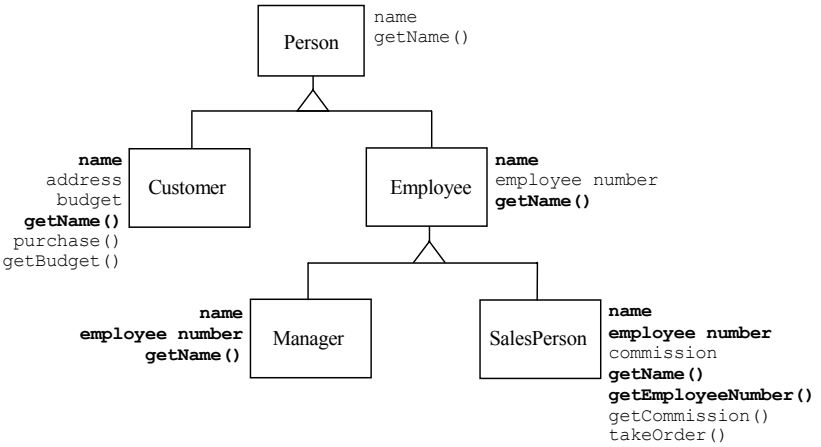


Figure 6-2: Classes with inherited properties.

The Employee class is reduced to (with inherited properties highlighted in bold):

```
Class Employee {
  Attributes :
    name                                (inherited from Person class)
    employee number
  Methods :
    getName()    {return name}          (inherited from Person class)
    getEmployeeNumber(){return employee number}
}
```

The SalesPerson class is simplified into (with inherited properties highlighted in bold):

```
Class SalesPerson {
  Attributes :
    name                          (inherited from Person Class)
    employee number               (inherited from Employee class)
    commission
  Methods :
    takeOrder(who, stock, address, date)   {
      check with warehouse on stock availability
      check with warehouse on delivery schedule
      if ok
      then {instruct warehouse to deliver stock
             to address on date
            return ok}
      else return not ok
    }
    getName()    {return name} (inherited from Person class)
    getEmployeeNumber(){return employee number}
                            (inherited from Employee class)
    getCommission() {return commission}
}
```

Note that attributes `name` and `employee name`, and methods `getName()` and `getEmployeeNumber()` of the SalesPerson class are not explicitly defined in the SalesPerson class but are propagated downward from the superclasses through inheritance.

Only downward propagation of properties from superclasses to subclasses is permissible. There is no upward propagation of properties in object-oriented programming. Therefore, information specific to subclasses are unique to subclasses and are not propagated to superclasses. For this reason, attribute `commission` and method `getCommission()` of the SalesPerson class do not form part of the Employee class definition.

6.3 Implementing Inheritance

Let us now extend the Person class hierarchy to include a new class, Secretary. Figure 6-3 shows the modified class hierarchy with inherited properties highlighted in bold. The information in the extended hierarchy suggests that all employees have a basic salary except managers and salespersons, who are paid an allowance and commission, respectively.

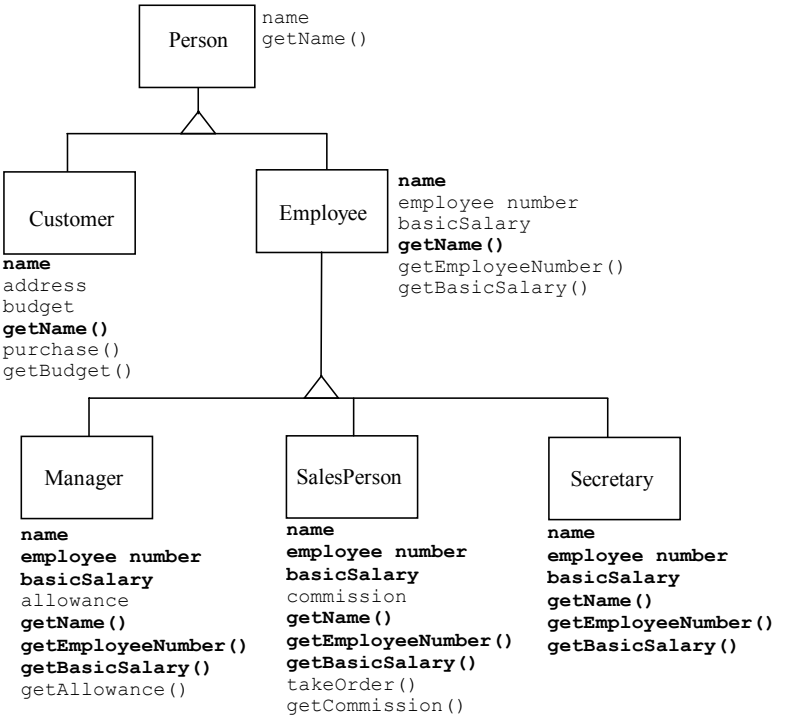


Figure 6-3: Including the Secretary class.

The Superclass–Subclass relationship in a class hierarchy is denoted in the code by the keyword `extends`. This suggests that a subclass is an extension of a superclass. For example, the following code fragment suggests that Employee is an extension of the Person class; Manager, SalesPerson and Secretary, being subclasses, are extensions of the Employee class:

```
class Employee extends Person {
   ...
}

class Manager extends Employee {
   ...
}

class SalesPerson extends Employee {
   ...
}

class Secretary extends Employee {
   ...
}
```

Listing 6-1 is the code implementing the Person hierarchy. Code execution begins with the Employee class since it is the only class that contains `static void main()`. To execute `main()`, type the following at the command prompt line:

```
$ java Employee
```

Executing the code produces the following output:

```
The Manager Simon (employee number 01234M) has a salary of 9000
The Secretary Selene (employee number 98765S) has a salary of 2500
The Manager Simon also has an allowance of 2000
```

The output suggests that some information was made available from the superclasses in deriving the manager's and secretary's salary. Objects instantiated from the Manager or Secretary class were able to respond to requests for their `basicSalary` because they have inherited from the Employee class the attribute `basicSalary` and method `getBasicSalary()`.

Let us examine `main()`. The code begins with the instantiation of two objects, a manager and a secretary. The manager object is referenced by variable `m` while the secretary object is referenced by variable `s`:

```
Manager m = new Manager("Simon", "01234M", 9000.0f, 2000.0f);
Secretary s = new Secretary("Selene", "98765S", 2500.0f);
```

The state of the two objects is depicted in Figure 6-4.

```
class Person {
  private String name;
  Person(String aName) {name=aName;}
  public String getName() { return name; }
}

class Employee extends Person {
  private float basicSalary;
  private String employeeNumber;

  Employee(String aName, String aEmployeeNumber,
           float aBasicSalary) {
    super(aName);
    employeeNumber = aEmployeeNumber;
    basicSalary = aBasicSalary;
  }

  public String getEmployeeNumber() { return employeeNumber; }
  public float getBasicSalary() { return basicSalary; }

  public static void main(String argv[]) {
    Manager m = new Manager("Simon", "01234M", 9000.0f,2000.0f);
    Secretary s = new Secretary("Selene", "98765S", 2500.0f);
    System.out.print("The Manager "+m.getName()+
                     " (employee number "+m.getEmployeeNumber()+")");
    System.out.println(" has a salary of "+m.getBasicSalary());
    System.out.print("The Secretary "+s.getName()+
                     " (employee number "+s.getEmployeeNumber()+")");
    System.out.print("The Manager "+m.getName());
    System.out.println(" also has an allowance of " +m.getAllowance());
  }
}

class Manager extends Employee {
  private float allowance;

  Manager(String aName, String aEmployeeNumber,
          float aBasicSalary, float aAllowanceAmt) {
    super(aName, aEmployeeNumber, aBasicSalary);
    allowance = aAllowanceAmt;
  }
  public float getAllowance() {
    return allowance;
  }
}

class Secretary extends Employee {
  Secretary (String aName, String aEmployeeNumber,
             float aBasicSalary) {
    super(aName, aEmployeeNumber, aBasicSalary);
  }
}
```

Listing 6-1: Inheritance in the extended Person hierarchy.

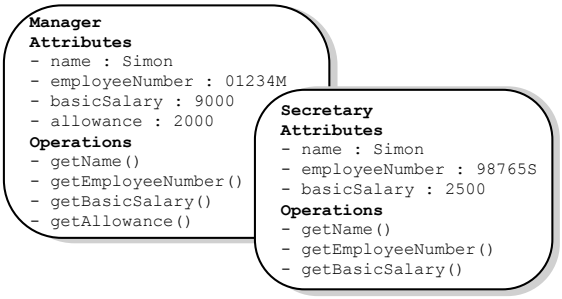


Figure 6-4: State of Manager and Secretary object.

The remainder of `main()` are output statements. Some methods from various classes, invoked in producing the outputs, for example, `getName()` from the Person class, and `getEmployeeNumber()` and `getBasicSalary()` from the Employee class, are propagated through the inheritance mechanism to the subclasses, Manager and Secretary.

6.4 Code Reuse

By allowing information of a superclass to be taken on by subclasses, the information is said to be reused at the subclass level. All newly created instances of the subclasses would have as part of their definition the inherited information. `employee number`, `basic salary`, and `getEmployeeNumber()` of the Employee class and `name` and `getName()` of the Person class are said to be *reused* by the Manager and Secretary class.

6.5 Making Changes in Class Hierarchy

Changes to software specification are inevitable. Let us consider how changes in a class hierarchy impact software maintenance as a whole. The following situations will be discussed:

- Change in property definition for *all* subclasses.
- Change in property definition for *some* subclasses.
- Adding/deleting a class.

6.5.1 Change in Property Definition for All Subclasses

Suppose a change in representational scheme of the `employee number` in Figure 6-3 is required. This change will affect not only the attribute `employee number` but also the method `getEmployeeNumber()` and possibly other classes that inherit `employee number`. We will examine this change in two possibilities:

a. inheritance is not available;
b. inheritance is available.

(a) Inheritance Is Not Available

In the case where inheritance is not available, the attribute `employee number` and method `getEmployeeNumber()` would have to be defined in all the relevant classes, for example, Employee, Manager, SalesPerson and Secretary. The change in representational scheme of `employee number` would thus have to be effected individually on these classes. The redundancy arising from the multiple definition of `employee number` and `getEmployeeNumber()` may lead to inconsistency in definition if the change is not carried out properly.

(b) Inheritance Is Available

With inheritance, the situation is different. We would first define attribute `employee number` and method `getEmployeeNumber()` in Employee class and let subclasses Manager, SalesPerson and Secretary inherit these definitions from Employee class. The required change in representational scheme for attribute `employee number` would be limited to the Employee class. The change would be propagated to the subclasses via inheritance. In this way, the change is thus limited to the superclass, enabling a uniform and consistent property definition for all subclasses. In addition, redundancy in property definition at the subclass level can be minimized and software maintenance enhanced.

6.5.2 Change in Property Definition for Some Subclasses

In some situations, a change in property definition at the superclass level may not necessarily apply to all subclasses. The above solution would therefore not apply in these situations. To illustrate, let us extend the Person class hierarchy further to include two more employee classes: Technician and Clerk.

Let us assume the following for a HomeCare employee:

- a manager—basic salary plus allowance;
- a salesperson—basic salary plus commission;
- a secretary—basic salary;
- a technician—basic salary;
- a clerk—basic salary.

At the Employee class, a `getPay()` method is defined to return the monthly pay of an employee since the method applies to all classes of employee. The definition of the Person class remains the same as before:

```
class Person {
  private String name;
  Person(String aName) {name=aName;}
  public String getName() { return name; }
}
```

Employee extends Person as follows:

```
class Employee extends Person {
  private float basicSalary;
  private String employeeNumber;

  Employee(String aName, String aEmployeeNumber,
           float aBasicSalary) {
    super(aName);
    employeeNumber = aEmployeeNumber;
    basicSalary = aBasicSalary;
  }

  public String getEmployeeNumber() { return employeeNumber; }
  public float getBasicSalary() { return basicSalary; }
  public float getPay() { return basicSalary; }
  public static void main(String argv[]) {
    Manager m = new Manager("Simon", "01234M", 9000.0f, 2000.0f);
    Secretary s = new Secretary("Selene", "98765S", 2500.0f);
    Technician t = new Technician("Terrence", "42356T", 2000.0f);
    Clerk c = new Clerk("Charmaine", "68329C", 1200.0f);
    System.out.print("The Manager "+m.getName()+
                     " (employee number "+m.getEmployeeNumber()+")");
    System.out.println(" has a pay of "+m.getPay());
    System.out.print("The Secretary "+s.getName()+
                     " (employee number "+s.getEmployeeNumber()+")");
    System.out.println(" has a pay of "+s.getPay());
    System.out.print("The Technician "+t.getName()+
                     " (employee number "+t.getEmployeeNumber()+")");
    System.out.println(" has a pay of "+t.getPay());
    System.out.print("The Clerk "+c.getName()+
                     " (employee number "+c.getEmployeeNumber()+")");
    System.out.println(" has a pay of "+c.getPay());
  }
}
```

As before, `main()` is defined in the Employee class with additional code for Technician and Clerk class highlighted in bold. There is no change in class definition for Manager and Secretary. Technician and Clerk extend Employee, since they are subclasses of Employee:

```
class Technician extends Employee {
  Technician (String aName, String aEmployeeNumber,
              float aBasicSalary) {
    super(aName, aEmployeeNumber, aBasicSalary);
  }
}
class Clerk extends Employee {
  Clerk (String aName, String aEmployeeNumber,
         float aBasicSalary) {
    super(aName, aEmployeeNumber, aBasicSalary);
  }
}
```

Executing `main()` produces the following output:

```
The Manager Simon (employee number 01234M) has a pay of 9000
The Secretary Selene (employee number 98765S) has a pay of 2500
The Technician Terrence (employee number 42356T) has a pay of 2000
The Clerk Charmaine (employee number 68329C) has a pay of 1200
```

A cursory examination of the output reveals an inaccuracy in the manager's pay: an omission of allowance amounting to $2000. What has gone wrong?

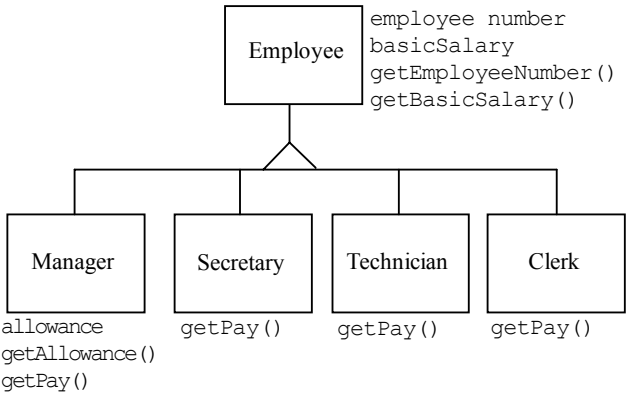


Figure 6-5: Extended Employee class hierarchy.

The above problem can be approached in two ways:

- Remove the `getPay()` method from the Employee class and define it individually in the subclasses (Secretary, Technician, Clerk, and Manager).
- Maintain the definition of `getPay()` method in Employee class and *redefine* it in the Manager class.

Figure 6-5 illustrates a class diagram for the first approach. Each of the subclasses has its own implementation of the `getPay()` method. One disadvantage of this approach is that the definition of the `getPay()` method has to be repeated in all the subclasses. This is highly inefficient and can be difficult to maintain especially in situations where the number of subclasses is large.

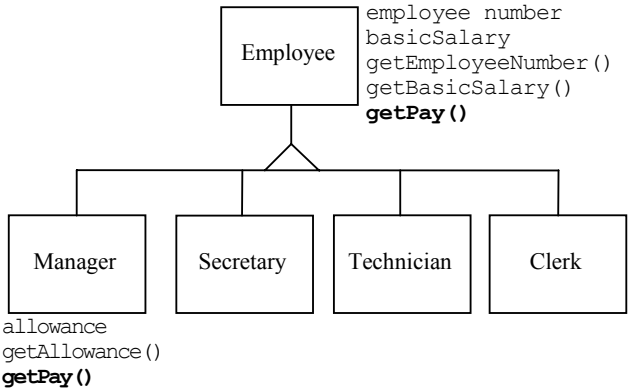


Figure 6-6: Redefining `getPay()` method of Manager.

In the second approach, the definition of the `getPay()` method is maintained at the Employee class but redefined in the Manager class. This ensures that the `getPay()` method is inherited by all subclasses of Employee, including the Manager class.

Since a similar `getPay()` method is defined in Manager, the `getPay()` method of the Manager class would be used in the resolution of method call by the object-oriented system instead. This is depicted in Figure 6-6 and Listing 6-2.

The `getPay()` method of the Manager class is said to redefine the `getPay()` method of the Employee class. Note that a redefined method has the same method name and parameter definition of a redefining method. While a redefining method has the same method signature with the redefined method, the implementation of the methods may differ. In this case, the `getPay()` method of the Manager class includes an additional computation of the allowance component.

```
class Person {
  ...
}
class Employee extends Person {
  ...
  public float getPay() { return basicSalary; }
  public static void main(String argv[]) {
  ...
  }
}
class Manager extends Employee {
   private float allowance;
   Manager(String aName, String aEmployeeNumber,
           float aBasicSalary, float aAllowanceAmt) {
      super(aName, aEmployeeNumber, aBasicSalary);
      allowance = aAllowanceAmt;
   }
   public float getAllowance() {
      return allowance;
   }
   public float getPay() {
      return (basicSalary + allowance);
   }
}
class Secretary extends Employee {
  ...
}
class Technician extends Employee {
  ...
}
class Clerk extends Employee {
  ...
}
```

Listing 6-2: Redefining the `getPay()` method.

Judging from the output of the two solutions, both approaches are correct:

```
The Manager Simon (employee number 01234M) has a pay of 11000
The Secretary Selene (employee number 98765S) has a pay of 2500
The Technician Terrence (employee number 42356T) has a pay of 2000
The Clerk Charmaine (employee number 68329C) has a pay of 1200
```

However, the second approach is better than the first approach as it enhances software reuse and minimizes the effect of change on other classes. Redefinition of methods is supported in object-oriented programming and closely connected with operation overloading. We will further discuss *operation overloading* in the next chapter.

6.5.3 Adding/Deleting a Class

Adding a class into an existing class hierarchy can be detrimental to the stability of the hierarchy. It is always recommended that the addition of a new class be created as a subclass in the class hierarchy. The definition of existing classes will not be adversely affected by this approach. To illustrate, let us consider an example of geometrical shapes.

Figure 6-7 is a class hierarchy of shapes. Shape is a generalized class of Circle and Square. All shapes have a name and a measurement by which the area of the shape is calculated.

The attribute `name` and method `getName()` are defined as properties of Shape. Circle and Square, being subclasses of Shape, inherit these properties (highlighted in bold in Figure 6-7).

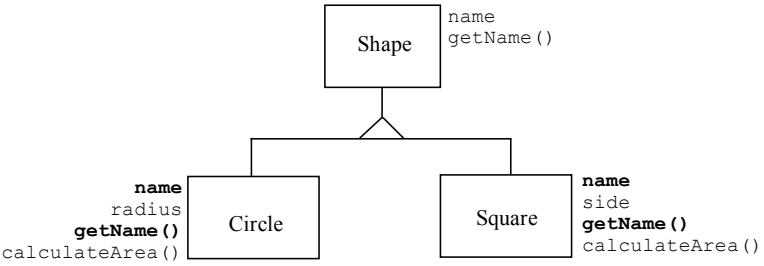


Figure 6-7: Class hierarchy of Shape, Circle and Square.

The Shape class has the following definition:

```
class Shape {
  private String name;
  Shape(String aName) {name=aName;}
  public String getName() {return name;}
  public float calculateArea() {return 0.0f;}

  public static void main(String argv[]) {
    Circle c = new Circle("Circle C");
    Square s = new Square("Square S");
    Shape shapeArray[] = {c, s};
    for (int i=0; i<shapeArray.length; i++) {
      System.out.println("The area of " + shapeArray[i].getName()
                + " is " + shapeArray[i].calculateArea()+" sq. cm.\n");
    }
  }
}
```

Note that the attribute `name` is declared as `private` in the Shape class. To make it known to other objects, a `getName()` method is defined in the Shape class to return the value of attribute `name`. The keyword `private` is an access control specifier which was first introduced in Chapter 3 and will be further discussed in Chapter 8.

Circle and Square extend Shape and have the following definition:

```
class Circle extends Shape {
  private int radius;
  Circle(String aName) {
    super(aName);
    radius = 3;
  }

  public float calculateArea() {
    float area;
    area = (float) (3.14 * radius * radius);
    return area;
  }
}

class Square extends Shape {
  private int side;
  Square(String aName) {
    super(aName);
    side = 3;
  }

  public float calculateArea() {
    int area;
    area = side * side;
    return area;
  }
}
```

As usual, program execution begins with `main()` and the following output is produced when `main()` is executed:

```
The area of Circle C is 28.26 sq. cm.
The area of Square S is 9 sq. cm.
```

Two objects are created in `main()`–a Circle object referenced by the variable `c` and a Square object referenced by the variable `s`. The creation of a Circle object involves a call to the Circle class constructor method via the `new` keyword. A `name` parameter is required to activate the Circle constructor method. For Circle, the `name` parameter is the string `"Circle C"`.

A call is made to the Circle's superclass constructor method via the statement

```
super(aName);
```

The call assigns the value of the parameter (`"Circle C"`) to the Circle object's attribute `name`. When the assignment is done, control returns to the Circle class's constructor method. Subsequently, the `radius` attribute of the Circle object is assigned the value 3 via the statement:

```
radius = 3;
```

Likewise, the Square object is created and its attributes updated in the execution. By now, the Circle and Square objects have a state as illustrated in Figure 6-8.

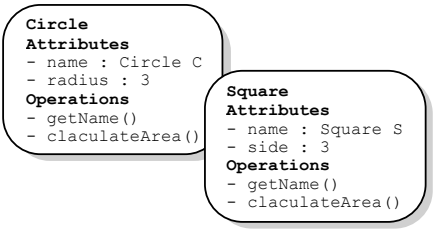


Figure 6-8: State of Circle and Square object.

An array `shapeArray` is declared in the next statement. The reference variables `c` and `s` are assigned to the array elements. Iterating through the array (via the `for`-loop), the area of the respective shape is produced by the statement

```
System.out.println("The area of " + shapeArray[i].getName()
              + " is " + shapeArray[i].calculateArea()+" sq. cm.\n");
```

We will explain the implications of `shapeArray` and `calculateArea()` in the next chapter when the topic of *polymorphism* is discussed. For now, we will focus on the impact arising from the addition of classes to an existing class hierarchy.

Suppose we want to add to the Shape class hierarchy a new class called Triangle. Listing 6-3 shows the modified code with new additions highlighted in bold.

```
class Shape {
  ...

  public static void main(String argv[]) {
    Circle c = new Circle("Circle C");
    Square s = new Square("Square S");
    Triangle t = new Triangle("Triangle T");
    Shape shapeArray[] = {c, s, t};
    for (int i=0; i<shapeArray.length; i++) {
      System.out.println("The area of " + shapeArray[i].getName()
                + " is " + shapeArray[i].calculateArea()+" sq. cm.\n");
    }
  }
}

class Circle extends Shape {
...
}

class Square extends Shape {
 ...
}

class Triangle extends Shape {
  private int base, height;

  Triangle(String aName) {
    super(aName);
    base   = 4;  height = 5;
```

```
  }

  public float calculateArea() {
    float area = 0.5f * base * height;
    return area;
  }
}
```

Listing 6-3: Adding a Triangle.

To add the new Triangle class, the following is involved:

1. Add a statement to create a Triangle object in `main()`.
2. Add a statement to include the newly created triangle into `shapeArray` in `main()`.
3. Create a new Triangle class as a subclass of Shape.

It is clear that subclassing the new Triangle class into the class hierarchy does not affect the definition of the other three classes. Subclassing is specialization and is thus a desired design practice in object-oriented software engineering because it has minimal impact on software maintenance.

Thus, the deletion of subclasses that are not superclasses to other classes has a minimal impact on software maintenance.

6.6 Accessing Inherited Properties

Inherited properties form part of the definition of subclasses, but they may not necessarily be accessible by other classes. Accessibility of inherited properties can be controlled using access control specifiers, which will be discussed in Chapter 8.

6.7 Inheritance Chain

We have so far discussed class hierarchies whose classes have only one parent or superclass. Such hierarchies are said to exhibit *single inheritance*. The path of inheritance over the classes is known as the inheritance chain.

A single inheritance chain can be *single-* or *multilevel*. In a single-level single inheritance chain, there is only one level of superclass that a subclass can inherit properties from. In contrast, in a multilevel single inheritance chain, a subclass can inherit from more than one level of superclasses. The difference is illustrated in Figure 6-9.

Besides single inheritance, there is also multiple inheritance. A class hierarchy is said to exhibit *multiple inheritance* if a subclass in the hierarchy inherits properties from two or more superclasses in more than one inheritance path.

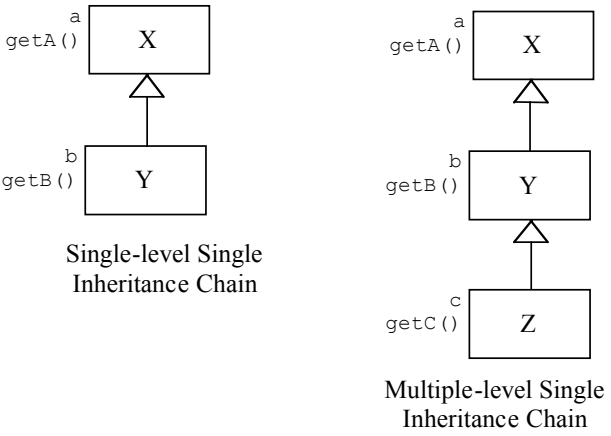


Figure 6-9: Single- and multiple-level single inheritance chains.

6.7.1 Multiple Inheritance

To appreciate the concept of multiple inheritance, let us consider the example of frogs. A frog is an amphibian that takes on characteristics typical of a land animal and a water animal. A frog can live both on land and in water. We can thus represent frogs as instances of a Frog class and specialize it as a subclass of Land-Animal class and Water-Animal class. The Frog class will inherit from the Land-Animal class the ability to live on land and from the Water-Animal class the ability to survive in water. Figure 6-10 shows the position of the Frog class in relation to its superclasses in a class hierarchy.

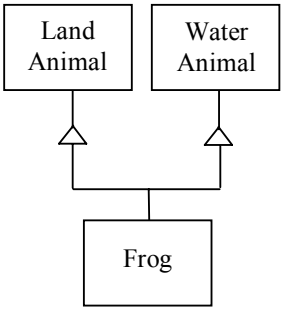


Figure 6-10: Multiple inheritance.

By inheriting properties from more than one superclass, the Frog class is said to exhibit multiple inheritance. All properties of the Land-Animal and Water-Animal classes are now part of the definition of the Frog class.

6.7.2 Problems Associated with Multiple Inheritance

Consider the example of a sales manager. A sales manager can take orders from a customer (as a salesperson would) and authorize orders (as a manager would). Thus, a SalesManager class is a subclass of two superclasses, namely, the Manager class and the SalesPerson class, as shown in Figure 6-11. The class diagram can be read as "a sales manager is both a salesperson and a manager". The SalesManager class would inherit from the Manager class the method `authorize()` and from SalesPerson class the method `takeOrder()`.

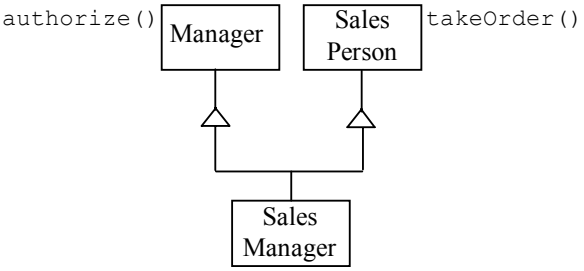


Figure 6-11: Class hierarchy for SalesManager.

Assuming that the concept of multiple inheritance is not available. We would have to represent SalesManager as a subclass in a single inheritance chain as shown in Figure 6-12.

Although the properties inherited in this hierarchy is the same as that of Figure 6-11, the class hierarchy of Figure 6-12 is semantically incorrect—a sales person is not a manager.

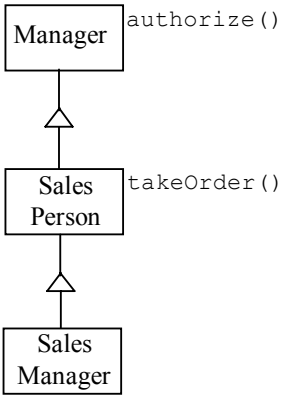


Figure 6-12: A SalesManager class hierarchy using single inheritance.

While it is clear from the above example that multiple inheritance is useful for a more natural approach to modeling information about the real world, it has its problems, particularly in situations where the same attributes or methods are redefined in

the multiple inheritance paths. To illustrate, let us consider the class hierarchy of Figure 6-13.

The attribute `a` and method `getA()` have been redefined in class X and Y. Which copy of `a` should a Z object inherit from? Or, which `getA()` method should a Z object use? Here, the Z object can be an X object or a Y object at some point in time.

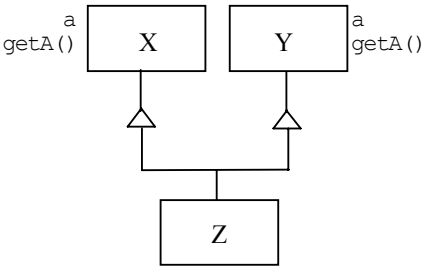


Figure 6-13: Redefined attribute and method in multiple inheritance paths.

In like manner, we can extend the class hierarchy for sales managers to that shown in Figure 6-14. We say that a sales manager is a manager and a salesperson. A manager and a salesperson in turn are employees in general.

By means of inheritance, the definition of a SalesManager object would include:

- attribute `name`; and
- inherited methods `getName()`, `authorize()`, `takeOrder()` and `collectPayment()`.

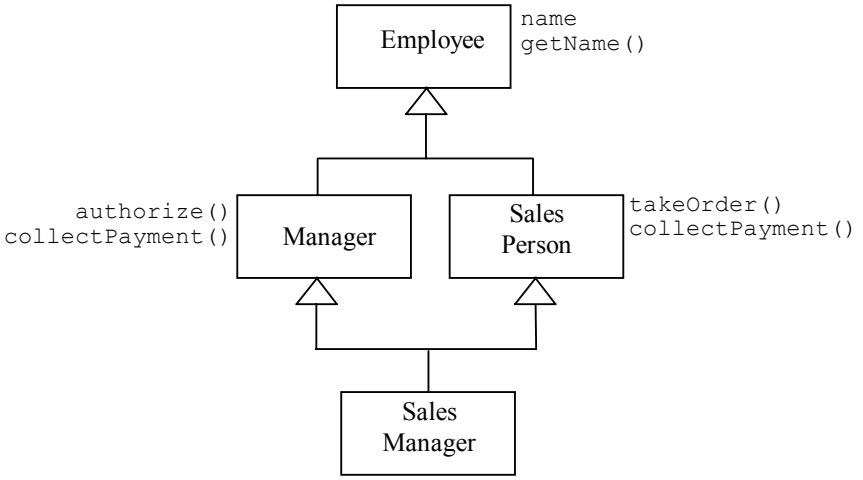


Figure 6-14: Extended class hierarchy for SalesManager.

Since the method `collectPayment()` occurs in two superclasses—Manager and SalesPerson—it would be difficult, from a language implementation point of view, to determine which of the `collectPayment()` method a SalesManager object should inherit.

There are thus ambiguities in language implementation when multiple inheritance paths redefine the same attributes or methods in multiple inheritance.

6.7.3 Contract and Implementation Parts

Basically, a method has two parts: *contract* and *implementation*. The *contract* part is also known as the method signature and specifies the name of the method, its return type and formal parameters (if any). The *implementation* part contains statements that are executed when a method is called. It is through the implementation of the method that the state of an object is changed or accessed. By changing the implementation of a method, the behavior of an object is altered, even though the contract part remains unchanged. In Listing 6-4,

```
public int getA()
```

is the contract part, while the two statements:

```
a = a+1;
return a;
```

enclosed within the block form the implementation part of the `getA()` method.

```
class X {
  int a;
  public int getA() {
    a = a+1;
    return a;
  }
}
```

Listing 6-4: Class X and method `getA()`.

Similarly, an attribute has a definition and an implementation: the name of an attribute defines the attribute while its structure specifies the implementation of the attribute.

6.7.4 Contract and Implementation Inheritance

A class inherits property definitions and the way the properties are implemented from its superclasses in a single inheritance chain. In other words, a class in a single inheritance chain inherits both the contract and implementation parts of the superclasses' properties.

The inheritance of implementation causes the ambiguities in multiple inheritance. When the same attributes or methods are redefined in the multiple inheritance paths, there is a need to determine which implementation of the redefined properties should be used.

To avoid the problems of multiple inheritance, Java does not support it *explicitly*. Instead Java supports single inheritance and provides a means by which *the effects of multiple inheritance* can be realized. This contingency is exercised when single inheritance alone may not be sufficient for representing real-world situations.

The approach in Java allows subclasses to inherit contracts but not the corresponding implementation. Subclasses must provide the implementation part of methods. Multiple inheritance is thus indirectly supported in Java with subclasses self-implementing the appropriate behavior. The implementation to be inherited is therefore resolved at the subclass level.

For redefined attributes in multiple inheritance paths, only constant values are allowed in Java. By definition, constants are not modifiable.

6.8 Interface

Contract inheritance is supported in Java via the interface construct. An interface definition is similar to a class definition except that it uses the `interface` keyword:

```
interface I {
  void j();
  int k();
}
```

All methods in an interface are *abstract methods*, that is, they are declared without the implementation part since they are to be implemented in the subclasses that use them.

6.8.1 Multiple Inheritance Using Interface

The interface construct in Java is used with single inheritance to provide some form of multiple inheritance. To illustrate, we will refer to the example on sales managers introduced earlier and examine how we can resolve the problems encountered in the single-inheritance solution.

The SalesManager class is first defined as a subclass of SalesPerson in a single inheritance chain. SalesPerson is in turn declared as a subclass of Employee. Given this inheritance hierarchy, a sales manager is a salesperson who in turn is an employee in general.

In order that a sales manager has the ability to manage, we add to the SalesManager class, appropriate behavior in an interface which is then inherited by the SalesManager class. We shall call that interface `Manage` as follows:

```
interface Manage {
  boolean authorize();
}
```

Figure 6-15 shows the class hierarchy for SalesManager that is described as a salesperson with *additional* ability to authorize orders.

In using the `Manage` interface, the subclass SalesManager must implement the abstract methods of the interface. This is reflected in the class declaration of Sales Manager:

```
class SalesManager extends SalesPerson implements Manage {
  ...
}
```

This indicates that a class, SalesManager, derived from the SalesPerson class implements the `Manage` interface.

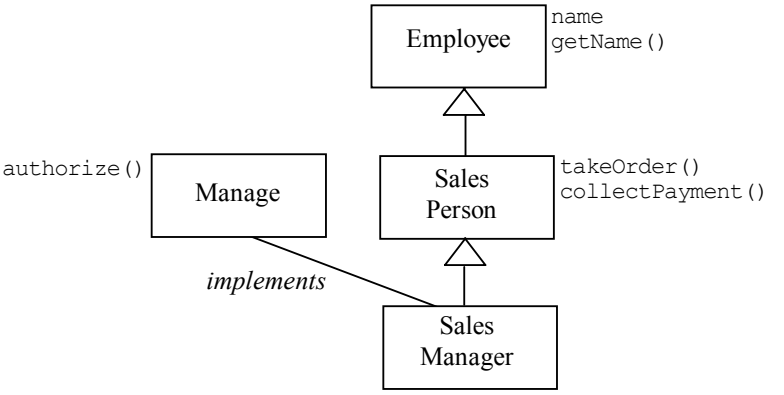


Figure 6-15: Single inheritance with interface.

The code for the classes in Figure 6-15 is illustrated in Listing 6-5. For illustration purposes, the code for `takeOrder()` and `collectPayment()` methods has been simplified to produce output to indicate their execution. The abstract method `authorize()` of the `Manage` interface has been implemented in the SalesManager class has been embolden.

As expected, the following output was produced:

```
Order taken
Order authorized
Payment collected
SalesManager Sara took order, authorized it and collected payment.
```

The `collectPayment()` method of the SalesPerson class has been predetermined and used in the above solution. A sales manager is basically a salesperson with an additional behavior to authorize payments (via the `authorize()` method).

```
interface Manage {
  boolean authorize();
}

class Employee {
  String name;
  Employee() {}
  String getName() {return name;}
}
```

```
class SalesPerson extends Employee {
  boolean takeOrder() {
    System.out.println("Order taken");
    return true;
  }
  void collectPayment() {
    System.out.println("Payment collected");
  }
}

class SalesManager extends SalesPerson implements Manage {
  SalesManager(String n) {name = n;}
  public boolean authorize() {
    // authorisation by a sales manager
    System.out.println("Order authorized");
    return true;
  }

  public static void main(String args[]) {
    SalesManager sm = new SalesManager("Sara");
    if (sm.takeOrder())
      if (sm.authorize()) {
        sm.collectPayment();
        System.out.println("SalesManager "+sm.getName()+
          " took order, authorized it and collected payment. ");
      }
      else System.out.println("SalesManager "+sm.getName()+
          " did not authorize order. No payment collected. ");
  }
}
```

Listing 6-5: SalesManager class and manageable interface.

An alternative implementation would be to consider SalesManager as a subclass of Manager taking on methods `authorize()` and `collectPayment()`, as shown in Figure 6-16. `takeOrder()` is derived from implementing the `CanTakeOrder` interface instead. Note that the implementation of `collectPayment()` method in this case is different from the implementation of `collectPayment()` method in Figure 6-15.

Which of these two solutions is preferred is a modeling problem and can be resolved if more information on the problem domain and requirements is provided.

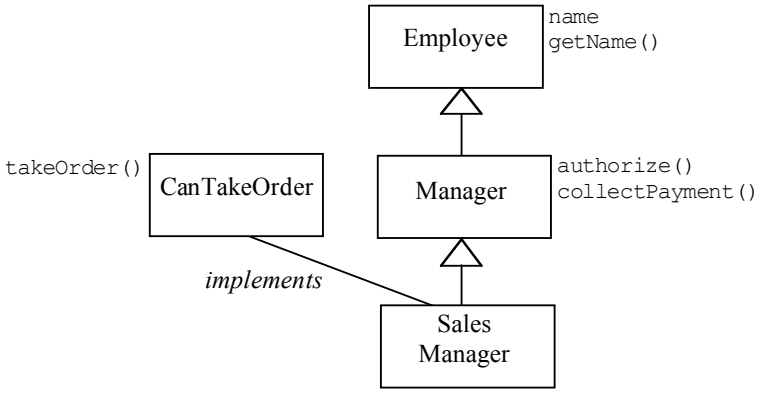


Figure 6-16: Inheriting from Manager class.

6.8.2 Attributes in an Interface

Data attributes declared in an interface construct are always `static` and `final`. They are `static` as there can only be one copy of the data available and `final` since they are not modifiable. By declaring data attributes in an interface, constant declarations for use in methods is possible. Constants are names for values with a specific meaning.

As all data attributes are implicitly declared as `static` and `final` in an interface definition, these keywords need not precede their declaration:

```
interface Colourable {
  int RED = 1;
  int GREEN = 2;
  int BLUE = 3;

  void setColour (int c);
  int getColour();
}
```

6.8.3 Methods in an Interface

All methods in an interface are abstract methods and any class that uses the interface must provide an implementation for them. Like data attributes, an interface does not have to explicitly declare its methods abstract using the keyword `abstract`.

Similarly, interface methods are always public, and the access modifier `public` keyword is not required since it is implied in the interface declaration. However, in contrast with data attributes in an interface, methods may not be `static` since static methods, being class specific, are never abstract.

6.8.4 Abstract Class and Interface

A class implementing an interface must implement all the abstract methods declared in an interface; otherwise, the class is considered as an *abstract* class and must be declared using the `abstract` keyword as follows:

```
abstract class ColourTest implements Colourable {
  int i;
  ColourTest() {}

  public void setColour (int c) {
    i=c;
  }

  public static void main(String args[]) {
    ...
  }
}
```

The class ColourTest is declared `abstract` since the `getColour()` method of the `Colourable` interface is not implemented. Note that the `setColour()` method has to be declared `public` as it is a method of the `Colourable` interface.

There are some differences between an abstract class and an interface. These differences are summarized as follows:

Abstract Class	Interface
May have some methods declared `abstract`.	Can only have abstract methods.
May have `protected` properties and `static` methods.	Can only have `public` methods with no implementation.
May have `final` and nonfinal data attributes.	Limited to only constants.

An abstract class can enhance inheritance as some or all parts of the class can be implemented and inherited by subclasses. An interface, on the other hand, is generally used for achieving multiple inheritance in Java.

An abstract class cannot be used to instantiate objects since it may contain parts that are not implemented. Given a declaration of an abstract class LandVehicle below,

```
public abstract class LandVehicle {
  int doors;
  LandVehicle() { doors = 4; }
  void drive();
}
```

the following statement will be considered as invalid:

```
LandVehicle l = new LandVehicle();
```

6.8.5 Extending Interface

Does a subclass of a class that implements an interface also inherit the methods of the interface? Consider the code in Listing 6-6:

```
interface I {
  void x();
}

class A implements I {
  public void x() { System.out.println("in A.x"); }
  public void y() { System.out.println("in A.y"); }
}

class B extends A {
  void z() {
    x();
    y();
  }

  public static void main(String args[]) {
    A aa = new A();
    B bb = new B();
    bb.z();
  }
}
```

Listing 6-6: Extending Interface to Subclasses

The following output

```
in A.x
in A.y
```

suggests that the methods `x()` and `y()` of class `A` have been invoked. Class `B`, being the subclass of class `A`, inherited not only method `y()` but also method `x()` which is a method of the interface `I`.

6.8.6 Limitations of Interface for Multiple Inheritance

Although the interface feature in Java provides an alternative solution to achieving multiple inheritance in class hierarchies, it has its limitations:

a. An interface does not provide a natural means of realizing multiple inheritance in situations where there is no inheritance conflict.
b. While the principal reason for inheritance is code reusability, the interface facility does not encourage code reuse since duplication of code is inevitable.

(a) No Inheritance Conflict

Consider the situation in Figure 6-17 where there is no inheritance conflict in attributes or methods.

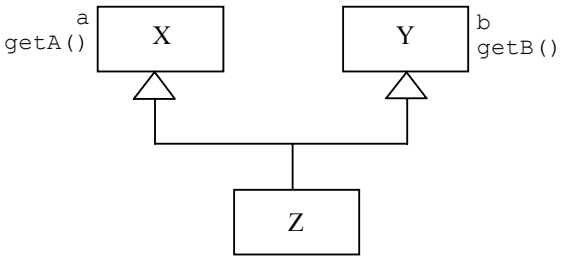


Figure 6-17: No inheritance conflict.

A Z object has attributes `a` and `b` together with methods `getA()` and `getB()`. Without the support of multiple inheritance in Java, such class hierarchy cannot be realized.

(b) No Code Reuse

Let us, as examples, consider land vehicles such as motor cars and trucks. A class hierarchy showing Motor Car and Truck as subclasses of LandVehicle is given in Figure 6-18.

Motor Car and Truck have been declared as subclasses to distinguish the different brake system used. By means of inheritance, data attributes `regnNumber` and `numberOfPassenger` of LandVehicle are inherited by Motor Car and Truck as part of their properties.

Let us extend this example in our discussion of the issue—no code reuse and further extend the class hierarchy to include information that distinguishes the drive system between the land vehicles. We will consider three alternative representations.

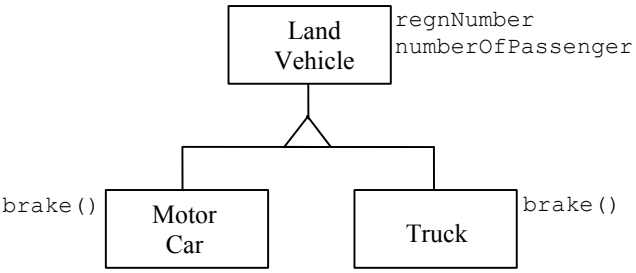


Figure 6-18: Motor car and truck.

In the first representation, the front-drive feature of a vehicle is defined as a superclass in Figure 6-19. The implementation of `drive()` method for the vehicles is indicated by the two statements:

```
frontWheelSys.engage();
this.accelerate();
```

The Front-Wheel-Drive class is an abstract class with a `drive()` method. This suggests that all motorcars and trucks are front-wheel-drive vehicles. This is incorrect as some motorcars and trucks may be back-wheel-drive vehicles. The class hierarchy in Figure 6-19 is therefore inappropriate for representing motor cars and trucks.

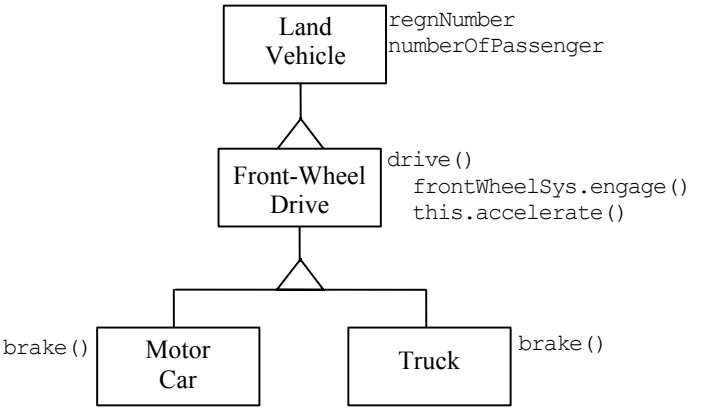


Figure 6-19: "Front-Wheel Drive Class as Superclass" Representation Scheme

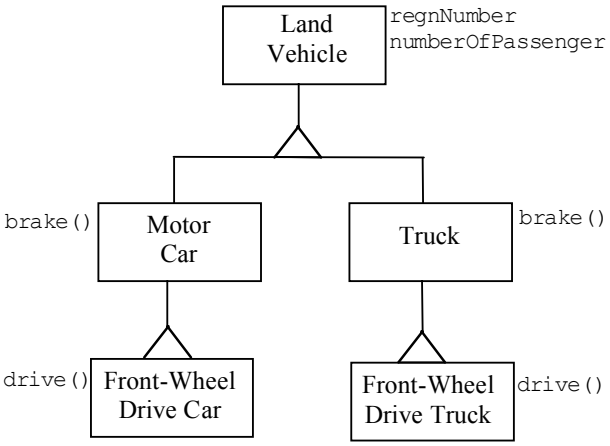


Figure 6-20: Front-Wheel Drive class as subclasses: representation scheme.

In the second representation, the front-drive feature is contained within a sub-class, Front-Wheel-Drive Car or Front-Wheel-Drive Truck in Figure 6-20. While this is semantically correct in satisfying the requirement, it does not take full advantage of inheritance – the `drive()` method is duplicated in Front-Wheel-Drive Car and Front-Wheel-Drive Truck class.

In the last representation, the `drive()` method is abstracted into a separate super-class and inherited by Front-Wheel-Drive Car and Front-Wheel-Drive Truck through a multiple inheritance chain in Figure 6-21.

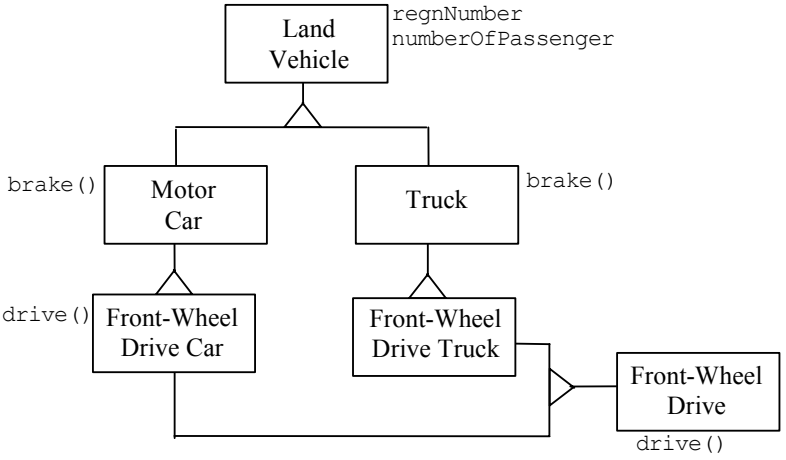


Figure 6-21: "Multiple inheritance" representation scheme.

This hierarchy is obviously desirable but implementing the solution with the interface construct would result in the definition of `drive()` method as an abstract method of a Front-Wheel-Drive interface and implemented in the Front-Wheel-Drive Car and Front-Wheel-Drive Truck classes. The effect is the same as that for the

"Front-Wheel Drive as Subclasses" representation scheme where the implementation of the `drive()` method was duplicated in the two subclasses.

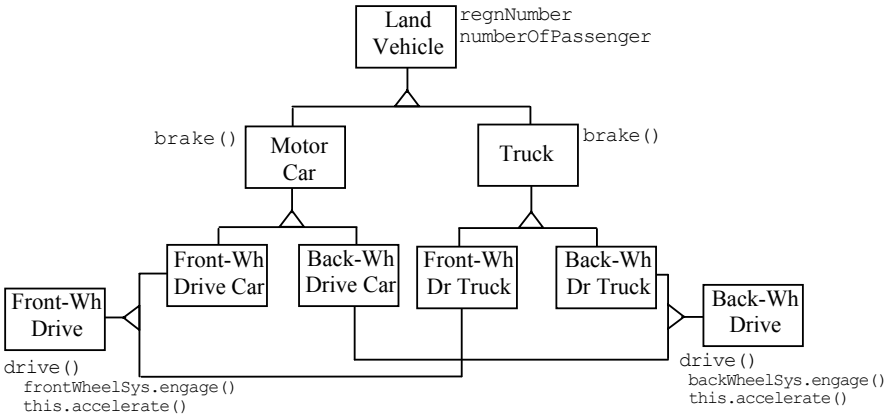


Figure 6-22: Front-Wheel and Back-Wheel Drive vehicles.

The problem with the interface solution is further amplified when we consider the need to implement the `drive()` method for Back-Wheel-Drive vehicles as well (see Figure 6-22). Using the interface solution, we need to implement the `drive()` method in Front-Wheel-Drive Car, Back-Wheel-Drive Car, Front-Wheel-Drive Truck, and Back-Wheel-Drive Truck. This approach can be error-prone.

From the above example, it is clear that the interface construct does not encourage code reuse and duplication of method implementation is inevitable. If multiple inheritance had been available in Java, implementing the `drive()` method would be easier.

6.9 Summary

The following points were discussed in this chapter:

- A class can take on the properties of a superclass via the *inheritance* mechanism.
- Inheritance is the ability of a subclass to take on the general properties of classes higher up in a class hierarchy.
- Properties can only be propagated downward from a superclass to a subclass.
- Inheritance enables code reuse.
- Inheritance enhances software maintainability.
- Inheritance enables class extension through subclassing.
- A class that takes on properties from only one superclass is said to exhibit *single inheritance*.

- A class that takes on properties from two or more superclasses is said to exhibit *multiple inheritance*.
- Multiple inheritance is not implemented in Java, hence, the interface construct implemented via the `interface` keyword is an alternative solution to achieve multiple inheritance. However, this solution has its limitations.

6.10 Exercises

The following points were discussed in this chapter:

1. Define and distinguish the terms *single inheritance* and *multiple inheritance*.
2. Give an example of multiple inheritance in a real-life situation.
3. A and B are two classes. A inherits properties from B, so A is a _____ class of B and B is a _____ class of A.

If A has attributes `a1` and `a2`, methods `getA1()` and `getA2()`, and B has attributes `b1` and `b2`, methods `getB1()` and `getB2()`, then by means of inheritance, the actual definition of A and B would be:

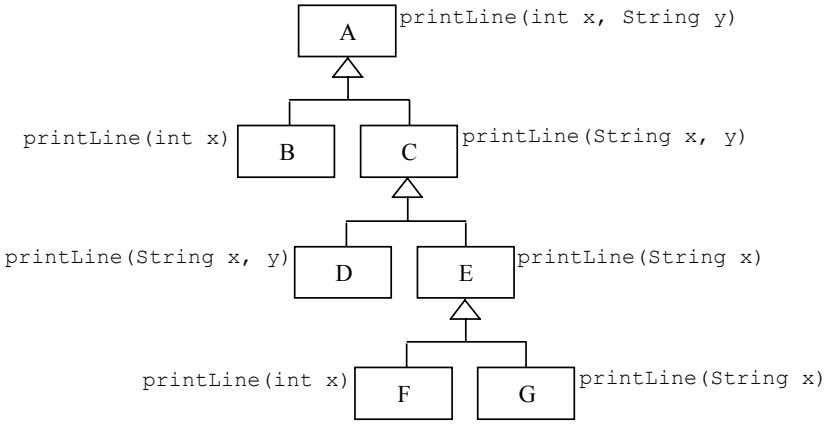
```
class A {                          class B {
Attributes:                        Attributes:
 __________                         __________
 __________                         __________
 __________                         __________
 __________                         __________
Methods:                           Methods:
 __________                         __________
 __________                         __________
 __________                         __________
 __________                         __________
```

4. Given the following information on classes A and B, declare A and B in Java:

   - attributes `a1`, `b1` are integers;
   - attributes `a2`, `b2` are strings;
   - method `getA1()`, `getA2()`, `getB1()`, `getB2()` returns `a1`, `a2`, `b1`, and `b2`, respectively.

