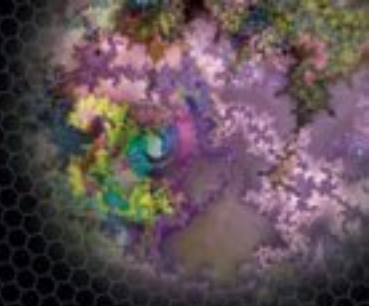
THE EXPERT'S VOICE® IN JAVA



SCJD Exam with J2SE 5

Master the practices and core concepts necessary to pass the Sun Certified Java Developer (SCJD) exam

SECOND EDITION

Andrew Monkhouse and Terry Camerlengo

Apress®

SCJD Exam with J2SE 5

Second Edition

Andrew Monkhouse and Terry Camerlengo

Apress®

SCJD Exam with J2SE 5, Second Edition

ISBN (pbk): 1-59059-516-5

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jason Gilmore
Technical Reviewer: Jim Yingst
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Project Manager: Beth Christmas
Copy Edit Manager: Nicole LeClerc
Copy Editor: Liz Welch
Assistant Production Director: Kari Brooks-Copony
Production Editor: Lori Bring
Compositor: Dina Quan
Proofreader: Elizabeth Berry
Indexer: John Collin
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section. You will need to answer questions pertaining to this book in order to successfully download the code.

Contents at a Glance

About the Authors xi
About the Technical Reviewer xiii
Acknowledgments xv
Introduction xvii

PART 1 Introduction and General Development Considerations

CHAPTER 1 Introduction 3
CHAPTER 2 Project Analysis and Design 11
CHAPTER 3 Project Overview 57

PART 2 Implementing a J2SE Project

CHAPTER 4 Threading 71
CHAPTER 5 The DvdDatabase Class 119
CHAPTER 6 Networking with RMI 163
CHAPTER 7 Networking with Sockets 199
CHAPTER 8 The Graphical User Interfaces 225

PART 3 Wrap-Up

CHAPTER 9 Project Wrap-Up 295

INDEX 325

Contents

About the Authors xi
About the Technical Reviewer xiii
Acknowledgments xv
Introduction xvii

PART 1 Introduction and General Development Considerations

CHAPTER 1 Introduction 3

J2SE 5 3
The SCJD Exam 4
The Certification Process 4
Downloading the Assignment 5
Documentation and Questions 5
Who Should Read This Book 6
About This Book 6
Setting Up the J2SE 5 JDK and Environmental Variables 8
Summary 8
FAQs 8

CHAPTER 2 Project Analysis and Design 11

Implementing a Project 11
Getting Started 12
Gathering Requirements 12
Using Accepted Design Patterns 14
Documenting Design Decisions 15
Testing 15
Organizing a Project 16
High-Level Documentation 17
Design Decisions Document 18

Java Coding Conventions 19
Naming Conventions 20
File Layout 22
Source Code Formatting 24
Formatting of Comments Within the Code 28
Suggested Coding Conventions for New Features in JDK 5 29
Javadoc 35
Coding Conventions 36
Working with Packages 44
Best Practices 47
Writing Documentation As You Go 47
Assertions 49
Logging 50
Summary 54
FAQs 54

CHAPTER 3 Project Overview 57

What Are the Essential Requirements for the Sun Certification Project? 57
Introducing the Sample Project 59
Application Overview 63
Summary 66
FAQs 66

PART 2 Implementing a J2SE Project

CHAPTER 4 Threading 71

Threading Fundamentals 71
A Brief Review of Threads 72
Multithreading 73
Java's Multithreading Concepts 73
Locks 87
Locking in JDK 5 96
Locking Summary 98
Understanding Thread Safety 98
Deadlocks 98
Race Conditions 100
Starvation 102
tions 104

Thread Safety Summary 106
Using Thread Objects 106
Stopping, Suspending, Destroying, and Resuming 106
Thread States 107
More on Blocking 108
Synchronization 111
Multithreading with Swing 113
Threading Best Practices 114
Summary 116
FAQs 116

CHAPTER 5 The DvdDatabase Class 119

Creating the Classes Required for the DvdDatabase Class 119
The DVD Class: A Value Object 119
Discussion Point: Handling Exceptions Not Listed in the Supplied Interface 126
The DvdDatabase Class: A Façade 134
Accessing the Data: The DvdFileAccess Class 137
Discussion Point: Caching Records 148
The ReservationsManager Class 148
Discussion Point: Identifying the Owner of the Lock 150
Creating Our Logical Reserve Methods 154
The Logical Release Method 155
Summary 160
FAQs 160

CHAPTER 6 Networking with RMI 163

What Is Serialization? 164
Using the serialver Tool 165
The Serialization Process 166
Customizing Serialization with the Externalizable Interface 169
Introducing RMI 171
The Delivery Stack 173
The Pros and Cons of Using RMI as a Networking Protocol 174
The Classes and Interfaces of RMI 175
What Is an RMI Factory? 177
Summary 196
FAQ 196

CHAPTER 7 Networking with Sockets 199

Socket Overview 199
Why Use Sockets 200
Socket Basics 200
Addresses 200
TCP and UDP Sockets Overview 201
TCP Socket Clients 203
The DvdSocketClient 205
Socket Servers 212
Multicast and Unicast Servers 212
Multitasking 212
The Server Socket Class 213
The Application Protocol 218
Summary 222
FAQs 222

CHAPTER 8 The Graphical User Interfaces 225

GUI Concepts 226
Layout Concepts 227
Human Interface Concepts 228
Model-View-Controller Pattern 234
Why Use the MVC Pattern? 234
MVC in Detail 234
Benefits of MVC 236
Drawbacks of MVC 236
Alternatives to MVC 237
Swing and the Abstract Windows Toolkit 237
Layout Manager Overview 237
Look and Feel 241
The JLabel Component 244
The JTextField Component 245
The JButton Component 248
The JRadioButton Component 249
The JComboBox Component 251
The BorderFactory 251
The JTable Component 254
The TableModel 255
Using the TableModel with a JTable 259
The JScrollPane 260

Bringing Denny's DVDs Together 262
Application Startup Class 262
The Client GUI 263
Specifying the Database Location 273
The Server GUI 286
Swing Changes in J2SE 5 289
Improve Default Look and Feel of Swing 289
Skins Look and Feel 290
Adding Components to Swing Containers Has Been Simplified 290
Summary 291
FAQs 291

PART 3 Wrap-Up

CHAPTER 9 Project Wrap-Up 295

Thread Safety and Locking 296
The Choice Between RMI and Sockets 296
Benefits of Using a Serialized Objects Over Sockets Solution 297
Benefits of Using an RMI Solution 299
The MVC Pattern in the GUI 300
Locating the Code Samples 301
Compiling and Packaging the Application 301
Creating a Manifest File 303
Running rmic on the Remote Package 304
Packaging the Application 305
Running the Denny's DVDs Application 306
Running the Client Application in Stand-alone Mode 307
Running Denny's DVDs Server 307
Running the Client Application in Networked Mode 309
Testing 309
Packaging Your Submission 318
Summary 321
FAQs 322

INDEX 325

About the Authors

ANDREW MONKHOUSE is a moderator on the JavaRanch web site, currently moderating the SCJD and SCJA forums.

Andrew has passed SCJP 1.2, SCJP 1.4, SCJD, SCWCD, SCBCD, and Part I of SCEA. He has been working with computers for too long (his first program was written on mark-sense cards, which are similar to punch cards).

Andrew has worked in a number of positions from programmer, to architect and IT manager, working on VMS, Unix, Macintosh, and Microsoft operating systems. He's built back-end, middleware, and front-end solutions for a variety of industries. Andrew is an Australian at heart, although he is frequently in other countries for work purposes.

TERRY CAMERLENGO has over 9 years of software engineering experience from numerous corporations, including Fortune 500s and dot-coms. He is experienced in all phases of the software life cyle, with a focus on object-oriented technologies such as Java, C#, C++, and .NET. His expertise includes front-end web design, server-side enterprise development, and relational database modeling and development. Terry holds both Sun and Microsoft certifications, and graduated with a degree in computer science and philosophy from Ohio State University. Currently Terry works for Ohio State University's James Cancer Center in the Biomedical Informatics department as a senior developer and research specialist and is pursuing advanced studies in computational biology.

About the Technical Reviewer

JIM YINGST studied engineering physics at the University of Arizona, but after graduating he got sucked into the IT job market instead because, well, it seemed like a good idea at the time. He now roams the West helping tech companies find solutions to their IT problems.

Jim is a sheriff (administrator) and longtime contributor at `www.javaranch.com`, where his duties include answering Java questions, redirecting off-topic posts, and dealing with troublesome Australians.

He seems to spend most of his free time obsessively visiting bookstores. On rare occasions he actually reads the stuff he buys, mostly science fiction. The rest of the time he's probably listening to obscure progressive rock bands or finding new Thai restaurants. Jim lives in Boulder, Colorado.

Acknowledgments

We would like to thank the following people:

- Mehran (Max) Habibi, who did so much work on the first edition and in getting this edition started, as well as introducing us to each other and to the Apress staff
- Our technical editor Jim Yingst, who not only verified our writing but made so many wonderful suggestions
- The fantastic staff at Apress for working with us on this project, and taking our raw work and producing a polished publication
- Each other

Without the help of all these people, this book would not be anywhere near as good as it is. We would also like to thank family, friends, and colleagues who put up with our bouncing between being totally unsociable when there were deadlines to meet, and desperately trying to catch up with everyone in the quiet times.

Andrew Monkhouse and Terry Camerlengo

Introduction

The Sun Certified Developer for the Java 2 Platform assignment offers a unique opportunity for Java developers to put their Java skills to practical use without requiring any specific development or runtime environment. The assignment also provides a great learning environment as many different APIs can be used, and many alternative solutions can be provided. This book introduces many of the concepts you will need to know in order to pass the SCJD assignment.

Many developers are a little daunted by the scope of the assignment, as it covers everything from a back-end database, a server application, a front-end application, API documentation, and user documention. This book covers each section in detail, helping you gradually build up your knowledge of each topic while working toward a sample project. This book will also introduce you to the new features of JDK 5, providing contextual usage of the new APIs and features within our sample project.

The Sun assignment deliberately does not specify an operating system platform or development environment to be used—all that is needed is a computer capable of running a current version of the JDK. Throughout this book we have used JDK 5 on Windows 2000. Since this book introduces JDK 5 features, and uses them throughout our sample project, you will need JDK 5 to run our sample applications; however, the sample application is not dependent on Windows 2000.

We hope you enjoy this book, and we look forward to hearing that you passed and any comments you may have on this book. You can contact both authors at `scjd@apress.com`.

PART 1

Introduction and General Development Considerations

CHAPTER 1

Introduction

Welcome to *The Sun Certified Java Developer Exam with J2SE 5, Second Edition*. By taking advantage of the new features of J2SE 5, passing the Sun Certified Java Developer (SCJD) exam is easier than ever before. Features such as generics, the enhanced for loop, autoboxing and unboxing of primitives, the new concurrency classes, and other new capabilities offer developers a richer and more robust tool set than ever before. This book and the accompanying sample project will help you acquire the understanding necessary to pass the SCJD examination while learning the finer points of the J2SE Development Kit (JDK 5). If you have been meaning to take the SCJD exam or you are ready to further explore the mysteries of Java, you have found the right book.

The best way to learn a new skill is to use it. This is true in tennis, pottery, and yes, programming. With that principle firmly in mind, this book helps you learn about J2SE 5 while detailing the strategies, skills, and information needed to pass the SCJD exam. Sun Microsystems designed the SCJD exam to be a realistic example of what a professional Java developer can expect to encounter in the real world. The SCJD exam covers a large portion of J2SE, including Remote Method Invocation (RMI), threading, file input/output (I/O), and Swing.

The sample project, Denny's DVDs (introduced in Chapter 3), is designed to explore the same concepts that the SCJD exam does. Unlike on the SCJD exam, however, the underlying concepts are explained in detail. When you have finished this book, you will have learned the skills necessary to take and pass Sun's exam.

The two major topics discussed in this chapter are

- Finding out how to download and take the SCJD exam
- Understanding the goals of this book

J2SE 5

J2SE 5 is a major update, designed to improve ease of development, increase scalability, provide for additional monitoring and manageability, and enhance the Java desktop clients. While J2SE 5 offers a slew of new and exciting features, this book focuses on bread-and-butter topics such as threading, RMI, Swing, sockets, exception chaining, logging, and serialization. Once you understand the foundations, everything else follows naturally.

The SCJD Exam

The SCJD certification is a comprehensive test used by Sun to verify the skills of advanced Java programmers. It is generally considered a strong benchmark of competence. This book focuses on the features of J2SE 5 that are relevant to this exam. Adequate preparation is essential to pass the exam. Due to its difficulty, the opportunity to take the SCJD exam is only available to programmers who have already passed the Sun Certified Java Programmer (SCJP) exam. Fortunately, this book explains the concepts you need to know to pass the SCJD exam.

■ **Note** Three Sun Certified Java Programmer exams are currently available: one for certification on each of the J2SE 1.2, J2SE 1.4, or J2SE 5.0 platforms, respectively. While the questions for each of these certifications differ slightly (for example, the J2SE 1.2 exam has Abstract Windowing Toolkit [AWT] questions), you only need to be certified in any one of these three certifications in order to be eligible for the SCJD certification.

The Certification Process

The SCJD certification process consists of two parts. The first part is an assignment consisting of a custom-designed sample project with a sample data file, an interface to be implemented, and specific requirements. You complete the assignment and return it for grading. Information on how to register for the assignment and download the assignment instructions and data file are presented in the next section of this chapter.

When you have completed the assignment and returned it for grading, you move on to the second part of the certification process, which is a written test designed to confirm that you wrote the assignment you submitted, and to investigate the understanding that led to the design and implementation decisions made during the first part of the exam. You may take up to 90 minutes to complete the written exam; however, since there are currently only four questions on the exam, you should find that you have more than enough time to complete it.

It is not possible to have notes or material with you for that exam. As such, it is best to take the written exam as soon as possible after you have submitted your assignment, while all the details are still fresh in your mind. Both sections are graded at once, even though the coding section is collected first. This means that the second exam will ask generic questions rather than specific ones about your individual project. You need a holistic understanding to pass both parts of the SCJD exam.

■ **Caution** The assignment you submit will not be passed to an assessor for grading until *after* you have completed the essay exam. If you do not take the essay exam, you will not receive any warning that your assignment is not being assessed—it will just sit in limbo until you finally do take the essay exam.

The goal of the SCJD exam is to validate your understanding of the most important Java skills, including threading, RMI, sockets, serialization, file I/O, and Swing. Each assignment project can be unique, testing these features to different degrees. For example, while you must write a server capable of handling concurrent requests, the interface provided might change which classes you allow multiple threads to run on. Or your requirements could call for strict search requirements versus more general searching ability. This book gathers together everything you need to know about all the relevant topics and integrates in the relevant changes in J2SE 5.

Sun also requires you to use a current JDK (one that has not been superseded by a newer JDK by more than 18 months) for developing your solution, ensuring that candidates stay current with the latest features of the JDK. A list of release dates may be found at `http://java.sun.com/j2se/codenames.html`.

Downloading the Assignment

You can register for the assignment and examination by visiting `http://www.sun.com/training/certification/java/java_devj2se.html` in the United States. Many other countries also allow online registration—you can view contact details at `http://www.sun.com/training/world_training.html`. After paying for the assignment, you will receive an e-mail from Sun telling you exactly how to download the Java archive (JAR) files that contain the assignment instructions. Receiving this e-mail may take a few days, or it may happen that same day. As soon as you receive the JAR files, make a couple of copies and store them safely. It is very expensive to get a second copy of the assignment to match the subject of this section.

■ **Tip** The Sun Education web site lists the web address where you may download your assignment. You may be able to download your assignment before you have received an e-mail specifying that your account has been configured for downloading.

Documentation and Questions

You are probably going to have questions regarding the requirements of the exam. Generally, Sun will not answer these questions. This may be because they want to see how well you can choose between different solutions (and describe why you made your decisions). It may be because they are trying to emulate real-world conditions where the client is not always willing to communicate. It may even be because answering questions for each test applicant is an untenable task. In any case, it is very important that you articulate your questions and deal with them in the documentation you must create as part of your assignment deliverables. If nothing else, you should document your assumptions and choices. For more help, we suggest using the excellent resources available at JavaRanch (`http://www.javaranch.com`) and the various helpful Java certification groups on Yahoo.

Chapter 2 provides good suggestions on how to work with Javadoc-style comments and offers some industry best practices. Don't use outlandish naming conventions or even Hungarian notation. If possible, use whatever style the material itself uses. As far as the SCJD exam is concerned, Sun really wants you to color inside the lines.

Who Should Read This Book

This book is for the working Java professional who needs an introduction to J2SE 5 and has an eye toward learning the material needed to pass the Sun Certified Java Developer exam. The SCJD exam gives programmers a slice of what they can expect on a real-world assignment, and you have to be ready for that challenge. A developer who has passed, or could pass, the Sun Certified Java Programmer (SCJP) exam will feel at home here. A developer with less than six months of experience should probably supplement this book with some of the excellent Java books available from Apress or other publishers.

This book describes in detail many features of the JDK, some of which have been part of standard Java for many years, and some of which have only been introduced in JDK 5. The only assumption we have made in this book is that the reader will be familiar enough with Java to pass the SCJP exam—so we do not need to spend time explaining the basics of the language (for example, the difference between an `int` and a `long`). However, we do go into details of changes to the language, so candidates who have not yet learned the JDK 5 language enhancements can discover them here.

About This Book

This book addresses the SCJD certification, which is one of several Java certification exams offered by Sun Microsystems. The SCJD and SCEA (Sun Certified Enterprise Architect) certificates require candidates to complete projects, whereas the other certifications only require theory-based exams in which the candidate typically has multichoice questions to answer. As far as programming goes, the SCJD exam is the most challenging of the exams that Sun offers, and that is precisely why it is the focus of this book.

This book is divided into three parts. Part 1 focuses on general development considerations and outlines a sample project. Part 2 teaches necessary concepts from the ground up, while facilitating both understanding and implementation. Part 3 concludes the book with a discussion of design and implementation decisions made and possible alternative paths.

A sample project is provided that offers challenges similar to those you'll find on the SCJD exam while introducing and taking advantage of the relevant new features of J2SE 5. Each topic related to the exam is explained in detail, and trade-offs are considered. Where appropriate, parallel development paths are explored and implemented.

Where applicable, chapters briefly discuss the design patterns being used and offer a brief explanation of those patterns. We strongly encourage you to purchase or download some pattern resources. Various web sites offer insightful tutorials, including the Sun site (`http://www.sun.com`) and TheServerSide.com (`http://www.theserverside.com`). There are also various excellent books on the topic, including *Head First Design Patterns*, by Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra (O'Reilly, 2004).

Throughout this book, we present numerous examples that aid in the development of a real-world Java application. Each chapter contributes directly to this application by addressing a critical topic such as threading, Swing, or networking. The text explores questions that naturally arise in these topics and explains how the challenges can be met. More important, the trade-offs and implications of these choices are discussed.

- Chapter 1, "Introduction." This chapter is a general introduction provided to help you decide if this book meets your needs. It lays out the structure of the educational program to follow, introduces the goals of the exam, and focuses the technology discussions to follow.
- Chapter 2, "Project Analysis and Design." This chapter details basic project considerations such as directory structure, package development, coding conventions, ReadMe files, and general approaches to starting a project for the SCJD exam.
- Chapter 3, "Project Overview." This chapter introduces the sample project, Denny's DVDs. This application requires that you develop classes to access a file in a database-like manner, build a Swing user interface, and design a networking layer. It is important to read this chapter carefully because it helps to define exactly what the project is trying to accomplish.
- Chapter 4, "Threading." This chapter starts from scratch and helps guide you to a clear understanding of threads. You will also learn about the `Runnable` interface, the `Thread` class, locks, synchronization, waiting, sleeping, notification of one or all waiting threads, the constraints of using threads with Swing, deadlocks, and thread scheduling. We will then move on to the new scheduling package of J2SE 5, and demonstrate how this can make threading easier. The focus is on the threading material you need to know in order to earn your Java developer certification.
- Chapter 5, "The DvdDatabase Class." This chapter demonstrates how to create a class that will meet the requirements of our sample project and implement a specified interface. We will use the Façade, Value Object, and Adapter patterns to simplify the code.
- Chapter 6, "Networking with RMI." This chapter provides an introduction to distributed computing. You will learn about RMI and how to utilize it when building your clients and servers.
- Chapter 7, "Networking with Sockets." This chapter provides an introduction to an alternative method of developing distributed computing. You will learn about sockets, and the pros and cons of working with sockets instead of RMI. Chapter 7 also briefly discusses security, serialization, the Command and Proxy patterns, and the sample project.
- Chapter 8, "The Graphical User Interfaces." This chapter provides an introduction to Swing. It is designed for Java programmers who have little to no Swing experience. The chapter assumes you are starting from scratch and quickly explains the fundamentals of how Swing works, what the MVC pattern is, how events are handled, how `JTables` work, and how all the pieces fit together.
- Chapter 9, "Project Wrap-Up." This chapter gives us a chance to examine the project in hindsight. We apply finishing touches and organize our JAR files. We also review the decisions made and the trade-offs involved, and prepare the project for submission.

The source code for the project, as well as various helpful diagrams and documents, can be obtained from the Source Code section of `http://www.apress.com`.

Setting Up the J2SE 5 JDK and Environmental Variables

Setting up the J2SE 5 JDK is very straightforward. Sun provides extensive documentation on how to do so for the various platforms, so we will not rehash all of that material.

Information on downloading and configuring the J2SE 5 JDK is available at `http://java.sun.com/j2se`.

Summary

In this chapter, we presented a broad overview of the Sun Certified Java Developer (SCJD) exam, as well as strategies you can use to meet your goal of passing the exam. We identified the areas that the exam covers, discussed the test itself, and offered some suggestions on taking it. We also provided a breakdown of the topics discussed in this book, integrated in the relevant J2SE 5 material.

With this information, you have already begun to prepare for the exam.

Congratulations! With about four weeks spent covering the issues set forth in this book, you should be able to take and pass the SCJD exam. In general, you should expect to spend a week on each of the four major topics: threading, Swing, networking, and the user interface. Of course, this will vary depending on your personal background. You now have a sense of what to expect. Good luck, study hard, and e-mail us at `scjd@apress.com` when you pass the exam.

FAQs

Q Am I ready to take the SCJD exam?

A If you have passed, or could pass, the Sun Certified Java Programmer (SCJP) exam, you are ready to prepare for the SCJD exam. There is no time limit on completing the assignment, so you could purchase it and learn as you work through the book and assignment simultaneously. However, be aware that there is a time limit on the exam voucher, and some Sun offices require you to purchase both the assignment and the exam voucher simultaneously.

Q Will this book help me if I am preparing for the Sun Certified Java Programmer (SCJP) exam?

A Using this book in conjunction with a book covering the SCJP topics may help your understanding of the topics; however, we do not recommend that this book be used as reference material for SCJP candidates. Although several of the topics required for SCJP certification are covered in this book in detail, many required topics are either not covered or are used without explanation on the assumption that their usage is well known. In addition, this book covers many topics that are not required for the SCJP.

Q I'm having some difficulty setting up my environment. Where should I turn for help?

A Look to the Sun Microsystems Java web site (`http://java.sun.com/j2se`) and follow their documentation exactly. If that doesn't work, contact Sun directly.

Q What topics does this book discuss?

A This book discusses and explains RMI, threading, Swing, networking, assertions, exception chaining, and logging.

Q How much does the SCJD exam cost?

A The exam costs roughly US$400. This price is, of course, subject to change at Sun's discretion.

Q I've lost my exam—what should I do?

A Try downloading the exam from Sun's site again. If that doesn't work, contact Sun directly.

CHAPTER 2

Project Analysis and Design

This chapter introduces project issues that are common to all software projects, discussing them in relation to this book's sample project and the Sun assignment that you need to complete to become a Sun Certified Java Developer. In particular, the following topics will be covered:

- Planning the beginning stages of the SCJD exam
- Organizing the layout of your project
- Documenting projects
- Becoming familiar with industry-standard principles on source code formatting and Javadoc, and incorporating these principles from the onset of project development
- Using Java packages to group code based on functional similarities
- Learning common development practices, including using assertions and logging

This chapter does not attempt to forge a new road, but rather leads down the well-worn paths of Java standards, such as coding conventions, Javadoc usage, and packaging concepts. Some of these tools are necessary in order to pass the SCJD exam, and all should be used every day by a Java developer.

By using these standards from the beginning, you will be well on your way to reaching your goal of being a certified Java 2 developer.

Implementing a Project

It is very tempting to start a project by jumping right into code. Doing so is fun and grants an immediate sense of progress. However, this approach often has significant drawbacks. Beginning a project without proper planning may tie the project to unspoken assumptions, cause you to overlook critical information, or introduce design flaws that manifest as the project progresses.

Generally, it is best to start by confirming requirements, designing data flow, and sketching a prototype of the graphical user interface (GUI) layout. After this step comes the design.

■ **Note** Many different development methodologies are used in varying degrees in the software industry. Some of the commonly used design approaches are the Iterative Process, the Rational Unified Process, the Boehm Spiral Model, and XP (Extreme Programming). In this book, we are using another common model: the Waterfall Model. As its name implies, this design model requires that development be an ongoing process, with one version of software based on a previous version, and so on. In addition, requirements are determined before project design and development begin in this model. The SCJD exam allows unlimited time to complete the project, and it requires that you work as a single developer. Both of these special criteria set forth in the SCJD exam fit perfectly into the Waterfall methodology.

Design is a fluid process that is grounded by coding. The best way to begin a project design is to explore the technical challenges ahead by coding a little, designing a little, and coding some more. In this respect, project design becomes an iterative process. Some suggested principles follow.

Getting Started

As a first step, spend a little time verifying your understanding of the requirements. Read the material several times and scrutinize its contents. Explore the logical breakdown of functionality and document your assumptions. Make sure that you note the umbrella activities that encompass several different variations under a given topic. This often helps with the package structure design. For example, it might make sense to have a GUI package that is responsible for visual presentation.

Gathering Requirements

Requirements are functions of a project that the client wants in the system you are creating. The requirements should detail everything that the system is required to do. Our suggestion is to ask questions. Better yet, ask a lot of questions. Write down the questions before you formulate answers, and ensure that they make sense. For the sake of clarity, phrase them differently and ask them again. It is probably best to be thought a little slow at the beginning of a project than to be proven careless at the end.

Confirm all assumptions either in writing or as a GUI layout. Of course, on this project, you are not going to be able to ask anyone your questions, but that should not prevent you from articulating potential issues and project risks. As a matter of fact, it should encourage you to formulate questions to organize your thoughts.

■ **Note** Chapter 3 introduces some example use cases derived from project requirements. Chapter 8 presents a full example of dealing with use cases and their translation to project functionality.

Prototyping the GUI

When you begin prototyping the GUI portion of the project, draw out simple layouts of the various command windows with pencil and paper. This activity will help you acquire a sense of what the user needs to see before you decide how the interface will work internally. This is often a crucial step in reconciling user expectations with the reality imposed by the system implementation. Chapter 8 presents examples of GUI prototyping and the interface layout process.

We recommend that you prototype the GUI using pencil and paper at this stage, rather than directly on the computer. Coding directly on computer runs the following risks:

- The design that you thought was so good, and that you spent so much time on, might be rejected by your sample testers (see the sidebar, "Sample Testers"), resulting in wasted time.
- You will almost certainly require more time to prototype a GUI on the computer compared with sketching the GUI layout on paper. We recommend you show your prototype to some sample testers (see the sidebar), and if they have recommendations for change (or, worst case, reject your prototype) there will be less time wasted if your prototype is only a penciled sketch.
- If you have a penciled sketch of your GUI, you can discuss it with your sample testers anywhere, regardless of whether there is a computer handy. And any changes they suggest can be incorporated in a few seconds.
- If you have spent a large amount of time coding your prototype, there is a natural resistance to changing it, which might result in you rejecting some otherwise excellent ideas from your sample users.
- You may get frustrated with implementation details long before a prototype is in place, and as a result you may sacrifice a good design for something that is easier to develop, and, accordingly a lower score for your GUI.
- If you code the GUI now, you may end up with something that you believe is very nice, but which you later find your users don't like. Once again, you run the risk of having to start from scratch.

■ **Note** In Chapter 8 we will be developing the GUIs for our sample project, and as part of this we will be showing some rough sketches of alternate screen layouts we might use for our project.

SAMPLE TESTERS

It is a truism that programmers write programs for programmers. That is, for any given assignment, we as programmers will tend to write a program that *we believe* is very logical but that most nonprogrammers will find difficult to use. This effect happens in all professions, and some professions employ staff simply to work on aesthetics—for example, some car manufacturers hire staff whose only job is to ensure that the car will look good to the final consumer.

We need to do something similar if we want to write GUI applications that the end users (in the case of the Sun assignment, the assignment assessor) will like and accordingly, approve (in the case of the Sun assignment, award full marks). Having end-user approval is extremely important—otherwise we end up in a never-ending cycle of making one change after another to the GUI. We need someone who is (preferably) not a programmer who can look at our prototype and our final application, and tell us what needs to be changed to make them feel like it is a great application, and not just a mediocre one.

These are our sample testers. They could be your spouse, your significant other, your mother, or the office secretary. They are the people who are likely to spot some feature that they consider standard, but that you have managed to leave out. And they are the ones who are likely to look at your application and tell you that something is in the wrong place. And when it comes to testing the final application, they are the ones who are likely to do the things you were not expecting—trying to open two applications at once, or trying to reduce the size of your application screen below the size you thought anyone would use.

The people to try to avoid are other programmers—they are the people who are most likely to not mention some feature because they don't like that particular feature themselves. Furthermore, they may ignore a usability issue because they are used to working around issues in others' programs.

So see if you can think of some sample testers, take your rough sketches of screens to them, and ask them what they think. Then listen to their comments, and go back and make any necessary modifications.

If possible, take some samples of totally different screens to your sample testers—give them a choice of what sort of interface they would like to work with. They will feel that they have more involvement, and they will often feel that they can suggest more modifications to one of your sample layouts since it has not yet been finalized.

Using Accepted Design Patterns

Failing to follow conventions is rarely worth the development time. Worse, it may cause you to fail Sun's exam. Worst of all, in the real world it will attract the hatred of programmers who will have to maintain the cryptic code that ensues.

While it is possible, and often clever, to implement custom solutions to general problems, you should resist the temptation to do so for this project and Sun's exam. In real life, however, custom solutions are occasionally faster and cheaper than general solutions. For example, a custom method that sorts the elements of an array may be faster than the methods that are built into the `Arrays` class.

Since this book's focus is not software design patterns, there are many design patterns that we cannot cover in this book. We strongly recommend that you read up on these yourself, as you will use them in your development career. The most widely recognized book on the subject is *Design Patterns* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional, 1995). This is commonly referred to as the "Gang of Four," or GOF, book. You may find that this is not the easiest book to read, so you may wish to investigate

some of the alternatives such as "Head First Design Patterns" by Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra (O'Reilly, 2004), the Portland Pattern Repository at `http://c2.com/ppr/index.html`, or the Wikipedia entries for design patterns starting at `http://en.wikipedia.org/wiki/Design_pattern_(computer_science)`. Sun's J2EE design patterns at `http://java.sun.com/blueprints/patterns/` are also very useful, but you should be aware that they are J2EE-centric, and may describe patterns in a form that may not seem to make sense for this assignment.

Documenting Design Decisions

Document the choices that you make during the development process. Write down the various decisions you make and the reasons you make them. For example, if you decide to use an `ArrayList` instead of a `Vector` for an internal data structure, document the fact the `ArrayLists` are not synchronized and thus were chosen because they are a more lightweight data structure. There is no need to go overboard with this type of documentation, so be mindful to use common sense. This sort of documentation is a crucial tool in debugging and performance tuning, and it also serves as an excellent source for anyone who needs to understand the code.

Whatever you do, don't leave documenting your design decisions until the end of the project. If you wait until after you have completed coding before beginning this document, not only will you have to remember why you chose a particular option, but you will also have to remember that you had other options in the first place! For example, if you decided at the start of the assignment that you were going to use a custom dialog box, at the end of the assignment you might have forgotten that you had originally also considered having an editable cell on the main window.

It is quite possible that by the time you get to the end of the assignment, you may find that your design decisions document is huge—minor decisions can be removed to reduce the size of the document.

■ **Tip** You might want to use bullet points to describe your design decisions. Not only will this reduce the amount you write, but also they are easier to remember when it comes time for the exam. This is especially useful for candidates for whom English is not their first language—you will not have to be so concerned about how good your spelling is or how well you have formed your sentences.

For instance, we might document a few decisions like so:

- Sockets used instead of RMI—allows complete control over threads
- `RandomAccessFile` used instead of separate `DataInputStream` and `DataOutputStream` classes—allows for random access to the file

Testing

You should begin testing by writing a unit test client for each class, or you can use a testing tool such as JUnit (`http://www.junit.org/`) to automatically generate test clients. When you design a custom test, it is best to write the test class before implementing the methods to be

We recommend that you design tests that cover the conditions set forth in a project's written requirements before beginning the development process. Under these conditions, if a class provides the functionality detailed in the project requirements without failing the prewritten tests, then the test can be considered a success. Additional functionality beyond what is required is unnecessary. Writing test cases before coding deliverable classes will prevent the overzealous programmer from dwelling on functionality that is not required.

As your project develops, you may find yourself adding to, or changing, methods within the class. Be sure to update test clients accordingly as the project classes evolve. It is generally recommended that test cases should never be deleted—once the test has been written, it provides valuable confirmation that basic functionality still works if you modify your deliverable classes.

■ **Note** A unit test simply tests a unit (usually a class) that you have written. If your class has a `setDvdName(String dvdName)` method, then your unit test should call it, preferably with the various kinds of input it can expect. You should do this for every method. Unit testing is not required for the exam, but we strongly recommend it as a sanity test. Of course, you should not send your unit tests or their results to Sun.

When the application is ready for system testing, recruit some volunteers to help. It is generally best to avoid system-testing your own code—use an unsympathetic and unbiased eye to look over the system. When you design the system test, it is a good idea to work with the client. If this is not possible (as it is not in this project), then use the requirements gathered at the beginning of the project.

■ **Note** A system test simply tests how the various units (usually classes) fit together. For example, your client class might need to call your DVD class's `setDvdName(String dvdName)` method. Even though you know the client and the DVD class both work correctly from the unit test, you don't know if they work well together. For example, there might be a network problem or the client class might store DVD names as an array of characters, where the DVD class uses `String`. System testing enables you to make sure that all of your classes play nicely together.

Organizing a Project

One of the first tasks in any software project is determining a sound organization for all related project materials. An organizational paradigm, in this case, is a directory structure aimed at organizing all files associated with the project. It is extremely difficult to decide such matters later in the development process and attempt a retrofit. You should decide upon a directory structure early in the project planning stage. A typical project directory structure follows. Sun does not dictate a directory structure for the development of the SCJD assignment; however, on some assignments they do specify some top-level directories to be used in the submission. The directory structure in Table 2-1 acts as a suggested organizational foundation

for the assignment. We use the structure detailed in Table 2-1 throughout this book's example application project.

Table 2-1. *Suggested Directory Structure*

Subdirectory	Recommended Use
`src\`	Contains all of the `.java` source files written during the course of the project.
`classes\`	Contains all compiled class files and any packaged JAR files. The classpath will point to this directory when we are running our application.
`bkp\`	A directory to hold any files needed for backup.
`tst\`	Contains all of the `.java` source files written for unit-testing the project.
`tmp\`	A "hold anything" directory for temporary storage.
`log\`	A directory to store all logged output.
`doc\`	Holds all documentation, including Javadocs, end-user documentation, and design decisions documentation.

High-Level Documentation

In addition to completing the code portion of the test, to pass the SCJD exam you must author and submit several forms of documentation. At the time of writing, the following documentation is required:

- Javadoc documentation (discussed in depth later in this chapter).
- A plain text file named `version.txt`.
- User instructions—unless the user instructions are built into your application and available while the application is running.
- A design decisions document. The design decisions document is discussed in detail in the following section.

The `version.txt` file must contain an explanation of the following items:

- The version of the JDK used for development
- The development platform

This book uses JDK version 5 and Microsoft Windows 2000.

■ **Caution** There are currently several different Sun assignments in use, and instructions may vary between assignments. Sun may also release assignments in the future with other minor differences in the instructions. While the information in this book will be generally applicable to any current assignment, you must take care to read the instructions you have received from Sun carefully, and ensure that you follow them.

WHAT EXTRA FILES SHOULD YOU INCLUDE IN SUBMISSIONS?

We often see questions from candidates asking whether they should include their test cases and/or their class diagrams in their submission.

Our general recommendation is not to include anything that you have not been asked for. The latest instructions from Sun include a comment that you will not be given extra marks for anything you do outside of the requirements, so you are not going to gain anything by providing the assessor with these extra files. However, it is unfortunately possible that in providing these files you may inadvertently lower your score simply by making a mistake that the assessor notices in a file that you didn't need to provide.

The one time we might consider changing this general recommendation is where the additional files make it much simpler to understand your submission. A class diagram might be one such example (however, the assignment is simple enough that if you need a class diagram to understand the submission, then you have probably overcomplicated your solution).

User directions are essential. After all, if you do not explain to the client how to use the application, then the application is rendered useless. As a result, you should take great care when writing these instructions. The only safe assumption you as the developer can make is that the end user has no experience with this particular application. Every step, no matter how minute, must be detailed in the instructions. After you list the instructions, test their clarity by handing them off to unsuspecting friends (preferably nonprogrammers). If they can follow the instructions, then the instructions are adequate.

■ **Caution** The current assignment instructions specify that the instructions you write may be placed in a specific directory or may be available online. This directive has caused confusion in the past, as some candidates have felt that this might mean that this requires them to run a web server—but this is not the case. Sun only requires that the assessor have access to the instructions, which can be achieved if the instructions are in the required directory, or alternatively can be called up from within your application (for example, pressing the F1 key in Microsoft Word will bring up "online help" even if you do not have an Internet connection).

Design Decisions Document

Throughout the SCJD exam, certain design and implementation choices are already dictated by Sun. One example is that the exam requires the use of the `JTable` Swing component. Other choices, however, are left up to you, the test taker, to decide. For instance, you may choose to implement a networking layer that uses RMI, or you may take a different approach that is built upon sockets. Each implementation has advantages and disadvantages. You must be certain to document your choices because it is necessary to defend these design decisions to the individual who will ultimately decide if your test submission passes or fails.

For the SCJD exam, clearly document your design choices in a design decisions document. This document should contain examples of specific decisions, such as your choice of design pattern or the use of one technology over another. Circumstances may also arise in which design decisions were made based on unclear functional requirements. If this situation

does occur, raise the issue as a design decision and document all the assumptions you made to deal with the problem. Be certain to complete the design decisions document, because it is the only chance the test allows you to defend your submitted project.

■ **Tip** It is worth noting that in the Sun assignment, as with projects in real life, there are sometimes several solutions to any given problem. It is also possible that for *every* possible solution there will be reasons why that particular solution is not optimal. You should not spend too much time trying to find the "one perfect solution"—it may not exist. Sun has deliberately left enough vagueness in the assignment instructions that there are very few areas where candidates have limited choices. In all other cases, it does not matter so much what choice is made, but it does matter how you came to your choice, which should be detailed in your design decisions document.

Java Coding Conventions

One of the common goals in our industry is the ability to hand over the project to someone else—let them do any maintenance in the future, as you won't be available (you will be working on more exciting projects and going on vacation).

To meet this goal, the code we write needs to be formatted in such a way that you can hand it over to somebody else, and they will happily accept it. It will not do your reputation any good if you hand over the code, and the other person throws it all away as being incomprehensible. In the same way, it would not do us any good if we did not organize this book into chapters, paragraphs, and sentences—if you can't read this book, you won't learn much from it.

The developers of Java, C, and C++ deliberately avoided forcing coders to follow a specific coding convention—there are syntactical requirements, but as long as you meet them, the code can appear on a printed page any way you like it to. For instance, consider the following code snippet:

```
public class MyTest {
    public static void main(String[] args) {
        System.out.println("Hello");
    }
}
```

That code is interpreted by the Java compiler in exactly the same way as

```
public
class
MyTest{public static void main(String
z[]){System.out.println("Hello");}}
```

You'll undoubtedly agree that the first format is far more readable than the second.

While it is easy to agree that a code-formatting style should be followed, it is less easy to agree on the code-formatting style itself. For example, often one person prefers to have the brace ({) at the end of an existing line; another prefers to have the brace on its own line. Both coders can have compelling arguments for their particular style, but realistically only one style

In the workplace, management will usually specify which particular style must be used. For this book, and for the SCJD assignment, we recommend you use the Sun Code Conventions for the Java Programming Language, which you can download at `http://java.sun.com/docs/codeconv/`. Sun has specified 11 areas where they believe coding guidelines are needed, and these can be grouped into the following major categories: naming conventions, file layout, source code format, and comment format. We introduce each of these categories next.

Naming Conventions

The Sun Code Conventions specify different naming conventions for packages, classes and interfaces, methods, variables, and constants.

For all of the naming conventions, you should try to avoid using abbreviations, except where the abbreviation is more commonly recognized than the complete word. At the same time, you should try not to make your names too long, or they will quickly become tedious to read and write. Consider the variable names shown in Table 2-2.

Table 2-2. *Variable Naming Examples*

Contents	Good Variable Name	Poor Variable Name
The balance of an account	`accountBalance`	`usersCurrentAccountBalance` (too long) `ab` (not a common abbreviation; doesn't mean much)
HTML editor class	`HtmlEditor`	`HyperTextMarkupLanguageEditor` ("HTML" is a common abbreviation, so using it will enhance the readability of this class name)

Package Naming Conventions

Package names start with your fully qualified domain name, written in lowercase and in reverse. So if you worked for a company with the domain name `example.com`, then your package names should start with the same name in reverse (`com.example`).

From that point on, you would follow your company's naming conventions. A sample naming convention might be to use the project name, followed by a conceptual grouping of classes. For instance, as we are working on the SCJD project, which contains a GUI client, we could have a combined package name of `com.sun.edu.scjd.gui` (for simplicity though, we have used the base package name of `sampleproject` throughout our project, and the GUI code is therefore in package `sampleproject.gui`).

■ **Tip** In the real assignment, Sun will typically specify a package name for at least one class. Therefore, you do not need to be concerned that you do not have an existing domain name that you can use as your base package name.

Class and Interface Naming Conventions

Class and interface names should always start with a capital letter, and should be a noun (they should describe an object, not an action on the object). For example, "Book" might be used as the name of the class containing information about a book.

It is common to combine two or more nouns or an adjective and a noun together to form the class name, in which case CamelCase is used (the first letter of each word is capitalized, producing the undulating pattern associated with camels). For example, "SocketFactory" might be used as the name of a class that creates socket connections.

■ **Tip** You should try to have only one responsibility for each class. For instance, a class that is responsible for creating an RMI connection to a server should not also be responsible for displaying data to the end user. If you can maintain "one responsibility per class," you will find it easier to name your classes. This provides a major benefit later when it is time to modify or maintain your classes—the separation of responsibilities and clear class names makes it much easier to determine which classes need to be modified.

Method Naming Conventions

Method names should always start with a lowercase letter, and should begin with a verb (they should describe an action on the object). For example, within our `DVD` class, the method name `getLeadActor` indicates that we can call this method to get the name of the lead actor for the DVD.