5. How does inheritance contribute to software reusability?
6. Given the following class hierarchy diagram,

which class's `printLine()` method would be used for each of the messages below (assuming `z` is an object instantiated from class F):



a) `z.printLine(1)`

b) `z.printLine(2, "Object-Oriented Programming")`

c) `z.printLine("Java")`

d) `z.printLine("Object-Oriented Programming", "Java")`

e) `z.printLine("Object-Oriented Programming", 3)`

7. What can you say about the method `printLine(String x)` of class G in Question 6?
8. Distinguish between contract inheritance and implementation inheritance.
9. Discuss the problem associated with multiple inheritance. How does Java overcome it and what feature is provided in Java to achieve multiple inheritance? Discuss the limitation of this feature.
10. An abstract class contains one or more _____ methods. (Fill in the blank.)Distinguish between an abstract class and an interface. Which is better and why?
11. What is the expected output for the code in Listing 6-7.

```
interface I {
  void x();
  void y();
}

class A implements I {
  A() {}
  public void w() {System.out.println("in A.w");}
  public void x() {System.out.println("in A.x");}
  public void y() {System.out.println("in A.y");}
}

class B extends A {
  B() {}

  public void y() {
    System.out.println("in B.y");
  }
```

```
  void z() {
    w();
    x();
  }

 static public void main(String args[]) {
    A aa = new A();
    B bb = new B();
    bb.z();
    bb.y();
  }
}
```

Listing 6-7

7

Polymorphism

Up till now we have been assuming all code is bound during compilation. Such binding is known as *static binding*. Binding can also take place at run-time and this form of binding is known as *dynamic binding*.

Static binding is limited and may lead to difficulty in software maintenance. Dynamic binding, on the other hand, provides design flexibility and may enhance software maintainability.

In this chapter, we will discuss static binding and dynamic binding. We will also examine how dynamic binding contributes to *polymorphism*—the ability of objects to respond to the same message with the appropriate method based on their class definition.

7.1 Static Binding

In Chapter 6, we introduced an example on shapes. Let us now examine how the conventional procedural programming approach handles the code for the Shape example.

In Listing 7-1, we list the pseudo-code for the Shape example using procedural declaration. Two shapes, a circle and a square, in the form of a record structure have been declared in the data section. Each shape has a shape name. The shape name for circle `c` is "circle C" and the shape name for square `s` is "square S." Circle C has a variable `radius` while square `S` has a variable `side`. In addition, an array, `shapeArray`, has been declared to contain up to five characters, each character referencing a shape.

```
Data Section
Type
  Circle = Record
             String name;
               int radius;
           End
  Square = Record
             String name;
               int side;
           End
Variable
  shapeArray : Array [1..5] of char;
  c : Circle;
  s : Square;
Main Code Section
  c.name  = "Circle C";
  c.radius = 2;
  s.name  = "Square S";
  s.side    = 3;
  A[1] = 'c';
  A[2] = 's';
  For int i = 1 to 2 do  {
    Switch shapeArray[i]
      'c' : calculateCircleArea();
      's' : calculateSquareArea();
      'n' : do nothing;
    End (Case)
  }

Routine Section
calculateCircleArea()   {
  float area = 3.14 * c.radius * c.radius;
  writeln ("The area of ", c.name, " is ", area, " sq. cm.");
}

calculateSquareArea()   {
  float area = s.side * s.side;
  writeln ("The area of ", s.name, " is ", area, " sq. cm.");
}
```

Listing 7-1: Procedural declaration of Shape example.

In the main code section, the circle and square have been initialized with some parameters. Iterating through the array, the area of circle C and square S is produced and printed.

```
Data Section
Type
  ...
  Triangle = Record
               String name;
               int base, height;
             End
Variable
  ...
  t : Triangle;

Main Code Section
```

```
  ...
  t.name   = "Trianlge T";
  t.base   = 4;  t.height = 5;
  ...
  A[3] = 't';
  For int i = 1 to 3 do  {
    Switch shapeArray[i]
      'c' : calculateCircleArea();
      's' : calculateSquareArea();
      't' : calculateTriangleArea();
      'n' : do nothing;
   End (Case)
  }

Routine Section
calculateCircleArea()   {
  ...
}

calculateSquareArea()   {
  ...
}

calculateTriangleArea()   {
  float area = 0.5f * t.base * t.height;
  writeln ("The area of ", t.name, " is ", area, " sq. cm.");
}
```

Listing 7-2: Static binding: adding a triangle.

Since each array element may reference a circle, a square, or something else (we do not really care which one), a `case` statement for determining which of the shape's `calculate area` routine to execute given a choice has to be coded in the program and bound at compile time. For example, a `calculateCircleArea()` routine has to be called for a circle and a `calculateSquareArea()` routine has to be called for a square, and so on.

Two distinct `calculate area` routines are required because the method for calculating the area of a circle is different from that for a square. Also, most procedural programming languages do not allow two routines to have the same name. Different routine names therefore have to be devised.

We say that the variable `shapeArray[i]` (or the choices) is *statically bound* to the routine `calculateCircleArea()` in the case when the value of `shapeArray[i]` is "c" and `calculateSquareArea()` when the value of `shapeArray[i]` is "s."

With static binding, problems may arise in code maintenance. To illustrate, let us now add another shape, Triangle, to the code. Listing 7-2 shows the solution with changes highlighted in bold. Note that changes were made at the following points:

- in the data section where a triangle is defined;
- in the main code section where the detection of a triangle and the appropriate routine call have to be included in the `switch` statement; and
- in the routine section where the specific routine for calculating the area of the triangle has to be included.

Multiple places are affected as a result of extending shape types, and this is error prone.

7.2 Dynamic Binding

An alternative approach to static binding is *dynamic binding*. Here, the binding of variables to routines (or *methods* in object-oriented programming terms) is done at run time. In Listing 7-3, an object-oriented version of the previous Shape example is produced. The code here resembles the one used in Chapter 6.

```
class Shape {
  private String name;
  Shape(String aName) {name=aName;}
  public String getName() {return name;}
  public float calculateArea() {return 0.0f;}

  public static void main(String argv[]) {
    Circle c = new Circle("Circle C");
    Square s = new Square("Square S");
    Shape shapeArray[] = {c, s};
    for (int i=0; i<shapeArray.length; i++) {
      System.out.println("The area of " + shapeArray[i].getName()
          + " is " + shapeArray[i].calculateArea()+" sq. cm.\n");
    }
  }
}

class Circle extends Shape {
  private int radius;
  Circle(String aName) {
    super(aName);
    radius = 3;
  }

  public float calculateArea() {
    float area;
    area = (float) (3.14 * radius * radius);
    return area;
  }
}

class Square extends Shape {
  private int side;
  Square(String aName) {
    super(aName);
    side = 3;
  }

  public float calculateArea() {
    int area;
    area = side * side;
    return area;
  }
}
```

Listing 7-3: Dynamic binding—Shape, Circle and Square classes.

Two objects, a circle and a square, are created by the first two statements of `main()`. Object variables for the circle and square are kept in the array, `shapeArray`, and iterating through the array, the area of the respective object is produced and printed by the statement:

```
System.out.println("The area of " + shapeArray[i].getName()
                + " is " + shapeArray[i].calculateArea()+" sq. cm.\n");
```

While the actual routine for the choice of shape in the array elements has to be pre-determined via a `switch` statement in static binding, `switch` statement is not required with dynamic binding.

Based on the output from the code, it is clear that the appropriate method for responding to the choice in `shapeArray` has been used:

```
The area of Circle C is 28.26 sq. cm.
The area of Square S is 9 sq. cm.
```

The method has been selected based on the class of the shape referenced in `shapeArray` at run-time. This is only possible in programming languages that support dynamic binding. With dynamic binding, the variable `shapeArray[i]` is bound to an object method at run time when the class definition of the shape referenced is known.

7.3 Operation Overloading

Circle and Square have similar `calculateArea()` method in their class definition. Although both methods have the same method signature, they have different method implementation, since the formula for calculating area of each is not the same.

While it is impossible in conventional imperative programming languages to have two routines having the same name, it is allowed in object-oriented programming. The ability to use the same name for two or more methods in a class is known as *operation overloading* in object-oriented terms (see also Section 3.6).

7.3.1 Same Method Signature

Two methods are said to have the same method signature if:

- the name of the methods are the same; and
- the number and type of formal parameters are the same.

The `calculateArea()` method of the Shape, Circle and Square classes, in Listing 7-3, is said to have the same method signature.

7.3.2 Overloading Method Names

Methods having the same method signature may pose problems in the compilation process depending on where they are used in the program. In this section, we will consider various situations and report on the validity of overloaded method names.

In the code fragment below, method `A()` is overloaded by `A(int x)`, `A(int x, int y)`, and `A(String s)`. These four methods are distinguished in the compilation process by the number and type of parameters present in the method call.

```
class A {
  ...
  A() { ... }
  A(int x) { ... }
  A(int x, int y) { ... }
  A(String x) { ... }
  ...
}
```

Indicated below are the actual methods called given the message on the left:

```
A thisA = new A();                  ➔   A()
A thisA = new A(3);                 ➔   A(int x)
A thisA = new A(4, 5);              ➔   A(int x, int y)
A thisA = new A("hello");           ➔   A(String x)
```

In the code fragment below, the definition of method `a1()` is reported as a duplicate method declaration by the compiler since the two methods have been declared in the same class. It is thus considered to be invalid.

```
class A {
 A() {}
 public void a1() {}
 public void a1() {}
}
```

The return type of a method does not distinguish overloaded methods from one another as the following example shows. Method `a1()` is flagged as invalid by the compiler as both of them are considered similar.

```
class A {
 A() {}
 public void a1() {}
 public void a1() {}
}
```

However, declaring methods of the same signature in different classes are considered as valid in object-oriented programming:

```
class A {
  A() {}
  public void a1() {}

  public static void main(String args[]) {
  }
}

class B {

  B() {}
  public void a1() {}
  public void b1() {}
}
```

Finally, consider the following code fragment:

```
class C {
  C() {}
  public void c1() {System.out.println("C.c1()");}
}

class D extends C {
  D() {}
  public void c1() {
    super.c1();
    System.out.println("D.c1()");
  }
  public void d1() {}

  public static void main(String args[]) {
    D thisD = new D();
    thisD.c1();
  }
}
```

Although method `c1()` is defined in two different classes, the situation is different from the previous case. In this case, method `c1()` is defined in subclass D and superclass C. The declaration of method `c1()` in this case is considered valid as in the previous case. However, the code produces the following output when it is run:

```
C.c1()
D.d1()
```

Method `c1()` in subclass D is said to *redefine* (or *override*) method `c1()` of superclass C. For a method to redefine a superclass's method, the method signature of the two methods must be the same; otherwise, they are considered as two different methods as shown by the output from the code below:

```
class C {
  C() {}
  public void c1() {System.out.println("C.c1()");}
}

class D extends C {
  D() {}
  public void c1(int i) {
    super.c1();
    System.out.println("D.c1()");
  }
  public void d1() {}

  public static void main(String args[]) {
    D thisD = new D();
    thisD.c1();
    thisD.c1(3);
  }
}
```

Output

```
C.c1()
C.c1()
D.c1()
```

7.4 Polymorphism

We noted earlier in Section 7.2 that we can achieve code binding at run-time with dynamic binding. Also, appropriate method call can be made without making any direct reference to it in the message. As is evident in the output of the code, the appropriate `calculateArea()` method for the respective object was selected.

It is apparent that the message (`calculateArea()`) from the sender (`main()`) has been interpreted appropriately by the receiver. A circle receiving the message has used its own method `calculateArea()` to calculate the area and a square receiving the same message has done the same with its own `calculateArea()` method. The ability of *different* objects to perform the appropriate method in response to the *same message* is known as *polymorphism* in object-oriented programming.

7.4.1 Selection of Method

In polymorphism, the interpretation of a message is not affected by a sender. What a sender knows is that an object can respond to a message but it does not know which class the object belongs to or *how* it will respond to the message. For example, the message `shapeArray[i].calculateArea()` of `main()` (see Listing 7-3) is sent to a Shape object (a circle or a square). The sender (`main()`) does not know which of the Shape objects will respond to the message, let alone *how* it will perform the method.

The selection of the appropriate method depends on the *class* the object belongs to. For a circle object, the `calculateArea()` method of the Circle class will be

called and for a square object, the `calculateArea()` method of the Square class will be called. Since the first element in `shapeArray` is a circle, the `calculateArea()` method of the Circle class is executed producing an area of 28.26 cm^2. Similarly, the `calculateArea()` method of the Square class is performed for the second element of `shapeArray`, resulting in the value 9 cm^2. The `calculateArea()` method of the Circle and Square class is thus said to be *polymorphic*.

7.4.2 Incremental Development

Polymorphism is facilitated by dynamic binding and the ability to use the same name for similar methods across class definitions. It would not be possible to achieve polymorphism if a programming language does not support these facilities.

Polymorphism encourages programmers to specify *what* method should happen rather than *how* it should happen. This approach allows flexibility in code design and promotes incremental program development. To illustrate, we will consider adding a new Triangle object type into the code of Listing 7-3. Listing 7-4 is the modified code with additional code highlighted in bold.

Note that minimal changes were made to the original code of Listing 7-3. The main changes occur in the following areas:

- A new Triangle class was added. The addition of the class does not affect the other parts of the program.
- A statement was added in `main()` to create the Triangle object.
- A statement was added in `main()` to include the newly created triangle into the `shapeArray`.

No change was made in the `println` statement in `main()`. No `switch` statement is required to determine which method to use. This example shows that adding a new object type is made easy with polymorphism, and there is greater scope for incremental development.

```
class Shape {
  ...
  public static void main(String argv[]) {
    Circle c = new Circle("Circle C");
    Square s = new Square("Square S");
    Triangle t = new Triangle("Triangle T");
    Shape shapeArray[] = {c, s, t};
    for (int i=0; i<shapeArray.length; i++) {
      System.out.println("The area of " + shapeArray[i].getName()
           + " is " + shapeArray[i].calculateArea()+" sq. cm.\n");
    }
  }
}
class Circle extends Shape {
...
}
class Square extends Shape {
 ...
}
```

```
class Triangle extends Shape {
  private int base, height;

  Triangle(String aName) {
    super(aName);
    base   = 4;  height = 5;
  }

  public float calculateArea() {
    float area = 0.5f * base * height;
    return area;
  }
}
```

Listing 7-4: Dynamic binding—adding a triangle.

7.4.3 Increased Code Readability

Polymorphism also increases code readability since the same message is used to call different objects to perform the appropriate behavior. The code for calling methods is greatly simplified as much of the work in determining which class's method to call is now handled implicitly by the language. The simplicity of the code is evident in the `println()` statement of Listing 7-3.

7.5 Summary

In this chapter, we discussed:

- *Static binding*—the binding of variables to operations at compile time;
- *Dynamic binding*—the binding of variables to operations at run time;
- *Operation overloading*—the ability to use the same name for two or more methods in a class; and
- *Polymorphism*—the ability of *different* objects to perform the appropriate method in response to the *same message*.

7.6 Exercises

1. Discuss the two facilities required in programming languages for supporting polymorphism.
2. How does polymorphism contribute to software maintainability?
3. Contrast between "method redefinition" and "operation overloading."

8

Modularity

We have so far discussed the basic facilities for creating objects through Java class definitions and code reuse by inheriting properties of similar but more general classes.

In this chapter, we look at the important issue of modularity and the related mechanisms available in Java.

8.1 Methods and Classes as Program Units

A method is comprised of statement sequences and is often viewed as the smallest program unit to be considered as a subprogram. It is self-contained and designed for a particular task which represents an object behavior.

Together with data, a coordinated set of methods completes the specification of objects. As we have seen, data and methods are the constituents of a class definition. Compared to a method, a class definition is the next bigger unit under design.

Properties defined in a class can be distinguished into object and class properties. Object properties are definitions that are specific to objects and apply to all objects from the same class. Class properties, on the other hand, apply only to the class even though the structure and behavior of objects of a class is defined by the class.

8.2 Object and Class Properties

In this section, we will examine the distinction between object and class properties.

8.2.1 Counting Instances

Listing 8-1 contains the code for an example that counts the number of objects instantiated from the SalesPerson class.

```
class SalesPerson {
  String employeeId;

  SalesPerson(String aEmployeeId) {
    employeeId = aEmployeeId;
  }

 public static void main(String arg[]) {
    int count = 0;
    SalesPerson s1 = new SalesPerson("12345S");
    count = count+1;
    SalesPerson s2 = new SalesPerson("33221K");
    count = count+1;
    System.out.println(count + " salespersons have been created");
  }
}
```

Listing 8-1: Counting instances.

The code begins with the declaration of a variable `count`. This variable is used to continually create the number of SalesPerson objects. For each creation of a SalesPerson object, `count` is incremented via the statement:

```
count = count+1;
```

This statement is executed twice since two SalesPerson objects were instantiated. Finally, the code prints out the number of SalesPerson objects instantiated via the statement

```
System.out.println(count + " salespersons have been created");
```

The output:

```
"2 salespersons have been created"
```

suggests that two SalesPerson objects were created, and this is clearly correct.

While the solution is correct, it is cumbersome since for each new instantiation of SalesPerson object, a fresh

```
count = count+1;
```

statement has to be added into `main()`.

An alternate class organisation is given in Listing 8-2, where `count` is incremented within the constructor method of the SalesPerson class. This strategy supports abstraction and is advantageous because the user does not have to bother with operations on `count`.

```
class SalesPerson {
   String employeeId;
   int count = 0;
   SalesPerson(String aEmployeeId) {
      employeeId = aEmployeeId;
      count = count + 1;
   }
   int getCount() { return count; }
   public static void main(String argv[]) {
      SalesPerson s1 = new SalesPerson("12345S");
      SalesPerson s2 = new SalesPerson("33221K");
      System.out.println(s1.getCount() +
                         " salespersons have been created");
   }
}
```

Listing 8-2: Alternative solution to counting instances.

It is clear from the output:

```
"1 salespersons have been created"
```

that the result is incorrect.[1]

The variable `count` in the alternative solution is declared as an object attribute rather than a local variable of the static method `main()` as was the case for the previous solution. As an object attribute, `count` can now be incremented in the constructor method of the SalesPerson class. However, with each instantiation of a SalesPerson object, the `count` variable of each newly created object is incremented. Since two instances of SalesPerson object were created, two independent copies of `count`, each having the value 1, were present. Figure 8-1 shows the state of the two created SalesPerson objects.

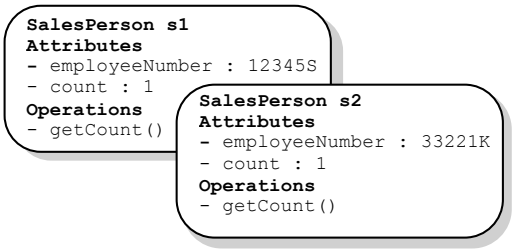


Figure 8-1: State of SalesPerson objects.

[1]Even if `s1.getCount()` has been substituted with `s2.getCount()` in the `println()` method, the result will still be incorrect.

Although only one copy of `count` is required, it is unclear which copy of the two instances should be used.

8.2.2 Shared Attributes

Another solution can be found in Listing 8-3 whereby `count` is declared as `static`. Declaring `count` as `static` allows the variable `count` to be shared among all instances of the SalesPerson class. Thus, `s1.count` refers to the same memory location as `s2.count`. The statement:

```
count = count+1;
```

in the constructor method therefore increments the same shared copy of `count` as shown in Figure 8-2. The output from the code:

```
"2 salespersons have been created"
```

is correct.

```
class SalesPerson {
   String employeeId;
   static int count = 0;
   SalesPerson(String aEmployeeId) {
      employeeId = aEmployeeId;
      count = count + 1;
   }
   int getCount() { return count; }
   public static void main(String argv[]) {
      SalesPerson s1 = new SalesPerson("12345S");
      SalesPerson s2 = new SalesPerson("33221K");
      System.out.println(s1.getCount() +
                         " salespersons have been created");
   }
}
```

Listing 8-3: Shared attributes.

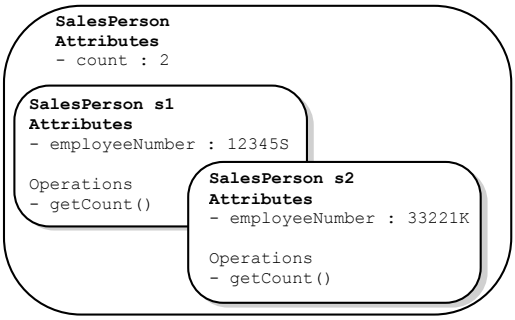


Figure 8-2: Shared variable `count`.

8.2.3 Class Attributes

The `static` variable `count` is also known as a *class attribute*. While a class definition specifies the structure and behavior of objects within, it may have its own attribute and method definitions.

An attribute definition that is preceded with the keyword `static` is a class attribute. While we previously viewed the `static` variable `count` as shared amongst all instances, its association with the class is consistent. As a class attribute, it is also accessible to instances of the class.

8.2.4 Class Methods

In another change in Listing 8-4, we make `getCount()` a class method by prefixing it with the `static` keyword. The results from the code is the same as that of Listing 8-3 with just `static count`.

Note the difference in representation between the `getCount()` method of Figure 8-3 for Listing 8-4 and Figure 8-2 for Listing 8-3. In Figure 8-3, both SalesPerson objects `s1` and `s2` do not own the `getCount()` method since the method belongs to the SalesPerson class as represented by the outermost bubble surrounding instances `s1` and `s2`.

```
class SalesPerson {
   String employeeId;
   static int count = 0;
   SalesPerson(String aEmployeeId) {
      employeeId = aEmployeeId;
      count = count + 1;
   }
   static int getCount() { return count; }
   public static void main(String argv[]) {
      SalesPerson s1 = new SalesPerson("12345S");
      SalesPerson s2 = new SalesPerson("33221K");
      System.out.println(s1.getCount() +
                         " salespersons have been created");
   }
}
```

Listing 8-4: Class methods.

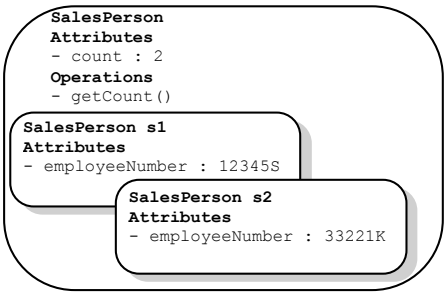


Figure 8-3: Class Method `getCount()`

8.2.5 Name Aliases

Within the class definition of SalesPerson in Listing 8-4, the method names `getCount()`, `s1.getCount()`, `s2.getCount()` and `SalesPerson.getCount()` are all aliases that reference the `static` method `getCount()`. `getCount()` is typically the most convenient usage within the class construct. Outside of the class definition (where there may be several `getCount()` methods in other class definitions), qualification by the class name as in `SalesPerson.getCount()` is the only way to access the method. Thus, within `static void main()`, the `println()` method could have been:

```
System.out.println(getCount() +
                   "salespersons have been created");
```

8.3 Controlling Visibility

Except for the discussion on "Representational Independence" in Section 3.5, we have mostly ignored the issue of visibility of attributes and methods. Any discussion on modularity is not complete without discussing visibility issues.

First, while modules should be as independent as possible with minimal coupling, no module can be totally isolated from other code since it is unusual for a module to work in isolation. Thus, there must be entities on an object that are accessible externally.

On the other hand, objects should reveal as little as possible of their internal workings so that there will be minimal dependence on such details. Ideally, objects will reveal information on a need-to-know basis.

We have earlier seen the use of the visibility specifiers `private` and `public` in Section 3.5. They precede attribute and method definitions. Both access control specifiers function at the extreme ends of the visibility spectrum. The `private` specifier makes entities that follow it hidden from code fragments external to the class. The `public` specifier makes entities that follow it fully visible from all other Java code.

We modify our SalesPerson class in Listing 8-5 so that `count` has the `private` access specifier, while `employeeId` and `getCount()` are declared as `public`. This means that the variable `count` is not visible by any code outside the class definition. As such, any part of the program outside the class definition of SalesPerson cannot *directly* access the `count` variable.

Private variables are either used within the class definition, or a `public` accessor method is implemented to provide the access required outside the class definition. We assume the latter to be the case and provided a publicly accessible `getCount()` to return the value of `count`.

```
class SalesPerson {
   public String employeeId;
   private static int count = 0;
   SalesPerson(String aEmployeeId) {
      employeeId = aEmployeeId;
      count = count + 1;
   }
   public static int getCount() { return count; }
   public static void main(String argv[]) {
      SalesPerson s1 = new SalesPerson("12345S");
      SalesPerson s2 = new SalesPerson("33221K");
      System.out.println(s1.getCount() +
                         " salespersons have been created");
   }
}
```

Listing 8-5: Restricted access of `count`.

Between the `public` and `private` extremes, Java also allows for two more categories: `protected` and the default visibility of "friendly."



While the `public` and `private` specifiers allow for all-or-nothing access outside of the class definition, the `protected` keyword makes entities that follow it accessible to code fragments with its immediate subclass. For entities with no access specifier, friendly access is assumed. Here, access is given to code fragments within the same package. Table 8-1 summarizes accessibility rules from most restrictive to least restrictive.

Table 8-1: Access specifiers.

Keyword	Visibility
`private`	Access to a `private` variable or method is only allowed within the code fragments of a class.
`pro-tected`	Access to a `protected` variable or method is only allowed within the code fragments of a class and its subclass.
(friendly)	Access to a friendly variable or method (with no access specifier) is only allowed within the code fragments of a class and any other class in the same package.
`public`	Access to a `public` variable or method is unrestricted. It may be accessed from the code fragments of any class.

8.4 Packages

While class definitions are practical modular units, the Java programming language has another mechanism that facilitates programming teams and reusable software units. The package facility allows for appropriate classes to be grouped into packages. As with standard design rationale for objects where relevant methods are placed in the same class definition, packages in Java form the next level of software containment for classes with logically related functionality.

Packaging also partitions the name space to avoid name clashes. Computations in Java are reliant on objects, and the result of system design is a set of class definitions. Where teams of programmers work independently with the intention of the results to be subsequently integrated, there is a chance that they may choose the same name for their classes. Packaging thus allows for the names of classes to be confined to the originating package.

8.4.1 The `package` Keyword

Package hierarchy is specified via the `package` keyword preceding a class definition as shown below. Here, class XYZ belongs within package A. Its complete qualified name is thus A.XYZ.

```
package A;
class XYZ {
  int h;
  void j() { ... }
}
```

It follows that a "`package B.C`" prefix before "class RST" definition makes class RST belong to the C package that is in the B package.

```
package B.C;
class RST {
  int y;
  void z() { ... }
}
```

Thus far, we have been using class definitions but without any package prefix. Such classes belong to the top-level anonymous package. As for locating and loading code files, the Java virtual machine maps the package hierarchy onto its directory structure. As such, code for A.XYZ will be at XYZ.class in directory A, or more succinctly, the pathname A/XYZ.class. Similarly, B.C.RST will be found at B/C/RST.class.

8.4.2 The `import` Keyword

The `import` keyword provides the complement function of the package facility. While the `package` keyword places succeeding class definitions in a separate name space of the said package, the `import` keyword makes package constituents visible.

Continuing from our earlier RST example, any client code outside of package B.C must either refer to it in its qualified form:

```
class another {
   B.C.RST x = new B.C.RST();
   ...
}
```

or, import the complete name space of the B.C package

```
import B.C.*;
class another {
   RST x = new RST();
   ...
}
```

or import just the single class

```
import B.C.RST;
class another {
   RST x = new RST();
   ...
}
```

Since the class RST is now used outside its package, RST must be a public class.

```
package B.C;
public class RST {
  int y;
  void z() { ... }
}
```

8.5 Encapsulation

Another concept in object-oriented programming closely associated with modularity is *encapsulation*. Simply, encapsulation means the bringing together of a set of attributes and methods into an object definition and hiding their implementational structure from the object's users. Therefore, how an object structures and implements its attributes and methods is not visible to other objects using it. Direct access to an object's attributes is not permitted and any changes to the object's data can only be effected indirectly via a set of publicly available methods.

Analogically speaking, encapsulation can be compared to the way an egg is formed. Within an egg is a yolk that is surrounded by the white. To get to the yolk, one has to traverse through the white. Thought of in this way, the data of an object is like the yolk and the methods are like the white. Data is thus protected by methods—that is, access to the data is only permissible via the methods.

Access control specifiers introduced in Section 8.3 facilitate encapsulation by controlling the visibility of data and methods by other objects.

8.5.1 Bundling and Information Hiding

Encapsulation is supported by two subordinate concepts: *bundling* and *information hiding*. Bundling is the act of associating a set of methods with a set of data as the only means of affecting the values of the data. Related data and methods are therefore brought together in bundling, thus increasing the cohesiveness of object definition.

Information hiding refers to the hiding of internal representation of data and methods from the users of these data and methods. By exercising information hiding, data access on an object is limited to a set of publicly available methods. While the client is aware of the existence of the methods, it does not know how the methods are internally structured. In this way, information hiding enables the separation of the *what* from the *how* of object definition. *What* specifies what behavior an object is capable of and *how* specifies how the data and methods of an object are implemented.

8.5.2 Enhanced Software Maintainability

By separating *what* from *how*, a client's access to an object is not affected by changes in the internal implementation of the object. This enhances software maintainability.

To illustrate, let us consider an example using stack, which is a software construct with operations such as `push()`, `pop()`, `empty()`, `full()` and `size()`. The operation `push()` adds an item into a data structure in the stack. The operation `pop()` removes the most recently pushed item. The operation `empty()` returns true if the stack is empty and `full()` returns true if the data structure has reached its limit. The operation `size()` returns the current number of items pushed into the stack.

There are two possible ways of implementing the stack—using *array* or *linked list* to store items pushed into the stack. The array implementation is shown in Listing 8-6.

```
class Stack {
  private int contents[];
  private int top, size=10;

  public int pop() {
    int x = 0;
    if (empty()) System.err.println("stack underflow");
    else x = contents[top--];
    return(x);
  }
  public void push(int x)  {
    if (full()) System.err.println("stack overflow");
    else {contents[++top] = x;
          System.out.println("Pushed "+x+" into Stack");}
  }
  public int size() { return(top+1); }
  public boolean empty() { return(size() == 0); }
  public boolean full() { return(size() == contents.length); }

   Stack() { contents = new int[size]; top = -1; }
```

```
  public static void main(String argv[]) {
    int i, numberOfItem;
    numberOfItem=Integer.parseInt(argv[0]) ;
    Stack s = new Stack();
    for (i = 0; i<numberOfItem; i++)
      s.push(i);
    System.out.println("\nDetails of Stack : ");
    for (i = numberOfItem; i>0; i--)
      System.out.println("Item popped = "+s.pop());
  }
}
```

Listing 8-6: Stack using an array implementation.

In this implementation, a number indicating the number of items for the array is entered via the main prompt and converted into an integer via the statement:

```
numberOfItem = Integer.parseInt(argv[0]);
```

`numberOfItem` is later used in a `for` loop

```
for (i = 0; i<numberOfItem; i++) s.push(i);
```

to push integers into a stack created by the statement:

```
Stack s = new Stack();
```

The `push()` method first checks if the stack is already full; if not, the item pushed is inserted into the array `contents`. An error message indicating "stack overflow" is displayed if the stack is already full. Finally, the pushed items are popped via the `for` statement:

```
for (i = numberOfItem; i>0; i--)
  System.out.println("Item popped = "+s.pop());
```

To test the program, a value 5 is entered:

```
$ java Stack 5
```

The output is as follows:

```
Pushed 0 into Stack       Details of Stack :
Pushed 1 into Stack       Item popped = 4
Pushed 2 into Stack       Item popped = 3
Pushed 3 into Stack       Item popped = 2
Pushed 4 into Stack       Item popped = 1
                          Item popped = 0
```

It indicates that the implementation is correct with an input value 5. However, when an input value 12 is entered, the weakness of the array implementation is reflected in the output:

```
Pushed 0 into Stack
Pushed 1 into Stack
Pushed 2 into Stack
Pushed 3 into Stack
Pushed 4 into Stack
Pushed 5 into Stack
Pushed 6 into Stack
Pushed 7 into Stack
Pushed 8 into Stack
Pushed 9 into Stack
stack overflow
stack overflow
```

```
Details of Stack :
Item popped = 9
Item popped = 8
Item popped = 7
Item popped = 6
Item popped = 5
Item popped = 4
Item popped = 3
Item popped = 2
Item popped = 1
Item popped = 0
stack underflow
Item popped = 0
stack underflow
Item popped = 0
```

The problem with the above implementation lies in the size of the declared array: 12 items cannot be pushed into an array with a size of 10. The extra two items resulted in a "stack overflow" during the `push` operation and "stack underflow" during the `pop` operation.

In the second implementation, the internal structure keeping the items is changed from an array to a linked-list. Listing 8-7 is the code for the second implementation. A separate `StackItem` class is needed in this solution to create space for storing integers pushed into the stack. As before, the solution works well with an input value 5, but unlike the previous implementation, no "stack overflow" or "stack underflow" messages were reported with an input value of 12:

```
Pushed 0 into Stack
Pushed 1 into Stack
Pushed 2 into Stack
Pushed 3 into Stack
Pushed 4 into Stack
Pushed 5 into Stack
Pushed 6 into Stack
Pushed 7 into Stack
Pushed 8 into Stack
Pushed 9 into Stack
Pushed 10 into
Stack
Pushed 11 into
Stack
```

```
Details of Stack :
Item popped = 11
Item popped = 10
Item popped = 9
Item popped = 8
Item popped = 7
Item popped = 6
Item popped = 5
Item popped = 4
Item popped = 3
Item popped = 2
Item popped = 1
Item popped = 0
```

Note that when the internal implementation of the stack was changed from an array to a linked-list representation, no change was required in `main()`, the client or user of the stack. Any change to the stack definition had been carried out on the implementation part, without a change to the contract part, of the stack. We say that the design of the Stack class exhibits *information hiding* by hiding the internal representation of the stack from its client, `main()`.

8.5.3 Trade-Off

Encapsulation enhances software maintainability by limiting the ripple effects, resulting from a change in object definition, from affecting other objects. This is done by:

- increasing the cohesiveness of data and methods through bundling; and
- reducing the strength of coupling between software components by hiding implementation details of objects from their users.

Enhanced software maintainability comes with a price. The trade-off to software maintainability using encapsulation is performance since access to data is carried out indirectly via the methods, and their execution would involve the execution of additional statements resulting in reduced performance efficiency.

```
class Stack {
  private StackItem top, temp;
  private int size;

  public int pop() {
    int x = 0;
    if (empty()) System.err.println("stack underflow");
    else {x = top.getItem(); top=top.getPrevious();size=size-1;}
    return(x);
  }
  public void push(int x)  {
    if (full()) System.err.println("stack overflow");
    else {temp=top; top=new StackItem();
          top.setPrevious(temp); top.setItem(x);
          size=size+1;
          System.out.println("Pushed "+x+" into Stack");}
  }
  public int size() { return(size); }
  public boolean empty() { return(size() == 0); }
  public boolean full() { return(false); }

  Stack() {
    top = null;
    size = 0;
  }

  public static void main(String argv[]) {
    int i, numberOfItem;
    numberOfItem=Integer.parseInt(argv[0]) ;
    Stack s = new Stack();
    for (i = 0; i<numberOfItem; i++)
      s.push(i);
    System.out.println("\nDetails of Stack : ");
    for (i = numberOfItem; i>0; i--)
      System.out.println("Item popped = "+s.pop());
  }
}

class StackItem {
  private int item=0;
```

```
  private StackItem previous;
  public int getItem() {return item;}
  public void setItem(int x) {item=x;}
  public StackItem getPrevious() {return previous;}
  public void setPrevious(StackItem p) {previous=p;}
  StackItem() {previous=null;}
}
```

Listing 8-7: Stack using a linked-list implementation.

8.6 Summary

The issues of modularity were discussed in this chapter. In particular, we noted that:

- A method is the smallest program unit to be considered as a whole. A class is the next bigger unit.
- Attribute and method definitions are distinguished into object attributes and methods; and class attributes and methods.
- Class attributes and methods are denoted in Java using the `static` keyword.
- The visibility of attributes and methods to code fragments external to a class can be controlled using access control specifiers—`private`, `public`, `friendly` and `protected`.
- The `private` specifier makes entities that follow it hidden from code fragments external to the class. The `public` specifier makes entities that follow it visible from all other Java code. The `protected` specifier makes entities that follow it accessible to code fragments of its subclass.
- For entities with no access specifier, the default specifier `friendly` is assumed. The `friendly` specifier makes entities that follow it accessible to code fragments within the same package.
- Appropriate classes with logically related functionality can be grouped together using the package facility. Package hierarchy is specified via the `package` keyword preceding a class definition.
- The `import` keyword provides the complement function of the package facility. The `import` keyword makes package constituents visible in program code.
- Encapsulation is the bringing together of a set of attributes and methods into an object definition and hiding their implementational structure from the object's users.
- Encapsulation is supported by two subordinate concepts: *Bundling* and *Information hiding*. Bundling is the act of associating a set of methods with a set of data as the only means of affecting the values of the data. Information hiding refers to the hiding of internal representation of data and methods from the users of these data and methods.

- Encapsulation enhances software maintainability by limiting the ripple effects, resulting from a change in object definition, from affecting other objects.

8.7 Exercises

1. What is encapsulation? How does encapsulation contribute to software maintainability?
2. How does the code in Listing 8-8 measure up to the principle of encapsulation? Comment.
3. How would you enhance the code in Listing 8-8 to achieve the desired effect of encapsulation? What is the trade-off of your enhancement? What are its advantages?

```
class time {
  int hour;
  int minute;

  time() {};

  public static void main (String arg[]) {
    time t = new time();
    t.hour = 3;
    t.minute = 25;
    System.out.println("The time now is "+t.hour+":"+t.minute);
  }
}
```

Listing 8-8: time.java.

9

Exception Handling

We have so far discussed the concept of object-oriented programming involving class definitions, object instantiation, the use of instance variables and methods, and the practice of code reusability via inheritance from superclasses.

This practice has given rise to payoffs in terms of the software engineering ideas of *abstraction* and *modularity*. The former allows a programmer to focus his thoughts on issues that are crucial and relevant, and delay decisions on other less pressing concerns. The latter characteristic ensures a degree of decoupling amongst software components, which leads to better maintainability.

This chapter examines the *exception handling* mechanism in Java. The fact that software modules should be robust enough to work under every situation, yet be flexible enough to work under any condition and even those not yet conceived, is indeed a tall order. The exception handling mechanism is key to achieving this goal.

9.1 Using Exceptions

The ideas of modularity and packaging promote software engineering, but there is a subtle conflict of requirements. While the advantages of modularity stem from reusability of well-tested and proven code, this is only achieved if class definitions are never modified once they are committed into the code repository.

While it may be true that the bulk of code may not typically require modifications, a single (or sometimes simple) modification is all it takes to introduce unintended side-effects. Yet, code modules typically require minor modifications before they are used in a different scenario or project.

In general, generic portions of code, say searching or sorting an array, often require no modifications even across different applications. However, it is in the contingency plans—say an empty array, a missing target during searching, or popping an empty stack—that requirements change and different measures are necessary.

A traditional solution to using generic status code in the face of different contingencies, as in this situation, relies on status codes via parameter passing. This approach is unfortunately clumsy, and in some situations there is a need for constant polling.

The exception mechanism in Java allows for contingency situations to be anticipated or identified within the class construct, but its handling of that condition is implemented elsewhere. It solves the dilemma we just discussed so that only generic code and the detection of contingencies are within the class construct, but handling of these contingencies is located at application specific modules.

Many API libraries such as input/output and networking rely on exception handling for flexible error handling. This mechanism is thus a key feature in reusable software development using Java.

9.2 Exception Terminology

Using the exception handling mechanism in Java involves:

- identifying exception *conditions* relevant to the application;
- locating exception handlers to *respond* to potential conditions; and
- monitoring when such conditions *occur*.

As with all representations in Java, exception conditions are denoted by objects. Similar with all objects, exceptions are also defined by class constructs, but inheriting attributes from the Exception superclass. While exception objects may be identified by object tags, additional attributes may be included for custom manipulation.

Exception handling is dynamically enabled for statement blocks within a `try`-block. Within it, normal facilities and rules for blocks apply but control-flow within may be transferred to associated exception handlers. An appropriate statement block prefixed by a `catch`-clause is then executed when the associated exception condition occurs.

The occurrence of an exception condition is indicated by a `throw`-statement. It allows for an exception object to be dynamically propagated to the most recent exception handler. Flow-control does not return following a `throw`-statement. Instead, execution control proceeds at the statement following the `try`-block that handles the exception.

9.3 Constructs and Exception Semantics in Java

We now consider the language primitives for realizing the exception handling framework in Java. As seen,

- exception objects are defined via class constructs that inherit from the Exception class;

- exception handling is enabled within a `try`-block, with handlers indicated by `catch` clauses; and
- an exception condition is identified by a `throw` statement. (Some predefined exception conditions are thrown implicitly by the Java Virtual Machine.)

9.3.1 Defining Exception Objects

The smallest exception object in Java merely extends from the Exception superclass, as outlined in the class definition for TransmissionError below:

```
class TransmissionError extends Exception {
}
```

Logically, its objects have the same structure as objects of the Exception parent class which implements the basic functionality of exception objects. However, subclass objects are appropriately tagged (as part of Java semantics), so that objects may be subsequently distinguished. It is often more productive to define a richer structure so that such exception objects may be accurately identified as well as easily manipulated.

```
class TransmissionError extends Exception {
   int errorKind;
   TransmissionError() { errorKind = 0; }
   TransmissionError(int x) { errorKind = x; }
   String toString() { return("Transmission Error: " +
                              errorKind); }
}
```

Encapsulating exception conditions in objects allow for rich representations and functionality (via instance variables and methods, respectively). An appropriate design for such objects would reduce any overheads of coupling between conditions and handlers.

9.3.2 Defining Exception Handlers

Exception handlers are introduced by the `catch`-clause within a `try`-block prefix, of which the following code fragment is representative.

```
class X {
  ...
  T m() {
    ...
    try {
      Y b = new Y();
      b.performOperation();
      ...
    } catch (TransmissionError t) {
      errorRecovery();
      ...
```

```
        } catch (IOException e) {
          errorReport();
          ...
        }
        n();
      }
    }
```

Code within the `try`-block, as well as code dynamically invoked from there, are regions where exception handling is enabled. In the representative class definition for `X` above, this region includes the statements within the `try`-block and other blocks within methods invoked from there such as `performOperation()`.

Exception objects thrown from a `try`-block may be potentially caught by a `catch`-block exception handler as long as the type of the former matches that expected for the latter as indicated by its formal parameter. In our previous example, the first handler catches `TransmissionError` exception objects, while the second handler catches `IOException` exception objects. Since objects of a subclass share the characteristics and are also considered objects of the base class, the said handlers will also cater to subclasses of `TransmissionError` and `IOException` objects, respectively.

The placement order of `catch`-blocks is significant. Due to the inheritance mechanism, `catch`-blocks work in a sieve-like manner, and handlers for subclasses should appear before handlers for superclasses. When exception objects are caught, control-flow is transferred to the exception handler concerned. Control-flow then resumes at the statement following the `try`-block; in our example, this is method `n()`.

9.3.3 Raising Exceptions

An exception condition is ultimately represented by an exception object derived from the predefined Exception class.[1] A condition is made known by throwing an appropriate object via a `throw` statement, to be subsequently caught by an associated handler.

[1] Actually, objects that can be thrown by the `throw` statement must be derived from the Throwable superclass. However, in addition to the Exception class, Throwable also includes the Error class which indicates serious problems that a reasonable application should not try to catch. As such, we will continue to work from the Exception class.

```
class Y {
  ...
  void performOperation() {
    ...
    if (F)
      throw new TransmissionError();
  }
}
```

In the event that the thrown exception object does not match what is expected by event handlers, it is further propagated to the caller of the method which contains the `try`-block. This caller chain (which forms a back-trace) proceeds until the `static void main()` method is encountered. The predefined environment supplies a default exception handler that aborts program execution with a execution back-trace from the run-time stack, and an appropriate error message.

`TransmissionError` is the typical case of a programmer-defined exception condition. The Java API also contains various exception classes which are used within the API, for example, `MalformedURL` and thrown from methods such as `performOperation()`. These are accessible via the normal mechanisms. There is also a unique set of exceptions that is thrown directly from the Java Virtual Machine. For example, integer division is translated into an operator for the Java Virtual Machine. Thus, if a 0 divisor is encountered in an expression, a `DivideByZero` exception is raised from the Java Virtual Machine without any corresponding `throw`-statement from the application code.

9.4 A Simple Example

We will now piece together the various Java constructs described in the previous section to provide the context for their usage. As a working example, we consider a stack object that allows for items to be placed, but removed in the reverse of placement order. Like a stack of plates, it allows for pushing and popping items from the "top."

Two situations may be anticipated during stack usage: when the stack is empty and no items are available for retrieval, and when the stack is full and cannot accommodate further items. The exception mechanism is ideal in that exceptions may be raised independently of how clients using a stack object may want to respond to such contingencies. This framework physically separates server code from client code, but yet provides for conceptual association so that contingencies in the server may be easily propagated and handled by the client.

To begin, the two stack conditions may be defined as follows:

```
class EmptyStack extends Exception {
}

class FullStack extends Exception {
}
```

A stack object may be implemented using an array to hold items pushed to it. By default, stacks will hold a maximum of 10 items, unless otherwise specified via its constructor.

```
class Stack {

  int height;
  Object items[];
  void push(Object x) throws FullStack {
    if (items.length == height)
      throw new FullStack();
    items[height++] = x;
  }
  Object pop() throws EmptyStack {
    if (height == 0)
      throw new EmptyStack();
    return(items[--height]);
  }
  void init(int s) {
    height = 0;
    item = new Object[s];
  }
  Stack (int s) { init(s); }
  Stack() { init(10); }
}
```

Note the `throws` suffix in the method signature forewarns callers of the possibility of an exception. Consistent with secure programming practice, Java would insist that coding within the client either sets up an appropriate exception handler, or appends the `throws` suffix in the caller method so that the stack exception is propagated.

The programming style in class Stack allows it to be used in varied situations without the concern for acceptable or "correct" responses to stack errors. As usage of Stack is available to various client classes, the code fragments may implement appropriate handlers for each application.

We first consider a scenario involving a parser for arithmetic expressions. A full stack arising from a deeply nested expression might cause parsing to be aborted with an appropriate message.

```
class Parser {
  ...
  void Expression() {
    Stack s = new Stack();
    try {
      ...
      s.push(x);
      ...
    } catch (FullStack e) { // respond to full stack condition
      error("expression nesting exceeds implementation limit");
      abort();
    } catch ... // other possible exceptions
  }
  ...
}
```

In a situation where an error may not be fatal, a value from the stack can be substituted with another by the exception handler so that processing may continue. Such recovery processing is strategically focused.

```
class Evaluator {
  Stack s = new Stack();
  ...
  void operand() {
    Integer value;
    ...
    try {
      value = (Integer) s.pop();
    } catch (EmptyStack e) { // respond to empty stack condition
      value = new Integer(0);
    }
    ...
  }
}
```

9.5 Paradigms for Exception Handling

We have seen exception handling in Java as being comprised of exception definition via a class definition, exception handlers via `try`- and `catch`–blocks, and raising exception incidents by throwing appropriate objects.

A general framework for exception handling has been outlined in the previous section. Ideally, the framework should be extended to fit various scenarios, which leads us to present various usage patterns.

9.5.1 Multiple Handlers

To facilitate monitoring more than one exception condition, a `try`-block allows for multiple `catch`-clauses. Without the exception handling mechanisms of Java, error handling code would be untidy, and especially so for operations where a sequence of erroneous situations can occur. For example, when sending email to a user `happy@xyz.com`, an email client program must: initiate a socket connection to the host machine `xyz.com`; specify the recipient; and send the contents of the mail message.

Complications arise when `xyz.com` is not a valid email host, `happy` is not a legitimate user on the host, or premature closure of the socket connection.

```
class EMail {
  ...
  void send(String address) {
     errorCode = 0;
     makeHostConnection(emailHostOf(address));
     if (connectionError) {
       errorMessage("host does not exist");
       errorCode = 1;
     } else {
       verifyUser(emailUserof(address));
       if (noUserReply {
```

```
            errorMessage("user is not valid");
            errorCode = 2;
          } else {
            while ((!endofInputBuffer()) && errorCode != -1) {
              line = readInputBuffer();
              sendContent(line);
            }
            if (networkError) {
              errorMessage("connection error occurred");
              errorCode = 3;
            }
          }
      }
      ...
    }
```

The above skeletal code for email processing may be structurally improved and made more transparent by using exception handling mechanisms. It is also useful from the maintenance point of view to separate processing logic from error processing. The code fragment below which uses multiple exception handlers is tidier if the appropriate exception objects are thrown by the methods `makeHostConnection()`, `verifyUser()` and `sendContent()`.

```
class EMail {
  ...
  void send(String address) {
    try {
      errorCode = 0;
      makeHostConnection(emailHostOf(address));
      verifyUser(emailUserof(address));
      while (!endofInputBuffer()) {
        line = readInputBuffer();
        sendContent(line);
      }
    } catch (SocketException s) {
      errorMessage("host does not exist");
      errorCode = 1;
    } catch (NoUserReply n) {
      errorMessage("user is not valid");
      errorCode = 2;
    } catch (WriteError) {
      errorMessage("connection error occurred");
      errorCode = 3;
    }
  }
  ...
}
```

The resultant structure is clearer—normal processing logic in the `try`-block, and error handling in `catch`-clauses.

9.5.2 Regular Exception Handling

Where there are multiple code fragments with similar error handling logic, a global exception handler would again be neater.

```
class EMail {
  ...
  void makeHostConnection(String host) {
    openSocket(host);
    if (!IOerror()) {
       checkResponse();
       giveGreetings();
    }
  }
  void giveGreetings() {
     writeMessage("HELO " + hostname);
     if (IOerror())
        errorCode = 9;
     else
        checkResponse();
  }
  void verifyUser(String user) {
     writeMessage("VERIFY " + user);
     if (IOerror())
        errorCode = 9;
     else
        checkResponse();
  }
  ...
}
```

In the code above, a transaction from an email client involves writing a message to the server and then reading if it receives an appropriate response. However, each message to the server might be unsuccessful due to a network error such as the termination of the connection.

With the exception handling mechanism in Java, generic errors may be handled by a common network error handler,for example, an `IOException` exception handler, so that such errors need not be constantly monitored.

```
class EMail {
  ...
  void send(String address) {
     try {
       errorCode = 0;
       makeHostConnection(emailHostOf(address));
       verifyUser(emailUserof(address));
       ...
     } catch (IOException x) {
       // network error detected
     }
  }
  void makeHostConnection(String host) {
    openSocket(host);
    checkResponse();
```

```
    giveGreetings();
  }
  void giveGreetings() {
    writeMessage("HELO " + hostname);
    checkResponse();
  }
  void verifyUser(String user) {
    writeMessage("VERIFY " + user);
    checkResponse();
  }
}
```

9.5.3 Accessing Exception Objects

So far, we have discussed how a `catch`-block responds to exceptions specified in its parameter type `T`, but without reference to the parameter name `e`.

```
try {
  ...
  throw new X();
} catch (X e) {
   ... // e refers to exception object thrown earlier
}
```

The fact that the parameter name of a `catch`-block is bound to the current exception object thrown allows for the means of transferring information to the exception handler.

9.5.4 Subconditions

We have seen that exception conditions are represented by objects that are described via class constructs. Since objects are dynamically distinguished from one another by their built-in class tags, this is a viable and productive method for representing different conditions.

As with other classes, a new exception condition may also be subclassed from an existing class to indicate a more specific condition. Incorporating the inheritance mechanism to exception handling allows for logical classification and code reusability in both condition detection and handler implementation. `CommError` and `ProtocolError` in the skeletal fragment below are typical examples of rich representations using inheritance.

```
class CommError extends Exception {
   int errorKind;
   Date when;
   CommError(int a) ...
}
class ProtocolError extends CommError {
   int errorSource;
   ProtocolError(int a, int b) ...
}
```

The language mechanism that allows exception conditions and subsequently flow of control to propagate to an appropriate handler provides for powerful and flexible processing.

```
try {
  ...
    throw new CommError(errorCode);
  ...
    throw new ProtocolError(errorCode, extraInformation);
  ...
} catch (ProtocolError e) {
  ... // handle ProtocolError by inspecting e appropriately
} catch (CommError f) {
  ... // handle CommError by inspecting f appropriately
}
```

Due to inheritance rules, the exception handlers of the above `try`-block are ordered so that specific (subclass) exceptions are caught first. If generic (superclass) exceptions were caught first, the handler for the specific exceptions would never be used.

Note that while the exception object thrown is caught within the same `try`-block in the above, in practice, a `throw`-statement may also be deeply nested within methods invoked from the `try`-block.