It is very common to combine several words to give more information on what the method does. For instance, using the method name `getLeadActor` makes it far more explicit when *using* this method that we are specifically retrieving the lead actor's name (and not the name of some other person associated with this DVD). As can be seen in this example, CamelCase is used when combining words.

Variable Naming Conventions

Variable names should always start with a lowercase letter, should be short, and should describe what data is stored in the variable. For example, within our `DVD` class, the variable name `leadActor` would contain the names of the lead actor of the film on DVD.

Again, it is very common to combine several words to provide more information on what the variable does. As you can see in this example, CamelCase is used when combining words.

■ **Note** The Sun Coding Conventions specify that you should apply the same naming convention to all instance, class, and local variables. Be aware that you may see code written by other coders where instance or class variables are signified by an underscore or some other special mark. Another way to achieve the same effect is by using the convention `variable` when referring to a local variable, `this.variable` when referring to an instance variable, and `Class.variable` when referring to a class variable. Doing so makes it explicit which type of variable you are referring to.

Constant Naming Conventions

Constants are always written in all capital letters, with individual words separated by underscores. An example might be the constant `DIRECTOR_LENGTH`, which would be set to the maximum size of the director's name stored in our database.

File Layout

A Java class or interface always consists of the following standard layout:

1. Beginning comments
2. Package and import statements
3. Class or interface declarations

The Sun Code Conventions state that two blank lines should appear between each of these major sections. That is, there should be two blank lines between the beginning comments and the package statement. Similarly, there should be two blank lines between the import statements and the class or interface declarations. In all other cases where a blank line will help readability (say, between method declarations), you would normally only have a single blank line. A simple example is shown in the following code snippet:

```
1  /*
2   * HelloWorld.java    version 1.0.0    date 2005-06-20
3   * Copyright © Andrew Monkhouse & Terry Camerlengo 2005
4   *
5   * This is a version of the hello world program
6   * The beginning comment, has two blank lines following it
7   */
8
9
10 package com.example.javaExamples;
11
12 import java.util.Date;
13
14
15 public class HelloWorld {
16     public static void main(String[] args) {
17         sayHello();
18     }
19
20     public static void sayHello() {
21         System.out.println("Hello, world at " + new Date() + "\n");
22     }
23 }
```

As shown here, there are two blank lines between each of the major sections: lines 8 and 9 separate the beginning comments from the package and import statements, and lines 13 and

14 separate the import statements from the class declaration. In all other cases, only a single blank line is used to separate minor sections: line 11 separates the package statement from the first import statement, and line 19 separates the contructor and method declarations.

Beginning Comments

Beginning comments are separate from Javadoc comments and, as such, are often not understood by Java programmers. The beginning comments contain some of the same information as the Javadoc comment for the class, but there are a couple of major differences: the information is provided in one standard place, and very specific information is listed. While the Javadoc comments might contain a superset of the same information as the beginning comments, there is no specific line number the information will appear on, and the desired information may be buried among API documentation. Beginning comments contain the following:

- Class name
- Version information (might be automatically filled in by your revision control system)
- Creation/modification date (might be automatically filled in by your revision control system)
- Author/last modifier (might be automatically filled in by your revision control system)

The entire comments block is a C type comment, not a Javadoc comment—that is, the comment block starts with /* and not /**.

Package and Import Statements

Following the beginning comments, you put your package statement, a blank line, and then your import statements, as shown in lines 10–12 of the preceding code example.

Although the Sun Coding Conventions document does not specify whether you should list every class or include the entire package, one common usage is to list individual classes in a package until there are three classes listed in a single package. After that, it is common to import the entire package.

Likewise, the Sun Coding Conventions do not specify whether import statements should be in any particular order. Worrying about such details is probably going beyond the scope of the requirements (and may cause your colleagues to look at you in a funny way). Many developers tend to keep them in alphabetical order, but don't get too concerned about this.

■ **Tip** Many integrated development environments (IDEs) have some of the following features: automatic addition of missing import statements; automatic removal of unused import statements; and automatic refactoring of too many or too few imports in a given package. While these features can help improve your coding speed in your real job, we recommend you switch these features off while working on the SCJD assignment. One of the problems with using some of the IDE "features" is that it can become difficult to determine what has gone wrong if something does go wrong—if you learn to work with import statements manually for the SCJD assignment, you are more likely to be able to handle any issues later in life.

It is common practice to include a blank line between an import of standard J2SE packages, standard J2EE packages, external packages, and internal packages.

Class or Interface Declarations

The Sun Coding Conventions specify that a class or interface declaration should contain some or all of the following elements in the specified order:

- Class/interface Javadoc comments
- Class/interface statement
- Class variables
- Instance variables
- Constructors
- Methods

Variables should be sorted according to accessibility, from most accessible (public) through to least accessible (private). For example:

```
public class VariableOrderExample {
    public int aVariableModifiableByAnyOtherClass;
    public String anotherPublicVariable;
    // protected variables appear after public variables
    protected int protectedVariable;
    // now list the variables with default access
    Character defaultAccessVariable;
    // finally list the variables with private access
    private int noOtherClassCanSeeMe;
}
```

■ **Tip** Although the location of constants is not specified by the Sun Coding Conventions, common usage is to list them prior to the class variables.

Methods, on the other hand, should be grouped by functionality rather than scope. This means you should put a common private method close to the public methods that call it.

Source Code Formatting

While you write your code, you should maintain a consistent approach to the following style issues:

- Indentation
- Line lengths/wrapping

- Spacing
- Statement formatting
- Variable declaration formatting

Most of these formatting rules have been designed so that people reading your code can do so using the IDE, editor, screen resolution, and so forth of their choice. If you were to choose a nonstandard formatting convention, then others may find your code hard to read.

■ **Caution** Do not ever forget that you will be submitting your source code to an unknown assessor for review. You really need to write your code so that it is a pleasure for them to assess. This also applies in your real job—you should always be writing code that your coworkers are happy to use.

Indentation

The Sun Coding Conventions contain the following indentation requirement: "Four spaces should be used as the unit of indentation. The exact construction of the indentation (spaces vs. tabs) is unspecified. Tabs must be set exactly every 8 spaces (not 4)" (Sun Coding Conventions, `http://java.sun.com/docs/codeconv/html/CodeConventions.doc3.html`, 1999).

This description seems to cause a great deal of confusion when first read, so an example is in order:

```
1  public class IndentationExample {
2      /* this line is indented once */
3      public IndentationExample() {
4              /* this line is indented twice */
5      }
6  }
```

Lines 2, 3, and 5 all have one indentation, so you should prefix them with four spaces. Line 4 has two indentations, so you *should* prefix them with eight spaces. However, eight spaces are also equivalent to a tab, so you *could* use a tab instead of the eight spaces for line 4. To avoid confusion, we recommend that you use eight spaces instead of a tab.

■ **Tip** Many IDEs and editors give you the option to insert a specified number of spaces whenever you press the Tab key (and even handle backspacing over the indentation/reformatting entire blocks of code). We recommend you check whether your IDE/editor provides this functionality and turn it on when available.

Line Lengths/Wrapping

As mentioned earlier, you cannot know what sort of an editor or what screen size your assessor will be using. Limiting line lengths to 80 characters will ensure that your code should be readable in most cases.

Lines that are longer than 80 characters should be broken after a comma or before an operator whenever possible. The second (and subsequent lines) should be indented to the beginning of the expression on the previous line, or eight spaces. For example:

```
int i = myMethod(longNamedVariable1, longNamedVariable2,
                 longNamedVariable3, longNamedVariable4);
```

Line wrapping for `if`, `for`, and `while` statements generally uses the eight-space indentation rule rather than the beginning of the expression, since using the beginning of the expression can cause confusion with the line that follows, which will be indented four characters. The following example shows both the preferred and nonpreferred way of breaking `if` statements:

```
// nonpreferred way
if (myMethod(longNamedVariable1, longNamedVariable2,
    longNamedVariable3, longNamedVariable4)) {
    // code starts here - see how confusing this is ?
    doSomething();
}

// preferred way
if (myMethod(longNamedVariable1, longNamedVariable2,
        longNamedVariable3, longNamedVariable4)) {
    // code starts here - now we can see the difference between
    // the condition and the code to be run within the condition
    doSomething();
}
```

In cases where you have multiple levels of code in one line, for example, calling a method and using the result as a parameter to another method call, it is preferable to break the line at the higher level—keep the call to the external method on one line where possible. For example:

```
int i = myMethod(variable1,
                 callToAnotherMethod(variable1, variable2),
                 (variable1 + variable2));
```

In such cases as this last example, you should consider whether your code would be more readable and understandable if you were to refactor it. For example, you could

- Move the external procedure call to a separate line.
- Move the calculation in parentheses to a separate line.
- Make both modifications.

Consider how much easier the following code might be to read and maintain:

```
int dbValue = callToAnotherMethod(variable1, variable2);
int calculated = variable1 + variable2;
int i = myMethod(variable1, dbValue, calculated);
```

Once you get beyond your third or fourth level of indentation, you may find that these rules are hard to follow. This is often also an indication that your code may be difficult to

follow, and you should think about whether you could move some of the indented code into a separate method.

Spacing

Spacing is intended to make code easier to read, but too much of it can have the opposite effect and make the code harder to read and maintain. In general, you should put one space between a keyword and a parenthesis, after commas, between expressions in `for` statements, after casts, and around all binary operators. Examples of these are shown in Table 2-3.

Table 2-3. *Spacing Examples*

Rule	Example
Space between keyword and parenthesis	`while (true)`
Space after commas	`myMethod(variable1, variable2);`
Space between expressions	`for (expression1; expression2; expression3) {`
Space after casts	`int i = (int) aLongValue;`
Space after binary operators	`a = b + c;`
	`c = 5 * 10;`

Statement Formatting

Most of the statement-formatting rules simply follow on from the rules we have already discussed.

Most programmers have a preferred method of writing compound statements—for example, where to place braces, and whether braces are optional. Rather than leaving this for endless debate, Sun has specified that braces must be used to enclose statements as part of a control structure—even if only one statement is used in the control structure. For instance, even if there is only one line of code to be executed following an `if` statement, it must still be enclosed in braces, as shown here:

```
if (variable == someValue) {
    doSomething();
}
```

As you can see, the statement between braces should be indented one indentation level (four spaces), and the closing brace should start on its own line at the original indentation level.

■ **Tip** Many tools are available that you can use to help you confirm that the code you have written confirms to the Sun coding standards. One such tool is Checkstyle (`http://checkstyle.sourceforge.net/`)—it integrates neatly into many popular IDEs, and supports many different coding styles, not just Sun's. A good style checker will provide you with a report on what it believes needs to be changed, after which you can manually verify each item.

■ **Caution** Try to avoid using automatic code reformatters for this assignment. Occasionally they will reformat your code in a manner allowed by the coding conventions but contrary to the way you (or the assessor) would want to see it. Unfortunately, once this has been done, it is hard to undo, and most automatic formatting tools do not provide an easy way to review the changes before accepting or denying them.

Variable Declaration Formatting

Variables should be declared one per line, preferably with comments following the declaration where applicable.

The Sun Coding Conventions state that you may separate variable names from their types by either a space or by tabs. Unfortunately, using multiple tabs can result in you spending too much time reformatting code when you add a new variable later, so we recommend you always conform to a single space between the type and the name of the variable.

Where possible, variables should be initialized when they are first declared. Furthermore, variables should be declared near the start of the smallest enclosing brace that provides the necessary scope. You should not wait to declare a variable until just before you use it.

Formatting of Comments Within the Code

Two forms of comments are allowed within Java source code: documentation comments (Javadoc comments) and implementation comments (all other comments).

Javadoc comments will be covered in more detail in the section on Javadoc later in this chapter; for now, suffice it to say that Javadoc comments are designed to create API documentation that another programmer can use to learn how to use your class and its associated constants, methods, (and possibly) variables without looking at your source code. Since Javadoc comments are designed to be used by external programmers, they should explain conceptually how your code works as a whole—they should never get into implementation details.

Implementation comments, on the other hand, are supposed to give hints to programmers (or the assessors) who are looking at your code as to what is happening.

■ **Caution** Do not go overboard with adding implementation comments to your code. If you are using good names for your classes, methods, and variables, your code will be generally self-documenting. In such cases, adding too many comments is self-defeating—the comments distract from the code and quickly fall out of date.

Implementation comments come in two flavors: the block comment (enclosed between `/*` and `*/` tags), and comments that start with the tag `//` and continue until the end of the line.

When adding a single comment line, or adding a comment to the end of a line of code, we recommend that you use the `//` comment delimiter, as shown in the following examples:

```
// this is an example of a comment using a single line of text
doSomething();        // this is an example of a comment at the end of a line
```

When a comment won't fit on a single line of text, use block comments, as shown here:

```
/*
 * This comment explains why the following code must be used instead of a more
 * "intuitive" way. As it takes more than one line, it is in a block comment.
 */
```

Try to avoid using block comments to comment out lines of code—you should use // comments instead. Using block comments to comment out lines of code makes it very hard for other programmers to determine which code is in use and which has been commented out. Comments that start on a line by themselves should always be indented to the same level as the code they apply to.

Suggested Coding Conventions for New Features in JDK 5

The Sun Coding Conventions have not yet been updated to reflect the additions in JDK 5. In this section, we will give a brief explanation of some of the new features, and demonstrate the coding convention as typified in the JDK 5 API, Sun sample code, and the Java Specification Request (JSR) that specified the new feature.

■ **Note** Before starting JDK 5, Sun asked the Java developers what features they would like to see in the new version. All requests were considered, and users were allowed to vote for those features they considered most valuable. A JSR number then specified these features, and the top requests were incorporated into JDK 5. You can find out more about the Java Community Process, and have your own say in future enhancements, by visiting the JCP website at `http://jcp.org/en/home/index`.

We will provide a quick overview of the new features in this chapter, and then cover them in more detail in later chapters where they can be seen in the context of our assignment.

Generics

Generics (supporting generic types at runtime in a type-safe manner) allow us to specify at compile time what a generic object will contain at runtime. An example will probably make this easier to understand. The formatting rules used in the example will be summarized at the end of this section. Consider the following non-type-safe code:

```
public List getBreed(String breedName) {
    List dogs = new ArrayList();
    // do some work to find the correct dogs
    String dogName = "";
    Dog pooch = new Dog(dogName);
    dogs.add(pooch);
    return dogs;
}
```

```
public void listDogs() {
    Collection c = getBreed("labrador");
    for (Iterator i = c.iterator(); i.hasNext(); )
        String name = ((Dog) i.next()).getName();
        System.out.println(name);
    }
}
```

This does work; however, although we know that the returned `List` must contain a list of `Dogs`, there is nothing to stop someone else from compiling source code that assumes that the list contains `Cats`—it will compile without problems. However, if that happens, a `ClassCast-Exception` will be thrown at runtime.

To get around that, you would have to write explicit type checking into your code (using the `instanceof` operator) and/or catch `ClassCastException`.

It is much nicer when we can ensure that a generic class will be handled correctly at runtime. Consider the following replacement for the `getBreed` and `listDogs` methods:

```
    public List<Dog> getBreed(String breedName) {
        List<Dog> dogs = new ArrayList<Dog>();
        // do some work to find the correct dogs
        String dogName = "Labrador";
        Dog pooch = new Dog(dogName);
        dogs.add(pooch);
        return dogs;
    }

    public void listDogs() {
        Collection<Dog> c = getBreed("labrador");
        for (Iterator<Dog> i = c.iterator(); i.hasNext(); ) {
            String name = i.next().getName();
            System.out.println(name);
        }
    }
```

We have now specified that the return type of the `getBreed` method will be a `List` of `Dogs`. (It may help to read `List<Dog>` as `List` of `Dog`).

Note that the line `String name = i.next().getName();` does not cast the class anymore—the line defining the `Iterator` specifies its type.

Attempting to cast an item in the collection to a nonrelated class produces the following error:

```
GenericExample.java:16: inconvertible types
found   : Dog
required: Cat
            String name = ((Cat) i.next()).getName();
                           ^
1 error
```

Note the formatting used in the above examples:

- There are no spaces within the angle brackets (< and >).
- There is no space between the collection name and the angle brackets.
- There is no space between the angle brackets and the curved brackets (when calling a constructor).

These are the coding conventions used in the JSRs and in several Sun documents describing the new features of JDK 5; however, currently there are no formal conventions.

Enhanced for Loop

In our example of generics, we used an `Iterator` to step through the items in the collection. To reiterate, our code example looked like this:

```
for (Iterator<Dog> i = c.iterator(); i.hasNext(); ) {
    String name = i.next().getName();
```

However, this is overly wordy: in most cases you will want to step through all the items in your iterator one by one. So why spell it out for the compiler in this way?

JDK 5 has made the use of iterators in for loops much easier. Consider the following construct, which does the same work:

```
for (Dog mutt : c) {
    String name = mutt.getName();
```

It may help to read the colon (`:`) as "in." Thus, the statement `for (Dog mutt : c)` would read "for each mutt in collection c."

■ **Note** Many people are curious about why Sun chose to use the colon instead of using the words "in" or "foreach." Quite simply, Sun wanted to avoid introducing any new keywords, which might potentially break existing source code. All your existing code should compile and run without any problems under JDK 5.

The enhanced for loop can also work with standard arrays. The following code gives an example of iterating over an array of `Strings` without using the enhanced for loop:

```
public static void main(String[] args) {
    for (int i = 0; i < args.length; i++) {
        System.out.println(args[i]);
    }
}
```

The enhanced for loop allows this to be simplified, as shown here:

```
public static void main(String[] args) {
    for (String arg : args) {
        System.out.println(arg);
    }
}
```

Note that the new for loop can't be used in every scenario. In particular, it hides the iterator, so you can no longer call any methods on the iterator that might change the underlying collection.

The formatting rule for the new for loops is to have a space before and after the colon.

Autoboxing

JDK 5 allows for automatic type conversions between primitives and their wrapper classes. For example:

```
Integer myInteger = 5; // automatically converts 5 (int) into an Integer
```

No special coding convention is needed for handling this.

VarArgs

Variable argument lists (also commonly referred to as VarArgs) allow the coder to specify that the number of arguments in a constructor or method signature is variable. Using this facility can reduce the number of overloaded methods and constructors.

However, VarArgs come at a price: when you specify the exact number and type of parameters a given method requires, the Java compiler can perform type checking to ensure that your usage of the method is correct. When you use VarArgs, only minimal checking is possible.

To give an example, consider creating a class `Dog`, which has two optional attributes (fields): `age` and `name`. If you wanted to create constructors that can handle all potential ways that a user could create a `Dog` class, you would need the following constructors:

```
Dog();
Dog(int age);
Dog(String name);
Dog(int age, String name);
```

Unfortunately, it doesn't end there. The number of constructors you need is 2 to the power of the number of parameters. So if the `Dog` class has the following seven attributes—`age`, `height`, `weight`, `name`, `owner`, `color`, and `pedigree`—we would potentially need 128 constructors to allow for every combination!

We would then have additional problems, because Java would be unable to differentiate between constructors where the method signatures are effectively the same. For example:

```
Dog(int age, String name);
Dog(int age, String owner);
```

JDK 5 allows us to specify that there will be a variable number of arguments *of the same type* passed to the method or constructor. So you could specify that the exact number of `String`s passed to the constructor is variable by specifying a constructor of

```
Dog(int age, String... args) {
    for (String parameter : args) {
        System.out.println("Received parameter " + parameter);
    }
}
```

As you can see in the previous code, the ellipsis (`...`) immediately follows the type it refers to. Once it's inside the body of your method, you can just treat the argument as an array of the named type.

■ **Note** You can only have one variable argument list in a given method or constructor declaration, and it must be the last parameter in the method or constructor. For example, you cannot have `Dog(int... ages, String name)`.

■ **Caution** Be very careful when overloading constructors or methods with versions containing VarArgs. It is very easy to end up with overridden methods that cannot be differentiated by the compiler (e.g., `Dog(String name, String... args)` and `Dog(String... args)` are effectively the same), or end up with a generic method that matches more than you bargained for (e.g., `Dog(Object... args)` will match *all* constructors, due to autoboxing).

You may have realized that a similar effect was possible under earlier versions of the JDK: namely, you simply passed an array of the named type into your method. However, to use this, you effectively had to create a new array. For example:

```
public static void lookupDog(String... searchCriteria) {
    for (String criterion : searchCriteria) {
        // do work here
    }
}

public static void lookupCat(String[] searchCriteria) {
    for (String criterion : searchCriteria) {
        // do work here - no different than working with Dog method
    }
}

public static void main(String[] args) {
    // first the easy code: use the Dog method:
    lookupDog("Breed", "Terrier", "Color", "Brown");
```

```
        // now for the Cat method
        lookupCat(new String[] {"Breed", "Burmese", "Coat", "Silky"});

        // or
        String[] criteria = {" Breed", "Burmese", "Coat", "Silky"};
        lookupCat(criteria);
}
```

The code to use the `lookupDog` method is much easier to read, write, and maintain than the equivalent `lookupCat` method.

Static Imports

Prior to JDK 5 some programmers were defining constants in interfaces, as this allowed them to *implement* the interface, giving them a shorthand way of writing the constant name. This is shown in the following example:

```
public interface BadInterface {
    public static final int FIRST_NAME_POSITION = 1;
}

public class BadClass implements BadInterface {
    public static void main(String[] args) {
        System.out.println("First name = " + args[FIRST_NAME_POSITION]);
    }
}
```

There are a couple of problems with this:

- A class that implements an interface is said to "be" an instance of that interface. If you have a reasonable name for your interface, it probably doesn't make sense to say your class "is" an instance of it.
- If your class implements an interface, then any subclasses of your class will also implement the interface. Effectively, the constants will become part of the namespace of the subclass—even though the subclass may have no need of these constants.

To get around these problems, JDK 5 introduces the idea of *static imports*—the ability to import the static members from another class or interface.

For instance, if you were using the logging features (discussed in the section "Logging" later in this chapter), you would normally have to qualify the logging levels as this code snippet shows:

```
        myLogger.log(Level.FINE, "This message is at FINE level");
```

However, static imports allow us to refer to the static object `FINE` as though it had been defined within our own class:

```
import static java.util.logging.Level.*;
// ...
        myLogger.log(FINE, "This message is at FINE level");
```

It should be noted that this "feature" was introduced to work around a bad programming practice. You might want to use it when you would otherwise be tempted to declare local copies of the constants, or when you are tempted to abuse the inheritance as mentioned earlier. But, in general, we recommend that you avoid this where possible, and use qualified constants.

Javadoc

Javadoc is a very simple, yet powerful, tool that helps programmers provide API documentation for other programmers.

The default Javadoc tool is very simple. It parses your source code, looking for special comments. It then generates HTML documentation based on that. To make this easier to understand, let's start with some sample Javadoc comments, and describe them:

```
/**
 * The <b>main</b> starting point for this application.
 * Instantiates an instance of this class, and runs it.
 *
 * @param args an array containing the command line arguments.
 * @throws IOException if files cannot be created
 */
public static void main(String[] args) throws IOException {
    // ...
```

A Javadoc comment starts with the special comment start indicator of `/**`, and continues until the comment closing indicator of `*/`. Tags are noted by the at symbol (`@`). All text from the comment start indicator through to the first tag is included directly in the generated output. Javadoc will ignore unknown tags and plain text after *any tag* has been found up until the next known tag, or the end of the doc comment—this bit of magic is how XDoclet can work.

The Javadoc code above produces the following HTML:

```
<A NAME="main(java.lang.String[])"><!-- --></A><H3>
main</H3>
<PRE>
public static void <B>main</B>(java.lang.String[] args)
                throws java.io.IOException</PRE>
<DL>
<DD>The <b>main</b> starting point for this application.
 Instantiates an instance of this class, and runs it.
<P>
<DD><DL>
<DT><B>Parameters:</B><DD><CODE>args</CODE> - an array containing the command
line arguments.
<DT><B>Throws:</B>
<DD><CODE>java.io.IOException</CODE> - if files cannot be created</DL>
</DD>
</DL>
```

When viewed in Microsoft Internet Explorer, this will appear similar to Figure 2-1.

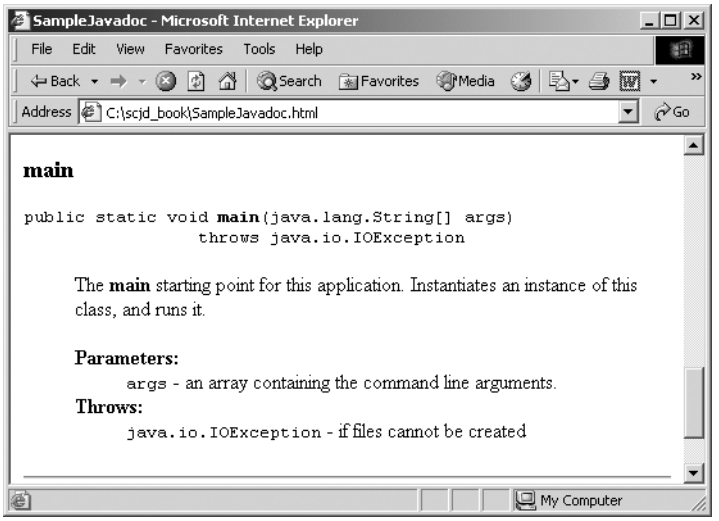


Figure 2-1. *Example Javadoc output*

■ **Note** Javadoc is more powerful than many programmers realize. It was developed as an extensible tool, which can behave in different ways depending on which module is plugged in. By default, it will parse the Java source code, generating API documentation. But if you were to change the plug-in to XDoclet (`http://xdoclet.sourceforge.net/`), for instance, the Javadoc engine could generate additional source code based on comments in the code you wrote (very useful when building J2EE applications—you could have XDoclet create your interfaces and deployment descriptors for you). Or you could plug in DocCheck (`http://java.sun.com/j2se/javadoc/doccheck`), and you could get a report on how well you are adhering to Sun's Javadoc conventions.

Coding Conventions

Sun published an article titled "How to Write Doc Comments for the Javadoc Tool," which is available online at `http://java.sun.com/j2se/javadoc/writingdoccomments/index.html`. A quick overview of the topics covered in this article follows.

What to Write

The names of your classes and methods should be reasonably self-documenting, so there is no point in writing Javadoc comments that just restate the name of the class or method. Your Javadoc comments should provide information that will help the user of your class and/or provide information that would be needed by a third party if they wanted to reimplement your class from scratch (without looking at your source code).

For example, if you were creating a class to provide a network connection to your server, you might call the class `NetworkConnection`. There is no point in having a Javadoc comment that states that this class provides "a network connection"—the user already knows that. Instead, you might consider specifying that the class provides "a connection to the database server over the network through which remote access to the database functions can be used."

■ **Caution** You should not write any implementation-specific details in your Javadoc comments.

Where to Put your Javadoc Comments

In theory, Javadoc comments can go anywhere in your source code. However, the Javadoc application will only include comments in specific locations:

- Class comments should appear immediately before the class declaration (Javadoc currently allows the comments to appear anywhere before the class declaration, even before the package declaration, but this behavior should not be relied on).
- Class and instance variable comments should appear immediately before the class or instance variable to which they belong.
- Method comments should appear immediately before the method signature.

The Javadoc tool will ignore Javadoc comments in any other locations entirely. If you want to put a comment inside a method, for example, there's no point in making it a Javadoc comment, as it won't appear in any generated API. Just use an implementation comment.

Formatting Codes and Special Tags

Javadoc comments can contain any HTML markup tags. Commonly used tags are shown in Table 2-4.

Table 2-4. *Commonly Used HTML Tags*

Tag	End Tag	Description
`<code>`	`</code>`	Text between these two tags will appear in monospaced font in the HTML output. This is usually designed for short words or phrases.
`<pre>`	`</pre>`	Text between these two tags will appear in monospaced font in the HTML output, maintaining your indentation. This is usually used for example code.
`<ul>`	`</ul>`	Denotes the start and end of an unordered list (not numbered).
`<ol>`	`</ol>`	Denotes the start and end of an ordered list (numbered).
`<li>`		Denotes the start of a list item within either an unordered or an ordered list.
`<p>`		Denotes the start of a new paragraph.

■ **Note** Javadoc produces output conforming to HTML standard 3.2, which does not require closing tags for certain elements (e.g., there is no requirement for the `<li>` tag to be closed with a `</li>` tag, nor for the `<p>` tag to be closed with a `</p>` tag); however, you may use them if you wish. In addition, the HTML 3.2 standard does not specify whether tags should appear in upper- or lowercase. Throughout this book we will use HTML 3.2 tags, but we will format them according to the later HTML 4.0 and XHTML formats (we will use closing tags, and we will put the tags in lowercase). You may also use tags and HTML constructs that were only created in HTML version 4 or later if you wish (Javadoc will handle them without problems). However, if you do so, you may find that some older browsers may not be able to display your generated documentation.

There are many more HTML codes you may use in your Javadoc comments. You generally don't need many of them, though, as you are generating API documentation for other programmers to read.

■ **Tip** You should use the `<code>` tag for Java keywords, package names, class and interface names, method names, field names, and argument names to make them appear in monospaced font when the browser supports it.

In addition to the HTML tags, Javadoc recognizes special *Javadoc tags*—and they will be handled in special ways. The Javadoc tags for classes and interfaces are shown in Table 2-5. The tags are listed in the order they should appear in your Javadoc comment.

Table 2-5. *Class and Interface Tags*

Tag	Description
`@author`	The name of the person who wrote the class or interface.
`@version`	The current version of the class or interface. You might choose to have your revision control software set this automatically as you check your code out of source control.
`@see`	Generates a link to the specified class or method—the link will appear in a special "See Also" section of the generated Javadoc. Refer to the comments below regarding the `@see` and `@link` tags for special instructions.
`@since`	Used to indicate the version number of your release in which this class first appeared.
`@deprecated`	Used to indicate that a class should no longer be used but still exists for compatibility reasons. The tag should be followed by text indicating what class the user should use instead of this class, or "No replacement" if there is no class with replacement functionality.

Tag	Description
`@serial`	Used to specify whether a class or field that would normally be serializable should be documented as being `Serializable`.
{`@link` *reference label*}	Generates an inline link to the specified class or method. Refer to the comments below regarding the `@see` and `@link` tags for special instructions. The label will appear in monospaced font.
{`@linkplain` *reference label*}	Same as the {`@link`} tag except that the label will appear in standard font.
{`@docRoot`}	Points to the base directory (where the `index.html` file resides) for your generated Javadoc. This will be correct no matter how many directories deep this tag is used.

The Javadoc tags for fields are shown in Table 2-6.

Table 2-6. *Field Tags*

Tag	Description
`@see`	Generates a link to the specified class or method—the link will appear in a special "See Also" section of the generated Javadoc.
`@since`	Used to indicate the version number of your package in which this field first appeared.
`@serial`	Used to specify whether a class or field that would normally be serializable should be documented as being `Serializable`.
`@deprecated`	Used to indicate that a field should no longer be used but still exists for compatibility reasons. The tag should be followed by text indicating what field or method the user should use instead of this field, or "No replacement" if there is no field with replacement functionality.
{`@link` *reference label*}	Generates an inline link to the specified class, method, or field. The label will appear in monospaced font.
{`@linkplain` *reference label*}	Same as the {`@link`} tag except that the label will appear in standard font.
{`@docRoot`}	Points to the base directory (where the `index.html` file resides) for your generated Javadoc. This will be correct no matter how many directories deep this tag is used.
{`@value`}	Displays the value of the constant being specified.

The Javadoc tags for constructors and methods are shown in Table 2-7.

Table 2-7. *Constructor and Method Tags*

Tag	Description
`@param`	Describes a parameter in the method signature. Each parameter should be described on its own line, with the list following the same order as the parameters in the method signature.
`@return`	Describes the value or object returned from the method.
`@exception`	Describes an exception that may be thrown from this method. (`@throws` is a synonym added in Javadoc 1.2.)
`@see`	Generates a link to the specified class, method, or field—the link will appear in a special "See Also" section of the generated Javadoc.
`@since`	Used to indicate the version number of your package in which this constructor or method first appeared.
`@deprecated`	Used to indicate that a constructor or method should no longer be used but still exists for compatibility reasons. The tag should be followed by text indicating what constructor or method the user should use instead of this constructor or method, or "No replacement" if there is no constructor or method with replacement functionality.
{`@link` *`reference label`*}	Generates an inline link to the specified class, method, or field. The label will appear in monospaced font.
{`@linkplain` *`reference label`*}	Same as the {`@link`} tag except that the label will appear in standard font.
{`@docRoot`}	Points to the base directory (where the `index.html` file resides) for your generated Javadoc. This will be correct no matter how many directories deep this tag is used.

The `@see` and `@link` tags will create hyperlinks to the first matching class, field, or method. The class name does not need to be specified when referring to a method or field within the current class; likewise, the package name does not need to be specified when referring to a class within the same package. Finally, unless you want to refer to a specific overloaded method, you do not need to specify the method parameters. Some examples are shown in Table 2-8.

Table 2-8. *Examples of @see Links*

Example	Comment
`@see #field label`	Creates a link to the specified field with the given label
`@see #method label`	Creates a link to the first matching method and gives the link the specified label
`@see #method (parameters) label`	Creates a link to the method with the matching signature and gives the link the specified label
`@see class#method label`	Creates a link to the first matching method in the named class and gives the link the specified label
`@see package.class#method label`	Creates a link to the first matching method in the named class in the named packages and gives the link the specified label

JDK 1.5 introduced two new tags that can be used anywhere within your document: {@code *text*} and {@literal *text*}. In both cases, the text will be displayed without interpretation. For example, if your text was <b>, normally this would be interpreted as the HTML tag to start bold text. However, using the tag {@code <b>} will result in the text <b> being displayed as desired. The {@code *text*} tag will display the text in monotype font and {@literal *text*} will display the text in normal font.

Package-Level Documentation

Javadoc can also incorporate package documentation into the generated output, and create a summary page for your overall submission. An example of the summary page is shown in Figure 2-2.

Figure 2-2. *Sample summary page*

To create package documentation, you will need to create a file named `package.html` in each package directory. A simple example might look like this:

```
<!doctype HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<html>
<head>
<title>com.example.db</title>
</head>
<body>
Provides database functionality.
<h2>Package Specification</h2>
These classes are the base that provide the basic data creation, reading, updating,
and deleting functions required to process data in a database. Basic locking
functionality exists, but this is not an advanced package.

@author Unattributed person from example.com.
@author Andrew Monkhouse
@version 1.0
@since 1.0
</body>
</html>
```

JDK 5 Changes

The obvious changes to the Javadoc tool from earlier versions include support for the two new tags ({`@code` *`text`*} and {`@literal` *`text`*}) mentioned earlier, and the obvious support for the generics, enums, and VarArgs features.

When JDK 1.4 was released, Sun was planning to change the way Javadoc would determine how much of a comment should appear in the summary (where the break should be between the summary and the remainder of the comment). Sun incorporated the logic for the new break iterator in the Javadoc tool, which resulted in large numbers of warnings being generated when the proposed change would cause different summaries to appear in the future version. Sun has now reversed their plans, and removed the 1.4 logic.

Running Javadoc from the Command Line

The basic command format for Javadoc is

```
javadoc [options] [packagenames] [sourcefiles] [@files]
```

Following from this, you could generate the API documentation for all your source files that do not belong to any package by typing the following command in the same directory as the source files:

```
javadoc *java
```

If you have your source files organized in packages, you could list the package names on the command line instead:

```
javadoc com.example.mypackage com.example.more.packages
```

Both those command lines will generate the API documentation in the current working directory, which may not be desirable. You can specify the directory where API documentation should be stored by using the `-d` option:

```
javadoc -d doc/api com.example.mypackage com.example.more.packages
```

■ **Tip** Javadoc will convert Unix-style pathnames into platform-specific pathnames. This is especially useful when you are developing scripts that may be run by unknown people on potentially any platform.

You can add many more options, turning features on or off as desired. For example:

```
1  javadoc \
2      -d doc/api \
3      -version \
4      -author \
5      -use \
6      -source 1.5 \
7      -windowtitle "Denny's DVD application" \
8      -doctitle "<h1>The SCJD Exam with J2SE 1.5 Project</h1>" \
9      -bottom "<i>Developers: Andrew Monkhouse and Terry Camerlengo</i>" \
10     -sourcepath src \
11     -linkoffline http://java.sun.com/j2se/1.5.0/docs/api /jdk_1.5/api \
12     sampleproject.gui sampleproject.db sampleproject.remote
```

The meaning of the new tags is explained in Table 2-9.

Table 2-9. *Meaning of Common Javadoc Command-Line Options*

Line	Option	Meaning
2	`-d`	Used to specify the output directory for generated API.
3	`-version`	Specifies that the version information in your Javadoc comments will be incorporated into the generated API.
4	`-author`	Specifies that the author information in your Javadoc comments will be incorporated into the generated API.
5	`-use`	Tells Javadoc to include a "use" page for each class, detailing where it is used as either a parameter in a method or as a return value.
6	`-source`	Specifies which version of the JDK the Javadoc should maintain compatibility with.
7	`-windowtitle`	Specifies the title to appear at the top of your browser (where supported).
8	`-doctitle`	Specifies the title to appear on the package summary page.
9	`-bottom`	Specifies text that appears on the bottom of every generated page.
10	`-sourcepath`	Specifies the base directory containing your packages.
11	`-linkoffline`	Creates links to existing Javadoc entries. See the description in the paragraph that follows.

The `-link` and the `-linkoffline` options enable you to link to other preexisting APIs. For example, if you used the `String` class as a parameter in one of your methods, you might like to have your API documentation for your method include a hyperlink back to Sun's API documentation for the `String` method.

You would normally use the `-link` option whenever you do not have a copy of the preexisting API local to you, but you can access it over the Web.

The `-linkoffline` option is preferred when you have a local copy of the preexisting API documentation, or when you have a local copy of the package-list file from the preexisting API documentation, or when you cannot access the preexisting API over the Web.

■ **Tip** Whenever possible you should refer to local documentation rather than looking at it over the Web, simply because it will be faster for you. Likewise, linking "offline" by using your local copy of the documentation will be faster than accessing the required file over the Web.

Obviously, you would not want to type such a complex command line every time you wanted to regenerate your API documentation. One way of working around this is to put all the options (one per line) into a plain text file, and then refer to that file on the command line. For example, if you had put all the options in a file named `javadoc.options`, you could use the following command line to generate your API documentation:

```
javadoc @javadoc.options sampleproject.gui sampleproject.db sampleproject.remote
```

Working with Packages

If you were to put all the files on your computer into one folder, this one folder would quickly become unmanageable, and finding any particular file would be a nightmare. To avoid this, you probably organize folders to store related files—one for accounting information, another for job hunting, another for music, and so on. Some of these folders may have subfolders to provide further subcategories.

Developing software has the same potential issue. Fortunately, Sun has provided us with a platform-independent equivalent of the folders: packages. You can locate classes belonging to a specific package by the fully qualified package and class name, even if the operating system has no concept of folders, directories, or a hierarchical file system.

When starting any project, consider what logical modules or functionalities your project might have, and place the classes related to that functionality into its own package. In the case of our project, we will have a set of classes that provide the graphical user interface, another set of classes that provide network functionality, and another set of classes that provide data access functions. We could therefore start with the following potential packages:

```
gui
network
database
```

After further investigation, we may decide to add more packages and/or refactor the existing packages—Chapter 3 will discuss how to analyze the sample project. For example, we might decide to subdivide the `network` package into two separate packages: one for RMI networking, and one for Sockets networking. We would therefore have the packages

```
network.rmi
network.sockets
```

Package names should be qualified to ensure that each class can be uniquely identified. That is, when you attempt to instantiate the `StartGui` class, you want to instantiate *your* class of that name, not mine. To do this, you create packages for your domain name, written in lowercase and in reverse. So if you worked for a company with the domain name `example.com`, then your package names should start with the same name in reverse (`com.example`). So our fully qualified package name would now become `com.example.network.sockets`.

On Microsoft Windows and Unix-based systems, each package should be in a separate directory. So for the `com.example.network.sockets` package mentioned earlier, we would have a directory for the first package `com`, which would contain a directory for the `example` package, which in turn would contain the directory for the `network` package, which finally would contain a directory for the `sockets` package. This final directory would contain all the Java class files that belong in that package.

■ **Note** Many operating systems differentiate between case for files and directories. So a file named `numberOfBoxes` will be different from a file named `NumberOfBoxes`. Since Java must request files from the underlying operating system, and cannot reasonably go through every permutation of upper- and lowercase, Sun has stipulated that the case of your files and directories must match the case of your classes and packages.

The Sun Java utilities are all designed to work with packages in the same way, so you can be sure that if the Java compiler can find and compile your classes, then the Java runtime and the Java documenter will also be able to work with your classes.

JAR files can also act as a package structure. After you change directory to the classpath root folder, running the command

```
C:\devProj\classes\> jar cf db.jar sampleproject.db
```

will result in a compressed file named `db.jar` containing the `sampleproject.db` package tree. Therefore, this file can now be added to a classpath the same way a directory is specified, and the compiler or Java Virtual Machine (JVM) will scan the file for referenced packages.

It is important to note the behavior of Java utilities when dealing with shared package definitions. For example, a situation could arise under which two directories containing different portions of a package definition exist on a computer's file system.

The classes for the first package are stored in `c:\devProj\classes`, and the second set of files is stored outside of the project's directory structure in `c:\tempClasses`. In the first directory, the package `sampleproject.db` is stored, while `sampleproject.testclient` is stored in

the second directory (this means that the files will be stored in the directories `c:\dev\Proj\classes\sampleproject\db\` and `c:\dev\Proj\classes\sampleproject\testclient\`, respectively).

Next, both directories are compressed into separate JAR files using the JAR utility, as shown here:

```
C:\devProj\classes\> jar cf db.jar sampleproject.db
C:\devProj\classes\> jar cf testclient.jar sampleproject.testclient
```

Finally, both JAR files are specified in a classpath when compiling a Java application, as shown here:

```
C:\devProj\tmp\> javac -cp c:\devProj\classes\db.jar;c:\tempClasses\testclient.jar
 MyClass.java
```

The question at this point is, what happens to the classpaths? Does the first JAR file overwrite the package path of the second file? The answer is no. Packages with the same package tree can be stored in separate JAR files or separate directories, and in this case, the Java compiler will combine the contents of these two files at runtime.

■ **Caution** If you have classes with the same name that are in the same package namespace in two JAR files, the first one found (based on classpath order) will be used—the second will be ignored.

The ability to store package definitions in different locations allows for a very modular structure. Thus, you should consider certain ideals when planning an application's package structure. First and foremost, packages should consist of all classes that are similar in function or are dependent on each other in design. This allows for the packages to function as complete units of functionality. Thus, a JAR file could conceivably contain an entire functional package that is "plugged" into a project.

■ **Note** The sample project introduced later in this book uses package structure to isolate the different network implementations. This structure enables the entire networking layer of the sample application to be changed without impacting the application's other packages.

Classes in the base node of every package should be the building blocks of that tree's functionality, and therefore they should be the most stable portions of the tree. Lower-level packages designate the stable classes that all others are built upon, and package definitions should be planned using this principle.

If it is not possible to place only stable, nonchanging classes in the base nodes of a package, the classes should be replaced by interfaces, and the volatile implementations of these interfaces moved to a higher level of the package definition or into a whole different package altogether.

It is also very important to ensure that dependencies between classes are contained within a single package. Essentially, packages should be autonomous units that can be compiled separately. For the most part, if classes in a package have to be compiled in order for another package to compile, the package structure should be reworked so that all class dependencies fall within a single package. A proper package design will also ensure that an application is built on the most stable foundation possible.