9.5.5 Nested Exception Handlers

Since an exception handler comprises mainly of a statement block, the sequence of statements within it may also contain other `try`-blocks with associated nested `catch`-blocks.

```
try {
  ...
    throw new X(errorCode);
  ...
} catch (X f) {
  ...
  try {
    ...
      throw new Y(errorCode, m);
    ...
  } catch (Y e) {
    ...
  }
}
```

As illustrated above, the scenario occurs when exceptions are anticipated within exception handlers.

9.5.6 Layered Condition Handling

Just as `catch`-blocks may be nested, we consider a related situation where a more specific exception handling is required. This can occur when the current handlers are not sufficient, and the new `try`-block is nested within another to override it.

```
try {
  ...
    throw new X(errorCode);
    try {
      ...
      throw new X(errorCode);
      ...
    } catch (X e) {
    ...
  }
} catch (X f) {
  ...
}
```

There are two applicable exception handling paradigms here: the nested handler may perform all necessary processing so that the enclosing handler does not realize that an exception has occurred; or the nested handler may perform processing relevant to its conceptual level and leave the remaining processing to the outer handler.

The former has been illustrated in the previous fragment, while the latter has been outlined in the framework below, where the nested handler throws the same exception after sufficient local processing.

```
try {
  ...
  try {
    ...
      throw new X(errorCode);
    ...
  } catch (X e) {
    ...
    throw e;
  }
  ...
} catch (X f) {
  ...
}
```

9.6 Code Finalization and Cleaning Up

The model for control-flow mechanisms involving statement sequences, conditional branching and iteration are unchanged for the Java programming language. The exception handling facilities in Java may be considered an advanced control-flow facility, which allows control to be dynamically transferred out of well-tested code modules. We continue to examine two more features relating to control-flow.

9.6.1 Object Finalization

The role of constructor methods was discussed in Chapter 3 when providing the necessary initialization for all newly created objects. This language feature imposes an invariant for all objects of the class. The default constructor with no parameters is the simplest and typically provides baseline initialization. Other constructors provide the means of initialization by various input parameters.

The complement of the constructor mechanism is a destructor facility. Its main purpose is to undo at the onset of object disposal, what was performed during initialization. If storage had been allocated during object initialization, the appropriate destructor behavior should dispose of such storage for subsequent use.

Such language mechanisms are typically in place for modular languages, and more so, object-oriented languages. However, destructor methods in Java are less necessary due to the automatic garbage collection scheme at run-time. Any storage areas that may have been allocated, but are nonaccessible, are reclaimed for subsequent reuse.

This technique is indeed useful when storage recycling is not always clear to the programmer, but instead is assured that unusable memory fragments will be ultimately recovered by the language run-time system. Thus, the Java programmer may allocate storage at will, and is not obliged to keep track of usage, nor required to de-allocate them.

De-allocating memory (which is no longer needed) is only one aspect of housekeeping for objects. Managing memory and variables happens to be an issue that is internal to a program, but can be handled automatically by the Java Virtual Machine.

There are other aspects of housekeeping that are external to a program, for example, releasing unused file descriptors or relinquishing a network socket connection. As such resources are also external to the Java Virtual Machine, automatic de-allocation is not practical.

In place of destructor methods, Java allows for finalization methods. Each class definition may include a parameterless method called `finalize()`. The run-time system ensures that this method will be invoked before the object is reclaimed by the garbage collector.

```
class  Email {
   Email() {
      // open network connection
   }
   ... other methods
   void finalize() {
      // close network connection
   }
}
```

9.6.2 Block Finalization

As described in the previous section, a `finalize()` method performs the last wishes for data before it is destroyed. Java also provides a similar mechanism for code blocks. A `finally` clause may optionally follow a `try` block. It guarantees that its

code will be executed regardless of whether an exception was thrown in the `try` block, and if thrown, whether it was caught by an associated handler.

```
try {
  // processing
} catch (TransmissionError t) {
  // handle TransmissionError exception
} finally {
  // perform clean up before leaving this block
}
```

This mechanism allows for a neat program structure, for example, when there are mandatory code fragments in both normal processing and exception handling. In elaborating our transmission example below, normal processing may involve opening the transmission channel, performing all the required transmission, and then terminating transmission by closing the channel. In the unfortunate event of a transmission error, we initiate contingency processing before closing the transmission channel. Maintenance is error prone due to the repetition of `channel.close()` in both the `try`- and `catch`-blocks.

```
try {
  channel = openTransmissionChannel();
  channel.transmit();
  channel.close();
} catch (TransmissionError t) {
  hasError = true;
  channel.close();
} catch (NoReplyError x) {
  toRepeat = true;
  channel.close();
}
```

The improved style using a `finally`-block avoids repetition of `channel.close()`.

```
try {
  channel = openTransmissionChannel();
  channel.transmit();
} catch (TransmissionError x) {
  hasError = true;
} catch (NoReplyError x) {
  toRepeat = true;
} finally {
  channel.close();
}
```

9.7 Summary

In this chapter, we discussed advanced control-flow facilities. The most significant of these is exception handling since it allows for flexible integration of modular code. The language primitives allow for:

- the definition of exception conditions;
- raising of exceptions within the virtual machine, as well as via the `throw` statement; and
- catching of exception objects via `try`- and `catch`-blocks.

Other control-flow facilities for neater program structures include finalization for objects and code blocks.

- A `finalize()` method is invoked before an object is destroyed so that it can complement the actions of a constructor.
- A `finally` block allows for fail-safe code execution before leaving a `try`-block, and is independent of exceptions or exception handlers.

9.8 Exercises

1. Consider the `CalculateEngine` and `CalculatorFrame` classes to implement the calculator in Chapter 4. Suggest how the framework can take advantage of the exception handling facility in Java so as to maintain modular boundaries and provide better error messages.
2. Implement a `Symbol` table class so that each symbol is associated with a numeric value. The two main methods are `set()` and `get()`:

```
void set(String sym, int value);
int get(String sym);
```

   The `set()` method associates the `int value` with the symbol `sym`, while the `get()` method performs the complement of retrieving the `int value` previously associated with `sym`.

   Define appropriate exceptions so that clients of the Symbol table may respond to conditions appropriately. For example, when they retrieve the value of a nonexistent symbol, clients can either stop execution and display an error message or use a default value of 0.
3. Incorporate the Symbol table object into the calculator so that intermediate results may be associated with user-defined symbols.

10

Input and Output Operations

We have covered all the basic mechanisms of Java, but not much has been said about input and output operations such as reading from and writing to files. In fact, the Java programming language excludes any description of performing such operations. Instead, this critical functionality is implemented by standard libraries.

In Java, many practical features are not built into the language proper. This functionality is included in libraries known as the Java Application Programming Interface (API)—mostly standard across Java platforms. In this chapter, we will briefly view the Java API and its relevance to reading from and writing to files and other generic devices.

10.1 An Introduction to the Java API

Java is an object-oriented programming language. This facilitates the creation of objects and message passing amongst such objects. We have seen too that objects are created from class definitions, and as such, all code written in Java exists within class definitions.

Since the Java API library is merely reusable code, they exist in class definitions too. The Java API is thus a large set of classes to make the task of programming development more productive. Often, our programs need not implement the data-structures it needs. Instead, code within the API may be used, or reused by specialization through inheritance to meet the needs of our custom applications.

The API is used in various generic ways:

- The simplest means of using the API is to create an instance of an API class; for example, to read from a file, we create an instance of `FileInputStream`. The `read()` method is used on the resultant object to read file contents,

and similarly `close()` will perform necessary clean-up to system resources like file descriptors for reuse.

- A new class may be defined based on an API class by inheritance. This facilitates reuse of generic code; for example, to create new threads for specific multithreaded applications, we define a new class based on the `Thread` class, but with new definitions relevant to our application.
- Often, class variables of API classes may be used directly without explicit initialization, that is, out is a public class variable in the `System` class and may be used directly for the purpose of printing to the standard output stream. This was how `System.out.println()` was used in Chapter 4.

As there are a large number of classes in the Java API, it is fruitful to organize and group them according to their functionality. In JDK 1.5, the basic Java API is organized into six packages among others: `java.lang`, `java.io`, `java.util`, `java.net`, `java.awt` and `java.applet`.

- The `java.lang` package consists of Java classes that are essential to the execution of Java programs; for example, the `Thread` and `System` classes belong to the `java.lang` package.
- The `java.io` package consists of Java classes that are used for input and output facilities; for example, the `FileInputStream` class mentioned earlier belong to the `java.io` package.
- The `java.net` package consists of Java classes that are relevant to networking; for example, the `Socket` class belongs to the java.net package and is used for network connections to hosts on other machines.
- The `java.util` package consists of Java classes for generic functionality such as list collections, date representation; for example, the `Vector` and `Date` classes belong to the `java.util` package.
- The `java.awt` package consists of Java classes that implement the Abstract Windowing Toolkit. These classes are used for creating graphical interfaces for a Windows-based environment.
- The `java.applet` package consists of Java classes that are used to support applet execution within the context of a Web browser.

10.2 Reading the Java API Documentation

The Java API is described in the Java API documentation, available in printed books, PostScript, or HTML. The HTML form is typically preferred in an online environment since it allows for convenient navigation.

From a standard distribution site, the zip or tar file should be downloaded, and the contents then extracted. Viewing local pages using a Web browser is straightforward. By default, the document is displayed using HTML frames (see Figure 10-1). There are two navigation frames on the left-hand side. In the top left frame you can either select “All Classes” to display all classes in the bottom left frame, or you can

select a specific package to display classes in the specific package in the bottom left frame. The class names in the bottom left frame are displayed in alphabetical order. In the bottom left frame you can then select a specific class to display the details (methods, fields, etc.) of a specific class in the main frame. The following screen snapshot in Figure 10-2 shows the details of the class `BufferedReader` in `java.io` package.

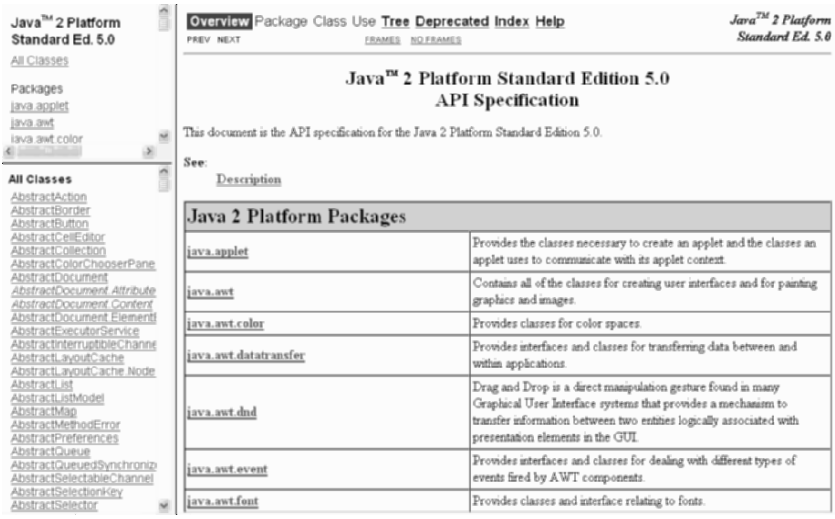


Figure 10-1: Viewing API overview.



Figure 10-2: Viewing class API documentation.

After selecting the “Index” on the navigation bar at the top of the main frame, the names of all variables and methods that have been indexed can be viewed, as seen in Figure 10-3. Links for these entities ultimately refer back to their class definitions.

For each class, the API documentation includes also a brief overview that is hyperlinked to more detailed descriptions.



Figure 10-3: Viewing API documentation by sorted names.

10.3 Basic Input and Output

The simplest start to input and output are pre-initialized objects `System.in`, `System.out`, and `System.err`. These correspond to the three standard file descriptors `stdin`, `stdout`, and `stderr`, respectively, in UNIX, being bound to the standard input, standard output and standard error streams. Typically for interactive processes, these streams correspond to the keyboard and screen. (While standard output and standard error are bound to the same device, they are logically different and very useful with stream redirection.)

The `System` class belongs to the `java.lang` package, but due to heavy use of this package, it is implicitly imported in every Java program unit. The commands `in`, `out` and `err` are class variables of the `System` class, and they have been pre-initialized with streams for input and output, respectively. All this may sound exceedingly strange, but it is fruitful to get the concepts clear at this stage as we begin to probe and use the Java API.

Field Summary	
`static PrintStream err`	The "standard" error output stream.
`static InputStream in`	The "standard" input stream.
`static PrintStream out`	The "standard" output stream.

If we look up the `System` class documentation, `out` is documented as:

```
public static final PrintStream out;
```

Looking up the `PrintStream` class documentation will reveal (not exclusive) the following methods for `PrintStream` instances:

```
print(boolean)
print(char)
print(char[])
print(double)
print(float)
print(int)
print(long)
print(Object)
print(String)
```

This shows that a `PrintStream` instance (e.g., `System.out`) can print out `boolean`, `char`, `char array`, `double`, `float`, `int`, `long`, `Object` and `String` values. In addition, the `println()` method may also accept similar parameters as `print()` and with similar behavior, except that a carriage return is also printed after the value.

Thus, the following statements are legitimate:

```
System.out.print('g');
System.out.print(3.142);
System.out.print(45);
System.out.print("hello there");
System.out.println(23);
System.out.println(3.32123);
System.out.println(234567654);
System.out.println();
System.out.println(System.out);
```

The command `println()` without any parameters will just print a carriage return. Note that `Object` instances are also legitimate parameters to `print()` and `println()`. Since all classes are (directly or indirectly) derived from `Object`, `println()` will print out any object. In practice, the actual value displayed depends on whether appropriate code is present to provide a suitable textual representation of the object concerned.[1]

Since `System.err` is also a static variable initialized as a `PrintStream` instance, it has the same behavior as `System.out`. It even prints to the same device, that is, the screen, but may be redirected to another via operating system facilities.

The `System.in` stream is quite different because its use is for input. Again, the documentation for the `System` class will reveal that it is a static variable that is initialized to an `InputStream` object.

```
public static final InputStream in;
```

Upon further checking of the documentation for the `InputStream` class, the principal statement of interest is revealed to be reading a byte from the stream.

```
System.in.read()
```

As this method returns the internal representation of the character itself, we typically typecast it to a `char` type by using the `( )` typecast operator.

```
char c = (char) System.in.read();
```

Two significant points are noted from the detailed documentation: first, `read()` potentially throws an `IOException` object to indicate an error in the input operation. As such, it is expected that clients using this method must catch the exception within a `try`-block. Next, the method returns an integer value of `-1` after it encounters the last character to be read. Clients must also anticipate against reading past this point.

Putting it all together, we can write a program that reads its input and copies the contents to the output. We put all this code into the `static void main()` function.

[1] The `print()` method relies on the `toString()` method to provide a textual representation of an object. Since `toString()` is defined in the Object class and cannot anticipate properties of future class definitions, it only performs generic text conversion. Of course subclasses are free to override `toString()` with a more appropriate definition to provide more comprehensive details of the object.

```
import java.io.*;
class CopyInputToOutput {
  public static void main(String args[]) {
    int x=0, c;
    try {
      while ((c = System.in.read()) != -1) {
        System.out.print((char) c);
        x++;
      }
      System.err.print(x);
      System.err.println(" bytes read");
    } catch (IOException e) {
      System.err.println("I/O Error Exception has occurred");
    }
  }
}
```

The code shows the following points:

- A `try`-block anticipates the `IOException` object from `read()`, as forewarned in the signature for `read()` or its equivalent API documentation.
- A `while`-statement is used to iteratively read all characters until the end-of-stream as indicated by the `-1` sentinel.
- The local variable `x` is initialized to `0`, and is incremented each time in the loop to count the number of bytes read (and written).
- Output is written into two logical streams. The output stream contains that which was read from the input stream, while the error stream is for diagnostic messages of byte count, or errors. (If any stream is redirected, the other proceeds with the original device binding.)

10.4 File Manipulation

Input and output using the predefined streams has given a preview of how other input and output operations will be performed—via methods such as `print()` and `read()`.

Input and output involving the standard streams is simple because they have been pre-initialized, and no finalization code is required. This consistently reflects the situation where input and output streams are available when typical programs begin execution and may read from the keyboard or write to the screen, and always remain available.

The situation is different for reading and writing to files that must be opened before use, and subsequently closed when operations are complete. This reflects the dynamic nature of file representation. There may be an instance of a particular file on disk, but it may be at different stages of being read by many other programs. This file representation is encapsulated within a stream object comprising of suitable data-structure to represent how much has been read or written.

10.4.1 File Input

The state of file input, as to how much of the file has been read, is represented by a `FileInputStream` object. After the file has been opened, reading proceeds similarly with the predefined standard input stream. Finally, the `FileInputStream` object is closed after use so as to reclaim system resources.

The `TestInput` code fragment in Listing 10-1 is adapted from the previous one for reading from the standard input stream. In reading from a file, modifications include the instantiation and closing of a `FileInputStream` object, and catching the `FileNotFoundException`.

The `FileInputStream`, `FileNotFoundException` and `IOException` classes belong to the `java.io` package. Due to separate name spaces, we bring the class into scope via the `import` statement. Instead of just importing specific classes, we can import the whole package instead.

The documentation of `FileInputStream` will reveal two important characteristics of `TestInput`:

- The `FileInputStream` class is derived from the `InputStream` class. In other words, instances of `FileInputStream` are also instances of `InputStream`, that is, it includes the behavior of the `InputStream` class. This explains why the two code fragments look similar in using the `read()` method.
- The documentation for `FileInputStream` describes the constructor as potentially throwing a `FileNotFoundException` object. This explains the extra exception handler.

```
import java.io.*;
class TestInput {
  public static void main(String args[]) {
    int x=0, c;
    FileInputStream f;
    try {
      f = new FileInputStream("input.txt");
      while ((c = f.read()) != -1) {
        System.out.print((char) c);
        x++;
      }
      System.err.print(x);
      System.err.println(" bytes read");
      f.close();
    } catch (FileNotFoundException n) {
      System.err.println("File not found");
    } catch (IOException e) {
      System.err.println("I/O Error Exception has occurred");
    }
  }
}
```

Listing 10-1: TestInput class.

10.4.2 File Output

The mechanism for file output is intuitively similar to that for file input. Based on the code fragment in `TestInput`, we can make minor modifications to try out file output, as in

```
import java.io.*;
class TestCopy {
  public static void main(String args[]) {
    int x=0, c;
    FileInputStream f;
    FileOutputStream g;
    try {
      f = new FileInputStream("input.txt");
      g = new FileOutputStream("output.txt");
      while ((c = f.read()) != -1) {
        g.write(c);
        x++;
      }
      System.err.print(x);
      System.err.println(" bytes read");
      f.close();
      g.close();
    } catch (FileNotFoundException n) {
      System.err.println("File not found");
    } catch (IOException e) {
      System.err.println("I/O Error Exception has occurred");
    }
  }
}
```

Listing 10-2: TestCopy class.

The complement of `FileInputStream` is `FileOutputStream`. To write to a file, we merely create a `FileOutputStream` instance by giving an appropriate file name to the constructor. As with previous files, the `close()` method is used to signal the end of file manipulation.

The most significant change in this code fragment is that the `print()` method, used previously with `System.out`, is not used with `FileOutputStream`. The reason is clear on checking the documentation for `FileOutputStream`.

```
import java.io.*;
class AnotherCopy {
  public static void main(String args[]) {
    int x=0, c;
    try {
      FileInputStream f = new FileInputStream("input.txt");
      FileOutputStream g = new FileOutputStream("output.txt");
      PrintStream p = new PrintStream(g);
      while ((c = f.read()) != -1) {
        p.print((char) c);
        x++;
      }
      System.err.print(x);
      System.err.println(" bytes read");
```

```
        f.close();
        p.close();
      } catch (FileNotFoundException n) {
        System.err.println("File not found");
      } catch (IOException e) {
        System.err.println("I/O Error Exception has occurred");
      }
    }
  }
```

Listing 10-2: TestCopy class.

The documentation shows that the `write()` method is available, but `print()` method is not. Remember that `System.out` is an instance of `PrintStream`, but `FileOutputStream` is not related to `PrintStream`. This mystery is cleared in the next section—as to how the `print()` method may be used with `FileOutputStream` objects.

10.4.3 Printing Using PrintStream

The following two points explain how a `PrintStream` object may be obtained from an `FileOutputStream` instance, thereby allowing `println()` to be used in file output.

- The `FileOutputStream` class is derived from `OutputStream`.
- The documentation for the `PrintStream` class reveals that its instance is created from an `OutputStream` object.

It follows that a `FileOutputStream` instance may be used (as an `OutputStream` object) to instantiate a `PrintStream` class, as indicated in the code example in Listing 10-3.

```
import java.io.*;
class AnotherCopy {
  public static void main(String args[]) {
    int x=0, c;
    try {
      FileInputStream f = new FileInputStream("input.txt");
      FileOutputStream g = new FileOutputStream("output.txt");
      PrintStream p = new PrintStream(g);
      while ((c = f.read()) != -1) {
        p.print((char) c);
        x++;
      }
      System.err.print(x);
      System.err.println(" bytes read");
      f.close();
      p.close();
    } catch (FileNotFoundException n) {
      System.err.println("File not found");
    } catch (IOException e) {
      System.err.println("I/O Error Exception has occurred");
    }
  }
}
```

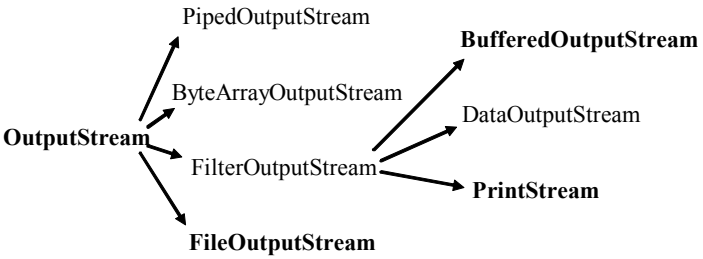
Listing 10-3: AnotherCopy class.

While the previous example shows how a `PrintStream` object may be obtained, its impact is minimal since the result is a mere change from using `write()` to using `print()`. A more significant advantage is that it allows the varied operations of `PrintStream` to be available.

10.5 Framework for Code Reuse

The use of `PrintStream` methods for `System.out` and an instance of `FileOutputStream` in the previous example shows the paradigm of code reusability in an object-oriented environment.

The following class hierarchy diagram clarifies the scenario that we just observed—that code in the `OutputStream` class is reused for `FileOutputStream` objects. A more subtle reuse paradigm is that a `PrintStream` object may be instantiated from an `OutputStream` object.



We now move to the bigger picture of the `java.io` package and code reusability. First, besides `FileOutputStream`, the other classes, for example, `PipedOutputStream` and `ByteArrayOutputStream` also benefit from deriving behavior from the `OutputStream` class. This is the typical scenario of reusing code via inheritance as discussed in Chapter 6.

It can be seen that `PipedOutputStream` and `ByteArrayOutputStream`, like `FileOutputStream`, are specializations of `OutputStream`. They all have the same model in that they allow `write()` operations, but whose contents are diverted to a file, which is a pipe or a byte array.

Note that the ability to create a `PrintStream` instance comes from `FilterOutputStream`. The latter provides the framework for adding functionality to an existing `OutputStream`. In the case of `PrintStream`, the additional functionality is higher-level output for values of different types, for example, `int`, `float`, `char`, `char array`, for the existing `OutputStream` instance used to create it.

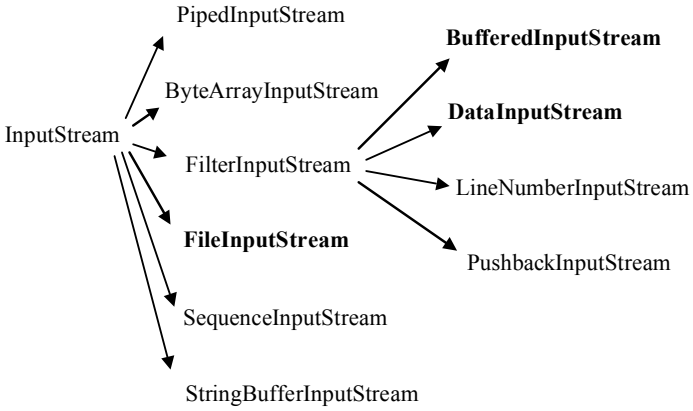
In the same way, buffered file output may be achieved by first creating a `FileOutputStream` and using it for creating a `BufferedOutputStream` object. Again, it (a superclass of `FilterOutputStream`) has added functionality to a `FileOutputStream` object (a superclass of `OutputStream`):

```
FileOutputStream f = new FileOutputStream("myOutput");
BufferedOutputStream buf = new BufferedOutputStream(f);
```

The combinations are, however, limitless since `BufferedOutputStream` is derived from `OutputStream`. As such, a buffered `PrintStream` can be obtained using the technique below.

```
FileOutputStream f = new FileOutputStream("myOutput");
BufferedOutputStream buf = new BufferedOutputStream(f);
PrintStream p = new PrintStream(buf);
...
p.println(...);
```

The same reuse framework is used for `InputStreams`, as shown in the class hierarchy diagram below:



The basic means of file input is via `FileInputStream`. Just as writing may be diverted, the input stream in this case comes from a file. Similarly, using `PipedInputStream`, `ByteArrayInputStream`, or `StringBufferInputStream`, input can be read from a pipe, byte array, or string.

Similar to the output case, buffered file input may be achieved by first creating a `FileInputStream` and using it for creating a `BufferedInputStream` object:

```
FileInputStream f = new FileInputStream("myInput");
BufferedInputStream buf = new BufferedInputStream(f);
```

Again, the combinations are limitless since the `BufferedInputStream` class is derived from `InputStream`.

10.6 `DataInputStream` and `DataOutputStream` Byte Stream Class

A `DataInputStream` is useful when binary data needs to be read. A buffered `DataInputStream` can be obtained using the same technique as obtaining the `BufferedInputStream`. Shown below in Listing 10-4, is reading a file and extracting the binary data such as toy name (`String`), toy price (`double`) and number of toys (`int`).

```
import java.io.*;
public class DataIn {
  static final String dataFile = "toydata";
  public static void main(String[] args) {
    String  inNames;
    double  inPrices;
    int     inUnits;
    try {
      DataInputStream in = new DataInputStream(
                           new BufferedInputStream(
                           new FileInputStream(dataFile)));
      while (true) {
        inNames = in.readUTF();
        inPrices  = in.readDouble();
        inUnits  = in.readInt();
        System.out.println (inNames  + " " + inPrices  + " " +
                            inUnits  + "\n");
      }
    } catch (EOFException e ) {
      System.out.println ("End");
    } catch (IOException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

Listing 10-4: DataIn class.

Similarly, the `DataOutputStream` class can be used to write binary data into files. The following methods among others are available in `DataOutputStream` among others:

```
void writeByte(int v)
void writeChar(int v)
void writeChars(String s)
void writeDouble(double v)
void writeFloat(float v)
void writeInt(int v)
void writeLong(long v)
void writeShort(int v)
void writeUTF(String str)
```

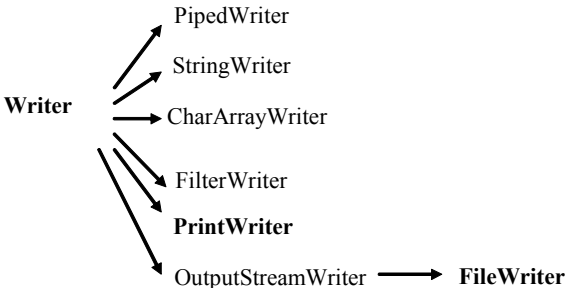
10.7 Character Stream Classes

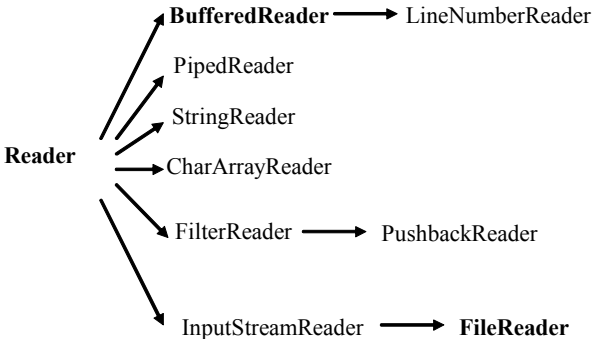
The standard input/output facilities available via the `InputStream` and `OutputStream` classes merely support 8-bit byte streams. The enhancement in JDK 1.1 (and beyond) relating to input and output provides support for *character* streams that allow for 16-bit Unicode characters. The advantage of character streams is that programs can now be independent of specific character encoding, which to some extent simplifies ongoing internationalization efforts.

The new character stream classes have been designed to parallel the byte stream equivalents in JDK 1.0. For example, the abstract input and output byte-stream classes `InputStream` and `OutputStream`, together with their subclasses, have new equivalent classes `Reader` and `Writer` with correspondingly similar functionality and usage paradigm.

JDK 1.0 byte-stream classes	JDK 1.1 (and beyond) character-stream classes
`InputStream`	`Reader`
`BufferedInputStream`	`BufferedReader`
`LineNumberInputStream`	`LineNumberReader`
`ByteArrayInputStream`	`CharArrayReader`
`FileInputStream`	`FileReader`
`FilterInputStream`	`FilterReader`
`PushbackInputStream`	`PushbackReader`
`PipedInputStream`	`PipedReader`
`StringBufferInputStream`	`StringReader`
`OutputStream`	`Writer`
`BufferedOutputStream`	`BufferedWriter`
`ByteArrayOutputStream`	`CharArrayWriter`
`FilterOutputStream`	`FilterWriter`
`PrintStream`	`PrintWriter`
`PipedOutputStream`	`PipedWriter`

The `InputStreamReader` and `OutputStreamWriter` classes form the bridge between byte and character streams through translation.





The copy program shown in Listing 10-3 can be translated using the Character Input-Output classes as shown in Listing 10-5.

```
import java.io.*;
class CharacterCopy {
  public static void main(String args[]) {
    int x=0, c;
    try {
      FileReader f = new FileReader("input.txt");
      FileWriter g = new FileWriter("output.txt");
      PrintWriter p = new PrintWriter(g);
      while ((c = f.read()) != -1) {
        p.print((char) c);
        x++;
      }
      System.err.print(x);
      System.err.println(" characters read");
      f.close();
      p.close();
    } catch (FileNotFoundException n) {
      System.err.println("File not found");
    } catch (IOException e) {
      System.err.println("I/O Error Exception has occurred");
    }
  }
}
```

Listing 10-5: CharacterCopy class.

Most often, programs would like to read character files in a line-by-line fashion. A line is defined as being terminated by a newline character (‘`\n`’). Using one of the subclasses of the `Reader`, `BufferedReader` Class, we are able to read character files line-by-line. Listing 10-6 shows how this can be done.

```
import java.io.*;
class LineReader {
  public static void main(String args[]) {
    int x=0;
    String line = null;
```

```
        FileReader f= null;
        FileWriter g = null;
        try {
          f = new FileReader("input.txt");
          BufferedReader b = new BufferedReader (f);

          g = new FileWriter("output.txt");
          PrintWriter p = new PrintWriter(g);

          while ((line = b.readLine()) != null) {
            p.println(line);
            x++;
          }
          System.err.print(x);
          System.err.println(" lines read");
          f.close();
          p.close();
        } catch (FileNotFoundException n) {
          System.err.println("File not found");
        } catch (IOException e) {
          System.err.println(
             "I/O Error Exception has occurred");
        } finally {
            try {
              if (f != null) f.close();
              if (g != null) g.close();
            } catch (IOException e) {
                System.err.println("I/O Error Exception has occurred");
            }
        }
      }
    }
```

Listing 10-6: LineReader class.

Buffered input and output streams are more efficient than the nonbuffered stream classes. These classes read/write a chunk of data from/to the OS and keep it in a buffer. Each read/write operation then reads/writes from/to this buffer, and the next OS call is made only when the buffer is empty/full.

10.8 Tokenizing the Input Using the Scanner Class

Input data is usually processed by breaking the input into tokens that are delimited by whitespaces (space, `\n`, `\t`) or by any other delimited (e.g., comma). Listing 10-7 shows the use of the `Scanner` class in the `java.util` package.

```
import java.io.*;
import java.util.Scanner;
public class TokenizeInput {
  public static void main(String[] args){
    Scanner s = null;
    try {
      s = new Scanner(new BufferedReader
                    (new FileReader("statement.txt")));
      while (s.hasNext()) {
```

```
        System.out.println(s.next());
      }
    } catch (IOException e) {
        System.err.println ("IO Exception has occured");
    } finally {
        if (s != null) {
          s.close();
        }
    }
  }
}
```

Listing 10-7: TokenizeInput class.

The `Scanner` class has methods to read the input and automatically convert the input to primitive types like `int`, `float`, `double`, and so on. The following statements read input, convert it to a double number, and add it to a total.

```
double s = 0;
while (s.hasNext()) {
  if (s.hasNextDouble()) {
    sum += s.nextDouble();
  } else {
      s.next();
  }
}
```

10.9 Formatting the Output Using the Format String

Output streams such as `PrintStream` and `PrintWriter` are capable of outputting formatted data. In order to do that, one needs to use the format method. The following snippet of code illustrates the formatting of data in integer and double precision formats:

```
int i = 10;
double d = 20.456107;
System.out.format ("Integer %d and Double %f", i,d);
```

The output looks like this:

```
Integer 10 and Double 20.456107
```

In general, the following characters, among others, if followed after a "`%`" in the format string, will produce the corresponding format of the output.

Formatting Character	Resultant format
"c", "C"	Unicode character
"d"	Decimal integer
"o"	Octal integer
"x", "X"	Hexadecimal integer
"e", "E"	Decimal number in scientific notation
"f"	Decimal number with decimal point

10.10 The File Class

It may be sufficient to represent file names as Strings and use the Input-Output classes to manipulate files for reading and writing data. But many times, there is a need to manipulate the properties of the file. Using File objects to open a file would also make programs truly portable across platforms. For example, a file name is represented differently in Windows as compared to Unix.

The following statements illustrate the kind of operations that you can perform using the `File` class:

```
File myFile = new File ("test.txt");
System.out.println ("Is Directory = " +
                    myFile.isDirectory());
System.out.println ("Is File = "+ myFile.isFile());
System.out.println ("Last Modified = " +
                                  myFile.lastModified());
System.out.println ("Absolute Path = " +
                    myFile.getAbsolutePath());
```

10.11 Random Access File Operations

Opening a file using the `RandomAccessFile` class would enable nonsequential access of a file. A `RandomAccessFile` can be opened in read, write, or both modes. This is specified in the constructor method. Common read/write methods found in `DataInput` and `DataOutput` classes can be used with random access files. The notion of a File Pointer that points to the current position is supported by these files. When the file is opened, the file pointer is positioned at the beginning of the file, indicated by location 0. As the file is read or written, the file pointer moves depending on the number of bytes read or written.

Files can be opened for random access by either specifying a file name or the `File` object in the constructor. The read/write mode is also specified in the constructor as follows:

```
RandomAccessFile fileR = new RandomAccessFile("data","r");
RandomAccessFile FileRW = new RandomAccessFile("test", "rw");
```

Some of the useful methods are illustrated below:

```
int     read(byte[] b, int off, int len)
byte    readByte()
char    readChar()
double  readDouble()
float   readFloat()
int     readInt()
String  readLine()
String  readUTF()
```

Methods commonly used to manipulate the File Pointer are shown below:

```
int  skipBytes(int n)
void seek(long n)
long getFilePointer()
```

10.12 Summary

This chapter has introduced the Java Application Programmer Interface (API). It provides the means by which programmers may code productively by reusing code. The Java API possesses the key success criteria for reusability in that it has a neat reuse framework and is adequately documented.

Code that judiciously uses the Java API is shorter and simpler to develop since it effectively builds on the work of others. There is a big user community and it is likely that any bugs discovered will be promptly fixed in future releases.

The `java.io` package is representative of the useful functionalities provided by the Java API. Our discussion has shown:

- the basic input/output functionality may be used via `System.out`, `System.err`, and `System.in`; and
- the generic input and output interfaces used in the abstract classes `OutputStream` and `InputStream`.

The next usage level of the `java.io` package involves:

- subclasses of `OutputStream` and `InputStream` that implement actual I/O operations on a suitable medium such as files or pipes via classes `FileInputStream` and `FileOutputStream`;
- incorporation of general I/O formats and options as implemented by `FilterInputStream` and `FilterOutputStream` classes and corresponding subclasses;
- the corresponding relationship with `Reader` and `Writer` classes in JDK 1.1 (and beyond) supporting internationalization for character representation; and
- random access files for nonsequential access of files:
- Data from files can be read using the `Scanner` object in a tokenized fashion and formatted data can be output into `PrintStream` and `PrintWriter` streams using the format method.

10.13 Exercises

1. Review the HTML-based API documentation to find more details on the following:

- operations allowable on Strings;
- constant value of PI;
- method to return the arc cosine of an angle.

2. Write a program to read file names specified in the command-line and copy their contents to the standard output.
3. Implement a method to read the contents of the file and write it out to the standard output stream, with all lowercase characters converted to uppercase.
4. Suggest how exception handlers may be installed so that input/output errors are reported and files appropriately closed after use.
5. Consider the similar code fragments for the solution to Questions 2 and 3, and suggest how object-oriented technology may be used to maximize reusability and maintainability

11

Networking and Multithreading

In Chapter 10, we previewed the Java API for input and output facilities, especially those associated with files requiring different formatting. The framework used to maximize code reusability was also discussed.

In this chapter, we will go beyond the local machine by looking at networking facilities in the Java API. Since the abstraction for networking primitives turns out to be byte streams, there is indeed much reuse of the classes seen in the previous chapter.

11.1 The Network Model

The networking facility available in Java using TCP/IP involves socket connections. This allows a host to link up with another so that a byte stream that is sent on one machine is received by the partner. A socket connection is symmetric and thus a host also receives what is sent by the partner.

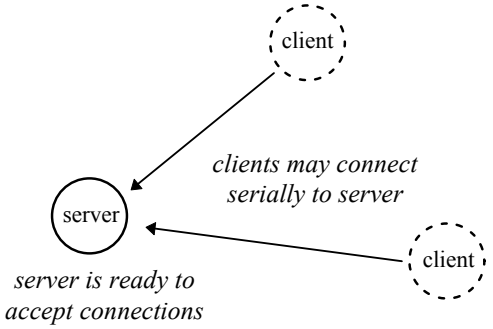


Figure 11-1: Server and clients.

In this model, as illustrated in Figure 11-1, machines may be asymmetrically classified as clients and servers. A client is one that initiates a network connection by naming the corresponding server. A server is one that is ready to receive a connection from a client.

A server host may provide more than one service, and thus a port number is required to distinguish between services. On the Internet, there are well-defined port numbers for the standard services. By UNIX convention, the first 1023 ports are reserved for system privileged services; thereafter other user programs may use the ports on a first-come-first-serve basis.

The following table gives some standard services and their corresponding port numbers:

Service	Port number
telnet	23
ftp	21
mail	25
finger	79
Web (httpd)	80

A socket connection is said to be established when a client successfully connects to a server machine. From this point, as illustrated in Figure 11-2, communication between both parties is symmetric. As such, writing at either end will cause the corresponding partner to receive the contents.

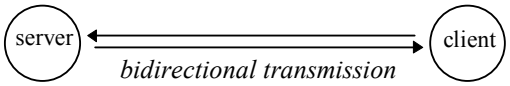


Figure 11-2: Bidirectional socket connection.

11.2 Sockets in Java

The implementation details of TCP/IP socket connections in Java are encapsulated in the `Socket` class which exists in the `java.net` package. As before, the `Socket` class must be brought into scope via an `import java.net.Socket` statement. A socket connection to a server involves instantiating a Socket object with the appropriate connection parameters.

In the typical scenario, we specify the host name of the server and the service port at which the server is listening to. A socket connection to the Web server (running at the default port 80) at `www.javasoft.com` would be made as such:

```
Socket soc = new Socket("www.javasoft.com", 80);
```

Note that the class documentation reveals that this constructor may throw two exceptions. An `UnknownHostException` is thrown if the hostname is not a valid domain name, while an `IOException` is thrown if a socket connection with a valid

host name cannot be established. This would be so if the server was not running during the connection attempt, or simply that the network was not operating.

If a socket object is successfully created, communication may commence by writing to and reading from the other party. These procedures are analogous to what was done in the previous chapter for file input and output. As expected, the streams model is used, as with `InputStream` and `OutputStream`.

At this point, we introduce the two methods that are relevant to socket objects. The `getInputStream()` and `getOutputStream()` methods return the `InputStream` and `OutputStream` objects associated with a socket. The former allows for reading from, while the latter allows for writing to the other party at the other end of the socket connection.

```
InputStream instream = soc.getInputStream();
OutputStream outstream = soc.getOutputStream();
```

Therefore, the necessary operations and paradigms in Chapter 10 on file input and output apply.

11.2.1 Example Client: Web Page Retriever

In this section, we consider how the `Socket` class may be used to retrieve content from Web servers. The generic framework for this client is also applicable to many other clients connected to a well-known host for services. (However, different applications may have their own application protocol to request the server for services.)

Section 11.2 gives the basics of how generic clients may connect with their corresponding servers. By following the methods laid out in Chapter 10 on file input and output, information exchange may proceed over the socket connection. The skeleton of the WebRetriever class is revealed in Listing 11-1:

```
import java.io.*;
import java.net.*;

class WebRetriever {

  Socket soc;
  OutputStream os; InputStream is;
  ....
  WebRetriever(String server, int port)
               throws IOException, UnknownHostException {
    soc = new Socket(server, port);
    os = soc.getOutputStream();
    is = soc.getInputStream();
  }
}
```

Listing 11-1: WebRetriever skeleton.

While this has laid the groundwork for clients and the server to "talk" over a network, nothing has been discussed about a common language as to what is exchanged. This is also known as the application protocol. In implementing a Web page retriever, we next consider how a Web client works in relation to a Web server.

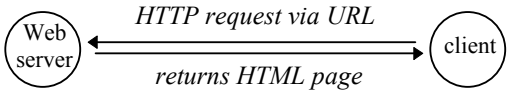


Figure 11-3: Web client/server communication

A Uniform Resource Locator (URL) is the abstraction of a resource available at a server. (While it is typically a Web server, it is not restricted to Web servers and may include FTP and news servers.) A highlighted anchor in a Web client browser has an underlying URL. Thus, when selecting a HTTP URL, such as `http://www.nus.edu.sg:80/NUSinfo/UG/ug.html`, the framework of a request issued by the client may be broken into four portions:

`http`	resource is to be retrieved by using the HTTP protocol
`www.nus.edu.sg`	hostname of the server
`80`	port number where service is offered
`/NUSinfo/UG/ug.html`	exact pathname on host root where resource is found

The following table enumerates the interaction between client and server, for the former to obtain a response from the latter.

State	Client Action
A Web server is generally happy to serve out pages to any client, and thus waits for a prospective client.	(Server is ready)
Clicking a link on a Web client browser will cause it to initiate a socket connection to the server, and then request the server for a particular page.	Create new socket
Client requests the server for a particular page.	Write HTTP GET request
On receiving a legitimate request in the form of a URL path, the server would return the contents of the corresponding file.	Read response from server

While there is an Internet RFC document that comprehensively describes the HyperText Transmission Protocol (HTTP), we only need to be aware of two details for the purpose of our example client:

- The `GET` *`path`* command requests that the server sends the resource at *path*.
- An empty text line indicates the end of client request to the server.

This basic HTTP request is handled by the `request()` method which packages the requested pathname into a `GET` command and sends it down the socket:

```
void request(String path) {
    try {
      String message = "GET " + path + "\n\n";
      os.write(message.getBytes());
      os.flush();
    } catch (IOException e) {
      System.err.println("Error in HTTP request");
    }
 }
```

Following an HTTP request, a response from the server is anticipated. The streams paradigm allows for a familiar code pattern via a `while`-statement in the `getResponse()` method.

```
void getResponse() {
  int c;
  try {
    while ((c = is.read()) != -1)
      System.out.print((char) c);
  } catch (IOException e) {
    System.err.println("IOException in reading from Web server");
  }
}
```

Tidying after a request is necessary to relinquish networking resources. This is easily achieved by closing the streams.

```
public void close() {
  try {
    is.close();
    os.close();
    soc.close();
  } catch (IOException e) {
    System.err.println("IOException in closing connection");
  }
}
```

Finally, the `static void main()` method pulls all the work together to implement the retriever.

```
public static void main(String[] args) {
   try {
     WebRetriever w = new WebRetriever("www.nus.edu.sg", 80);
     w.request("/NUSinfo/UG/ug.html");
     w.getResponse();
     w.close();
   } catch (UnknownHostException h) {
     System.err.println("Hostname Unknown");
   } catch (IOException i) {
     System.err.println("IOException in connecting to Host");
   }
 }
```

Note that where the resource from a server is an HTML file, a typical Web browser will render it according to the semantics of HTML. This functionality is not considered here. The complete WebRetriever class is shown in Listing 11-2.

```
import java.io.*;
import java.net.*;

class WebRetriever {
  Socket soc; OutputStream os; InputStream is;

  WebRetriever(String server, int port)
              throws IOException, UnknownHostException {
    soc = new Socket(server, port);
    os = soc.getOutputStream();
    is = soc.getInputStream();
  }
  void request(String path) {
    try {
      String message = "GET " + path + "\n\n";
      os.write(message.getBytes());
      os.flush();
    } catch (IOException e) {
      System.err.println("Error in HTTP request");
    }
  }
  void getResponse() {
    int c;
    try {
      while ((c = is.read()) != -1)
        System.out.print((char) c) ;
    } catch (IOException e) {
      System.err.println("IOException in reading from " +
                         "Web server");
    }
  }
  public void close() {
    try {
      is.close(); os.close(); soc.close();
    } catch (IOException e) {
      System.err.println("IOException in closing connection");
    }
  }
  public static void main(String[] args) {
    try {
      WebRetriever w = new WebRetriever("www.nus.edu.sg", 80);
      w.request("/NUSinfo/UG/ug.html");
      w.getResponse();
      w.close();
    } catch (UnknownHostException h) {
      System.err.println("Hostname Unknown");
    } catch (IOException i) {
      System.err.println("IOException in connecting to Host");
    }
  }
}
```

Listing 11-2: WebRetriever class.

11.3 Listener Sockets in Java

So far, we have considered Java code to initiate a socket connection to a server. It is time to consider how a server might be implemented in Java to be of service to other clients (which may or may not be implemented in Java). As pointed out earlier in the chapter, the asymmetric nature of clients and servers only occurs initially during matchmaking. Following that, communication over a socket is symmetric.

In the Java networking model, the additional functionality of listening out for prospective clients is handled by the `ServerSocket` class. Unlike the `Socket` class which requires a port number and the host machine which is providing service during object instantiation, a `ServerSocket` merely needs a port number since it is inviting requests from any machine.

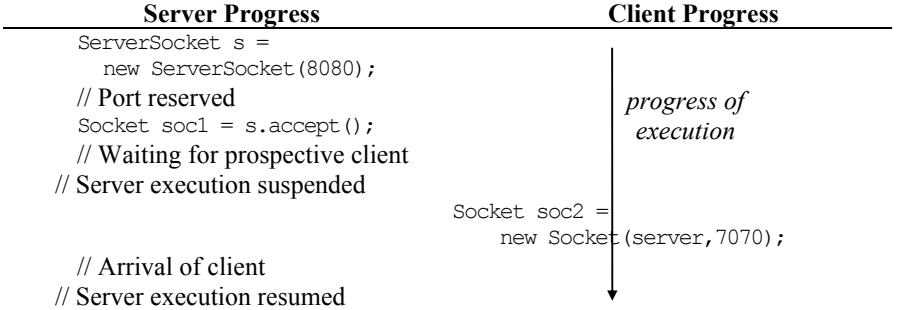
The creation of a `ServerSocket` reserves a port number for use, and prevents other prospective servers on the host from offering services at the same port:

```
ServerSocket s = new ServerSocket(8080);
```

The `accept()` method waits for the arrival a client. As such, execution of the server suspends until some client arrives, at which time, a socket connection is successfully established. In resuming execution, the `accept()` method also returns a suitable Socket instance to communicate with its client.

```
Socket soc = s.accept();
```

Assuming Java implementations of a server and corresponding client, the following chart illustrates relative progress.



`soc1` and `soc2` are complementary sockets, where one may read the contents written at the other, and vice versa.

11.3.1 Example Server: Simple Web Server

We now proceed to implement a simple Web server using the new functionality of the `ServerSocket` class. In comparison, it complements the Web retriever client developed earlier. The similar framework of socket communication for the Web

client and server has resulted in a very similar structure in the class skeleton in Listing 11-3.

```
import java.io.*;
import java.net.*;

class WebServe {

  Socket soc;
  OutputStream os; DataInputStream is;
  ....
  public static void main(String args[]) {
    try {
      ServerSocket s = new ServerSocket(8080);
      WebServe w = new WebServe(s.accept());
      w.getRequest();
      w.returnResponse();
    } catch (IOException i) {
      System.err.println("IOException in Server");
    }
  }
  WebServe(Socket s) throws IOException {
    soc = s;
    os = soc.getOutputStream();
    is = new DataInputStream(soc.getInputStream());
  }
}
```

Listing 11-3: WebServe skeleton.

The rationales for changes with respect to the client implement are elaborated below:

- The instantiation of the `ServerSocket` object is done in `main()` so that a WebServe object still contains Socket and Stream objects. It not only preserves the structure, but a more significant reason will be elaborated on later in the chapter.
- The input stream is a `DataInputStream` object so that HTTP request may be easily read on a per-line basis. Previously, the retriever client read data from the server and line structure was not considered.
- While the retriever client sends a request and then waits for a response from the server (via `request()` and `getResponse()`), the server performs the complement of receiving a request and then sending a response back to the client (via `getRequest()` and `returnResponse()` in `main()`).

As seen previously, a Web client makes a request for a resource via a HTTP `GET` command. However, in a typical Web client, this command is interspersed in a block of other HTTP request, such as:

```
GET /public_html/quick.html HTTP/1.0
Referer: http://sununx.iscs.nus.sg:8080/public_html/ic365/index.html
Connection: Keep-Alive
User-Agent: Mozilla/4.0 [en] (Win95; I)
Host: sununx.iscs.nus.sg:8080
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
```

Thus, while the `getRequest()` method scans the command block for a `GET` command, all other commands are ignored. Where a `GET` word is found it scans for the subsequent pathname of the requested resource. Processing terminates when the end of the block indicated by an empty line is detected.

```
void getRequest() {
   try {
     String message;

     while ((message = is.readLine()) != null) {
       if (message.equals(""))
         break;  // end of command block
       System.err.println(message);
       StringTokenizer t = new StringTokenizer(message);
       String token = t.nextToken();  // get first token
       if (token.equals("GET"))       // if token is "GET"
          resource = t.nextToken();   //   get second token
     }
   } catch (IOException e) {
     System.err.println("Error receiving Web request");
   }
}
```

The `getRequest()` method relies on the `StringTokenizer` class to strip away unwanted whitespaces to return raw symbols. On instantiating a `StringTokenizer` with the input string, tokens are returned on each call to `nextToken()`.

It would also be apparent that we need a new instance variable to hold the pathname of the resource requested. The `returnResponse()` method may then use this as a file name to return its contents.

```
void returnResponse() {
   int c;
   try {
     FileInputStream f = new FileInputStream("."+resource);
     while ((c = f.read()) != -1)
       os.write(c) ;
     f.close();
   } catch (IOException e) {
     System.err.println("IOException in reading in Web server");
   }
}
```

Note that "." has been added to the pathname of the resource as the Web server is expected to return contents of the current directory. Since the `finalize()` method

is unchanged, we have presented all the necessary code for our Web server. Again, the clean-up is implicit when the WebServe instance is not accessible.

The complete mini-WebServer is shown in Listing 11-4.

11.3.2 Running the Web Server

We have discussed the workings of Web client and Web server programs written in Java. While the former allows the retrieval of resources across the Internet, the latter can be tested in various scenarios.

- Where networking facilities are available, the Web server can be executed on one machine, and the Web client on another to make requests to the server.
- Where networking facilities are not available, both the Web server and client may be executed on different sessions of the same machine. The client may refer to the server host as the standard name "`localhost`".

In the same way that the Web client can access another server not written in Java, a typical Web client (such as FireFox or Internet Explorer) may be used to access our Java server.

```
import java.io.*;
import java.net.*;
class WebServe {
  Socket soc; OutputStream os; DataInputStream is;
  void getRequest() {
    try {
      String message;

      while ((message = is.readLine()) != null) {
        if (message.equals(""))
          break;  // end of command block
        System.err.println(message);
        StringTokenizer t = new StringTokenizer(message);
        String token = t.nextToken();  // get first token
        if (token.equals("GET"))       // if token is "GET"
          resource = t.nextToken();    //   get second token
      }
    } catch (IOException e) {
      System.err.println("Error receiving Web request");
    }
  }
  void returnResponse() {
    int c;
    try {
      FileInputStream f = new FileInputStream("."+resource);
      while ((c = f.read()) != -1)
        os.write(c);
      f.close();
    } catch (IOException e) {
      System.err.println("IOException in reading in Web " +
```

```
                        "server");
    }
  }
  public static void main(String args[]) {
    try {
      ServerSocket s = new ServerSocket(8080);
      WebServe w = new WebServe(s.accept());
      w.getRequest();
      w.returnResponse();
    } catch (IOException i) {
      System.err.println("IOException in Server");
    }
  }

  public void close() {
    try {
      is.close(); os.close(); soc.close();
    } catch (IOException e) {
      System.err.println("IOException in closing connection");
    }
  }
  WebServe(Socket s) throws IOException {
    soc = s;
    os = soc.getOutputStream();
    is = new DataInputStream(soc.getInputStream());
  }
}
```

Listing 11-4: WebServe class.

In practice, the scenario implemented is not practical because the server terminates after serving one resource. Most Web servers continue to run until the machine is rebooted or switched-off. This improvement may be effected by instantiating multiple WebServe objects by using an infinite `for`-loop. Note that old WebServe instances are garbage collected away. This minor change allows WebServe to work as a more practical Web server, and suitable with standard Web browsers.

```
public static void main(String args[]) {
  try {
    ServerSocket s = new ServerSocket(8080);
    for (;;) {
      WebServe w = new WebServe(s.accept());
      w.getRequest();
      w.returnResponse();
      w.close();
    }
  } catch (IOException i) {
    System.err.println("IOException in Server");
  }
}
```

11.4 Considering Multiple Threads of Execution

Many applications that require networking and graphical user interfaces have fairly complicated internal workings due to complex tasking schedules. For example, while

Webserve is processing the bulk of HTTP requests from a Web browser, it cannot perform other tasks such as servicing another Web client. In fact, Webserve currently allows only one socket connection at any time because instructions are executed sequentially. As such, it can only handle retrieval requests serially.

A graphical user interface typically has many concerns. For example, it needs to monitor mouse activity (in particular, if it has moved over hot areas); monitor keyboard activity (if keys have been depressed or special combinations of keys have been used); update screen areas with modified views; or even play an audio clip. If all these activities were to be merged into a sequence of serial instructions, it would be untidy and complex, and possibly error-prone.

Ideally, it would be simpler and neater if concurrent but independent activities were specified separately, yet easily executed concurrently. However, this is impossible with Java mechanisms because of strict sequential execution, which ultimately implies that one set of instructions is serially executed. This is also known as a *single thread of execution*.

Operating systems such as UNIX provide primitives like `fork()` to enable the creation of multiple processes to run multiple jobs. Unfortunately, processes tend to be expensive due to the need to maintain separate address spaces for each instance.

Threads have recently been promoted as a reasonable means for parallel execution, but without the high execution overheads since they share the same address space.

11.5 Creating Multiple Threads of Execution

So far, our code has been executed as a single thread. In this section, we consider the abstraction used in Java programs to initiate multithreaded execution. There are two specific ways to create threads: using the `Thread` class or using the `Runnable` interface.

11.5.1 Thread Creation Using the Thread Class

As we have seen with other mechanisms, the thread abstraction in Java is also encapsulated in an object. The basic functionality of threads is implemented by the `Thread` class in the `java.lang` package.

The following three properties about this `Thread` class allow for easy creation and execution of threads in Java:

(i) A new thread of execution may be created by instantiating a `Thread` object.
(ii) A newly created thread does not begin execution explicitly. Execution starts by invoking the method `start()`.
(iii) Execution of a thread involves invoking the `run()` method. An application-specific thread would typically be a class that extends from the `Thread` class, and then overrides the `run()` method to run application-specific code.

The following class definition in Listing 11-5 exploits the three properties and provides an extra thread for concurrent execution of the `run()` method.

```
class AddAndPrint extends Thread {

  private static final int TIMES = 30;
  private int val;

  AddAndPrint(int x) { val = x; }

  public void run() { // overrides default run()
    for (int i=val+TIMES; i>val; i--)
      System.err.println("run() : " + i);
  }
  public static void main(String arg[]) {
    AddAndPrint a = new AddAndPrint(243); // create thread
    a.start();                            // start thread run
    for (int i=0; i<TIMES; i++)           // proceed with code
      System.err.println("main() " + i);
  }
}
```

Listing 11-5: Multithread using the thread class.

A fragment from the execution of `AddAndPrint` in Listing 11-6 shows output from `main()` and `run()` interspersed. This indicates that execution of the two blocks is appropriately scheduled and proceeds conceptually in parallel.

```
...
main() 14
main() 15
main() 16
main() 17
main() 18
main() 19
main() 20
main() 21
main() 22
run() 253
run() 252
run() 251
run() 250
run() 249
main() 23
main() 24
main() 25
main() 26
main() 27
main() 28
main() 29
run() 248
run() 247
run() 246
run() 245
run() 244
```

Listing 11-6: Sample output.

11.5.2 Thread Creation Using the Runnable Interface

The class derivation method in the previous section has demonstrated a convenient means to creating new threads of execution, but it assumes that such classes must always be derived from the class `Thread`. This is a limitation for single inheritance languages such as Java, where very often a class should be derived from another class.

Fortunately, there is a second method of creating threads within classes that are not derived from the `Thread` class. Here, thread functionality involves implementing the `Runnable` interface, as illustrated in Listing 11-7.

While this method may initially seem different from the previous method, it does have some similarities:

(i) The side-effect of a thread is still encoded within the `run()` method which becomes mandatory for classes that implement `Runnable`.
(ii) Thread creation is still effected via a thread object. This time, however, an instance of the class that implements the `Runnable` interface (with a specific `run()`) method is passed as a parameter to the constructor.
(iii) The `start()` method begins thread execution.

```
class Special extends X implements Runnable {

  private final int TIMES = 30;
  private int val;

  AddAndPrint(int x) { val = x; }

  public void run() {  // necessary for Runnable
    for (int i=val+TIMES; i>val; i--)
      System.err.println("AddAndPrint value: " + i);
  }
  public static void main(String arg[]) {
    Special a = new Special(243);  // create non-thread object
    Thread thr = new Thread(a);    // create thread
    thr.start();                   // start thread execution
    for (int i=0; i<TIMES; i++)    // proceed with code
      System.err.println("main() " + i);
  }
}
```

Listing 11-7: Multithreading using the runnable interface.

11.6 Improvement of Web Server Example

As noted earlier, the simple Web server developed earlier processes requests from Web clients serially. As such, unless an HTTP request is processed and the specified resource is chosen to be returned to the Web client, the next request cannot even be accepted.

The brief modifications shown in Listing 11-8 improves the Web server so that instead of proceeding with `getRequest()` and `returnResponse()`, a thread object is

created to execute these methods within the `run()` method. In effect, while the new thread executes these methods, control in `static void main()` continues with another loop iteration to accept any new clients.

```
class WebServe implements Runnable {
  ....

  public void run() {
    getRequest();
    returnResponse();
    close();
  }
  public static void main(String args[]) {
    try {
      ServerSocket s = new ServerSocket(8080);
      for (;;) {
        WebServe w = new WebServe(s.accept());
        Thread thr = new Thread(w);
        thr.start();
      }
    } catch (IOException i) {
      System.err.println("IOException in Server");
    }
  }
}
```

Listing 11-8: Multithreading in WebServe.

The fact that other methods in the class definition remains unchanged shows how unobtrusively multithreading can be introduced into Java code.

11.7 Thread Synchronization and Shared Resources

Concurrent execution in the form of multiple threads of execution can allow for simpler code structures, but can also lead to intricate situations where resource sharing must be carefully considered. An example of a shared resource is a common variable.

While the relative progress of **independent** threads have no bearing to the end result (e.g., threads in our Web server are independent as they proceed on their own and do not depend on each other for more inputs), concurrent threads that interact (i.e., dependent) must be appropriately synchronized. The objective of synchronization ensures that the end result is deterministic, and does not vary depending on the relative execution speeds of each thread.

We first consider the issue of concurrent access/updates on shared variables. For simple expressions involving atomic variables, unexpected interactions will give nondeterministic results.

```
class JustAdd extends Thread {

  private final int N = 100000;
  private int val;

  JustAdd() { val = 0; }
  int value() { return val; }
  public void operate() {
    for (int i=0; i<N; i++) val = val+1;
  }
  public void run() {
    for (int i=0; i<N; i++) val = val+1;
  }
  public static void main(String arg[]) {
    JustAdd a = new JustAdd();     // create thread
    a.start();                     // start run()
    a.operate();                   // add using operate()
    System.out.println(a.value());
  }
}
```

Listing 11-9: Dependent threads.

The objective of `JustAdd` in Listing 11-9 is to create a thread to increment `val` (via `run()`) while the original thread increments `val` (via `operate()`). Depending on the Java implementation, the result printed may not be 200,000. That was the expected value since `val` would have been incremented 2*`N` times, `N` being 100,000.

Now, consider the circumstance where the execution of the assignment statements `val = val+1` in both threads is interleaved. Overwriting results if *both* expressions `val+1` are evaluated before `val` is reassigned. This scenario is possible if after evaluating `val+1`, the thread's time-slice runs out, thus allowing the other thread to evaluate using the "old" value of `val`. Here, the expected result of 200,000 will not be obtained.

Java provides the language keyword `synchronized` to demarcate code and data such that access of these regions by concurrent threads is serialized. This restriction allows shared data to be updated in mutual exclusion, and removes the possibility of interleaved execution.

Each Java object has an associated *use* lock, and a `synchronized` statement must acquire that lock before execution of its body. In the code fragment below, the lock associated with `g` is first obtained before `h.carefully()` is executed. Similarly, the lock is implicitly released on leaving the block.

```
synchronized (g) {
   h.carefully();
}
```

An instance method may also be specified as synchronized—such as `workAlone()` below. It is equivalent to its whole statement block being synchronized with respect to the current object as indicated by `this`. As such, the `synchronized` method `workAlone()`:

```
class X {
   ...
   synchronized void workAlone() {
      p();
      q();
   }
}
```

is equivalent to

```
class X {
   ...
   void workAlone() {
      synchronized (this) {
         p();
         q();
      }
   }
}
```

To enable consistent increments, the `JustAdd` class could be modified as in Listing 11-10.

We next consider the traditional consumer–producer synchronization problem by taking the example of two threads: an input thread that reads file contents into a buffer, and its partner output thread that writes buffer contents to the printer. The progress of each thread is dictated by the other: if the output thread proceeds faster, it must ultimately suspend when the buffer is empty. Similarly, if the input thread proceeds faster, it must ultimately suspend when the buffer is full.

```
class JustAdd extends Thread {

  private final int N = 100000;
  private int val;

  JustAdd() { val = 0; }
  int value() { return val; }

  synchronized void increment () { val = val+1; }
  public void operate() {
    for (int i=0; i<N; i++) increment();
  }
  public void run() {
    for (int i=0; i<N; i++) increment();
  }
  public static void main(String arg[]) {
    JustAdd a = new JustAdd();     // create thread
    a.start();                     // start run()
    a.operate();                   // add using operate()
    System.out.println(a.value());
  }
}
```

Listing 11-10: Synchronized threads.

The buffer is jointly used by both input and output threads, and is referred to as a shared resource. While it is accessible to both threads, but for correct operation, a thread must be delayed if it is progressing too fast. Such longer-term synchronization involving resource scheduling may be effected by using an object's lock as a monitor and the following methods for waiting and notification:

`wait()`	The `wait()` method causes the thread that holds the lock to wait indefinitely (so that it makes no progress in its execution) until notified by another about a change in the lock condition.
`notify()`	The `notify()` method wakes up a thread from amongst those waiting on the object's lock.

Let us consider a Writer thread that adds items to a buffer, and a partner Reader thread rhat removes items from the same. To simplify item generation in the Writer class (as illustrated in Listing 11-11), it will add the contents of a file.

```
class Writer extends Thread {
  Buffer b;
  FileInputStream fs;
  public void run() {
    int x;
    try {
      while ((x = fs.read()) != -1)
        b.put((char) x);
      b.put('\032');
   } catch (Exception e) {
     System.err.println("Cannot read");
     System.exit(1);
   }
  }
  Writer(String fname, Buffer b) {
    this.b = b;
    try {
      fs = new FileInputStream(fname);
    } catch (Exception e) {
      fs = null;
      System.err.println("Cannot open "+fname);
      System.exit(1);
    }
  }
}
```

Listing 11-11: Writer thread.

Similarly, the Reader class will read from the buffer and confirm the contents by writing to the standard output stream, as shown below in Listing 11-12.

```
class Reader extends Thread {
  Buffer b;
  public void run() {
    char x;
    while ((x = b.get()) != '\032')
      System.out.print(x);
  }
  Reader(Buffer b) {
    this.b = b;
  }
}
```

Listing 11-12: Reader thread,

The following are some points concerning the `Writer` and `Reader` classes:

- The `Buffer` object is shared between the `Reader` object which reads from it, and the Writer object which writes to it. It must be accessible to both objects, and is achieved via passing it through the constructor method.
- Unless the `Reader` object is notified about the end of stream, it will wait indefinitely when no more items from the `Writer` object is forthcoming. To avoid this situation, the `Writer` thread puts out the character `^Z` to signal the end of the stream. As such, the `Reader` terminates on receiving this item.

We now consider how the `Buffer` class, with `put()` and `get()` methods may be implemented to work consistently despite concurrent accesses and different rates of thread execution.

Firstly, the basic requirement of a buffer is to keep items placed by `put()` in its internal state until they are retrieved by `get()`. This is illustrated in Listing 11-13. To ensure smooth execution of `Reader` and `Writer` threads, we allow the buffer to hold more than one item via a circular queue indicated by `front` and `rear` indexes.

Note that `front` and `rear` moves down the array and wraps around from the last item to the first via the remainder operation `%`. The distance between the two indexes indicates the number of buffered items.

```
class Buffer {

  final int MAXSIZE = 512;
  char keep[];
  int count, front, rear;

  public char get() {
    char x = keep[rear];
    rear = (rear+1) % MAXSIZE;
    count--;
    return x;
  }
```

```
  public void put(char x) {
    keep[front] = x;
    front = (front+1) % MAXSIZE;
    count++;
  }
  Buffer() {
    keep[] = new char [MAXSIZE];
    count = 0;
    front = rear = 0;
  }
}
```

Listing 11-13: Shared buffer.

Secondly, concurrent access of the `Buffer` object from `Writer` and `Reader` threads dictate that calls to `get()` and `put()` should not overlap so that the integrity of the internal state is preserved during updates. This may be achieved by tagging these methods as `synchronized`, so that access to `Buffer` instances must be implicitly preceded by acquiring the access lock. Subsequently, the access lock is released following access.

Thirdly, the `get()` method should cause the calling thread to wait when the `Buffer` object is already empty. Correspondingly, the `put()` method should notify a thread waiting to access the `Buffer` object that an item is available. However, the `put()` method should also cause the calling thread to wait when the Buffer object is already full. Similarly, the `get()` method must notify a thread waiting to access the `Buffer` object that a slot is now available for an item. This is illustrated in the improved `Buffer` class in Listing 11-14.

```
class Buffer {

  final int MAXSIZE = 512;
  char keep[];
  int count, front, rear;

  public synchronized char get() {
    while (count == 0)
      wait();
    char x = keep[rear];
    rear = (rear+1) % MAXSIZE;
    count--;
    notify();  // that a space is now available
    return x;
  }
  public synchronized void put(char x) {
    while (count == MAXSIZE)
      wait();
    keep[front] = x;
    front = (front+1) % MAXSIZE;
    count++;
    notify(); // that an item is now available
  }
```

```
  Buffer() {
    keep[] = new char [MAXSIZE];
    count = 0;
    front = rear = 0;
  }
}
```

Listing 11-14: Synchronized buffer.

To summarize, the additional code for thread synchronization includes the `synchronized` tag together with `wait()` and `notify()` method calls. The former involves object access locks for short-term synchronization to solve the problem of concurrent access to object attributes. The latter concerns long-term synchronization to solve the problem of resource allocation.

Note that our Java buffer solution is slightly different from other languages. Firstly, waiting for a resource (either for an item in the buffer to retrieve or a slot in the buffer to receive a new item) involves repeated testing of the condition in a `while`-loop, for example:

```
while (count == 0)
   wait();
```

instead of

```
if (count == 0)
   wait();
```

Secondly, the method call to `notify()` does not specify explicitly which thread should be alerted. In fact, this seems more dubious when we consider that there are two conditions for which threads look out for—an empty buffer and a full buffer.

To explain these observations, we recall the nature of the `wait()` and `notify()` methods. The `notify()` method awakens one of the threads waiting on the object's lock queue. This thread may be waiting for a buffer item or an empty buffer slot. Since the `notify()` method does not allow the specification of a condition, awaking from a `wait()` call does not guarantee that the condition for it is fulfilled. As such, the `while`-loop ensures that an anticipated condition be confirmed when waking up from `wait()`. Therefore, the thread continues to wait if the condition is not yet fulfilled.

11.8 Summary

This chapter has discussed the basics of two important areas in Java programming: *networking* and *multithreading*. While implementing these functionalities in other languages may be nontrivial, Java has the advantage in that these functionalities are well-encapsulated into objects, and supplied with appropriate APIs. Any additional functionality is thus well-integrated into the language framework.

Network socket connections in Java depend on:

- Socket; and
- ServerSocket classes

as found in the `java.net` package. The `Socket` class enables Java applications to initiate client connections to other machines with a listening server. The complementary role of implementing a listening server is provided by the `ServerSocket` class. Successful connections return `InputStream` and `OutputStream` objects (via the `getInputStream()` and `getOutputStream()` methods), with subsequent operations similar to that seen for input/output operations on streams and files.

Multiple socket connections (involving multiple hosts) provide a real example when multithreading is very useful in easing processing logic. Without this means of specifying parallel tasks, these tasks must be serialized, or additional logic may be used to multiplex multiple tasks into a single subroutine via state save/restore operations. However, this is error-prone and an additional chore that many programmers would gladly avoid.

The Java API facilitates thread creation and execution via:

- `Thread` class; and
- `Runnable` interface.

Both are almost identical in that thread execution is initiated by the `start()` method, which causes the user-defined `run()` method to execute but does not wait for its termination.

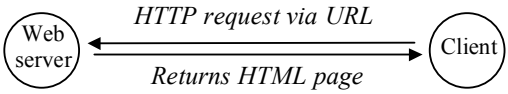
11.9 Exercises

1. A telnet client may be used to simulate a finger request to obtain user information from a finger server (which typically listens to port 79). The user name is entered after connection to the finger server is established:

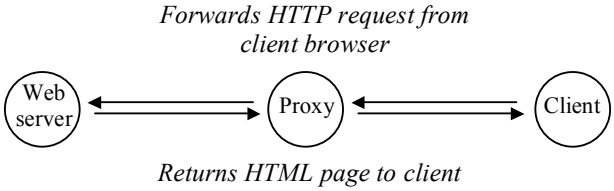
```
$ telnet leonis.nus.edu.sg 79
Trying 137.132.1.18...
Connected to leonis.nus.edu.sg.
Escape character is '^]'.
isckbk
Login name: isckbk                    In real life: Kiong Beng Kee
Directory: /staff/isckbk              Shell: /usr/bin/ksh
Last login Fri Feb 13 14:56 on ttym4 from dkiong.iscs.nus.
No Plan.
Connection closed by foreign host.
```

   Simplify the procedure by implementing a finger client in Java that will obtain the user name and server from the command-line, and print out details obtained from the finger server via a socket connection.
2. Implement a reverse server in Java. It works by listening for clients on port 1488. For each client socket connection, it reads the first line and returns the reversed string to the client.
3. Many sites with a firewall implement a proxy server as a legitimate means to go through the network barrier. The Web proxy thus sits between server

and clients. It forwards a client's request to the server, and thereafter relays the server's reply to the original client.



Compared with the proxyless framework shown above, the proxy application has two socket connections. It listens for clients at one end, and then launches the request to the server proper via the other, as shown graphically.



Implement a proxy application to eavesdrop on how a standard Web browser requests for HTML pages as well as submit forms.

4. Suggest a suitable representation for matrices. Discuss sequential and threaded solutions for matrix multiplication.

12

Generics and Collections Framework

Prior to Java version 1.5, a Java class could only be written with its attribute and parameter types in mind. One could never translate a type-independent concept into a class. With the introduction of Java Generics in Java 2 version 1.5, we can define a class without specifying the type for certain attributes and/or parameters. The type is specified when the class is instantiated.

In this chapter you will learn how to use this feature of Java. You will also learn about the Java Collections framework that has a predefined set of generic classes and algorithms that can be used to store and manipulate a collection of objects.

12.1 Introduction

In Java 5 (i.e., J2SE 1.5), many new features are introduced. One of the significant features is the concept of generic programming. This feature had been debated for many years before it was introduced in Java. C++ programmers will be very familiar with the concept of generic programming, as in C++ templates. Though "Java generics" is quite similar to templates, there are also many significant differences between the two implementations. We will first discuss the Java generics and then elaborate on the Collections Framework that contains many useful generic interfaces, methods and classes.

12.2 Rationale Behind Generics

Before understanding the concept of generics, it is important that we are familiar with the problems and issues in using the Java language without generics. This will lead us to the motivation behind Java generics.

12.2.1 The Problem

Let us consider some generic concepts such as the data structure `Stack` or the algorithm to sort an array. When we start to code classes or methods relating to such concepts, we have to consider the type of elements that we are dealing with. Take a look at the code for the slightly modified `Stack` class in Listing 12-1.

```
class StackItem {
  private Integer item = 0;
  private StackItem previous;
  public Integer getItem() { return item;}
  public void setItem(Integer x) { item = x; }
  public StackItem getPrevious() { return previous; }
  public void setPrevious(StackItem p) { previous = p; }
  public StackItem() { previous = null; }
}

class Stack {
  private StackItem top, temp;
  private int size=10;
  public Integer pop() {
    Integer x = 0;
    if (empty()) System.err.println("stack underflow");
    else {
      x    = top.getItem();
      top  = top.getPrevious();
      size--;
    }
     return x;
  }
  public void push(Integer x) {
    if (full()) System.err.println("stack overflow");
    else {
      temp = top;
      top  = new StackItem();
      top.setPrevious (temp);
      top.setItem(x);
      size++;
      System.out.println("pushed " + x + "into stack");
    }
  }
  public int size() {return (size); }
  public boolean empty() { return (size() == 0); }
  public boolean full() { return false;}
  public Stack() {
    top = null; size = 0;
  }
}
```

Listing 12-1: Stack implementation for storing integer objects.

Using this `Stack` class you are only able to store elements of the type `Integer`. What if you want to store elements of the type `String`? You will have to rewrite the `Stack` class by substituting the type Integer to type `String`, although most of the logic behind the code is the same. The only changes you will need to make are in the *type* of contents array and the *type* information in all the

methods that manipulate the contents. This poses a reuse problem. How do we reuse the common code and let the *type* alone vary?

We actually have a solution in Java to deal with the above reuse problem. Now consider the code for another version of the `Stack` class (created by substituting the `Integer` type to `Object` type) in Listing 12-2.

```
class StackItem {
  private Object item = 0;
  private StackItem previous;

  public Object getItem() { return item;}
  public void setItem(Object x) { item = x; }
  public StackItem getPrevious() { return previous; }
  public void setPrevious(StackItem p) { previous = p; }
  public StackItem() { previous = null; }
}

class Stack {
  private StackItem top, temp;
  private int size=10;

  public Object pop() {
    Object x = 0;
    if (empty()) System.err.println("stack underflow");
    else {
       x = top.getItem();
       top = top.getPrevious();
       size = size - 1;
     }
     return x;
  }
  public void push(Object x) {
    if (full()) System.err.println("stack overflow");
    else {
      temp = top;
      top = new StackItem();
      top.setPrevious (temp);
      top.setItem(x);
      size = size + 1;
      System.out.println("pushed " + x + " into stack");
    }
  }
  public int size() {return (size); }
  public boolean empty() {
    return (size() == 0);
  }
  public boolean full() { return false;}

  public Stack() {
    top = null;
    size = 0;
  }
}
```

Listing 12-2: Stack implementation for storing any type of objects.

As we know, in Java, the class called `Object` is the superclass of all classes. We have changed the type of contents and the type information in all the methods that manipulate the contents to `Object` type. We can reuse this `Stack` class as a `Stack` that can hold `Integer` objects or any other type of objects. Now take a look at the code that makes use of this `Stack` class in Listing 12-3.

```
class StackApp {
  public static void main(String args[]) {
    int numberOfItems;
    numberOfItems = Integer.parseInt(args[0]);
    Stack sInt = new Stack();
    for (int i = 0; i < numberOfItems; i++)
      sInt.push(new Integer(i));
    Stack sStr = new Stack();
     String str1 = "Testing ";
     for (int i = 0; i < numberOfItems; i++)
       sStr.push(str1 + i);
  }
}
```

Listing 12-3: Usage of the Stack class for storing different type of objects.

You notice in the highlighted statements that we are able to use the `Stack` class to store `Integer` objects as well as `String` objects.

12.2.2 Run-time Type Identification (RTTI)

While the above solution seems to solve the problem of reuse, that is, reuse of the same class `Stack` to store different types of objects, it poses yet another problem. Take a look at the usage of `Stack` class in Listing 12-4.

In Listing 12-4, you will notice that the same `Stack` object called s is used to store not only `Integer` objects but also `String` objects. This is often undesirable.

```
class StackApp {
  public static void main(String args[]) {
    int numberOfItems;
    numberOfItems = Integer.parseInt(args[0]);

    Stack s = new Stack();
    for (int i = 0; i < numberOfItems; i++)
      s.push(new Integer(i));

    String str1 = "Testing ";
    for (int i = 0; i < numberOfItems; i++)
      s.push(str1 + i);
  }
}
```

Listing 12-4: Using the same Stack object to store different type of objects.

You can detect the type of object before pushing an element into the Stack object and prevent this from happening by using the run-time-type identification (RTTI) facility of Java. This can be done by using the keyword *instanceof* as shown in Listing 12-5.

```
class InvalidTypeException extends Exception {}

class StackApp {
  private Stack s = new Stack();

  void pushInteger(Object o)
           throws InvalidTypeException {
     if (o instanceof Integer) s.push(o);
     else throw new InvalidTypeException();
  }

  public static void main(String args[]) {
    int numberOfItems;
    numberOfItems = Integer.parseInt(args[0]);
    StackApp sapp = new StackApp();
    try {
      for (int i = 0; i < numberOfItems; i++)
        sapp.pushInteger(new Integer(i));
        String str1 = "Testing ";
        sapp.pushInteger(str1 + 0);
    } catch (InvalidTypeException e) {
        System.out.println("WrongType pushed");
    }
  }
}
```

Listing 12-5: Run-time-type identification in Java.

The output is as follows:

```
pushed 0 into stack
pushed 1 into stack
pushed 2 into stack
pushed 3 into stack
WrongType pushed
```

This solution also does not seem elegant. Besides being clumsy, this solution relies on programmers' diligence in performing the run-time-type identification, and is therefore error prone. Also, such errors are not detectable during compile time. These arguments have led to the introduction of the concept of generics in the Java language.

12.3 Java Generics

"Generics" is also known as parameterized types, where the types can be specified as parameters to a class or a method. This feature allows us to express some generic concepts without having to specify *type* for some variables.

12.3.1 Generic Class

Let us rewrite the `Stack` class, where we parameterize the type of objects that can be stored in the `Stack`. See Listing 12-6.

With all the "pointy" brackets introduced, the syntax needs a little bit of getting used to. "E" is the type that can be parameterized. It can be specified using any character such as E, X, and so on. The convention is to use a single capital letter. A class that needs a parameter type is declared with an addition of <E>, as seen in the declaration of class StackItem<E> and class Stack<E>. A class can also have more than one parameterized types, specified by a comma-separated list. For example, class Map<K,V>.

```
class StackItem<E>{
  private E item = null;
  private StackItem<E> previous;
  public E getItem() { return item;}
  public void setItem(E x) { item = x; }
  public StackItem<E> getPrevious() { return previous; }
  public void setPrevious(StackItem<E> p) { previous = p; }
  StackItem() { previous = null; }
}
class Stack<E>{
  private StackItem<E> top, temp;
  private int size=10;
  public E pop() {
    E x = null;
    if (empty()) System.err.println("stack underflow");
    else {
      x = top.getItem();
      top = top.getPrevious();
      size = size - 1;
    }
    return x;
  }
  public void push(E x) {
    if (full()) System.err.println("stack overflow");
    else {
      temp = top;
      top = new StackItem<E>();
      top.setPrevious (temp);
      top.setItem(x);
      size = size + 1;
      System.out.println("pushed " + x + " into stack");
    }
  }
  public int size() {return (size); }
  public boolean empty() { return (size() == 0); }
  public boolean full() { return false;}

  public Stack() {
    top = null;
    size = 0;
  }
}

class StackApp {
  public static void main(String args[]) {
    int numberOfItems;
    numberOfItems = Integer.parseInt(args[0]);
```

```
    Stack<Integer> sInt = new Stack<Integer>();
    for (int i = 0; i < numberOfItems; i++)
      sInt.push(new Integer(i));

    Stack<String> sStr = new Stack<String>();
    String str1 = "Testing with Generics ";
    for (int i = 0; i < numberOfItems; i++)
      sStr.push(str1 + i);
  }
}
```

Listing 12-6: Stack class and usage using Java generics.

So, we have managed to write a Generic class that can be parameterized using any type! The actual type is specified when we instantiate the class, as can be seen in the following lines extracted from the `StackApp` class:

```
Stack<Integer> sInt = new Stack<Integer>();
Stack<String>  sStr = new Stack<String>();
```

The first object `sInt` is an instance of a `Stack` that can hold objects of the type `Integer`. The compiler will flag a compile-time error if objects of any other type are pushed into `sInt`.

Similarly, the second object `sStr` is an instance of a `Stack` that can hold objects of the type `String`. Note that generics do not work with primitive types (e.g., `int`, `char`, etc.).

12.3.2 Generic Method

In the previous section, we learnt how to write and use a generic class. Individual methods in a class similarly can be declared as a generic method. Here is an example in Listing 12-7. We can declare a method to be generic by placing the `<T>` in front of the method, where `T` is the parameterized type. There is no change in the way generic methods are invoked. By declaring a method as a generic method, we are able to reuse the `printarray` method for any type of array.

```
public class GenApp {
  private static <T> void printarray(T[] a)     {
    for (Object o : a)
      System.out.println (o);
  }
  public static void main(String args[]) {
    Integer iarr[] = new Integer[3];
    iarr[0] = new Integer(10);
    iarr[1] = new Integer(20);
    iarr[2] = new Integer(30);
    printarray(iarr);

    Float farr[] = new Float[3];
    farr[0] = new Float(48.0);
    farr[1] = new Float(59.0);
    farr[2] = new Float(67.0);
    printarray(farr);
  }
}
```

Listing 12-7: A generic method example.

Does the `for` loop inside the `printarray` look strange? Well, this is another new feature of Java 5. This form of `for` loop is called "`for-each`" construct. The following two `for` loop constructs are equivalent:

```
for (Object o : a)
  System.out.println (o);

for (int i = 0; i < a.length; i++)
  System.out.println(a[i]);
```

The expression `(Object o: a)` stands for all objects in the array `a`, referred to using the variable `o`.

12.4 Collections Framework

We are familiar with an array that is used to store a set of objects. For example a set of `String` objects can be stored in an array as follows:

```
String strArray[] = new String[10];
strArray[0] = "first";
strArray[1] = "second";
strArray[2] = null;
```

The problem with arrays is that we need to know the length of the array right upfront when we create it. In situations where we are not sure about the number of objects that we would need to store, an array is not very convenient to use. In Java 5, some predefined classes and interfaces that help us store and manipulate a collection of objects, are available in the Collections Framework.

12.4.1 Collections Interface

The framework consists of a few core interfaces known as the *Collection Interfaces* as shown in Figure 12-1.

Some of the commonly used concrete implementation classes included in the Collections Framework are listed below in Figure 12-2.

- **List:** This is a data structure that can contain a collection of objects, similar to an array, without having to specify the size. The list can grow or shrink according to the number of objects. *Example*: List of items in a shopping cart.
- **Set:** This is a data structure that can contain a collection of objects without any duplicates. *Example*: A set of courses registered by a student.
- **Map:** A data structure that allows storage objects that have a `<key, value>` pair. Example: Student mark list, the key being the student matriculation number and the value being the mark obtained. This data structure allows us to retrieve a value for a given key. *Example*: We are able to obtain the mark obtained by a student with a specific matriculation number.

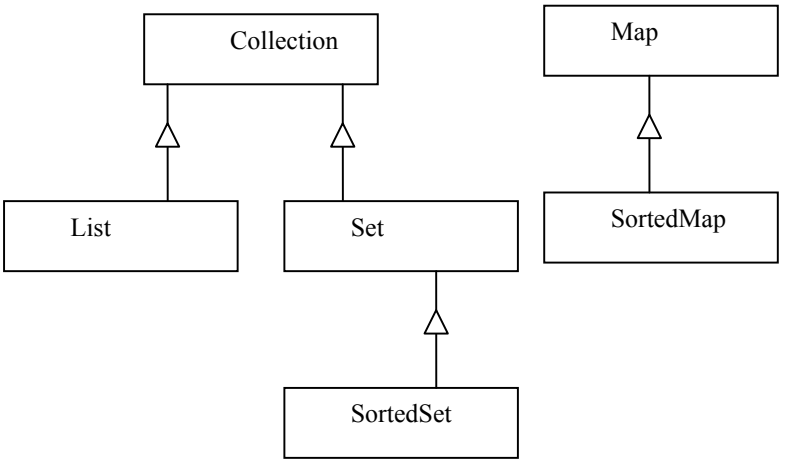


Figure 12-1: Java Collections Framework's core interfaces.

Many of these interfaces, classes, and methods are coded using the Java generics facility.

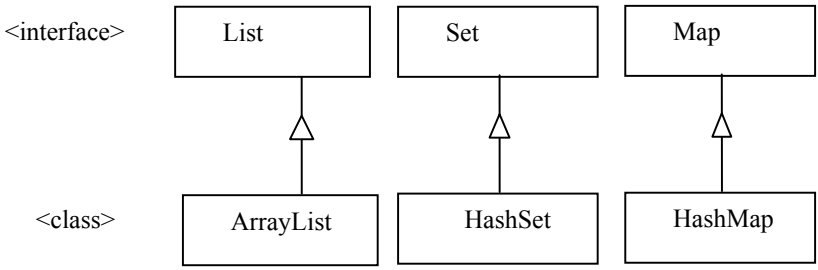


Figure 12-2: Commonly used Collection classes.

12.4.2 ArrayList Class

`ArrayList` is an implementation class provided in the Collection Framework that implements the `List` interface. We can make use of this class to store objects, retrieve and manipulate them as a collection.

12.4.2.1 Declaring and Using ArrayList Class

Listing 12-8 shows a typical declaration, instantiation and use of an `ArrayList` that can hold a collection of `String` objects. You can similarly declare, instantiate and use `ArrayLists` of other types of objects.

```
import java.util.*;
public class ArrayListEx {
  public static void main(String[] args) {
    ArrayList<String> strList = new ArrayList<String>();
    String s1 = "first";
    String s2 = "second";
    String s3 = "third";
    strList.add(s1);
    strList.add(s2);
    strList.add(s3);
}
```

Listing 12-8: Illustration of instantiation and use of ArrayList.

In the above code, it is important to include the import statement in the beginning of the program.

```
import java.util.*;
```

This statement ensures that the predefined classes from the Collection Framework, such as `ArrayList`, are included during compilation.

The following statement creates an `ArrayList` instance. The `ArrayList` is a generic class for which we need to specify the type of objects that it can hold. In this example, the `ArrayList` object called `strList` is defined to hold objects of the type `String`.

```
ArrayList<String> strList =
                new ArrayList<String>();
```

Objects can be added to the `ArrayList` using the `add()` method. The available methods for an `ArrayList` are shown in Table 12-1.

```
String s1 = "first";
strList.add (s1);
```

Elements can be added to an `ArrayList` from yet another collection. For example:

```
ArrayList<String> strList2 = new ArrayList<String>();
strList2.addAll(strList);
```

Table 12-1: Methods available for ArrayList.

Collection Interface			
Query Operations	Modification Operations	Bulk Operations	Comparison and Hashing
`int size()`	`boolean add(E o)`	`boolean containsAll (Collection<?> c)`	`boolean equals (Object o)`
`boolean isEmpty()`	`boolean remove (Object o)`	`boolean addAll (Collection <? extends E> c)`	`int hashCode()`
`boolean contains (Object o)`		`boolean removeAll (Collection<?> c)`	
`Iterator <E> iterator()`		`boolean retainAll (Collection<?> c)`	
`Object[] toArray()`		`void clear()`	
`<T> T[]toArray (T[] a)`			

List Interface			
Query Operations	Search Operations	Bulk Operations	Positional Access
`ListIterator <E> listIterator (int index)`	`int indexOf (Object o)`	`boolean addAll (int index, Collection <? extends E> c)`	`E get (int index)`
`int lastIndexOf (Object o)`	`int lastIndexOf (Object o)`		`E set (int index, E element)`
`List<E> subList (int fromIndex, int toIndex)`			`void add (int index, E element)`
			`E remove (int index)`

ArrayList Class	
Constructors & Creational Operations	Other Operations
`ArrayList()`	`ensureCapacity (int minCapacity)`
`ArrayList(Collection<? extends E> c)`	`trimToSize()`
`ArrayList(int initialCapacity))`	`removeRange(int fromIndex, int toIndex)`
`clone()`	

12.4.2.2 Traversing an ArrayList

Traversing through the elements of an `ArrayList`, or for that matter many of the Collection Framework classes, can be done either by using the `for-each` construct

or by using an `Iterator`. The code below shows iteration of an `ArrayList` using a `for-each` construct:

```
for (String s : strList)
 System.out.println(s);
```

The code below shows iteration of an ArrayList using an `Iterator`:

```
Iterator<String> iter = strList.iterator();
while( iter.hasNext())
 System.out.println(iter.next());
```

An `Iterator` has the following method:

```
boolean hasNext()
E next()
void remove()
```

Alternatively, you can also use a `ListIterator`. A `ListIterator` has the following methods:

```
void add(E o);
boolean hasNext();
boolean hasPrevious();
E next();
int nextIndex();
E previous();
int previousIndex();
void remove();
void set(E o);
```

12.4.3 HashSet Class

`HashSet` is an implementation class provided in the Collection framework that implements the `Set` interface. We can make use of this class to store objects, retrieve and manipulate them as a collection, without having duplicates.

12.4.3.1 Declaring and Using a HashSet Class

We can use `HashSet` to store unique objects in a Collection. Listing 12-9 shows a typical declaration, instantiation and use of an `HashSet` that stores objects of the type `String`.

```
import java.util.*;
public class HashSetEx {
  public static void main(String[] args) {
    HashSet<String> hashList = new HashSet<String>();
    String s1 = "John";
    String s2 = "Elliot";
    System.out.println("Added = "+ hashList.add(s1));
    System.out.println("Added = "+ hashList.add(s2));
    System.out.println("Added = "+ hashList.add(s2));
  }
}
```

Listing 12-9: Illustration of instantiation and use of HashSet.

The execution of this program would result in the following output. As you can see in the last statement, a `HashSet` does not allow duplicates to be added.

```
Added = true
Added = true
Added = false
```

You can detect duplicates by checking the return value of the `add()` method as illustrated in Listing 12-10.

```
import java.util.*;
public class HashSetEx {
  public static void main(String[] args) {
    HashSet<String> hashList = new HashSet<String>();
    String sArray[] = new String[6];
    sArray[0] = "John";
    sArray[1] = "Francis";
    sArray[2] = "Elliot";
    sArray[3] = "Francis";
    sArray[4] = "Mary";
    sArray[5] = "John";
    for (int i = 0; i < 5; i++){
      if (hashList.add(sArray[i])== false)
        System.out.println("Duplicate found: " + sArray[i]);
      else System.out.println ("Added: " + sArray[i]);
    }
  }
}
```

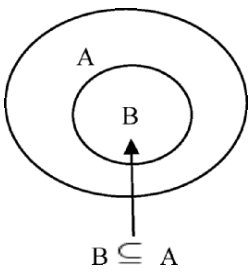
Listing 12-10: Detecting duplicates in `HashSet`.

The execution of this program would result in the following output. As you can see that a `HashSet` does not allow duplicates to be added.

```
Added: John
Added: Francis
Added: Elliot
Duplicate found: Francis
Added: Mary
Duplicate found: John
```

12.4.3.2 Subset, Union, Intersection and Complement

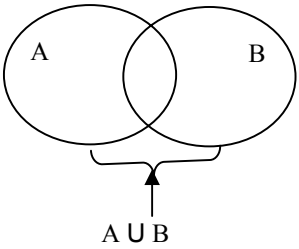
If every member of the Set *B* is also contained in Set *A*, then *B* is said to be a **subset** of A.



You can determine whether a `HashSet` is a ***Subset*** of another `HashSet` by using the `containsAll()` method as follows:

```
if (firstSet.containsAll(secondSet) == true)
  System.out.println("secondSet is a subset of firstSet");
else
  System.out.println("secondSet is not a subset of firstSet");
```

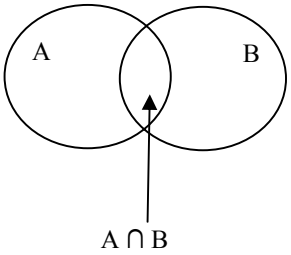
The ***Union*** of Set A and Set B is defined as the set of all elements that can be found either in Set A or in Set B.



The ***Union*** of two `HashSets` can be derived by using the `addAll()` method as follows:

```
Set<String> unionSet = new HashSet<String>(firstSet);
unionSet.addAll(secondSet);
```

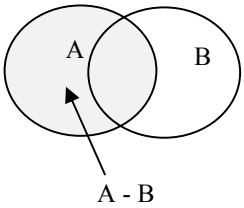
The ***Intersection*** of Set A and Set B is defined as the set of all elements found in both Set A and Set B.



The ***Intersection*** of two `HashSets` can be derived by using `retainAll()` method as follows:

```
Set<String> intersectionSet = new HashSet<String>(firstSet);
intersectionSet.retainAll(secondSet);
```

The ***Complement*** of Set B in Set A is defined as the set of all elements in Set A that are not found in Set B.



The ***Complement*** of two `HashSets` can be derived by using `removeAll()` method as follows:

```
Set<String> differenceSet = new HashSet<String>(firstSet);
differenceSet.removeAll(secondSet);
```

The available methods for a `HashSet` are shown in Table 12-2.

Table 12-2: Methods available for `HashSet`.

Collection Interface/Set Class			
Query Operations	Modification Operations	Bulk Operations	Comparison and Hashing
`int size()`	`boolean add(E o)`	`boolean containsAll (Collection<?> c)`	`boolean equals (Object o)`
`boolean isEmpty()`	`boolean remove (Object o)`	`boolean addAll (Collection <? extends E> c)`	`int hashCode()`
`boolean contains (Object o)`		`boolean removeAll (Collection<?> c)`	
`Iterator<E> iterator()`		`boolean retainAll (Collection<?> c)`	
`Object[] toArray()`		`void clear()`	
`<T> T[] toArray(T[] a)`			

`HashSet` Class
Constructors and Creational Operations
`HashSet()`
`HashSet(Collection<? extends E> c)`
`HashSet(int initialCapacity))`
`HashSet(int initialCapacity, float loadFactor)`
`clone()`

12.4.4 `HashMap` Class

`HashMap` is an implementation class provided in the Collection framework that implements the `Map` interface. A `HashMap` is used to hold elements with `<key,value>` associations. A key is associated with a value and stored in a `HashMap`. Values can then be retrieved for a given key. By definition, `HashMaps` are not allowed to contain duplicates. This class does not guarantee the order in which the `<key, value>` pairs are stored.

12.4.4.1 Declaring and Using HashMap Class

The program in Listing 12-11 illustrates the declaration, instantiation and use of a `HashMap` that contains objects of the type Student. The **key** for each student is his/her matriculation number which is of the type `Integer`.

```
import java.util.*;
import java.lang.Integer;
class Student {
  int matricNumber;
  String name;
  public Student(int matricNumber, String name){
    this.matricNumber = matricNumber;
    this.name = name;
  }
  int getMatricNumber() { return matricNumber; }
  String getName() { return name; }
}
public class HashMapEx {
  public static void main(String[] args) {
    HashMap<Integer, Student> studentMap =
      new HashMap<Integer, Student>();
    Student s1 = new Student (1,"Tan Puay Hwee");
    Student s2 = new Student (2,"David Brown");
    Student s3 = new Student (3,"Satish Gupta");
    studentMap.put(new Integer(s1.getMatricNumber()), s1);
    studentMap.put(new Integer(s2.getMatricNumber()), s2);
    studentMap.put(new Integer(s3.getMatricNumber()), s3);

    Student retStudent = null;
    for (int i = 1; i < 4; i++){
      Integer key = new Integer(i);
      retStudent = studentMap.get(key);
      System.out.println("Student with key = " +
         key + " is called " + retStudent.getName());
    }
  }
}
```

Listing 12-11: Illustration of instantiation and use of `HashMap`.

On executing the program, the following output will be produced:

```
Student with key = 1 is called Tan Puay Hwee
Student with key = 2 is called David Brown
Student with key = 3 is called Satish Gupta
```

A `HashMap` is declared by specifying the two parameterized types, the first being the key type and the second being the value type. In our example the `HashMap` called `studentMap` is declared and instantiated as shown below. The key type is `Integer` (matriculation number) and the value type is `Student` (a `Student` object).

```
HashMap<Integer, Student> studentMap =
 new HashMap<Integer, Student>();
```

Entries can be added to the `HashMap` using the `put()` method as shown below:

```
Student s1 = new Student (1,"Tan Puay Hwee");
studentMap.put(new Integer(s1.getMatricNumber()), s1);
```

An `Integer` object is created using the matriculation number (`1`) of the student (`s1`).

`Student` objects can be retrieved by using the `get()` method with the matriculation number of the student as shown below:

```
Integer key = new Integer(i);
retStudent = studentMap.get(key);
```

The available methods for a `HashMap` are shown in Table 12-3.

Table 12-3: Methods available for `HashMap`.

AbstractMap Class		
Query Operations	Modification Operations	Alternative Views
boolean containsKey (Object key)	void clear()	Set<Map.Entry<K,V>> entrySet()
Boolean containsValue (Object value)	V put(K key, V value)	Set<K> keySet()
V get (Object key)	void putAll(Map<? extends K, ? extends V> m)	Collection<V> values()
boolean isEmpty()	V remove (Object key)	
int size()		

HashMap Class
Constructors & Creational Operations
`HashMap()`
`HashMap(int initialCapacity)`
`HashMap(int initialCapacity, float loadFactor)`
`HashMap(Map<? extends K,? extends V> m)`
`clone()`

12.5 Sorting Collections

JDK1.5 has yet another class called `java.utils.Collections`. This class contains many general purpose algorithms that work with the various Collection classes such as `ArrayList`.

12.5.1 Sort Algorithm

An `ArrayList namesList` may be sorted in the ascending order using the sort method in the Collections class as follows:

```
Collections.sort(nameList);
```

Note that the name of this class is called Collections—with an "s." A sample program illustrating the use of sort algorithm is shown in Listing 12-12:

```
import java.util.*;
public class SortExampleApp {
  public static void main(String[] args){
    ArrayList<String> nameList = new ArrayList<String>();
    nameList.add("One");
    nameList.add("Two");
    nameList.add("Three");
    System.out.println ("Before Sorting");
    for (String s : nameList) System.out.println(s);
    Collections.sort (nameList);
    System.out.println ("\nAfter Sorting");
    for (String s : nameList) System.out.println(s);
  }
}
```

Listing 12-12: Illustration of `Collections.sort` method.

When the above program is executed, you will see the following output:

```
Before Sorting
One
Two
Three

After Sorting
One
Three
Two
```

By default the `ArrayList` is sorted in the Lexicographic order. Lists consisting of different types of objects will be sorted in different order. The default sorting order for lists containing various types of objects is shown in Table 12-4.

All these classes implement the `Comparable` interface, which provides a natural ordering for a class.

```
public interface Comparable<T> {
 public int comparedTo(T o);
}
```

Table 12-4: Default ordering used by `Collections.sort` algorithm.

Class	Default ordering
`Byte`	Signed Numeric
`Character`	Unsigned Numeric
`Long`	Signed Numeric
`Integer`	Signed Numeric
`Short`	Signed Numeric
`Double`	Signed Numeric
`Float`	Signed Numeric
`BigInteger`	Signed Numeric
`BigDecimal`	Signed Numeric
`Boolean`	Boolean.FALSE < Boolean.TRUE
`String`	Lexicographic

12.5.2 Comparator Interface

If you want to sort a list in an order other than the natural order, you will need to provide a `Comparator`. A comparator is a class that implements the Comparator interface shown below:

```
public interface Comparator<T> {
  public int compare(T o1, T o2);
  public Boolean equals(Object o);
}
```

For example, if you want to sort the `ArrayList` called `namesList` in the descending order, you will write a `Comparator`. The compare method in the `Comparator` class will return objects in descending order as shown in Listing 12-13:

```
import java.util.*;
class DescendComparator implements Comparator<String> {
  public int compare(String s1, String s2) {
    return (s2.compareTo(s1));
  }
}
public class SortDescendApp {
  public static void main(String[] args){
    ArrayList<String> nameList = new ArrayList<String>();
    DescendComparator sComp = new DescendComparator();

    nameList.add("One");
    nameList.add("Two");
    nameList.add("Three");
```

```
        System.out.println ("Before Sorting");
        for (String s : nameList)
          System.out.println(s);
        Collections.sort (nameList, sComp);
        System.out.println ("\nAfter Sorting");
        for (String s : nameList)
          System.out.println(s);
  }
}
```

Listing 12-13: Illustration of `Comparator` class.

When the above program is executed, the following output will be produced:

```
Before Sorting
One
Two
Three

After Sorting
Two
Three
One
```

12.6 Searching Collections

In this section, you will learn different ways of searching a particular element in a Collection such as `ArrayList`.

12.6.1 `indexOf` and `contains` Methods

The `indexOf` method of an `ArrayList` returns the position (an integer) in which a particular object is stored. Consider the unsorted `arrayList` called `nameList` as shown in Listing 12-13. The following statement will return `1`.

```
int pos = nameList.indexOf("Two");
```

Note that the index starts from `0`. The `indexOf` method returns the first occurrence of the object passed as the parameter. If the object is not found the method returns `-1`.

You can use the `contains` method to find out if the object exists in the List. If the method is invoked on `nameList`, the return value will be true.

```
boolean doesExist = nameList.contains("Two");
```

12.6.2 `binarySearch` Method

A list that is sorted in the ascending natural order can be searched using this method. If the `binarySearch` method is invoked on the sorted `nameList arrayList` shown in Listing 12-13, the return value will be `2`.

```
Collections.sort(nameList);
int pos = Collections.binarySearch(nameList, "Three");
```

This method returns the position (starts from `0` index) if the object is found; otherwise it returns a negative number specified by the formula (insertion point –1). For example, if you search the `String` "Seven" in `nameList`, it will return `-2`, as the insertion point is `1`.

```
int pos = Collections.binarySearch(nameList, "Seven");
```

If you write your own `Comparator`, the `binarySearch` method can be used to search lists that are sorted in the order as specified by your `Comparator`. The method will require one more parameter; that is, your `Comparator`.