In summary, packages act as a mechanism to compartmentalize portions of a project into functional blocks. Thus, an effective package design allows for portions of a project to be removed and swapped around without impacting other portions of your code.

Best Practices

In your programming career, you may have noticed practices that you (or others) follow that do not fit into the categories above, and yet they make the overall delivery of a final solution much easier. While these practices are not required, you should consider using them in your assignment.

Writing Documentation As You Go

It seems strange to have to mention this, but you really must write your documentation at the same time as you are working on your assignment, preferably before you write your code, or as you write it.

You must provide three major forms of documentation as part of this assignment:

- Design choices
- Javadoc
- User documentation

Design Choices

The design choices document will contain a quick overview of what *major* choices you made while developing your solution, and what alternatives you considered—possibly explaining why you discarded the alternative.

You do not need to write a book on your choices—the assessors are only interested in two things:

1. Did you consider alternatives?
2. Did you write the code you submitted?

If you go on to become a system architect, you will be expected to *know* multiple ways of achieving any software goal—often using multiple architectures and languages. As a developer, you should also be able to consider and reject alternative solutions within your area of expertise—J2SE. For example, in a case where you could use a radio button or a check box, you should be able to recognize that both alternatives are possible, and decide which is the correct choice (note that the choice between a radio button and check box is *probably* a minor decision, and possibly not something you would want to document).

■ **Tip** Where possible, you should also put your design decisions in an implementation comment close to the code itself. For example, if you had chosen a radio button instead of a check box, having an implementation comment near the constructor for that button will save the person maintaining your code from spending time trying to decide whether this was the "right" choice or not.

Sun has a difficult task—how do they prove that *you* wrote the assignment that *you* submitted? One step in the process is to have multiple assignments (different business domains), each with multiple versions (different interfaces to be implemented), so it is unlikely that anyone you know will happen to have the same assignment as you. Another step is to check that the information you write in the written exam (where you must provide proof of ID) matches the design choices document, and that the design choices document describes the code. Combine these, and Sun can be reasonably certain that the person who sits for the exam is the person who wrote the code.

It is very important that you write down your design decisions as you make them. If you leave that task until the end of the project, you will have a hard time remembering what choices you made, let alone why you made them.

Javadoc

It has been shown time and time again that if the documentation is separate from the code, then it is often not updated at the same time as the code is updated, which renders it useless.

Javadoc, and similar code documenting tools, were designed to solve this problem. By putting the code documentation with the code itself, it is much easier to update the documentation at the same time the code is updated.

But this relies on one major premise—that you write and update the comments at the same time as you write and update the code.

If you wait until the end of a three-month assignment to do the source code commenting, you may find that you have another three-month job ahead of you just to finish the comments, simply because you have to reread all your code to work out what you were doing in order to document it.

User Documentation

In the section "Prototyping the GUI," we recommended drawing rough sketches of your user interface before starting the project. One of the benefits we mentioned is that you will then have a definite design to work toward, reducing the risk that you will end up with a less usable interface simply because it is easier to implement.

When you write the user documentation before coding, you get the same benefit: you have described up front what your user interface is going to do, so you have a definite target to achieve.

You also have a definite end point in mind—when your user interface meets your user documentation, you know that the user interface should be ready for testing.

User documentation is also where users go when they have a problem, or when they want to determine how they can do something more advanced. Incorrect or badly worded user

documentation will cause frustration for the end user, and in your real employment will cause your support calls to be more aggravating than they need to be. Writing your documentation up front means that it is less likely to be rushed, and also means that you are more likely to go back to it several times during the course of your development and improve it.

Assertions

Assertions were added in JDK 1.4, and provide a useful confirmation and documentation of assumptions for you as the developer without affecting deployment of your application.

The syntax of assertions is as follows:

```
assert <expression with a boolean result>;
```

or

```
assert <expression with a boolean result> : <any statement>;
```

If `<expression with a boolean result>` returns false, and assertion checking is turned on at deploy time, an `AssertionError` is thrown.

Programmers sometimes make assumptions about how two parts of their program interact—for example, they might assume that the value of a parameter in a private method will never exceed a certain value. However, if the assumption is incorrect, the problems it causes might not become obvious for quite some time. In such a case, it can be worthwhile to include an assert statement at the start of the method that can be used while testing to confirm your assumptions but that will not have any effect on the deployed code.

■ **Caution** You should never use assertions to validate the inputs on public methods—these are methods that other programmers may use, and they may not honor your requirements. You should validate these inputs regardless of whether or not assertions are turned on at runtime.

Here is some example code to show validating a value:

```
public class AssertionTest {
    public static void main(String[] args) {
        new AssertionTest(11);
    }

    public AssertionTest(int withdrawalAmount) {
        int balance = reduceBalance(withdrawalAmount);

        // reduceBalance should never return a number less than zero
        assert (balance < 0) : "Business rule: balance cannot be < 0";
    }
}
```

■ **Note** In JDK 1.4 you had to specify the `-source 1.4` option to the Java compiler in order to enable the compilation of assertions. In JDK 5 this is no longer required.

By default, assertions are turned off at runtime. This has two major benefits:

- `AssertionError` is an `Error`, and as such you do not want it thrown in production.
- Evaluating the expression in the assertion could be time consuming—having it switched off by default ensures better performance.

■ **Caution** You should never use assertions to perform actions required for the method to work—assertions should only validate that the values are correct. Since assertions are normally switched off at runtime, any actions you perform within the assertion will not normally be performed at runtime.

To enable assertions at runtime, you must specify either the `-ea` or the `-enableassertions` command-line option. For example:

```
java -ea AssertionTest
```

Without the `-ea` option, the `AssertionTest` program will run without errors. With the `-ea` option, it will throw an `AssertionError`.

Assertions can be enabled for individual classes or packages as well, by specifying the classes or packages on the command line:

```
java -ea:<packageName> -ea:<className>
```

Logging

When debugging, you will find it useful to know when you have reached a certain method, and what the values of some of your parameters and variables are.

In a debugger you might set breakpoints at certain locations and watches on some variables. But this can be tedious to do each time you want to debug a program.

The next logical step might be to add `System.out.println(...);` statements throughout your code. Watching the output in the command window will then give you an idea of what your program is doing. However, this is not a good idea for code that someone else (your client or assessor) is deploying: at best it is distracting for them; at worst they may assume that the "normal" debug messages are signs of an error. In addition, any application that may be deployed as a server application may not even have a window in which to watch the messages. And, if something does go wrong with your program, it can be difficult to get the user of the program to copy the correct messages and send them to you. Furthermore, some of the messages you want to appear while debugging the application make no sense at deploy time—you would want to be able to selectively turn off some of the messages.

Obviously this problem has cropped up many times, and there is a common solution: use a logging framework. Doing so ensures that we can

- Send logging messages to a desired location (file/screen/printer/nowhere)
- Log messages even if the application is running in "server" mode and does not have a window for messages to appear in
- Ensure that logging is done in a consistent manner throughout the entire application
- Turn logging on and off selectively throughout the application at deploy time

JDK 1.4 and later provide a standard API for logging. We recommend that you use this API throughout your application whenever you consider sending something to the standard or error output.

To use the logger, you must first get an instance of a logger you can use. Loggers are normally named (although there is an anonymous logger that everyone can use), allowing you to configure different loggers in different ways. To get a reference to the logger for the `example.com.testApplication` context, you would create code similar to this:

```
Logger myLogger = Logger.getLogger("example.com.testApplication");
```

All classes that "get" a logger using the same name will get the same instance of the logger. This ensures that the configuration applied to that logger will apply to all instances of that logger.

■ **Note** The name of the logger is part of a hierarchical namespace, meaning that `example.com.testApplication` is a child of the namespace `example.com`. We will comment further on this later in this section.

You can define the level at which your messages will be logged. There are several predefined levels, and we recommend that you use them rather than define your own. The predefined levels are shown in Table 2-10 in order from the least amount of logging to the most amount of logging.

Table 2-10. *Predefined Logging Levels*

Logging Level	Recommended Usage
Severe	Serious failure
Warning	Potential problem
Info	Informational messages
Config	Configuration messages
Fine	Tracing (debugging) messages
Finer	Fairly detailed tracing messages
Finest	Highly detailed tracing messages

The logger class has several utility methods for logging information (`Logger.info(<msg>)`), warning (`Logger.warning(<msg>)`), and severe (`Logger.severe(<msg>)`) messages, allowing you to log simple messages, for example:

```
myLogger.severe("Sending message to standard error");
```

If you have been putting these commands into simple test applications, you may be wondering what the fuss is about—after all, so far we have not done anything that we cannot do with a `System.out.println(<msg>)` statement. But consider that you could turn off all logging to your *entire* application with one statement near the start of your application, such as

```
myLogger.setLevel(Level.OFF);
```

That is, all your classes could be logging messages, at different levels, and that one line could turn them all off—no need to go hunting for all those `System.out.println(<msg>)` statements.

You may want to execute a block of logging code only if you know that it is actually going to be logged. In such a case, you can make a call to the `isLoggable(<level>)` method first to determine whether you should call your (potentially performance-reducing) logging code.

One benefit we mentioned earlier was that we could log messages to a file—this is done through a `Handler` object. We can add a simple file handler to our logger with one simple command:

```
import java.util.logging.*;
import java.io.IOException;

public class TestLogging {
    public static void main(String[] args) throws IOException {
        Logger myLogger = Logger.getLogger("Test");
        myLogger.addHandler(new FileHandler("temp.log"));
        myLogger.severe("My program did something bad");
    }
}
```

■ **Note** The `FileHandler` class can handle storing logs in common locations regardless of operating system, and can handle the usual issues such as rotating log files. Refer to the API for `FileHandler` to see how such options can be utilized.

Running this example produces the following log message:

```
C:\TEMP> java TestLogging
11/12/2004 19:22:40 TestLogging main
SEVERE: My program did something bad
```

■ **Note** Your time and date will obviously differ.

There should also be a `temp.log` file in the current working directory, which contains the log message in XML format:

```
C:\Temp>dir temp.log
 Volume in drive C has no label.
 Volume Serial Number is E0F1-2766

 Directory of C:\Temp

11/12/2004  07:22p                388 temp.log
               1 File(s)            388 bytes
               0 Dir(s)   2,066,956,288 bytes free

C:\Temp>type temp.log
<?xml version="1.0" encoding="windows-1252" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
  <date>2004-12-11T19:22:40</date>
  <millis>1102753360550</millis>
  <sequence>0</sequence>
  <logger>Test</logger>
  <level>SEVERE</level>
  <class>TestLogging</class>
  <method>main</method>
  <thread>10</thread>
  <message>My program did something bad</message>
</record>
</log>
```

When you have a program to analyze your log messages, you may find this useful. However, when you want to read the log messages yourself, you may find it more beneficial to have the output in plain text. You can do this by adding a `Formatter`.

Changing the code to read:

```
import java.util.logging.*;
import java.io.IOException;

public class TestLogging {
    public static void main(String[] args) throws IOException {
        FileHandler myFileHandler = new FileHandler("temp.log");
        myFileHandler.setFormatter(new SimpleFormatter());
        Logger myLogger = Logger.getLogger("Test");
        myLogger.addHandler(myFileHandler);

        myLogger.severe("My program did something bad");
    }
}
```

will result in the following output on the screen:

```
11/12/2004 19:39:27 Test main
SEVERE: My program did something bad
```

and the following output in the `temp.log` file:

```
11/12/2004 19:39:27 Test main
SEVERE: My program did something bad
```

If you wanted to have some other form of output, consider making your own formatter. The source code to `SimpleFormatter` is available in the JDK sources, and is only 80 lines for the entire formatter—it would be easy to create your own formatter.

You will have noticed that logging has so far been going to the screen, even though we have not asked for this. This is because by default each logger will use its parent's log handlers in combination with its own. The default anonymous logger will, by default, send output to the screen. That is, if you get a logger for `example.com`, and another logger for `example.com.test`, the logger for `example.com.test` is a child of `example.com`. So any message logged to the `example.com.test` logger will also be sent to the handler for `example.com`. You can turn this behavior off by calling the `setUseParentHandlers(boolean useParentHandlers)` method on the child logger.

You should not remove logging code before deploying in production. Doing so runs the risk that you may remove something you didn't intend to remove, and if you later need the logging back again, you will have to add it all again manually. It is a much better idea to use the logger's rotating log scheme to ensure that log files never get too large, in combination with setting the log level so that you are not producing too many log messages—usually either the `Level.WARNING` or `Level.SEVERE` log level. You would normally also turn off any logging to the screen (if you have not already done so).

Summary

Spending a little bit of time up front in planning your project can pay big dividends in reducing total time spent on this assignment. Ensuring that your code is easy to read and maintain will win you better marks in this assignment, as well as more respect from your colleagues at work. And using the tools provided by Sun can make your submission far more professional.

We have introduced some of the features that will help you make a more professional submission, and assist you in your day-to-day life.

FAQs

Q Is it necessary to follow the methodologies and standards in this chapter?

A There are requirements in your instructions that you must follow, and using Javadoc is one such requirement. However, the use of methodologies and standards is not always fully stipulated—if you do not have a specific requirement, you can ignore the standard, but you do so at your own risk.

Q Should I include unit tests in my submission?

A Current assignments have a warning stating that you will not get extra credit for going beyond the specifications. That being the case, we recommend that you do not include the test cases in your submission: at best the assessor will ignore them, and at worst you may receive a lower score if the assessor finds a fault with additional code. Plus if you use JUnit, your test code won't compile without `junit.jar`—and that's external code you're not allowed to include for the assignment.

Q Should I leave logging code in my submission?

A As mentioned in the section on logging, we recommend that you leave the logging in your code—this saves you the effort of trying to remove it, and more importantly, saves you the effort of trying to re-add it if you later need it again. (Yes, we know this seems to contradict the answer about unit tests, but unit tests are separate from your major project code, and logging is integrated into your project code. Also, `java.util.logging` *is* part of the standard libraries, whereas JUnit is not. So that's another reason why it's okay to include logging code in your assignment, but not JUnit code.)

Q Can a custom directory structure be used instead of the one presented in the book?

A For your Sun assignment, check the instructions carefully—if they specify a directory structure, then you *must* follow it. Outside of any Sun restrictions, any directory structure may be used. If you intend to follow along with the sample project in this book but decide to change the directory structure, then the instructions and examples in the book may need alterations in order to run as explained. You should only use custom directory structures if you are already comfortable with Javadoc, classpaths, and package structures.

CHAPTER 3

Project Overview

In this chapter, we introduce the sample application, which will serve as a wellspring for the myriad of topics required for developer certification. The sample project, Denny's DVDs, has a structure and format similar to the one you will encounter during the Sun Certified Java Developer (SCJD) exam, and it will demonstrate each of the essential concepts necessary for successful completion of the certification project. Each chapter adds an integral component to the project and builds from the preceding chapter, so that by the end of the book you will have a complete and properly functioning version of Denny's DVDs version 2.0. As an added benefit, Denny's DVDs utilizes Java 2 Platform Standard Edition (J2SE) 5, and some of the "Tiger" features, such as autoboxing and generics, are elucidated in the following chapters.

Note "Tiger" is the code name for J2SE 5, and the two terms will be referred to interchangeably throughout the book. Sometimes we will use the term "Tigerize," which means to add a J2SE 5 language feature to code originally composed as a J2SE 1.4 program. We do this quite a bit in the sample project, and even the `DOSClient` makes use of generics. For more information on J2SE 5 and the plethora of new features that have been added (some of which are discussed in this book), go to `http://java.sun.com/developer/technicalArticles/releases/j2se15`.

What Are the Essential Requirements for the Sun Certification Project?

To demonstrate "developer-level" competency for Sun certification, your project submission must successfully accomplish the following objectives:

- It must implement an application interface provided by Sun.
- It must use either RMI or serialized objects over sockets for networking.
- It must use Swing with a `JTable` for display.
- It must be entirely contained within one single executable JAR and, consequently, should not require any command-line options to run.
- Configuration settings must be persisted between application runs.

Your project submission must consist of these components:

- An executable JAR file, which will run both the stand-alone client and the network-connected client
- An executable server-specific JAR file, which will run the networked server
- A common JAR file, which will contain code common to both client and server applications:
  - An `src` directory containing the source files for the project
  - The original data file supplied with the instructions
- A `docs` directory that will contain
  - The API generated by Javadoc
  - The end-user documentation
- The file summarizing your design choices

The entire submission should be packaged in a JAR file, as described in Chapter 9.

■ **Note** JARs were initially created to allow applets to be downloaded in a single HTTP request, rather than multiple round-trips (request-response pairings) to retrieve each applet component, such as class files, images, and so forth. Executable JAR files have file associations so that clicking on them will run `javaw -jar` on Windows and `java -jar` on Unix.

The Denny's DVDs sample will eventually be bundled into one JAR file. That JAR file will contain an executable JAR file named `runme.jar`, containing the database files, documentation, and source code. The `runme.jar` will handle both the server and the client depending on how it is invoked. Running the final application from the executable JAR file will be explained in Chapter 9.

■ **Note** The previous edition of this book included a chapter on Java's New I/O, or NIO. At the time, NIO was a new 1.4 topic, and the extent that NIO could be used for the certification exam was unclear. Subsequently, after the publication of the first edition and the release of J2SE 1.4, Sun explicitly disallowed the use of NIO as a networking solution for the certification project, but permitted its use as a mechanism for file I/O. Admittedly, the main reason for including NIO in the previous edition was just to demonstrate this cool new technology. Our primary examples (downloadable from the Apress web site) did not use NIO in the networking layer. Unfortunately, there was some confusion since many believed that the inclusion of NIO indicated that our proposed solution required NIO. It did not. For this reason, we have decided to drop the discussion of NIO in the current edition of this book to avoid any further confusion, but would like to make it clear that sockets or RMI are required for the networking layer but that channels can safely be used for plain old file I/O.

Introducing the Sample Project

Denny's DVDs is a DVD rental store for a small community. The certification project requires that the application be built on a provided interface. Sun will include an interface as part of the assignment, and you will be responsible for implementing that interface and developing a fully featured application. Our sample project will take the same approach. We will imagine that the infamous Denny of Denny's DVDs will define an interface that he expects the developer (i.e., you the reader) to implement. That interface, `DBClient.java`, will be our starting point for development. Figure 3.1 shows a UML use case diagram for the primary operations the system must support.

■ **Note** The acronym *UPC* will be bandied around quite a bit, so a little background information may come in handy. UPC, which stands for Universal Product Code, is an official designation for a product, whether it be a book, a CD, or a DVD. It is a unique number identifying that product worldwide. For more trivia-related information on UPC (and related concepts such as European Article Numbering [EAN] and checksums), refer to such sites as Wikipedia (`http://en.wikipedia.org/wiki/Main_Page`) and the various online UPC databases. Of course, such a topic is way outside the scope of this book. For our purposes, UPC is a nice surrogate for a system-wide identifier or primary key.

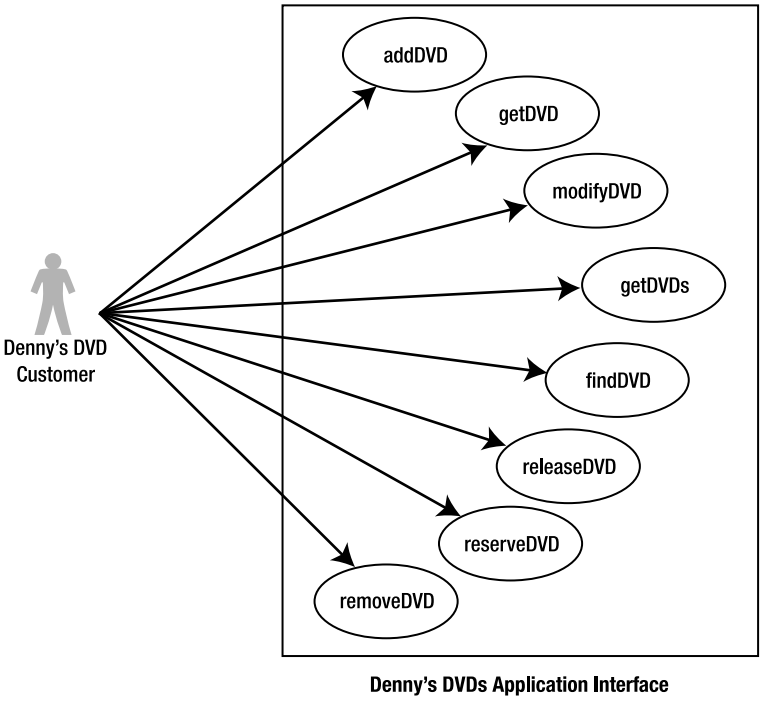


Figure 3-1. *Denny's DVDs use case diagram*

The operations in Figure 3.1 make up the public interface, `DBClient.java`, that the Denny's DVDs application must implement. Here are descriptions of the interface methods:

`addDVD`—Requires a DVD object as an input parameter. Will add the DVD to the system if the UPC is unique; otherwise, will throw an exception.

`getDVD`—Given a UPC value for a DVD, this method should return the corresponding DVD object. A null object is returned if the UPC is not found in the database.

`modifyDVD`—Requires a DVD object with a UPC that exists in the database. Will overwrite an existing DVD with new DVD values. If the database does not contain a DVD with the supplied UPC code, then false is returned and the database is not updated.

`getDVDs`—Should return a collection of all the DVDs in the database. If there are no DVDs in the database, then an empty collection should be returned.

■ **Note** The Denny's DVDs system available for download comes with a starter database, which is referred to throughout the book's examples.

`findDVD`—This operation should use a regular expression query as the input parameter. The supplied input parameter should match on multiple attributes with a regular expression query. The input parameter is the regular expression query, so the application must translate the user's search criteria into a regular expression query prior to invoking this method. Transforming the user's search criteria into a regular expression can be done either in the GUI or via some intermediate class, but not in the `findDVD` method itself. Of course, you must not require or expect the user to enter the regular expression syntax as the search criteria. A collection of DVDs should be returned.

`releaseDVD`—Will increment the count of available DVDs. This operation is equivalent to a rental return. The operation should return true, which indicates whether the DVD identified by the UPC exists in the database and has copies out for rental. Otherwise, the operation returns false.

`reserveDVD`—Will decrement the DVD count indicating that someone has rented one of the available copies. Requires a UPC as an input parameter. If the UPC of the DVD to reserve does not exist, false is returned; false is also returned if no more copies of the DVD are available for rental.

`removeDVD`—Will remove the DVD with the matching UPC and return true. If the UPC is not found in the database, then no DVD is removed and the method returns false.

The interface must have a GUI to allow the execution of each of the required methods listed here. Also, the GUI must be capable of connecting to a server on a network, or work in stand-alone mode and connect to a server on the localhost. You must also consider design issues related to concurrent user access and record locking. Listing 3.1 contains the code for the `DBClient.java` interface.

■ **Tip** Before inspecting the Denny's DVDs `DBClient.java` source code available for download, try writing the class yourself and see how close your interface is to the one used in the sample project. Since you will also be given the interface in the actual assignment, this exercise should be performed just for fun.

Listing 3-1. *The DBClient Interface*

```
package sampleproject.db;

import java.io.*;
import java.util.regex.*;
import java.util.*;

/**
* An interface implemented by classes that provide access to the DVD
* data store, including DVDDatabase.
*
* @author Denny's DVDs
* @version 2.0
 */

public interface DBClient {

    /**
     * Adds a DVD to the database or inventory.

     *
     * @param dvd The DVD item to add to inventory.
     * @return Indicates the success/failure of the add operation.
     * @throws IOException Indicates there is a problem accessing the database.
     */
    public boolean addDVD(DVD dvd) throws IOException;

    /**
     * Locates a DVD using the UPC identification number.
     *
     * @param UPC The UPC of the DVD to locate.
     * @return The DVD object which matches the UPC.
     * @throws IOException if there is a problem accessing the data.
     */
    public DVD getDVD(String UPC)throws IOException;

    /**
     * Changes existing information of a DVD item.
     * Modifications can occur on any of the attributes of DVD except UPC.
     * The UPC is used to identify the DVD to be modified.
```

```
     *
     * @param dvd The DVD to modify.
     * @return Returns true if the DVD was found and modified.
     * @throws IOException Indicates there is a problem accessing the data.
     */
    public  boolean modifyDVD(DVD dvd) throws IOException;

    /**
     * Removes DVDs from inventory using the unique UPC.
     *
     * @param UPC The UPC or key of the DVD to be removed.
     * @return Returns true if the UPC was found and the DVD was removed.
     * @throws IOException Indicates there is a problem accessing the data.
     */
    public  boolean removeDVD(String UPC) throws IOException;

    /**
     * Gets the store's inventory.
     * All of the DVDs in the system.
     *
     * @return A List containing all found DVD's.
     * @throws IOException Indicates there is a problem accessing the data.
     */
    public List<DVD> getDVDs() throws IOException;

     /**
      * A properly formatted <code>String</code> expressions returns all
      * matching DVD items. The <code>String</code> must be formatted as a
      * regular expression.
      *
      * @param query The formatted regular expression used as the search
      * criteria.
      * @return The list of DVDs that match the query. Can be an empty
      * Collection.
      * @throws IOException Indicates there is a problem accessing the data.
      * @throws PatternSyntaxException Indicates there is a syntax problem in
      * the regular expression.
     */
    public Collection<DVD> findDVD(String query)
            throws IOException, PatternSyntaxException;

    /**
     * Lock the requested DVD. This method blocks until the lock succeeds,
     * or for a maximum of 5 seconds, whichever comes first.
     *
     * @param UPC The UPC of the DVD to reserve
```

```
     * @throws InterruptedException Indicates the thread is interrupted.
     * @throws IOException on any network problem
     */
    boolean reserveDVD(String UPC) throws IOException, InterruptedException;

    /**
     * Unlock the requested record. Ignored if the caller does not have
     * a current lock on the requested record.
     *
     * @param UPC The UPC of the DVD to release
     * @throws IOException on any network problem
     */
    void releaseDVD(String UPC) throws IOException;

}
```

■ **Note** The sample project is not a full-featured e-commerce system. Instead, think of it as a program that will demonstrate the concepts needed to successfully complete the project portion of the SCJD exam.

Application Overview

So, what's next? In this section, we present an overview of the new system and describe the steps necessary to successfully implement the sample project.

Architecturally, the application is a traditional client-server system composed of three key parts: the server-side database with network server functionality, the client-side GUI, and a client-side database interface that handles the networking on behalf of the user interface. Figure 3-2 shows a high-level overview of the new system.

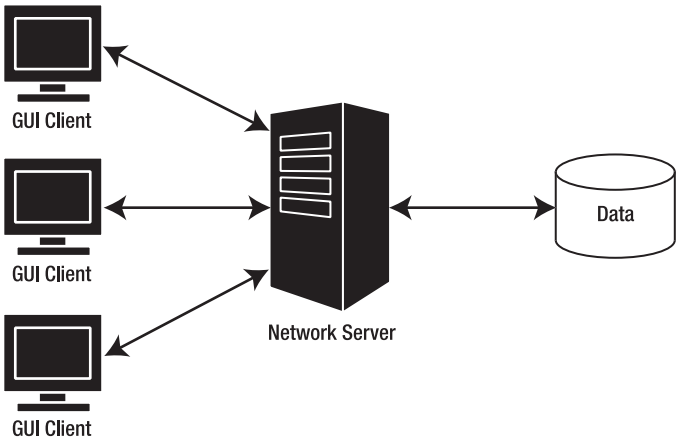


Figure 3-2. *Denny's DVDs system overview*

The users, Denny's employees, must be able to do the following:

- Perform the operations defined in the `DbClient.java` through a GUI interface.
- Enable multiuser networked access to a centralized DVD database.
- Allow network access via RMI or sockets.
- Make the database implementation thread-safe (implicitly required for multiuser networked access).

■ **Tip** Included in the source code available for download is a `DOSClient.java` class. This is a useful command-line program that was available in the download that accompanied the first edition of this book but, due to recent changes in the format of the certification project, is not necessary in the second-edition version of Denny's DVDs. Even though we do not discuss the `DOSClient.java` in this book, we include it anyway as a convenience to those downloading the code. Think of the `DOSClient.java` as a simplified command-line version of the GUI tool.

Creating the GUI

The GUI must allow an employee to view DVD information such as the UPC, title, rental status, director, actors, actresses, composer, and number of store copies. The GUI must also allow an employee to conduct a search on any of these DVD attributes. These attributes are listed in the class diagrams in Figure 3-3. The GUI should provide the user with the option of connecting to the database locally or through a network. It does not need to take into account any security features such as authentication and logon. Because the SCJD exam currently requires the use of Swing for the GUI, you will also use Swing for your project's GUI.

■ **Note** Swing is a Sun technology built on top of the Abstract Windowing Toolkit (AWT). AWT is part of the Java Foundation Classes (JFC). Knowledge of the AWT is no longer necessary for programmer-level certification, and a direct understanding of how Swing utilizes the AWT is not essential for developer certification.

DVD
upc name composer director copy leadActor supportingActor year
getUPC() setUPC(String upc) getName() setName(String name) getComposer() setComposer(String composer) getDirector() setDirector(String director) getCopy() setCopy(int copy) getLeadActor() setLeadActor(String leadActor) get SupportingActor() setSupportingActor(String supportingActor) getYear() setYear(String year) setRented(boolean renting)

Figure 3-3. *Class diagrams*

Network Server Functionality for the Database System

Version 1.0 of Denny's rental-tracking system can be run either locally or across a network. In local mode, the GUI will connect to the database only if it is located on the same machine. In network mode, the GUI should connect to the data server from any machine accessible on the network. The network implementation can make use of either sockets or RMI (see Figure 3-4). We demonstrate both approaches in Chapters 5 and 6.

■ **Note** Even though there isn't a version 2.0 discussed in this book, we will still refer to the system we are describing as Denny's DVDs version 1.0. Perhaps version 2.0 will show up in a third edition.

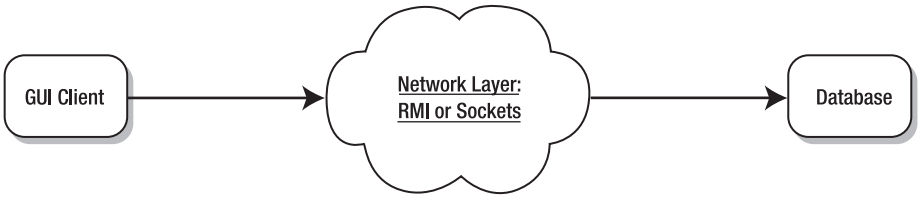


Figure 3-4. *High-level client server functionality*

Because it is possible for multiple clients to connect over a network and attempt to modify the same records simultaneously, the application must be thread-safe. We discuss thread safety in more detail in Chapter 4. For now, it will suffice to know that thread-safe code protects an object's state in situations where multiple clients are accessing and modifying the same object. It is your responsibility to make sure that your certification submission is thread-safe, as you'll learn in Chapter 4 in the section, "Understanding Thread Safety."

There is no need to notify clients of *nonrepeatable reads*—that is, it is not a requirement that all clients viewing a record that has been modified be notified of the modification. However, if two employees attempt to rent the same DVD simultaneously, then only one customer will get the DVD. This is referred to as *record locking*. In Chapter 4, record locking will be explored in more detail (with an example given in Chapter 5).

The application should be able to work in a non-networked mode. In this mode, the database and user interface run in the same virtual machine (VM), no networking is performed, and no sockets should be created. Later, in Chapter 5 (which covers networking), separate implementations for both RMI and sockets are demonstrated.

Summary

In this chapter, we introduced the public interface of Denny's DVDs rental-tracking program. We discussed the project requirements and each method of the interface that you will be responsible for implementing. All of the code for the sample application can be obtained from the Source Code section of the Apress web site (`http://www.apress.com`).

The new system should be accessible by multiple clients either across a network or locally. The GUI should be intuitive and easy-to-use, and the application must be thread-safe. The remainder of this book examines the requirements covered in this chapter. Denny's DVDs rental-tracking program will evolve as each new concept is introduced.

FAQs

Q The Sun assignment instructions tell us that we must include the instructions and data file in our submission. Why is this needed?

A There are multiple assignments available from Sun, and multiple versions of these assignments. Some of the changes in the assignment versions include variations in the data file format, the provided interface, the required class names, or any combination of these variations. Providing both the instructions *you* implemented and *your* data file ensures that the assessor will be comparing your submission with your instructions, and not anyone else's. Likewise, this means that the assessor is guaranteed to receive the correct instructions at the same time he or she receives your submission.

Q The provided interface does not include an exception I would like to throw. Can I add it?

A No. As stated in the instructions, other applications are expecting to use this interface, so if you add exceptions, the other application will need to catch them. This could prevent the other application from working.

Q I think this application will work better if I add another method to the interface. Can I do this?

A You can; however, we advise against it for the Sun assignment. Adding new interface methods will break the contract between the data formats and user interface. Additionally, if you do this on the actual assignment you may end up failing. A better approach is to add methods to the implemented classes. We encourage you to add class methods (that is, not interface methods provided by Sun) or to modify implementations as a way to better understand the project and source code.

Q How similar is this chapter's example to the one Sun will provide?

A There is no guarantee what future exams will look like from Sun, but this sample demonstrates the concepts required to achieve the Java developer certification, and it also introduces new features in J2SE.

Q How should I use the provided code samples?

A The completed version of the Denny's DVDs system is available for download. Each chapter will describe the evolution of the project from the required `DBClient` interface developed in this chapter to the remainder of the application. As you read the book, refer to the relevant section of the code base in order to understand the full implementations. Chapter 9 explains in detail how to compile and execute the entire Denny's DVDs system in a manner similar to the way your actual Sun submission will need to be executed.

An alternative approach is to study the sample project source code first and then read the book to provide an explanation for the design choices made and how those choices relate to similar decisions you will encounter while developing the actual Sun certification project. We recommend reading the book first and then referring to the code, but the alternative approach of reading the code and referring to the book should also produce successful results. It really depends on which approach makes more sense for you and your natural learning methods.

Q Can I add features not discussed in this book?

A Of course you are free to modify the project and make it more sophisticated. In fact, inquisitiveness and a sense of experimentation are very important qualities for a software engineer. However, the project has been carefully designed with two goals in mind. The first and more important goal is to cover all of the concepts required for the SCJD exam. So if you do decide to ad-lib, keep in mind the purpose of the sample project. The second goal is to introduce the new features of J2SE 5, such as autoboxing and generics.

PART 2

Implementing a J2SE Project

CHAPTER 4

Threading

Welcome to threading. In this chapter we will demystify this topic by breaking it down into manageable sections and subsections using real-world examples and metaphors. The purpose of this chapter is to explain the conceptual and technical details you need to pass the SCJD exam, including the written part of the exam.

The first sections of this chapter introduce threading and the challenges of multithreading. Waiting is explained in detail, as is locking. Finally, there is a subsection on thread safety issues, including deadlocks, starvation, race conditions, and monitors.

The sections that follow discuss the various ways that `Thread` objects can be used directly, considerations of threading when working with Swing, and the dos and don'ts of threading. Finally, we'll conclude with a FAQ section that covers some common threading questions. Specifically the following topics are covered:

- An introduction to threads and multithreading
- Locking and synchronization
- The new locking capabilities of JDK 5
- Effective thread management through waiting, sleeping, and yielding
- The importance of thread safety
- Record-locking strategies

Note This chapter is only designed to be an introduction to threading. If you want to gain an in-depth understanding of threading, we recommend Allen Holub's excellent book, *Taming Java Threads* (Apress, 2000).

Threading Fundamentals

Before you start this chapter, there are few things you should already know about threads. You should know that there are two ways to create a thread of execution in Java: the first by extending the `Thread` object, and the second by implementing the `Runnable` interface. You should know that to start a thread you must call the `start` method, and you should know that threads

can appear to do their work in parallel with each other. This is the basic understanding required for the SCJP exam, and it should be material you have mastered if you are thinking about taking the SCJD exam.

A Brief Review of Threads

A *thread* is an independent stream of execution for a set of instructions. For example, imagine a word processor needs to accomplish two activities, as shown in Figure 4-1. It needs to print and save the document you are currently working on.

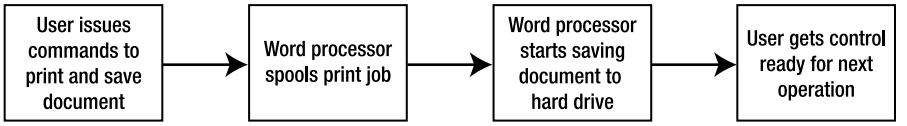


Figure 4-1. *Word processing program example*

If the word processor were single-threaded, it could not start saving the document until it was finished printing the document, as depicted in Figure 4-1. This approach would certainly work, but it would be overly inefficient, as those of us who remember the earliest word processors can attest. We can do better.

Now imagine that the program spawns two independent paths of execution, or threads. One is printing a document, and the other is quietly backing up the document in the background. This approach will yield potentially more efficient results, because there is no need for the two activities to wait on each other. For example, even as the network connection is being established with the printer, another thread could be saving the file. This concept of multithreading is illustrated in Figure 4-2 and described in detail in the next section.

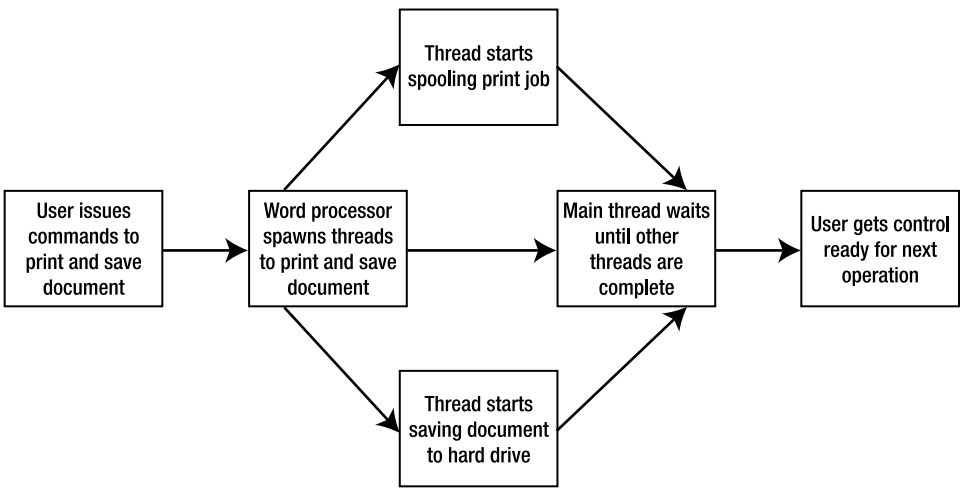


Figure 4-2. *Word processing program with threads*

Multithreading

Multithreading is the coordination of the various threads that run in a system. It is done for the purpose of improving overall system efficiency. This could mean taking turns with the resources or sharing them, decreasing overhead, or respecting the boundaries of other threads.

There are inherent challenges in managing multiple threads. These challenges arise as a result of the way in which central processing units (CPUs) handle threads. We will be discussing these challenges throughout this chapter, with explicit details in the section "Understanding Thread Safety."

A CPU will execute a number of instructions from a given thread, then switch over to executing a set of instructions from another thread, and so on. Because this switching happens quickly, an illusion of independent and parallel execution is created for all the threads. In addition, threads that are inactively waiting—say, for a network connection—will not act as bottlenecks for other threads. This is one of the greatest advantages of multithreading. However, this increased efficiency comes at a price: It is entirely possible for one thread to corrupt the data that another thread is using, unless precautions are taken. We will demonstrate these problems, and show you how to program defensively to counteract these problems in the section "Understanding Thread Safety."

■ **Note** Throughout this chapter we will be discussing threading from the perspective of the JVM operating on a single CPU computer. On computers with multiple CPUs, it is quite possible for threads to be operating on different CPUs concurrently. Regardless, the issues that arise will be the same.

Returning to the word processing program example, suppose that another thread is justifying the text, and this thread makes no effort to coordinate with the thread that is printing the document. The result might be that some of the printed document is justified and some of it is not.

These are some of the challenges that come with the territory when you start to multithread. The SCJD exam requires that you have a strong understanding of multithreading and the dangers that lurk therein.

Java's Multithreading Concepts

Java is one of the few languages that has built-in support for multithreading. Two important behaviors arise from this support. First, every Java object can be locked down for exclusive use by a given thread. Synchronizing on an object achieves this. A locked object is inaccessible to any thread other than the one that explicitly claimed it as long as all the other threads honor the locking. Second, each Java object can keep track of all the threads that want exclusive access to it. Think of this as having a sign-up sheet for each object.

The basic challenge of multithreading is similar to one of the challenges of raising young children: How do you keep the threads (children) from fighting over a limited supply of resources (toys)?

The solution, at least in Java, is twofold. First, every method that potentially uses a global object must adhere to the convention of checking to see if it is currently in use by another thread. Second, every method that modifies the object needs to put a sign on it noting that the object is currently in use. (If only it were so easy with children…)

A real-world metaphor for the latter part of the solution is signing up to use the treadmill at the gym. If you (the thread) want to use the globally available object (the treadmill), then you have check to see whether other people (other threads) are using it. If somebody is using it, then you wait. If not, then you sign it out and use it.

The first activity is referred to as waiting. The next section explains waiting in detail.

Waiting

Waiting is the act of getting out of the way when you can't be productive. As the name implies, waiting is an inactive process. This means that a waiting thread does not waste CPU time it cannot use. If a thread is waiting, then other threads are free to use its allotted quota of CPU time. The waiting thread will certainly not need it.

Imagine a group of children trying to buy ice cream from an ice cream truck. If the boy who currently has the ice cream vendor's attention does not have enough money to pay for his ice cream, he won't force everyone else to wait while his father brings him the funds he needs. He will get out the way, probably weep quietly to himself, and wait. Then he'll step back into the fray and try to claim the ice cream vendor's attention. The other children won't wait for him, because he can't order anyway.

■ **Note** The metaphor of children greedily trying to get ice cream is not an arbitrary one. Competing threads do not behave like polite adults in a grocery store line. There is no sense of order. All compete voraciously, and based on the underlying operating system, priorities, and other variables, one or another wins.

In general, there are two distinct families of pausing execution while waiting for something to happen. There is a version of pausing that maintains the lock(s) that your thread has, as well as a version that releases a lock. Typically the first type of pausing happens when you need some event to happen before continuing (for example, waiting for data to be sent over a network socket), and the second usually happens when you call some method that explicitly states that it will release the lock—for example, calling the `Object.wait` method will release the lock on the particular object it is invoked on.

■ **Note** We will explore locking in the "Locks" section of this chapter. For now, think of locking as sticking an "in use" sign on an object.

If your thread goes into a wait state by calling `Thread.yield` or `Thread.sleep`, or needs something to occur before continuing, such as receiving I/O, then your thread will maintain all locks it holds. That means that other threads will not be able to use the locked objects. If they attempt to do so, they will block until those resources are released.

Calling `myObject.wait` for a given object `myObject`, however, does release that object's lock, assuming you have the lock on that object—if you do not have the lock on that object, attempting to call the `wait` method will result in an `IllegalStateException` being thrown. This is actually very useful, because it allows other threads to modify `myObject`. This is illustrated later in this chapter in the "Ice Cream Man" example.

To understand the `wait` method, think of the minister of a church passing around a collection plate. He expects people to modify the collection plate's state—as a matter of fact he's waiting for them to do so. If they don't, he'll probably continue to wait until they do. In code, this might look something like Listing 4-1. Don't be discouraged if the code samples aren't 100 percent clear to you right now. Read the entire chapter, and it will all fall into place.

Listing 4-1. *Waiting Example*

```
public class Minister {
    private CollectionPlate collectionPlate = new CollectionPlate();

    public static void main(String[] args) {
        Minister minister = new Minister();

        // create a Thread that checks the amount of
        // money in the collection plate.
        minister.new CollectionChecker().start();

        //create several threads to accept contributions.
        for (int i = 0; i < 6; i++) {
            minister.new CollectionAcceptor(20).start();
        }
    }

    /**
     * the collection plate that get passed around
     */
    private class CollectionPlate {
        int amount = 0;
    }

    /**
     * Thread that accepts collections.
     */
    private class CollectionAcceptor extends Thread {
        int contribution = 0;

        public CollectionAcceptor(int contribution) {
            this.contribution = contribution;
        }

        public void run() {
            //Add the contributed amount to the collectionPlate.
```

```
36              synchronized (collectionPlate) {
37                  int amount = collectionPlate.amount + contribution;
38                  String msg = "Contributing: current amount: " + amount;
39                  System.out.println(msg);
40                  collectionPlate.amount = amount;
41                  collectionPlate.notify();
42              }
43          }
44      }
45
46      /**
47       * Thread that checks the collections made.
48       */
49      private class CollectionChecker extends Thread {
50          public void run() {
51              // check the amount of money in the collection plate. If it's
52              // less than 100, then release the collection plate, so other
53              // Threads can modify it.
54              synchronized (collectionPlate) {
55                  while (collectionPlate.amount < 100) {
56                      try  {
57                          System.out.println("Waiting ");
58                          collectionPlate.wait();
59                      } catch  (InterruptedException ie) {
60                          ie.printStackTrace();
61                      }
62                  }
63                  // getting past the while statement means that the
64                  // contribution goal has been met.
65                  System.out.println("Thank you");
66              }
67          }
68      }
69  }
```

In this case, there is one thread representing the minister waiting for the collection to exceed $100. There are a further six threads representing attendees adding $20 to the collection plate. Each of these threads synchronizes on the `collectionPlate` object. Since the threads adding to the collection plate need to obtain the lock on the `collectionPlate` object, the minister thread must temporarily release it from time to time, which it does by calling the `wait` method. As each thread adds to the collection plate, it wakes the minister by calling the `notify` method.

■ **Note** In Listing 4-1, it does not really matter whether the collection threads call `notify` or `notifyAll` in line 41—either call will wake the minister thread. Since there is only one thread waiting on the condition (the amount in the collection plate) to change, we have chosen to call `notify`. If multiple threads were waiting on different conditions to change, then it would make more sense to call `notifyAll`.

Or imagine a producer/consumer relationship where one thread populates an object and another consumes that object when it is populated. In this case, the consumer thread wants the object to be modified; thus, the consumer wants to know when that modification occurs. `Object.wait`, `Object.notify`, and `Object.notifyAll` support the latter kind of waiting.

Calling the `wait` method is different than calling `sleep` or `yield`, or even pausing for I/O. The `wait` method actually allows other threads to acquire the lock on the object in question, while `sleep`, `yield`, and pausing for I/O do not.

■ **Note** The `wait` and `notify`/`notifyAll` methods almost always go together. If one thread `wait`s, there should be another thread that can call `notify` or `notifyAll`. The exception to this would be if you were doing something with timeouts and needed to release locks—but this would be a very rare situation. If there's no call to a `wait` method, then there's no reason at all to call `notify` or `notifyAll`.

Yielding

Yielding is the act of politely offering up your turn in the queue while maintaining your resources. For example, suppose you're trying to use a bank's automated teller machine (ATM). You know that this process will take a while because you're going to transfer funds, check your balance, make a deposit, and so forth. Yielding would be the act of offering the person behind you in line an opportunity to use the ATM before you. Of course, you wouldn't give that person your resources (money, ATM card, and so on). However, you would offer to give up your spot at the ATM (analogous to the CPU). The process of making this offer, even if there's no one to accept it, is yielding.

It is possible that, even though you yielded, you may still get first access. In our ATM scenario it is possible that the person behind you is too nice to accept your kindness. In computer terms, though, when you yield, all threads (including your own) then get to compete for the lock on the object—your thread does not get a lesser or greater chance of getting the lock on the object. But what decides which thread actually wins that competition and gets access to the lock on the object? The thread scheduler does.

The *thread scheduler* is like an office manager. It takes threads into consideration and decides which thread should be allowed to run based on the underlying operating system, thread priorities, and other factors.

There are two important points to note. First, you must be aware that your thread of execution is not guaranteed to be next in line when you yield, even though you originally volunteered to give up the CPU. Once a thread yields and the CPU is taken by another thread, the original yielding thread does not have any special status in being next in line. It must wait and hope, like every other thread.

Second, it's important to realize that your thread didn't give up resources that you have exclusive locks on. That is, even though you're letting someone else use the ATM, you didn't give him your ATM card. When a thread yields, it steps out of the CPU, but it retains control over any resources it had originally marked for exclusive use.

So why yield? Imagine that your thread entails six small bank operations: checking your balance, moving money from your checking account to your everyday account, checking your balance, moving money from your savings account to your everyday account, checking your balance, and finally withdrawing money. The thread you're yielding to might

just put the money it owes you into your everyday account, making your financial footwork unnecessary and thus removing your need to continue. If your thread had not yielded, it might not have received the money until after it was done with all six bank operations. If your thread is performing a prolonged or expensive operation, it's a good idea to yield occasionally.

■ **Caution** You should not rely on thread priorities, and by extension yielding, to resolve your threading issues. Implementation of thread scheduling, including yielding, is completely up to the JVM manufacturer. So it is possible that different operating systems, or even two different JVMs on one operating system, may deal with yielding differently. So go ahead and yield in your programs, but never count on the yielding to actually occur.

Blocking

Blocking is the act of pausing execution until a lock becomes available. If your thread is attempting to obtain a lock some other thread already owns, then it will block until the other thread releases its lock. This simply means that the thread goes into a state of hibernation until the event comes to pass. In this case, the lock becoming available is the event.

What does this mean for other threads? It means that they are free to use the CPU cycles that the blocking thread has a right to but is not using. However, the blocking thread keeps exclusive control over any other resources it had explicitly locked. Also, other threads should be prepared for the blocking thread to start again at any point, because the event that will trigger it could happen at any time.

■ **Note** The JVM handles changing threads from the `Thread.State.BLOCKED` state to the `Thread.State.RUNNABLE` state for you. You do not have to do anything special in your code to tell any other threads that you are about to release a lock on an object.

Sleeping

Sleeping is the act of waiting for at least a specified amount of time. Imagine that you are using the ATM, but get a message on screen telling you that the system is congested. You might decide that you want to give up for 5 minutes and then wander over to try to use it again after that time has passed. This act frees up the ATM for at least the next 5 minutes. After that time period expires, you walk over to the ATM and join the line waiting for its use.

■ **Note** A sleeping thread is guaranteed to wait at least as long as specified. However, it might wait longer, depending on the whims of the thread scheduler.

The difference between sleeping and yielding is that a yielding thread never knows how brief or long the wait will be. A sleeping thread, on the other hand, always knows that it will wait for at least the amount of time specified.

Child.java Example

This example is probably more complicated than anything you'll have to do on the SCJD exam, but it provides a good illustration of waiting. Study it carefully and you won't be overwhelmed by the threading demands of the SCJD exam.

The first 41 lines in this example are very straightforward. An `IceCreamMan` object is created in its own thread and goes into a loop, waiting for clients to hand him `IceCreamDish` objects. The `IceCreamMan` is a static object, which ensures that there will only ever be one `IceCreamMan` for all the children in our example.

In line 15 we set the `IceCreamMan` thread to be a daemon thread. The JVM will exit if the only threads still running are daemon threads. In this example, we have explicitly created several non-daemon threads: the three threads for each of the children. There is also one other non-daemon thread that was created for us: the main thread. When these four threads have completed, the only thread we *explicitly* created that is still running will be the `IceCreamMan` daemon thread, so the program will terminate.

■ **Note** You cannot change a thread's daemon status after the thread has started—if you want your thread to be a daemon, you must explicitly set it as such before starting the thread. A thread created from non-daemon threads defaults to being a non-daemon thread. Likewise, a thread created from daemon threads defaults to being a daemon thread. In either case, if you do not like the default type of thread, you can change it by calling the `setDaemon` method on the thread.

The `IceCreamMan` class extends the `Thread` class, so we can call the start method directly, as shown in line 16.

Line 25 shows an example of using the enhanced for loop to iterate over the items in an array. In this array we create several `Child` objects, each of which is an independent thread. Their role is to request a dish of ice cream from the `IceCreamMan`.

Lines 27 and 28 show how to create a thread based on a `Runnable` class. In line 27 we create a new thread from the `Runnable` class, and in line 28 we start the thread as we would any other thread object.

For this example we have decided that getting ice cream for our three children is the most important task—what happens after that is not important. So we have decided that we can end the application once all three children have eaten their ice cream. To do this, our main thread must pause until all three `Child` threads have completed running. We accomplish this in lines 32 through 38, where we call the `join` method on each of the child threads.

■ **Tip** If you are having trouble understanding the terminology of "joining a thread," it may make more sense if you refer back to Figure 4-2. In the diagram it *appears* as if each thread split from the main thread, then rejoined it after completion.

Once all the `Child` threads have completed, the main thread prints a status message, then exits. As mentioned earlier, at this point there will be no more non-daemon threads running, and the application itself will exit.

■ **Note** If we did not have the join statements in lines 32 through 38, the application would still work in a very similar fashion. The major difference would be that the main method would complete before the `Child` threads complete. However, since the `Child` threads are not daemon threads, they would still continue to run. So all the children would still get their ice cream. Once all the children have their ice cream, the program would still exit.

```
1  /**
2   * a Child object, designed to consume ice cream
3   */
4  public class Child implements Runnable {
5      private static IceCreamMan iceCreamMan = new IceCreamMan();;
6      private IceCreamDish myDish = new IceCreamDish();
7      private String name;
8
9      public Child(String name) {
10         this.name = name;
11     }
12
13     public static void main(String args[]) {
14         // start the ice cream man's thread.
15         iceCreamMan.setDaemon(true);
16         iceCreamMan.start();
17
18         String[] names = {"Ricardo", "Sally", "Maria"};
19         Thread[] children = new Thread[names.length];
20
21         // create some child objects
22         // create a thread for each child
23         // get the Child threads started
24         int counter = -1;
25         for (String name : names) {
26             Child child = new Child(name);
27             children[++counter] = new Thread(child);
28             children[counter].start();
29         }
30
31         // wait until all children have eaten their ice cream
32         for (Thread child : children) {
33             try {
34                 child.join();
```

```
35              } catch (InterruptedException ie) {
36                  ie.printStackTrace();
37              }
38          }
39
40          System.out.println("All children received ice cream");
41      }
```

`Child` objects attempt to hand their personal `IceCreamDish` to the ice cream man, as shown in line 44. Then they eat the ice cream when it is returned to them—this is handled by the `eatIceCream` method.

■ **Note** The Sun Code Conventions for the Java programming language recommend inserting a blank line between methods; however, there is no value in showing these blank lines when code listings have been split to allow room for comments within this book. Nevertheless, the blank lines do exist in the downloadable source, so to ensure the line numbers remain consistent between the book and the downloadable source, you may see cases where blank lines and Javadoc comments have been removed from the text. Therefore, while it appears that lines are missing, rest assured that all the relevant code is presented.

```
43              public void run() {
44                  iceCreamMan.requestIceCream(myDish);
45                  eatIceCream();
46              }
```

As we will see when we review the `IceCreamMan` code in the next section of this chapter, after receiving an `IceCreamDish` the `IceCreamMan` instance fills it and signals that he is done with that particular `IceCreamDish`. The `Child` instances wait until the `IceCreamMan` modifies their personal `IceCreamDish` objects and notifies them of the change. As soon as they receive that notification, they eat their `IceCream`.

Notice that in this example the `Child` instance actually releases its lock on the instance of the `IceCreamDish` by calling `myDish.wait()` in line 60. This then provides the opportunity for the `IceCreamMan` to gain the lock on the dish, after which he can fill it.

The next interesting part of the code happens between lines 48 and 69:

■ **Note** You cannot call the `wait` method on an object unless you are in a block or a method that is synchronized on that object.

```
48      public void eatIceCream() {
49          String msg = name + " waiting for the IceCreamMan to fill dish";
50          /*
51           * The IceCreamMan will notify us when the dish is full, so we should
52           * wait until we have received that notification. Otherwise we could
```

```
53           * get a dish that is only half full (or even empty).
54           */
55          synchronized (myDish) {
56              while (myDish.readyToEat == false) {
57                  // wait for the ice cream man's attention
58                  try {
59                      System.out.println(name + msg);
60                      myDish.wait();
61                  }  catch (InterruptedException ie) {
62                      ie.printStackTrace();
63                  }
64              }
65              myDish.readyToEat = false;
66          }
67          System.out.println(name +": yum");
68      }
69  }
70
71  class IceCreamDish {
72      public boolean readyToEat = false;
73  }
```

Line 55 synchronizes on the `IceCreamDish` reference, `myDish`. This means that the current `Child` thread will not move past line 55 until it has exclusive access to the `IceCreamDish` reference of this particular `Child` object.

Remember that each `Child` and the `IceCreamMan` share access to that particular child's `IceCreamDish`. The `IceCreamMan` could be modifying the `IceCreamDish` at any time; we don't want the `Child` object to use the `IceCreamDish` until the `IceCreamMan` is through.

Now assume that the `Child` has achieved access to the `IceCreamDish`. This could happen for two reasons. First, the `IceCreamMan` is not currently using that `Dish` (he does, after all, have other `Child` objects to attend to). Second, the `IceCreamMan` is finished with that `Dish`. Line 56 checks the value of `dish.readyToEat`. This value tells the `Child` whether the `IceCream` is ready to be eaten. A monitor is used between the `Child` objects and the `IceCreamMan`. Monitors will be defined shortly. For now, think of a monitor as a prearranged signaling device between the `IceCreamMan` and the `Child` objects.

If the `dish.readyToEat` value is false, then the `IceCreamMan` has not finished filling the bowl. The `Child` can release the lock on the `Dish` by calling `wait` on it, as in line 60.

Notice that the `Child` thread checks the condition of the `dish.readyToEat` variable in a while loop, not an if statement. The reason for this is due to a small problem with when the JVM will return control to the application after a call to the `wait` method. The Sun Javadoc comments for the `wait` method say, in part: "*A thread can also wake up without being notified, interrupted, or timing out, a so-called spurious wakeup. While this will rarely occur in practice, applications must guard against it by testing for the condition that should have caused the thread to be awakened, and continuing to wait if the condition is not satisfied. In other words, waits should always occur in loops.*"

IceCreamMan.java Example

The IceCreamMan object starts a loop inside his run method. The method checks to see if any IceCreamDish objects need to be serviced. This all happens in the run method between lines 15 and 32:

```
 1  import java.util.*;
 2
 3  public class IceCreamMan extends Thread {
 4      /**
 5       * a list to hold the IceCreamDish objects
 6       */
 7      private List<IceCreamDish> dishes = new ArrayList<IceCreamDish> ();
 8
 9      /**
10       * Start a thread that waits for ice cream bowls to be given to it.
11       */
12      String clientExists = "IceCreamMan: has a client";
13      String clientDoesntExist = "IceCreamMan: does not have a client";
14
15      public void run() {
16          while (true) {
17              if (!dishes.isEmpty()) {
18                  System.out.println(clientExists);
19                  serveIceCream();
20              } else {
21                  try {
22                      System.out.println(clientDoesntExist);
23                      // sleep, so that children have a chance to add their
24                      // dishes. see note in book about why this is not a
25                      // yield statement.
26                      sleep(1000);
27                  }  catch(InterruptedException ie) {
28                      ie.printStackTrace();
29                  }
30              }
31          }
32       }
```

Line 16 starts an infinite loop for this thread. Remember that the thread for the IceCreamMan is a daemon thread, so even though this is an infinite loop it will not stop the application from exiting once all Child threads have eaten their ice cream.

Line 17 checks to see if any IceCreamDish objects have been queued up for processing. If not, the IceCreamMan thread sleeps for a second and then checks again, per lines 22 through 26. If the queue has an entry, then the IceCreamMan thread calls the serveIceCream method.

At line 26 we had a choice—we could have yielded to other threads, or we could sleep for some time. Either choice would have provided the Child threads a chance to run. However, sleeping provides a better chance for the Child threads to run, as the JVM knows that the IceCreamMan will not be running for at least a second—if we had yielded, any of the Child

threads or the IceCreamMan thread could have started running immediately afterwards. In addition, yielding is something you would normally do before you start (or in the middle of) a lengthy or complex task—it is a way of being nice to other threads. However, this is not the case here. Finally, if there are no other threads that can run, the JVM will immediately pass control back to the while loop—this will result in the thread consuming as much CPU cycles as are available; it can even lead to your computer having the appearance of hanging.

The serveIceCream method is the most interesting part of our code. Assuming that there is currently a Dish in the queue, line 49 synchronizes on that IceCreamDish instance. This has the effect of forcing any Child objects that want to use the IceCreamDish instance to wait. Specifically, it causes line 55 in Child to pause until line 53 in IceCreamMan is reached or, if the Child thread had paused at line 60, it will not be able to resume until line 53 in IceCreamMan is reached.

Conversely, if a Child instance is executing a method that synchronizes on the IceCreamDish, then hitting line 49 in IceCreamMan will cause IceCreamMan to wait until lines 50 through 61 in Child finish executing, or until the Child instance releases the lock on the IceCreamDish by calling wait in line 55.

This is an example of the threads respecting each other's boundaries. By synchronizing on the same object, the two threads can be assured that they will not be simultaneously modifying or using the same object.

When the IceCreamMan owns the lock on the dish, he can fill it with ice cream, and then notify the Child that it is available. This happens in line 52. After line 52 is run, the Child thread that was waiting on that particular plate will no longer be waiting—the JVM scheduler will now be able to run it. However, until the IceCreamMan releases the lock on the dish at the completion of line 53, the Child will not be able to regain the lock on the dish, so it will remain blocked.

■ **Note** You cannot call the notify method on an object unless you are in a block or a method that is synchronized on that object.

```
34      /**
35       * Serve Ice Cream to a Child object.
36       */
37      private void serveIceCream() {
38          // get an ice cream dish
39          IceCreamDish currentDish = dishes.get(0);
40
41          // wait sometimes, don't wait sometimes
42          if (Math.random() > .5) {
43              delay();
44          }
45
46          String msg = "notify client that the ice cream is ready";
47          System.out.println("IceCreamMan: " + msg);
48
```

```
49          synchronized (currentDish) {
50              currentDish.readyToEat = true;
51              //notify the dish's owner that the dish is ready
52              currentDish.notify();
53          }
54
55          //remove the dish from the queue of dishes that need service
56          dishes.remove(currentDish);
57      }
```

The synchronized blocks in lines 49–53 in the `IceCreamMan`'s `serveIceCream` method here, and lines 55–66 of the `Child`'s `eatIceCream` method, require both threads to synchronize on the same object. We have to assume that any programmers working on the `Child` class and the `IceCreamMan` class will continue to use the synchronized blocks. However, there is a risk that a `Child` may remove their synchronized code, which would mean that they could grab their dish back before it has been filled with ice cream. This actually gives us a safeguard against the program being changed—we can explain to other programmers that removing the synchronized code could result in the `Child` thread getting a dish that has not been completely filled (or could even be empty).

■ **Tip** Whenever code is written that depends on a particular way of implementing it, you should add an implementation comment to explain the details to other programmers. We have shown this in the comments in lines 50–54 of the `Child` class in the previous code listing.

While it is relatively easy to explain to a `Child` that they should not try to take the plate before they have been told it is full (because they won't want to miss out on more ice cream), it is harder to get them to agree not to try to hand their dishes over all at once—from their perspective the sooner they hand the dish over the sooner it will be filled. They don't particularly care about fairness to the other children, or how well the `IceCreamMan` can handle receiving multiple plates at once.

To guard against this, we have synchronized the `IceCreamMan`'s `requestIceCream` method. By synchronizing the method, we can ensure that only one `Child` is ever handing a dish to the `IceCreamMan` at a time.

```
59      /**
60       * Allow client objects to add dishes
61       */
62      public synchronized void requestIceCream(IceCreamDish dish) {
63          dishes.add(dish);
64      }
65
```

```
66      /**
67       * build in a delay
68       */
69      private void delay() {
70          try {
71              System.out.println("IceCreamMan: delayed");
72              Thread.sleep((long) (Math.random()*1000) );
73          } catch (InterruptedException ie) {
74              ie.printStackTrace();
75          }
76      }
77  }
```

As Figure 4-3 shows, the `IceCreamMan` is waiting for clients in his own thread. Each `Child` object is also in its own thread. The `Child` objects interact with the `IceCreamMan` by giving him their `IceCreamDish`, and then they step out of the CPU and `wait`. The `IceCreamMan` prepares the `IceCreamDish`, which is the signal that the `Child` associated with that `IceCreamDish` needs to wake up and eat their `IceCream`.

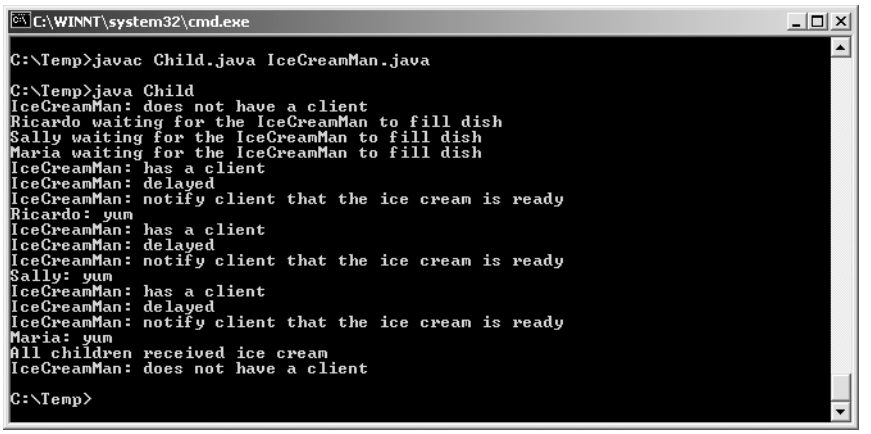


Figure 4-3. *IceCreamMan waiting for clients*

Waiting Summary

A thread that needs a resource becomes paused without you specifically requesting that it pause. In contrast, a thread that pauses because of a call to `yield`, `wait`, or `sleep` is paused specifically at the request of the programmer. Regardless, all cases result in the thread pausing execution.

In the case of a thread paused until a resource becomes available and a thread that yielded to other threads, the programmer does not know when the thread will resume processing—for the former, it will be whenever the resource becomes available, and for the latter it will be at the discretion of the JVM thread scheduler. If you call `wait` without any parameters, you will not know when the thread will resume processing—it is dependent on another thread notifying it that it can continue operating. However, if you call `wait` with a time limit,

you will know that the thread will resume processing when it has been notified, or soon after the time limit has expired. Remember that time limits cannot be strictly enforced; the JVM will pause the thread at least as long as you requested and make its best attempt to schedule the thread after the time limit has expired, but the exact time the thread resumes is not guaranteed. If you call `sleep`, you will know that the thread will resume processing soon after the time limit has expired—again, time limits cannot be strictly enforced. Returning to the ice cream example, imagine that little Sally has finally gotten the ice cream man's attention and given him some money, but she hasn't yet actually picked her flavor. If Sally yielded, then she would move out of the way voluntarily to let another child try, but she would be secure in the knowledge that the ice cream man won't sell the cone that she explicitly locked down. If no children were in fact interested, then she would jump back in and have her cone filled with ice cream.

If Sally decided to `sleep`, it might be as if she has put her money down for a cone but is delaying making her choice for up to 1 minute—say, for her younger brother to arrive. After the minute has elapsed, then she will try again (whether the younger brother has shown up or not). When she finally does get the ice cream man's attention, she will be confident that the ice cream cone she had locked down will still be there. But just because Sally has decided that the minute is up does not mean that she will be able to get her cone immediately—another child may be getting his or her cone filled at the end of Sally's minute. Sally is blocked whenever she cannot get the ice cream man's attention. Having slept for 1 minute in the previous example, she now finds that the ice cream man is serving another child. Although she still owns the lock on her cone, she does not own the lock on the ice cream man and is blocked until she can get that lock (and maybe her brother can turn up in the meantime).

However, if Sally had to wait for her father to bring her money, then she does not have any cone locked down, and by the time the money arrives, all the cones may be sold.

Locks

Locks are tokens of exclusive use. Each Java object has one. Think of locking as the ability to stick a note on any object and claim that object for your thread's exclusive use. By claiming an object's lock, your thread tells other methods (those that respect synchronization) not to modify that object in any way until your thread releases it. A method that respects synchronization is one that synchronizes on the object in question.

In Java, you can lock an object by synchronizing on it. Consider the following example:

```
public void addElement(Object item) {
      synchronized (myArrayList) {
      //do stuff
     }
}
```

This lets the JVM know that other threads should not modify the `myArrayList` object. Of course, this only applies to synchronized methods. For instance, in the following example the method `elementExists` might have no need to synchronize access, because it is a read-only operation:

```
public boolean elementExists(Object item) {

    return myArrayList.contains(item);
}
```

However, if it were required that `elementExists` be absolutely accurate (that is, if you could not afford to receive an incorrect answer while another thread was in the middle of inserting or removing elements into the `ArrayList` by another thread), then you would synchronize access here as well.

This section integrates some of the ideas already presented. The concept of locking is fundamental to understanding threading, and you will gain a lot by paying careful attention here.

Locking Objects

You lock objects all the time in everyday life. When you go the movies, you might "lock" your seat by leaving your coat on it. Even if you leave the seat to buy some popcorn, it is understood that no one should sit down in that seat. The act of leaving your coat on the seat effectively marks the seat for your exclusive use until you choose to release it.

Any Java object can be claimed for exclusive use. This is one of the explicit ways that Java supports multithreading. Java further supports multithreading by allowing objects to broadcast when they are freed and by forcing threads that want a locked resource to become inactive until those resources are freed.

If an object is not explicitly claimed, it is open for any thread's use. Similarly, a seat in a movie theater that is not explicitly claimed can be taken at any time. This means that if you leave your seat to buy some refreshments and don't lock the seat by leaving your coat on it, then you shouldn't be surprised if someone else has claimed your seat when you get back. Even if this strategy has worked in the past, it is not thread-safe for future use. The same is true in Java. Unless your classes are made explicitly thread-safe, the fact that they have been uncorrupted in the past is no guarantee that they will not be corrupted in the future.

There is a second level of depth to this metaphor. Someone might take your movie theater seat anyway, even though your coat is on it. The same is true in Java. A locked object can be violated if the thread that is modifying it does not respect synchronization.

For example, suppose you have a method that synchronizes on a member variable:

```
public void goodMethod() {
    synchronized (myObject) {
      //do stuff to myObject
    }
}
```

A second method could modify `myObject` if it chooses not to synchronize on `myObject`. Synchronizing on an object means "I'll respect other people's locks on this object, and I hope they respect mine." However, it is unenforced. Thus, the following example would refuse to respect the synchronization established in `goodMethod`, and would execute without complaint. This could cause problems if `badMethod` modifies `myObject`, as other threads that have synchronized on `myObject` would be expecting to have exclusive access to `myObject`.

```
public void badMethod() {
    //do stuff to myObject
}
```

Of course, these methods' names don't really imply that one way is "good" and another is "bad." For example, `badMethod` might just need to read `myObject`. If so, depending on context, it may be perfectly okay not to synchronize on `myObject`. As is often the case, "good" and "bad"

Locking Class Instances

Java provides a mechanism for locking classes as well as objects. This can be a little confusing, but the concept is actually very straightforward. A class is just an object, used to create other objects, just as an axe is an object for creating firewood.

The JVM generates a `Class` object when your program initially loads for every class the program uses. Per JVM, there is a single `Class` object for all of the instance objects of a given class. You have limited access to the `Class` object. Because the `Class` object is itself an object, you can lock it just as you can lock any other object. What makes the `Class` object interesting is its capability to contain static variables and static methods.

Static variables are universal for every object of a class, which means that a single instance is shared by every object of that class. For example, for objects of the class `McBurgerPlace`, `chiefExecutiveOfficer` is a static member variable, because it is common to all. No matter which `McBurgerPlace` you are eating in, it shares the same `chiefExecutiveOfficer` with every other `McBurgerPlace`. If any `McBurgerPlace` restaurants were to change the `chiefExecutive➥Officer`, every other `McBurgerPlace` would instantly be able to sense that change.

A static method is universal for every object of a class, which means that the single method is shared by every object of the class, per JVM. For example, objects of the class `McBurgerPlace` might have `createFranchise` as a static member method, assuming that creating a franchise happens at the corporate level. Any `McBurgerPlace` object can create a new franchise by calling corporate headquarters and asking to do so. No matter which `McBurgerPlace` store you are talking about, the exact same `createFranchise` method is called.

Using static methods is different from using instance methods in one very important way: The method exists on the class, not the objects. The proof of this lies in the fact that you can create a franchise, even if there is no `McBurgerPlace` store present, by getting in touch with corporate headquarters directly.

So what does all of this have to do with locking? Assume that you want to lock a given `McBurgerPlace` for exclusive use while executing a method. For example, if your `McBurgerPlace` (say the one on High Street) has a `waxFloor` method, then you don't want to be doing anything else while waxing the floor—after all, lawsuits abound.

How do you enforce this exclusivity? In the real world, you make sure that nothing else is happening in that particular store while waxing is in progress: no making burgers, no selling French fries, no preparing Joyful Meals. Almost every activity waits until you are done waxing the floor. To achieve the same exclusivity in Java, you synchronize the method in question by putting the keyword `synchronized` in the method declaration:

```
public synchronized void waxFloor()
```

This means that any other synchronized methods called on the `McBurgerPlace` on High Street will wait until the floor has been waxed.

■ **Note** Unsynchronized methods do not respect synchronized methods. Thus, an unsynchronized method can be executed at any time. It is important to make sure that such methods can do no damage in your object's state. For example, `countMoney` is probably safe to leave unsynchronized because it cannot interfere with the waxing of the floor. The method `wipeDownTables` should probably be synchronized, however, because it does interfere with the floor waxing.

This exclusivity has nothing to do with other objects of the McBurgerPlace class. For example, the McBurgerPlace on Fifth Street is free to make burgers, sell French fries, and so forth, even if the McBurgerPlace on High Street is currently waxing the floor. Why shouldn't they?

What about dealing with an activity that does affect other objects? For example, what about receiving the chiefExecutiveOfficer? Obviously, you only want a single McBurgerPlace to receive the chiefExecutiveOfficer at a given time: She can't be in two places at once. How do you represent this exclusivity of access in Java? For that matter, how do you control access to her? You don't want one store to interrupt her while she is in the middle of talking to another store.

There are two steps in the solution to this dilemma. First, make sure that the chiefExecutiveOfficer variable is static—that is, it exists only at the class level. You can achieve this by using the keyword static when declaring the chiefExecutiveOfficer variable at the class level:

```
private static Object ceo = new Object();
```

The second step is to synchronize the static method that accesses the chiefExecutive➥Officer, like this:

```
public static synchronized Object receiveCeo (){
    return ceo;
}
```

Because the method is static, this synchronization occurs at class level rather than at instance level.

Locking Objects Directly

There is a second way to lock objects (both instance objects and class objects). You can synchronize on the object you want to lock explicitly. For example, imagine that your McBurgerPlace class has a getSoda method that requires exclusive control of the sodaFountain member variable. You can lock the entire McBurgerPlace object, or you can lock just the sodaFountain member variable:

```
public void getSodaEfficiently() {
    synchronized (sodaFountain) {
        //do Stuff
    }
}
```

Locking a member variable object is like forcing only those people who want to use the sodaFountain to wait while you are using the sodaFountain, as opposed to forcing everyone in the entire store (including those who only want to purchase a burger) to wait.

Synchronizing a method is a form of locking an object—specifically, the this object. Synchronizing a method is really just shorthand for synchronizing on the this reference object. Thus, this code presented in Listing 4-2 is equivalent to the code presented in Listing 4-3.

Listing 4-2. *Synchronizing a Method*

```
public synchronized void myMethod() {
    //code
}
```

Listing 4-3. *Synchronizing on the this Reference Object*

```
public void myMethod() {
    synchronized (this) {
        //code
    }
}
```

Locking some object other than `this` can provide more concurrency than synchronizing a method because it allows other blocks of code that are synchronized on different objects to work concurrently. But efficiency, alas, is in the eye of the beholder. If an individual method needs to obtain multiple locks, then the steps of locking and unlocking the multiple locks can lead to more overhead than synchronizing the method.

In Listing 4-4, `synchThisObjectExample` is more efficient than `synchAllLocksExample` because it doesn't have the overhead of locking and unlocking three times. If your object is carefully constructed not to allow unsynchronized access to the resources you need to lock, then synchronizing methods is probably the simpler approach.

Listing 4-4. *Synchronization Example*

```
 1  public class SynchExample {
 2
 3      private Resource resourceOne = new Resource();
 4      private Resource resourceTwo = new Resource();
 5      private Resource resourceThree= new Resource();
 6
 7      public synchronized void synchThisObjectExample() {
 8          resourceOne.value = -3;
 9          resourceTwo.value = -2;
10          resourceThree.value = -1;
11      }
12
13      public void synchAllLocksExample() {
14          synchronized (resourceOne) {
15              resourceOne.value = -3;
16          }
17
18          synchronized (resourceTwo) {
19              resourceTwo.value = -2;
20          }
21
```

```
22          synchronized (resourceThree) {
23              resourceThree.value = -1;
24          }
25      }
26
27      private static class Resource {
28          int value;
29      }
30  }
```

The notify and notifyAll Methods

Every Java object has the capability to broadcast a call, notifying threads that have called the `wait` method on that object that some event that they might be interested in has occurred. If you call the `notify` method, one of the threads waiting on that object is told to come out of the waiting state and block until it can regain the lock on the object, after which it can continue processing. If you call `notifyAll`, every thread waiting for the object is told to come out of the waiting state and block until it can regain the lock on the object, after which it can continue processing.

■ **Caution** It is easy to become confused when talking about threads waiting. `Thread.State.WAITING` has a specific meaning in Java, namely that the thread has called the `wait` or the `join` methods. When a thread enters the `Thread.State.WAITING` state by calling the `wait` method, it consumes no CPU cycles until some other thread calls either `notify` or `notifyAll` on the same object that `wait` was called on, or until the thread is interrupted. When a thread enters the `Thread.State.WAITING` state by calling the `join` method, it consumes no CPU cycles until the thread it is trying to join terminates, or until the thread is interrupted. In contrast, if several threads attempt to obtain the same lock simultaneously, one will obtain it, and the remainder will enter the `Thread.State.Blocked` state until the lock is released, at which time another thread will obtain the lock—the JVM scheduler handles automatically selecting another thread from those in the `Thread.State.Blocked` state.

Listing 4-5 demonstrates the difference between calling `notify` and `notifyAll`.

Listing 4-5. *NotifyVersusNotifyAll.java*

```
1  public class NotifyVersusNotifyAll extends Thread {
2      private static Object mutex = new Object();
3
4      public static void main(String[] args) throws InterruptedException {
5          for (int i = 0; i < 5; i++) {
6              new NotifyVersusNotifyAll().start();
7          }
8
9          Thread.sleep(2000);
```

```
10          synchronized(mutex) {
11              mutex.notifyAll();
12  //            mutex.notify();
13          }
14      }
15
16      public void run() {
17          try {
18              synchronized (mutex) {
19                  System.out.println(getName() + " waiting");
20                  mutex.wait();
21                  System.out.println(getName() + " woken up");
22                  mutex.wait(2000);
23                  System.out.println(getName() + " waking up another thread");
24                  mutex.notify();
25              }
26          } catch (InterruptedException ie) {
27              ie.printStackTrace();
28          }
29      }
30  }
```

Figure 4-4 shows the results of running `NotifyVersusNotifyAll`, first with line 12 and line 11 commented out (so that `notify` is called), then with line 11 and line 12 commented out (so that `notifyAll` is called).

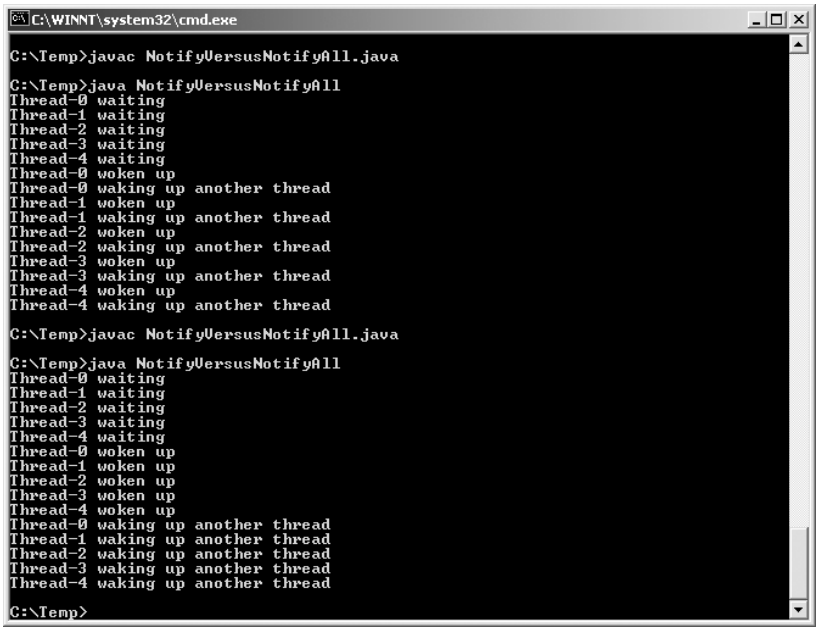


Figure 4-4. *Notify returns one thread to the runnable state whereas notifyAll returns all threads to*

■ **Caution** Although Figure 4-4 shows the threads being notified in the same order they called `wait`, this should never be relied on. The order in which threads are notified, and the order in which they regain ownership of the lock on the common object, is entirely at the discretion of the JVM's thread scheduler.

In general, you would use `notify` only when you are certain that the thread that will be notified will be able to use the notification. If you have multiple threads that you are certain will be able to use the notification but you want only one of them to transition to the runnable state, then you should use `notify`. However, in this case you must ensure that the remaining threads do not stay in limbo—some thread will have to notify them at an appropriate time that they can continue processing. This is complex and requires detailed analysis in order to get it right. If you have multiple threads waiting on one event but they have different conditions to meet, you are better off calling `notifyAll` so that all the threads will recheck their condition. As an example, consider the following code snippets:

Producer Thread:

```
synchronized (lock) {
    value = Math.random();
    lock.notifyAll();
}
```

Consumer Thread 1:

```
synchronized  (lock) {
    while (value < 0.5) {
        lock.wait();
    }
}
```

Consumer Thread 2:

```
synchronized (lock) {
    while (value >= 0.5) {
        lock.wait();
    }
}
```

This code shows one example where calling `notifyAll` is preferable to calling `notify`. By calling `notifyAll` after setting `value`, the producer can be sure that both consumers will check the current contents of the `value` variable, and one of them will be able to continue processing. If `notify` had been used instead of `notifyAll`, it is possible that after setting `value`, the wrong thread may have been notified, and upon finding that the `value` variable did not contain the desired number, would have gone back to the `Thread.State.WAITING` state, meaning that neither consumer thread would be working!

Dealing with Nonimplicit Locking

You should be aware of two important issues when you deal with locks. First, locking an object does not lock member variables of that object. This may seem counterintuitive at first, but it makes sense from the JVM's point of view. Specifically, it allows the thread to avoid locking objects that may be nested *n* layers deep. Imagine the overhead of locking an object, locking all of its member variables, locking all of their member variables, and so on. Listing 4-6 shows an example of locking objects but not their member variables. The output is shown in Figure 4-5.

Listing 4-6. *A Locking Objects Example*

```
1  import java.util.*;
2
3  public class LockObjectNotMemberVariables{
4      private List myList = new ArrayList();
5
6      public static void main(String args[]){
7          LockObjectNotMemberVariables lonmv =
8                  new LockObjectNotMemberVariables();
9          lonmv.lockTest();
10     }
11
12     public synchronized void lockTest(){
13         System.out.println("Is the THIS object locked? " +
14                            Thread.holdsLock(this));
15
16         System.out.println("Is the list object locked? " +
17                            Thread.holdsLock(myList));
18     }
19 }
```

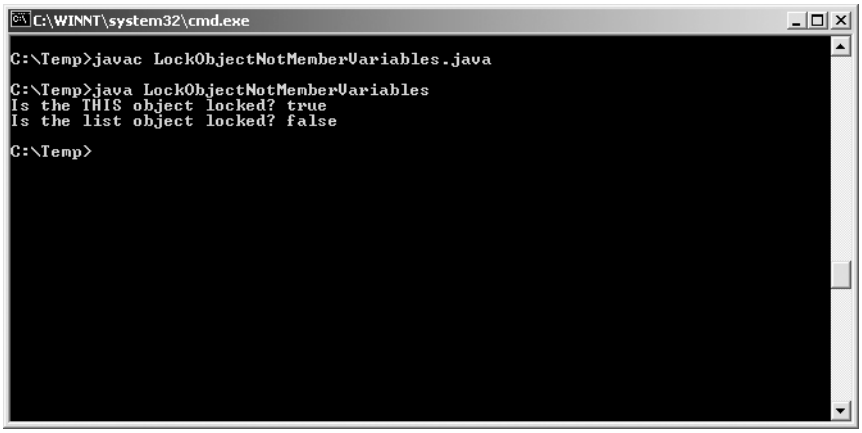


Figure 4-5. *Locking an object does not lock member variables of that object.*

The second important point is that locking a class does not lock instance variables of that class. This is demonstrated in Listing 4-7. Notice that the method `lockTest` is both static and synchronized. This means that it achieves a lock on the `Class` object. Figure 4-6 shows that locking a class does not lock objects of that class.

Listing 4-7. *A Locking Example*

```
1  public class ClassLockNotObjectLock {
2      public static void main(String args[]) {
3          lockTest();
4      }
5
6      public static synchronized void lockTest() {
7          ClassLockNotObjectLock clnoc = new ClassLockNotObjectLock();
8          System.out.println("Is the class object locked? " +
9                             Thread. holdsLock(clnoc.getClass()));
10
11         System.out.println("Is the object instance locked? " +
12                            Thread. holdsLock(clnoc));
13     }
14 }
```

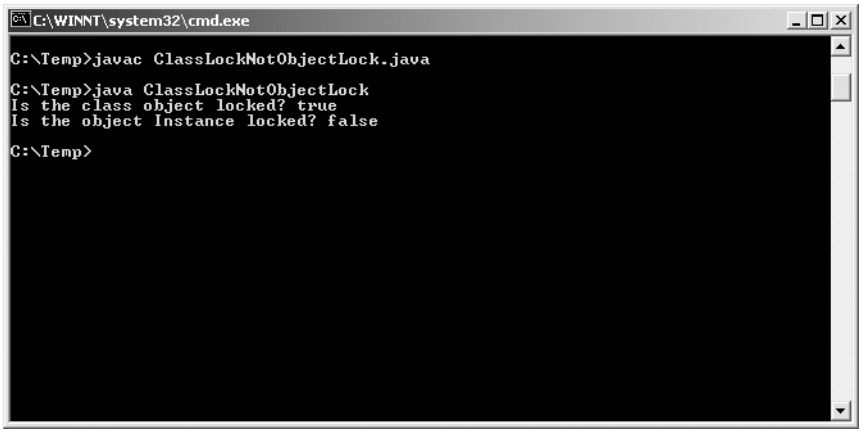


Figure 4-6. *Locking a class doesn't lock objects of that class.*

Locking in JDK 5

JDK 5 includes several packages that provide the ability to lock and wait on conditions separate from the synchronization and locking mechanisms described previously. These have several benefits, including the ability to specify whether locking should be granted fairly (remember that the methods described earlier make no guarantees about the order in which threads will be granted—a thread that has only just started waiting could be notified before a thread that has been waiting for a long time).