```
DescendComparator sComp = new DescendComparator();
Collections.sort (nameList, sComp);
int pos = Collections.binarySearch(nameList, "Two", sComp);
```

Using the `binarySearch` method is more efficient that using the `indexOf` or contains methods of the `List` interface, as it works on a sorted list and uses an efficient searching algorithm, that is, binary search algorithm.

12.7 Summary

In this chapter, we discussed:

- The concept of generics (i.e., parameterized types), which helps us to eliminates run-time-type identification errors. Using Java Generics, we can encapsulate algorithms and concepts in type independent methods and classes
- The Java Collections Framework consists of many predefined classes that encapsulate data structures that can hold a collection of elements.
- The use of three types of Collection classes, namely; `ArrayList`, `HashSet`, and `HashMap`. We have seen the use of these generic collection classes in manipulating a collection of same type of objects.
- The sorting and searching methods found in the Collections classes.

12.8 Exercises

1. Define an application called `SimpleBookApp` which stores distinct ISBN numbers (e.g., `Integer`) of a few books in an `ArrayList` and lists them in the ascending order.
2. Define a class called Book. This class should store attributes such as the title, ISBN number, author, edition, publisher and year of publication. Provide get/set methods in this class, to access these attributes. Define a

class called `BookApp`, which contains the main method. This class should create a few book objects with distinct names and store them in an `ArrayList`. This class should then list the name of all books in the `ArrayList`.

3. Enhance the program in Exercise 2 to sort the `ArrayList` in the ascending order of the year of publication.
   Hint: You will need to define a comparator class that takes two Book objects as parameters to the compareTo method. This method should return a boolean value after comparing the year of publication of the two book objects.
4. In the program in Exercise 2, create a few more Book objects with same names but a different edition, ISBN, and year of publication. Add these new `Book` objects in the `ArrayList` and display the book list sorted by name of the book and for duplicate names of books, sorted by year of publication.
   Hint: You will need to define a comparator class that takes two Book objects as parameters to the `compareTo` method. This method should do a two-step comparison. The first comparison should compare the name of the book. If the name is the same, the second comparison should compare the year of publication.
5. Define a class called `Account` that contains attributes such as account number, name of the account holder, identification number of the account holder, and year of account opening. Provide get/set methods in this class to access the attributes. Define a class called `AccountApp`, which contains the main method. This class should create a few account objects with distinct account numbers and store it in a `HashMap`. The program should then ask for the account number as the input from the console and display the `Account` information pertaining to that account number. *Hint:* The account number is the key for the HashMap.
6. Define a class called `Student` with three attributes, viz. name, program, and year. Define a class called `StudentApp`, which contains a main method. Create several `Student` objects with different names, programs (e.g., Bio-Informatics, Computer Science), and year (e.g., 2006, 2007). Add some students in one `HashSet` and some students in another `HashSet`. Display the union, intersection and difference of two sets. Display also if one set is a subset of another. *Hint:* Try adding different combination of students in the two sets, so that you can print some student details in deriving the Union, Intersections, and Difference.

13

Graphical Interfaces and Windows

In the earlier chapters, we previewed the Java API for input and output mechanisms, networking, and multithreading. In this chapter, we proceed to look at the facilities for incorporating graphical user interfaces. With the availability of powerful and cheap hardware, and widespread and diverse use of computers, easy and intuitive interfaces have become an important aspect to developers.

To reduce software costs, developers must be able to easily create and modify code that implements these interfaces. The Abstract Windowing Toolkit (AWT) in Java provides a simple yet flexible object-oriented model for building applications that use graphical user interfaces.

13.1 The AWT Model

Consistent with the principles of abstraction, the AWT model for graphical user-interfaces in Java is broken into several constituents with its own concerns and functionality. Briefly, AWT constituents include:

- frames;
- components;
- panels;
- layout managers;
- events.

The frame abstraction allows for independent windows in the Java host. While a Java application typically runs in a window, it may also create more graphical windows to provide alternative or complementary views.

The AWT components consist of a custom set of widgets for various styles of user interaction. The range of widgets includes:

- text labels;
- buttons;
- choice and list selections;
- scrollbars;
- text fields and editing areas;
- canvas painting areas.

The AWT panels are used to contain a set of logically related AWT components. For example, user authentication by a user name and password may be presented by two text field components together with two buttons. One button allows the user to proceed with authentication, while the other cancels the request. These components may be logically included in an AWT panel.

The layout manager constituent of the AWT model allows for layout control of components within panels.

Panels are also components, and thus the containment relationship is hierarchical. Where there are two logical groups of components, these can be placed and arranged in two separate panels. These panels are in turn placed into the parent panel, and may be arranged collectively.

Finally, most implementations of graphical user interfaces typically adopt an event-based approach over polling. It is tedious to anticipate and poll for every input that a user can make: mouse clicks, keyboard input, audio command/feedback, and so on. It is more difficult to determine when subcombinations of these are legitimate.

The event-based model allows for suitable code to be associated with significant events, and to be invoked implicitly. We will cover the containment model used in JDK 1.0, as well as the delegation model in JDK 1.1 (and later versions), which is based on Java Beans.

The object-oriented features in Java allow the AWT library to be modular for maintainability. This is evident from the constituents described earlier. Yet they must ultimately be integrated to allow the intended user interaction at run-time. In addition, while basic behavior is in-built for easy usage, it is also highly configurable and many combinations with custom-built components are easily achieved. The implementation highly relies on Java features such as inheritance and polymorphism.

13.2 Basic AWT Constituents

We will first discuss the AWT through incremental incorporation of its constituents. Following that, we will incorporate event handlers to complete the functionality of an interface by binding it to application level code.

As with other abstractions in Java, the functionality of the AWT constituents are encoded in class definitions, and ultimately its usage manifested in object instances.

13.2.1 Frames

An application window is created by instantiating a `Frame` object. A program that merely creates a `Frame` object is shown in Listing 13-1. Since a `Frame` object is initially invisible, the `setVisible(true)` method is used to bring it to the top of the desktop.

```
import java.awt.Frame;

class ExampleFrame {
  public static void main(String arg[]) {
    Frame f = new Frame("Example");
    f.setVisible(true);
  }
}
```

Listing 13-1: `ExampleFrame` class.

Two observations deserve comment when the class `ExampleFrame` is executed. The `Frame` instance is considered empty because it does not contain any components. As such, there is no view of interest, and the corresponding window is displayed with its title but no viewable area, as in Figure 13-1.



Figure 13-1: Empty frame.

While the window may be minimized and maximized, it cannot be closed as the expected custom event-handler has yet to be installed. We will return to this issue in subsequent sections. For the moment, the new window is destroyed by aborting the execution of the `ExampleFrame` class.

Besides creating a window by instantiating a `Frame` object, we may also define a new class by inheriting from the `Frame` class. This is the means for defining a new Frame-like abstraction, but with new default settings or specific behavior. The class skeleton in Figure 13-2 with a default size of 150 × 100 pixel window is one such example.

```
import java.awt.Frame;

class ExampleFrame2 extends Frame {
  ExampleFrame2(String m) {
    super("Example2: "+m);
    setSize(150,100);
    setVisible(true);
  }
  // new functionality
  public static void main(String arg[]) {
    new ExampleFrame2("Subclassing");
  }
}
```

Listing 13-2: `ExampleFrame2` class.



Figure 13-2: Viewable empty frame.

13.2.2 Components

The `Component` class defines the generic behavior of GUI components that may appear in a frame. Specific components, such as those mentioned in the previous section, are defined as subclasses of this generic `Component` class.

The insertion of a component into a `Frame` instance proceeds in two steps: first, an instance of the component is created with the appropriate parameters to the constructor method. Subsequently, it is inserted into the `Frame` via the `add()` method, with signature:

```
void add(Component comp)
```

Thus, any subclass instance of Component may be added into our `Frame`. We proceed by adding various components to our currently empty `Frame`. The code in Listing 13-3 is similar to `ExampleFrame2` except a `Checkbox` component has been added.

```
import java.awt.*;

class ExampleFrame3 extends Frame {
  ExampleFrame3(String m) {
    super("Example3: "+m);
    setSize(100,150) ;
    add(new Checkbox("Save settings"));
    setVisible(true);
  }
  public static void main(String arg[]) {
    new ExampleFrame3("Checkbox");
  }
}
```

Listing 13-3: `ExampleFrame3` class.



Figure 13-3: Frame with Checkbox Component.

As before, no event handler has yet been installed. While the `Checkbox` provides visual feedback as to its state in response to mouse clicks, there is no code to interrogate that state, nor is any code executed as a result of state changes.

A `Button` component may be similarly placed as shown in Listing 13-4.

```
import java.awt.*;

class ExampleFrame4 extends Frame {
  ExampleFrame4(String m) {
    super("Example4: "+m);
    setSize(100, 150) ;
    add(new Button("Save"));
    setVisible(true);
  }
  public static void main(String arg[]) {
    new ExampleFrame4("Button");
  }
}
```

Listing 13-4: `ExampleFrame4` class.

Since the Frame may currently only contain a single component, the `Button` component has used the full viewable area for display, and thus looks oversized, as in Figure 13-4. If this is unsightly in appearance, the consolation is that it is improved with Panels. Components will be discussed further in the next section.



Figure 13-4: Frame with `Button` component.

13.2.3 Panels

Panels may be succinctly described as components holding other components. As a component, it may be placed in a `Frame` via `add()` as already seen with `Checkbox` and `Button`. As a container, it will also accept constituent components via its own `add()` method.

This is illustrated in Listing 13-5, with the corresponding view in Figure 13-5.

```
import java.awt.*;

class ExamplePanel extends Frame {
  ExamplePanel(String m) {
    super("ExamplePanel: "+m);
    setSize(100,150) ;
    Panel p = new Panel();                  // create Panel
    add(p) ;                                 // add Panel into Frame
    p.add(new Button ("Save"));              // add Button into Panel
    p.add(new Checkbox ("Save settings"));  // add Checkbox into Panel
    setVisible(true);
  }
  public static void main(String arg[]) {
    new ExamplePanel("Inserting Components");
  }
}
```

Listing 13-5: `ExamplePanel` class.



Figure 13-5: Frame with Panel.

13.2.4 Layout in Panels

The layout of components in Panels is the responsibility of Layout Managers. The default layout used by Panels is known as `Flow Layout`. In this layout scheme, components are automatically placed in a left-to-right manner, and down to the next row if the right margin is exceeded. Formatting components within a row may be centered (default), left- or right-justified.

Thus, resizing the `Frame` for more column space will bring the two components (`Button` and `Checkbox`) into the same row, as seen in Figure 13-6.



Figure 13-6: Frame with Stretched Panel.

Two other layout schemes are commonly used: The `GridLayout` class implements a table-style layout with fixed rows and columns. Sample code is shown in Listing 13-6, with results visible in Figure 13-7. It is a simple and useful layout scheme when components in a `Panel` have fairly equal dimensions.

```
import java.awt.*;

class ExampleGridLayout extends Frame {
  ExampleGridLayout(String m) {
    super("ExampleLayout: "+m);
    setSize(240,80);
    Panel p = new Panel();
    add(p) ;
    p.setLayout(new GridLayout(3,2)) ;  // use a 3x2 grid
    p.add(new Checkbox ("Save config"));
    p.add(new Button ("Save"));
    p.add(new Checkbox("Save changes"));
    p.add(new Button("Abort & do not save"));
    p.add(new Checkbox("Ignore colors"));
    p.add(new Button("Quit"));
    setVisible(true);
  }
  public static void main(String arg[]) {
    new ExampleGridLayout("GridLayout");
  }
}
```

Listing 13-6: `ExampleGridLayout` class.

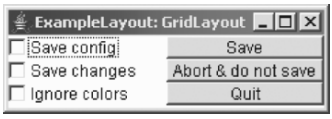


Figure 13-7: Panel with `GridLayout`.

The `BorderLayout` class allows for even placement of components near borders of containers. The top, left, right, and bottom localities of a `Panel` using this layout scheme are denoted as "North," "West," "East," and "South," as seen in Listing 13-7 and corresponding view in Figure 13-8. This scheme allows for convenient component placement without concern as to absolute coordinates or sizes.

```
import java.awt.*;

class ExampleBorderLayout extends Frame {
  ExampleBorderLayout(String m) {
    super("ExampleLayout2: "+m);
    setSize(340,280) ;
    Panel p = new Panel();
    p.setLayout(new BorderLayout());
    p.add("North", new TextArea(8,40));
    p.add("West", new Checkbox("Save config"));
    p.add("East", new Checkbox("Ignore colors"));
    p.add("South", new Button ("Exit"));
    add(p);
    setVisible(true);
  }
  public static void main(String arg[]) {
    new ExampleBorderLayout("BorderLayout");
  }
}
```

Listing 13-7: `ExampleBorderLayout` class.

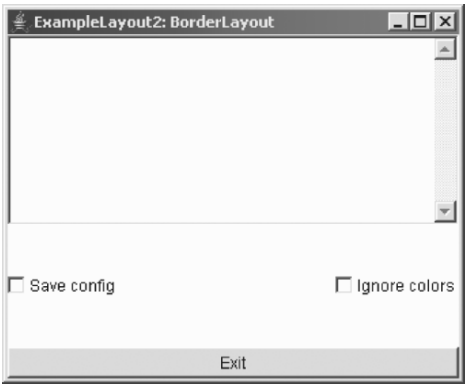


Figure 13-8: Panel with `BorderLayout`.

As in earlier uses of `Button` components, the `Exit` button here is stretched too wide. Placing Button within a Panel first, and then the Panel into the outer panel helps, as seen in Figure 13-9.

```
Panel p = new Panel();
p.setLayout(new BorderLayout());
p.add("North", new TextArea());
p.add("West", new Checkbox("Save config"));
p.add("East", new Checkbox("Ignore colors"));
Panel q = new Panel();
q.add(new Button("Exit"));
p.add("South", q);
add(p);
```

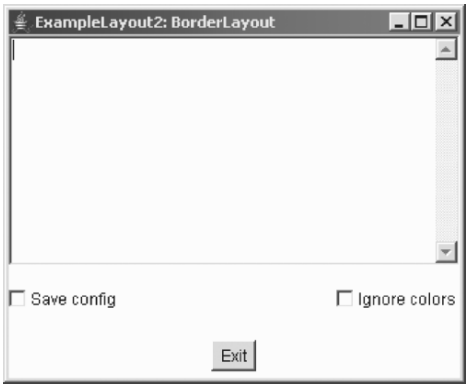


Figure 13-9: Panel with `BorderLayout` and nested Panel.

13.2.5 Events

Currently, the views painted by the application frame with its constituent components are like empty shells with no application processing logic underneath. In a typical scenario, processing logic would involve:

- retrieving the state of switches, such as to determine the states of the "Save config" and "Ignore colors" `Checkbox`es, or reading input keyed into the top `TextArea`; or
- reacting to events, such as the click of the `Exit` Button.

The API documentation for the `Checkbox` class shows that the `getState()` method returns the state of a `Checkbox` object:

```
boolean getState();
```

Similarly, the API documentation for the `TextComponent` class shows that the `getText()` method returns the input text of both `TextInput` and `TextField` objects (`TextComponent` being the superclass).

```
String getText();
```

Event handling is more complex due to the relationships between various handlers. Instead of polling, frames and components wait until events occur, and appropriately defined code within methods are invoked. The `handleEvent()` method is most significant as it is the main method which is invoked when an event occurs.

The list of events is documented in the API documentation for the `Event` class. These events may be divided into: action, selection, mouse, keyboard, focus, scroll and window events. Due to the common handling of the first five types of events, `handleEvent()` has channeled the necessary processing to the following handler methods:

`action()`	The `action()` method handles events that request some action by the user, e.g., `Button`, `Checkbox`, `List`, `Choice` or `TextField` components.
`mouseEnter()`, `mouseExit()`, `mouseMove()`, `mouseDrag()`, `mouseDown()`, `mouseUp()`	The mouse methods relate to mouse activities—moving a mouse in/out of a component, moving the mouse with/without its button depressed, and mouse button clicks.
`keyDown()`, `keyUp()`	Like mouse events, a keyboard activity also corresponds to two (push/release) events.
`gotFocus()`, `lostFocus()`	Keyboard input for a component in a frame is dictated by whether it has the *keyboard focus*.

We now demonstrate how events may be handled. We will elaborate in subsequent sections.

Because external events ultimately invoke the above methods, custom event handlers may be intuitively installed for components via a new class and inheriting from an existing component. In handling the `Exit` Button in the previous example, we might use the `MyExitButton` class which inherits from `Button`, as seen in Listing 13-8.

```
class MyExitButton extends Button {
  MyExitButton(String label) {
    super(label);
  }
  public boolean action(Event e, Object  what) {
    System.exit(0);
    return true;
  }
}
```

Listing 13-8: New button with event handler.

The next example in Listing 13-9 has a responsive `Exit` button because `new MyExitButton()` is used instead of `new Button()`.

```
import java.awt.*;

class ExampleEvent extends Frame {
  ExampleEvent(String m) {
    super("ExampleButton: "+m);
    setSize(340,280) ;
    Panel p = new Panel();
    p.setLayout(new BorderLayout());
    p.add("North", new TextArea());
    p.add("West", new Checkbox("Save config"));
    p.add("East", new Checkbox("Ignore colors"));
    Panel q = new Panel();
    q.add(new MyExitButton("Exit"));
    p.add("South", q);
    add(p);
    setVisible(true);
  }
  publicstatic void main(String arg[]) {
    new ExampleEvent("Event");
  }
}
```

Listing 13-9: ExampleEvent class

While this method for installing custom event handlers work, it is tedious in that each component requires a new class definition to accommodate appropriate customization. We would thus require two additional class definitions for the two `Checkbox` components. Fortunately, the following two design characteristics of components provide for another means of installing event handlers:

- Just as properties are inherited down the inheritance chain, event handlers which are methods are also inherited and are installed for subclass unless redefined.

- The default event handler for an AWT component passes an uncaught event to its parent container.

In our earlier examples, any uncaught events for standard components were diverted to the containing panel and ultimately the frame that contains them. However, no effort was made to catch them. The next example is mostly unchanged from the previous, except for the label of the `Button` component and two top-level event handler methods, `handleEvent()` and `action()`, to capture events from nested components. (See Listing 13.10.)

```
import java.awt.*;

class Example extends Frame {
  TextArea txt;
  public static void main(String arg[]) {
    new Example("Event Handling");
  }
  Example(String m) {
    super("Example: "+m);
    setSize(340,280) ;
    Panel p = new Panel();
    p.setLayout(new BorderLayout());
    p.add("North", txt = new TextArea());
    p.add("West", new Checkbox("Save config"));
    p.add("East", new Checkbox("Ignore colors"));
    Panel q = new Panel();
    q.add(new Button("Clear"));
    p.add("South", q);
    add(p);
    setVisible(true);
  }
  public boolean action(Event e, Object target) {
    if (e.target instanceof Button) {
      txt.setText("");
      return true;
    } else if (e.target instanceof Checkbox) {
      Checkbox x = (Checkbox) e.target;
      txt.appendText(x.getLabel()+
                     (x.getState() ? " is on\n" : " is off\n"));
      return true;
    }
    return(super.action(e, target));
  }
  public boolean handleEvent(Event evt) {
    if (evt.id == Event.WINDOW_DESTROY)
      System.exit(0) ;
    return(super.handleEvent(evt));
  }
}
```

Listing 13-10: Event Handling in the Containment model

The `WINDOW_DESTROY` event is sent following a request to close a frame. Unlike the common events discussed above, window events are not distributed to other more specialized methods, and we handle this in `handleEvent()`. We have redefined

`handleEvent()` to be on a lookout for the `WINDOW_DESTROY` event, but at the same time, rely on the original event handler for normal processing using the super self-reference.

The `action()` method handles the action group of events. It is invoked when there is a request to perform an action (e.g., `Button` click or change of `Checkbox` state). Here, we check the target component where an event was initially directed to, and provide the necessary code to handle it. As an example skeleton code, we substitute code to write to a `TextArea` component via the `setText()` and `appendText()` methods.

This section has provided an overview of the AWT package in JDK 1.0. While it may co-exist with the new event-handling model in JDK 1.1 (and later versions), it is nevertheless discouraged. We thus take a quick look at the delegation-based event-handling model.

13.2.6 Events in JDK 1.1 (and later versions)

The containment model in JDK 1.0 has the advantage of requiring minimal effort in understanding and use. However, the associated event handlers tend to be bulky in deciphering which events are relevant and, as a result, becomes untidy in larger applications.

Event handling in JDK 1.1 adopts the delegation model and consists of *event sources* and *event listeners*. One or more event listeners may register to be notified of particular events with a source. Any object may be a listener by implementing the appropriate `EventListener` interface. Just as in JDK 1.0, where there are various groups of events, there are also various groups of `EventListener`s in JDK 1.1.

As any object may be delegated the job of event handling, a direct connection may be established between event sources and listeners. Other objects may then be oblivious to this event handling activity.

The following code fragment Listing 13-11 uses components as in previous examples, but using JDK 1.1–styled event handling. Corresponding to `Button` and `Checkbox` components and the use of `Frame`s, there are three groups of listeners required. We need `EventListener`s for action, item, and window events. The corresponding listeners are `ActionListener`, `ItemListener`, and `WindowListener`, respectively.

```
import java.awt.*;
import java.awt.event.*;

class Example1 extends Frame
            implements ActionListener, ItemListener, WindowListener {
  Example1(String m) {
    super("Example1: "+m);
    Button b; Checkbox c;
    setSize(240,140);
    Panel p = new Panel();
    p.add(new Label("Conference Registration"));
    p.add(c = new Checkbox("Attend Tutorial")); c.addItemListener(this);
    p.add(c = new Checkbox("Require Hotel")); c.addItemListener(this);
    p.add(b = new Button("Reset")); b.addActionListener(this);
    p.add(b = new Button("Submit")); b.addActionListener(this);
```

```
    addWindowListener(this);
    add(p);
    setVisible(true);
  }
  public static void main(String arg[]) {
    new Example1("Event Handling");
  }
  public void actionPerformed(ActionEvent evt) {
    System.out.println("actionPerformed: " + evt.getSource().toString());
  }
  public void itemStateChanged(ItemEvent itm) {
    System.out.println("itemStateChanged: " + itm.getItemSelectable());
  }
  public void windowActivated(WindowEvent we) { }
  public void windowClosed(WindowEvent we)  { }
  public void windowClosing(WindowEvent we) { System.exit(0) ; }
  public void windowDeactivated(WindowEvent we) { }
  public void windowDeiconified(WindowEvent we) { }
  public void windowIconified(WindowEvent we) { }
  public void windowOpened(WindowEvent we) { }
}
```

Listing 13-11: JDK 1.1–style event handling using the Delegation model.



Figure 13-10: Responding to events.

In this example, the `Frame` subclass instance also implements the listener interfaces and, as such, acts as a listener for all those events. As such, our example performs two generic actions:

- It registers with the event source that it will handle events.
- It implements the necessary interfaces for a handler.

For the three sources of events, the example application performs the following concrete tasks:

- It registers with the `Button` components that it will handle action events (via `addActionListener()`), makes itself a listener by implementing the `ActionListener` interface, and defines `actionPerformed()`.
- It registers with the `Checkbox` components that it will handle action events (via `addItemListener()`), makes itself a listener by implementing `ItemListener` interface, and defines `itemStateChanged()`.

- It registers with the `Frame` that it will handle action events (via `addWindowListener()`), makes itself a listener by implementing `WindowListener` interface, and defines `windowActivated()`, `windowClosed()`, `windowClosing()`, `windowDeactivated()`, `windowDeiconified()`, `windowIconified()` and `windowOpened()`. (In this case, event source and listener is the same object.)

Judging from the resultant code, the reader may conclude that the delegation model is more complex due to increased code length. This is somewhat true, but this model provides for additional flexibility.

- Our example looks cluttered because it contains all the handlers. However, the delegation model allows for `ActionListener` and `ItemListener` to be distinct objects.
- The other complaint is that a `WindowListener` needs to define seven methods even when six of them have null bodies. This drawback is easily solved by using Adapters. Each `EventListener` interface that has more than one abstract method has a corresponding Adapter class which implements the interface using standard event handlers. For example, compare the API documentation for the `MouseListener` and `MouseAdapter` classes. Subsequently, we may instantiate new custom Adapters by inheriting from standard Adapters and redefining the necessary methods, instead of implementing all methods necessary in interfaces.

Adaptor classes are further discussed in Section 13.5 with other kinds of class definitions.

13.3 Basic Components

Having seen an overview of creating graphical user interfaces using basic AWT components in the preview section, we now proceed to study each component more closely. For each component, we will see:

- how it may be created and added to a Panel;
- its corresponding visual layout; and
- how appropriate `EventListener`s may be created and installed.

13.3.1 Label Component

The `Label` component allows for the display of fixed text strings in a container such as a Panel.

```
Panel p = new Panel();
p.add(new Label("WEB Search"));
```

13.3.2 Button Component

The `Button` component allows for clickable buttons in a container.

```
Button b;
Panel p = new Panel();
p.add(b = new Button("Submit query"));
b.addActionListener(actionListener);
```

When clicked, a `Button` instance invokes the `actionPerformed()` method of the `actionListener` object that is registered via `addActionListener()`.

13.3.3 Checkbox Component

The `Checkbox` component allows for a boolean choice in a container.

```
Checkbox c;
Panel p = new Panel();
p.add(c = new Checkbox("Quick query"));
c.addItemListener(itemListener);
```

Corresponding to a modified state, a `Checkbox` instance invokes the `itemStateChanged()` method of the `ItemListener` object that was registered via `addItemListener()`.

13.3.4 `CheckboxGroup` Component

The `CheckboxGroup` component allows for a group of `Checkbox` components where only one of them may be on.

```
Checkbox c;
CheckboxGroup cbg = new CheckboxGroup();
Panel p = new Panel();
p.setLayout(new GridLayout(3, 1));
p.add(c = new Checkbox("Birthday", cbg, true));
c.addItemListener(itemListener);
p.add(c = new Checkbox("Engagement", cbg, false));
c.addItemListener(itemListener);
p.add(c = new Checkbox("Wedding", cbg, false));
c.addItemListener(itemListener);
```

Within a `CheckboxGroup`, a `Checkbox` object on turning true invokes the `itemStateChanged()` method of the `ItemListener` object that was registered via `addItemListener()`. The previous `Checkbox` in the group is implicitly turned false.

13.3.5 `TextArea` Component

The `TextArea` component allows for the display and manipulation of text. Its placement within a `Panel` is similar to other components we have seen so far. In the following example code, contents are placed into a `TextArea` via `setText()` and `append()`, and after user manipulation, may be retrieved via `getText()`.

```
import java.awt.*;
import java.awt.event.*;

class MixedComponents extends Frame
              implements ActionListener, ItemListener, WindowListener {
  TextArea txt;
  MixedComponents(String m) {
    super("Mixed Components: "+m);
    Button b; Checkbox c;
    setSize(255,420);
    Panel p = new Panel();
    p.add(new Label("Conference Registration"));
    Panel sub = new Panel(); sub.setLayout(new GridLayout(2,1));
    sub.add(c = new Checkbox("Attend Tutorial"));
      c.addItemListener(this);
    sub.add(c = new Checkbox("Require Hotel"));
      c.addItemListener(this);
    p.add(sub);
    sub = new Panel(); sub.setLayout(new GridLayout(3,1));
    CheckboxGroup g = new CheckboxGroup();
    sub.add(c = new Checkbox("no food restrictions", g, true));
      c.addItemListener(this) ;
    sub.add(c = new Checkbox("no seafood", g, false));
      c.addItemListener(this);
    sub.add(c = new Checkbox("vegetarian food", g, false));
      c.addItemListener(this);
    p.add(sub);
    p.add(txt = new TextArea(10,30));
    p.add(b = new Button("Reset")); b.addActionListener(this) ;
    p.add(b = new Button("Submit")); b.addActionListener(this);
    addWindowListener(this) ;
    add(p) ;
    txt.setText("Events:\n");
    setVisible(true);
  }
  public static void main(String arg[]) {
    new MixedComponents("Event Handling");
  }
  public void actionPerformed(ActionEvent evt) {
    txt.append("actionPerformed: " + evt.getSource().toString() + "\n");
  }
  public void itemStateChanged(ItemEvent itm) {
    txt.append("itemStateChanged: " + itm.getItemSelectable() + "\n");
  }
  public void windowActivated(WindowEvent we) { }
  public void windowClosed(WindowEvent we)   { }
  public void windowClosing(WindowEvent we)  {
    System.out.println(txt.getText());
    System.exit(0) ;
  }
```

```
    public void windowDeactivated(WindowEvent we)   { }
    public void windowDeiconified(WindowEvent we)   { }
    public void windowIconified(WindowEvent we)   { }
    public void windowOpened(WindowEvent we)   { }
}
```

Listing 13-12: `MixedComponents` class with various handlers.

The code in Listing 13-13 shows a slightly cluttered `Frame`, with subgroups of components in a nested `Panel` for the benefit of a neat layout in Figure 13-11. Events are logged into the `TextArea`. The `TextField` component may be used in place of `TextArea` when a one-line display suffices.



Figure 13-11: Testing event handling.

13.3.6 Choice Component

The `Choice` component allows for the selection of items via a pop-up menu. Selectable items are initially specified via the `add()` method. The current selection is highlighted, and may be interrogated via `getSelectedItem()`.

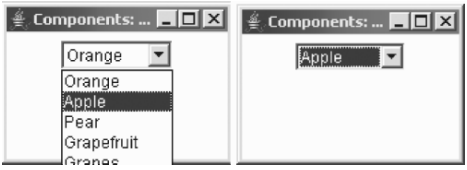


Figure 13-12: Selecting from a choice component.

```
import java.awt.*;
import java.awt.event.*;

class ChoiceExample extends Frame
                    implements ItemListener, WindowListener {
   Choice c;
   ChoiceExample(String m) {
      super("Components: "+m);
      setSize(170,130);
      Panel p = new Panel();
      p.add(c = new Choice()); c.addItemListener(this) ;
      c.add("Orange");
      c.add("Apple");
      c.add("Pear");
      c.add("Grapefruit");
      c.add("Grapes");
      c.add("Jackfruit");
      c.add("Apricot");
      add(p) ;
      addWindowListener(this);
      setVisible(true);
   }
   public void itemStateChanged(ItemEvent itm) {
      System.out.println("itemStateChanged:" + itm.getItemSelectable());
   }
   public static void main(String arg[]) {
      new ChoiceExample("Event Handling");
   }
   public void windowClosing(WindowEvent we) {
      System.out.println(c.getSelectedItem());
      System.exit(0) ;
   }
   // other methods as before ...
}
```

Listing 13-13: Using a choice component.

13.3.7 List Component

The `List` component in Figure 13-13 allows for the selection of items via a scrolling list. Clicking on an unselected item selects it, and vice versa. As with the `Choice` component, its setup and interrogation are via the `add()` and `getSelectedItem()` methods, respectively.

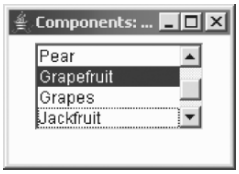


Figure 13-13: Selecting from a List Component.

The `List` component also allows for multiple selections. In Figure 13-14 this feature may be enabled via the `setMultipleMode()` method, correspondingly

illustrated in Listing 13-14. The `getSelectedItems()` method returns a list of selected strings.

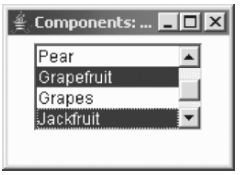


Figure 13-14: Multiple selections from a List Component.

```
import java.awt.*;
import java.awt.event.*;

class ListExample extends Frame implements WindowListener {
   ListExample(String m) {
      super("Components: "+m);
      setSize(170,130) ;
      List l;
      Panel p = new Panel();
      p.add(l = new List());
      l.setMultipleMode(true); // allow for multiple selections
      l.add("Orange");      l.add("Apple");
      l.add("Pear");        l.add("Grapefruit");
      l.add("Grapes");      l.add("Jackfruit");
      l.add("Apricot");
      add(p);
      addWindowListener(this);
      setVisible(true);
   }
   public static void main(String arg[]) {
      new ListExample("Event Handling");
   }
   public void windowClosing(WindowEvent we)  {
      String x[] = l.getSelectedItems();
      for (int i=0; i<x.length; i++) System.out.println(x[i]);
      System.exit(0);
   }
   // other methods as before ...
}
```

Listing 13-14: Using a List Component.

In making multiple selections, we are never sure whether there are other selections to be made. In this case, it is probably less useful to implement an `ItemListener`. Instead, a simpler model might rely on an external gesture (e.g., clicking another `Button` component) to trigger the end of selections.

13.3.8 Menus and Menu Items

The `Frame` that implements the main window of an application typically includes a menu bar with various menus and menu items. A typical scenario is seen in Figure 13-15. As with other AWT components, the way in which menu structures are specified,

and how events are handled when menu items are chosen are similar—appropriate object instances are created and appropriately placed within components.

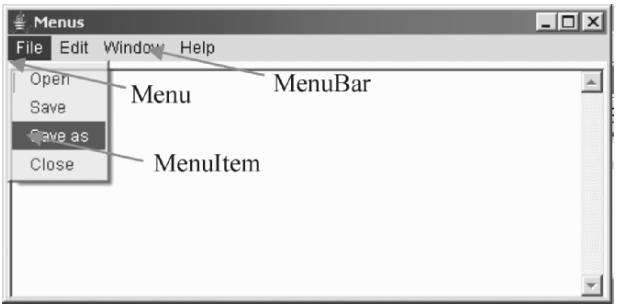


Figure 13-15: Frame with Menu Bar.

In this case, the three classes of objects used are `MenuBar`, `Menu`, and `MenuItem`. When seen hierarchically, each menu item is represented by a `MenuItem` instance, and all items in the group are collected via the `add()` method for a menu object. Similarly, menu instances are collected into the menu bar via the `add()` method for a menu bar object. Finally, the single menu bar object is added to the Frame via the `setMenuBar()` method. This is shown in the code fragment in Listing 13-15: Running the `MenuBar` example produces the frame in Figure 13-15.

```
import java.awt.*;
import java.awt.event.*;

class MenuExample extends Frame {
   MenuExample() {
      super("Menus");
      setSize(450,220) ;
      MenuBar mb;
      Menu m;

      mb = new MenuBar();
      setMenuBar(mb) ;
      mb.add(m = new Menu("File"));
      m.add(new MenuItem("Open"));
      m.add(new MenuItem("Save"));
      m.add(new MenuItem("Save as"));
      m.add(new MenuItem("Close"));
      mb.add(m = new Menu("Edit"));
      m.add(new MenuItem("Undo"));
      m.add(new MenuItem("Cut"));
      m.add(new MenuItem("Copy"));
      m.add(new MenuItem("Paste"));
      mb.add(m = new Menu("Window"));
      m.add(new MenuItem("New Window"));
      m.add(new MenuItem("Arrange"));
      m.add(new MenuItem("Split"));
      mb.add(m = new Menu("Help"));
      m.add(new MenuItem("Index"));
```

```
        m.add(new MenuItem("Wizard"));
        m.add(new MenuItem("About"));

        Panel p = new Panel();
        p.add(new TextArea());
        add(p);
    }
    // other methods for event handling ...
}
```

Listing 13-15: Working with Menu Bar.

As with `Button` components, event handling for menu items is effected via a suitable `ActionListener` by using the `addActionListener()` method. For brevity, only a skeletal structure for the modified class definition is shown in Listing 13-16.

```
class MenuExample extends Frame
                  implements ActionListener, WindowListener {
   MenuExample() {
      // ... code as before
      MenuItem i;
      mb.add(m = new Menu("File"));
      m.add(i = new MenuItem("Open")); i.addActionListener(this) ;
      m.add(i = new MenuItem("Save")); i.addActionListener(this);
      m.add(i = new MenuItem("Save as")); i.addActionListener(this);
      m.add(i = new MenuItem("Close")); i.addActionListener(this);
      // ... other Menu setup code – similar to above
      addWindowListener(this);
      setVisible(true);
   }
   public static void main(String arg[]) {
      new MenuExample();
   }
   public void actionPerformed(ActionEvent evt) {
      System.out.println("menuItem: " + evt.getSource().toString());
   }
   public void windowClosing(WindowEvent we) { System.exit(0); }
}
```

Listing 13-16: Handling menu events.

13.3.9 Dialog Frames

The `Dialog` class allows for a dialog window where input is expected from the user. By adding a Label component in a new class, it can be used to deliver a message, as seen in Figure 13-16.



Figure 13-16: Dialog Frame.

A simple implementation of a `Message` frame may inherit from the basic `Dialog` class, as illustrated in Listing 13-17. The parent parameter in the constructor is passed to the constructor of the superclass. It allows focus to return to the calling parent when the `Dialog` window is closed. As with other windows, the `setVisibletrue` method makes the frame visible while the `dispose()` method destroys it and reclaims resources to represent it.

```
class Message extends Dialog implements WindowListener {
   Message(Frame parent, String message) {
      super(parent, "Message");
      setSize(400,100) ;
      setLayout(new FlowLayout());
      setResizable(false) ;
      add(new Label(message));
      addWindowListener(this) ;
      setVisible(true);
   }
   public void windowActivated(WindowEvent we)  { }
   public void windowClosed(WindowEvent we)   { }
   public void windowClosing(WindowEvent we)  { dispose(); }
   public void windowDeactivated(WindowEvent we)  { }
   public void windowDeiconified(WindowEvent we)  { }
   public void windowIconified(WindowEvent we)  { }
   public void windowOpened(WindowEvent we)  { }
}
```

Listing 13-17: Implementation of a Dialog Frame.

In place of printing a string to the standard output, the previous `MenuExample` class may use the `Message` class to signal the delivery of an event. This is seen in Listing 13-18.

```
class MenuExample extends Frame
                  implements ActionListener, WindowListener {

      // ... code as before

   public void actionPerformed(ActionEvent evt)  {
      new Message(this,((MenuItem)evt.getSource()).getLabel());
   }

}
```

Listing 13-18: Invoking a Message Dialog.

A `Dialog` window may also be tagged as Modal via the `setModal()` method. In this case, it will receive all input from the parent window. The side-effect is that it must be closed before the parent application may proceed.

```
class Message extends Dialog implements WindowListener {
   Message(Frame parent, String message) {
      super(parent, "Message");
      setSize(400,100);
      setLayout(new FlowLayout());
      setResizable(false);
```

```
        add(new Label(message));
        addWindowListener(this);
        setVisible(true);
    }
    ...
    public void windowClosing(WindowEvent we) { dispose(); }
}
```

Listing 13-19: A Modal Dialog Frame.

13.3.10 File Dialog Frames

A common use of dialog is requesting for an input file to read from, or an output file to write to. An example view is shown in Figure 13-17. The `FileDialog` class, which is inherited from the `Dialog` class, easily provides this functionality.

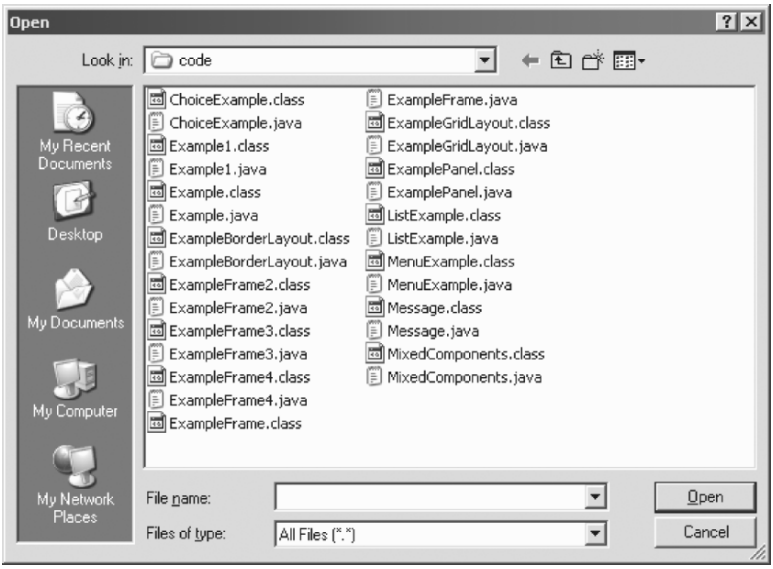


Figure 13-17: File Dialog for opening a file.

```
public FileDialog(Frame parent, String title, int mode);
```

The constructor method allows three parameters to be specified:

- the parent application;
- a title for the window; and
- whether it is the selected file to be loaded or saved.

```
public void actionPerformed(ActionEvent evt)  {
   FileDialog fd;
   String m = ((MenuItem) evt.getSource()).getLabel();
   if (m.equals("Open")) {
      fd = new FileDialog(this, "Open", FileDialog.LOAD);
      fd.setVisible(true);
      System.out.println(fd.getDirectory()+fd.getFile());
   } else if (m.equals("Save as")) {
      fd = new FileDialog(this, "Save As", FileDialog.SAVE);
      fd.setVisible(true);
      System.out.println(fd.getDirectory()+fd.getFile());
   } else
     new Message(this, ((MenuItem)evt.getSource()).getLabel());
}
```

13.4 Custom Components

Having seen the basic components provided in AWT, we will next see how custom components might be built. Here, it is easiest to reuse the framework set out by the Component class via inheritance.

For example, referring to the API documentation for Component, we determine that the `paint()` method is significant as it allows for a component to show itself. As such, in building a custom component, this method is redefined to give an appropriate graphical view of itself.

We consider building a `Puzzle` component that plays the 15-tile puzzle game, as shown in Figure 13-18. The object of the game is to maneuver the tiles via the single empty slot such that they are appropriately ordered.

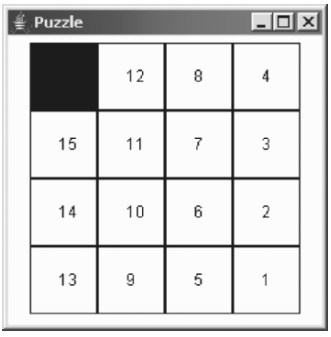


Figure 13-18: Puzzle board.

First, a component must maintain its state (in terms of board position) so that it can always paint itself, whether the window is restored after being minimized or brought to view after being hidden. In the case of the Puzzle component, we maintain the board with tiles at their current positions. We thus arrive at a skeletal structure as follows:

```
import java.awt.*;
class Puzzle extends Component {
  final int squares = 4;       // size of board
  int tileSize;                // pixels per tile
  int state [][];              // board
  int emptySlotX, emptySlotY;  // position of empty slot
  int moveX, moveY;            // current move
}
```

It is usual to have the constructor method provide the appropriate initialization, consisting of setting up the board and marking the empty slot.

```
Puzzle(int size) {
  tileSize = size / squares;
  state = new int[squares][squares];      // make the board
  for (int i=0;i<squares;i++)             // mess up the tiles
    for (int j=0;j<squares;j++)
        state[squares-j-1][squares-i-1] = i*squares+j+1;
  moveX = moveY = 0;                      // position of empty space
  state[moveX][moveY]=0;                  // mark the empty space
  addMouseListener(mouseListener) ;
}
```

The first of two important tasks that the Puzzle must fulfill is provide visual feedback in response to a changed board. This is done by overriding the `paint()` method originally defined in the superclass. As shown below, displaying the Puzzle is achieved by displaying each of the 15 tiles.

```
public void paint(Graphics g) {
   // clear board
   g.fillRect(0, 0, tileSize*squares, tileSize*squares) ;
   for (int i=0;i<squares;i++)  // print all tiles
      for (int j=0;j<squares;j++)
         square(g,i,j);
}
```

Each square is displayed by clearing a rectangle at the appropriate *x*-*y* coordinate, which is extrapolated from multiplying row/column with tileSize.

```
void square(Graphics g, int x, int y) {
   if (state[y][x] != 0) {                 // paint a tile
     g.clearRect(x*tileSize+1, y*tileSize+1, tileSize-2, tileSize-2) ;
     g.drawString(new Integer(state[y][x]).toString(),
                  x*tileSize+tileSize/2-4, y*tileSize+tileSize/2+4) ;
   }
}
```

The second task is to receive input as to where the user is playing the next move. Mouse activity is obtained by installing a `MouseListener` in about the same way as what we have done with `ActionListener` and `ItemListener` previously.

A `MouseListener` must implement five methods to track basic mouse activity: `mouseClicked()`, `mouseEntered()`, `mouseExited()`, `mousePressed()` and `mouseReleased()`. Two options are available here: the `Puzzle` component could

implement these methods and thus become a legitimate `MouseListener`. However, as only `mouseReleased()` is significant, the other four methods may be considered extra baggage.

A `MouseAdapter` is useful here. It is basically a class that provides a basic implementation for the five mouse activity methods. An appropriate `MouseAdapter` object merely inherits from `MouseAdapter` but redefines `mouseReleased()`. For the moment, we just provide the method:

```
public void mouseReleased(MouseEvent e)  {
  int newX = e.getX() / tileSize;
  int newY = e.getY() / tileSize;
  if (newX < squares && newY < squares &&
        emptySlotX == newX && (Math.abs(emptySlotY-newY) == 1) ||
        emptySlotY == newY && (Math.abs(emptySlotX-newX) == 1)) {
    //update board
    state [moveY=emptySlotY][moveX=emptySlotX] = state[newY][newX];
    state[emptySlotY=newY][emptySlotX=newX] = 0;
    repaint();
  }
}
```

Updates of the Puzzle state corresponding to tile movement as indicated by a mouse click must translate to update of the screen display. This is signaled by the method `repaint()` which schedules a display refresh by the `paint()` method.

We have now seen most of the code for our `Puzzle` component. However, we pause to consider, in the next section, an elegant means of including an `Adapter` object.

13.5 Other Kinds of Class Definitions

The delegation approach to event handling using listener objects is useful in that role of event handling may be distributed to various objects. Where event handling is minimal event handling is neater if it is handled by one object. In our Puzzle example, while we need to redefine `mouseReleased()`, redefining the other methods with null bodies becomes clumsy.

An example `MouseAdapter` class with null handlers for mouse events might be:

```
class MouseAdapter implements MouseListener {
   public void mouseClicked(MouseEvent e) { }
   public void mouseEntered(MouseEvent e) { }
   public void mouseExited(MouseEvent e) { }
   public void mousePressed(MouseEvent e) { }
   public void mouseReleased(MouseEvent e) { }
}
```

From this standard `MouseAdapter` class, we may derive a `PuzzleMouseListener` and redefine the `mouseReleased()` method:

```
class PuzzleMouseListener extends MouseAdapter {
   public void mouseReleased(MouseEvent e)  {
     // check move and update Puzzle board
   }
}
```

However, we encounter a scope problem in `PuzzleMouseListener` where its method `mouseReleased()` is expected to access instance variables of Puzzle. Even if this was done via proper accessor methods, the problem remains as to how two object instances of different classes may be logically associated.

13.5.1 Inner Classes

The revision of Java in JDK 1.1 allows for inner class, local class, and anonymous class definitions to solve this problem. While all classes in Java JDK 1.0 may only be declared at the top level, an inner class is one that is declared in another class. An example is given in Listing 13-20.

Such classes are unknown outside the enclosing class because its instances are implicitly used within instances of the enclosing class.

```
class Puzzle extends Component {
  ...
  int state [][];               // board
  ...

  class PuzzleMouseListener extends MouseAdapter {
    public void mouseReleased(MouseEvent e)  {
      // check move and update Puzzle board
      // has access to state
    }
  }
  PuzzleMouseListener listener = new PuzzleMouseListener();

  Puzzle() {
    ...
    addMouseListener(listener) ;
  }
}
```

Listing 13-20: Inner Class definition.

As such, this solves the scope problem in that `PuzzleMouseListener` may be instantiated within Puzzle, and yet the resultant instance has easy access to instance variables such as state.

13.5.2 Anonymous Classes

While the inner class facility is convenient in terms of overcoming scoping restrictions, they are useful within the enclosing class and unlikely to be reused outside the scope.

When there is to be only one instance of an inner class, the creation of an anonymous class instance provides the syntactic sugar for more succinct code. The

example code in Listing 13-21 provides identical functionality to the previous fragment in Listing 13-20.

```
class Puzzle extends Component {
   ...
   int state [][];              // board
   ...
   MouseAdapter listener = new MouseAdapter() {
     public void mouseReleased(MouseEvent e)  {
       // check move and update Puzzle board
       // has access to state
     }
   };
   Puzzle() {
     ...
     addMouseListener(listener) ;
   }
}
```

Listing 13-21: Anonymous class definition.

The classname `PuzzleMouseListener` has been omitted since it is used in the definition and only one instantiation. In an anonymous class, the definition occurs with instantiation, that is, after the new allocator.

13.5.3 Local Classes

While inner class definitions occur within an enclosing class, local classes are even more restricted in that they occur within blocks of method definitions. Listing 13-22 shows an example local class definition.

```
class Puzzle extends Component {
   ...
   int state [][];              // board
   ...

   Puzzle() {
      class PuzzleMouseAdapter {
         public void mouseReleased(MouseEvent e)  {
            // check move and update Puzzle board
            // has access to state
         }
      }
      ...
      addMouseListener(new PuzzleMouseAdapter());
   }
}
```

Listing 13-22: Local class definition.

In fact, if we desire, we can also create an instance of an anonymous class from the Puzzle constructor, as in Listing 13-23, but this does not necessarily improve code readability.

```
class Puzzle extends Component {
  ...
  int state [][];               // board
  ...

  Puzzle() {
    ...
    addMouseListener(new MouseAdapter() {
      public void mouseReleased(MouseEvent e)  {
        // check move and update Puzzle board
        // has access to state
      }
    });
  }
}
```

Listing 13-23: Another anonymous class definition.

Using instances of anonymous `MouseAdapter` and `WindowAdapter` (sub)classes, we again present the complete solution for our `Puzzle` component in Listing 13-24.

```
import java.awt.*;
import java.awt.event.*;

class Puzzle extends Component {

   final int squares = 4;      // size of board
   int tileSize;               // pixels per row
   int state [][];              // state of board
   int moveX, moveY,
       emptySlotX, emptySlotY; // positions of empty space
   Dimension psize;

   Puzzle(int size) {
      psize = new Dimension(size, size);
      tileSize = size / squares;
      state=new int[squares][squares];     // mess up the tiles
      for (int i=0;i<squares;i++)
         for (int j=0;j<squares;j++)
             state[squares-j-1][squares-i-1] = i*squares+j+1;
      emptySlotX = emptySlotY = 0;         // position of empty space
      state[emptySlotX][emptySlotY]=0;     // mark the empty space
      addMouseListener(mouseListener) ;
   }
   void square(Graphics g, int x, int y)  {
     if (state [y][x] != 0) {                // paint a tile
       g.clearRect(x*tileSize+1, y*tileSize+1, tileSize-2, tileSize-2) ;
       g.drawString(new Integer(state[y][x]).toString(),
                    x*tileSize+tileSize/2-4, y*tileSize+tileSize/2+4) ;
     }
   }
```

```
    public void paint(Graphics g)  {
      // clear board
      g.fillRect(0, 0, tileSize*squares, tileSize*squares) ;
      for (int i=0;i<squares;i++)            // print all tiles
         for (int j=0;j<squares;j++)
            square(g,i,j);
    }
    public Dimension getPreferredSize() { return psize; }
    MouseAdapter mouseListener = new MouseAdapter() {
      public void mouseReleased(MouseEvent e)  {
        int newX = e.getX() / tileSize;
        int newY = e.getY() / tileSize;
        if (newX < squares && newY < squares &&
            emptySlotX == newX && (Math.abs(emptySlotY-newY) == 1) ||
            emptySlotY == newY && (Math.abs(emptySlotX-newX) == 1)) {
          // update
          state[moveY=emptySlotY][moveX=emptySlotX] = state[newY][newX];
          state[emptySlotY=newY][emptySlotX=newX] = 0;
          repaint();
        }
      }
    };
   public static void main(String arg[]) {
     WindowAdapter windowListener = new WindowAdapter() {
       public void windowClosing(WindowEvent e) { System.exit(0) ; }
     };
     Frame f = new Frame("Puzzle");
     f.setSize(240,240) ;
     f.addWindowListener(windowListener) ;
     Panel p = new Panel();
     p.add(new Puzzle(200)) ;
     f.add(p);
     f.setVisible(true);
   }
}
```

Listing 13-24: Puzzle class Revisited.

13.6 Swing Components

The AWT components we have been discussing were the first graphical components Java ever had. For easy implementation, they are ultimately mapped to the native graphical widgets provided by the underlying windowing systems by various machine platforms such as Windows, Linux, or Mac OS.

Thus, while AWT runs on different platform (just like all other Java codes), AWT components look different on the various platforms. This is not too significant except that different dimensions can sometimes drastically alter how designed interfaces look.

A more significant concern here is that AWT components are constrained by what native graphical widgets are available on various target platforms. As such, AWT components tend to be plain and GUI programmers have always lamented about insufficient AWT interactivity.

This shortcoming has led to the introduction of Swing components in JDK 1.2. Swing components work in a similar fashion to AWT components in that we still instantiate suitable component classes and attach handlers for relevant events.

The wonderful aspect of Swing components is that they are implemented in pure Java. As such, they exhibit the same look-and-feel across platforms. While earlier versions of Swing were slow, Java implementors at SUN has improved the performance over releases of JDK. The more important side-effect of a pure Java implementation is that Swing is no longer constrained by native widgets. Swing provides a rich set of components for developing a responsive user-interface.

13.6.1 Transiting from AWT to Swing

Moving from AWT to Swing is generally easy. While the AWT component classes such as `Button`, `Label`, `Checkbox`, and `Frame` are defined in the `java.awt` package, the Swing package `javax.swing` contains classes `JButton`, `JLabel`, `JCheckBox` and `JFrame`.

Attaching a suitable event handler is unchanged. The following is the Swing equivalent of the previous `MixedComponents` example. Notice that most of the AWT components have been changed to the "J" equivalents. We can also mix the usage of AWT and Swing components, but this is in practice not encouraged.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

class JMixedComponents extends JFrame
              implements ActionListener, ItemListener, WindowListener {
  JTextArea txt;
  JMixedComponents(String m) {
    super("Mixed Components: "+m);
    JButton b; JCheckBox c;
    setSize(255,420);
    JPanel p = new JPanel();
    p.add(new JLabel("Conference Registration"));
    JPanel sub = new JPanel(); sub.setLayout(new GridLayout(2,1));
    sub.add(c = new JCheckBox("Attend Tutorial"));
      c.addItemListener(this);
    sub.add(c = new JCheckBox("Require Hotel"));
      c.addItemListener(this);
    p.add(sub);
    sub = new JPanel(); sub.setLayout(new GridLayout(3,1));
    ButtonGroup g = new ButtonGroup();
    sub.add(c = new JCheckBox("no food restrictions", true));
      g.add(c); c.addItemListener(this);
    sub.add(c = new JCheckBox("no seafood", false));
      g.add(c); c.addItemListener(this);
    sub.add(c = new JCheckBox("vegetarian food", false));
      g.add(c); c.addItemListener(this);
    p.add(sub);
    p.add(txt = new JTextArea(10,30));
    p.add(b = new JButton("Reset")); b.addActionListener(this) ;
    p.add(b = new JButton("Submit")); b.addActionListener(this);
    addWindowListener(this) ;
```

```
        add(p) ;
        txt.setText("Events:\n");
        setVisible(true);
      }
      public static void main(String arg[]) {
    new JMixedComponents("Event Handling");
  }
  public void actionPerformed(ActionEvent evt) {
    txt.append("actionPerformed: " + evt.getSource().toString() + "\n");
  }
  public void itemStateChanged(ItemEvent itm) {
    txt.append("itemStateChanged: " + itm.getItemSelectable() + "\n");
  }
  public void windowActivated(WindowEvent we) { }
  public void windowClosed(WindowEvent we)    { }
  public void windowClosing(WindowEvent we)   {
    System.out.println(txt.getText());
    System.exit(0) ;
  }
  public void windowDeactivated(WindowEvent we)   { }
  public void windowDeiconified(WindowEvent we)   { }
  public void windowIconified(WindowEvent we)   { }
  public void windowOpened(WindowEvent we)   { }
}
```

Listing 13-25: Swing version of MixedComponents example.