A new package, `java.util.concurrent.locks`, has been created in JDK 5, which provides some benefits to SCJD candidates, including the ability to have a `ReadWriteLock` (where multiple threads could all lock a record for reading, but only one thread could lock the record for writing—this will improve concurrency). We will examine `ReadWriteLocks` in the section discussing the `DvdFileAccess` class in Chapter 5.

In JDK 1.4, after returning from a call to `wait(milliseconds)`, it was not possible to know *directly* whether the thread had been notified, or whether the timeout had elapsed. The `Lock.tryLock(time, unit)` method returns a `boolean` indicating whether it had gained the lock or whether the time expired. An example of using the similar `Lock.await(time, unit)` method is shown in the "Creating Our Logical Reserve Methods" section in Chapter 5. And while this technique is not required for the SCJD assignment, it is now possible to set multiple conditions upon which a thread might be notified.

Although the new lock code looks different from using synchronized blocks, it is not too difficult to convert code from one style to another. For example, the following code uses a synchronized block:

```
Object lock = new Object();
public void doSomething() {
    synchronized (lock) {
        while (true) {
            try {
                lock.wait();
                // something happens here
            } catch (InterruptedException ie) {
                // handle exception
            }
        }
    }
}
```

This code can be changed to the new format like this:

```
private static Lock lock = new ReentrantLock();
private static Condition lockReleased = lock.newCondition();
public void doSomething() {
    lock.lock();
    try {
        while (true) {
            lockReleased.await();
            // something happens here
        }
    } catch (InterruptedException ie) {
        // handle exception
    } finally {
        lock.unlock();
    }
}
```

An example of using the new locking classes is shown in the "Creating Our Logical Reserve Methods" section of Chapter 5, and further thoughts are presented in the same chapter, in the "Discussion Point: Multiple Notification Objects" section.

To ensure that a lock is released, we highly recommend that you place the call to unlock in a finally block as demonstrated earlier.

Regardless of whether you use synchronized blocks or the new concurrency classes, the same rules of thread safety apply.

Locking Summary

Object locks and locks on the member variable within the object do not interact in any way. That is, synchronizing an object does not lock member variables of that object. Nor does synchronizing a member variable lock the object that owns that variable. Threads that own locks on different objects can run concurrently as long as they do not also share a lock on a common object. In addition, locking a class does not lock instances of that class. Unsynchronized access to member variables can violate thread safety.

Understanding Thread Safety

Threading presents unique logical pitfalls, which can often be reached unexpectedly and seemingly randomly. In this section, we define some of the more common pitfalls and offer advice on ways to avoid them. The following subsections explain what thread safety is and the sorts of horrors it helps to prevent.

Deadlocks

Deadlocks occur when threads are blocked forever, waiting for a condition that cannot occur. Deadlock is like a cartoon where Daffy Duck and Bugs Bunny are stranded on an island. Daffy has a can of food, and Bugs has a can opener. Daffy won't give up the food until he gets the can opener, and Bugs won't give up the can opener until he gets some food. They are deadlocked.

In Listing 4-8, you can see that `thread1` acquires a lock on `lock1` but needs `lock2`. `thread2` has acquired a lock on `lock2` and needs to acquire `lock1`. Neither thread will allow the other thread to progress, nor will it progress itself. This is deadlock. Figure 4-7 shows the output.

Listing 4-8. *Deadlock Example*

```
1  public class DeadlockExample {
2      /**
3       * Entry point to the application. Creates 2 threads that will deadlock.
4       */
5      public static void main(String args[]) {
6          DeadlockExample dle = new DeadlockExample();
7
8          Object lock1 = "Lock 1";
9          Object lock2 = "Lock 2";
10
```

```
11          Runner thread1 = new Runner(lock1, lock2);
12          Runner thread2 = new Runner(lock2, lock1);
13
14          thread1.start();
15          thread2.start();
16      }
17
18      /**
19       * Lock two objects in the order they were specified in the constructor.
20       */
21      static class Runner extends Thread {
22          private Object lockA;
23          private Object lockB;
24
25          public Runner(Object firstLockToGet, Object secondLockToGet) {
26              this.lockA = firstLockToGet;
27              this.lockB = secondLockToGet;
28          }
29
30          public void run() {
31              String name = Thread.currentThread().getName();
32              synchronized (lock1) {
33                  System.out.println(name + ": locked " + lockA);
34                  delay(name);
35                  System.out.println(name + ": trying to get " + lockB);
36                  synchronized (lock2) {
37                      System.out.println(name + ": locked " + lockB);
38                  }
39              }
40          }
41      }
42
43      /**
44       * build in a delay to allow the other thread time to lock the object
45       * the delaying thread would like to get.
46       */
47      private static void delay(String name) {
48          try {
49              System.out.println(name + ": delaying 1 second");
50              Thread.sleep(1000L);
51          } catch (InterruptedException ie) {
52              ie.printStackTrace();
53          }
54      }
55  }
```

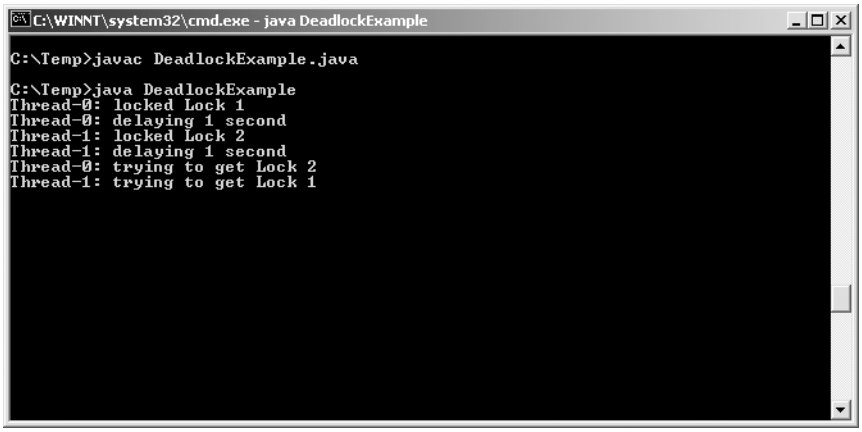


Figure 4-7. *Output from the deadlock example*

This is an example of bad design. There is almost always a better way to handle these sorts of situations than resorting to nested locks. If you find yourself unable to come up with a solution, reconsider the larger design of your project. Sun gives difficult problems on the SCJD exam, but most can be solved elegantly.

Race Conditions

Race conditions occur when two or more threads compete for the same resource, and the behavior of the program changes depending on who wins. For example, you and Johnson race to get be the first person to reach the test server in the morning. Depending on who wins, the server is tied up for a few minutes or for a few hours. Thus, based on somewhat random factors—say, who got stuck in traffic that morning—your program's behavior changes. This is rarely a good thing. Race conditions can be very difficult to track down because they happen sporadically and thus are difficult to reproduce.

Race conditions can also lead to deadlock because locks could be achieved in an order other than the one expected. Thus, your application, which had always achieved `lock1` followed by `lock2`, could suddenly achieve `lock2` first and attempt to release those locks in an unexpected order.

These sorts of problems are manifested when your application appears to "spontaneously" generate new behaviors, as shown in Listing 4-9. They are difficult to debug and often difficult to reproduce. That's why it's so important to make design decisions that will steer you clear of the general dangerous area of race conditions. Figure 4-8 shows the output of our example.

Listing 4-9. *Race Condition Example*

```
1  public class RaceConditionExample {
2      public static void main(String args[]) {
3          //create an instance of this object
4          RaceConditionExample rce = new RaceConditionExample();
5
```

```
6           //create two runners
7           Runner johnson = rce.new Runner("Johnson");
8           Runner smith = rce.new Runner("smith");
9
10          //point both runners to the same resource
11          smith.server =  "the common object";
12          johnson.server = smith.server;
13
14          //start the race, based on a random factor, one thread
15          //or the other gets to start first.
16          if (Math.random() > .5) {
17              johnson.start();
18              smith.start();
19          } else {
20              smith.start();
21              johnson.start();
22          }
23      }
24
25      /**
26      * Creates a thread, then races for the resource
27      */
28      class Runner extends Thread {
29          public Object server;
30
31          public Runner(String name) {
32              super(name);
33          }
34
35          public void run() {
36              System.out.println(getName() + ": trying for lock on " + server);
37              synchronized (server) {
38                  System.out.println(getName() + ": has lock on " + server);
39                  // wait 2 seconds: show the other thread really is blocked
40                  try {
41                      Thread.sleep(2000);
42                  } catch (InterruptedException ie) {
43                      ie.printStackTrace();
44                  }
45                  System.out.println(getName() + ": releasing lock ");
46              }
47          }
48      }
49  }
```

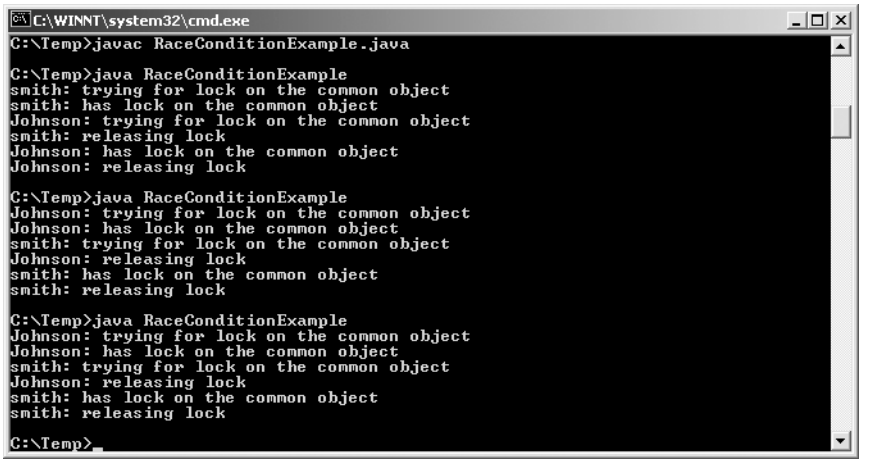


Figure 4-8. *Output from the race condition example*

It is clear here that there is no consistency in terms of who gets access to the server. This could be perfectly all right, or it could be catastrophic. It all depends on context. While it is not necessary to resolve every race condition in order to achieve thread safety, it is important to be aware of them.

Starvation

Starvation occurs when a thread never gets a chance to run. Its most common manifestation occurs when higher-priority threads keep getting preferential treatment over those with a lower priority. Imagine that your task requires that you speak to the CTO of your company. However, every time it seems as if she might be free, a senior executive steps in and takes the slice of time you were going to use. This could be for very legitimate reasons, but the end result is that you never get to speak with her (your thread never gets a chance to run).

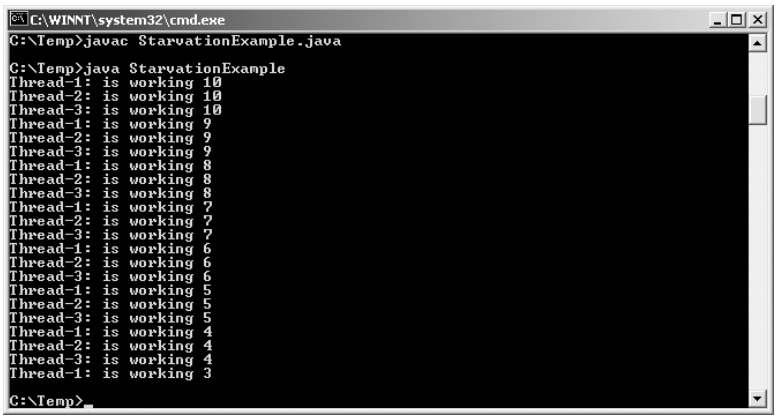
The example in Listing 4-10 shows a contrived example of three high-priority threads and one low-priority thread all trying to use the same resource.

Listing 4-10. *Starvation Example*

```
1  /**
2   * Demonstrate the concept of a starving thread.
3   */
4  public class StarvationExample {
5      public static void main(String args[]) {
6          // Ensure the main thread competes with the other threads
7          Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
8
```

```
 9          // Create an instance of this object
10          // Create 4 threads, marking number 1 as a very low priority
11          for(int i = 0; I < 4; i++) {
12              //create a runner
13              Runner r = new Runner();
14              r.setPriority(Thread.MAX_PRIORITY);
15
16              //set the first thread to starve
17              if (i == 0) {
18                  r.setPriority(Thread.MIN_PRIORITY);
19                  r.setName("Starvation Thread");
20              }
21              //start the thread.
22              r.start();
23          }
24
25          // Exit as soon as we possibly can
26          System.exit(0);
27      }
28
29      /**
30       * Create a thread, then cycle through its command ten times.
31       */
32      static class Runner extends Thread {
33          public void run() {
34              for (int count = 10; count > 0; count--) {
35                  System.out.println(getName() + ": is working " + count);
36              }
37          }
38      }
39  }
```

The starving thread never ran, as you can see in Figure 4-9.



■ **Note** The exact output varies with each running of this application, as thread priorities and scheduling cannot be guaranteed. However, in the majority of executions of this application, you will note that the `StarvationThread` either does not run at all, or runs far fewer times than the other three threads. In the cases where the `StarvationThread` does run, it is almost always after the other three threads have completed.

Understanding Atomic Operations

Atomic operations are indivisible. A thread will not be swapped out while in the middle of an atomic operation. In practical terms, the only naturally occurring atomic operations in Java are assignments. For example,

```
x = 45;
```

is an atomic operation when `x` is an `int`. The only exceptions to this rule are assignments involving doubles and longs. Those are not atomic.

Operations involving longs and doubles are essentially two operations, because longs and doubles are so big. The first part of the operation sets the high 32 bits, and the next part sets the low 32 bits. This means that mid-assignment to a long or double variable, your thread could be sliced out.

Your thread consists of a number of programmatic statements. These in turn consist of atomic statements. Your thread of execution could be swapped out anywhere, except while Java is in the middle of an atomic statement. For example:

```
1          x = 7;
2          y = x++;
```

is really

```
1          x =7;
2a         int temp = x + 1;
2b         x = temp;
2c         y=x;
```

Assume `x`, `y`, and `z` are class member variables. A thread swap between lines 2a and 2b could lead to unexpected results, for example, if the thread that has swapped in changes the value of `x` to, say, 13. You could end up with the bizarre result of `y` being 13 (and a headache). Of course, this problem would not exist if `x`, `y`, and `z` were local method variables, because then no other method could have access to them.

By wrapping your method in a synchronized block, you are effectively treating the entire method as a single atomic operation, as far as other threads' clients of that object are concerned. Even if another thread swaps in before your method is done executing, as long as the thread calls methods on your object that are synchronized, you are guaranteed that corruptions such as the previous one will not occur, because those methods will not execute until your method returns.

The example in Listing 4-11 shows a contrived example of 10 threads yielding at an inopportune time, causing invalid results.

Listing 4-11. *Nonatomic Operation Example*

```
 1  public class NonAtomic {
 2      static int x;
 3
 4      public static void main(String[] args) {
 5          for (int i = 0; i < 10; i++) {
 6              new Runner().start();
 7          }
 8      }
 9
10      static class Runner extends Thread {
11          private int validCounts = 0;
12          private int invalidCounts = 0;
13
14          public void run() {
15              for (int i = 0; i < 10; i++) {
16  //                synchronized (NonAtomic.class) {
17                      int reference = (int) (Math.random() * 100);
18                      x = reference;
19
20                      // either yielding or doing something intensive
21                      // should cause the problem to manifest.
22                      yield();
23  //                  for (int y = 0; y < 20000; y++) {
24  //                      Math.tan(200);
25  //                  }
26
27                      if (x == reference) {
28                          validCounts++;
29                      } else {
30                          invalidCounts++;
31                      }
32                  }
33  //              }
34
35              System.out.println(getName()
36                                 + " valid: " + validCounts
37                                 + " invalid: " + invalidCounts);
38          }
39      }
40  }
```

When you run this program, you will see that in the majority of cases, another thread has changed the value of x between when it is set (in line 18) and when it is checked (in line 28). An example of this is provided in Figure 4-10.

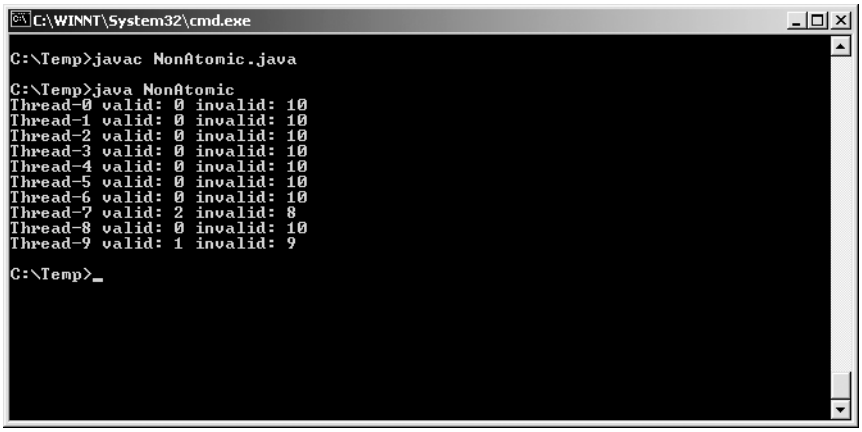


Figure 4-10. *Output from the nonatomic example*

In calling `yield` we caused this problem to be more obvious than would normally be the case; however, it is not the call to `yield` that is the problem—it is the coding itself. You can prove this by commenting out the call to `yield` in line 22, and uncommenting the for loop in lines 23–25. If you change the program in this way, and rerun the program, you will see that there are still often errors caused by the nonatomic nature of the code, even though there is no explicit yielding. You will also notice that changing the program in this way will cause it to take significantly longer to run, and if you watch your computer's CPU usage, you should find that these loops require a lot of computational effort.

If you put the code inside a synchronized block by uncommenting lines 16 and 33, you will find that the code now behaves in an atomic fashion regardless of whether you are using `yield` or the for loop.

Thread Safety Summary

You could code around any of the situations presented in this section as they occur, but that is the wrong approach. The best idea is to avoid the sorts of situations that lead to unsafe threading. There are exceptions to every rule, but in general don't nest locks, don't count on thread priorities, and watch for race conditions. The section titled "Threading Best Practices" later in this chapter helps you reduce the chances of creating situations where these sorts of problems can fester.

Using Thread Objects

This section contains some important general information you should keep in mind when working with thread objects directly.

Stopping, Suspending, Destroying, and Resuming

You should never use the `Thread.stop`, `Thread.suspend`, `Thread.resume`, or `Thread.destroy` methods, as they have been deprecated because they are inherently unsafe.

Instead of using these methods, you should have some variable that can be accessed by both the thread whose state you wish to change, and by the thread that wishes to change the state. The thread whose state is being changed should monitor this variable periodically, and safely change its state if the variable changes (by releasing any resources, closing files, etc.).

For more information, read the description of why these methods have been deprecated at `http://java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html`.

Thread States

Threads have six states: `Thread.State.NEW`, `Thread.State.RUNNABLE`, `Thread.State.WAITING`, `Thread.State.TIMED_WAITING`, `Thread.State.BLOCKED`, or `Thread.State.TERMINATED`. Java has specific rules about the transitions from state to state, and it's important to know them.

The most important rule is that a `Thread.State.TERMINATED` thread cannot go into any other state. Ever. The next most important rule is that a thread can only execute commands if it is `Thread.State.RUNNABLE`. When you first call the start method on a thread, the thread scheduler will transition it to `Thread.State.RUNNABLE`. At some point after that, depending on the whims of the thread scheduler, the thread will start to execute the statements in the `run` method body.

From the `Thread.State.RUNNABLE` state, a thread has four paths it can take, as illustrated in Figure 4-11:

- It can go on to completion, after which it enters the `Thread.State.TERMINATED` state.
- A thread can enter the `Thread.State.WAITING` state by calling `wait` or `join`.
- If the thread calls `sleep` or `wait` with a timeout value, it will enter the `Thread.State.TIMED_WAITING` state.
- If a thread cannot access a lock, it enters the `Thread.State.BLOCKED` state.

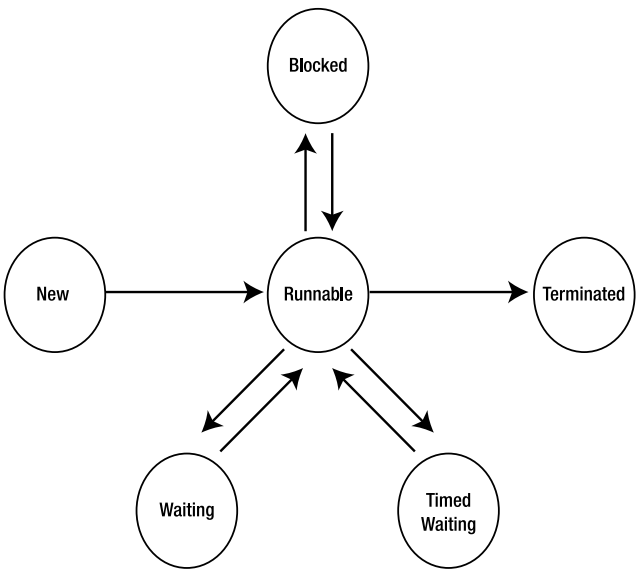


Figure 4-11. *Thread states*

More on Blocking

Blocking is an important phenomenon in threading and thus deserves a few more words of explanation. To quickly review, blocking threads do not use CPU cycles and are in a state of suspended animation. They do not wait for a time period to expire, but rather for an event to occur; specifically, a lock they were trying to acquire becomes available. Blocking threads deserve a healthy dose of respect because they do not release locked resources when they slice out of the CPU. A blocking thread that does not receive its lock can refuse to release resources that other threads might need, in addition to never coming out of hibernation, as shown in Listing 4-12. Figure 4-12 shows the output of this example.

Listing 4-12. *Blocking Example*

```
import java.net.*;

public class BlockingExample extends Thread {
    private static Object mylock = new Object();

    public static void main(String args[]) throws Exception {
        LockOwner lo = new LockOwner();
        lo.setName("Lock owner");
        lo.start();

        // Wait for a little while for the lock owner thread to start
        Thread.sleep(200);

        // Now start the thread that will be blocked
        BlockingExample be = new BlockingExample();
        be.setName("Blocked thread");
        be.start();

        // Wait for a little while for the blocked thread to start
        Thread.sleep(200);

        // Now print the two threads states
        printState(lo);
        printState(be);
    }

    // start a thread.
    public void run() {
        // wait for the mylock object to be freed,
        // which will never happen
        synchronized (mylock) {
            System.out.println(getName() + " owns lock");
            System.out.println("doing Stuff");
        }
```

```
35          System.out.println(getName() + " released lock");
36      }
37
38      private static void printState(Thread t) {
39          System.out.println();
40          System.out.println("State of thread named: " + t.getName());
41          System.out.println("State: " + t.getState());
42          System.out.println("begin trace");
43          for (StackTraceElement ste : t.getStackTrace()) {
44              System.out.println("\t" + ste);
45          }
46          System.out.println("end trace");
47      }
48
49      static class LockOwner extends Thread {
50          public void run() {
51              synchronized (mylock) {
52                  System.out.println(getName() + " owns lock");
53                  try {
54                      ServerSocket ss = new ServerSocket(8080);
55                      ss.accept();
56                  } catch (Exception e) {
57                      e.printStackTrace();
58                  }
59              }
60              System.out.println(getName() + " released lock");
61          }
62      }
63  }
```

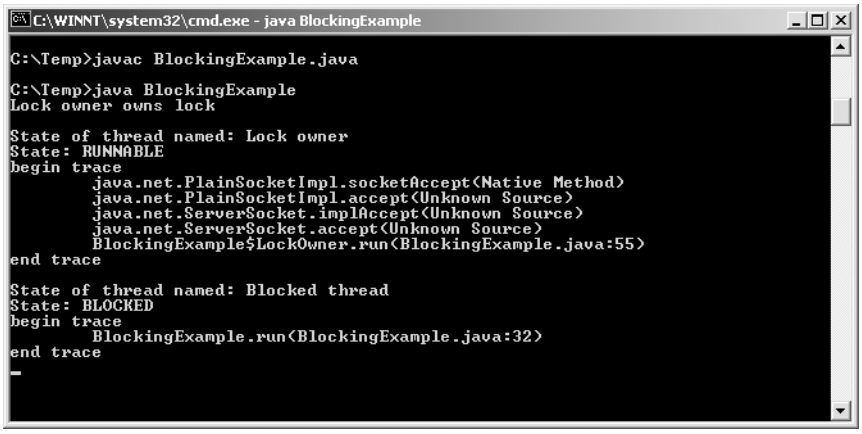


Figure 4-12. *Output from the blocking example*

■ **Note** At line 54 we attempt to start listening on port 8080. There is no particular reason for picking this port, other than the fact that it is not a port that is officially assigned to any other application, and therefore there is a reasonable chance that this port will be available. However, if you are running another application that also uses port 8080 (for example, the Tomcat Web Application Server by default uses port 8080), then an exception (`java.net.BindException`) will be thrown. If you see that exception, just change the port number in line 54 from 8080 to some other number, for example 9090.

The `myLock` object is never released, because the `LockOwner` thread never finishes. Because the `myLock` object is never released, the thread that is waiting for `myLock` (in this case, the `BlockingExample` thread) is blocked and never gets a chance to execute.

The main thread would continue executing, however, if it were not dependent on the locked resource. Again, this is because blocking threads do not use CPU cycles.

■ **Caution** When talking about an application or thread pausing until it can get some I/O, many publications use the standard terminology of saying that the thread or application is blocked for I/O. According to their terminology the `LockOwner` thread would be blocked until a client connects to the socket. However, as the previous example shows, the JVM state for a thread waiting for I/O may be `Thread.State.RUNNABLE`, or it may be some other state depending on how the code for the I/O class is implemented. When debugging your code, or when perusing other publications, you need to be aware that Java does not have a separate state for threads that are paused for I/O.

InterruptedException

`InterruptedExceptions` typically occur when a thread has been paused (sleeping, or waiting, or trying to join several threads) for too long, and another thread deliberately calls `interrupt` on the thread to be interrupted to bring it out of that state. Think of an `InterruptedException` as one thread violently shaking another thread back into consciousness. When a thread is shaken awake in this manner, the first thing it does is execute whatever code is inside its catch (`InterruptedException`) clause if it has one.

As an example, consider if you wrote a `Data` class that has a `lockRecord` method that must wait for a record to become available before locking it, but it does not have a timeout. After the `Data` class has been approved, you are asked to create a subclass of the `Data` class that does handle timeouts when locking. You could rewrite the entire locking code from scratch, or you could create a separate thread that interrupts the thread locking of the record if the lock is not acquired within the timeout period.

■ **Note** It is possible for a thread to be interrupted at any time, even if it has not called `sleep`, `wait`, or `join`. In such a case the thread will continue to execute normally until it calls `sleep`, `wait`, or `join`, at which time an `InterruptedException` will immediately be thrown from the called method.

Synchronization

There are two natural divisions when you talk about synchronization: object synchronization and client synchronization. This section explains the differences between the two.

Internally Synchronized Classes

A `Vector` is a synchronized object and an `ArrayList` is not. What does this mean? It means all the methods that can access a `Vector`'s internal data are synchronized. Any changes made to a `Vector` object by one thread are guaranteed to be immediately visible to any other thread, while changes made to an `ArrayList` are not so guaranteed. It does not mean that you are using the `Vector` object in a thread-safe way in your code. If you wanted that guaranteed, you would have to synchronize on the `Vector` object directly, just as you would have to synchronize directly on the `ArrayList`.

■ **Note** `Vector` is an example of a thread-safe collection; however, it is rare that the thread safety offered by `Vector`s is required (see the next paragraph for information on what thread safety is guaranteed). If you do need this kind of thread safety, you may be better off using `Collections.synchronizedCollection` or similar methods to create a thread-safe collection with the characteristics of your preferred base collection.

A synchronized object only promises this: A method called on that object will behave atomically. For example, imagine that you are running two threads and that `myVector` and `myArrayList` are class member variables. `Thread1` has started to run line 1, and `Thread2` has yet to start line A.

Thread1:

```
1          myVector.add("Hello");
2          myArrayList.add("Hello");
```

Thread2:

```
A          myVector.add("Cruel");
B          myArrayList.add("Cruel");
```

Now, say that in the middle of line 1, `Thread1` slices out. `Thread2` will be unable to slice in because `Vector`s are synchronized,. You are therefore guaranteed that `Thread2` line A will add `"Cruel"` after `"Hello"` for `myVector`.

This guarantee does not exist for `myArrayList`. That is, if `Thread1` sliced out in the middle of line 2 and `Thread2` sliced in, the order of adding the items to the `ArrayList` is indeterminate. Even worse, if one of the adds causes a resize of the internal array, all sorts of strange things can happen. We might see a `NullPointerException` or `ArrayIndexOutOfBoundsException`, or one of the adds might be "lost." All sorts of unpredictable nastiness can emerge from unsynchronized access by multiple threads.

■ **Note** The reason that we've specified threads slicing out in the *middle* of line 1 or 2 is that the `add` method for both `Vector`s and `ArrayList`s are methods that perform several actions before actually adding the requested object to the underlying store.

`Vector`s are not superior to `ArrayList`s, nor are `ArrayList`s superior to `Vector`s. There are times when you do not need the overhead of synchronization on the `Collection`, and there are times when you absolutely must have it. For example, a local method variable might as well be an `ArrayList`, because it only exists in the context of that particular method; thus, there is no opportunity for any other threads to modify it.

However, where possible we recommend against using the `Vector` class. There is the possibility that a junior programmer may incorrectly believe that use of a `Vector` is providing thread safety other than what synchronized classes actually guarantee. Plus, there is the risk that a later programmer may change the `Vector` to some other class, not realizing that a synchronized class was required.

Synchronized objects have their purpose, as do unsynchronized ones. Part of earning your Java developer certification is knowing what those purposes are.

Client Synchronization

Client synchronization is the process of making sure that no other thread interrupts your method while it is mid-stride. For example, while `myVector.add(Object)` is a synchronized operation, the internal synchronization of the `Vector` object did not guarantee that another thread would not empty the `myVector` before the next step of your method executed. Consider the following example:

```
1      public boolean addDVD(DVD dvd ){
2          for (int i = 0; i < myVector.length(); i++) {
3              doSomethingWith(myVector.get(i));
4          }
5      }
```

While you are guaranteed that line 2 will not be hijacked mid-stride, you are not guaranteed that some other thread will not set empty the `myVector` object by the time you get to line 3. The current thread could slice out between lines 2 and 3, and the thread that picks up could just happen to be one that corrupts `myVector`. The chances are probably small, but the possibility for mischief does exist.

Be aware of such risks, even if you decide to accept them: Another thread could always slice in between two unsynchronized lines of code even if they're calls to synchronized methods.

To guarantee exclusivity, you could modify the method in one of two ways. The first way is to synchronize the method:

```
1      public synchronized boolean addDVD(DVD dvd ) {
2          for (int i = 0; i < myVector.length(); i++) {
3              doSomethingWith(myVector.get(i));
4          }
5      }
```

This denies any other threads the capability to call any synchronized methods on this particular object until addDVD returns. Depending on your class's structure, this may be sufficient.

The second way is to synchronize on the myVector member variable:

```
1      public boolean addDVD(DVD dvd ) {
2          synchronized (myVector) {
3              for (int i = 0; i < myVector.length(); i++) {
4                  doSomethingWith(myVector.get(i));
5              }
6          }
7      }
```

This keeps any other method that calls from modifying the myVector object for the duration of lines 3, 4, 5, and 6. This holds even if your thread slices out during those lines.

As with all exclusivity, this only applies to methods that follow synchronization rules. If those methods do not internally synchronize on the myVector object, then they won't obey the protocol. Thus, the guarantee of thread safety would no longer hold.

Incidentally, this is a great justification for controlling access to your class's member variables through encapsulation. If all member variables are private, then you can control access to sensitive data by synchronizing all the relevant methods. This is exactly what the Vector object does. Synchronized objects such as Vectors do not synchronize on their internal data state; they synchronize the methods used to access that data. Sun recommends and encourages encapsulation. For example, it is explicitly evident in JavaBeans.

Multithreading with Swing

Swing components, by and large, are not thread-safe. Swing uses a single thread to deal with UI events from the underlying operating system. That is, in general Swing components don't *need* to be thread-safe as they're typically accessed from only one thread. This means that processing an event, such as a button click, will cause other events to be ignored until the first event is finished. Your UI could be very unresponsive if the event handler associated with the event is long.

The point here is that your Swing client could become slow if you are doing a big operation for a given event. This may be okay on the SCJD exam, but you need to be aware of it and document it. If your requirements state that UI responsiveness is very important, then you need to think about spawning threads to deal with more intensive operations, and that's when you have to worry about all the thread safety concerns, synchronizing as appropriate.

General Principles of Threading with Swing

Once a Swing component has called setVisible(true) or pack(), all updates should be done via the event dispatcher thread. This helps avoid race conditions that can occur as a result of the various requests from the RepaintManager.

In general, you only need to be concerned about threading in your Swing application when you are updating your components based on a thread other than the event dispatcher thread—specifically, in cases where you need a responsive GUI that cannot wait for RMI and/or socket calls across the network.

You should make sure that you can't afford to wait for these before you start down this path: It's a long and error-prone approach, and it will probably complicate your life unduly as

far as the SCJD exam is concerned. If you do decide to take this route, execute your lengthier calls through a separate thread, or worker thread, and then feed your results back into the GUI using the event dispatcher. For an excellent example of this technique, please visit `http://java.sun.com/products/jfc/tsc/articles/threads/threads2.html`.

■ **Caution** You must not use the `SwingWorker` class directly in your assignment. The assignment specifies that you are only allowed to submit code you write yourself, so submitting the `SwingWorker` class could lead to disqualification. You can, however, use the `SwingWorker` class to learn the techniques needed, then make your own class(es).

Remember that calling the various setters on a component is not a thread-safe way to get that event into the event-dispatching queue. This process does generate events, but the event dispatcher thread does not always handle these.

Updating Components in the Event Dispatcher Thread

So how do you get your events into the event dispatcher thread? You use the `SwingUtilities.invokeAndWait` and `SwingUtilities.invokeLater` methods. The first method, `invokeAndWait`, blocks until the event fires. `invokeLater` does not block; it simply adds your code to the queue of events to be fired. The parameter to both of these methods is a `Runnable` interface. The `run` method of the parameter is then called by the event dispatcher.

Threading Best Practices

Here are some general practices that should help you avoid the dark side of threading:

- Don't nest locks. If you have two synchronized blocks on two different objects in your execution path, you're probably heading down the wrong road.
- Avoid multithreading with Swing, if possible. It can be argued that multithreading activities for distributed calls from a Swing front-end, for example, only offer illusionary responsiveness. That is, just because the GUI appears to respond quickly doesn't mean that it actually does so. Do you really want your client to think an operation is finished when in fact it isn't? Does the illusion actually serve a useful purpose in your application?
- For threads that are providing lengthy services, it's a good idea to yield occasionally. For example, if your thread is starting a lengthy process but the user wants to cancel, that canceling event might not get read until your thread stops executing. By yielding occasionally, you give other threads a chance to slice in at a time that is opportune for your thread's execution state. It's best to release any locks before yielding.

- There's no need to synchronize methods that don't use state information in the class. If your method doesn't use internal member variables, doesn't modify an external object, and doesn't modify an object that was passed in as a parameter, don't synchronize it.
- Don't oversynchronize. Make sure you know exactly why your method or data structure is synchronized. If you can't articulate the need for synchronization to your own satisfaction, you probably should reexamine the original need.
- Immutable objects, such as `String`s and `Integer`s, never need synchronization. This applies even to member variables, because immutable objects can't be modified. However, a member variable referring to an immutable object may still need synchronization if the member variable itself is mutable.
- Don't multithread needlessly. Unless your threads are doing a lot of blocking, single threading is often faster.
- When working with Swing, feel free to create and insert components into any container that hasn't been realized yet. To be realized, a component must be able to receive paint or validation events—this means before `show`, `setVisible(true)`, or `pack` has been called on the component.
- Don't get confused by internally synchronized classes such as `Vector`. All an internally synchronized class promises is to perform its actions atomically. This means that if one thread code executes `myVector.add("test")`, and another thread slices in, if that new thread executes `myVector.contains("test")` it will return true. This behavior is not guaranteed with, say, an `ArrayList`. You will usually need to use synchronized methods to interact with these variables.
- Try to minimize actions performed within your synchronized blocks, since no other thread will be able to obtain the lock your code is synchronized on for the duration, thus reducing concurrency.
- Avoid depending on thread priority as a mechanism of thread control. Different operating systems use distinct algorithms to deal with thread priority, and different JVMs use different algorithms still. This could lead to your program acting wildly different from one platform to the next. You best bet is to avoid the ambiguity of thread priorities, especially where the SCJD exam is concerned.
- Don't synchronize your thread's `run` method. Synchronizing your thread's run method can lead to a lot of complications, and it's a bad idea.
- If you are checking a condition in a synchronized block of code, use a while loop and not an if statement. If your thread slices out in the middle of executing your if statement, then the thread will resume exactly where it left off when it slices back in. Counterintuitively, this is not a good thing, because the condition that was true when your thread sliced out might no longer be true, because another thread could have changed it.

- While loops, on the other hand, are re-checked before your thread exits the block. Thus, relevant state changes will be detected. The following snippet shows an example of this technique:

```
synchronized (lockedRecords) {
    while (lockedRecords.contains(upc)) {
        lockedRecords.wait();
    }
    //do stuff
    lockedRecords.notifyAll();
}
```

- Use `notifyAll` instead of `notify`. `notify` only wakes a single waiting thread, while `notifyAll` wakes all waiting threads.

Summary

In this chapter, we discussed some of the possibilities available to you when using threads, and we pointed out some trouble spots to avoid. We discussed safe threading issues, threads and Swing, blocking, waiting, and a host of other topics.

The best way to understand threading is to design your threading scheme, make predictions about how it will function, and then test those predictions with the handy method in the `Thread` class, `holdsLock(Object)`. `Thread`'s `getState()` and `getStackTrace()` methods can be very handy for checking what another thread is doing. If your threads aren't behaving the way you expected them to, explicitly record your assumptions (we suggest writing them down) and then examine them one by one. Threading is a lot like grammar: There are a lot of rules, but eventually you develop a sense for what works and what doesn't. (Or so I'm told.)

As you read the next chapters, please don't hesitate to refer back to this chapter when needed.

FAQs

Q What happens when a synchronized block/method calls an unsynchronized one?

A The unsynchronized block/method is treated as if it was synchronized.

Q What happens when a synchronized block/method calls a synchronized one within the same object?

A If both methods are synchronizing on the same object, then nothing unusual happens as far as you're concerned—there's no double synchronization or risk of deadlock. The thread acquires an extra lock on the object, which in turn dissipates when the lock is released.

Q Does locking an object lock all of its internal variables?

A No, absolutely not. Doing so would place a tremendous burden on the JVM, because it would lock all their internal objects, and then the internal object of member variables in turn, and so on. Java assumes that you understand this, and that you are only locking what you mean to lock.

Q How much slower is a synchronized method than an unsynchronized one?

A It's difficult to say, but unscientific tests seem to indicate that a synchronized method is anywhere from 1.5 to 150 times slower than an unsynchronized method, depending on contention. Noncontended synchronization may have negligible impact on time.

Q Can you synchronize data?

A No, you can only synchronize access to data.

Q Aren't vectors synchronized data?

A No, vectors provide synchronized methods that access their data. It's an important difference.

Q What's the difference between synchronizing on a static member variable and a local one?

A There's no difference, really. You're still obtaining a lock on an object. But by synchronizing on a static member variable, the lock can be shared across multiple objects who share that static object as a class member variable.

Q What does it mean when people say that an object, such as a `Vector`, is synchronized?

A It means that all public methods of that object are synchronized when they access shared data. Generally, it means that calling a method on that object is an atomic action. Any other thread calling a method on that object will block until your thread completes its operation on that object.

Q What happens when one method synchronizes on a member variable, and another synchronizes on the `this` keyword, and the two have to interact?

A Then you are acquiring two locks on two distinct objects—namely `this` and the member variable—and it is very important to be careful. Otherwise, you are opening the door to deadlock. You should consider carefully whether you really need to do this, or whether you can synchronize on the one object.

Q How can I restart a thread after it has died?

A You can't. You will have to create a new thread.

Q Most of the examples provided show the main application implicitly waiting until all the created threads had finished before exiting. Is there a way of exiting the program without waiting for a thread to complete?

A Yes—you can explicitly call `System.exit`, or you can call `Thread.setDaemon(true)` on the thread you don't want to wait for *before* you start it running. See the creation of the `IceCreamMan` thread for an example of how to set a daemon thread.

CHAPTER 5

The DvdDatabase Class

In this chapter, we will start working on our assignment by developing a `DvdDatabase` class, which will implement the `DBClient` interface described in Chapter 3. Since both the client and the server need this class, starting the assignment with this class is quite logical.

As you work through this chapter, you will use several important concepts introduced in Chapters 2 and 3:

- Design patterns
- Generics

As mentioned earlier in this book, there are several areas in our sample project where we feel we *must* deviate from the project provided by Sun. Nowhere is this more important than with the classes presented in this chapter. Either the code used in this section of our sample assignment will quite often be more detailed than your assignment requirements, or we will use shortcuts that you cannot use in your assignment. In particular, we specify that `java.io.IOException` can be thrown in our interfaces and classes, greatly simplifying our development; we complicate our assignment by having timeouts on locks; and we use `ReadWriteLock`s to improve concurrency. Regardless, all the information you need to develop a good solution for *your* requirements is presented in this chapter.

We also discuss issues that you might want to consider for your assignment that are not required but that may make your submission more professional. This includes the topics of caching data, handling deadlocks, managing client crashes, and using multiple notification objects to reduce CPU usage by threads trying to obtain a logical record lock.

Creating the Classes Required for the DvdDatabase Class

The `DvdDatabase` class uses a few other classes as inputs, outputs, or exceptions. We need to build these before we can build the `Data` class itself.

The DVD Class: A Value Object

The `DvdDatabase` class uses a class named `DVD` to contain the data corresponding to a DVD that can be reserved. This type of object is referred to as a value object. The Value Object design pattern is also referred to as the Transfer Object pattern. The name *value object* represents the type of object it is; it contains all the values relating to a particular object (in our case, all the

values we wish to track relating to a DVD). *Transfer object* is named for one of the more common uses of this design pattern: creating a single class to transfer data between two separate systems, usually between a client and a server.

The reason for such an object is fairly simple—it can make your code easier to read, and improve performance of your system as a whole. Your code will be referencing fields within a single object, which will make it clearer within your code that the fields are all related. And since you can send and retrieve the entire value object in one call to another method, your code will perform much better than if it had to retrieve each of the fields separately. This is especially useful when retrieving value objects over a network, as only a single network call is required, in contrast to the multiple network calls required if you were retrieving each field separately.