The following shows the resultant Swing `JFrame`.



Figure 13-19: Application with Swing components.

Similarly, the following code shows the Swing equivalent of the `MenuExample` with the resultant Swing `JFrame`.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

class JMenuExample extends JFrame {
   JMenuExample() {
      super("Menus");
      setSize(450,220) ;
      JMenuBar mb;
      JMenu m;
      mb = new JMenuBar();
      setJMenuBar(mb) ;
      mb.add(m = new JMenu("File"));
      m.add(new JMenuItem("Open"));
      m.add(new JMenuItem("Save"));
      m.add(new JMenuItem("Save as"));
      m.add(new JMenuItem("Close"));
      mb.add(m = new JMenu("Edit"));
      m.add(new JMenuItem("Undo"));
      m.add(new JMenuItem("Cut"));
      m.add(new JMenuItem("Copy"));
      m.add(new JMenuItem("Paste"));
      mb.add(m = new JMenu("Window"));
      m.add(new JMenuItem("New Window"));
      m.add(new JMenuItem("Arrange"));
      m.add(new JMenuItem("Split"));
      mb.add(m = new JMenu("Help"));
      m.add(new JMenuItem("Index"));
      m.add(new JMenuItem("Wizard"));
      m.add(new JMenuItem("About"));

      JPanel p = new JPanel();
      p.add(new JTextArea());
      add(p);
      setVisible(true);
   }
   // other methods for event handling ...
   public static void main(String arg[]) {
      new JMenuExample();
   }
}
```

Listing 13-26: Swing version of `MenuExample`.

The Swing components consist of more than just adding a "J"-prefix. For example, a `JButton` may be associated with an icon and buttons can be incorporated into menus. We will leave the reader to explore such enhanced features.



Figure 13-20: Application with Swing menu.

13.6.2 Model versus View

An important aspect of object-oriented programming is decoupling classes so that each class is as independent as possible from the others. For applications with graphical interfaces, this principle translates to separating the model (or data) from the view (or how the data is visualized). This strategy will allow the data to be rendered differently when requirements change.

Handling data and view in a single class may in the short term be simple. This may be reasonable with smaller applications or less complex data. But in the longer term with new requirements, tight coupling of data and view will hinder code maintenance and code reuse.

Swing components facilitate the separation of data from the view it provides. As an example, the `JTable` class provides for a table view. Its constructor allows for an array containing the data, or alternatively any class with implements the `TableModel` interface. This separation is illustrated in the `TableDemo` class below where the view provided by `JTable` is separated from the model in class `MultiplicationTable`.

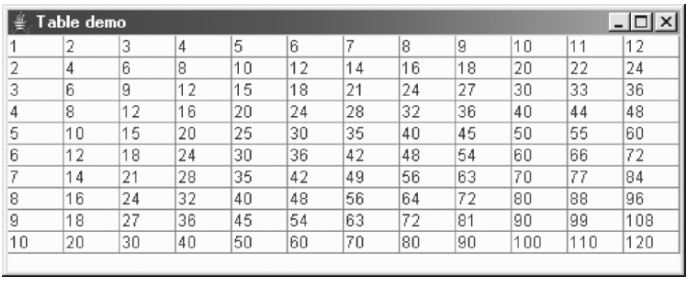
```
import java.awt.*;
import javax.swing.*;
import javax.swing.table.*;
import java.awt.event.*;
import javax.swing.event.*;

public class TableDemo extends JFrame {
   public TableDemo() {
      super("Table demo");
      add(new JTable(new MultiplicationTable()));
      setSize(500, 200);
      setVisible(true);
      addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
             dispose();
     }
      });
   }
   public static void main(String arg[]) {
      new TableDemo();
   }
}
```

```
class MultiplicationTable implements TableModel {
   public void addTableModelListener(TableModelListener l) { }
   public void removeTableModelListener(TableModelListener l) { }
   public Class getColumnClass(int columnIndex) {
     return("".getClass());
   }
   public int getColumnCount() {
      return(13);
   }
   public String getColumnName(int columnIndex) {
      return(String.valueOf(columnIndex));
   }
   public int getRowCount() {
      return(10);
   }
   public Object getValueAt(int rowIndex, int columnIndex) {
      return(String.valueOf((rowIndex+1)*(columnIndex+1)));
   }
   public boolean isCellEditable(int rowIndex, int columnIndex) {
      return(false);
   }
   public void setValueAt(Object aValue, int rowIndex, int columnIndex) {
   }
}
```

Listing 13-27: Swing separation between Model and View.

The class `MultiplicationTable` implements the model and only contains the data that will be rendered by the `JTable` view. As such, it needs to implement the `TableModel` interface that `JTable` expects. The resultant view is shown below:



1	2	3	4	5	6	7	8	9	10	11	12
2	4	6	8	10	12	14	16	18	20	22	24
3	6	9	12	15	18	21	24	27	30	33	36
4	8	12	16	20	24	28	32	36	40	44	48
5	10	15	20	25	30	35	40	45	50	55	60
6	12	18	24	30	36	42	48	54	60	66	72
7	14	21	28	35	42	49	56	63	70	77	84
8	16	24	32	40	48	56	64	72	80	88	96
9	18	27	36	45	54	63	72	81	90	99	108
10	20	30	40	50	60	70	80	90	100	110	120

Figure 13-21: Swing JTable.

13.7 Summary

While graphical user interfaces allow for convenient interaction and intuitive user models, good interfaces still require careful design and much implementation effort in event handling. The AWT and Swing packages in the Java API allows for a

consistent user interface at a fraction of the typical implementation effort. This is achieved by its object-oriented framework with reusable GUI components and integration mechanisms for attaching event handlers.

In this chapter, we discussed:

- the constituents of the AWT model and the framework for their integration in an application;
- detailed incremental development of an example user interface'
- the basic components in AWT available and their control within an application;
- the use of custom components when predefined components are inadequate;
- event handling using both containment and delegation models in JDK 1.0 and JDK 1.1, respectively,
- JDK 1.1 language extensions relating to inner class, local class, and anonymous class definitions; and
- Swing components and the separation of Model from View.

13.8 Exercises

1. Review the API documentation for the `File` class. Note that while the `File` instance is a directory, the `list()` method returns the list of files in it in a `String` array. Using this feature, implement a directory browser consisting of List and nonediting `TextArea` components. The former presents the set of files in a directory, while the latter displays the selected file.
2. Implement the following enhancements to the browser application discussed in Exercise 1:

   - allow for enabling edit-mode in the `TextArea` component, and subsequently saving the contents back to the original or new file;
   - allow for block-move edit operations via custom pop-up mouse menu;
   - search-and-replace edit operations via menu and dialog frames; and
   - allow for changing into new directories, which is reflected by displaying a corresponding new set of files in the `List` component.

3. Using the `drawRect()` method in the `Graphics` class, implement a rectangle drawing application that paints rectangles of various colors and positions indicated by pressing and releasing the mouse to denote opposite corners.

   - Include a rubber-band effect to indicate rectangle borders between selecting the two opposite corners of the rectangle.
   - Are existing rectangles redrawn when the frame is restored after being minimized? Suggest a scheme to implement this requirement.
   - Detect the right-hand mouse button to erase rectangles that enclose the mouse position.

14

Applets and Loaders

We have learned that Java is an interpreted language. A Java compiler (`javac` in JDK) translates source code to Java bytecodes so that it may execute as long as a Java Virtual Machine (`java` in JDK) is implemented to run on the underlying hardware. As such, Java is described as platform independent. In fact, the Java Virtual Machine not only allows Java to be source-code compatible, but object-code compatible as well. Object-code derived from compilation on one machine will execute unmodified on another.

This characteristic was exploited at the opportune time of exponential Internet growth, to pave the way for Java bytecodes to travel across a network, and subsequently be loaded into a virtual machine elsewhere for execution. A Java applet, which we have heard so much about over the media, is essentially Java bytecode which have traveled from a Web server to execute within an HTML page as displayed by a Web browser with an embedded Java Virtual Machine.

The ease of distributing Java applets over a network has destined this framework to revolutionize Web applications. With good GUI facilities and intuitive interfaces, enthusiasm for the Internet has been further heightened. The promise of Java applets was to turn static HTML pages to dynamic Internet applications.

We have deliberately left the popular subject of Java applets till now, because the applet framework relies on the AWT package, as well as networking and dynamic code loading facilities of Java. We will examine the execution environment of applets, some of its restrictions and how applications can take advantage of dynamically loaded code.

14.1 Applet Characteristics

The API documentation reveals that Java applets are derived from the `java.applet.Applet` class. This Applet class is in turn derived from the `java.awt.Panel`

class. Thus, an applet may be viewed as a special Panel that runs within the context of a Java-enabled Web browser such as Netscape Communicator or Microsoft's Internet Explorer.

Due to inheritance from its superclasses, operations applicable to a Panel such as placement of predefined or custom components, as discussed in Chapter 12, will also apply to an applet. There are additional properties peculiar to an applet, such as how it behaves within the operations of a Web browser and its life-cycle with respect to `init()`, `start()` and `stop()` methods. In addition, the Applet class implements additional interfaces that allow it to easily retrieve related Web resources from its originating server. It may also display an image and play an audio clip and communicate with other applets in the enclosing HTML document.

Listing 14-1 shows an example applet with a method `init()` which overrides the one defined in its superclass.

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

public class EgApplet extends Applet {
   public void init() {
      add(new TextArea(10, 60));
      add(new TextField(30)) ;
   }
}
```

Listing 14-1: Example applet.

This class definition is unique from those we have seen in that it does not have a `static void main()` method, as well as does not seem to have code which invokes `init()`. The significance of such particularities and the overriding method will be discussed in subsequent sections. For the moment, we present the corresponding HTML document in Listing 14-2, which is used to embed an `EgApplet` applet object.

```
<HTML>
<HEAD><TITLE>Example Applet</TITLE></HEAD>
<BODY>
<P>Example Applet</P>
<APPLET CODE="EgApplet.class" HEIGHT="270" WIDTH="450">
</APPLET>
</BODY>
</HTML>
```

Listing 14-2: Example HTML with embedded applet.

While most of the HTML tags are common, the `<APPLET>` tag stands out as one used to embed a Java applet into an HTML document. Here, two additional attributes are mandatory: the code file of the compiled applet proper, and dimensions (in terms of pixel height and width) of the display area within the browser required.

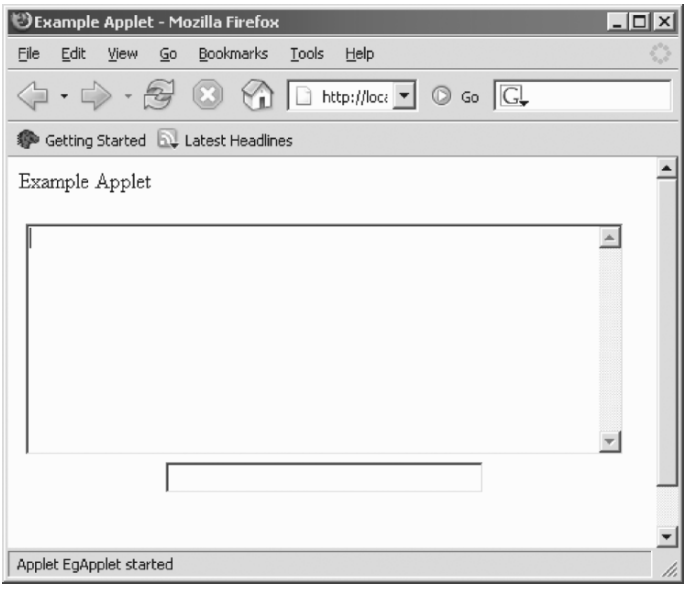


Figure 14-1: Applet in a Java-enabled browser.

In placing both HTML document and Java bytecode files in the same directory, and calling a Java-enabled Web browser to load the former (either through a local file load or an HTTPD server) will also load the applet and commence execution as shown in

Figure 14-1.

As with `Panel`s and `Component`s in Chapter 12, we may also attach event handlers for our components. We extend our original applet with an event handler and accessor methods to retrieve other HTML source files as follows:

- The `TextArea` and `TextField` components are assigned to instance variables `txt` and `inp` in order that we can interrogate them subsequently (as in `inp.getText()` and `txt.setText()` to retrieve input and write output, respectively).
- JDK 1.0 styled event handling is installed via the `action()` method to retrieve input in the `TextField` component in response to the return key. (This would ensure that the applet will still execute on Java-enabled Web browsers without AWT 1.1 libraries.) This is illustrated in Listing 14-3.
- The relative URL specified in `TextField` is used to retrieve a Web resource, whose contents are displayed in the `TextArea`. This is implemented by the `display()` method, and it uses the API class in `java.io` and `java.net`.

```
import java.applet.Applet;
import java.awt.*;
import java.io.*;
import java.net.*;

public class EgApplet extends Applet {
   TextArea txt;
   TextField inp;
   public void init() {
      add(txt = new TextArea(10, 60));
      add(inp = new TextField(30)) ;
   }
   public boolean action(Event evt, Object obj)  {
      if (evt.target instanceof TextField) {
         display(inp.getText());
         return true;
      }
      return(super.action(evt, obj));
   }
}
```

Listing 14-3: Event Handling in an Applet

The `EgApplet` applet in Listing 14-3 is essentially unchanged from Listing 14-1, except for instance variable declarations and the `action()` event handling method. For the purpose of modular descriptions, the `display()` method is shown in Listing 14-4. It relies on the URL class to construct an URL when given a relative address and the base address of the applet. The `openStream()` method initiates a Web request for the said URL, but relies on `InputStream` methods to retrieve results.

```
void display(String n) {
   try {
     URL doc = new URL(getDocumentBase(), n);
     InputStream is = doc.openStream();
     StringBuffer b = new StringBuffer();
     int c;
     while ((c = is.read()) != -1)
        b.append((char) c) ;
     is.close();
     txt.setText(b.toString());
   } catch (Exception e) {
     txt.setText("Cannot retrieve "+n);
   }
}
```

Listing 14-4: URL content retrieval.

As with applications which use predefined AWT components, applets may also be installed with JDK 1.1–styled event handlers. The restriction here is that such code can only be executed in a Web browser with the JDK 1.1 class libraries. (It is likely that these days, Netizens would be running Java-enabled browsers with JDK 1.1 libraries.)

The `EgApplet` version in Listing 14-5 uses JDK 1.1 styled event handlers. The applet differs from the JDK 1.0 version in that:

- it implements the interface for an `ActionListener` by defining the additional method `actionPerformed()`; and
- it installs itself as a listener for the `TextField` so that input there would subsequently trigger `actionPerformed()`.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;

public class EgApplet extends Applet implements ActionListener {
   TextArea txt;
   TextField inp;

   public void init() {
      add(txt = new TextArea(10, 60));
      add(inp = new TextField(30));
      inp.addActionListener(this) ;
   }
   public void actionPerformed(ActionEvent evt)  {
      display(inp.getText());
   }
   ... // void display() remains unchanged
}
```

Listing 14-5: EgApplet using JDK 1.1–styled event handling.

14.2 Applet Life Cycle

The two versions of the `EgApplet` applet in the previous section (which uses both JDK 1.0 and JDK 1.1-styled event handling) show that applets are very much like `Panel`s in the `java.awt` package. This is not surprising since the `Panel` class occurs in its inheritance chain.

However, because an applet executes within an HTML document that is rendered by a Web browser, its behavior is in a small way dictated by the browser's operation. For example, after an applet has been loaded and is running, provision must be made for when the browser leaves the page, or revisits the page. In the former case, the applet should be notified so that execution does not continue (unproductively). Similarly, when a page containing an applet is revisited, creating two instances of the same applet is wasteful in terms of computing resources.

A well-behaved applet responds to these state changes via the methods `init()`, `start()`, `stop()` and `destroy()`, as illustrated in Figure 14-2. The methods `init()` and `start()` are invoked for the first time when an applet is loaded and execution commences. After that, `stop()` and `start()` are invoked when the browser leaves the HTML page and revisits it. Finally, the `destroy()` method is invoked just before the applet is unloaded and disappears into oblivion. Any change in behavior corresponding to this state change should thus be effected by these methods.

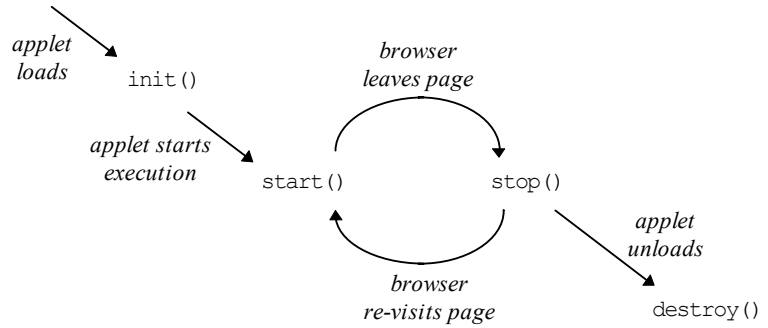


Figure 14-2: Applet life-cycle.

Consistent with object-oriented programming methodology, applets inherit the standard behavior for these methods `init()`, `start()`, `stop()`, and `destroy()`. Where no additional processing is required for the corresponding states, the standard behavior is adopted. Where required, custom processing is achieved by new method definition (which overrides the otherwise inherited method in the superclass).

The `EgApplet` class in the previous section has a custom `init()` method, but did not require special processing for `start()`, `stop()`, or `destroy()`. In the case of an applet with additional threads, it would be considerate of the applet to stop threads when the browser leaves the HTML page. In this case, the thread would be restarted if the page was revisited. Thus, the `start()` and `stop()` methods would then be defined accordingly.

14.3 Custom Applets

While the use of AWT components continues to be the same for applets, we now compare applets with applications developed in the previous chapter. In reincarnating the Puzzle application as an applet in Listing 14-6, we note two major differences:

- Applets are not invoked from the command line, but instead instantiated from a Web browser and with a life-cycle just discussed. As such, `static void main()` is not used for applet execution. (However, we might still include it so that the core applet code will run both as an applet and application. This will be discussed later.) Following from this, applets cannot be terminated via `System.exit()` because they exist as additional execution threads of the browser.
- While it is natural for an application class to use constructor methods for initialization, this is not the case with applets. In the former case, initialization is implicit at object instantiation. In the latter case, applets cannot always be initialized at instantiation since some properties are derived from its environment which is determined after applet instantiation. As such, applet initialization is thus best via the `init()` method.

```
public class Puzzle extends Applet {

   final int squares = 4;      // size of board
   int tileSize;               // pixels per row
   int state [][];              // state of board
   int moveX, moveY,
       emptySlotX, emptySlotY; // positions of empty space

   public void init() {
      tileSize = getSize().height / squares;
      state=new int[squares][squares];     // mess up the tiles
      for (int i=0;i<squares;i++)
         for (int j=0;j<squares;j++)
             state[squares-j-1][squares-i-1] = i*squares+j+1;
      emptySlotX = emptySlotY = 0;         // position of empty space
      state[emptySlotX][emptySlotY]=0;     // mark the empty space
      addMouseListener(mouseListener) ;
   }
   ... // other methods unchanged, static void main() now not required
}
```

Listing 14-6: Applet version of Puzzle.

14.4 Images and Audio

An applet may easily retrieve images and audio clips from its originating server via the methods

```
public Image getImage(URL url) ;
public Image getImage(URL url, String name);
public AudioClip getAudioClip(URL url);
public AudioClip getAudioClip(URL url, String name);
```

The resultant `Image` object retrieved by `getImage()` may be displayed via the `drawImage()` method in the `Graphics` class. The skeletal code for this is shown in Listing 14-7.

```
class DrawImage extends Applet {

   Image img;
   ...
   img = getImage(getDocumentBase(), "picture.gif");
   ...
   void paint(Graphics g)  {
      ...
      g.drawImage(img, x, y, this) ;
   }
}
```

Listing 14-7: Skeletal code for retrieving and displaying images.

Similarly, an `AudioClip` object retrieved via `getAudioClip()` may commence and stop playback via `play()` and `stop()` respectively. The skeletal code for this is shown in Listing 14-8.

```
class PlayAudio extends Applet {

   AudioClip song;
   ...
   song = getAudioClip(getDocumentBase(),"sound.au") ;
   ...

   void start() {
      ...
      song.play();
   }

   void stop() {
      ...
      song.stop();
   }
}
```

Listing 14-8: Skeletal code for retrieving and playing an audio clip.

The Media applet in Listing 14-9 demonstrates how these methods are used: it reads in an image and audio clip via `getImage()` and `getAudioClip()` in its `init()` method. The former is drawn by `drawImage()` in the `paint()` method, while the mouse event handler plays and stops the audio clip via `play()` and `stop()` corresponding to mouse press and release events.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Media extends Applet {

  Image image;
  AudioClip audio;
  MouseAdapter listener = new MouseAdapter() {
    public void mousePressed(MouseEvent e)  {
      audio.play();
    }
    public void mouseReleased(MouseEvent e) {
      audio.stop();
    }
  };
  public void init() {
    image = getImage(getDocumentBase(),"image.gif");
    audio = getAudioClip(getDocumentBase(),"audio");
    addMouseListener(listener) ;
  }

  public void paint(Graphics g) {
    g.drawImage(image, 0, 0, this) ;
  }
}
```

Listing 14-9: Media applet.

14.5 Animation in Applets

Since the introduction of the Java programming language, animation applets have been a popular means to liven up a Web page. The common technique for animating a Java component is making the `paint()` method redraw a new image between a specified time interval.

However, applets themselves should not be making continuous calls to the `paint()` method, or it will not allow other work to be done such as responding to events. Instead, an extra thread allows the internal state of the applet to be updated transparently. As with other threads, this work is done within the `run()` method.

The `NumberAnim` applet in Listing 14-10 creates a thread in the method `init()`, and starts it running to increment the instance variable count once every 100 milliseconds. In doing so, the display is also scheduled for refresh via `repaint()` over the same time interval.

```
      import java.awt.*;

public class NumberAnim extends java.applet.Applet implements Runnable {
      Thread updateThread;
      int counter = 0;
      public void init() {
      updateThread = new Thread(this);
      updateThread.start();
 }
      public void run() {
      for (;;) {
      counter++;
 repaint();
 try {
      Thread.sleep(100);
 } catch (Exception e) {
      }
      }
      }
      public void paint(Graphics g)  {
      g.drawString("counter " + counter, 40, 40) ;
      }
}
```

Listing 14-10: Basic animation.

This version of `NumberAnim` is inconsiderate as it continues to run even after the HTML page is no longer in view of the Web browser. While the previous `EgApplet` applet had no housekeeping chores to perform, the `NumberAnim` applet must however stop the active thread created by `init()`. This is a good situation where an overriding `stop()` method should be defined for the `NumberAnim` class:

```
public void stop() {
   updateThread.stop();
   updateThread = null;
}
```

The applet now prevents a runaway thread in that it stops its execution when it is now longer required. However, the counter does not resume if the applet page was

revisited. Here, we define another method `start()` to restart thread execution. Since this is similar to the `init()` method, we no longer need it but instead rely on `start()`.

```
public void start() {
   updateThread = new Thread(this);
   updateThread.start();
}
```

We now attempt something more interesting. Instead of a display of incrementing numbers, we will instead display a coordinated series of GIF images to provide the animation effect. Here are the modifications required:

- The initialization method `init()` loads the required set of images `image0.gif`, `image1.gif`, `image2.gif`… from the applet directory so that it is ready for subsequent display.
- In counting up, the method `run()` applies the `%` modulo operator on `count` so its value rolls over to index the next image to display.
- The `paint()` method draws the appropriate image to the display area.

The resultant `Animate` applet is shown in Listing 14-11.

```
import java.awt.*;

public class Animate extends java.applet.Applet implements Runnable {
   Thread updateThread;
   final int MAXIMAGES = 10;
   Image img[]; int counter=0;
   Color background;
   public void init() {
      background = getBackground();
      img = new Image[MAXIMAGES];
      for (int j=0; j<MAXIMAGES; j++)
         img[j] = getImage(getDocumentBase(),"image" + j + ".gif");
   }
   public void start() {
      updateThread = new Thread(this); updateThread.start();
   }
   public void stop() {
      updateThread.stop(); updateThread = null;
   }
   public void run() {
      for (;;) {
         repaint();
         counter++; counter = counter % MAXIMAGES;
         try { Thread.sleep(200); } catch (Exception e) { }
      }
   }
   public void paint(Graphics g) {
      g.drawImage(img[counter], 0, 0, background, null) ;
   }
}
```

Listing 14-11: Animate applet.

14.6 Efficient Repainting

While the `Animate` applet works in that it continuously displays a sequence of images, a slight display flicker is noticed on slower machines due to the way screen updates are performed for AWT components.

So far, we have only seen the `paint()` method which is called in response to a repaint request to refresh the display of an AWT component. In attempting optimum screen updates, we distinguish the two situations in which the `paint()` method is currently invoked.

- We have seen that repainting is necessary due to state changes, for example, a tile in the `Puzzle` has moved, or the next image in the animation sequence in `Animate` is to be displayed.
- The other situation that requires a display update is not caused by state changes within the applet, but instead by its windowing environment, for example, the applet is visible after being overlapped by another window, or the applet is restored to normal display size after being iconized.

In the second situation, it is likely that a complete refresh is necessary, but incremental updates might suffice in the former case. In the Puzzle applet example, the `paint()` method updates the whole puzzle board, and this is suitable for a formerly iconized or hidden applet. However, in response to movement of a tile, only two positions need to be updated—the original and final position of the tile that has been moved, while the display of all other positions remain unchanged.

The AWT component framework allows such distinctions to be made. In fact, the `paint()` method is invoked for whole updates, whereas the `update()` method is invoked for partial updates. We have thus far ignored the latter because its predefined definition is to clear the background area and rely on `paint()` for repainting. While this is a functionally correct definition and works for all situations, screen flicker in our demonstration applets is caused by erasing the background needlessly just before repainting it again.

A display optimization for the Puzzle applet merely redraws the affected squares may be easily deployed by overriding the `update()` method.

```
public void update(Graphics g) {
   // merely update affected board positions
   square(g, moveX, moveY);
   square(g, emptySlotX, emptySlotY);
}
```

Since the empty position was previously implied by no drawings and relied on the `paint()` method, using `update()` for incremental display requires the `square()` method to now also draw the blank position.

```
void square(Graphics g, int x, int y) {
  if (state[y][x] != 0) {                // paint a tile
    g.clearRect(x*tileSize+1, y*tileSize+1, tileSize-2, tileSize-2);
    g.drawString(new Integer(state[y][x]).toString(),
                 x*tileSize+tileSize/2-4, y*tileSize+tileSize/2+4);
  } else                                 // paint a blank position
    g.fillRect(x*tileSize+1, y*tileSize+1, tileSize-2, tileSize-2);
}
```

Display optimization for the `Animate` applet is also minimal, and consists of placing the body of `paint()` into `update()`, and having `paint()` invoke `update()`. Clearing the background explicitly is not necessary because `drawImage()` does so for invisible portions of an image using the selected background color.

```
public void update(Graphics g) {
   g.drawImage(img[counter], 0, 0, background, null) ;
}
public void paint(Graphics g) {
   update(g) ;
}
```

14.7 Applet Parameters

The `Animate` applet could be more useful if working parameters were not hardwired into code. Currently, the statically determined values include:

- the prefix of image files — `image`
- the number of images — 10
- the delay between 2 images — 200 (ms)
- the delay in restarting image sequence — (same as delay between two images)

Applet parameters may be specified via the `<PARAM>` tag within the `<APPLET>` tag. The associative scheme maps the parameter name in the "name" attribute of a `<PARAM>` tag to the corresponding value specified by the "value" attribute.

```
<APPLET code=Animate.class height=100 width=100>
 <PARAM name="imagesource" value="image">
 <PARAM name="maximages" value="10">
 <PARAM name="delay" value="200">
 <PARAM name="delaynext" value="1000">
</APPLET>
```

Parameter values are subsequently interrogated from an applet via the `getParameter()` method. Given the *name* attribute of a `<PARAM>` tag, it returns the

associated *value* attribute. A more flexible version of the `Animate` applet is shown in Listing 14-12.

```
import java.awt.*;

public class Animate2 extends java.applet.Applet implements Runnable {
   String IMAGESOURCE = "image";
   int MAXIMAGES = 10;
   int DELAY = 200;
   int DELAYNEXT = 200;

   Thread updateThread;
   Image img[]; int counter=0;
   Color background;
   public void init() {
      String val;
      if ((val = getParameter("imagesource")) != null)
        IMAGESOURCE = val;
      if ((val = getParameter("maximages")) != null)
        MAXIMAGES = val;
      if ((val = getParameter("delay")) != null) {
        DELAY = Integer.parseInt(val);
        if (DELAY < 50) DELAY = 50;
      }
      if ((val = getParameter("delaynext") ) != null) {
        DELAYNEXT = Integer.parseInt(val);
        if (DELAYNEXT < 50) DELAYNEXT = 50;
      }
      background = getBackground();
      img = new Image[MAXIMAGES];
      for (int j=0; j<MAXIMAGES; j++)
         img[j] = getImage(getDocumentBase(), IMAGESOURCE+j+".gif");
   }
   public void start() {
      updateThread = new Thread(this); updateThread.start();
   }
   public void stop() {
      updateThread.stop(); updateThread = null;
   }
   public void run() {
      for (;;) {
         repaint();
         counter++; counter = counter % MAXIMAGES;
         try {
            Thread.sleep(counter == 0 ? DELAYNEXT : DELAY);
         } catch (Exception e) {
         }
      }
   }
   public void paint(Graphics g) {
      g.drawImage(img[counter], 0, 0, background, null);
   }
}
```

Listing 14-12: Applet parameters.

To aid the documentation process, the redefinition of `getAppletInfo()` and `getParameterInfo()` methods allow an applet to provide information as to itself and expected parameters.

```
public String getAppletInfo() {
   return("Kiong B.K., Animate Applet, (c) 1998");
}

public String[][] getParameterInfo() {
   String info[][] = {
      {"imagesource","string","prefix name of image files"},
      {"maximages",  "int",   "number of images in 1 sequence"},
      {"delay",      "int",   "delay between 2 images (ms)"},
      {"delaynext",  "int",   "delay before 2 sequences (ms)"}
   };
   return(info);
}
```

Requesting applet information from `Appletviewer` returns a window similar to that shown in
Figure 14-3.

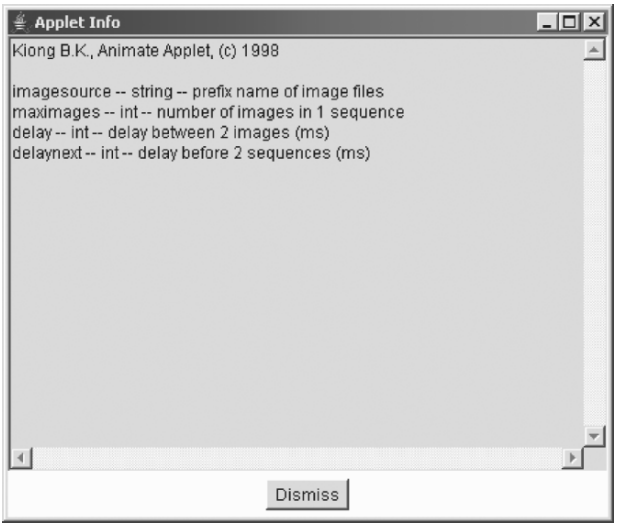


Figure 14-3: Applet information.

14.8 Loading Code Dynamically

Static analysis of a program (such as in Pascal, C, or C++) spread over a set of files will reveal what subroutines or class definitions are required, and whether sufficient

code is available for program execution. This is often the case for Java too, but not always true. Java is said to allow for dynamically loaded code.

The successful compilation of each class definition produces a bytecode file whose name is the name of the class with a "`.class`" suffix. This scheme allows for code to be loaded on a per class basis and as required.

Consider the case with `A.class` where the source `A.java` makes references to classes `B` and `C`. This would also be apparent in the compiled bytecodes in `A.class`, and loading it will subsequently require loading of codes in `B.class` and `C.class`. `D.class` would however not be loaded since it was not referenced from classes `A`, `B` or `C`, even if it existed in the same directory.

The `MyAppletLoader` class in Listing 14-133 reflects this situation. Loading `MyAppletLoader.class` will cause `EgApplet.class` to be loaded too. (The `MyAppletLoader` class does not work properly in that it does not have a `DocumentBase`, and it does not respond to Window events. These have been intentionally left out for the sake of simplicity. The main issue here is dynamically loading class code.)

```
import java.awt.*;
import java.applet.*;

class MyAppletLoader {
   public static void main(String args[]) {
      Frame host = new Frame("Host");
      Applet a = new EgApplet();
      host.add(a);
      host.setSize(450,270) ;
      a.init();
      a.start();
      host.setVisible(true);
   }
}
```

Listing 14-13: Static applet loader.

However, the `MyAppletLoader` class does not reflect the case of applet execution adequately. There are currently two common applet environments for applet execution: (i) a Java-enabled Web browser such as Netscape Communicator or (ii) an `appletviewer` that is bundled in the Java SDK distribution from JavaSoft. In this "real" situation, applet class files cannot be determined prior to the start of execution. Based on the attributes in the `<APPLET>` tag, the appropriate bytecodes are retrieved over the network, and loaded dynamically for continued execution. Thus, the applet to be loaded cannot be determined statically (when the environment was built), but at runtime when the `<APPLET>` tag has been read.

An applet loader functions by anticipating code for an Applet object, and then loading code over a network and assembling it into an appropriate class representation. After object instantiation, the applet environment supports the agreed life-cycle framework for applets involving initialization, execution, suspension and destruction through the methods `init()`, `start()`, `stop()` and `destroy()` respectively. Regardless of the exact class definition, the framework holds since the object is derived from the `Applet` class. The framework also holds even if the class was not directly

derived from the `Applet` class, or methods have been overridden. This is consistent with late binding and polymorphism in object-oriented programming methodology.

The `MyAppletLoader2` class in Listing 14-14 is slightly more realistic as now the applet name is not hardwired. It reads the applet name from the command line, just as a Web browser would determine the applet file from the HTML document.

```
import java.awt.*;
import java.applet.*;

class MyAppletLoader2 {
   public static void main(String args[]) throws Exception {
      Frame host = new Frame("Host");
      Applet a;
      Class c = Class.forName(args[0]) ;
      host.add(a = (Applet) c.newInstance());
      host.setSize(450,270) ;
      a.init();
      a.start();
      host.setVisible(true);
   }
}
```

Listing 14-14: Local applet loader.

`Class.forName()` is a static method which given a class name, attempts to load its code so as to represent it as a `Class` object. This is subsequently used to create instances via method `newInstance()`.

We now consider the situation where code files are not found in the standard local `CLASSPATH` set of directories, but instead must be retrieved over the network. This framework now approaches that of a typical applet environment.

We introduce the new class `MyNetworkLoader` in Listing 14-15. It inherits from the predefined `ClassLoader` class. The latter is an abstract class with the ability to process Java bytecodes into `Class` objects. We overload the `findClass()` method so that it may read compiled bytecodes from across the network (with an appropriate base URL prefix), and obtain the resultant `Class` object via the `defineClass()` method.

```
import java.io.*;
import java.net.*;

class MyAppletLoader3 {
   public static void main(String args[]) throws Exception {
      java.awt.Frame host = new java.awt.Frame(args[0]);
      java.applet.Applet a;
      Class c = new MyNetworkLoader(args[0]).loadClass(args[1]);
      host.add(a = (java.applet.Applet) c.newInstance());
      host.setSize(450,270) ;
      a.init();
      a.start();
      host.setVisible(true) ;
   }
}
```

```
class MyNetworkLoader extends ClassLoader {
   URL base;
   public MyNetworkLoader(String b) {
      try { base = new URL(b) ; } catch (Exception e) { }
   }
   public Class findClass(String name) throws ClassNotFoundException {
      try {
         System.err.println("Loading "+name+" from network...");
         URL f = new URL(base, name+".class");
         InputStream is = f.openStream();
         ByteArrayOutputStream b = new ByteArrayOutputStream();
         int x;
         while ((x = is.read()) != -1) b.write(x) ;
         byte data[] = b.toByteArray();
         return(defineClass(name, data, 0, data.length));
      } catch (Exception f) {
         throw new ClassNotFoundException();
      }
   }
}
```

Listing 14-15: Network applet loader.

Using the `MyAppletLoader3` class, an applet (say, `Puzzle.class` from the base location `http://localhost/java/demo`) may be retrieved from a Web server and executed via the command line:

```
$ java MyAppletLoader3 http://localhost/java/demo Puzzle
```

14.9 Security Restrictions for Untrusted Code

The ease of applet execution over the network makes the Java environment very attractive in terms of reducing code maintenance and distribution costs. Source code modifications are merely re-compiled and deployed on the Web server. The distribution of new code to all client machines is implied the next time the applet is required.

While this situation works reasonably well on an Intranet with good network bandwidth, there is a potential problem in open and untrusted networks such as the Internet. In making code execution and distribution as easy as clicking a Web hyperlink, malicious applets can in principle invade the machines of many unsuspecting users to wreck havoc.

The early JDK 1.0 solution to executing untrusted applets and yet maintain a secure environment was to isolate such code within a restricted sandbox environment. Here, malicious code cannot cause any damage to the client browser machine because access to local machine resources is denied.

The sandbox prevents remote applet code from:

- reading and writing files on the client machine;
- making network connections except to the host from where the applet originated;
- starting other programs on the client; or
- load library code or call native methods (to effect the above).

Consider the following `ReadFile` applet. In particular, the `readData()` method could be used to scan for confidential data in the local file system. Executing the code from within a standard appletviewer or browser will reveal restricted privileges within the restricted sandbox.

```
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;

public class ReadFile extends java.applet.Applet implements ActionListener
{
    TextField input;
    TextArea txt;

    public void init() {
        add(new Label("Source:"));
        add(input = new TextField(50));
        input.addActionListener(this);
        add(new Label("Contents:"));
        add(txt = new TextArea(10, 40));
    }

    String readData(String source) {
       try {
          BufferedReader r = new BufferedReader(new FileReader(source));
          String lin;
          StringBuffer buf = new StringBuffer();
          while ((lin = r.readLine()) != null) {
             buf.append(lin);
             buf.append('\n');
          }
          return(buf.toString());
       } catch (Exception ex) {
            return("Error in reading from "+source);
       }
    }

    public void actionPerformed(ActionEvent e) {
       txt.setText(readData(input.getText()));
    }
}
```

Listing 14-16: ReadFile applet.

Of course, executing it from the command line via a static `main()` method will in fact show that the code runs well. This is the main distinction between applets and applications.

```
public static void main(String arg[]) {
    final Frame f = new Frame();
    f.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent w) {
            f.dispose();
            System.exit(0);
        }
    });
```

```
        ReadFile g = new ReadFile();
        g.init();
        f.add(g);
        f.setSize(500, 300);
        f.setVisible(true);
    }
```

Listing 14-17: Running the ReadFile applet from the command-line.

While this sandbox implementation keeps malicious code out, it can also at times be too restrictive for code with legitimate reasons to break out of the sandbox, such as requiring access to the local file system, say for remembering user preferences.

JDK 1.1 addresses this shortcoming by allowing applets that legitimately require privileged access to breakout of the otherwise restrictive sandbox. Java bytecodes may be digitally signed so that they may be recognized by the client as trusted. In providing full access to machine resources, trusted remote code can become on par with local applications.

While the privileged position of trusted applets in JDK 1.1 solves the sandbox issue, it remains that there is too great a gulf between trusted and unsigned applets, with nothing in between. It is somewhat inconsistent with the security principle to provide only the minimum privileges for code to execute. A flexible and fine-grain access control security policy implies that different applets could be assigned different security policies as appropriate. JDK 1.2 improves on the JDK 1.1 model by providing fine control over what permissions to grant to different code. This is best illustrated by working on our `ReadFile` applet.

14.9.1 Security Policy

A Java security policy is a text file which specifies the set of permissions to be granted to code fragments. The `policytool.exe` program is part of the standard Java SDK and helps in the syntax of the policy file such as `~/.java.policy`.

The policy fragment in Listing 14-18 grants permissions to code loaded from the URL `http://localhost/book/secure/`. The `ReadFile` applet in Listing 14-16 loaded from any other location will fail to read any file, but loaded from `http://localhost/book/secure/` will at least have read access to `C:\boot.ini`.

```
grant codeBase "http://localhost/book/secure/*" {
  permission java.io.FilePermission "C:/boot.ini", "read";
};
```

Listing 14-18: CodeBase policy specification.

Alternatively, the policy fragment in Listing 14-19 grants permissions to code signed by user `dkiong`.

```
grant signedBy "dkiong" {
  permission java.io.FilePermission "C:/boot.ini", "read";
};
```

Listing 14-19: SignedBy policy specification.

14.9.2 Keys

Key pairs for signing Java bytecodes are stored in a keystore. Both are generated using the `keytool.exe` program distributed in the standard Java SDK:

```
keytool -genkey -alias dkiong
```

The signing process does not work on individual `.class` files but instead of a `.jar` Java archive. As such, we bundle our `.class` files into a suitable `ReadFile.jar` file.

```
jar -cf ReadFile.jar ReadFile.class ReadFile$1.class
```

The `.jar` file may now be signed via:

```
jarsigner ReadFile.jar dkiong
```

Finally, we augment the policy file with the location of the keystore.

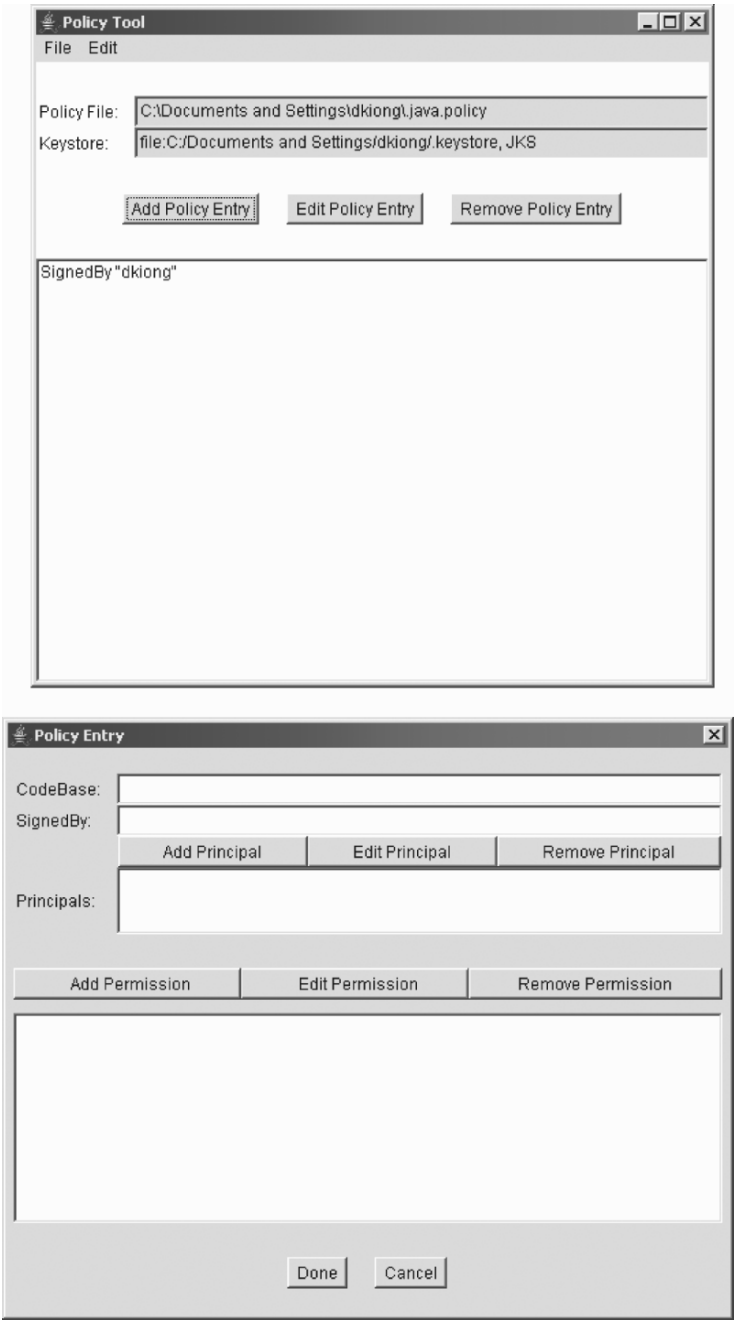
```
keystore "file:C:/Documents and Settings/dkiong/.keystore", "JKS";
grant signedBy "dkiong" {
 permission java.io.FilePermission "C:/boot.ini", "read";
};
```

14.9.3 Permissions

While a policy file might be worked out based on the sample examples, the GUI `policytool.exe` supplied as part of the SDK tools simplifies the specification of various permissions and their associated options.

```
AllPermission                    PropertyPermission
AudioPermission                  ReflectPermission
AuthPermission                   RuntimePermission
AWTPermission                    SecurityPermission
DelegationPermission             SocketPermission
FilePermission                   SerializablePermission
LoggingPermission                ServicePermission
NetPermission                    SQLPermission
PrivateCredentialPermission      SSLPermission
```

The granting of various permissions shows the fine grain control allowable in JDK 1.2 as opposed to privileged vs. sandbox option in JDK 1.1, which we discussed earlier.


Policy Tool
File Edit
Policy File:
C:\Documents and Settings\dkiong\.java.policy
Keystore:
file:C:/Documents and Settings/dkiong/.keystore, JKS
Add Policy Entry
Edit Policy Entry
Remove Policy Entry
SignedBy "dkiong"
Policy Entry
CodeBase:
SignedBy:
Add Principal
Edit Principal
Remove Principal
Principals:
Add Permission
Edit Permission
Remove Permission
Done
Cancel

14.10 Summary

Chapter 14 builds upon AWT components in the previous chapter to show how Java applets may be easily implemented and shipped across the network for execution. We discussed:

- the applet framework and its life-cycle;
- JDK 1.0 and 1.1 event handling for applets;
- converting between Java applications and applets;
- animation in applets;
- efficient screen updating via the `update()` method;
- applet parameters and security;
- dynamic loading of Java code using a custom `ClassLoader` and the `findClass()` method.

14.11 Exercises

1. The example Puzzle game is functional in that it allows users to rearrange tiles to achieve the desired ordering on the puzzle board. However, it does not give a clear indication of the state change because the view is updated instantaneously.

   Animate the display of tiles to give the effect of the selected tile moving to its new board position.
2. Extend the Media Applet so that six images may be displayed at one time. Prepare 6 different audio clips to be played when the mouse button is depressed over the corresponding image. In addition, allow for different messages when the mouse is over each image.
3. The applet in Exercise 2 could be used as a starting point to build an Internet CD kiosk. Extend it such that different sets of six images/audio clips may be selected. Remember the use of `<PARAM>` tags for customizable parameters.
4. Remote applets are prevented from reading and writing local files on the client browser to prevent malicious applets from reading sensitive files or even damaging the file system. However, applets are allowed to make socket connections back to the originating server.

   Design and implement a means whereby an applet may store and retrieve its data from the server via its network resource.

15

Java Servlets

We learned about dynamic loading of Java code in Chapter 13 where the name of a class may be determined dynamically at run-time and then subsequently loaded via the `Class.forName()` method. We also saw how this is exploited for applets. When an applet class is found embedded in an HTML document, its corresponding `.class` files are retrieved from the Web server and loaded into a typical Java-enabled host browser.

All applets inherit from `java.applet.Applet` superclass. The host browser loads the `.class` file and instantiates a corresponding applet object, regardless of the specific applet class. The browser-hosting environment merely has to invoke the appropriate life-cycle methods as polymorphism will ensure that any overriden methods will be correctly invoked. Such dynamic loading is an efficient way to dynamically increase the functionality of the host. The existing code need not be designed to expect only particular class names, nor do they need be recompiled. This offers much flexibility in providing opportunities to limitless functionality via plug-ins to be developed after the host system is complete.

15.1 Dynamic Web Pages and Servlets

While applets execute to increase the functionality on the host client browser, Java servlet technology also involves dynamic code loading for increased functionality but executes on the server-side instead.

A standard Web server may publish static HTML documents, but servlets can provide the customised backend processing for dynamic content personalized for a customer.

CGI-BIN scripts produced dynamic content in first generation Web servers. However, Java servlets can leverage on existing Java code, still take advantage of the

Java virtual machine and large set of Java API class libraries, as well as provide simplicity and scalability in large installations with concurrent users.

15.2 Tomcat Installation

The Servlet API and environment for servlets is not included in the standard Java SDK distribution. Apache Tomcat offers a Web server and a servlet-hosting environment to run servlets.

15.2.1 Downloading and Installation

Tomcat may be downloaded from the Apache Tomcat site. While it comes in various binary distributions, a single `.zip` file (say, from `http://tomcat.apache.org/download-60.cgi`) is probably easiest for novice servlet programmers whose prime motivations are to run servlet code as soon as possible. (See Figure 15.1)

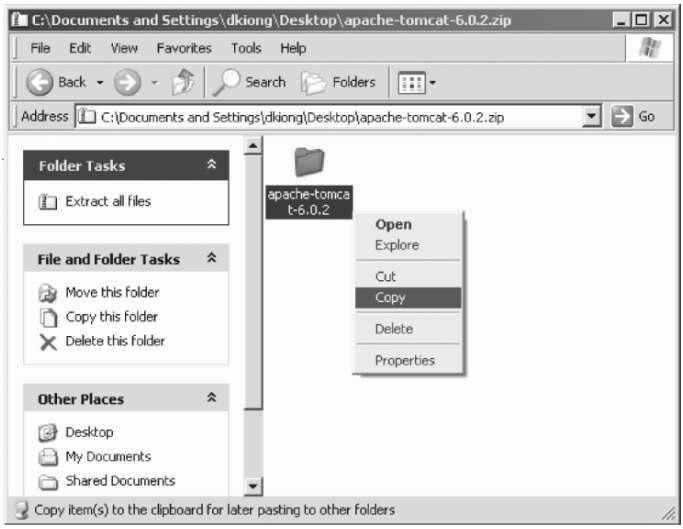


Figure 15-1: Tomcat binary distribution.

Extract the zip archive into a directory (Figure 15.2):

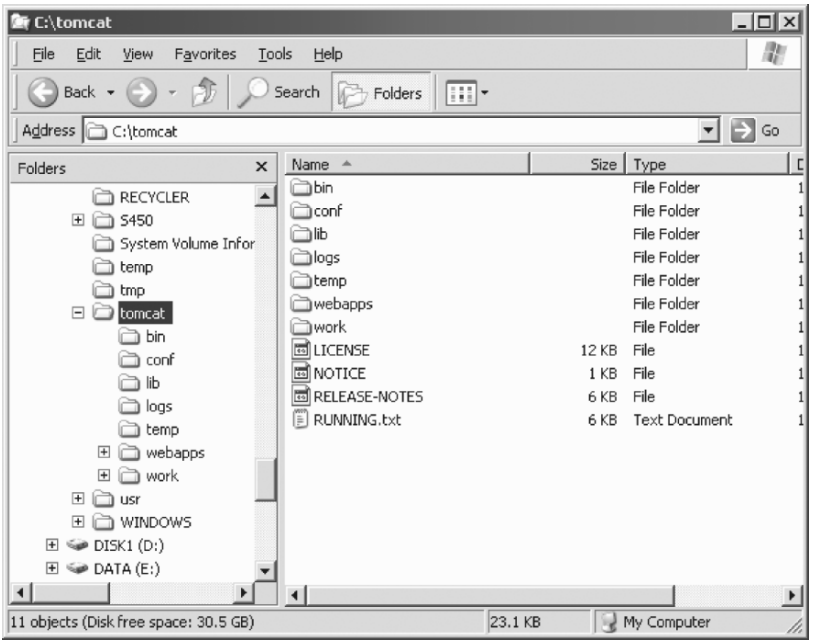


Figure 15-2: Tomcat home directory.

15.2.2 Configuration

The `bin/` directory contains `startup.bat` and `shutdown.bat` for starting and stopping Tomcat respectively.