Sections of the `DVD` class are detailed next; however, since much of the code is repetitive, we won't display all the code here but only important sections of it. The entire code for this method can be downloaded from the Apress web site in the Source Code section; more details on what can be downloaded and how the code can be compiled and packaged are listed in Chapter 9. The line numbers shown here correspond to the real line numbers in the online source code—therefore, you'll note gaps in the numerical sequence because we skip comments and repetitive code.

```
 1  package sampleproject.db;
 2
 3  import java.io.*;
 4  import java.util.*;
 5  import java.util.logging.*;
 6
..
13  public class DVD implements Serializable {
..
19      private static final long serialVersionUID = 5165L;
```

We will be using this class to transfer the data that represents a DVD between the client and the server, potentially on different computers. Doing this, however, relies on the client and server both agreeing on what a `DVD` object is. If either system believes that the internal data structure is different (for instance, if the server believes that the issue date is a Java `Date` object, while the client believes that it is a `String` object), then the whole system of transferring data is called into question. If you are lucky, you will receive some form of class cast exception, or a similar problem. If you are unlucky, your client and server will be able to work with the data, even though it is potentially wrong, and your data will be corrupted.

Java provides a mechanism for classes that use a serialized object to confirm that the serialized object conforms to a known structure: the `serialVersionUID`. This is done for us by the JVM itself; if a class tries to deserialize an instance of a class where the `serialVersionUID` does not match the known value, an `InvalidClassException` will be thrown.

■ **Note** Serialization is covered in detail in the first few sections of Chapter 6.

You can declare a `static final long serialVersionUID` in your code, as shown in line 19. Note that the `static final` and `long` modifiers are mandatory. Any access modifier may be used, including the `private` modifier we used in this code. This enables you to decide whether a change to your `Serializable` class requires recompilation of your client classes. For example, if we added an extra field to our `DVD` class to contain the number of Emmy nominations the DVD received, it would not affect those clients that were compiled prior to the addition of the field; they would still be able to access every field that they were previously able to access. The same would **not** be true if you deleted a field, even if it was one your clients were currently not using—*potentially* they would be unable to access the field, so the change to the class would require clients to recompile.

If you do not declare a `serialVersionUID` in your `Serializable` class, the Java compiler will generate one for you, based on various aspects of the class, such as the name of the class, the names of the methods, the names of the fields, and so on. If any of these change, then your `serialVersionUID` will change as well. So even for our example of adding an unimportant field earlier, an autogenerated `serialVersionUID` will change. For this reason, it is always recommended that you define your own `serialVersionUID`. (For those who are curious, the `serialVersionUID` we used is just the last four digits of the ISBN number for this book.)

■ **Tip** To learn more about object serialization, refer to the Sun documentation available at `http://java.sun.com/j2se/1.5.0/docs/guide/serialization/`.

Our `DVD` class then specifies all the fields we want to associate with a DVD. Again, comments have been excluded, so the line numbers are not in numerical sequence:

```
 89      private String upc = "" ; // The record UPC identifier
 94      private String name = ""; // The movie title
100      private String composer = ""; // The music composer
106      private String director = ""; // The director's name
112      private String leadActor = ""; // The lead actor's name
118      private String supportingActor = ""; // The supporting actor's name
123      private String year = ""; // The movie's release date
129      private int copy = 1; // The number of DVDs in stock
```

■ **Note** A UPC (Universal Product Code) number is a unique number assigned to retail items in U.S. and Canadian stores. The equivalent code in Japan is the JAN code, and the next standard will be the European EAN code, which has been designed to increase the number of potential codes to cater for the entire world. For the purposes of this book we have stuck with the original UPC code, which we use to ensure that each record in our data file has a unique index.

We then have three constructors: a default constructor (required for any serialized object), a constructor that assumes we have only one copy of the DVD in stock, and a constructor that allows us to specify how many copies of the DVD we have.

```
140     public DVD() {
141         log.finer("No argument constructor for DVD called");
142     }
...
157     public DVD(String upc, String name, String composer,
158                 String director, String lead, String supportActor,
159                 String year) {
160         this(upc, name, composer, director, lead, supportActor , year, 1);
161     }
162
163     /**
164      * Creates an instance of this object with a specified list of initial
165      * values.
166      *
167      * @param upc The UPC value of the DVD.
168      * @param name The title of the DVD.
169      * @param composer The name of the movie's composer.
170      * @param director The name of the movie's director.
171      * @param leadActor The name of the movie's leading actor.
172      * @param supportingActor The name of the movie's supporting actor.
173      * @param year The date the of the movie's release (yyyy-mm-dd).
174      * @param copies The number of copies in stock.
175      */
176     public DVD(String upc, String name, String composer, String director,
177                String leadActor, String supportingActor, String year,
178                int copies) {
179         log.entering ("DVD", "DVD",
180                       new Object[]{upc, name, composer, director, leadActor,
181                                    supportingActor, year, copies});
182         this.upc = upc;
183         this.name = name;
184         this.composer = composer;
185         this.director = director;
186         this.leadActor = leadActor;
187         this.supportingActor = supportingActor;
188         this.year = year;
189         this.copy = copies;
190         log.exiting ("DVD", "DVD");
191     }
```

The comments were left in the final constructor, to show how parameters are declared in Javadoc comments. As you can see, there is no requirement to specify the type of each parameter.

It might also be noted that there are no calls to the logger in the second instantiator. Since this instantiator immediately calls the third instantiator, we are relying on the logging available there.

For each variable listed in lines 89–129, we have a getter and a setter. For example, the `composer` variable specified in line 100 has a `getComposer` getter and a `setComposer` setter as follows:

```
199    public String getComposer() {
200        log.entering("DVD", "getComposer");
201        log.exiting("DVD", "getComposer", this.composer);
202        return this.composer;
203    }
...
211    public void setComposer(String composer) {
212        log.entering("DVD", "setComposer", composer);
213        this.composer = composer;
214        log.exiting("DVD", "setComposer", this.composer);
215    }
```

Using getters and setters (so named because the first verb of the method is either get or set) enables us to expose only the sections of the method we want to expose, and allow only certain operations on them. For example, we might decide that the number of copies of a DVD can never go below zero; if we allow clients to directly modify the field, then they can set it to any number they like, including negative numbers. But by having a setter for numbers of DVDs, we can include our business logic to ensure that the client cannot set a negative number.

■ **Note** In line 200 we call `log.entering` and then call `log.exiting` immediately afterwards in line 201. As always, there was a design decision to be made here: Do we create our own log message (probably calling `log.finer`), or do we use the two calls that really don't provide much useful information? Once again, there is no one "right" way to handle this. The disadvantage of our chosen method is that we are providing unnecessary logging, which does not produce much useful information. However, on the positive side we are still using the same standard logging format for **all** "entering" and "exiting" log messages—contrast this with the probability that if we were to write our own specialized log method for these getters and setters, it is likely that the log messages would vary from method to method. In addition, if we ever needed to do more work in this method, the work can be put directly between the two log messages; if we had created our own specialized log message, we would probably have to remove it and replace it with the standard log messages.

The logging provided for the getters and setters is possibly overkill; however, it does highlight one valuable debugging tool: It can be very useful to see what parameters are passed into a method, and what the end result of executing the method is. By logging the `composer` method parameter at line 212, we will have a record of what was passed into the method, and by logging the `this.composer` instance variable at line 214, we will have a record of what the end result of executing the `setComposer` method was. If at a later date this method is updated (for example, adding code to ensure the first letter of each part of the composer's name is capitalized), we will still get logging showing what the input and results are.

The remaining getters and setters have not been shown; however, they all follow the same form as the `getComposer` and `setComposer` methods shown earlier.

■ **Tip** Most IDEs have the ability to automatically generate getters and setters for you, including generating rudimentary Javadoc comments. After using such facilities, you can go to each generated method, add any required business logic, and modify the Javadoc comments to suit.

One last point worth considering is whether we would ever want to compare two instances of a `DVD` to see if they are the same. In a stand-alone application, we might consider allowing only a single instance of each `DVD` object to be created, in which case we would be able to check that they were equal by using the `==` comparison. However, since we might be getting multiple copies of the same `DVD` record over a network, each copy will be deserialized into a separate instantiation of the `DVD` class, so we cannot use the `==` comparator. We should therefore look at overriding the `equals` method of the `Object` class.

```
416     public boolean equals(Object aDvd) {
417         if (! (aDvd instanceof DVD)) {
418             return false;
419         }
420
421         DVD otherDvd = (DVD) aDvd;
422
423         return (upc == null) ? (otherDvd.getUPC() == null)
424                              : upc.equals(otherDvd.getUPC());
425     }
```

At line 417, we ensure that we have been given an instance of `DVD` to compare against. Once we have confirmed this—and only when we have confirmed this—we can convert the supplied object into an object of type `DVD`, as shown in line 421. Finally, in lines 423 and 424 we utilize the fact that UPC numbers are unique for DVDs, and compare the UPC numbers.

■ **Note** Lines 423 and 424 use a ternary operator to determine whether to return true or false. Everyone has their own opinion as to whether ternary operators increase or decrease readability. This is something you will have to decide for yourself, possibly on a case-by-case basis. In this particular case, we believe it improves readability, compared with the alternative:

```
if (upc == null) {
    return otherDVD.getUPC() == null;
} else {
    return upc.equals(otherDVD.getUPC());
}
```

In that last `return` statement, we can safely use the `String.equals` method to compare the contents of the two `UPC` fields. We could not use it earlier, as we did not know if the local `UPC` field contained `null` (in which case the last `return` statement would have generated a `NullPointerException`); likewise, we did not know if the other `UPC` value contained `null`, so we could not reverse the logic (`otherDVD.getUPC().equals(upc)`) as it may also have thrown a `NullPointerException`. It is only after we have validated that *at least* one of the two `UPC` values is not `null` that we can use the `equals` method.

If the UPC did not uniquely identify the DVD, we might consider having more complex logic; for example, we might compare the UPC, the title, and the main actor (it is unlikely that an actor would work in a film with the same title as one they had previously worked in, and that had the same UPC). However, we would probably not check the number of DVDs available, as they are not unique to the DVD.

It is **strongly** recommended that whenever you override the `equals` method, you should also override the `hashCode` method, as the two are often used together. Once again, we have used the fact that the UPC number uniquely identifies the DVD, and simply reused the UPC's hash code as the DVD's hash code:

```
462      public int hashCode() {
463          return upc.hashCode();
464      }
465  }
```

It is important to try to return different hash codes for instances of a class that are not considered equal where possible, while at the same time the same hash code value must be returned for instances of a class that are considered equal no matter how many times it is run on a single JVM (unless some of its internal field data is changed in such a way that it is no longer considered equal to the old instance). If we had added extra checking to our `equals` method (such as checking the actor's name), then we might want to consider modifying our hash code generator to take into account the extra uniqueness checking. For example, we might choose to add the hash codes for both the UPC and the actor's name, and return the sum as the new hash code.

■ **Tip** A unique hash code per unique instance of a class may not be possible. Furthermore, even if you spent considerable time developing an algorithm to generate a hash code that is highly likely to be unique, there would almost certainly be a loss of efficiency caused by the extra time required to execute the algorithm. If you need to generate your own `hashCode` methods, we recommend that you don't spend too much time in this method **unless** code profiling shows that a significant amount of time is being spent in methods that rely on the hash code (such as the `HashMap.get` method).

Discussion Point: Handling Exceptions Not Listed in the Supplied Interface

You may find that in order to create your `Data` class, you must call some methods that throw exceptions that are not listed in the interface supplied by Sun. You will then have to decide what to do with those exceptions, one by one. There is no single perfect solution for how to handle this (although there *are* some bad solutions).

You will have to decide for yourself which approach is right for you. Look at the instructions *you* downloaded from Sun (remember, each set of instructions can be different in small ways), and then decide how best to meet *your* requirements. Whatever you decide, be sure to document your decision in your design decisions document, as well as in the source code itself.

To help illustrate the various possibilities, let's use the following base code:

```
1  import java.util.logging.*;
2
3  public class InterruptedExceptionExample extends Thread {
4      static Logger log = Logger.getAnonymousLogger();
5
6      public static void main(String[] args) throws InterruptedException {
7          InterruptedExceptionExample iee = new InterruptedExceptionExample();
8          iee.start();
9
10         while (iee.isAlive()) {
11             log.info("main: waiting 5 seconds for other thread to finish");
12             iee.join(5000);
13
14             if (iee.isAlive()) {
15                 log.info("main: interrupting other thread.");
16                 iee.interrupt();
17             }
18         }
19         log.info("main: finished");
20     }
21
22     public void run() {
23         try {
24             getLock();
25         } catch (LockAttemptFailedException dle) {
26             log.log(Level.WARNING, "Lock attempt failed", dle);
27         }
28     }
29
30     public void getLock() throws LockAttemptFailedException {
31         // try to get some resource that we will presumably never get.
32         for (;;) {
33             try {
```

```
34                  synchronized (InterruptedExceptionExample.class) {
35                      log.info(getName() + ": waiting for some resource.");
36                      InterruptedExceptionExample.class.wait();
37                  }
38              } catch (InterruptedException ie) {
39                  // this is the bit we are interested in
40              }
41          }
42      }
43
44      public class LockAttemptFailedException extends Exception {
45          public LockAttemptFailedException(String msg, Throwable t) {
46              super(msg, t);
47          }
48      }
49  }
```

■ **Note** When we call `join` in line 12 there is the risk that an `InterruptedException` may be thrown. Handling this exception is not relevant to our discussion point, so we have decided to allow the `main` method to throw the exception as specified in line 6.

As noted in the comment at line 31, the `getLock` method needs something to happen before it will complete—but for the purposes of this example we have not declared what that will be. If this were a section of code from a real-life application it might be waiting for a lock to be released, or it might be waiting for a resource to become free. All we care about for this example is that, regardless of what we were waiting for, it won't ever happen.

The important point of this example code is that we are calling the `wait` method in line 36, and the `wait` method can throw an `InterruptedException`; however, the signature for the `getLock` method states that only the `LockAttemptFailedException` (which does not extend `InterruptedException`) will be thrown. So we must do something with the `InterruptedException` that we are catching at line 38.

■ **Tip** As shown in lines 34 and 36, every class can itself be used as the object to be synchronized on. You do not even need an instance of the class—you can just use the `class` literal. In most cases, you will find that there are specific objects you wish to synchronize on; however, if there are no apparent objects, do not automatically assume that you are going to have to create an object just so you can use it as a mutual exclusion lock (a lock that mutually excludes access to all other threads as long as one thread owns the lock; also known as a mutex). Conversely, don't use this feature where it doesn't apply—if you want to make explicit what a particular mutex is being used for, it might be worthwhile creating an `Object` with a suitably descriptive name just to make the code clear. Making these sorts of decisions is all a part of being a developer.

Swallowing the Exception

In the code shown in the previous section, lines 38 through 40 do absolutely nothing with the exception. This is commonly referred to as "swallowing the exception." Once those lines have completed, it is as though your code has swallowed the exception whole: there is no trace of it left for anyone to see.

While the Java language will allow you to get away with doing this, it is considered extremely poor programming, and will almost certainly cost you some marks in your assignment (and cause heated discussions between yourself and whoever has to maintain your code at work). An exception is, as its name states, something that should not have happened. But by swallowing the exception, we have not allowed for any handling of this event, nor have we provided any form of tracking down what went wrong later.

If we were to run this program now, we would find that the main method could never complete. This is shown in Figure 5-1.

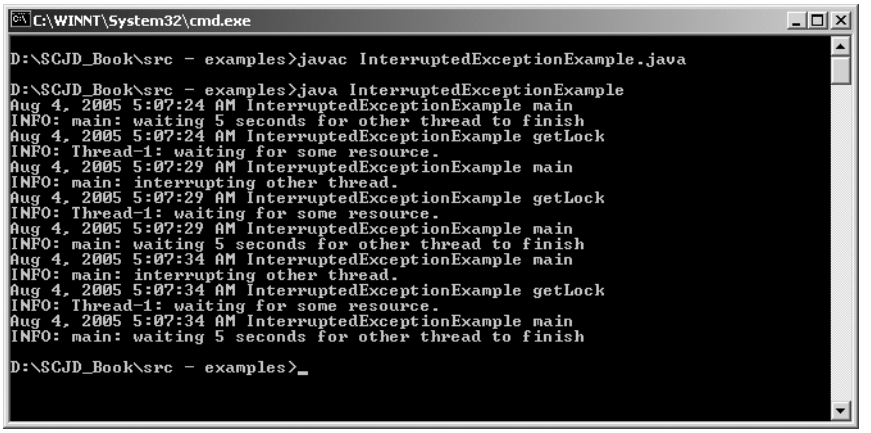


Figure 5-1. *Swallowing the exception*

Logging the Exception

We might decide that for business reasons the exception should be ignored. For example, a business rule might state that for audit reasons transactions cannot be canceled—they must be processed fully, then a new transaction created to undo the first transaction (this is a common business requirement). In such a case, we might decide to ignore the fact that the client is trying to interrupt us, and just continue trying to gain the lock.

In such a case, though, we do not want to just swallow the exception—that would leave us with no evidence that the client is trying to interrupt the thread. So what we should do is some form of logging. Here are some examples:

```
38              } catch (InterruptedException ie) {
39                  log.info("Ignoring InterruptedException in transaction");
40              }
```

The output from this change is shown in Figure 5-2.

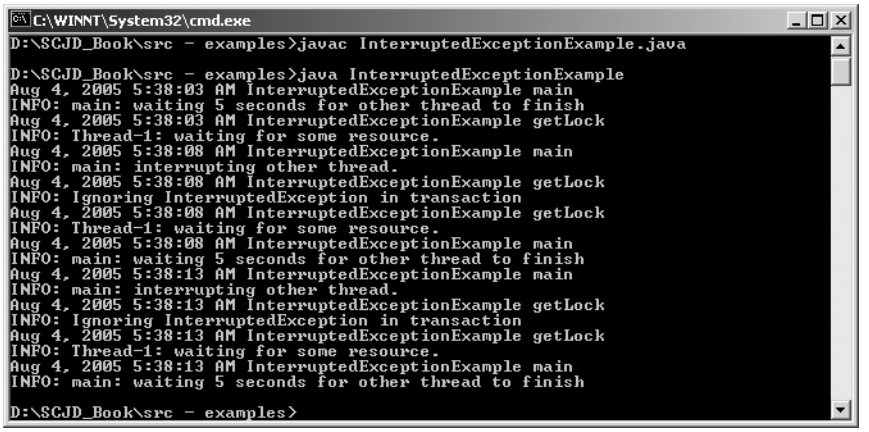


Figure 5-2. *Extremely simplistic logging*

However, this does not tell us much about what has happened but only that we have decided to ignore the exception. In this particular case, it is fairly easy to see which line caught that particular exception, and what it did with it. But what if we were running on a server, and the `getLock` method could have been called from any one of a number of different methods? In such a case, a stack trace might be useful to see why our `getLock` method was called. So we could use more sophisticated logging, as shown in the following code and in Figure 5-3:

```
38              } catch (InterruptedException ie) {
39a                 log.log(Level.WARNING,
39b                         "Ignoring InterruptedException in transaction",
39c                         ie);
40              }
```

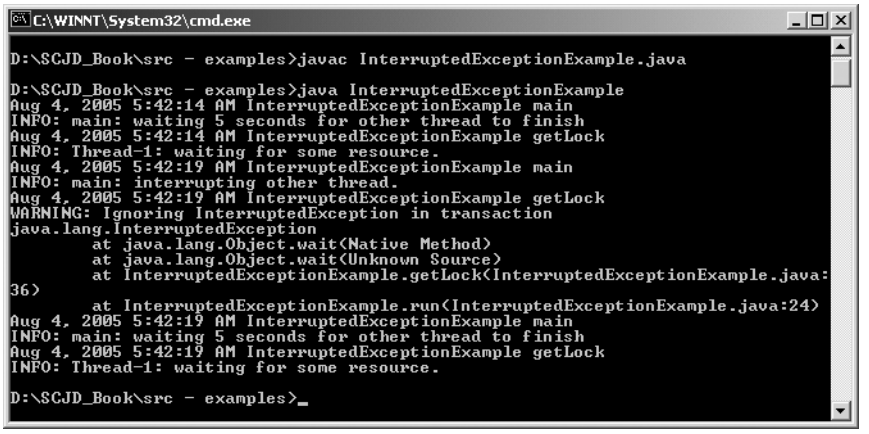


Figure 5-3. *More detailed logging*

If you decided that your method will ignore interruptions, then it would be wise to mention this in your Javadoc comments so that users are aware of this—and don't phone you in the middle of the night asking why they can't interrupt a thread they created.

Wrapping the Exception Within an Allowed Exception

In most cases exceptions cannot be as easily ignored as in the last example. If an `IOException` occurs while writing to file, you cannot just ignore it or you will end up with corrupted data. We created a fictitious business rule that allowed us to ignore the `InterruptedException` in the last example, but if that business rule did not exist, then clients may have a right to expect that they can interrupt their own threads.

One way to handle this is to wrap the caught exception within the exception we are allowed to throw, as shown in the next bit of code. The output from running this is shown in Figure 5-4.

```
38              } catch (InterruptedException ie) {
39a                 throw new LockAttemptFailedException(
39b                         "InterruptedException in getLock",
39c                         ie);
40              }
```

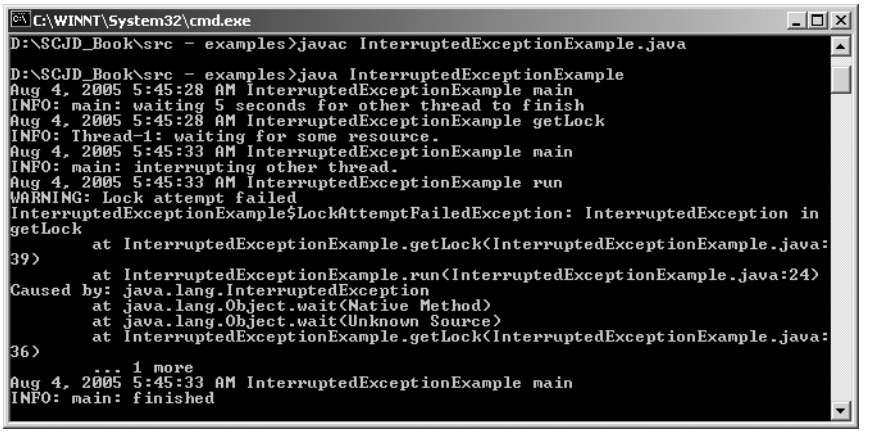


Figure 5-4. *Wrapping the exception within an allowed exception*

As can be seen in Figure 5-4, we received a warning message, telling us that we had a `LockAttemptFailedException` with the expected stack trace. This log message came from our `LockAttemptFailedException` handler at line 26. More importantly, though, the log also contains lines starting with `Caused by:`. If you look at these lines, you will see that they are the same lines that were output in Figure 5-3. By wrapping the `InterruptedException` inside the `LockAttemptFailedException`, we ensured that we still have the complete stack trace from the `InterruptedException`.

■ **Caution** You need to consider carefully whether it will make sense to wrap an exception in this way. If the only exception you are allowed to throw is a `RecordNotFoundException`, you might consider that it is reasonable to wrap an `EOFException` within your `RecordNotFoundException`—if you got to the end of the file without finding the record, then it *might* be a reasonable assumption. However, it might also be an invalid assumption—if you received an `EOFException` halfway through reading a record, then your file may be corrupted, and implying that the record cannot be found might be misleading at best, or cause further problems and corruption. Similarly, it would probably not make sense to wrap an `InterruptedException` within a `RecordNotFoundException`—if you were waiting on a lock for the record, then presumably the record does exist.

Wrapping the Exception Within a RuntimeException

There are cases where wrapping the caught exception within a declared exception does not make sense, as discussed in the previous note. However, we may not be able to add a new checked exception to the method signature since this may stop other programs from working. This is especially true when writing a class that implements an interface—another programmer could write their program to use your class based on the published interface, and have their program fail in unexpected ways if you change the interface—or they may find that recompiling their code no longer works. Either way, you are probably going to become unpopular very quickly.

`RuntimeException`, and subclasses of `RuntimeException`, do not need to be declared in method signatures, nor do they need to be caught. So it is possible to wrap the caught exception within a `RuntimeException` as shown in the following code:

```
38              } catch (InterruptedException ie) {
39a                 throw new RuntimeException (
39b                         "InterruptedException in getLock",
39c                         ie);
40              }
```

Unfortunately, if you do this it becomes difficult for the user of your class to catch the exception and handle it properly. Consider the following code, which demonstrates a poor way to handle it:

```
22      public void run() {
23          try {
24              getLock();
25a         } catch (RuntimeException re) {
25b             log.log(Level.WARNING, "Caught the interrupt", re);
25c         } catch (LockAttemptFailedExceptiondle) {
26              log.log(Level.WARNING, "Lock attempt failed", dle);
27          }
28      }
```

If we try and catch RuntimeException, we risk catching a whole lot of subclasses of RuntimeException that we really probably don't want to handle in this exception block, where we're only trying to handle the "interrupted" exception. For example, let's assume that somewhere in the getLock method, something throws a NullPointerException. NullPointerException is a subclass of RuntimeException, so it will be caught in line 25a. However, we do not handle NullPointerException, so it will not be handled appropriately. To ensure that we only handle the one RuntimeException that we are really interested in, we have to look at the cause of the RuntimeException, and if it is not the RuntimeException we are interested in, we should rethrow the exception.

```
22      public void run() {
23          try {
24              getLock();
25a         } catch (RuntimeException re) {
25b             if (re.getCause() instanceof InterruptedException) {
25c                 log.log(Level.WARNING, "Caught the interrupt", re);
25d             } else {
25e                 throw re;
25f             }
25g         } catch (LockAttemptFailedException dle) {
26              log.log(Level.WARNING, "Lock attempt failed", dle);
27          }
28      }
```

If we make this change and run the program, we will see the now familiar output as shown in Figure 5-5.

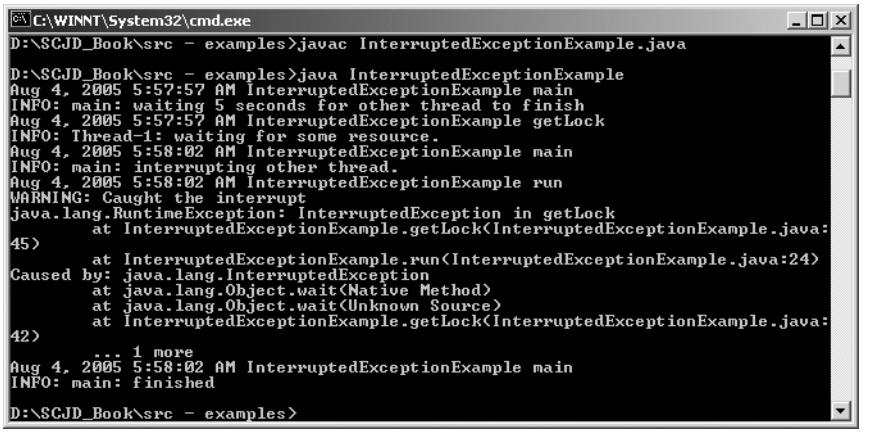


Figure 5-5. *Wrapping the exception within a RuntimeException*

Wrapping the Exception Within a Subclass of RuntimeException

We needed to add five lines of code to handle the RuntimeException, instead of the normal two lines we would have added for a checked exception, significantly adding to our code complexity.

However, as stated previously, subclasses of RuntimeException do not need to be declared or caught. So a better solution is to create a new exception that is a subclass of RuntimeException, similar to this:

```
public class UserInterruptionException extends RuntimeException {
    public UserInterruptionException(String msg, Throwable t) {
        super(msg, t);
    }
}
```

We can then use this in exactly the same way that we had used the RuntimeException in our previous example, namely

```
38              } catch (InterruptedException ie) {
39a                 throw new UserInterruptionException(
39b                         "InterruptedException in getLock",
39c                         ie);
40              }
```

However, our code for catching the exception returns to being nice and simple:

```
22      public void run() {
23          try {
24              getLock();
25a         } catch (UserInterruptionException uie) {
25b             log.log(Level.WARNING, "Caught the interrupt", uie);
25c         } catch (LockAttemptFailedException dle) {
26              log.log(Level.WARNING, "Lock attempt failed", dle);
27          }
28      }
```

■ **Tip** If you are creating a subclass of RuntimeException that you do not expect to be caught, then it is considered standard programming practice not to declare it in the method signature. However, if you are trying to work around a limitation of a provided interface, then you might want to declare it in your method signature and in your Javadoc documentation. Many common IDEs will show the exceptions a method will throw when it is entered, and some will even create standard catch blocks based on method signatures. In such cases, listing the subclass of RuntimeException will help your users.

If you do list the RuntimeException in the method signature and/or the Javadoc, it would be wise to add a code comment stating why you did this for the benefit of the person who is maintaining your code. In the specific case of the Sun assignment, you might also want to consider putting a comment in the Javadoc itself stating why you did it.

The big downside of using RuntimeExceptions and its subclasses is that they do not need to be caught. If a programmer left out the additions to lines 25a–c, the program will still run fine, but a RuntimeException will propagate up the stack to the top of the thread. This is shown in Figure 5-6.

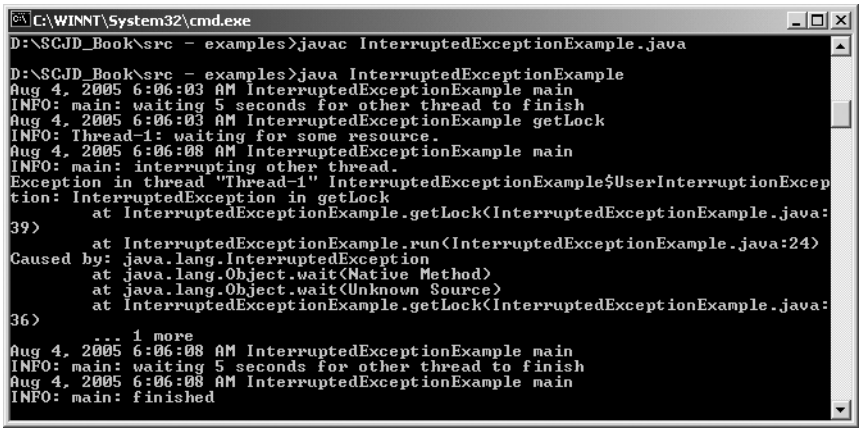


Figure 5-6. *RuntimeException propagating to the top of the thread stack*

■ **Caution** It is important to realize that the `RuntimeException` propagates to the top of the stack *for the thread it is running in, not for the* ***entire*** *JVM*. As can be seen in Figure 5-6, even after the `RuntimeException` has been thrown, and Thread-1 has died, the main thread is still operational. This has important ramifications when developing your server code—if you throw a `RuntimeException` and don't catch it, it is possible that your server will still be running while being unable to process any requests.

The DvdDatabase Class: A Façade

Before building any class, it is worthwhile considering what it is that the class does. The same applies to methods—for each method, try to determine just what the method does. If you find yourself using the word "and" when describing a class or method, there is the *possibility* that the class or method is trying to be responsible for more than it should. You might then consider whether it makes sense to break a class or method into two or more classes or methods; it might make your code a bit more manageable and maintainable.

In the case of our `DvdDatabase` class, we have been told in our instructions that this is the public class that all other classes will use if they want to access the data file. However, when we look at what the `DvdDatabase` class provides, we find that there are two separate functions:

1. Physically accessing the data
2. Providing logical record locking

If we tried to describe what this class does in a short sentence, we would probably have to use the word "and": "This class provides physical access to the data **and** provides logical record locking."

We *could* provide both these functions in one class, but if we split them out, then the code will be more maintainable later. If you need to work on a method that physically accesses the database, you will be able to go to a class that only deals with accessing the database; you will not have to wade through all the logical locking code.

There is a design pattern that describes what we are trying to achieve here: the Façade pattern. The English meaning of the word "façade" is the front face of something, typically a building. So we might refer to the front of a shop or building as its façade: the view of the building that the average user gets to see. In the same way, we can think of the `DvdDatabase` as the front face, or façade, shown to external users, hiding the classes that external users may not need to know about.

■ **Note** Although using a façade does not stop us from using the word "and" in our description of what the class does, it does change the class itself so that the faced class is not providing all the functionality; it is handing off to the other classes behind the scene.

Our `DvdDatabase` class is therefore very simple. We start by creating references to the classes that do the real work:

```
package sampleproject.db;

import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Collection;
import java.util.List;
import java.util.regex.PatternSyntaxException;

public class DvdDatabase implements DBClient {
    private static ReservationsManager reservationsManager
            = new ReservationsManager();

    private static DVDFileAccess database = null;

    public DvdDatabase() throws FileNotFoundException, IOException {
        this(System.getProperty("user.dir"));
    }

    public DvdDatabase(String dbPath) throws FileNotFoundException, IOException {
        database = new DvdFileAccess(dbPath);
    }
```

We now have to define our constructors for `DvdDatabase`. Since the instructions for our sample project do not specify how the `DvdDatabase` constructor should appear, we have some flexibility. This prompts two primary concerns:

1. What parameters should we use for the constructor?

2. What exceptions, if any, should be thrown from the constructors?

Regarding the parameters, we have to consider how this class will be used. We know that it might be used in a stand-alone application, and for that purpose it might be handy not to

provide any parameters at all, and assume that the data file is in the current working directory. However, we also know that this class will be used in a server environment, and in such cases, it is common for the data file to reside in a different directory (and sometimes a different hard drive) than the application. In this case, we would need to be able to specify the directory where the data file can be found.

The first constructor is just a special case of the second constructor, and as such we might be tempted to leave it out. We should make a decision on whether it will be used often. If so, adding the constructor will make our users happy. If not, we can leave the constructor out, thereby simplifying our code, and in the rare cases where a user wants to use a database in the current working directory they can call the constructor that takes a directory as a parameter with the current working directory (`System.getProperties("user.dir")`) as the parameter.

We have decided to leave both constructors in the class, primarily to demonstrate the technique of having one constructor call the other constructor. This is a very common way of handling overloaded methods and constructors, as it ensures that the same business logic is executed no matter which version gets called.

Our constructors are going to open the data file, which means we could get a `FileNotFoundException` if the file is not in the specified directory, and we could get an `IOException` if there is a problem with opening the data file (for example, if we don't have adequate permissions). We're faced with the decision of handling them within our constructor, or passing them back to the calling class.

Another way of thinking about this issue is, what can we realistically do if we get either of those exceptions? About the only thing we can do within our `DvdDatabase` class is try the operation again, but if the file does not exist when we first look for it, is it likely to be there the second time we look for it? We cannot simply log these exceptions and ignore them; then the user will think that the `DvdDatabase` class was instantiated correctly and is ready for use when it isn't. So we have chosen to pass these exceptions back to the class that is constructing the `DvdDatabase`.

■ **Tip** To keep this project simple, we have elected to rethrow the `FileNotFoundException` and `IOException`, but in doing this we have implicitly declared that we are dealing with a file-based data store. A future enhancement might be to convert this backing store to a SQL database, in which case these exceptions would no longer apply. A better solution might be to create our own `DatabaseFailureException`, and wrap `FileNotFoundException` and `IOException` in it where necessary. This way, if we later change to a SQL database, we could similarly wrap the SQL exceptions within the same `DatabaseFailureException`, and the client code should still continue to work. Doing this is left as an exercise for the reader.

For each method in our `DBClient` interface, we create a method that calls the appropriate method in our worker classes:

```
    public boolean addDVD(DVD dvd) throws IOException {
        return database.addDVD(dvd);
    }

    public DVD getDVD(String upc) throws IOException {
        return database.getDVD(upc);
    }

    public boolean removeDVD(String upc) throws IOException {
        return database.removeDVD(upc);
    }

    public boolean modifyDVD(DVD dvd) throws IOException {
        return database.modifyDVD(dvd);
    }

    public List<DVD> getDVDs() throws IOException {
        return database.getDVDs();
    }

    public Collection<DVD> find(String query)
            throws IOException, PatternSyntaxException {
        return database.find(query);
    }

    public boolean reserveDVD(String upc) throws InterruptedException {
        return reservationsManager.reserveDVD(upc, this);
    }

    public void releaseDVD(String upc) {
        reservationsManager.releaseDVD(upc, this);
    }
}
```

Accessing the Data: The DvdFileAccess Class

We will present the `DvdFileAccess` class section by section, rather than trying to present all the code at once. Once again, the code is available online at the Apress web site in the Source Code section.

Since there is only one physical file on disk, it is tempting to consider making the `DvdFileAccess` class a singleton—coding the class in such a way that only one instance of `DvdFileAccess` can exist at any given time. However, a lot of work can be performed in

parallel if multiple clients are working on a multiple-CPU system, for example, converting between a DVD value object and the bytes on file, or searching through the data file. In addition, if we were to make the DvdFileAccess class a singleton, any class that uses the DvdFileAccess class would have to be coded differently than if it is a standard class—if we were to later decide that this same class can be used to process multiple data files (with some simple modifications), we would have to modify all the classes that use DvdFileAccess. Therefore, this class is not a singleton.

As mentioned earlier, DvdDatabase is the façade through which all other classes should access the data. Therefore, no other classes should call DvdFileAccess directly. To ensure this, default access is set on the class itself—only classes within the sampleproject.db package can access this class—as shown in line 31. As mentioned earlier, line numbers are not contiguous, as source code comments have been removed.

```
 1  package sampleproject.db;
 2
 3  import java.io.*;
 4  import java.util.*;
 5  import java.util.concurrent.locks.*;
 6  import java.util.logging.*;
 7  import java.util.regex.*;
..
31  class DvdFileAccess {
..
35      private static final String DATABASE_NAME = "dvd_db.dvd";
..
41      private Logger log = Logger.getLogger("sampleproject.db");
..
46      private static RandomAccessFile database = null;
..
51      private static Map<String, Long> recordNumbers
52              = new HashMap<String, Long>();
```

While most of the fields listed use the standard format from all previous versions of the JDK, the recordNumbers collection uses the new generics declarations so that the compiler can check that we are using the *generic* collection in a type-safe manner. This almost removes the risk of us getting a ClassCastException at runtime. We will discuss the use of this particular variable in the section following the class constructors.

```
58      private static ReadWriteLock recordNumbersLock
59              = new ReentrantReadWriteLock();
..
66      private static String emptyRecordString = null;
..
71      private static String dbPath = null;
..
77      static {
78          emptyRecordString = new String(new byte[DVD.RECORD_LENGTH]);
79      }
```

When writing a record to file, we could write each field separately, filling with spaces if required, or we could write the entire record in one operation.

Since a disk drive is constantly spinning when doing operations, writing field by field would be considerably slower than writing the entire record at once. This is because the disk would have continued spinning between each call to write the field, and there would be a small delay while the disk heads return to the correct location for writing (such an operation would be handled by the drive controller, but it would still slow down this operation). Although the delays caused by writing individual fields to a file would be small, they do exist, and they would be a bottleneck in a multiuser environment since only one user can ever be writing to the disk at any given time. We have, therefore, decided to write the entire record at once.

In the section describing the `persistDvd` method, we will show one method of building a record before writing it to disk. For speed and simplicity, we have decided to build a record by starting with a `StringBuilder` of known length, and replacing the bytes within it with the contents of the DVD fields. Since we want the `StringBuilder` to be a known length and contain all nulls before we start inserting our field data, we have created an `emptyRecordString` that can be used in the constructor of our `StringBuilder` to quickly create a known starting point. Since the `emptyRecordString` is a static field, it is constructed in the static initializer shown earlier.

```
90      public DvdFileAccess(String suppliedDbPath)
91              throws FileNotFoundException, IOException {
92          log.entering("DvdFileAccess", "getDvdFileAccess", suppliedDbPath);
93          if (dbPath == null) {
94              database = new RandomAccessFile(
95                      suppliedDbPath + File.separator + DATABASE_NAME, "rw");
96              getDvdList(true);
97              dbPath = suppliedDbPath;
98              log.fine("database opened and file location table populated");
99          } else if (dbPath != suppliedDbPath) {
100             log.warning("Only one database location can be specified. "
101                         + "Current location: " + dbPath + " "
102                         + "Ignoring specified path: " + suppliedDbPath);
103         }
104         log.exiting("DvdFileAccess", "DvdFileAccess");
105     }
```

Although the `DvdFileAccess` class is not a singleton, it does not make sense to rerun the constructor code each time the constructor is called. Setting the `database` field is one example of code we do not want run multiple times—the first time the constructor is called, the database field will be set to the `RandomAccessFile` for our data file, and after that there is no need or desire to reset it. As shown in line 93, we check whether the database path has already been configured, and if so, we do not perform initialization logic a second time. However, there is a risk that somebody may call this constructor multiple times with different paths; if they do this, a warning message will be logged stating that the newer path is being ignored.

We as developers must always be aware of how well our code performs. While it does not make sense to agonize over every line of code trying to make it perform better, we should attempt to spot common areas where code may perform badly.

■ **Caution** If it has not already happened, then one day you may be asked to improve the performance of a class. While it can be useful to read through the code manually, looking for known problem areas, it is always recommended that you use a profiler—a program that will attach to your program while it is running, and tell you where your program is spending most of its time. When you know which methods take the most time to complete, you can look at improving their performance, rather than trying to fix random methods. The information provided here about improving the performance of our `DvdDatabase` class is designed to give us discussion points into several sections of the class and JDK 5. They are not necessarily the only (or even the best) places for us to concentrate on improving our programs.

Reading and writing from the disk is one of the slowest operations we have to deal with. We may also have multiple users all trying to access different records; with only one data file, they must queue up to access the file, effectively creating a bottleneck.

When reading or writing a record, we can speed up the operation if we can go straight to the location in the file where the record is stored. If our primary key was the record number, we would be able to calculate the file position on the fly based on the record number multiplied by the size of the record. However, our application is based on using the UPC string as the primary key, so we need a way to map UPCs to the location in the file. The only way to do this is to read the data file at least once, storing UPCs and file locations in the map as we go. We could create a method just to do this for us, but the interface we must implement already has a public method that we must implement that will read the entire file and return a `List` of the DVDs in that file, so we can piggyback that logic to populate our map.

However, there is a danger in doing this. Our constructor is relying on this method to populate a needed field. But it is possible for the `getDVDs` method to be overridden, which would result in our field not being populated. So we should either make this method `final` (which will stop any other class from overriding it), or call a private method that does the same function (private methods are similar to final methods, since they cannot be overridden, because no other class can even see them).

The `getDVDs` method is a business method. That is, we only created it because the business requirements stated that we must—we could easily write our client application without it. Likewise, although our implementation of it reads the entire database file from end to end, it would be possible to implement it differently, such that it does not access the file directly but rather calls other public methods (such as the `getDVD` method). Given this, it is possible that somebody might later decide to override our implementation, so it would not be appropriate for us to make this method final.

However, the `getDVDs` method is one of the methods that we must make public according to our provided interface.

Fortunately, we have a simple workaround for this dilemma: create a private method that does the work, and have the `getDVDs` method call it. That way, if somebody later overrides `getDVDs`, our private method will still exist in the background populating our required map.

This is still not quite a perfect solution, though. In using this method to populate our map, we will be creating the list for no purpose and discarding it immediately. Normally creating a collection and then destroying it without using it would be a bad thing, but in this case it is more than justified. It is only the constructor that wastes this `List`, and the alternative is to have a method that is almost identical but exists only to populate our map. Here is our

method that reads all the records from file, and along the way populates our map of UPC numbers to file locations, along with the getDVDs method that calls it:

```
113      public List<DVD> getDvds() throws IOException {
114          return getDvdList(false);
115      }
```

■ **Note** The getDvds method in the DvdFileAccess class uses a different naming convention from the getDVDs method specified in the DBClient interface. We are able to do this because DvdFileAccess is the class behind the façade—it does not implement DBClient. We have made a design decision to use the Sun code conventions for the method names in our nonpublic classes, even though this means that they do not match perfectly with the method names in the public classes and interfaces. There is a trade-off here—a programmer working with the DvdFileAccess class may appreciate working with class and method names that conform to one convention, but changing the code convention between the public class and the nonpublic class could also be slightly confusing for a junior programmer.

```
130      private List<DVD> getDvdList(boolean buildRecordNumbers)
131              throws IOException {
132          log.entering("DvdFileAccess", "getDvdList", buildRecordNumbers);
133          List<DVD> returnValue = new ArrayList<DVD>();
134
135              if (buildRecordNumbers) {
136                  recordNumbersLock.writeLock().lock();
137              }
138
139          try {
140              for (long locationInFile = 0;
141                      locationInFile < database.length();
142                      locationInFile += DVD.RECORD_LENGTH) {
143                  DVD dvd = retrieveDvd(locationInFile);
144                  log.fine("retrieving record at " + locationInFile);
145                  if (dvd == null) {
146                      log.fine("found deleted record ");
147                  } else {
148                      log.fine("found record " + dvd.getUPC());
149                      if (buildRecordNumbers) {
150                          recordNumbers.put(dvd.getUPC(), locationInFile);
151                      }
152                      returnValue.add(dvd);
153                  }
154              }
155          } finally {
156              if (buildRecordNumbers) {
157                  recordNumbersLock.writeLock().unlock();
```

```
158             }
159         }
160
161         log.exiting("DvdFileAccess", "getDvdList");
162         return returnValue;
163     }
```

As with the file itself, the map of UPC numbers to file locations is a single object used by many threads. However, in most cases, the threads will only be reading from the `recordNumbers` map—it will be much rarer for a method to update this map. Prior to JDK 5, we would have synchronized all access to the `recordNumbers` map, which would have meant that only one thread could ever access it at any given time. With JDK 5 we now have `ReadWriteLocks` that allow for greater concurrency. Instead of synchronizing code, we encapsulate the code inside calls to `lock` and `unlock` methods. Multiple threads can own a read lock on a single object at any given time, but only one thread can own a write lock at any given time.

If we are running this from the constructor, then we will be updating the `recordNumbers` map, so we will not want any other thread to be accessing the map in the meantime—a write lock will ensure this for us. This is set in line 136, and released in the finally block at line 157. It is important to ensure that the lock is released even if an exception is thrown; hence the call to unlock is in the finally block to ensure it is always run. This is recommended whenever you are using the new locking classes.

Line 150 adds our UPC string and the location in the file into the map. Using generics and autoboxing (introduced in Chapter 2) allows us to keep this code simple, while simultaneously ensuring that invalid data is not entered into our `recordNumbers` map. At the very start of the class we declared that the `recordNumbers` could only contain a `String` as the key and a `Long` as the value with the following code:

```
51      private static Map<String, Long> recordNumbers
52              = new HashMap<String, Long>();
```

The compiler will then use this to validate *at compile time* that we are storing `Strings` as the key and a `Long` as the value within this `Map`. Using autoboxing allows us to add a `String` and a *primitive* `long` to the `Map`, knowing that Java will automatically convert the *primitive* `long` to the wrapper `Long` class required for the `Map`.

Prior to JDK 5, there was no way for the compiler to validate that the type of data we were adding to a collection was the type of data we actually wanted to allow into the collection. You will now get an error message if you attempt to add the wrong type of data to our collection. If you would like to see an example of how this works, try changing line 150 as follows:

```
recordNumbers.put(dvd.getUPC(), (int) locationInFile);
```

The JDK 5 Java compiler will now produce an error message, because the type of data we are potentially adding to the `Map` no longer matches our declared allowable contents:

```
sampleproject\db\DvdFileAccess.java:150: put(java.lang.String,java.lang.Long)
in java.util.Map<java.lang.String,java.lang.Long> cannot be applied to
(java.lang.String,int)
                        recordNumbers.put(dvd.getUPC(), (int) locationInFile);
                                      ^
1 error
```

It is important to realize that this is a compile-time validation only—it is still possible to provide invalid data at runtime, resulting in a `ClassCastException`.

Line 143 calls a private method to read the DVD record based a location in a file. This method is also used by our public method that will read a DVD based on a UPC number. We will show the public `getDVD` method first:

```
172     public DVD getDvd(String upc) throws IOException {
173         log.entering("DvdFileAccess", "getDvd", upc);
174
175         recordNumbersLock.readLock().lock();
176         try {
177             // Determine where in the file this record should be.
178             // note: if this is null the record does not exist
179             Long locationInFile = recordNumbers.get(upc);
180             return (locationInFile != null) ? retrieveDvd(locationInFile)
181                                             : null;
182         } finally {
183             recordNumbersLock.readLock().unlock();
184             log.exiting("DvdFileAccess", "getDvd");
185         }
186     }
```

Line 175 requests a read lock on our mutex for the `recordNumbers` map. Remember that many threads can be operating simultaneously, so asking for a read lock allows the other threads to also gain read locks and work with the `recordNumbers` map.

If the UPC requested does not exist, null will be returned to the calling method, as shown in line 181. Otherwise, the `retrieveDvd` method will be called in line 180 and the results of that call will be returned to the calling method. If the `retrieveDvd` method throws an `IOException` or a `RuntimeException`, it is important to ensure that we do not leave the `recordNumbersLock` mutex locked, so we have put the call to `unlock` in the finally block at line 183. It is important to remember that the finally block is still executed, even though the `return` statement is at line 180.

```
196     private DVD retrieveDvd(long locationInFile) throws IOException {
197         log.entering("DvdFileAccess", "retrieveDvd", locationInFile);
198         final byte[] input = new byte[DVD.RECORD_LENGTH];
...
202         synchronized(database) {
203             database.seek(locationInFile);
204             database.readFully(input);
205         }
```

Multiple threads work with the `recordNumbers` map, but the majority of them will only be reading the map, and multiple reads can occur simultaneously without affecting the other threads. Therefore, access to the `recordNumbers` map is a perfect candidate for a `ReadWriteLock`.

However, in the case of reading from the data file, one thread could affect another thread if they were allowed to operate simultaneously. When reading from the data file, we need to perform two steps: move to the correct location in the file, then read the entire record. It is very important that these two operations behave as a single atomic operation; otherwise, if

the position in the file was changed by another thread between lines 203 and 204 we would end up reading from the wrong location in the file. Using a `ReadWriteLock` would be counter-productive in this case, as all operations would need a `WriteLock`, and the extra overhead of confirming that there are no outstanding `ReadLocks` would result in poorer performance. A much better solution is to use a standard synchronized block, as shown in lines 202 through 205.

Since a synchronized block will block any other thread from accessing the data file, we want the block to be as small as possible to reduce the blocking time. Therefore, the synchronized block only lasts until the record is read fully. It is important to note that the `read` method does not guarantee that an entire record will be read but only that *at least* one byte will be read. It is therefore important to ensure that an entire record is read, which we achieved by calling `readFully`.

Having read an entire record into an array of bytes, we can extract the various strings for each field from that array.

```
211          class RecordFieldReader {
212              int offset = 0;
213              String read(int length) throws UnsupportedEncodingException {
214                  String str = new String(input, offset, length, "UTF-8");
215                  offset += length;
216                  return str.trim();
217              }
218          }
219
220          RecordFieldReader readRecord = new RecordFieldReader();
221          String upc = readRecord.read(DVD.UPC_LENGTH);
222          String name = readRecord.read(DVD.NAME_LENGTH);
223          String composer = readRecord.read(DVD.COMPOSER_LENGTH);
224          String director = readRecord.read(DVD.DIRECTOR_LENGTH);
225          String leadActor = readRecord.read(DVD.LEAD_ACTOR_LENGTH);
226          String supportingActor = readRecord.read(DVD.SUPPORTING_ACTOR_LENGTH);
227          String year = readRecord.read(DVD.YEAR_LENGTH);
228          int copy = Integer.parseInt(readRecord.read(DVD.COPIES_LENGTH));
229
230          DVD returnValue = ("DELETED".equals(upc))
231                          ? null
232                          : new DVD(upc, name, composer, director, leadActor,
233                                    supportingActor, year, copy);
234
235          log.exiting("DvdFileAccess", "retrieveDvd", returnValue);
236          return returnValue;
237      }
```

Finally, if the record is not marked as being deleted, we create a new DVD value object and return it. While creating the value object, we remove any trailing spaces or nulls from the field.

Once we have populated the `recordNumbers` map, it rarely needs to be changed. The only time we need to modify it is when a record is added or deleted. The `addDVD` method calls a private `persistDVD` method, which is also used when we modify a DVD record.

```
246     public boolean addDvd(DVD dvd) throws IOException {
247         return persistDvd(dvd, true);
248     }
```

We start the `persistDVD` method by seeing if the provided UPC is in our map of known UPCs, and verifying whether we are creating or modifying the DVD record. If we are creating a record the UPC must not be in our map. If we are modifying the DVD, then the UPC must be in our map. In any other case we cannot proceed, so we return false to indicate that the operation failed. Since we may be potentially adding the DVD's UPC to our map of known UPCs, we need to obtain a write lock on the `recordNumbersLock`.

■ **Note** Normally it is a bad idea to use return values to indicate success or failure of a call to a method—we should be able to throw an exception if we cannot continue. An exception would provide greater detail of what went wrong, and more importantly what the stack trace was at the time. However, the `DBClient` interface has specified that the `addDVD`, `modifyDVD`, and `removeDVD` methods must return a Boolean to indicate success or failure, so we must follow the dictates of the interface or we risk causing other programmer's code to fail. As with the assignment you get from Sun, we have simulated an interface where the reasons for the methods, parameters, return values, and exceptions have not been specified.

Leaving this method without releasing our write lock could be disastrous: no other thread would ever be able to gain a read or a write lock on `recordNumbersLock`, effectively rendering most of our methods inoperable. To guard against this, immediately after gaining the write lock we enter a try … finally block, and release the lock in the finally clause at line 357.

```
338     private boolean persistDvd(DVD dvd, boolean create) throws IOException {
339         log.entering("DvdFileAccess", "persistDvd", dvd);
340
341         // Perform as many operations as we can outside of the synchronized
342         // block to improve concurrent operations.
343         Long offset = 0L;
344         recordNumbersLock.writeLock().lock();
345         try {
346             offset = recordNumbers.get(dvd.getUPC());
347             if (create == true && offset == null) {
348                 log.info("creating record " + dvd.getUPC());
349                 offset = database.length();
350                 recordNumbers.put(dvd.getUPC(), offset);
351             } else if (create == false && offset != null ) {
352                 log.info("updating existing record " + dvd.getUPC());
353             } else {
354                 return false;
355             }
356         } finally {
357             recordNumbersLock.writeLock().unlock();
358         }
```

The persistDVD method has a private StringBuilder variable that we will use to create a representation of a complete record. Prior to JDK 5, we might have used a StringBuffer to build this; however, the StringBuffer is internally synchronized to make it thread safe. Since the variable we will be using is a method variable, not a class variable, we do not need the synchronization. The StringBuilder class will therefore provide us with better performance.

Since each record is a fixed length, it is easy to start with a blank record of the correct length, then replace the blanks with the field data. We have a static empty field that was declared in line 66 and initialized in lines 77–79 as described earlier in the chapter. This makes it easy for us to create a StringBuilder of the correct length and contents in line 360:

```
360         final StringBuilder out = new StringBuilder(emptyRecordString);
```

Having created our variable, we will use StringBuilder's replace method to put the field data in the correct locations within the record field. By making a utility inner class we can save replicating the code:

```
362         class RecordFieldWriter {
363             int currentPosition = 0;
364             void write(String data, int length) {
365                 out.replace(currentPosition,
366                             currentPosition + data.length(),
367                             data);
368                 currentPosition += length;
369             }
370         }
371         RecordFieldWriter writeRecord = new RecordFieldWriter();
```

We can then use our utility inner class to convert the DVD record to the StringBuilder equivalent:

```
373         writeRecord.write(dvd.getUPC(), DVD.UPC_LENGTH);
374         writeRecord.write(dvd.getName(), DVD.NAME_LENGTH);
375         writeRecord.write(dvd.getComposer(), DVD.COMPOSER_LENGTH);
376         writeRecord.write(dvd.getDirector(), DVD.DIRECTOR_LENGTH);
377         writeRecord.write(dvd.getLeadActor(), DVD.LEAD_ACTOR_LENGTH);
378         writeRecord.write(dvd.getSupportingActor(),
379                           DVD.SUPPORTING_ACTOR_LENGTH);
380         writeRecord.write(dvd.getYear(), DVD.YEAR_LENGTH);
381         writeRecord.write("" + dvd.getCopy(), DVD.COPIES_LENGTH);
```

■ **Caution** When working on your Sun assignment you must make the design decisions that make sense to you and that you are willing to defend when you go to do the exam portion of the certification. This can (and in some cases should) mean that you may make design decisions that contradict our design decisions—we are, after all, working on different assignments with different requirements. Do not be afraid to consider other options. A good place to discuss one of our design decisions or one of your design decisions is JavaRanch (http://www.javaranch.com).

Finally we write the record to file, and return true to show that the record was persisted to file.

```
384          // now that we have everything ready to go, we can go into our
385          // synchronized block & perform our operations as quickly as possible
386          // ensuring that we block other users for as little time as possible.
387
388          synchronized(database) {
389              database.seek(offset);
390              database.write(out.toString().getBytes());
391          }
392
393          log.exiting("DvdFileAccess", "persistDvd", persisted);
394          return true;
395      }
```

Most of the remaining methods do not need explaining. However, we will end with the `find` method, which shows the enhanced for loop, as well as using the regular expressions classes introduced in JDK 1.4.

```
311      public Collection<DVD> find(String query)
312              throws IOException, PatternSyntaxException {
313          log.entering("DvdFileAccess", "find", query);
314          Collection<DVD> returnValue = new ArrayList<DVD>();
315          Pattern p = Pattern.compile(query);
316
317          for (DVD dvd : getDvds()) {
318              Matcher m = p.matcher(dvd.toString());
319              if (m.find()) {
320                  returnValue.add(dvd);
321              }
322          }
323
324          log.exiting("DvdFileAccess", "find", returnValue);
325          return returnValue;
326      }
```

In line 315 we take the string provided from our GUI application and compile it into a Java `Pattern`. We then need another class, the `Matcher` class, which can attempt to match the pattern against the provided string in various ways; we generate the `Matcher` for each DVD read in line 318. Finally, in line 319 we tell the `Matcher` to find the next occurrence of the pattern in the current DVD's string representation; if found we add the DVD to the collection of DVDs to be returned. While we are only interested in finding the pattern as a subset of the entire DVD, the `Matcher` can perform other types of matching as well; for instance, it can compare two strings in their entirety, or match starting with the beginning of the string.

Line 317 shows the enhanced for loop syntax in action. Using this form saved us the drudgery of manually creating our own iterator, and using generics saves us from casting objects. Contrast the simple use of line 308 with the code we would have had to use if the for loop had not been enhanced:

```
317a        List<DVD> dvds = getDVDs();
317b        for (Iterator i = dvds.iterator(); i.hasNext(); ) {
317c            DVD dvd =   i.next();
318             Matcher m = p.matcher(dvd.toString());
319             if (m.find()) {
320                 returnValue.add(dvd);
321             }
322         }
```

Discussion Point: Caching Records

So far we have made minor concessions to speeding up the data access code, but we still have the situation where each DVD record is read from the physical file on disk—possibly the slowest operation of all. We have deliberately done this so that we have not overcomplicated this chapter; however, it is worthwhile considering whether or not we can cache the data.

The first question we should address is whether caching the data will save us any disk operations. If most operations on the data were writing to disk, then caching would not make much sense, since we would have to update the cache and update the file for the majority of operations. In fact, we might even end up with poorer performance than if we did not cache the data at all. However, the application we are developing is likely to have a large number of searches for matching records, and displaying of records, before one record is chosen for updating. Therefore, the majority of operations could benefit from having the records cached.

The next issue is whether the memory requirements would preclude caching. Each DVD record is relatively small, and it is likely that caching 1,000 DVD records would require far less than 10 MB of RAM. Because most new computers being sold have at least 128 MB of RAM, we should not have a problem caching our data.

Finally, we should take into account the impact this change will have on our code. If adding any feature makes the code significantly harder to read, then we should consider whether it is worth adding—certainly whoever maintains our code will not thank us if they have to maintain unreadable code. However, adding a cache to the existing code is reasonably simple; all we need is an extra map to hold the UPC keys and DVD records. Access to this cache can be handled in the same way as we use the `recordNumbersLock` mutex for the UPC-to-file-location map.

You should read the instructions you received from Sun carefully before deciding whether or not to implement a cache. Check whether there are any performance requirements, or any requirements for the simplest possible code. If there are no specific requirements, you can make your own decision as to whether to implement a cache—just remember to document your decision in your design choices document.

The ReservationsManager Class

The `ReservationsManager` class provides the ability to logically reserve a record so that a client can modify it, secure in the knowledge that no other client can modify it until the logical reservation is released.

To help explain the purpose of logical record locking, let's first consider a scenario where there is no logical record locking, and two clients try to rent the only copy of a particular DVD:

- Client A retrieves the DVD record for the movie *Office Space*.
- Client B retrieves the DVD record for *Office Space*.
- Client A verifies that there is one copy of the DVD still in stock.
- Client B verifies that there is one copy of the DVD still in stock.
- Client A rents the DVD.
- Client B rents the DVD.

We now have a problem; according to the electronic records, both clients A and B have rented the same DVD.

One way to solve this problem would be to make the retrieval-verification-rental operations atomic. However, this could only be done on the server side since two separate clients working in their own JVMs would be unaware of any synchronized blocks operating in other JVMs. Having this code operate on the server side makes building a thin client very simple, but we already know that we have to build a Swing client, so we know that the client computers can support thick clients. A bigger problem is that having this code within an atomic block will reduce concurrency.

THIN AND THICK CLIENTS

The term "thin client" is used to denote a client interface that can run on a computer with minimal processing power. A typical example is a web interface to an application—the computer accessing the web interface can be a very low-powered computer. It is even possible to set up a computer that has no hard drive or floppy disk drive that can access powerful software as long as there is a thin client interface for the software. It would be possible, for example, to set up a web browser on a personal computer with a 386 CPU running at 16 MHz with 4 MB of RAM. If you did this, you would still be able to access your e-mail, read news, and perform web searches, among other things.

For those who are not as old as the authors, before the current personal computers with Pentium 4 CPUs running in excess of 2 GHz with a minimum of 128 MB of RAM, there were personal computers with Pentium III CPUs. Going further back in the timeline were Pentium II CPUs, Pentium CPUs, 486 CPUs—and before that were 386 CPUs. Before the personal computers with 386 CPUs were 286, 8086, and 8088 CPUs, but the authors wouldn't want to set up a web browser on one of them (the first Unix-like system one of the authors worked on was a Coherent system on a 286).

A "thick client" (or a "fat client") is one where a large proportion of the processing is done on the client computer, and therefore a more powerful client computer is required. For example, Microsoft recommends running Microsoft Office on a PC with at least a Pentium III CPU, and 128 MB of RAM.

There is a trend toward using thin client software where possible within organizations for many reasons, such as the fact that thin clients can typically still run on older computers (the 386 was released in 1985, while the Pentium III was released in 1999; there are 14 years' worth of computers that would be obsolete if all software required a thick client running on a Pentium III), plus system administrators have an easier job if they only have to administer one or two servers running the applications, and all the client machines only run thin client software.

■ **Caution** Before deciding whether to build a thin or a thick client for your submission, carefully read the instructions you downloaded from the Sun site to determine whether there are any requirements you must adhere to. There have been many discussions about this on JavaRanch (`http://www.javaranch.com`) regarding this topic (one, started by one of the authors, had 133 views put forward), and you are welcome to join in the discussions there.

But consider the same scenario with logical record locking:

- Client A logically locks the DVD record for *Office Space*.
- Client B attempts to logically lock the DVD record for *Office Space*; however, it cannot do so until client A releases the logical lock. Client B must now wait for client A to finish.
- Client A retrieves the DVD record for *Office Space*.
- Client A verifies that there is one copy of the DVD still in stock.
- Client A rents the DVD.
- Client A releases the logical lock for the DVD record for *Office Space*.
- Client B logically locks the DVD record for *Office Space*.
- Client B retrieves the DVD record for *Office Space*.
- Client B finds that there are no more copies of the DVD in stock.
- Client B releases the logical lock on *Office Space* and (hopefully) goes off to find some other DVD to rent.

One of the major advantages of logical record locking is that it allows a greater concurrency. It can also work across a network, so we can use the power of our thick clients.

Discussion Point: Identifying the Owner of the Lock

It is rarely good enough to simply have a way of logically reserving a record and releasing the reservation. You typically need some way of identifying which client has reserved the record, so that only they can release it.

An example from real life might help explain why we need this. Imagine if you rang the theater and reserved the last seat for a show. But then some other person turned up at the theater and claimed that they had reserved the seat. If the theater does not have some way of identifying who reserved the seat, you could lose your seat.

Likewise, if we don't identify the owner of a lock, an unscrupulous programmer could write a program in such a way that if they have not received a lock within a certain amount of time, they will just unlock the record anyway.

How you identify your lock owner depends on what your instructions state, and possibly on how you develop your network server. We will offer some suggestions in the following sections, but you will have to determine for yourself what will work for your specific instructions.

Using a Token to Identify the Lock Owner

If we can return a token from the lock method to the client requesting the logical lock, then we can mandate that the client must use that token when performing other operations, including when they release the logical lock.

The token itself (also referred to as a "cookie" or a "magic cookie") is something that our class would use to identify the owner of the logical lock. As such, it does not have to be a meaningful object, since the client does not need to do anything with this token other than store it for use when calling other methods. In fact, it is probably better that this token be random, since if the token has some meaning then the unscrupulous programmer might be able to guess it and still unlock our records.

However, our interface requires that our lock method return a Boolean and the `releaseDVD` method does not allow us to insert a token either, so this is unfortunately not an option for us.

Using the Thread to Identify the Lock Owner

If we opt to use sockets for our network interface, we will have total control over the threads used by each client. We will create a new thread when the client connects to the server, so we can use the thread identifier as a proxy for the client identifier.

When doing this, we would simply store the thread identification with the record identifier at the time we logically lock the record. Then, whenever the client attempts to do anything that needs the lock, we can compare the current thread identifier with the stored value; if they match, then we allow the operation.

This saves the client worrying about storing and reusing a token, but it will only work for our stand-alone client and for a network solution based on sockets. If our network solution uses Remote Method Invocation (RMI), this won't work.

Using a Class Instance to Identify the Lock Owner

The RMI specification states that there are no guarantees about which threads will be used for any given remote method. This means that the following scenario is possible according to the specification:

- Client A uses thread 1 to logically lock a record.
- Client B uses thread 1 to attempt to logically lock the same record; it waits for the logical lock to be released.
- Client A uses thread 2 to logically release the record.
- Client B uses thread 1 to logically release the record.

More complex scenarios are also possible. Listing 5-1 shows an example of this problem.

Listing 5-1. *An Example of Thread Reuse in RMI*

```
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;

interface ServerReference extends Remote {
    public void serverThreadNumber(String id) throws RemoteException;
}

class Server extends UnicastRemoteObject implements ServerReference {
    public Server() throws RemoteException {
        // do nothing constructor
    }

    public void serverThreadNumber(String id) throws RemoteException {
        System.out.println(id + " running in thread "
                           + Thread.currentThread().hashCode());
        try {
            Thread.sleep(2000);
        } catch (InterruptedException ie) {
            ie.printStackTrace();
            System.exit(1);
        }
    }
}

public class RmiProblem extends Thread {
    public RmiProblem(String id) {
        super(id);
    }

    public static void main(String[] args) throws Exception {
        LocateRegistry.createRegistry(1099);
        Naming.rebind("rmi://localhost:1099/RmiProblem", new Server());

        Thread a = new RmiProblem("A");
        a.start();

        Thread.sleep(1000);

        Thread b = new RmiProblem("B");
        b.start();

        a.join();
        b.join();
```

```
        System.exit(0);
    }

    public void run() {
        try {
            ServerReference remoteCode =
                    (ServerReference) Naming.lookup("RmiProblem");

            for (int i = 0; i < 5; i++) {
                remoteCode.serverThreadNumber(getName());
                Thread.sleep(2000);
            }
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(0);
        }
    }
}
```

Don't worry if you do not understand this code completely. Chapter 6 discusses RMI in depth, so it may be easier to come back to this code after reading that chapter.

Although the results of running this code will vary from computer to computer (and indeed from run to run), one example of running the code is shown in Figure 5-7.

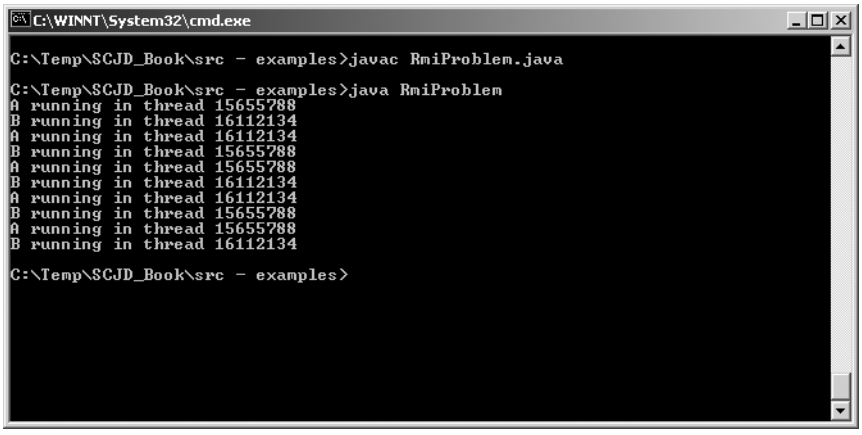


Figure 5-7. *Example of thread reuse within RMI*

As can be seen in Figure 5-7, both client A and client B use threads 15655788 and 16112134.

With this in mind, if we cannot use tokens and we have chosen to use RMI, then we need to find some other way of uniquely identifying our clients.

One way of handling this is to build our server using the Factory design pattern, where our factory creates a unique object for each connected client. We can then use the unique instance of our `DvdDatabase` class to identify the client.

Using a factory works for both a sockets-based solution (although it is generally overkill) and for an RMI solution. Since we are presenting both solutions in this book, we have decided to use a factory, and we will describe it in Chapter 6.

Creating Our Logical Reserve Methods

To create the `reserveDVD` and `releaseDVD` methods needed for logical locking, we need some class variables. For a start, we need to track the owners of the locks; a `Map` containing the UPC number and the owner is ideal. We also need a common `Lock` object to ensure that different threads do not try to lock the record simultaneously. Finally, we need a condition that threads can monitor to determine when they can attempt to acquire a lock again. These three variables are as follows:

```
private static Map<String, DvdDatabase> reservations
        = new HashMap<String, DvdDatabase>();

private static Lock lock = new ReentrantLock();
private static Condition lockReleased  = lock.newCondition();
```

Listing 5-2 contains the `reserveDVD` method.

Listing 5-2. *The reserveDVD Method*

```
boolean reserveDVD(String upc, DvdDatabase renter)
        throws InterruptedException {
    log.entering("ReservationsManager", "reserveDvd",
                 new Object[]{upc, renter});

    lock.lock();
    try {
        long endTimeMSec = System.currentTimeMillis() + 5000;
        while (reservations.containsKey(upc)) {
            long timeLeftMSec = endTimeMSec - System.currentTimeMillis();
            if (!lockReleased.await(timeLeftMSec, TimeUnit.MILLISECONDS)) {
                log.fine(renter + " giving up after 5 seconds: " + upc);
                return false;
            }
        }
        reservations.put(upc, renter);
        log.fine(renter + " got Lock for " + upc);
        log.fine("Locked record count = " + reservations.size());
        log.exiting("ReservationsManager", "reserveDvd", true);
        return true;
    } finally {
        // ensure lock is always released, even if an Exception is thrown
        lock.unlock();
    }
}
```

We have decided to allow any UPC to be reserved, even if no such record exists. This ensures that a DVD can also be reserved when we are first adding it to the system. The alternative would be to have the `reserveDVD` method start by verifying that the record does exist, throwing some exception if the record does not exist. If we had any delete methods, we would also have to verify that the record existed after we acquired the lock. However, if we did this, we would also have to change the logic of the `addDVD` method, and possibly the `updateDVD` method (remember, they both defer to the same private `persistDVD` method, which might also have to be updated).

Our `reserveDVD` method acquires a mutual exclusion lock, and then goes into a while loop, waiting until we either time out or acquire the lock.

Our `DBClient` interface specifies that we must return false if we were unable to lock the record within 5 seconds. Under JDK 1.4 there was no guaranteed way of determining whether a call to `wait(timeout)` had timed out or whether notification had been received. JDK 5 has a new `Condition.await` method, which will return a Boolean `true` if we were notified by the unlock method that we can continue processing, and `false` if we timed out.

If we have acquired the lock, we add a record to the map of lock owners, indicating that we own the lock.

Finally, we release the mutual exclusion lock.

The Logical Release Method

The `releaseDVD` method is the counterpoint to the `reserveDVD` method shown earlier, and the code is very similar.

```
void releaseDvd(String upc, DvdDatabase renter) {
    log.entering("ReservationsManager", "releaseDvd",
                 new Object[]{upc, renter});
    lock.lock();
    if (reservations.get(upc) == renter) {
        reservations.remove(upc);
        log.fine(renter + " released lock for " + upc);
        lockReleased.signal();
    } else {
        log.warning(renter + " cant release lock for " + upc + ": not owner");
    }
    lock.unlock();
    log.exiting("ReservationsManager", "releaseDvd");
}
```

First, we ensure we have a lock on the mutual exclusion `Lock`; then if it is the owner of the reservation who is releasing it, we remove our lock indication from the map. Then we signal any waiting threads that they can try to acquire locks. If the wrong `renter` instance is passed to this method, a warning message is logged. Finally, we release our mutual exclusion `Lock`.

Discussion Point: Deadlock Handling

Deadlocks occur when a thread is blocked forever, waiting for a condition that cannot occur. Chapter 4 discusses deadlock handling from a threading perspective, but the same issues apply at an application logic level as well. Consider what would happen if two clients were both trying to get logical locks on the same two records, but in different orders, as shown in the following example:

- Client A gets a logical lock on record 1.
- Client B gets a logical lock on record 2.
- Client A attempts to get a logical lock on record 2.
- Client B attempts to get a logical lock on record 1.

Our `reserveDVD` method times out after 5 seconds. But if both client A and client B immediately retried to get the lock, the result would be the same as if we didn't have a timeout: both threads would effectively be deadlocked.

There are many solutions to prevent deadlocks, among them:

- Don't allow a client to ever lock more than one record at a time. If they cannot lock multiple records, then you cannot get a logical deadlock.
- Only allow clients to lock records in numerical order. Under these rules, client B would not be allowed to attempt to lock record 1 in our previous example, and would have to give up the lock on record 2 at some point, which would allow client A to continue.
- Have a dedicated mechanism for tracking that locks are owned and which locks are in contention. Each time that a new lock goes into contention, this mechanism would be checked to see that a deadlock will not occur, and if it will, the lock is cancelled. This is the most complex of the possible solutions, but the code required is mostly recursive and can be written simply with a little thought.
- Ignore the problem. Seriously—is it a requirement of your assignment? Is there a possibility that attempting to handle this problem could result in you making a mistake that might cost you marks? Do you feel that this is out of scope for the assignment?

Once again, we are going to recommend that you read the instructions you received from Sun very carefully to determine what you need to do about deadlock handling (if anything). However, if your instructions do not mention this at all, you will have to decide for yourself whether you want to handle deadlocks. Some of the questions you might like to ask include the following: Is it more professional to have deadlock handling? Does deadlock handling add unnecessary complexity to the code? Can you handle deadlocks with the exceptions you are allowed to throw?

Whatever you decide, this is a design decision that you might like to document.

Discussion Point: Handling Client Crashes

Earlier in this chapter, we discussed the potential for having thick clients, where the client will be responsible for obtaining a logical record lock and later releasing it. But what happens if the client crashes (or is just shut down) sometime after requesting a logical record lock but

before releasing the logical record lock? In such a case, the record will be locked for all time—no other client will ever be able to lock the record.

Once again there are many possible solutions, some of which include the following:

- Having a thinner client (where the client just calls a `rentDVD` method on the server, and the server calls both the `reserveDVD` and `releaseDVD` methods) will bypass the problem totally. The lock should never become totally unavailable. We recommend you refer back to the section on fat/thin clients earlier in this chapter to see the ramifications of this (and possibly join in the discussions on JavaRanch on this topic).

- If we have a socket network solution, then the server thread servicing the client will receive an exception when the client disconnects. If that thread keeps track of which locks have been granted, then it could release them if it receives this exception.

- If we have a server factory, with unique `DvdDatabase` objects as our client identifiers, we could store the reservation data in a `WeakHashMap` with the client identifier as the key. When the client disconnects, the `DvdDatabase` for that client will (eventually) be garbage collected and the lock will be automatically removed from the `WeakHashMap`. In this case, we would probably want a separate thread monitoring the `WeakHashMap` so that it can notify any waiting threads that a reservation has been cleared.

- If we have a server factory with unique workers per RMI client, we could have the worker implement `java.rmi.server.Unreferenced`. The `unreferenced` method from this interface will be called sometime after the RMI client disconnects. If the worker instance keeps track of which locks have been granted, then it could release them if `unreferenced` is called.

- Ignore the problem. Again, seriously—is it a requirement of your assignment? Is there a possibility that attempting to handle this problem could result in you making a mistake that might cost you marks? Do you feel that this is out of scope for the assignment?

Once again, we are going to recommend that you read the instructions you received from Sun very carefully to determine what you need to do about clients dying (if anything). However, if your instructions do not mention this at all, you will have to decide for yourself whether you want to handle the possibility of locks never being released. Some of the questions you might like to ask include the following: Is it more professional to handle disconnected clients? Does handling disconnected clients add unnecessary complexity to the code?

Whatever you decide, keep in mind that this is a design decision that you might like to document.

Discussion Point: Multiple Notification Objects

Consider what will happen if 100 threads are all waiting to reserve different records. When a thread releases its lock on any one record, it will call `lockReleased.signal`, and *all 100 threads* will be notified that a lock has been released, so all 100 threads will attempt to regain the `lock` mutex and check whether it is the record that they are interested in that was released—and potentially the released record was not of interest to any of them! So there would be a sudden burst of CPU activity each time a record is released, potentially lowering productivity on the server each time.

A better solution would be to have each thread get notified only when the record it is interested in becomes free.

Under JDK 1.4 this would have been difficult to achieve; you would have had to synchronize on different objects to be able to ensure that only a specific thread gets notified. Under JDK 5 you can have all your threads obtain a lock on the same object but use different `Conditions` upon which they should be notified.

JDK 1.5's reentrant locks, with their different syntax to synchronized blocks, provide the ability to create hand-over-hand locking, where a lock is requested, then a second lock is requested, then the first lock is released, and so on. No lock in the chain is released until the next lock is granted.

Conceptually the replacement for the `reserveDVD` method would look like the code in Listing 5-3. Note that the line numbers are specific to this discussion, and have no relationship to the line numbers for the original `reserveDvd` method.

Listing 5-3. *A Less CPU-Intensive reserveDvd Method*

```
1      private static Map<String, LockInformation> reservations
2              = new HashMap<String, LockInformation>();
3
4      private static Lock masterLock = new ReentrantLock();
5
6      public boolean reserveDvd(String upc, DvdDatabase renter)
7              throws InterruptedException {
8          LockInformation dvdLock = null;
9          masterLock.lock();
10         try {
11             dvdLock = reservations.get(upc);
12             if (dvdLock == null) {
13                 dvdLock = new LockInformation();
14                 reservations.put(upc, dvdLock);
15             }
16             dvdLock.lock();
17         } finally {
18             masterLock.unlock();
19         }
20
21         try {
22             long endTimeMSec = System.currentTimeMillis() + 5000;
23             Condition dvdCondition = dvdLock.getCondition();
24             while (dvdLock.isReserved()) {
25                 long timeLeftMSec = endTimeMSec - System.currentTimeMillis();
26                 if (!dvdCondition.await(timeLeftMSec, TimeUnit.MILLISECONDS)) {
27                     return false;
28                 }
29             }
30             dvdLock.setReserver(renter);
31         } finally {
```

```
32              dvdLock.unlock();
33          }
34          return true;
35      }
```

We start by getting a lock on the `masterLock`, which allows us to retrieve the lock for our specific UPC at line 11 or create a new lock and add it to the map if necessary at lines 13 and 14. Without this master lock, another thread could be modifying the map while we are trying to get our lock.

In JDK 1.4 we would now have a problem—we no longer want to keep the lock on `masterLock`, but if we release the `masterLock` before we gain a lock on our particular `dvdLock`, there is the risk that some other thread could modify the `dvdLock` before we managed to lock it. However, because JDK 1.4 locking works in terms of synchronized blocks, it is not possible to lock `dvdLock` and then release `masterLock`.

Hand-over-hand locking solves this problem for us. Before releasing the lock on `masterLock` we lock `dvdLock` in line 16. At this point we own two locks, but we release the `masterLock` in line 18, bringing us back to a single lock. At no time did we own zero locks, and the time that we owned two locks was reduced to a minimum.

When we get to line 26, each client will be waiting for the condition specific to the lock they are after. They will no longer be woken up when *any* lock is released.

This relies on a `LockInformation` class, which is shown in Listing 5-4.

Listing 5-4. *The LockInformation Class*

```
1       class LockInformation extends ReentrantLock {
2           private DvdDatabase reserver = null;
3           private Condition notifier = newCondition();
4
5           void setReserver(DvdDatabase reserver) {
6               this.reserver = reserver;
7           }
8
9           void releaseReserver() {
10              this.reserver = null;
11          }
12
13          Condition getCondition() {
14              return notifier;
15          }
16
17          boolean isReserved() {
18              return reserver != null;
19          }
20      }
```

The `releaseDVD` method would use the `Condition` from the instance of the `LockInformation` class for the record being unlocked to only notify threads waiting on this particular record. If there are a large number of threads waiting for records, this could significantly reduce the

Again, we recommend that you read the instructions you received from Sun very carefully to determine what you need to do about multiple threads receiving notification simultaneously (if anything).

■ **Tip** We have presented this section on having multiple notification objects as it does appear to meet a requirement listed in at least some of the Sun instructions. However, at the time of this writing Sun does not appear to penalize those candidates who ignore this requirement.

Summary

In this chapter we showed how to build classes that can read the data file and provide locking. We discussed some of the common pitfalls that can occur, and explained how to avoid them. We also presented several examples of design patterns, and examined how they can be used.

We used many of the techniques that you need to use in your Sun SCJD assignment, including reading and writing from random locations in files, and locking and releasing records. We also used many of the new techniques introduced in JDK 1.5, and introduced some of the new APIs.

One word of caution: As we mentioned in Chapter 3, we have made some parts of our sample assignment more difficult than the real assignment. You may find far simpler solutions for your Sun assignment than those presented here (and we strongly recommend you look for them). Similarly, some portions of the Sun assignment may be more difficult than what we have presented here. You will also note that there are methods in your `Data` class you must implement that we have not even mentioned; we believe we have provided you with the basic information you need to create these methods on your own.

FAQs

Q Will I have to create my own data file in the Sun SCJD assignment?

A No; at the time of this writing, Sun provides a sample data file for you, along with the details of the file format necessary for you to read and write the file.

Q Should I provide my test classes and build scripts in my submission?

A We recommend against providing anything outside of the requirements. At the time of writing, the instructions from Sun include a statement that going beyond the requirements will not gain you extra marks. So providing the test classes and build scripts cannot gain you anything.

Q Should I build the `Data` class in stages?

A We strongly recommend you do build each of your classes in stages, ensuring that each part is correct before moving on to the next part.

Q Is it necessary/possible for my `Data` class to use a value object?

A You will have to check your instructions to see whether or not this is possible. All the instructions to date have been very explicit about the required method signatures for the `Data` class. However, even if you cannot use them in the `Data` class, your instructions might allow value objects to be used elsewhere in your solution.

Q Is it necessary for my `Data` class to use the Façade pattern?

A There are no specific patterns that must be used within your submission; you are free to use any that you feel fit your requirements. Similarly, there are no requirements for you to acknowledge which patterns you are using, but it is considered good programming practice to mention the patterns in your design decisions documentation and/or your Javadoc APIs as doing so will assist other programmers in understanding your code.

Q Should I use a cache in my Sun SCJD assignment?

A Usually the Sun instructions tell you that a clear design is preferred over a higher performance design. So you should decide for yourself whether the addition of a cache makes your code easier to read, and whether the gains are worth your while.

Q What happens if using a cache causes the JVM/computer to run out of memory?

A You could use lazy loading of records (only loading them when required) in conjunction with `SoftReferences` to ensure that the JVM can clear the records if it is running out of memory. If the JVM does clear the record from memory, using a lazy loading scheme will result in you reloading it next time you need it (and presumably some other low-usage object being cleared). That said, you might want to calculate just how many records it will take to use all the memory on your computer—and whether you could even access that many records with the provided APIs.

Q Am I allowed to use the `java.nio` (NIO) packages or the `java.util.concurrent` packages in my application?

A In the past Sun has displayed information on its web site indicating that use of the NIO packages was not allowed for the SCJD assignment. However, Sun has also stated that the instructions you download from its site are authoritative; if your official instructions state that you must not use a particular package, then you must not use it. However, if your instructions do not ban a particular package, then you are free to use it.

CHAPTER 6

Networking with RMI

In this chapter, you will develop the background needed to implement a complete networking solution using Remote Method Invocation (RMI). To that end, this chapter will cover RMI and, to a lesser extent, serialization. Figure 6-1 illustrates where the networking solution fits in the overall architecture of the system, which is nestled between the GUI and the data layer.