The minimum configuration is to be explicit about the paths of the Java SDK and the home directory of Tomcat in the `catalina.bat` file. Edit to include two environment variables `JAVA_HOME` and `CATALINA_HOME`:

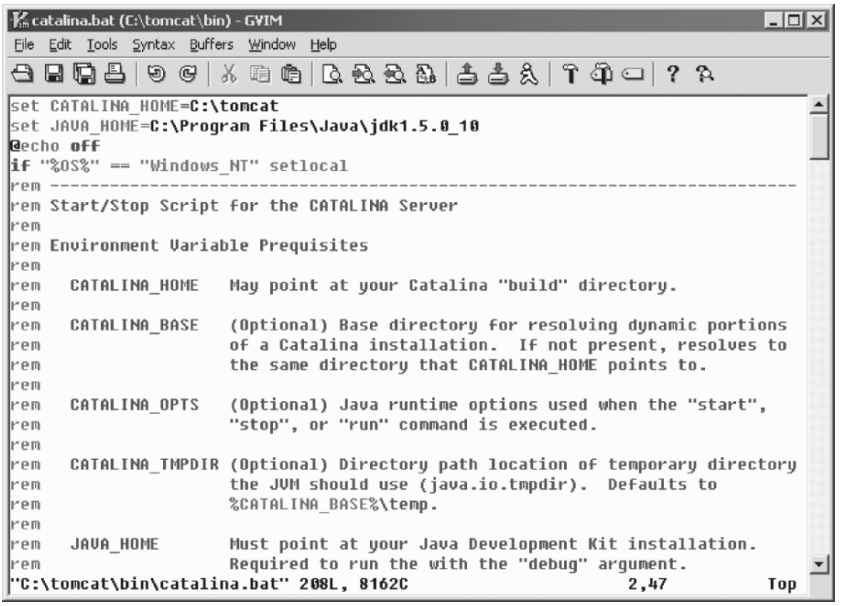


Figure 15-3: Setting up paths for Tomcat.

15.2.3 Starting and Stopping Tomcat

Double-clicking on `startup.bat` will start Tomcat and show a text log window that stays open for as long as Tomcat is running.

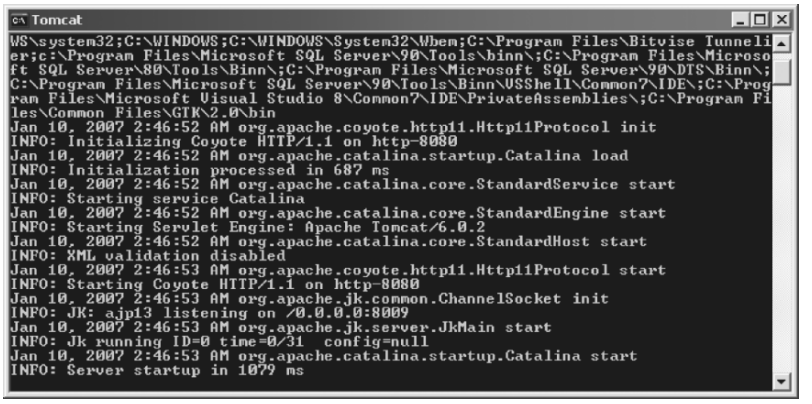


Figure 15-4: Log window when running Tomcat.

You may confirm that Tomcat is indeed running by pointing your browser to the URL `http://localhost:8080/` (8080 being the default network port from

which the Web server will listen for incoming requests). Seeing the start-up home page in your browser as below confirms that Tomcat has started correctly and is delivering content to requesting clients.

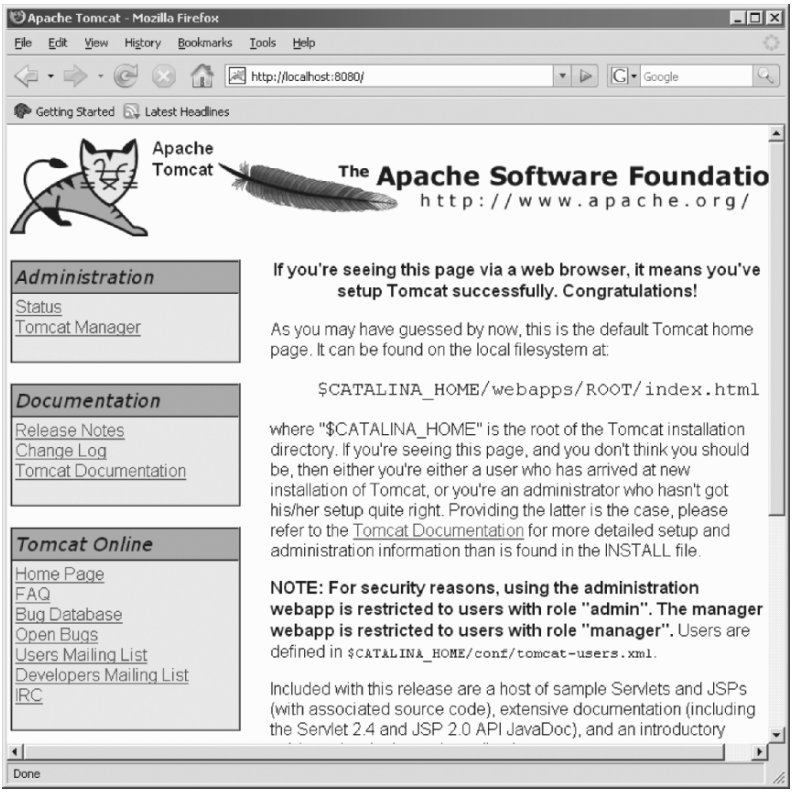


Figure 15-5: Tomcat start-up page.

15.3 Sample Servlet

A trivial servlet is shown below before we explain how it actually works:

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Sample extends HttpServlet {
   int count;

   public void init() throws ServletException {
      count = 0;
   }
```

```
    public void service(ServletRequest request,
                        ServletResponse response)
       throws IOException, ServletException {
       PrintWriter out = response.getWriter();
       out.println("<html>");
       out.println("<head>");
       out.println("<title>Sample</title>");
       out.println("</head>");
       out.println("<body>");
       out.println("<h3>Sample</h3>");
       out.println("<P>It is now "+new Date().toString());
       out.println("<P>This page has been accessed " +
                   ++count + " time(s)");
       out.println("</body>");
       out.println("</html>");
       out.println("</body>");
       out.println("</html>");
    }
}
```

Listing 15-1: Sample Servlet

We will use `${catalina.home}` to represent where Tomcat has been installed, eg `C:\tomcat` in describing how we might compile and deploy the sample servlet in Tomcat.

1. Place `Sample.java` in `${catalina.home}\webapps\examples\WEB-INF\classes`

2. Compile `Sample.java`

   Since we are accessing the servlet API classes (which are not included in Java SE), we will need to inform the compiler of its location via the `-classpath` option.

   ```
   $ javac -classpath ${catalina.home}\lib\servlet-api.jar Sample.java
   ```

3. Edit `${catalina.home}\webapps\examples\WEB-INF\web.xml`

   Insert a `<servlet>` tag for the `Sample` servlet after a similar tag for the `HelloWorldExample` servlet

```
<servlet>
    <servlet-name>HelloWorldExample</servlet-name>
    <servlet-class>HelloWorldExample</servlet-class>
</servlet>
<servlet>
    <servlet-name>Sample</servlet-name>
    <servlet-class>Sample</servlet-class>
</servlet>
```

Insert a `<servlet-mapping>` tag for the `Sample` servlet after a similar tag for the `HelloWorldExample` servlet

```
<servlet-mapping>
    <servlet-name>HelloWorldExample</servlet-name>
    <url-pattern>/servlets/servlet/HelloWorldExample</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>Sample</servlet-name>
    <url-pattern>/servlets/servlet/Sample</url-pattern>
</servlet-mapping>
```

4. Enter the URL `http://localhost:8080/examples/servlets/servlet/Sample` at the browser.



Figure 15-6: Servlet result.

15.4 Servlet Characteristics

The `Sample` class servlet shares some similar characteristics with applets:

(a) It does not contain a `static void main()` method, but instead inherits from a `HttpServlet` superclass so as to reuse framework code common to all servlets.

(b) `init()` and `service()` methods are overridden to provide customized servlet functionality in the form of backend processing. They are invoked at the appropriate point of a servlet life-cycle.

- The `init()` method is invoked when a servlet is dynamically loaded—corresponding to the first time that its corresponding URL is requested by a client.

- The `service()` method is invoked for every request by a client browser and corresponding response by the server-side processing.

Since the browser communicates with the Web server via a TCP socket, there is really a bidirectional exchange of messages or protocol. We saw this briefly in Chapter 11 when we discussed TCP sockets with the `Socket` class and the HTTP protocol. It is interesting that in the case of servlets, the bidirectional stream is encapsulated by the parameter types of the `service()` method—`HttpServletRequest` and `HttpServletResponse`.

The monotonically increasing value of the `count` instance variable of the `Sample` servlet shows that there is only one instantiation of the class, regardless of the number of browsers accessing the servlet via its corresponding URL. (We will investigate how the servlet may process inputs from different concurrent users in subsequent sections.)

15.5 Servlet Paramters and Headers

So far, we have seen how a Web server may merely return a static Web resource, or a servlet performing some backend execution to return some dynamic content. Just as typical programs and methods, it would be more flexible if servlet execution was accompanied by parameters. (In the case of Java applications with a `static void main(String arg[])` method, command line arguments are translated into method arguments.)

Servlet parameters typically come from user input in an HTML form. For example, the following HTML document

```
<html>
  <body>
    <p>Details of meal
    <form method="GET"
          action="http://localhost:8080/examples/servlets/servlet/Sample2">
      <p> Name: <input name="user">
      <p> Favourite meal:
      <ul>
        <input type="radio" name="meal" value="Breakfast"> breakfast
        <br><input type="radio" name="meal" value="Lunch"> lunch
        <br><input type="radio" name="meal" value="Dinner"> dinner
      </ul>
      <p><input type="submit">
    </form>
  </body>
</html>
```

Listing 15-2: Sample HTML form.

is rendered on a standard browser as follows:

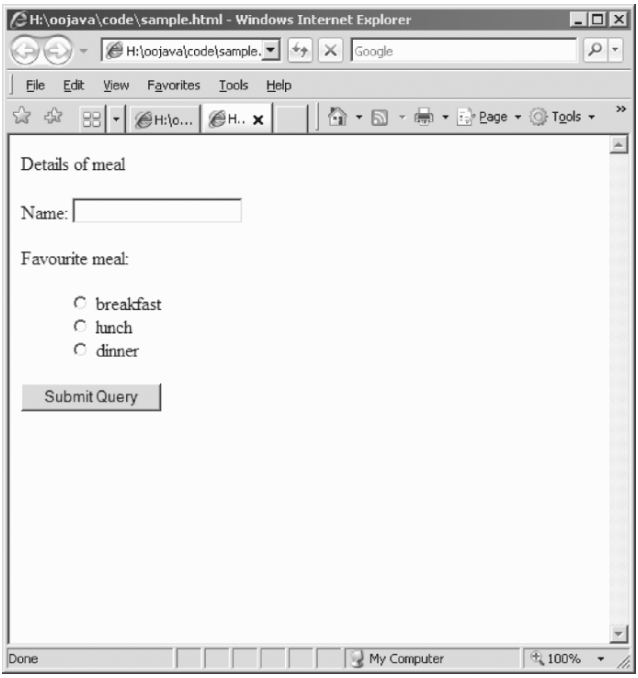


Figure 15-7: Rendered HTML form.

We could have the `Sample2` servlet extract user input submitted from an HTML form and interpolated into a message as follows:

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Sample2 extends HttpServlet {
    public void service(HttpServletRequest request,
                        HttpServletResponse response)
        throws IOException, ServletException {
    String name = request.getParameter("user");
    String whichMeal = request.getParameter("meal");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head><title>Sample2</title></head>");
        out.println("<body>");
        out.println("<h1>Accessing Parameters</h1>");
        out.println("<p>"+name+" would like "+whichMeal);
        out.println("</body>");
        out.println("</html>");
    }
}
```

Listing 15-3: Sample2 servlet.



Figure 15-8: `Sample2` servlet result.

Notice that parameters in a `GET` HTTP form submission request (as specified by the `method` parameter in Listing 15-2) are transmitted as part of the submission URL.

In the case of a `POST` HTTP form submission request (as specified in Listing 15-4), parameters are not seen in the URL because they are transmitted as part of the body of the HTTP message. In any case, the `HttpServletRequest.getParameter()` method still functions in the same way and the `Sample2` servlet gives the same results as seen in:

```
<html>
  <body>
    <p>Details of meal
    <form method="POST"
          action="http://localhost:8080/examples/servlets/servlet/Sample2">
      <p> Name: <input name="user">
      <p> Favourite meal:
      <ul>
        <input type="radio" name="meal" value="Breakfast"> breakfast
        <br><input type="radio" name="meal" value="Lunch"> lunch
        <br><input type="radio" name="meal" value="Dinner"> dinner
      </ul>
      <p><input type="submit">
    </form>
  </body>
</html>
```

Listing 15-4: Sample POST submission HTML form.



Figure 15-8: Sample2 servlet result with POST submission.

The `HttpServlet` class actually includes two instance methods—`doGet()` and `doPost()` to differentiate between GET and POST requests. In differentiating the two requests by overriding these methods, we can avoid a separate HTML document. We show this in the `Sample3` servlet in Listing 15-5.

Note that the `doPost()` method has the same implementation as `service()` in Sample2. However, now `doGet()` dynamically constructs the HTML form so that it is ready for submission. We use the method `HttpServletRequest.getRequestURI()` to reference the same URL for the POST submission.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Sample3 extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException {
        PrintWriter out = response.getWriter();
```

```
        out.println("<html><body><p>Details of meal");
        out.println("<form method=\"POST\" action=\""+
                        request.getRequestURI()+"\">");
        out.println("<p> Name: <input name=\"user\">");
        out.println("<p> Favourite meal:");
        out.println("<ul> <input type=\"radio\" name=\"meal\"");
        out.println("value=\"Breakfast\"> breakfast");
        out.println("<br><input type=\"radio\" name=\"meal\"");
        out.println("value=\"Lunch\"> lunch");
        out.println("<br><input type=\"radio\" name=\"meal\"");
        out.println("value=\"Dinner\"> dinner </ul>");
        out.println("<p><input type=\"submit\">");
        out.println("</form></body></html>");
    }

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
        throws IOException, ServletException {
        String name = request.getParameter("user");
        String whichMeal = request.getParameter("meal");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head><title>Sample2</title></head>");
        out.println("<body>");
        out.println("<h1>Accessing Parameters</h1>");
        out.println("<p>"+name+" would like "+whichMeal);
        out.println("</body>");
        out.println("</html>");
    }
}
```

Listing 15-5: `Sample3` servlet with overriden `doGet()` and `doPost()` methods.

Note that while we have used `HttpServletRequest.getParameter()`, `HttpServletRequest.getParameterNames()` is used when parameter names are dynamically known. Further, `HttpServletRequest.getParameterValues()` is used when there are multiple values for a parameter name.

In addition, while parameter values will be the main form of user input, there might be special occasions when HTTP headers are also examined. For example, the Web browser typically supplies information as to its implementation and referer page of the current request via the headers `user–agent` and `referer`.

Just as with parameters, `HttpServletRequest.getHeaderNames()` will return the set of headers as an `Enumeration`. `HttpServletRequest.getHeader()` will return the value of a specified header.

The sample code fragment in Listing 15-6.

```
Enumeration e = request.getHeaderNames();
while (e.hasMoreElements()) {
        String header = (String) e.nextElement();
    String val = request.getHeader(header);
    out.println("<p>"+header+"="+val);
}
```

Listing 15-6: Traversing set of HTTP headers

produces the following key/value pairs:

```
accept=image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-
shockwave-flash, application/vnd.ms-powerpoint, application/vnd.ms-excel,
application/msword, application/xaml+xml, application/vnd.ms-xpsdocument,
application/x-ms-xbap, application/x-ms-application, */*
referer=http://localhost:8080/examples/servlets/servlet/Sample3
accept-language=en-us
content-type=application/x-www-form-urlencoded
ua-cpu=x86
accept-encoding=gzip, deflate
user-agent=Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR
2.0.50727; .NET CLR 3.0.04506.30)
host=localhost:8080
content-length=31
connection=Keep-Alive
cache-control=no-cache
```

15.6 Servlet Output

So far, servlet output is performed via either a `Stream` or `PrintWriter` instance via `getOutputStream()` and `getWriter()` methods respectively using the `HttpServerResponse` parameter of `service()`, `doGet()` or `doPost()` methods.

While we have only created HTML documents for the Web client browser, a servlet may produce any other output formats such as PNG and JPEG images or even SVG vector graphics. The `HttpServletResponse.setContentType()` is used to specify the MIME type of the intended output format. We did not use it in our samples as the default is assumed to by `text/html`.

A `PrintWriter` instance is most convenient for text and HTML output, whereas other binary formats such as PNG and JPEG may easily rely on a `Stream` instance. In the case of the latter, it is also necessary to use the method `HttpServletResponse.setContentLength()` so that the client browser may prepare for output coming its way.

15.7 Handling Sessions

So far, our servlets cannot distinguish concurrent users because the HTTP protocol is stateless. This means that each HTTP request is serviced independently and without recollection of the previous requests. As such, only one instance of the servlet is ever created—which means that very little effort is expanded in serving an HTTP request. This helps the servlet framework work efficiently and that is very useful for a busy Web site.

HTTP requests are serviced by calls to `doGet()`, `doPost()` and `service()` methods. State changes of any local variables are discarded after processing a request and this is consistent with HTTP requests being stateless with no recollection after it is completed.

On the other hand, instance variables hold values across multiple requests (and possibly by multiple users too). The value of the `count` variable in the `Sample` servlet in Listing 15-1 can be seen to accumulate as the corresponding URL is accessed by various Web client browsers.

Between the use of local and instance variables, there is no other means to cater to a user session unless each browser is uniquely identified. This is achieved via browser cookies or URL rewriting—both schemes attempt to distinguish between user browser sessions. If a repeated request of the URL is attempted, the servlet may then restore a state from the previous request transaction.

The servlet framework handles cookie operations and thus such lower level implementation details are abstracted away for ease of use. As far as servlet API, the servlet can distinguish between new sessions from repeated visits. In the case of the former, a new set of state variables are created. For the latter, the old state from the previous service request is restored. Furthermore, before completing service, the current state must be stored, so that it may be restored at the next visit. This strategy is seen in the `Sample4` servlet below which accummulates wishes (just like a shopping cart):

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Sample4 extends HttpServlet {

    void show(HttpServletRequest request, HttpServletResponse response,
              ArrayList<String> wish)
        throws IOException, ServletException {
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Sample4</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<p>My wish list is ");
        if (wish.size() == 0)
            out.println("empty");
        else  {
            Iterator<String> i = wish.iterator();
            while (i.hasNext()) {
                String w = i.next();
                out.println("<br>+ "+w);
            }
        }

        out.println("<form method=\"POST\" action=\""+request.getRequestURI()+"\">");
        out.println("To add <input name=\"wish\">");
        out.println("<input type=\"submit\" value=\"submit\">");
        out.println("</form>");
        out.println("</body>");
        out.println("</html>");
    }
```

```
    public void service(HttpServletRequest request,
                        HttpServletResponse response)
        throws IOException, ServletException {
        ArrayList<String> wishList;
        HttpSession session = request.getSession(false);
        if (session == null) {
            // new session initialisation
            session = request.getSession();
            wishList = new ArrayList<String>();
        } else
            // restore from previous request
            wishList = (ArrayList<String>) session.getAttribute("wishlist");

        // service request
        String wish = request.getParameter("wish");
        if (wish != null)
            wishList.add(wish);
        show(request, response, wishList);

        session.setAttribute("wishlist", wishList); // save for next request
    }
}
```

Listing 15-7: `Sample4` servlet.

Session management involves instantiating an `HttpSession` object to represent the session (using the `getSession()` method) and being able to save and restore variables (using `setAttribute()` and `getAttribute()` methods) that make up the state of a session.

Note that the `getSession()` method is overloaded. `getSession()` will always return a `HttpSession` instance associated with previous requests. If it was the first request (and there was no previous), a new instance would be returned. It is convenient but does not distinguish between the first and subsequent requests.

`getSession(false)` is more selective and will only return a `HttpSession` instance if one already exist, that is, there was a previous request. `getSesion(true)` is similar to `getSession()`.

An `HttpSession` instance needs to recognise HTTP request from the same user session, as well as maintain the set of variables associated with that session. As this set of variables could differ from application to application, and possibly be dynamic as well, `HttpSession` uses an associative store of key/value bindings.

`Sample4` uses `getSession()` to distinguish between a first and subsequent requests. In the case of the former, appropriate initialization may be performed, such as creating a new `ArrayList`. In the case of the latter, we retrieve an `ArrayList` via the `getAttribute()` method. This assumes that an `ArrayList` instance was saved in the previous request via the `setAttribute()` method.

15.7.1 Session Timeout

Session handling involves saving session state for the next HTTP request when it will be restored. How long should we keep this session state for before we believe the user will no longer return (the network might be disrupted or his computer might have crashed)?

Storing session state indefinitely will mean that more storage space is required, especially for a large user base. At some stage, we must assume that the session must have terminated and that the user will not return for another request. Thus, we could perform some housekeeping and reclaim the storage allocated to session variables.

However, a short timeout interval could cause user frustration. A suitable timeout interval should be longer than the standard "think" time when a user is either reading or thinking how to proceed with his intended transaction.

The ideal session timeout interval is likely to be domain dependent. An application with high security requirements such as Internet banking is likely to have a shorter timeout interval, compared with a Web-based e-learning application where significant portion of time is spent reading Web-based materials.

The default Tomcat session timeout is specified as 30 minutes in `${catalina.home}/conf/web.xml`.

15.8 Concurrency

The possibility of concurrent users implies that the `service()`, `doGet()` and `doPost()` methods can be invoked and executed concurrently. A race condition can occur if these methods are to update shared variables simultaneously.

We have reviewed how to guard against overwriting shared variables in Chapter 11 on "Networking and Multithreading" by using the `synchronized` keyword. Note that in the case of servlets synchronized statements are only required for accessing to instance variables.

Local variables are allocated on the stack of each thread and not shared across threads. As such, they are not exposed concurrent access and would not need such restrictions. Thus, session data should always be held in local variables until they are saved across sessions using the `setAttribute()` method.

15.9 Customized Processors

While we have shown servlets being invoked explicitly from HTML forms, this need not be the only means. Most servlet environments allow for URLs with particular patterns (such as a file suffix) to implicitly invoke a servlet to process or translate its contents.

This is effected via the `web.xml` file. The `<servlet>` tag, which associates the servlet name with a class of the same name, remains the same as previous tags involving the `Sample` servlets.

```
<servlet>
    <servlet-name>Transform</servlet-name>
    <servlet-class>Transform</servlet-class>
</servlet>
```

The `<servlet-mapping>` is where you will see the difference; instead of a static URL to invoke the servlet, any pattern with a `.tml` invokes the servlet for preprocessing before returning the output to the client browser.

```
<servlet-mapping>
    <servlet-name>Transform</servlet-name>
    <url-pattern>*.tml</url-pattern>
</servlet-mapping>
```

The `Transform` servlet in Listing 15-8 does not look all too different from those we have written. We had previously used the `getRequestURI()` method so that the HTML form is submitted to the same servlet for processing. Here it is used to map the URL to a physical file for processing.

```
import java.io.*;
import java.util.*;
import java.util.regex.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Transform extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException {
        Pattern pat = Pattern.compile("\\\\b\\{([^}]*)\\}");
        PrintWriter out = response.getWriter();
        String file = "C:\\tomcat\\webapps" + request.getRequestURI();
        try {
            BufferedReader in = new BufferedReader(new FileReader(file));
            String ln;
            while ((ln = in.readLine()) != null) {
                Matcher m = pat.matcher(ln);
                if (m.find())
                    out.println(m.replaceAll("<font color=\"red\">"+
                                             m.group(1)+"</font>"));
                else
                    out.println(ln);
            }
            in.close();
        } catch (Exception e) {
            out.println("<pre>");
            e.printStackTrace(out);
            out.println("</pre>");
        }
    }
}
```

Listing 15-8: `Transform` servlet

While the example processing in the `Transform` servlet is trivial—substituting `\b{xxx}` substrings with `<font color="red">xxx</font>`—it demonstrates that, in principle, it can provide useful a transformation facility (e.g., XSLT transformations).

```
<h1>Welcome</h1>
<p>
This is a \b{sample} document
to be \b{transform} by the \b{Transform} servlet.
<p>
Does it work?
```

Listing 15-9: `hello.tml` data file

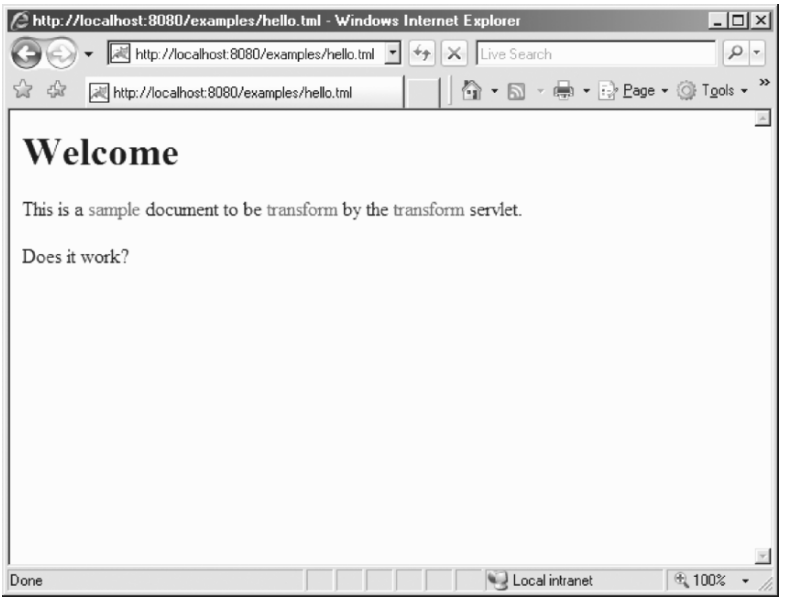


Figure 15-9: Rendering results of processing `hello.tml`.

15.10 Summary

Chapter 15 introduces Java servlets as an application of dynamic code loading to conveniently extend basic functionality without recompiling the base application. We discussed:

- the servlet rationale, its framework, and its life-cycle;
- the Tomcat servlet hosting environment (although other environments would have been suitable too);

- HTML `<form>` submission, sample servlets; and accessing input parameters;
- session handling and timeout;
- servlet mapping.

15.11 Exercises

1. Improve on `Sample4` so that the wish list may be both expand and contract.

2. Implement a servlet that will present a set MCQ (multiple-choice questions) to the user and mark them on completion.

16

Object Serialization and Remote Method Invocation

Almost every application requires some means of keeping data across program runs. Most applications use a file or database for the storage or *persistence* of data. However, databases are not typically used to store objects, particularly Java objects. On the other hand, flat files alone do not cope well with object structure. What is required is some means to preserve the state of a Java object so that it may be easily stored and subsequently restored to its original state.

Object serialization is a facility that enables objects to be "flattened" out so that they can be stored in a file or sent in a stream across a network. This is accomplished by "writing" the object into an `ObjectOutputStream` instance, which is then used to resurrect the object from the corresponding flattened representation. The serialization classes convert graph (hierarchies) of objects into bytestreams. Serialized objects may be written to a storage device for persistent retention of their state information or shipped across networks for reconstruction on the other side.

The JDK 1.1 (and beyond) provides the Object Serialization mechanism to tackle this once notorious problem of object persistency. Serialization also allows objects to be easily distributed across various Java Virtual Machines (JVMs). As such, we will also discuss Remote Method Invocation (RMI) where a program running on one JVM may invoke methods of objects on another JVM. In this scenario, Java RMI uses the Object Serialization API to pass and return objects during remote method invocation. We will examine RMI and what it brings to Java applications, with an emphasis on understanding the key concepts behind RMI. We will also develop simple applications to illustrate these concepts.

16.1 Object Serialization

The design of object serialization allows for most common cases to be handled easily. The following example code in Listing 16-1 shows:

- a `Serialize` class program that accepts a filename argument, and with methods `write()` and `read()` as representative code for serializing operations'
- an `ObjectOutputStream` being created from an `OutputStream` instance (in the form of a `FileOutputStream` object), and writing out via the method `writeObject();`
- an `ObjectInputStream` being created from an `InputStream` instance (in the form of a `FileInputStream` object), and reading via the method `readObject()`.

```
import java.util.*;
import java.io.*;
class Serialize {
   String filename;
   public static void main(String[] args) {
      Serialize a = new Serialize(args);
      a.write("This is a Serialization Test");
      System.out.println(a.read());
   }
   public Serialize(String[] args) {
      if (args.length < 1) {
         System.err.println("Usage: Serialize filename");
         System.exit(0);
      } else
         filename = args[0];
   }
   public void write(String str) {
      try {
         FileOutputStream out = new FileOutputStream(filename);
         ObjectOutputStream outobj = new ObjectOutputStream(out);
         outobj.writeObject(str) ;
         outobj.flush(); outobj.close();
      } catch (Exception e) {
         System.err.println("Failure while writing: "+ e.getMessage());
         e.printStackTrace();
      }
   }
   public String read() {
      try {
         FileInputStream in = new FileInputStream(filename);
         ObjectInputStream inobj = new ObjectInputStream(in);
         String str = (String) inobj.readObject();
         inobj.close();
         return str;
      } catch (Exception e) {
         e.printStackTrace(); return null;
      }
   }
}
```

Listing 16-1: Serialize.java.

16.2 Components in Object Serialization

Object serialization applies to objects such as a `String`, as in the previous example. Typically, a serialized object is a standard Java object, but it must implement the `java.io.Serializable` interface to be used with object serialization. The `serializable` interface does not have any methods, but instead it is merely used to indicate that the object may be serialized. (There are a few reasons why this empty interface is needed, but more about that later.)

The next concern of serialization is an input/output stream. An output stream is used to save data, as with the file output we saw earlier. Object serialization requires an instance of `ObjectOutputStream`, which is a subclass of `FilterOutputStream`. Like all such streams, `ObjectOutputStream` wraps itself around another output stream to use the output functionality.

On the face of things, serialization is trivial. We could save a serialized string to a file like this:

```
FileOutputStream fos = new FileOutputStream("obj.out");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject("Save me!") ;
```

The `writeObject()` method can be called any number of times to save any number of objects to the output stream. The only restriction is that each object that is passed to the `writeObject()` method must implement the `Serializable` interface.

Not surprisingly, reading a serialized object is equally trivial:

```
FileInputStream fis = new FileInputStream("obj.out");
ObjectInputStream ois = new ObjectInputStream(fis);
Object o = ois.readObject();
```

Once again, the `readObject()` method may be called unlimitedly to read any number of objects from the input stream. The potential pitfall when reading a stream of serialized data, is knowing what data is expected in the stream. Nothing in the stream identifies the types of objects that are there.

We can use the `instanceof` operator to determine the class of the object that the `readObject()` method returned, but that technique is useful only for verification. If we are expecting a `String` object, it can confirm a `String` object, but if another type of object is read, there will be no easy way to tell what type of object we have actually received. Hence, programs that serialize data streams must be kept in sync with the corresponding programs that de-serialize data, so that the latter may know what type of data to expect.

16.3 Custom Serialization

Because almost all classes in the Java API implement the `Serializable` interface, why should an empty interface be needed? One reason is due to the way in which

these objects are de-serialized from an object stream. De-serialization requires that an object be created, in a special way. Rather than creating the object by calling its constructor, object de-serialization creates the object directly on the heap and then begins to assign the saved values in the stream to the instance variables of the newly created object.

The JVM will only construct serializable objects in this manner. An interesting case arises when a serializable class extends a nonserializable class. In this case, the JVM will first construct the nonserializable object like any other object, that is, it creates the nonserializable object by calling its constructor, which must not require any arguments. Hence, a serializable class can only extend a nonserializable class when the latter has a default constructor.

The important benefit of distinguishing serializable objects, (from an administrative perspective, the important feature of the `Serializable` interface) has to do with the security of serialized objects.

Consider the situation with sensitive information, as in the following `CreditCard` class:

```
public class CreditCard implements Serializable {
   private String acctNo;
   ...
}
```

If this object is serialized into a file, the written data will include the account number too. Although there is other data in the file, the account number string will be readable to anyone with access to the file.

This happens because object serialization has access to all instance variables within a serializable class, which includes private instance variables. The instance variables will be sent in the I/O stream with the rest of the object. Anyone who reads the file where the object is saved will be able to see the private data. Similarly, anyanyone who is snooping the network, when a serialized object is sent over a `SocketOutputStream`, will also see the private data.

In a way, then, implementing the `Serializable` interface can be thought of as a flag to the JVM that says, "Hey JVM, I have thought about the security issues of my object, and it's OK with me if you write the private state of the object out to a data store." This raises the issue of security that requires special consideration, particularly when using sockets. A serialized object traveling across the Internet is subject to the same privacy violations as Email or any other unencrypted communication. It may be read by unintended parties, or it may be tampered with while in transit.

There are two ways to have the best of both worlds whereby the object may still be serialized, but without exposing any sensitive data. The first of these is to mark any sensitive data fields as `transient`, as in:

```
private transient String acctNo;
```

When it is time for the JVM to serialize an object, it will skip any fields in the object that are marked as transient (including any `public` or `protected` fields). In

other words, the `transient` keyword prevents selected fields from being written to a stream.

When an object is read in from a stream, transient data fields are set to their default values, such as 0 for integers and null for objects such as `Strings`. The programmer can restore transient data by implementing a `readObject()` method.

In general, sensitive data in serializable objects, such as file descriptors, or other handles to system resources, should be made both `private` and `transient`. This prevents the data from being written when the object is serialized. Furthermore, when the object is read back from a stream, only the originating class can assign a value to the private data field. A validation callback can also be used to check the integrity of a group of objects when they are read from a stream.

On the other hand, if transient data must be serialized together with the rest of the object, this may be achieved by overriding the `writeObject()` and `readObject()` methods. These methods provide control over what data is sent (or read) from the data store, and how that data looks while it is in transit. For example, we may redefine our previous `CreditCard` class as follows:

```
public class CreditCard implements Serializable {
   private transient String AcctNo;
   private int exprYear;

   ...

   public void writeObject(ObjectOutputStream oos) throws IOException {
      oos.defaultWriteObject();
      String s = modify(acctNo);
      oos.writeObject(s);
   }

   public void readObject(ObjectInputStream ois)
                          throws IOException, ClassNotFoundException {
      ois.defaultReadObject();
      String s = (String) ois.readObject();
      acctNo = unModify(s);
   }
}
```

Assuming that we have appropriate implementations of the `modify()` and `unModify()` methods, then this technique allows us to save the entire object including potentially sensitive data in a secure way. The `defaultWriteObject()` method is responsible for writing any nontransient fields (such as `exprYear`) to the output stream to be subsequently read by the `defaultReadObject()` method.

The `modify()` and `unModify()` methods may work by encrypting the string, or adding a few characters to confuse snoopers. As long as the symmetric operation is available when it is read in, we can take whatever necessary steps to protect such data.

16.3.1 The Externalizable Interface

With the `Externalizable` interface, the programmer takes full responsibility for reading and writing the object from a stream, including subclass and superclass data. This allows for complete flexibility, such as when a data format has already been defined, or the programmer has a specific format in mind. It also requires more programming, which is beyond the scope of this chapter, but may be an interesting topic for a future book.

16.4 Distributed Computing with Java

So far, we have merely worked with objects represented by one JVM. In distributed object computing, an object reference may be created locally and bound to a (remote) server object. The local program can then invoke methods on the local reference as if it was a regular local object. The distributed object infrastructure (generally referred to as an Object Request Broker, or ORB) transparently intercepts these method invocations and transmits the method request with its arguments to the server object (via a process known as marshalling), where the work is performed. The return values are then transmitted back to the local invocation.

The ability to pass information from one computer to another is the core of distributed computing. It allows multiple machines (presumably connected by a network) to work cooperatively on a single problem. Java can be treated like any other language in a distributed system where standard connection mechanisms include Remote Procedure Call systems (Distributed Computing Environment or Open Network Computing) or object request brokers CORBA (Common Object Request Broker Architecture).

In the Java distributed object model, a remote object is one whose methods can be invoked from another JVM, potentially on a different host. An object of this type is described by one or more remote interfaces, which are Java interfaces that declare the methods available on the remote object.

Distributed object computing offers many advantages over traditional approaches such as remote procedure calls (RPC) or socket-based client/server communication:

- The programmer is shielded from the complexity of the underlying communication mechanism. The developer interacts with a remote object via familiar method invocations, just as if the object was local.
- Distributed objects inherit the distinction between interface and implementation imposed by object-oriented programming. By separating the two, developers can work in parallel without concerning themselves about the implementation details of another developer's objects.

The elegance of Java lends itself well to the distributed object paradigm. The Java model boasts all of the object-oriented features necessary to build robust and highly maintainable object-oriented applications. With its rich and continually improving library of network classes, Java is also a network-centric language, as

demonstrated by the applet concept. These features beg for an elegant implementation of distributed Java objects. Not surprisingly, several efforts are already underway to add distributed extensions to the Java language.

16.4.1 RMI and CORBA

The Remote Method Invocation (RMI) standard in Java provides a distributed object model that crosses Java Virtual Machines seamlessly. Although RMI is an ORB in the generic sense that it supports making method invocations on remote objects, it is not a CORBA-compliant ORB. RMI is native to Java. It is, in essence, an extension of the core language. RMI depends on many of the other features of Java object serialization. Thus, the inclusion of RMI as a standard part of the JDK has caused much controversy, being a direct competitor of the CORBA standard for distributed objects.

One of the major differences between CORBA and RMI is that RMI allows objects to be passed by value. There are of course other differences, from the low-level protocol that each uses (CORBA uses a protocol called IIOP, and RMI uses its own protocol) to the programming interface that each provides (CORBA is programmed via IDL, and RMI is programmed using a normal Java interface).

Along with Java Database Connectivity (JDBC) and the Java Interface Definition Language (Java IDL), RMI forms part of the Java Enterprise API. Although it does not address all the issues of deploying objects in a heterogeneous environment, it provides the facilities needed by a wide range of distributed Java appplications in a "Java world."

By compromising on some generality, RMI has been assigned to retain the semantics of the Java object model and provide close integration with the rest of the Java system. It allows objects in one JVM to call methods of objects residing in other JVMs, with very little change in either the local or remote code. The main difference from the user's perspective is the need to handle the additional exceptions that may be generated by a remote object, mostly related to issues of communication.

Enterprise Java addresses how network-centric computing is changing the way applications are developed and deployed. It is a huge initiative and consists of a number of Java APIs. RMI addresses the incorporation of the network into a programming language, a key issue in network computing.

16.4.2 Java Limitations

In a distributed application, it is the designer's responsibility to select the protocol used to move data between client and server. Sometimes a well-known and supported protocol may be available, such as FTP for transferring files. More often, with a custom database application, for example, no such protocol exists. It is necessary to both design and build an application-specific protocol to connect to both parts of the system.

In this situation, Java itself offers no advantage over other languages. If performance bottlenecks are discovered when the system is deployed, functionality will have to migrate to rectify the problem. The protocol between client and server may require change, and so must the code that implements the protocol. In fact, code has to change for each different arrangement, making empirical tuning an expensive business. In other words, Java lacks support for location transparency.[1]

16.5 An Overview of Java RMI

Until the release of the RMI API, sockets were the only facility built into Java that provided direct communication between machines. RMI is quite like RPC (Remote Procedure Call), which is intended for use in procedural languages such as C. It allows programs to call procedures over a network as if the code was stored and executed locally. Think of RMI as the object equivalent of RPC. Instead of calling a procedure over a network as if it were local, RMI invokes an object's methods. In short, RMI abstracts the socket connectivity and data streaming involved with passing information between the hosts, so that method invocation to remote objects are made no differently than method invocation to local objects.

It is common practice for an object to invoke methods of other objects. This "local" method invocation forms the basis of object interaction in a program. For example, when a button object is clicked, it triggers a message that causes your program to invoke a method in a graphics object that causes it to calculate a 3D-rendered image.

RMI allows us to leverage on the processing power of another computer. This is called *distributed computing*. Clicking the button can cause the program to invoke a method in a graphics object on the server computer. The server then calculates the values needed to render the 3D object locally, and returns those values to the client program.

RMI attempts to make communication over the network as transparent as possible for the programmer. It may be used to interlink a number of objects that are distributed throughout a network and are physically residing on different machines. RMI brings the distributed objects under a virtual umbrella. From the application's point of view, a remote method and a local method are invoked in the same manner following the same semantics. RMI takes care of the details at the lower implementation level.

[1] This is addressed in JavaEE (Java Enterprise Edition) within the scope of Enterprise JavaBeans (EJB) architecture.

The end result is that programs enjoy advantages similar to those of client/server database programming without the complexity overhead. With RMI, a client program may invoke methods on the server object as if it was local. The method is then invoked and executed on the server machine (as required), but via a local syntax in the client program. This greatly simplifies the design of the application while leveraging on the processing power of possibly many computers.

16.6 Using Java RMI

The RMI API is a set of classes and interfaces designed to enable the developer to make calls to remote objects that exist in the runtime of a different JVM invocation. This "remote" or "server" JVM may be executing on the same machine or on an entirely different machine from the RMI "client."

16.6.1 Setting Up the Environment on Your Local Machine

The currently available JDK environment (J2SE) has RMI incorporated within the run-time seamlessly.

16.6.2 How RMI Works

Writing an RMI application is not inherently complex, but it has to be done in the correct order. The following are the steps to create an RMI application:

- Create an interface.
- Create a class that implements the interface.
- Create a server that creates an instance of this class.
- Create a client that connects to the server object using `Naming.lookup()`.
- Compile these classes.
- Run the RMI interface compiler (`rmic`) on the .class file of the implementation class to create the stubs. The stub classes provide the actual implementation for the underlying RMI functionality.
- Start the RMI registry (`rmiregistry`).
- Start the server class.
- Run the client program.

Creating the interface is perhaps the most important portion of the design of a RMI-driven multitiered client/server application. It defines the functionality the server will provide to the clients. Because the Java language does not allow multiple inheritance, the interface mechanism is used to allow classes to exhibit multiple types of behavior. An interface contains method declarations, but cannot contain method implementations.

RMI interfaces must extend the `java.rmi.Remote` interface, and every method declared in the interface must be declared as throwing a `java.rmi.RemoteException` (a generic exception that is reported when an unexpected network problem occurs). This is because a lot of work goes on behind the scenes to allow remote objects to be used in a seamless manner, and any number of problems can occur. For instance, the server could shutdown unexpectedly, or a network cable could be cut.

Each time a method is called, the parameters to that method must be serialized and sent back. The reverse occurs with results from methods.

16.6.3 An RMI Example

Let us now consider a simple example. Say we have two objects: a client and a server. We want the client object to invoke a method on the server object. Because the two objects reside on different machines, we need a mechanism to establish a relationship between the two.

RMI uses a network-based registry to keep track of the distributed objects. The server object makes a method available for remote invocation by binding it to a name in the registry. The client object, in turn, can check for availability of an object by looking up its name in the registry. The registry acts as a limited central management point for RMI and functions as a simple name repository. It does not address the problem of actually invoking the remote method.

Recall that the two objects physically reside on different machines. A mechanism is needed to transmit the client's request to invoke a method on the server object to the server object and provide a response. RMI uses an approach similar to RPC in this regard. The code for the server object must be processed by an RMI compiler called `rmic`, which is part of the JDK. This is depicted in Figure 16-1.

The `rmic` compiler generates two files: a stub that resides on the client machine and a skeleton that resides on the server machine. Both comprise Java code that provides the necessary link between the two objects.

When a client invokes a server method, the JVM looks at the stub to do type checking (since the class defined within the stub is an image of the server class). The request is then routed to the skeleton[2] on the server, which in turn calls the appropriate method on the server object. In other words, the stub acts as a proxy to the skeleton, while the skeleton is a proxy to the actual remote method.

[2] The skeleton class is not generated from JDK 1.2 onwards.

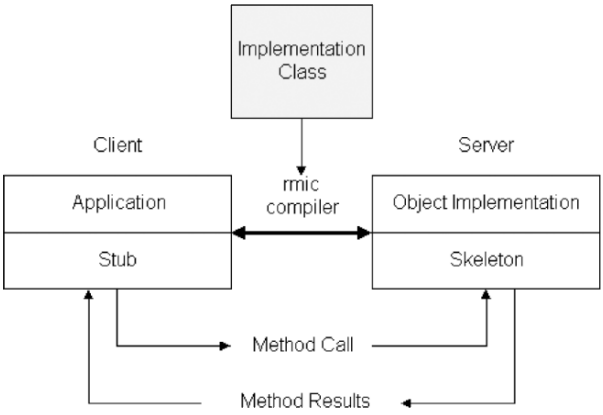


Figure 16-1: Java remote method invocation.

16.7 RMI System Architecture

The RMI system in Figure 16-2 is built in three layers: the stub/skeleton layer, the remote reference layer, and the transport layer. These layers are built using specific interfaces and defined by specific protocols in order to make the layers independent of one another. This was done intentionally to make the system flexible and allowing modification of the implementation of any given layer without affecting the other layers. For example, the TCP-based transport can be modified to use a different transport protocol. As mentioned earlier, RMI uses stubs and skeletons to act as surrogate placeholders (proxies) for remote objects. The transport of objects between address spaces is accomplished through the use of object serialization, which converts object graphs to bytestreams for transport.

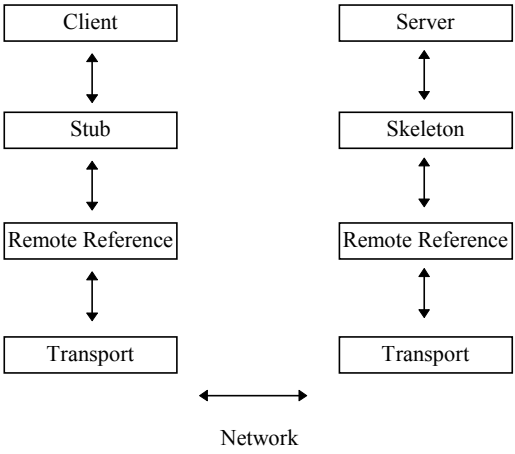


Figure 16-2: Java RMI architecture.

The stub/skeleton layer is the interface between the Application layer and the rest of the RMI system. This layer does not deal with any of the specifics of any transport, but transmit data to the Remote Reference Layer (RRL).

A client invoking a method on a remote server object actually makes use of a stub or proxy for the remote object as a conduit to the remote object. A skeleton for a remote object is a server-side entity that dispatches calls to the actual remote object implementation.

Stubs interact with the client-side RRL in the following ways:

- The stub receives the remote method invocation and initiates a call to the remote object.
- The RRL returns a special type of I/O stream, called a marshal stream, which is used to communicate with the server's RRL.
- The stub makes the remote method call, passing any arguments to the stream.
- The RRL passes the method's return value to the stub.
- The stub acknowledges to the RRL that the method call is complete.

Skeletons interact with the server-side RRL in the following ways:

- The skeleton unmarshals (receives and interprets) any arguments from the I/O stream, established by the RRL.
- The skeleton makes the up-call to the actual remote object implementation.
- The skeleton marshals the return value of the call (or an exception, if one occurred) onto the I/O stream.

The Remote Reference Layer (RRL) is responsible for carrying out the semantics of the method invocation. It manages communication between the stubs/skeletons and the lower-level transport interface using a specific remote reference protocol that is independent of the client stubs and skeletons. The RRL's responsibilities include managing references to remote objects and reconnection strategies if an object should become unavailable.

The RRL has two cooperating components: the client-side and the server-side. The client-side component contains information specific to the remote server, and communicates via the transport layer to the server-side component. The server-side component implements the specific remote reference semantics prior to delivering a remote method invocation to the skeleton.

The reference semantics for the server are also handled by the RRL. It abstracts the different ways of referring to objects that are implemented on servers—those that are always running on some machines, and those that are run only when some method invocation is made on them (activation). These differences are not obvious at the layers above the RRL.

The Transport Layer is a low-level communication layer that provides the actual shipment of marshal streams between different address spaces or virtual

machines. It is responsible for setting up and managing connections, listening for incoming calls, passing data to and from the remote reference layer. It also maintains a table of remote objects residing in particular address spaces.

The Transport Layer performs the following tasks:

- Receives a request from the client-side Remote Reference Layer.
- Locates the RMI server for the remote object requested.
- Establishes a socket connection to the server.
- Passes that connection back to the client-side Remote Reference Layer.
- Adds this remote object to a table of remote objects with which it knows how to communicate.
- Monitors connection "liveness."

At the Transport Layer, remote objects are represented by object identifiers and endpoints. An object identifier is used to look up which objects should be the targets of remote calls. Endpoints represent particular address spaces or virtual machines. The transport layer creates channels between endpoints by establishing connections and physically transferring data through input/output.

The RMI system uses a TCP-based transport, but the transport layer supports multiple transports per address space, so it is also capable of supporting UDP-based transport or even TCP and UDP.

16.8 Under the Hood

We have covered enough theory so that we can examine a simple application that uses RMI. Listing 16-2 shows a client application that invokes the `doSomething()` method of a remote object (of type `Server1`).

Firstly, all RMI-based applications will need to import `java.rmi` and `java.rmi.server` packages. The `static void main()` method sets the Java security manager and it is the job of the security manager to grant or deny permissions on the operations performed by the application (such as reading and writing a file). If the security manager is not set, RMI will only load classes from local system files as defined by `CLASSPATH`.

We then create a `try/catch` block that performs the remote method invocation. Recall that a registry acts as the repository of names for objects whose methods can be invoked remotely. The server object, in our case, has registered itself using the name "`ServerObject`." The client application must do a lookup in the registry using that name.

```
import java.rmi.*;
import java.rmi.server.*;

public class Client1 {
   public static void main(String[] args) {
      System.setSecurityManager(new RMISecurityManager());
      try {
```

```
            Server1 ro = (Server1) Naming.lookup(”doSomething”) ;
            System.out.println("Location:"+System.getProperty("LOCATION"));
            ro.doSomething();
         } catch (Exception e) {
            e.printStackTrace();
            System.exit(-1);
         }
      }
   }
```

Listing 16-2: Client1.java.

The returned object from the lookup operation is then assigned to the object `ro` which is of type `Server1`. The client can then use this object and invoke its methods as if it was a local object. In particular, the client application invokes the `doSomething()` method of the object `ro`, which in turn invokes the method with the same name on the `Server1` object.

`Server1`, which is used in the client application to declare the remote object type, is actually an interface. In fact, all remote objects are referenced through interfaces. Listing 16-3 shows the `Server1` interface. However, two points must be made about the `Server1` interface:

- Like all RMI interfaces, it extends the `Remote` interface.
- The method `doSomething()` throws a `RemoteException` exception which all remote operations must be able to handle.

```
import java.rmi.*;
public interface Server1 extends Remote {
  public void doSomething() throws RemoteException;
}
```

Listing 16-3: Server1.java.

We now have a client and a server application. The last piece is a class that implements the `Server1` interface shown above. Such a class is shown in Listing 16-4, which forms the heart of the application.

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

public class Server1Impl extends java.rmi.server.UnicastRemoteObject
                         implements Server1 {
   public static void main(String[] args) {
      System.setSecurityManager(new RMISecurityManager());
      try {
         Server1Impl obj = new Server1Impl();
         Naming.rebind("doSomething", obj) ;
         System.out.println("doSomething bound in registry");
      } catch (Exception e) {
         e.printStackTrace();
         return;
      }
   }
```

```
    public Server1Impl() throws RemoteException {}

    public void doSomething() throws RemoteException {
       System.out.println("This is printed by the Server1 object");
       System.out.println("Location: "+System.getProperty("LOCATION"));
    }
}
```

Listing 16-4: Server1Impl.java.

The `Server1Impl` class extends the `UnicastRemoteObject` class. This class defines a nonreplicated remote object whose references are valid only while the server process is alive. It supports point-to-point active object references via TCP streams. Once again, the first thing the main method does is to set the security manager. It then instantiates an object of type `Server1Impl`. This object is registered in the registry using the name `doSomething`.

We can use the `bind()` method instead of `rebind()`. The difference is that `rebind()` will replace the name in the registry if it already exists. Method `doSomething()` of `obj` is now available for remote invocation.

Next, the constructor for the `Server1Impl` is defined. A remote implementation class must have a zero-argument constructor. In addition, the constructor method must throw `RemoteException`.

Finally, the `doSomething()` method is defined. Note that since it is a remote method, it throws `RemoteException`. In our case, the remote method does not really do anything useful. It merely prints a couple of lines to the console. In a more meaningful application, the remote method may perform a query on a database, or read and process data from a file.

16.9 RMI Deployment

Our simple RMI application consists of a client, a server, and an implementation of the server interface. We still need two more pieces: the stub and the skeleton. To generate these two, we use the `rmic` compiler that comes with the JDK. You can generate them by the following command[3]:

```
$ rmic rmi1.Server1Impl
```

[3] Use `$ rmic -v1.2 rmi1.Server1Impl` for JDK 1.2 onwards. This suppresses the generation of the skeleton class.

This will create the two files: `ServerImpl_Stub.class` and `ServerImpl_Skel.class`. Should you want to see the Java source code for these files, use the `-keepgenerated` option.

One may wonder why `rmic` is used with the implementation of the server and not the server itself. The answer is that the method to be invoked remotely is defined in the `ServerImpl` class, and the purpose of `rmic` is to generate stubs and skeletons for the remote methods.

We now have all the necessary pieces for our application. The following three commands may be executed in separate windows in the order shown:

```
$ rmiregistry &
$ java -DLOCATION=server rmi1.Server1Impl
$ java -DLOCATION=client rmi1.Client1
```

The first command is not going to generate any output. It basically starts the registry in the background.

The second command starts the server, which registers the remote object with the registry. We use the `-D` option to set a system parameter `LOCATION` which is used to indicate that the code belongs to the server.

The last command starts the client. Again, we use the `-D` option to set the value of the system parameter `LOCATION` to client. Whenever the client starts, a message will be printed on the server window, since the remote method `doSomething()` simply prints a couple of lines.

The client may be started several times, and the messages would be correspondingly printed. This shows that the client application has started successfully and invoked the method on the server application using RMI.

The above three commands may be executed on three different machines, with the same results. Remember to distribute the skeleton and stub class files appropriately. The stub goes with the client application while the skeleton goes with the server application.

With very little coding effort, we have created a prototype for a potentially valuable and marketable service that could be deployed on the Internet today.[4] As distributed object computing becomes more viable with time, we should see the rise of a rich library of distributed objects available on the Internet. Entire class libraries could be deployed on the Internet as distributed objects, enabling Java programmers around the world to use them in their own applets and applications.

[4] The Sun Java Enterprise System implements RMI-IIOP (a cross between RMI and IIOP) to enable objects from different operating platforms to communicate with one another.

16.10 Summary

This chapter has introduced Java solutions to key technologies: object serialization for object persistence and distributed objects for effortless client/server communication.

- Object serialization allows objects to be flattened and represented in a bytestream, for subsequent reconstruction to its original state.
- Java RMI provides a framework for communication between Java programs running in different virtual machines.
- Using object serialization, Java RMI allows for parameters in the form of object graphs to be converted into bytestreams for transport across the network and reconstruction on the other side.

The RMI system is unique in that it preserves the full Java object model throughout the distribution, allowing true polymorphism of both remote and local objects. The syntax of a remote method call is exactly the same as the syntax of a local remote call, making distributed programming easy and natural. Any Java object can be passed during remote method calls, including local objects, remote objects and primitive types.

16.11 Exercises

1. Implement an ordered binary tree so that it will accept a list of words. Serialize the tree, and then retrieve it in another program to confirm that the resultant tree structure is unchanged.
2. Using RMI, implement a service to accept a filename and retrieves the contents of the remote file.

17

Java Database Connectivity

Applications that require complex manipulation of data will need to use a database to store information. Many Java applications fall into this category. Although many database vendors provide Java APIs, they are proprietary to their database implementations. Therefore, it becomes very difficult to port applications from one database to another. Java Database Connectivity (JDBC) APIs provide a simplified and uniform access to the database management systems from different vendors. In this chapter you will learn the essential classes and techniques to use JDBC APIs.

17.1 Introduction

Data can be stored in normal text or binary files. As data grows, and applications need to manipulate data in complex ways, this simplistic approach of storing data may not be sufficient. One of the best alternatives to files is to store data in databases. Databases allow us to manipulate and store data in an organized way, and hence transform data into meaningful information. In today's world, a database is an integral part of any application.

Definition of a database in WikiPedia, the free online encyclopedia, is as follows: "A database is a collection of records stored in a computer in a systematic way, so that a computer program can consult it to answer questions. The items retrieved in answer to queries become information that can be used to make decisions."

17.2 Java Database Connectivity

Different database vendors provide different products to implement data access and manipulation mechanisms for storing and retrieving data stored from databases. These products are called Database Management Systems (DBMS). Oracle 10g, MySQL, MS-SQL, MS-Access are examples of some popular DBMSs. DBMS

vendors also provide language-specific Application Programming Interfaces (APIs), so that applications developed using specific languages can use these APIs to access and manipulate data. Unfortunately, APIs greatly differ from one vendor to the other even for the same language.

Java Database Connectivity (JDBC) APIs are a set of Java classes that can be used to develop Java applications that need to access data from a relational database. JDBC APIs are independent of any vendor-specific implementations of DBMS, and hence makes applications portable. For example, if you write a database application for Oracle DBMS, the same application will work for MySQL DBMS with minimal modifications. The specifications for the JDBC APIs have been developed by Sun Microsystems, in conjunction with many popular database vendors.

17.3 JDBC Architecture

JDBC APIs can be used in any form of Java program such as stand-alone java applications, Applets, Servlets, and so on. Figure 17-1 shows JDBC in the context of an application.

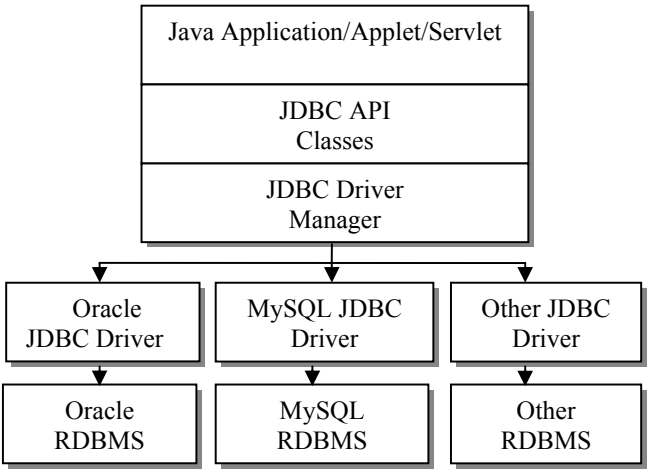


Figure 17-1: JDBC Architectural Stack.

17.4 JDBC Drivers

As can be seen from the JDBC Architectural Stack, a Java program accesses the DBMS via specific drivers. Drivers are a set of Java classes that translate the standard JDBC API calls to the DBMS specific API calls. They can be thought of as adapters that make the DBMS-specific calls on behalf of the applications. This is the layer that makes a Java database application portable across various relational

DBMSs. If a Java application developed with Oracle DBMS is to be ported to other DBMSs, it can be done by substituting the Oracle driver with another suitable driver.

17.4.1 Types of Drivers

JDBC Drivers come in four different flavours. DBMS vendors usually supply different types of drivers for their products.

17.4.1.1 Type 1: JDBC-ODBC Bridge

These types of drivers make use of the existing ODBC (Open Database Connectivity) connection to databases. Native drivers and libraries need to be installed and relevant configuration setting need to be done at the client side. Hence, these drivers are not very portable across databases. Figure 17-2 shows the architecture of Type 1 drivers.

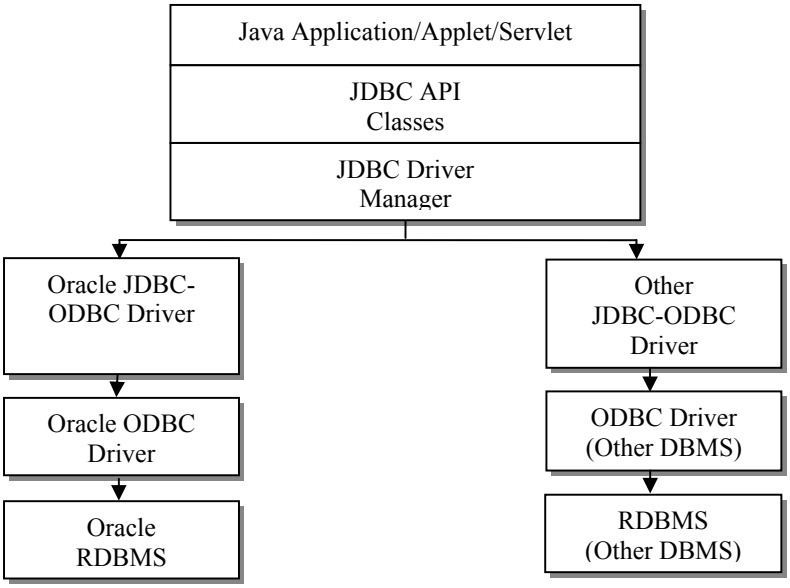


Figure 17-2: Type 1 JDBC drivers.

17.4.1.2 Type 2: Java–Native API adapter

These types of drivers make use of Java Native Interface (JNI) APIs to connect to the native database drivers. Native drivers and libraries need to be installed and relevant configuration setting need to be done at the client side. These drivers are more efficient than Type 1. For similar reasons as that of Type 1 drivers, these are also not very portable across databases. Figure 17-3 shows the architecture of Type 2 drivers.

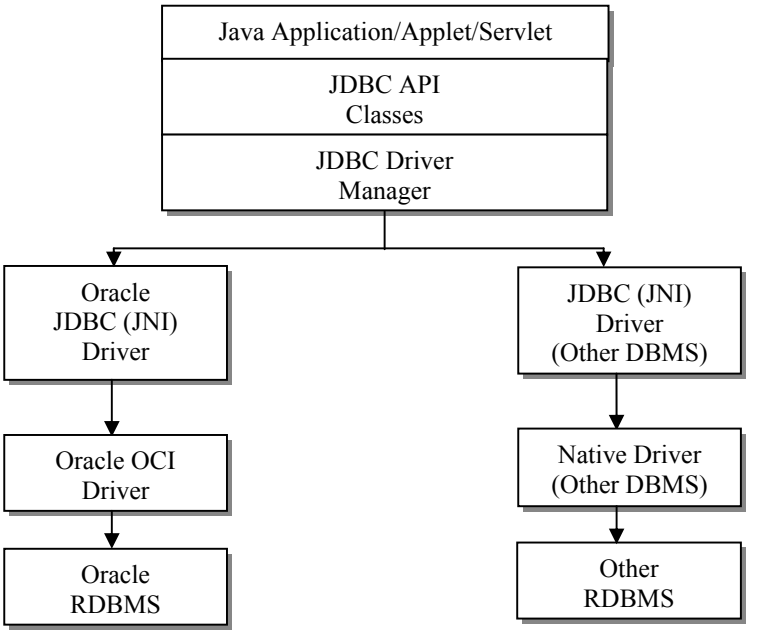


Figure 17-3: Type 2 JDBC drivers.

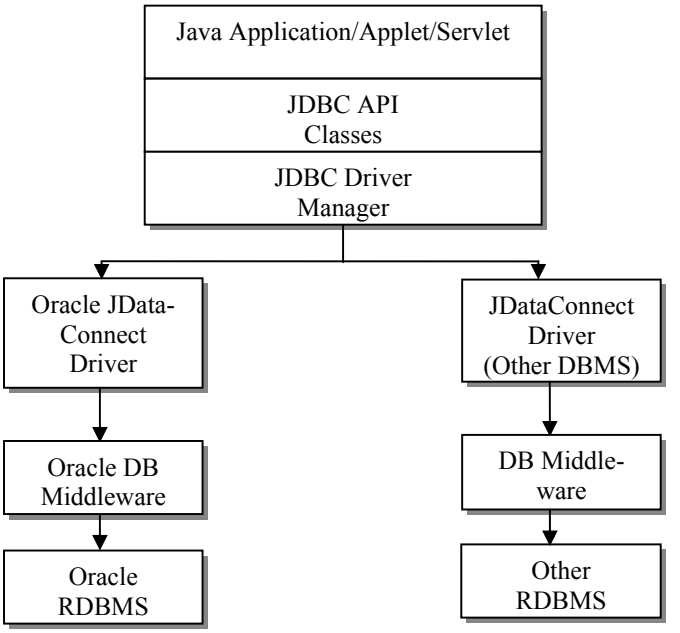


Figure 17-4: Type 3 JDBC drivers.

17.4.1.3 Type 3: JDBC-Net Protocol

These types of drivers use a pure JavacClient to connect to a server-side driver. They use standard network protocols to communicate to the DBMS server. No specific client-side libraries are required to be installed. Hence these are more portable than any other type of drivers. Figure 17-4 shows the architecture of Type 3 drivers.

17.4.1.4 Type 4: Pure Java

These drivers connect directly to the DBMS, using proprietary database protocols. Hence, they are the most efficient type of drivers. As they use database-specific protocols, they are not portable across DBMSs. Like Type 3 drivers, they also do not require any client side installation. Figure 17-5 shows the architecture of Type 4 drivers.

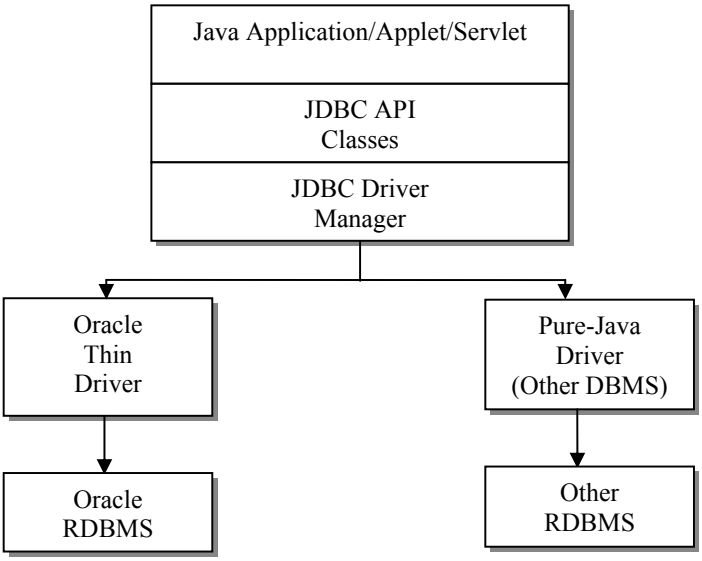


Figure 17-5: Type 4 JDBC drivers.

Table 17-1 summarizes the comparison between the four different types of drivers.

Table 17-1: Types of JDBC drivers.

	Type 1	Type 2	Type 3	Type 4
Portability across databases	Low	Low	High	Low
Efficiency	Low	Medium	Medium	High
Client side installation	Yes	Yes	No	No

17.5 JDBC APIs

Developing a typical database application using JDBC involve performing the following steps:

1. Establishing a connection.
    a. Loading a driver.
    b. Connecting to a DBMS.
2. Data manipulation.
    a. Creating a Statement object.
    b. Formatting an SQL statement.
    c. Executing the statement (CRUD operations).
    d. Closing the Statement and Connection.

17.5.1 Establishing a Connection

17.5.1.1 Loading the Driver

In order to establish a connection with a DBMS, the required JDBC driver needs to be first loaded. This is done by using the `DriverManager` class found in the package called `java.sql`. There are two ways in which this can be done. The first, using the Reflection API class called `Class`. The second, using the driver system properties to determine the exact Driver to be loaded.

Using Reflection APIs: The driver can be loaded dynamically by specifying the name of the driver class in the reflection API's method called `forName()` in the class called `Class`. The class loader will look for the driver class in its `classpath`.

For example, if you are using the MySQL database, you may copy the jar file containing the driver class (e.g.: `mysql-connector-java-3.1.12-bin.jar`) into the JDK's `jre\lib\ext\` directory. For instance, if your JDK is installed at `C:\Program Files\Java\jdk1.5.0_03` directory, the driver class may be copied at `C:\Program Files\Java\jdk1.5.0_03\jre\lib\ext`. This directory is automatically included in the `classpath`. The driver class in the `mysql-connector-java-3.1.12-bin.jar` file is included in the package called `com.mysql.jdbc`. The statement shown below, will dynamically load the driver class into the memory.

```
Class.forName("com.mysql.jdbc.Driver").newInstance ();
```

The statement uses the reflection API class called `Class`. This class contains a method called `forName()`. The parameter specified in this method identifies the name of the class to be loaded. The `newInstance()` method of the loaded class will then create an instance of the class and load it in the memory.

Alternatively, you can load the driver by creating an instance of the class and registering it with the driver as shown below:

```
Driver driver = new com.mysql.jdbc.Driver();
DriverManager.registerDriver (driver);
```

Using System properties: By specifying the System property called `jdbc.drivers` in the command line, during execution of the database application, we can get the Driver class to load the specified drivers dynamically during run-time.

```
$java –Djdbc.drivers=com.mysql.jdbc.Driver MyApplication
```

17.5.1.2 Connecting to the DBMS

As DBMSs can be located across the network, database applications connect to them using their Universal Resource Locator (URL). In other words, JDBC Connections are identified using an URL. Although URL formats may differ from DBMS to DBMS, they all have the essential parts that identify the JDBC Connection uniquely; viz. location of the DBMS (host and port numbers), connection protocol designator and the name of the database. An example of a URL for the MySQL DBMS running on the localhost is as follows:

```
URL: "jdbc:mysql://localhost:3306/florist";
```

"`jdbc:mysql`" specifies the connection protocol designator, "`localhost:3306`" specifies the hostname and port number and "`florist`" is the name of the database.

The connection can be established by using the `getConnection()` method of the `DriverManager` as shown below:

```
try {
  String dbUrl "jdbc:mysql://localhost:3306/florist";
  String userName = "florist";
  String password = "password";
  Connection conn = DriverManager.getConnection(dbUrl, userName, password);
} catch (SQLException e) {
    System.err.println ('Error during Connection");
}
```

If the `DriverManager` is unable to establish the connection, it will throw the `SQLException`.

Note: Prior to making a connection, a user account and a database must be created in the DBMS using the admin tools. For example, in MySQL, it is done by using the MySQL Administrator tool to add a new Connection.

17.5.2 Data Manipulation

After the connection to the database is established, the data can be retrieved, inserted, updated, or deleted from the database. This is done by using the `Statement` object of the `java.sql` package.

17.5.2.1 Creating Statement Objects

The `createStatement()` method of the connection object is used to create a statement as shown below:

```
Statement stmt = conn.createStatement();
```

`SQLException` will be thrown if the creation fails.

This object is used to send SQL statements to the database. There are three types of Statement objects as shown in Table 17-2.

Table 17-2: JDBC Statement types.

Statement Type	Purpose
`Statement`	Used to execute simple SQL statements without parameters
`PreparedStatement`	Used to reuse an SQL statement by passing different parameters
`CallableStatement`	Used to execute a stored procedure in the database

17.5.2.2 Executing SQL Statements

After the `Statement` is created, the `Statement` object is used to execute SQL statements on the database. For example, the following statement will retrieve the name and description of a Product whose id is `p001`.

```
ResultSet rSet = stmt.executeQuery("SELECT name, description " +
                                   "FROM ProductTable WHERE id='p001'");
```

Note: It is assumed that a table called `ProductTable` is already created in the database. Some of the columns in the `ProductTable` are named as 'name', 'description' and "id."

The `ResultSet` object that is returned is like a cursor to a `Collection` of results. So, in order to access the retrieved values, the cursor needs to be positioned at the first object. This is done by using the `rSet.next()` method. And the column values can be retrieved from the `ResultSet` object, using the `getXXX()` methods. The `XXX` represents the type of column.

```
rSet.next();
String name = rSet.getString();
String description = rSet.getString();
```

Other data manipulation statements are shown in Table 17-3.

Table 17-3: Statement execute methods.

Statement Execute Methods	Purpose
`executeQuery`	For retrieving single result-set, e.g., using SELECT SQL statement.
`executeUpdate`	For modifying the database, e.g., INSERT, UPDATE, DELETE, CREATE TABLE and DROP TABLE SQL statements.
`execute`	For statements that return multiple result-sets and/or update counts.

17.6 Data Definition Language (DDL) with JDBC

In this section you will learn how to perform the Data Definition Language operations such as creating and dropping tables using JDBC.

17.6.1 Creating a Table

In order to create tables, the user must have permission to do so. This is done by the administrator granting permission to the user to create tables in that database, after which the database program can create tables by specifying the user name. The `executeUpdate()` method of the `Statement` class is used to specify the `CREATE` SQL statement. Listing 17-1 illustrates creating a table called `ItemTable` in the database called `florist`.

```
import java.sql.*;
public class CreateTable {
  public static void main (String[] args){
    Connection conn = null;
    try {
      String userName = "florist";
      String password = "password";
      String url = "jdbc:mysql://localhost:3306/florist";
      Class.forName("com.mysql.jdbc.Driver").newInstance ();
      conn = DriverManager.getConnection(url, userName, password);
      Statement stmt =conn.createStatement();
      String tableName = "'florist'.'ItemTable'";
      stmt.executeUpdate ("CREATE TABLE " + tableName +
                          " ('Id' varchar(255) NOT NULL, " +
                          "'name' varchar(255) default NULL, " +
                          "'description' varchar(255) default NULL, " +
                          "'quantity' int(5), PRIMARY KEY ('Id'));");
    } catch (Exception e) {
        System.err.println (e.getMessage());
    }
  }
}
```

Listing 17-1: Creating tables using JDBC.

The highlighted statement executes the `CREATE` SQL statement. The schema of the table is provided in the SQL statement. There are four columns in the table, namely, `Id`, `name`, `description`, and `quantity`. The primary key of the table is the `Id` column. The primary key is used by the DBMS to identify a record uniquely.

In general, the syntax of a `CREATE SQL` is as follows:

```
CREATE TABLE <table>(<column type> [, <column type>]...)
```

A column type is of the form:

```
<column-name> <type> [DEFAULT <expression>]
       [<column constraint> [, <column constraint>]...]
```

A column constraint is of the form:

```
NOT NULL or UNIQUE or PRIMARY KEY
```

The exact syntax can differ from one DBMS to the other.

17.6.2 Dropping a Table

In order to delete (drop) tables, the user must have permission to do so. This is done by the administrator granting permission to the user to drop tables in that database. After which the database program can drop tables, by specifying the user name. The `executeUpdate()` method of the `Statement` class is used to specify the `DROP` SQL statement. The highlighted statement in Listing 17-2 illustrates deleting a table called `ItemTable` in the database called `florist`.

```
import java.sql.*;
public class DropTable {
  public static void main (String[] args){
    Connection conn = null;
    try {
      String userName = "florist";
      String password = "password";
      String url = "jdbc:mysql://localhost:3306/florist";
      Class.forName("com.mysql.jdbc.Driver").newInstance();
      conn = DriverManager.getConnection(url, userName, password);
      Statement stmt =conn.createStatement();
      String tableName = "'florist'.'ItemTable'";
      stmt.executeUpdate ("DROP TABLE " + tableName + ";");
    }catch (Exception e) {
       System.err.println (e.getMessage());
    }
  }
}
```

Listing 17-2: Dropping tables using JDBC.

17.7 Data Manipulation Language (DML) with JDBC

In this section you will learn how to perform the Data Manipulation Language operations such as creating, retrieving, updating and deleting records (CRUD operations) from tables using JDBC.

17.7.1 Creating (Inserting) Records Using JDBC

Records (i.e., rows) can be inserted into database tables using the `executeUpdate()` method of the `Statement` class. The return value of the `executeUpdate()` method is the number of rows affected, that is, the update count.

```
String sqlStmt = "INSERT INTO 'florist'.'ItemTable' "
+ "VALUES ('p002', 'Rose', 'Beautiful Flower', 101);";

int updateCount = stmt.executeUpdate (sqlStmt);
```

The equivalent SQL statement is formatted as a `String` and passed to the `executeUpdate()` method. The resulting update count will be `1` in this case.

In general, the `INSERT` SQL statement has the following syntax:

```
INSERT INTO <table> [(<column> [, <column>]...)]
VALUES (<expression> [, <expression>]...)
```

The exact syntax can differ from one DBMS to the other.

17.7.2 Deleting Records Using JDBC

Rows can be deleted from database tables using the `executeUpdate()` method of the `Statement` class, as shown below.

```
String sqlStmt = "DELETE FROM 'florist'.'ItemTable' "
                                + "WHERE 'Id' = 'p001';";
int i = stmt.executeUpdate ( sqlStmt);
```

The equivalent SQL statement is formatted as a `String` and passed to the `executeUpdate()` method. The resulting update count will be 1 in this case.

In general a `DELETE` SQL has the following syntax:

```
DELETE FROM <table> WHERE <condition>
```

The exact syntax can differ from one DBMS to the other.

17.7.3 Retrieving Records Using JDBC

DBMSs are frequently used for retrieving information from tables. This is done by using SQL Queries. SQL Query statements can be executed by using the

`executeQuery()` method of the `Statement` class. The return value is a `Collection` of `ResultSet` objects. The `ResultSet` object can be thought of as a pointer or cursor to the retrieved value sets. For example, consider the following SQL statement being executed.

```
String sqlStmt = "SELECT 'Id', 'name' FROM 'florist'.'ItemTable'";
ResultSet rSet = stmt.executeQuery(sqlStmt);
while (rSet.next()) {
  System.out.println ("Id = " + rSet.getString("Id") + " Name = " +
                      rSet.getString("name"));
}
```

The result retrieved by the `executeQuery()` statement will be a query result table as shown below:

Id	name
p001	daisy
p002	rose
p003	lily
p004	jasmine

`rSet` is the cursor that initially points just before the first row of the retrieved result table. The method `rSet.next()` positions the cursor to the first element. Subsequent calls to the same method advances the cursor to the next element (if it exists).

The values of the columns can then be retrieved from the result set using the various `getXXX()` methods, depending on the column type.

17.7.3.1 Mapping of SQL Type to JDBC Type

Most SQL typed data can be retrieved from the result set using the `getXXX()` methods The mapping between SQL types and JDBC types is shown in Table 17.3.

Table 17-3: Mapping of SQL types to JDBC types.

SQL Type	JDBC Type
VARCHAR	String
CHAR	String
NUMERIC	BigDecimal
BIT	boolean
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
DATE	java.sql.Date
TIME	java.sql.Time

Corresponding to every JDBC type, there is a `get` method in the `ResultSet` class, for example, `getString()`, `getInt()`, `getDate()`, `getFloat()`, `getTime()`, and so on.

17.7.3.2 Query SQL Statements

Many complex query statements can be formulated using the SQL Query language. Some examples are shown below:

• Retrieving Ids of Items with description that matches the pattern "Beautiful":

```
SELECT Id
FROM 'itemtable'
WHERE 'description' LIKE 'Beautiful%';
```

• Retrieving Ids of Items with quantity greater than 100:

```
SELECT Id
FROM 'itemtable'
WHERE 'quantity' > 100;
```

• Retrieving Ids and names of Items with quantity less than 100. The retrieved list should be ordered in the ascending order of name.

```
SELECT Id, name
FROM 'itemtable'
WHERE 'quantity' < 100
ORDER BY name;
```

In general, a `SELECT` SQL has syntax similar to what is shown below. (Other clauses are available.)

```
SELECT [ALL | DISTINCT] <columns>
   FROM <table>
   WHERE <condition>
   LIKE <pattern>
   [ORDER BY <column> [ASC | DESC]
          [, <column> [ASC | DESC]]...]
```

The exact syntax can differ from one DBMS to the other.

17.7.4 Updating Records Using JDBC

Records can be updated into the database using the `executeUpdate()` method of the `Statement` class as shown below:

```
String sqlStmt = "UPDATE 'florist'.'ItemTable'
                 SET 'name' = 'freesia'
                 WHERE 'Id' = 'p001'";
int updateCount = stmt.executeUpdate(sqlStmt);
```

The equivalent SQL statement is formatted as a `String` and passed to the `executeUpdate()` method. The resulting update count will be 1 in this case.

In general a `UPDATE` SQL has the following syntax:

```
UPDATE <table>
   SET <column = {<expression> | NULL}
   [, <column = {<expression> | NULL}]...
   WHERE <condition>
```

17.7.5 Updatable Result Sets

JDBC 2.0 allows creating updatable result sets. This enables using the `SELECT` statement to do all CRUD operations. An example is shown in Listing 17-3 below:

```
import java.sql.*;
public class UpdatableRS {
  public static void main(String[] args) {
    Connection conn = null;
    try {
      String userName = "florist";
      String password = "password";
      String url = "jdbc:mysql://localhost:3306/florist";
      Class.forName("com.mysql.jdbc.Driver").newInstance();
      conn = DriverManager.getConnection(url, userName, password);
      Statement stmt = conn.createStatement(ResultSet.TYPE_FORWARD_ONLY,
                                            ResultSet.CONCUR_UPDATABLE);
      String sqlStmt = "SELECT * FROM itemtable WHERE `Id`='p001';";
      ResultSet rs = stmt.executeQuery(sqlStmt);
      while (rs.next()) {
        rs.updateString ("name", "Camillia");
        rs.updateRow();
      }
    } catch (Exception e) {
        System.err.println("Database Exception");
    } finally {
        if (conn != null) {
          try {
            conn.close ();
          } catch (Exception e) { }
        }
    }
  }
}
```

Listing 17-3: Illustrating the use of Updatable ResultSet.

The highlighted statement in Listing 17-3 creates a `Statement` object that can be traversed in the forward direction and is concurrently updatable. Using these constants in the `createStatement()` method will enable updates on the query `ResultSet` objects, as shown below:

```
ResultSet rs = stmt.executeQuery(sqlStmt);
while (rs.next()) {
  rs.updateString ("name", "Camillia");
  rs.updateRow();
}
```

Updatable Result Sets also allow Insertion and Deletion of records. Insertion of Records is illustrated below:

```
rs.moveToInsertRow();

rs.updateString("Id", "p005");
rs.updateString("name", "Hibiscus");
rs.updateString("description", "Hibiscus");
rs.updateInt("quantity", 100);

rs.insertRow();
```

`moveToInsertRow()` method of the `ResultSet` class will position the cursor to a new row, and hence prepares the result set object for inserting a new record. The column values are then filled up in the result set using the `updateXXX()` methods of the `ResultSet` class; where `XXX` is the type of column. `insertRow()` method of the `ResultSet` class will insert the new record in the position of the cursor.

Rows can be deleted as shown below:

```
ResultSet rs = stmt.executeQuery(sqlStmt);
rs.next();
rs.deleteRow();
```

`deleteRow()` method of the `ResultSet` class will delete the row at the cursor position.

17.7.6 Prepared Statements

The SQL statements that have been introduced until now were executed using JDBC one at a time. Many times, it may not be optimal to do so, especially, if the same SQL statement has to be executed several times.

JDBC APIs provide a special type of `Statement` class called `PreparedStatement`, which can be used to execute an SQL statement over and over again. Prepared statements are generally very efficient because they are compiled before sending them to the DBMS. This requires an SQL statement is supplied to a `PreparedStatement` when it is created. This prepared statement can then be executed many times by varying some parameters. An example of using a `PreparedStatement` is shown in Listing 17-4.

```
public class PreparedStmt {
  private Connection conn = null;
  private  void upDate(String desc, String Id){
    try {
      String sql = "UPDATE itemtable SET description = ? WHERE Id = ?;";
      PreparedStatement prepStmt = conn.prepareStatement(sql);
      prepStmt.setString(1,desc);
      prepStmt.setString (2, Id);
      prepStmt.executeUpdate();
    } catch (SQLException e) {
       System.out.println("Exception in Prepared Statement");
    }
  }
  public static void main(String[] args) {
    PreparedStmt p = new PreparedStmt();
    try {
      String userName = "florist";
      String password = "password";
      String url = "jdbc:mysql://localhost:3306/florist";
      Class.forName ("com.mysql.jdbc.Driver").newInstance();
      p.conn = DriverManager.getConnection(url, userName, password);
      p.update("Pretty Flowers", "p001");
    } catch (Exception e) {
       System.err.println ("Cannot connect to "
                                          + "database server"
                                          + e.getMessage());
    } finally{
       if (p.conn != null) {
         try{
           p.conn.close ();
         } catch (Exception e) {}
       }
    }
  }
}
```

Listing 17-4: Illustration of `PreparedStatement`.

The following string `sql` (extracted from Listing 17-4) formulates the `sql` statement to be used by the `PreparedStatement` object. The "`?`" specifies the parameters that can be varied during execution of this statement.

```
String sql = "UPDATE itemtable SET description = ? WHERE Id = ?;";
```

Each of the "`?`" in the `sql` string can be referred to by an integer starting with `1` in the order of their appearance in the SQL statement. In the `String sql` shown above, the first "`?`" stands for the value of the `description` column and the second "`?`" stands for the value of the `Id` column. These values are supplied as parameters in the `update()` method of the `PreparedStmt` class as shown in Listing 17-4.

The following statement in the Listing 17-4 shows the invocation of `update()` method of the `PreparedStmt` class. The value for the `description` column is "`Pretty Flowers`" and `Id` column is "`p001`."

```
p.update("Pretty Flowers", "p001");
```

Multiple updates can be performed by looping through a set of values as shown below:

```
String [] desc = {"Pretty Flowers", "For your eyes",
         "Violet Splash", "Rainbow display", "Red Roses"};
String [] Id = {"p002", "p004", "p006", "p008", "p010"};
for (int i = 0; i < 5; i++)
 p.update(desc[i], Id[i]);
```

17.8 Summary

In this chapter, we have learned the architecture of the Java Database Connectivity (JDBC) APIs and the different types of database drivers supported by JDBC APIs. JDBC APIs are used from Java programs to interact with any Relational Database Management Systems (RDBMS). We have learned how to use these APIs to create, delete and modify table definitions using the Data Definition Language (DDL) of the SQL. We have also learned how to manipulate tables to perform the create, retrieve, update and delete (CRUD) operations.

When a JDBC API returned the queried result, it stores it in a `ResultSet`. JDBC2.0 supports updatable `ResultSets`, using which other operations such as insert, update can be performed efficiently.

Queries can also be prepared before they are executed. We use `Preparedstatement` objects to do so. These statements can be parameterized to formulate the exact query with varying parameters.

17.9 Exercises

Create a database called `florist` in your favorite database management system (e.g., MySQL). Develop a class called `MemberApp`, which contains a `main` method. In this class do the following using the JDBC APIs:

- Load the driver.
- Open a connection to the `florist` database.
- Create a table called `florist.members`, with the following fields:
  - Name
  - Member Id
  - Address
  - PrivilegesStatus (ordinary, silver, gold)
- Populate the table by inserting a few records.
- Formulate an SQL Query to do the following and print the result on screen.

- Retrieve the Name and Member ID of all member.
- Retrieve the Name and Member ID of all persons with Gold Privilege Status.
- Retrieve all information of Names starting with “Chan.”
- Take the Member ID as the input from the console and update the address information of this member. Read the new address from the console.
- Take the Member ID as the input from the console and delete the member’s record from the table.

Index

A

Abstract classes, 57, 83
Abstract methods, 80, 83
Abstract window toolkit
  components, 204, 205
  events, 209–214
  frames in, 203
  Layout Managers, 206–208
  panels, 205–208
Abstraction, concept of, 23, 40, 104, 166
`accept()` method, 161
Access control specifiers, 73, 108, 111
acl.read ( )
acl.write ( )
`action()` method, 212
`add()` method, 188, 191
`addActionListener()` method, 221
`addAll()` method, 192
`AddAndPrint` class, 167, 168
`Animate` applet, 246–248
Anonymous class, 227–229
Apache Tomcat, 260. *See also* Tomcat
API. *See* Application Program Interface
API documentation, 136–138, 224
API libraries, 47, 48, 120
API packages, 136
`Applet` class, 237, 238, 251, 252
Applets. *See* Java applets
`Appletviewer`, 250
Application Program Interface, 43, 123, 137
Arguments, 3, 10, 83, 266
Arithmetic operators, 24, 25
`ArrayList` class
  declaring, 187, 188
  methods for, 188, 189
  traversing the, 189, 190
Arrays creation, 34
Arrays, sorting alogorithms, 196–199
Assignment operators, 27, 28
Associativity rules, and Java operators, 30
Attribute, in objects, 7, 8, 18, 22, 79
`AudioClip` object, 243, 244
AWT. *See* Abstract window toolkit
AWT components
  CheckboxGroup, 215
  Choice, 217
  custom made, 224–226
  dialog frames, 221–224
  label, 214
  List, 218, 219
  menus and menu items, 219–221
  `TextArea`, 216, 217

B

`binarySearch` method, 198, 199 AQ
`bind()` method, 293
Bitwise operators, 26
Block, declaration of, 32, 33
Boolean primitive, 18, 25
`break`-statement, 33
`Buffer` class, 173
`BufferedInputStream` class, 146, 147
`BufferedOutputStream` object, 145, 146
`BufferedReader` class, 46
`BufferedWriter` class, 148
Bundling. *See* Encapsulation
Button component, in AWT, 215
Byte, primitive type, 17
`ByteArrayInputStream` class, 146, 148
`ByteArrayOutputStream` class, 145, 148, 253
Bytecode. *See* Java byte code

C

CalculatorEngine class, 40, 44
`CalculatorFrame` class, 48
`CalculatorInterface` class, 45

case statement, 95
Catch block, 121, 122, 126, 128, 130, 132, 133, 291
CGI-BIN scripts, 259
Character stream classes, 148–150
CharArrayReader class, 148, 149
CharArrayWriter class, 148
Checkbox class, 209
Checkbox component, in AWT, 215
CheckboxGroup component, in AWT, 215
Choice class, 217, 218
Class attributes, 107
Class construct, 18–22, 36, 43
Class hierarchy diagram, 54
Class methods, 43
Class objects, 252
Classes
  classification, 52–54
  common properties of, 61, 62
  definition, 8
  generalization, 55
  hierarchical relationship, 54
  inheritance in, 62–80
  properties of, 103–106
  specialization, 56
Classification, of objects, 52–55
ClassLoader class, 252, 253
Client/Server communication, 158, 284
close() method, 143
Collection Interfaces, 186
Collections class, 196
Collections.sort method, 196
Common Object Request Broker Architecture (CORBA) and RMI, 284, 285
Comparable interface, 196
ComparableT interface, 197
Comparator interface, 197
Complement, 192, 193
Component class, 204
Concrete classes, 57
Conditional expression operator, 28
Constructors, 23
Consumer–producer synchronization, 171
containsAll() method, 192
continue-statement, 34
Contract inheritance, 80
Contract part, 79
Control-flow statements, 30, 31
Cookie operations, 272
Counter class, 13, 18–20
CREATE SQL statement, 305, 306
CREATE SQL statement, 305, 306
createStatement() method, 304, 310
CRUD operations, 302, 310
Custom serialization, 281–283

D

Data Definition Language (DDL) table creation and deletion, 305, 306
Data manipulation, 303
Data Manipulation Language (DML)
  record creation and deletion, 307
  record retrieval, 307
Database, 297. *See also* Java Database Connectivity
Database Management Systems (DBMS), 297, 303
DataInputStream class, 147
DataInputStream object, 162
DataOutputStream class, 147
defaultReadObject() method, 283
defaultWriteObject() method, 283
Delegation model, in JDK, 212–214
DELETE SQL statement, 307
Dialog class, 221–223
Display flicker, 247
dispose() method, 222
Distributed object computing, 284, 286
Distribution, SDK, 251, 260
DivideByZero exception, 123
doGet() method, 269, 270
doPost() method, 269, 270
doSomething() method, 291, 294
double, primitive type, 17, 18, 28
DriverManager class, 302
DROP SQL statement, 305, 306
Dynamic binding, 93, 96, 97
Dynamic web pages, and java servlets, 259

E

EmptyStack class, 123
Encapsulation
  advantages, 112–114
  bundling in, 112

definition, 111
trade-off, 115
Environment, 45, 46, 48, 123, 136, 237, 242, 251, 253, 260, 276, 287
`Event` class, 209
Event handling, 209–211
Event-driven programming, 48
`EventListener` interface, 212, 214
Events, in JDK, 212–214
`ExampleFrame` class, 203
Exception class, 120, 122
Exception handling
block, finalization of, 131, 132
definition, 121, 122
multiple, 125, 126
nested, 129
object finalization, 131
raising exceptions, 122, 123
regular, 127
semantics, 120, 121
stack object and, 123–125
subconditions in, 128
Exceptions in Java
`IOException`, 156
`UnknownHostException`, 156
Execute methods, for SQL statements, 304
Execution, 10, 23, 30, 31, 42–44, 65, 136 (96 instances)
`executeQuery()` method, 308
`executeUpdate()` method, 309
Expression statements, 30
`extends` keyword, 65
Externalizable interface, 284

F

`false`, boolean value, 18, 25
`File` class, 152
File descriptors, 138
File manipulation, in Java
file input, 142
file output, 143
printing, 144
`FileInputStream` object, 142
`FileNotFoundException` class, 142
`FileOutputStream` class, 143
`FileReader` class, 148, 149
`FilterInputStream` class, 146, 148
`FilterReader` class, 148, 149
`FilterWriter` class, 148
`final` attributes, 83
`finalize()` method, 131, 133
`finally` clause, 131
Finger, 156
`float`, primitive type, 17, 18, 28
Florist database, 305, 306
`for-each` construct, 186, 189
`fork()` function, 166
Formal parameter, 122
`forName()` method, 302
`Frame` class, 203, 213
`friendly`, access specifier, 109
`FullStack` Exception, 123, 124

G

Generalization, of classes, 55
Generic Class, 183–185
Generic programming
basic concept, 179, 183
classes for, 183, 184
methods in, 185
problems with, 180–182
GET HTTP form, 268
`GET` *`path`* command, 158
`get()` method, 174
`getAttribute()` method, 273
`getInputStream()` method, 157, 176
`getOutputStream()` method, 157, 176
`getRequest()` method, 163, 168
`getSelectedItems()` method, 219
`getSession()` method, 273
`getState()` method, 209
`getText()` method, 209
Graphical user interfaces, 165, 166
`Graphics` class, 243

H

`handleEvent()` method, 209
`HashMap` class
declaration, 194
methods available for, 195
`HashSet` class
declaration, 190
methods available for, 193
subset and intersection in, 191–193

HTML class, 238, 267
HTML form, 136, 266, 267, 269, 274, 275
HTTP. *See* HyperText Transmission Protocol
HTTP headers, traversing set of, 270
HTTP requests, 271
`HttpServlet`, 269
`HttpServletRequest` method, 266, 270
`HttpServletResponse`, 266
`HttpSession`, 273
HyperText Transmission Protocol, 158, 160, 168

I

`Image` class, 243
Implementation part, of inheritance, 79, 80
`import java.net.Socket` statement, 156
`import` keyword, 110, 111
`indexOf` method, 198
Information hiding, concept of, 11, 112, 114
Inheritance
  adding/deleting a class, 72–75
  chain in, 75–80
  class hierarchy, 67–72
  code reuse, 67
  definition, 62
  implementation, 64–67
  interface concept in, 80–88
`init()` method, 241, 246
Initialization, of object, 23
Input and output operation
  character classes, 148–150
  classes for, 138–141
  code reuse, 145, 146
  file input, 143
  file output, 144
  output, formatting of, 151
  token for input, 150
`InputStream` object, 140, 157, 176
`InputStreamReader` class, 148
INSERT SQL statement, 307
Instance variable, 18, 41, 163, 240, 245
`int`, primitive type, 17, 18
`Integer` object, 195
Interface construct
  abstract classes and, 84
  attributes in, 83
  methods in, 83
  multiple inheritance and, 80–82, 85–88
`interface` keyword, 80
Internal representation, 21, 112, 114, 140
Intersection, in `HashSet` Class, 192
`IOException` object, 122, 140, 156
`ItemListener` object, 214–216, 218, 219, 225
`Iterator` class, 189, 190

J

Java API
  networking and multithreading in, 155
  networking model in, 155, 156
Java applets
  animation in, 245, 246
  code, dynamic loading of, 250–253
  custom made, 242
  event handling in, 238
  HTML and, 238, 241
  life cycle of, 241, 242
  multimedia and, 243, 244
  parameters in, 248
  security issues, 253–257
  URL content, retrieval of, 240
Java Beans, 202
Java byte code, 44, 237, 256
Java Database Connectivity, 308
  architecture of, 298
  data definition language (DDL) with, 305
  data manipulation language (DML) with, 307
  database application using, steps in, 302
  drivers, 299, 300
  record creation and deletion, 307
  record retrieval, 307
  table creation and deletion, 305–306
  types, 308
  updatable result sets, 310
  updating records using, 309
Java Development Kit, 44, 136, 148
Java Development Kit 1.1, 148, 153, 202, 279
  events in, 212
Java Enterprise API, 285

Java Interface Definition Language (Java IDL), 285
Java Native Interface (JNI) APIs, 299, 300
Java programming language
  abstraction used in, 166
  concept of generics in, 183
  limitations, 285, 286
  object definition, 18
  packages in, 110, 111
  primitive data types, 17
  variable definition, 18, 19
Java servlets
  characteristics of, 266
  and dynamic web pages, 259
  output of, 271
  parameters and headers, 266–271
Java virtual machine, 4, 110, 121, 131, 237, 279, 280
Java, implementation of
  code execution, 42
  concept, 39, 40
  event driven programming in, 48
  methods for, 40–42
  user interface, 44–46
Java-enabled Web browser, 239, 251
`java.applet` package, 136
`java.applet.Applet` class, 237
`java.awt package`, 47, 241
`java.io` package, 145
`java.lang` package, 166
`java.net` package, 156, 176
`java.sql` package, 302, 303
`java.util` package, 150
`java.utils.Collections` class, 196
`javac` command, 237
`JButton` class, 231, 233
JDBC. *See* Java Database Connectivity
JDBC drivers, 298
  comparison of, 301
  types of
    Java–Native API adapter, 299, 300
    JDBC-Net protocol, 301
    JDBC-ODBC Bridge, 299
    Pure Java, 301
JDBC Statement types, 304
JDBC types, mapping of SQL types to, 308
JDBC-Net protocol, 300, 301
JDBC-ODBC Bridge, 299
JDK. *See* Java Development Kit
`JTable` class, 234
JVM. *See* Java virtual machine

K

`keytool.exe` program, 256

L

`Label` class, 231
Label component, in AWT, 214
Layout Manager, 202, 206
`LineNumberInputStream` class, 146, 148
`LineNumberReader` class, 148, 149
Linked list, 112
`List` interface, 187
Listener sockets, 161
`ListIterator` class, 190
`ListIterator` method, 190
Local class, 228, 236
Local variable, 19, 32, 36, 271, 274
Logical operators, 25
`long`, primitive type, 17, 26

M

`MalformedURL` exception class, 123
`Map` interface, 194
Memory de-allocation, 131
Menu class, 220
`MenuBar` class, 220
`MenuItem` class, 220
Message, 9, 10
Message passing, 10, 36, 135
Methods
  definition, 19, 20
  parts of, 79
  signature, 19, 24, 35, 71, 97, 789
`MixedComponents` class, 217
`modify()` method, 283
Modularity, 103–115
`MouseAdapter` class, 214, 226, 229
`MouseListener` class, 214, 225, 226
`moveToInsertRow()` method, 311
Multiple inheritance
  concept of, 76
  interface and, 80–82
  problems associated with, 77, 78

Multithreading, concept of, 166–175
MyAppletLoader class, 251, 252
MyExitButton class, 210
MyNetworkLoader class, 252, 253
MySQL database, 302, 303

N

Network-centric computing, 285
new operator, 34, 42
newInstance() method, 302
notify() method, 172, 175

O

Object (s)
  classification, 52
  creation of, 12, 13
  definition, 9, 18
  initialization, 23
  instantiation, 20
Object class, 182
Object de-serialization, 282
Object instantiation, 119, 161, 242, 251
Object Request Broker (ORB), 284, 285
Object serialization, 279. *See also* Custom serialization; Object de-serialization
  components in, 281
  sensitive data in, 283
Object-oriented programming
  basic concept, 1
  class definition, 7–9
  class generalization, 55
  Java and, 4
  message, definition of, 9, 10
  methods in, 10, 11
  object classification in, 52
  object definition, 7–9
ObjectInputStream object, 280
ObjectOutputStream, 280
Open Database Connectivity (ODBC), 299
Operation overloading, 97, 100
Operators. *See also* Operation overloading
  precedence, 29
  types, 24–28
OutputStream class, 145, 157, 176
OutputStreamWriter class, 148
Overloaded constructors. *See* Constructors

P

package keyword, 110
paint() method, 225, 245
Panel class, 241
Parameter, 22, 41, 71, 120, 122
performOperation( ) method, 122
PipedInputStream class, 146, 148
PipedOutputStream class, 145, 148
PipedReader class, 148, 149
PipedWriter class, 148
Platform independence, of Java, 4, 135
policytool.exe program, 256
Polymorphism, 100–102
POST HTTP form, 268, 269
Precedence. *See* Operators
Prepared statements, 311, 312
PreparedStmt class, 312
printarray method, 185, 186
println() method, 139
PrintStream class, 139
  printing using, 144
PrintWriter, 271
Private data field, 283
private keyword, 21, 73
protected keyword, 109, 116
Proxy server, 176
public keyword, 21, 83
Pure Java Driver, 301
push() method, 113
PushbackInputStream class, 146, 148
PushbackReader class, 148, 149
put() method, 174
Puzzle class, 230

R

RandomAccessFile class, 152, 153
read() method, 135
Reader classes, 173
readLine() method, 45, 46
readObject() method, 281, 283
rebind() method, 293
Receiver, 3, 9, 10, 100

Records
insertion and deletion of, 307, 311
retrieval of, 307
Reflection API, 302
Relational operators, 26
Remote Method Invocation (RMI), 286, 289
architecture of, 289–291
and CORBA, 285
deployment, 293, 294
example of, 288
remote reference layer (RRL) of (*see* Remote Reference Layer (RRL))
steps for creating, 287
stub/skeleton layer of, 289
transport layer of, 290, 291
Remote object, 284, 287, 290, 291
Remote procedure call (RPC), 284, 286
Remote Reference Layer (RRL), 289
client and server side, 290
`RemoteException`, 293
`removeAll()` method, 193
`request()` method, 158
`ResultSet` class, 309, 311
`return`-statement, 35
`returnResponse()` method, 163, 168
`rmic` compiler, 288, 293
`run()` method, 166–168, 176
Run-time type identification (RTTI), 182
`Runnable` interface, 176

S

`Scanner` class, 150
Security issue, in Java, 255–257
SELECT statement, 305, 308, 310
Sender, 3, 9, 19, 35, 100
`SequenceInputStream` class, 146
Serializable interface, 281
Serialization
custom, 281–283
object, 279
`Serialize` class, 280
Server. *See* Client/Server communication
`ServerImpl` class, 294
`ServerSocket` class, 161, 176
`service()` method, 266
Servlet-mapping, 275
Servlets. *See* Java servlets
Session handling, 271–273
Session management, 273
Session timeout, 274
`Set` interface, 190
`setAttribute()` method, 273, 274
`setMenuBar()method`, 220
`setModal()` method, 222
`setMultipleMode()` method, 218
Shared variable, 106, 169, 274
`short`, primitive type, 17, 26
Single inheritance, 75–77, 79, 80
Single thread of execution, 166
Smalltalk-80, 13
`Socket` class, 156, 157, 161, 176
Socket programming
client side, 157–160
code execution, 164, 165
server side, 161–163
Software engineering, 40, 75, 120
`SortedMap` interface, 187
`SortedSet` interface, 187
Sorting algorithm, for arrays, 196–199
Specialization, of classes, 54, 56
SQL Query statements, 307, 309
SQL statements, execution of, 304, 305
SQL types, 308
`SQLException` class, 303
`Stack` class, 180, 182
`StackE` class, 184
`StackApp` class, 185
`StackItem` class, 114
`start()` method, 168, 176
State, of the object, 7, 8, 282
`Statement` class, 305–309, 311
Static binding, 93–95
`static` method, 43, 83, 108
`static void main()` method, 159, 169, 291
`stderr`, 138
`stdout`, 138
`String` class, 182, 186
`StringBufferInputStream` class, 146, 148
`StringReader` class, 148, 149
`StringTokenizer` class, 163
`strList` object, 188
Subclass, definition, 53, 54
Subset, and `HashSet` class, 192
Superclass, definition, 53, 54

Swing components, JDK 1.2
AWT, transition from, 231–234
Swing `JFrame`, 233
`switch` statement, 97
`Synchronized` statement, 170
`synchronized tag`, 175
`System` class, 138
System privileged services, 156
`System.err` object, 138, 140, 153
`System.in` object, 138, 140, 153
`System.out.println()` method, 37 , 43

T

Table creation and deletion, 305, 306
TCP sockets, 266
TCP-based transport, 289, 291
TCP/IP socket connections, 156
`TextArea` component, in AWT, 21, 217
`TextComponent` class, 209
`TextField` class, 239, 241
Thread creation, methods for
`Runnable` class, 166, 168, 176
using `Thread` class, 166, 167
Thread synchronization, 169–175
`throw`-statement, 120
`throws Exception` statement, 46
Tomcat
binary distribution, 260
configuration of, 261
downloading and installation, 260
home directory of, 261
log window, 262
sample servlet in, 263
compilation of, 264
servlet tag for, 264
servlet-mapping tag for, 265
session timeout of, 274
setting up paths for, 262
start-up page of, 263
starting and stopping, 262
`toString()` method, 140
Transform servlet, 275
Transient data field, 283
`TransmissionError` exception object, 121–123
`true`, boolean value, 18, 25, 26, 28
Trusted applets, 255
`try`-block, 120–123, 125, 129, 131
Typecast operator, 28, 29, 140

U

Uniform Resource Locator (URL), 158, 303
Union, in `HashSet` class, 192
UNIX
convention, 156
operating systems, 166
`UnknownHostException`, 156
`unModify()` method, 283
Untrusted applets, 253
Updatable ResultSet, in JDBC, 310
UPDATE, SQL statement, 309, 310

V

Variable definition, 18

W

`wait()` method, 172, 175
Web browser, 136, 165, 177, 238–242, 245
Web client/server communication, 158, 161
WebRetriever class, skeleton of, 157
WebServe
instances in, 165
multithreading in, 169
Web server
codes for, 164
processes requests for, 168
`while`-statement, 31
`WindowAdaptor` class, 48
`windowClosing()method`, 48
`write()` method, 144
`writeObject()` method, 281
`Writer` classes, 173