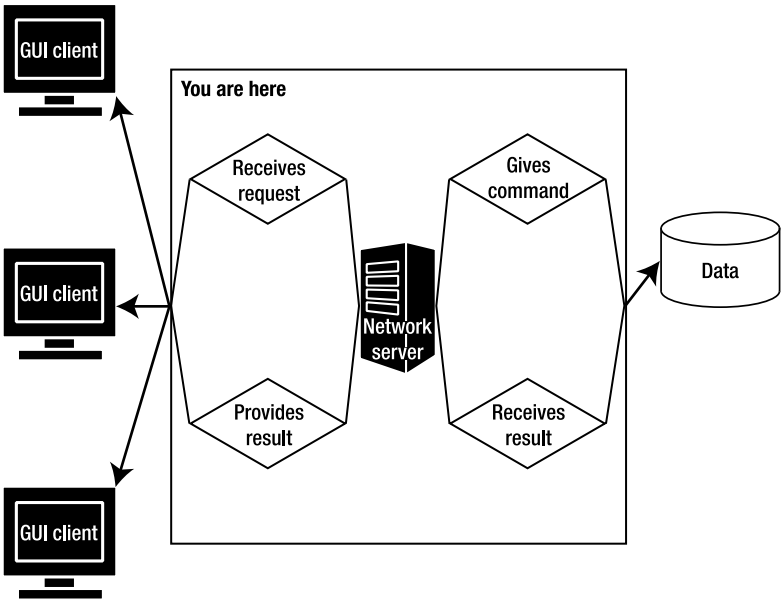


Figure 6-1. *The networking tier of Denny's DVDs version 2.0*

Before you can fully understand RMI, it is important to familiarize yourself with some of the ins and outs of serialization, since RMI relies on this feature. Serialization is a topic that you may already be familiar with from the Sun Certified Java Programmer (SCJP) exam, but we will still cover the topic briefly. These topics are some of the big-picture points developed in this chapter:

- Understanding object serialization in Denny's DVDs
- Evaluating the pros and cons of using RMI as a network protocol

- Implementing a remote object and defining a remote interface
- Examining the Factory pattern and learning why we are using it for our RMI implementation
- Marshaling and demarshaling in RMI
- Registering your remote objects

■ **Caution** The next chapter describes Java sockets, which is the other networking protocol the examinee may opt to use for the certification project. However, be advised that for the certification project, you must send serialized objects if opting for a sockets solution as a networking protocol. Java RMI requires serialization, but it is not absolutely necessary to use serialization in a Java socket implementation. You could, for instance, implement your own wire format in a socket implementation so that any client-side technology could submit requests to a Java socket server. In the next chapter, we explain the technique of using sockets with serialization objects. If you decide to implement your solution using sockets as the networking protocol, the next section, "What Is Serialization?", will still be useful.

What Is Serialization?

So what is serialization, and why would you need it in your program? Let's consider an example. Suppose there are two machines on the same network. Machine A sends a message to a Java program on machine B; how does machine B receive and understand the message? What's required is a transfer protocol. A *transfer protocol* will flatten the data that needs to be transferred by putting it into a format that can be sent across a network.

Part of the process of sending data across a network involves serialization. *Serialization* is the process of deconstructing an object into its components of type information and state, and then reconstructing a copy of the original object on the receiving end by reading in the type information and then the field values. The term often used to describe the process of transferring serialized objects across a network is *marshaling*. Sometimes the reverse process, restoring the serialized object on the other end, is referred to as *demarshaling*, or *unmarshaling*. Serialized objects can be persisted to a local file system, courtesy of Java file I/O, or sent across a network via a socket. This is accomplished by converting the object into a serial stream of bytes, transmitting that stream across a network or to a file system, and then reading the persisted byte stream back into memory in order to re-create the object graph (see Figure 6-2), which is Java vernacular for reconstructing the original object and its state as if nothing ever happened. When a byte stream is loaded into memory in the form of an object, it is said to be in an *active* state. When that same byte stream is located in a file, rather than in a Java Virtual Machine (JVM) memory bank, it is said to be in a *passive* state. Keep in mind the notion of an object being in an active or passive state, as it will come in handy when we introduce the `Activatable` interface later in this chapter. Let's discuss the process in a little more detail.

The first of three steps is converting the object to a serial stream of bytes. First, type information gets written as header information to the stream, and then state information is written. The state information consists of the values of the class members, except for the static and transient members (assuming the default serialization mechanism, which does not implement `readObject` and `writeObject`; this is discussed a bit later in more detail). On the deserialization end, the receiving remote object first reads the type information in from the header and creates an instance of the class. If the class cannot be located (either locally or remotely), then a `ClassNotFoundException` is raised. Once the class is instantiated, the field values are read in from the stream and the values set in the marshaled object.

■ **Note** Serialized objects are a copy of the original, not a reference to them. Parameters and return values in RMI are passed by values or copies. An object can be referenced on another machine only if the object is a remote object—that is, the object is exported via the `UnicastRemoteObject.export()` method or `UnicastRemoteObject` is extended.

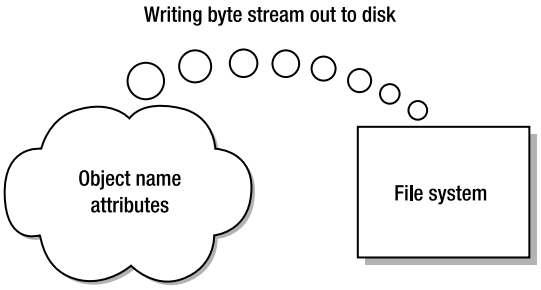


Figure 6-2. *Creating the byte stream from an object graph*

Using the serialver Tool

Serialization involves implementing the `java.io.Serializable` interface. This interface is of interest because it does not have any methods that require implementing. In fact, the main purpose of implementing `Serializable` is to inform the JVM that the object can be serialized. `Serializable` is often referred to as a marker interface. Formally a marker interface is an interface, which does not define any methods that a class or subclass would be required to implement but rather identifies the object as being of a certain type. To determine if a class is serializable, use the tool `serialver`. To start the `serialver` tool, from the project root enter the following at a command prompt:

```
serialver -classpath classes/. -show
```

Let's run the `serialver` tool on two of our classes, one that is serializable and one that is not serializable. We will use the `DVD` class as our serializable class and `DvdFileAccess` as the class that is not serializable. Figures 6-3 and 6-4 demonstrate the use of `serialver` on our `DVD` and `DvdFileAccess` classes, respectively.

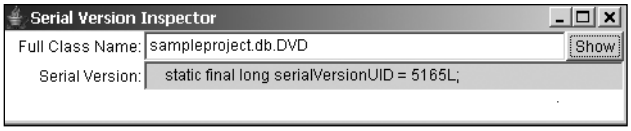


Figure 6-3. *Running the serialver tool on the DVD class*

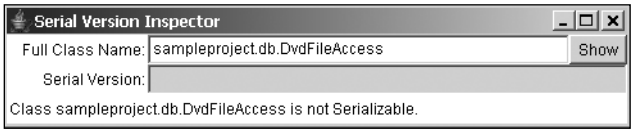


Figure 6-4. *Running the serialver tool on the DvdFileAccess class*

Since we started `serialver` from the `DennyDVD` classes directory, we can inspect our project's classes for the `Serializable` interface. If we had started from another directory, we would have had to make sure that our project was listed in our system classpath. *Inspecting* our classes for the `Serializable` interface is not very interesting since we have the source code and are well aware which classes are `Serializable` and which are not (i.e., we can simply inspect the file visually to see if `Serializable` is implemented). But using `serialver` on classes in which we do not have source code is much more useful—for instance, classes in a JAR file or classes that are part of the JDK.

For fun, try inspecting other classes in your classpath, such as those in the Java JDK (i.e., `java.util.Date` or `java.lang.String`).

If you inspect the source code for both the `DVD` and `DVDFileAccess` classes (downloadable from `http://www.apress.com`), you will note that `DVD` does indeed implement `Serializable`, while `DVDFileAccess` does not.

The Serialization Process

To actually persist an object's state, there are two classes in the `java.io` package that perform the bulk of work required for reading and writing serializable objects: `ObjectOutputStream` and `ObjectInputStream`. `ObjectOutputStream` has a method called `writeObject`, while `ObjectInputStream` has a method called `readObject`. In the Denny's DVDs project, serialization is used to send method parameters across the network between the GUI and the server using the default serialization mechanism (e.g., whether the server is an RMI or a socket server). We do not actually serialize our method parameters explicitly for RMI to work. In fact, our only concern is that the object sent across the network be serializable. Let's assume that we want to persist the state of our DVD objects explicitly to the file system. In that case, we could do so by implementing the `persistDVD` and `retrieveDVD` methods. When the client wants to save the state of a database record or DVD object, such as after a `setRecordNumber` call, the client invokes the `DVDFileAccess` method `persistDVD`. `persistDVD` serializes the `DVD` object using the `ObjectOutputStream` class and its `writeObject` method. The `persistDVD` method is shown in Listing 6-1. Keep in mind that the actual member we are serializing is a DVD. Our `DVD` class must be `Serializable`. The `retrieveDVD` method reads the serialized object and re-creates the object, as demonstrated in Listing 6-2.

Listing 6-1. *persistDVD Method Demonstrating FileOutputStream to Serialize a DVD Object*

```
private boolean persistDVD(DVD dvd) throws IOException{
       boolean retVal=false;
       //open a FileIOutputStream associated with the data
       //notice that if the DVD does not already exist,
       //it will be created.
       String filePath = dbName+"/"+ dvd.getUPC() + fileExtension;
       FileOutputStream fos = new FileOutputStream(filePath);
       try {
                  ObjectOutputStream oos = new ObjectOutputStream(fos);
                  //Read in the data from the object
                 oos.writeObject(dvd);
       }
        finally {
                     //close all references
                    oos.close();
                           fos.close();
        }
}
```

Listing 6-2. *retrieveDVD Method Demonstrating FileInputStream to Deserialize a DVD Object*

```
private DVD retrieveDVD(String upc, String fileExtension) throws IOException,
ClassNotFoundException {
               DVD retVal = null;
               //get the path to the object's serialized state.
               try {
                             String filePath = dbName+"/"+ upc + fileExtension;
                             FileInputStream fis = new FileInputStream(filePath);
                             //Read in the data from the object
                            ObjectInputStream ois = new ObjectInputStream(fis);
                            retVal = (DVD)ois.readObject();
               }
               finally {
                             ois.close();
                             fis.close();
               }
               return retVal;
}
```

■ **Caution** The code shown in these listings is not contained in the actual project since Denny's DVDs relies on the JVM and the default serialization mechanism. The code is used as a pedagogical device to illustrate how to explicitly serialize an object.

As mentioned previously, information pertaining to the object's state is persisted to a file via a stream of bytes. But that is not all of the information that is saved. Information regarding the class type and its version number is also persisted. This is important because the class loader needs to properly deserialize the class from its persisted state. If the `serialVersionUID` is not declared, then the `ObjectOutputStream` will generate a unique version number for the class. Any subsequent changes to the class's members will generate a new version number at the next compilation. In the `sampleproject.db.DVD` class, the version ID is defined as a private member defined as follows:

```
private static final long serialVersionUID = 5165L;
```

■ **Note** Often, serialization compatibility between versions needs to be maintained even though minor changes that will not break serialization compatibility have occurred. In these situations you need to declare the `serialVersionID`. You can use `serialver` to do this initially. This will allow you to define your own numbering schemes for the `versionID` for the class. Interestingly, even though this is a private field, the JVM knows it is there and uses it when defining a class.

Tweaking the Default Serialization Mechanism

There may be times when you want more control over how an object's state is serialized. For instance, perhaps you are only concerned with persisting part of the object's state, but not all of it. This is useful if the class contains a private member for holding a reference to a database connection, and you do not want to persist that information because the connections are only meaningful during a single session and are assigned on an as-needed basis. It wouldn't make sense to try to persist a database connection for the next time the application is run. The next time you load the object with the connection reference, you can simply assign that member a new connection. Another situation where you would not want to persist object information is when an object member is of a type that is not serializable. For instance, in our sample project the J2SE 1.5 `Logger` instance in the `DVD` class is not serializable.

But how do we indicate to the JVM that we don't want to persist a specific class member but would like to persist the other class members? Java provides for this functionality through the use of the keyword `transient`. The byte stream that gets persisted will not include any members that are declared using the keyword `transient`.

We use the keyword `transient` in our serialization implementation since the `Logger` instance cannot be serialized. (Hint: Try running `serialver` on the logger.) Conceptually this works out well since a logger member does not really add anything essential to the notion of a DVD. In general, a logger is the sort of thing that does not need to be persisted, and we can re-create a complete DVD record by reinitializing the logger. The following is an example of how you can use the keyword `transient` in the `Logger` instance in `DVD.java`:

```
private transient Logger log = Logger.getLogger("sampleproject.db");
```

There is another approach to serialization that we should mention. Rather than implement `Serializable`, an alternative approach is to implement a subinterface of `Serializable`:

the Externalizable interface. The big advantage of the Externalizable interface is performance. The algorithm for serialization uses reflection for marshaling and demarshaling. The serialization algorithm will systematically determine the nontransient and nonstatic class members through the use of reflection.

However, this can be quite expensive because the reflection algorithm costs in terms of CPU cycles and memory. The costs result from the JVM having to dynamically discover class properties during runtime. In situations where performance is more important than flexibility, consider using a reflectionless approach to serialization, such as Externalizable. For more information on reflection, refer to the following Sun tutorial: http://java.sun.com/docs/books/tutorial/reflect/.

Customizing Serialization with the Externalizable Interface

If performance is a must, then you can serialize using the java.io.Externalizable interface instead of the Serializable interface. So just how does Externalizable improve on the default serialization mechanism? Externalizable requires that you write the details of reading and writing the object's state to the byte stream. This is much more tedious than relying on reflection, but ultimately it provides more of a speed burst to your application. The ObjectOutputStream class no longer simplifies this process. You must use the methods readExternal and writeExternal and be aware of the member's type—whether it is a primitive type, a String (i.e., UTF), or some other type of object, other than String, that is serializable. Since you are involved in the low-level handling, you must read your object's members in the same order in which they were written to the stream.

Listing 6-3 presents an example of how our DVD class might look if it were Externalizable. For illustration purposes, the nuts and bolts of the class are not shown—just the code that relates to writeExternal and readExternal. Both methods can be private since the serialization mechanism can circumvent the normally applicable accessibility rules for classes (i.e., the JVM can invoke an object's private serialization methods).

■ **Note** If the readObject and writeObject methods are implemented in the Serializable object, the heavy reliance on reflection is not required and there is actually a performance edge over the default serialization mechanism (i.e., implementing Serializable without overriding the readObject and writeObject methods).

This approach is similar to externalization, with some minor differences with regard to inheritance and the handling of class metadata. The signatures of the methods to override appear in Listing 6-3. When you override these methods, it is crucial that the order in which the class attributes are written to the stream is the same as the order they are read back. As long as the serializable objects are not part of a class hierarchy, readObject and writeObject are implemented exactly as readExternal and writeExternal, as discussed later in this section. Here are the method signatures for readObject and writeObject:

```
private void writeObject(java.io.ObjectOutputStream out) throws IOException
private Object readObject(java.io.ObjectInputStream in) throws IOException, ➥
ClassNotFoundException
```

Listing 6-3. *An Externalizable Version of DVD.java: Implementing the readExternal and writeExternal Methods of DVD.java*

```
/**
* Required method for Externalizable interface.
* Specifies how the object graph gets converted to a byte stream.
*/
private void writeExternal(ObjectOutput out) throws IOException{
               out.writeUTF(upc);
        out.writeUTF(name);
        out.writeUTF(composer);
        out.writeUTF(director);
        out.writeUTF(leadActor);
        out.writeUTF(supportingActor);
        out.writeUTF(year);

        out.writeInt(copy);
}
/**
* Required method for Externalizable interface.
* Specifies how to recreate the object graph from
* a byte stream. The order members are read must
* match the order in which the object members were
* written to the stream.
*/
private void readExternal(ObjectInput in) throws IOException{
              out.readUTF(upc);
              out.readUTF(name);
              out.readUTF(composer);
              out.readUTF(director);
              out.readUTF(leadActor);
              out.readUTF(supportingActor);
              out.readUTF(year);
              out.readInt(copy);
}
```

■ **Caution** As was the case with `persistDVD` and `retrieveDVD`, the `readExternal` and `writeExternal` methods are not included in the actual project code base, but are used merely as a pedagogical device for demonstrating externalization.

Since `Externalizable` is considered a more advanced serialization approach, we will not cover `Externalizable` any further in this book. Rather, we have chosen to implement `Serializable` for simplicity, but `Externalizable` is mentioned so that you understand the alternative. `Serializable` is simpler to implement but sacrifices performance and flexibility.

Using `Externalizable` results in code that is more difficult to maintain. For instance, if we were to add a new class member to the `DVD` class—say, a serializable type such as boolean—our preceding read and write code would not change. However, if `DVD` implemented `Externalizable` instead of `Serializable`, then we would have to update the `readExternal` and `writeExternal` methods to incorporate the change. In addition, the externalization methods should include logic to check for specific versions of serial version IDs to determine if they were dealing with older versions of the class. In a production system, this can be an annoyance if you have a lot of classes that tend to change over time. Table 6-1 presents a comparison of serialization to externalization.

■ **Tip** The use of `transient` in a class that implements `Externalizable` is not required. Since the `Externalizable` interface requires that the details of reading and writing the object's state be defined, the `transient` keyword is not necessary. In fact, it is completely ignored in an `Externalizable` object.

■ **Note** Since application performance is not a consideration in our design; we have opted for plain serialization over externalization in our implementation of Denny's DVDs version 2.0.

Table 6-1. *Comparing Serialization and Externalization*

Serialization	Externalization	Advantage Goes To . . .
Easier to use. Just implement `Serializable` and use `writeObject` and `readObject` to persist state.	More complex. Must implement the `Externalizable` interface methods `readExternal` and `writeExternal`.	`Serializable`
Less efficient algorithm. Uses reflection to determine object's makeup.	Better performance. No need to use reflection since you write the code.	`Externalizable`
More data gets serialized due to larger class description and versioning information.	More control over what gets serialized.	`Externalizable`

Introducing RMI

An important objective of the SCJD exam is for the examinee to develop a solution that will allow machines on a network to exchange messages. Such communication, known as distributed computing, can be a challenging task, but RMI is one of the helpful tools at your disposal. However, RMI is not the only game in town. There are other technologies, such as RPC, Common Object Request Broker Architecture (CORBA), and Microsoft's .NET technology. In fact, RMI can be thought of as object-oriented RPC.

Note .NET has a similar and competing technology called .NET remoting.

This is a good time to define some terms. When we use the term *server* in RMI programming scenarios, we are referring to a remote object that has methods that can be invoked from another JVM. A *client* is the object that invokes the remotely accessible methods of the server. An RMI *distributed object system* provides remote objects that can be invoked by clients. Communication between a client and a remote object is two-way. A client must be able to locate and communicate with remote objects, invoke their methods, and receive their return values.

RMI specifically enables Java objects on different machines to communicate with each other. One of the motivations behind RMI is for developers to interact with remote Java objects as if they were local objects. The actual location of the object is transparent to the developer. Network transparency is a very appealing feature, because it abstracts the complexities involved in distributed object systems so that they behave as if they were local object systems. So in an RMI system, the client runs locally, while the rest of the system runs on a different machine. These types of systems are often referred to as distributed object systems. Figure 6-5 illustrates this type of system.

Note An alternative to sockets and RMI is Remote Procedure Call (RPC). RPC is language- and processor-independent, assumes that parameters are network representations of simple data types such as ints and chars, and can be run on almost any platform. RMI is only processor-independent, assumes an object-oriented framework, and requires the use of Java.

In addition to RPC, CORBA and Simple Object Access Protocol (SOAP) are other technologies that allow processor and language independence between different platforms. But because RMI assumes the use of Java, many of the tedious protocol-level tasks have been built into RMI, leaving the developer with more time to worry about application logic. Java's RMI is very good for Java systems, since it can assume that a JVM will be present on both sides of the network connection. For these reasons, RMI is preferable to RPC and CORBA in distributed object systems built entirely with Java. However, as a scandalous side note, CORBA systems tend to perform better than RMI systems, mainly because they can be written in C or C++. Closely related to RMI is RMI-IIOP, which stands for Java Remote Method Invocation Run Over Internet Inter-Orb Protocol. RMI-IIOP was coproduced by Sun and IBM in order to achieve interoperability between CORBA-compliant applications and Java. Using the Java development language, you can create the Interface Definition Language (IDL) that will work with CORBA applications written in languages such as C++. The `rmic` compiler supports the option `-iiop`, which will produce the IDL files.

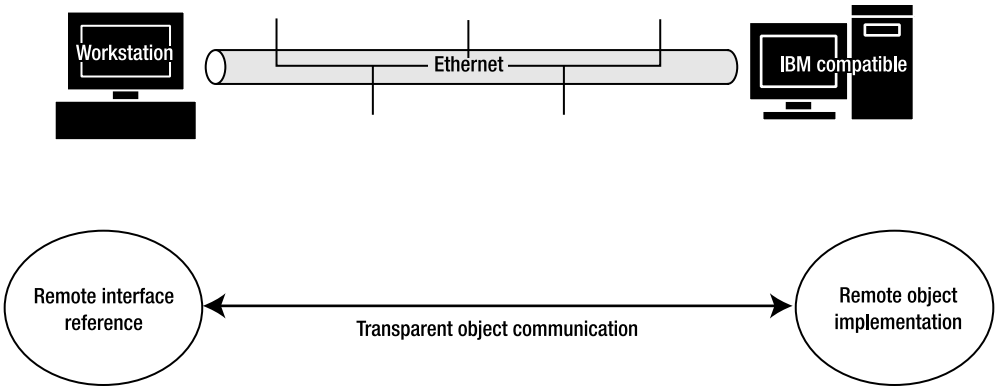


Figure 6-5. *Distributed object system*

The Delivery Stack

A transport protocol is a protocol built directly above the network layer; the two most common examples are TCP/IP and UDP. Above the transport layer is the transfer layer, which is responsible for the transparent transfer of data between hosts, flow control, and end-to-end error recovery and data transfer. The transport layer issues requests to the network layer. Figure 6-6 shows the technology stack complete with the network layer that RMI applications are built upon.

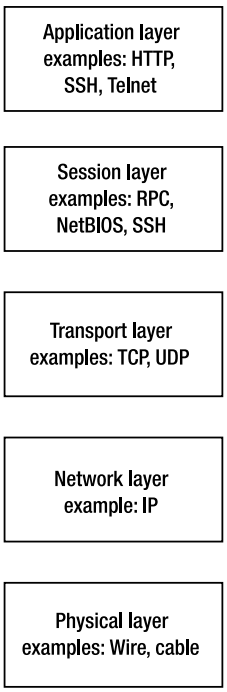


Figure 6-6. *The delivery stack*

A transfer protocol resides above a transport protocol and establishes how information is trasnferred between hosts. A very common example of a transfer protocol is Hypertext Transfer Protocol (HTTP). HTTP is the protocol for the World Wide Web. Java RMI does not use HTTP as a transfer protocol but instead uses the Java Remote Method Protocol (JRMP). One shortcoming to JRMP is that Java must be understood on both ends, the sending and receiving end, tying the solution to a Java RMI framework. To alleviate the dependency on Java, CORBA-related clients can be used by specifying `-iiop` for stub generation with the `rmic` tool. IIOP ensures the CORBA clients can be used with Java remote object implementations, thus reducing the dependency on a strictly Java solution.

Table 6-2 summarizes the three choices developers have for choosing a transfer protocol when using RMI.

Table 6-2. *RMI Transfer Protocols*

Protocol	Description
RMI-JRMP	The default transfer protocol for RMI. For client-server applications relying entirely on Java as the programming language. A Java-to-Java solution.
Java-IDL	For CORBA developers wanting to use Java in conjunction with interfaces defined in CORBA-compliant applications. Essentially this is a CORBA-to-Java solution.
Java RMI-IIOP	For Java developers needing to maintain compatibility with legacy applications. A Java-to-non-Java solution.

Return values and parameters are copies. Objects that are exported, or that implement a remote interface, are referenced via their stub, or client-side proxy. To export an object, extend `UnicastRemoteObject` or explicitly call `UnicastRemoteObject.export(<remote object>)`.

The Pros and Cons of Using RMI as a Networking Protocol

For the developer exam, the examinee can choose between RMI or serialized objects over sockets for the networking protocol. Here are some of the reasons for selecting RMI over sockets:

Object-based semantics—Remote objects look and feel just like local objects. The complexities of network-aware objects can be hidden from the programs using RMI remote objects.

No protocol burden—Unlike sockets, when working with RMI there is no need to worry about designing a protocol between the client and server, a process that is error-prone.

Method calls are type-safe in RMI—RMI is more strongly typed than sockets. Errors can be caught at compile time.

It's easy to add new objects with RMI or extend the protocol—You can more easily add new remote object interfaces or add methods to an existing remote object.

IIOP can be used for non-Java end points—You are not explicitly tied to a Java-to-Java solution.

Generating stubs just got easier with J2SE 5.0—If JRMP is used, there is no need to generate stubs explicitly with `rmic` any longer. But we will still be required to do so for our project.

■ **Caution** Even though Java 5.0 has added dynamic stub generation to alleviate RMI developers from explicitly invoking `rmic` on their remote classes prior to runtime, you must still do so for the certification project. This is important: **The use of `rmic` is still required** as of this writing for the Java developer certification project.

However, the relatively intuitive object semantics of RMI come at the expense of the network. There is communication overhead involved when using RMI, and that is due to lookups in the RMI registry and client stubs or proxies that make remote invocations transparent. For each RMI remote object, there is a need for a proxy, which slows the performance down.

Also it is important to be aware that thread management can sometimes cause issues if working with classes that are not designed around thread safety. Control and management of threads in RMI is delegated to the JVM and not the program.

The Classes and Interfaces of RMI

Figure 6-7 presents an overview of the classes and interfaces of RMI.

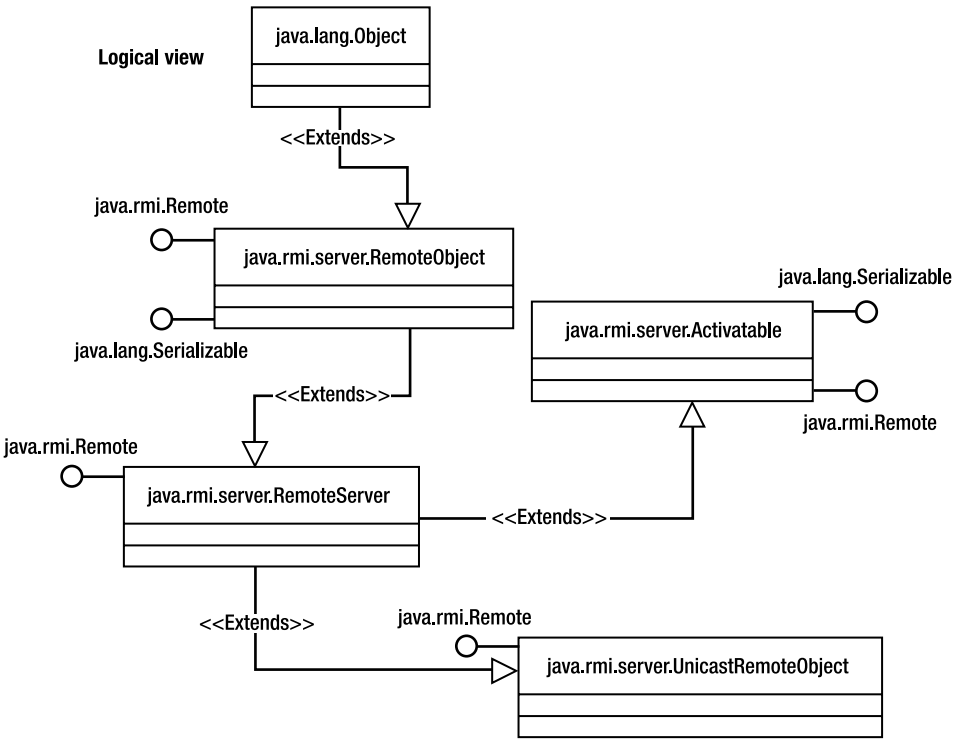


Figure 6-7. *The RMI classes and interfaces*

The Interfaces

The `Remote` interface is another marker interface, similar to the `Serializable` interface discussed earlier. The term *remote* indicates that the methods defined may be accessed from a different virtual machine than the one containing the remote object. For an object to be considered a remote object, the `Remote` interface must be implemented. Implementing this interface is accomplished by extending `java.rmi.Remote`.

Any public method defined in the `Remote` interface must throw a `java.rmi.RemoteException` or a super interface of `java.rmi.RemoteException` such as an `IOException`, `Throwable`, or `Exception`. This is the super class for all communication exceptions that occur during the invocation of a remote method. `RemoteException` is a checked exception and must therefore be enclosed in the try/catch block or specified in the method signature of any object that uses a remote object. A `RemoteException` may occur for either of the following reasons:

- The server is down, unreachable, or has terminated communication with the client.
- Errors were encountered during marshaling of params and return values.

One more interface is needed to create a remote object. The abstract class `java.rmi.RemoteObject`, which is displayed in Figure 6-7, must also be extended. You can think of `RemoteObject` as a remote object analog to `java.lang.Object`. `RemoteObject` provides the network implementations for the `java.lang.Object` methods `toString()`, `hashCode()`, and `equals()` that are appropriate for remote objects. Figure 6-8 shows the RMI layers.

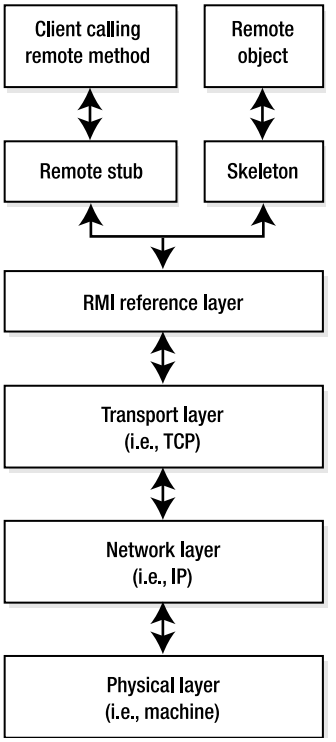


Figure 6-8. *RMI layers*

Another abstract class required for writing RMI programs is `java.rmi.RemoteServer`. The `RemoteServer` class is the common super class for server implementations and provides the framework to support a wide range of remote reference semantics. Specifically, the functions needed to create and export remote objects (i.e., to make them remotely available) are provided abstractly by `RemoteServer` and concretely by its subclasses. `RemoteServer` has two important subclasses: `java.rmi.server.UnicastRemoteObject` and `java.rmi.Activatable`.

Exporting remote objects is the responsibility of `Activatable` or `UnicastRemoteObject`. An `Activatable` object executes when requested and can turn itself off when necessary, whereas `UnicastRemoteObject` runs only when the server is up and running. An active object is one that has been instantiated and exported to a JVM. A passive object is one that has not been instantiated. Activation is the process of transforming a passive object into an active one. Lazy activation is the process of deferring activation until its first use. The `Activatable` interface provides RMI with the option of delaying access to persistent objects until needed. It is undesirable to permit an RMI server to use extensive system resources by loading a lot of remote objects that are infrequently used or not needed. Ideally, distributed systems should have access to thousands of persistent objects over very long, even indefinite, periods of time.

An RMI server that makes use of `UnicastRemoteObject` requires that remote objects be instantiated prior to a remote client invoking one of the remote methods. But in the case where we have many remote objects and the cost of instantiating all of them is prohibitive, an `Activatable` RMI server is an excellent alternative. With the `Activatable` interface used in conjunction with `rmid`, the RMI daemon, the server will instantiate the remote object when it is needed.

Luckily, we do not need to worry about this facet of resource management when implementing our RMI server, since we have only one remote object, `BookDatabaseImpl`, that acts as our RMI server and gateway to the database. We do not demonstrate an `Activatable` implementation in this book. Instead, we extend the `UnicastRemoteObject` class.

What Is an RMI Factory?

An RMI factory is just what the name would imply: an RMI version of the software design pattern aptly named the Factory pattern. As explained in Chapter 5 (which discussed data locking), we make use of an RMI factory to have a proper identifier mechanism for lock ownership. By ensuring that each RMI client has its own version of the `DvdDatabase` class, we avoid the issue of thread reuse, which is common in RMI and cannot be controlled explicitly through a JVM or RMI property. Since we cannot be sure how threads are being used and we are not allowed to use cookies or tokens to identify the requester, we exploit the Factory pattern to produce discrete instances of the `DvdDatabase`. The sequence-like diagram in Figure 6-9 depicts the typical actions and actors in a Factory pattern.

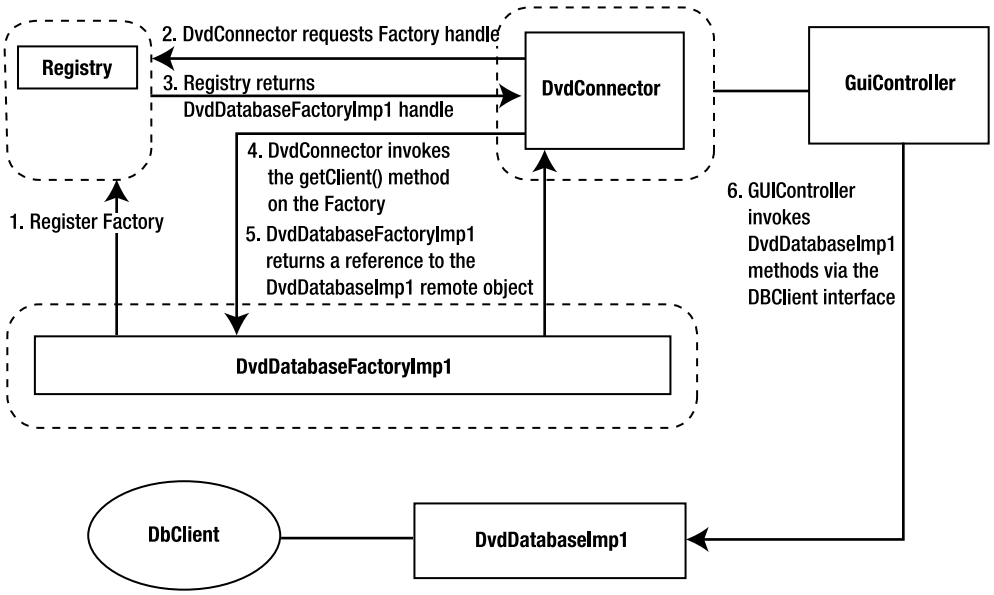


Figure 6-9. *The RMI Factory pattern applied to Denny's DVDs*

The Factory Pattern—What Is a Factory?

A *factory* is a software entity that creates other types of software entities. The client makes a request to a factory for a certain type of object, and out that object comes. Request a different type, and the factory will create and return the type requested. Of course, the factory needs to know how to create the requested object; otherwise the factory will complain that it doesn't understand the request. For instance, you wouldn't request a personal computer from a factory specializing in producing cars. Likewise, our factory will understand how to create remote objects. In particular, our factory will make a special type of remote object, a `DvdDatabase`.

Our factory is not a very robust factory in the sense that we only mass-produce one object type, a `DvdDatabase`. One of the most common motivations for implementing an RMI factory is to reduce the number of remote objects that need to be registered. This allows us to register the factory only once and gradually evolve and add new remote classes without having to register the new classes or even restart the registry. This is a nice by-product of an RMI factory even though it doesn't apply in our situation.

The RMI factory we implement in Denny's DVDs is an example of a parameterized Factory pattern, which is a type of creational pattern (i.e., patterns that involve the creation of objects and/or types; a singleton is another common example of a creational pattern). It is a variant of the parameterized pattern because we do not actually supply a name or parameter to the `getClient` method. It could be written `getClient(String class_name_to_create)`. However, since our application only has one remote interface, `DvdDatabaseRemote`, the type of class is implied. We do not need to specify. But we could easily modify this method or the factory to return multiple remote objects.

The RMI Implementation

To create a version of our server that acts as a factory, we first define the factory and its only method, `getClient()`, which allows us to obtain a unique instance of the `DvdDatabase` class (see Listing 6-4).

■ **Note** In this section we are talking about using the Factory pattern with our RMI solution, but it would work equally well with a sockets solution. However, the socket server already works essentially as a factory, creating a unique thread for each connected client, so we don't really need to explicitly create a unique object for each connected client. The very nature of a multithreaded socket server addresses the issue of thread reuse in an RMI thread pool.

Listing 6-4. *DvdDatabaseFactory.java*

```
interface DvdDatabaseFactory extends Remote  {
    public DvdDatabaseRemote getClient() throws RemoteException;
}

class DvdDatabaseFactoryImpl extends UnicastRemoteObject
    implements DvdDatabaseFactory {
    /**
     * A version number for this class so that serialization can occur
     * without worrying about the underlying class changing between
     * serialization and deserialization.
    */
    private static final long serialVersionUID = 5165L;

    public DvdDatabaseFactoryImpl() throws RemoteException {
                    //do nothing constructor
    }

    public DvdDatabaseRemote getClient() throws RemoteException {
                        return new DvdDatabaseImpl();
    }
}
```

Now we define the `DvdDatabase` remote class, which like any other Remote object, extends `java.rmi.Remote` in the interface and extends `UnicastRemoteObject` in the `Implementation` class:

```
public interface DvdDatabaseRemote extends Remote, DBClient {
}
public class DvdDatabaseImpl extends UnicastRemoteObject implements
DvdDatabaseRemote {
 ... refer to code base for the rest of implementation...
}
```

Finally, we can export the factory instance to the RMI registry found in the `register()` method in the `RegDvdDatabase` class. The `createRegistry()` method starts the Java RMI registry programmatically:

```
public static void register(int port) throws java.rmi.RemoteException, ➥
                          java.net.MalformedURLException{

             //the default rmi port is 1099.
            java.rmi.registry.LocateRegistry.createRegistry(port);

            DvdDatabaseFactoryImpl dvdFactoryImplementation
                     = new DvdDatabaseFactoryImpl();

            //register
            Naming.rebind("DvdMediator", dvdFactoryImplementation);
    }
```

■ **Note** Starting the registry programmatically is a new project requirement. Sun no longer allows starting the RMI Registry in a separate step.

Our `RegDvdDatabase` class binds the name `DvdMediator` to our remote object instance. When the client (soon to be a Swing GUI in the next chapter) requests a remote reference to our database application, it will require the services of the `DVDConnector` class. The `DVDConnector` either returns a remote object or a `DvdDatabase`, which is the database wrapper for the `DvdFileAccess` class; both are `DBClients`.

It is important to realize that we are only exporting the factory—we do not ever need to export the `DvdDatabase` remote object. In the `DvdDatabaseFactory` class we return the remote object `DvdDatabase`:

```
        public DvdDatabaseRemote getClient() throws RemoteException {
                        return new DvdDatabaseImpl();
              }
```

And the reason for the factory in the first place is the new instance of the database itself, which can be found in the constructor of our `DvdDatabase` remote implementation:

```
public DvdDatabaseImpl() throws RemoteException {
        try {
            db = new DvdDatabase();
        } catch (FileNotFoundException e) {
            throw new RemoteException(e.getMessage());
        } catch (IOException e) {
            throw new RemoteException(e.getMessage());
        }
}
```

We now have a unique instance of the DvdDatabaseImpl class for each connected client. If each DvdDatabaseImpl has its own instance of DvdDatabase, then the instance of DvdDatabase can be used to identify the client to the ReservationManager.

Since using a factory will work for both a socket solution and an RMI solution, and since we are building **both** socket connectivity and RMI connectivity in this book, we will present the ReservationManager code as though a factory solution is being used.

Let's demonstrate the use of a factory and compare it to a nonfactory scenario. Included in the downloaded code are two classes, RmiFactoryExample.java and RmiNoFactoryExample.java. As you can probably surmise from the names, one class serves as a factory implementation test case, while the other serves as a test case for a nonfactory implementation. The RmiFactoryExample class demonstrates how multiple instances of the DvdDatabase is instantiated, and the RmiNoFactoryExample class demonstrates how multiple requests result in reuse of our DvdDatabase instance.

Both sample programs extend Thread and implement the run method. The main in each sample test program also spawns two threads to simulate a multiuser environment. The only difference between the two test programs involves which remote object gets registered. The RmiNoFactoryExample registers a DvdDatabase remote object, while the RmiFactoryExample registers the DvdDatabaseFactory remote object. Correspondingly, the run methods must cast the appropriate remote interface returned from the lookup method. Refer to Listings 6-5 and 6-6.

Listing 6-5. *The Main Method in the RmiNoFactoryExample Class*

```
public static void main(String[] args) throws Exception {
    LocateRegistry.createRegistry(1099);
    Naming.rebind("RmiNoFactoryExample", new DvdDatabaseImpl());
    Thread a = new RmiNoFactoryExample("A");
    a.start();
    Thread.sleep(1000);
    Thread b = new RmiNoFactoryExample("B");
    b.start();
    a.join();
    b.join();
    System.exit(0);
}

public void run() {
    try {
        System.out.println("getting a remote handle to a DvdDatabase."
           + this.hashCode());
        DvdDatabaseRemote remote
           = (DvdDatabaseRemote)Naming.lookup("RmiNoFactoryExample");
     }
     catch (Exception e) {
        System.err.println(e);
        e.printStackTrace();
     }
}
```

The code for RmiFactoryExample is very similar, with the following differences in the main method:

```
Naming.rebind("RmiFactoryExample", new DvdDatabaseFactoryImpl());
```

Listing 6-6 shows the run method for RmiFactoryExample.

Listing 6-6. *The Main Method in the RmiNoFactoryExample Class*

```
public void run() {
       try {
            System.out.println("Getting a remote handle to a factory. " ➥
               + this.hashCode());
            DvdDatabaseFactory factory
               = (DvdDatabaseFactory)Naming.lookup("RmiFactoryExample");
            DvdDatabaseRemote worker = factory.getClient();
       }
       catch (Exception e) {
            System.err.println(e);
            e.printStackTrace();
       }
}
```

To run the RmiNoFactoryExample test program, type the following command in the classes directory of the sampleproject:

```
java sampleproject.remote.RmiNoFactoryExample
```

And to run the RmiFactoryExample, type:

```
java sampleproject.remote.RmiFactoryExample
```

Figure 6-10 shows the output of running the two test programs.

In the DvdDatabase class, an output line was added to indicate that the constructor was called. The following println statement has been inserted into the DvdDatabase constructor:

```
System.out.println(" constructing a DvdDatabase object " + this.hashCode());
```

Examining the output of the two test programs indicates a critical difference: the number of DvdDatabase objects constructed. The RmiFactoryExample constructs two separate instances of the object. The RmiNoFactoryExample constructs only one. This demonstrates how RMI reuses threads. Each thread has a separate DvdDatabase object, but since the RmiNoFactoryExample reuses one of the threads, the DvdDatabase instance is shared between the two remote invocations. Thus we cannot ensure thread safety and our locking strategy fails. (Refer to the section in Chapter 5 on locking to review why the locking strategy requires a unique DvdDatabase object.)

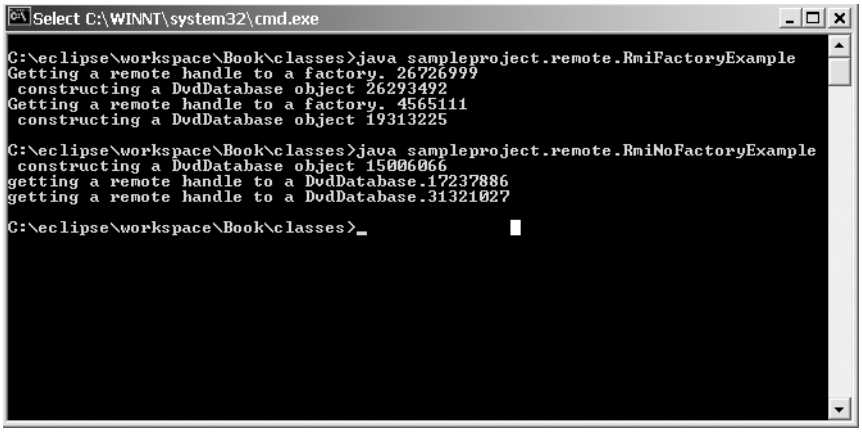


Figure 6-10. *RMI factory examples*

However, the `RmiFactoryExample` creates two separate instances of the `DvdDatabase` object, as evidenced by the two constructor output messages, for each remote invocation.

We would like to expose the public methods as remote methods so our Swing GUI can invoke them. Figure 6-11 shows a class diagram of our remote class.

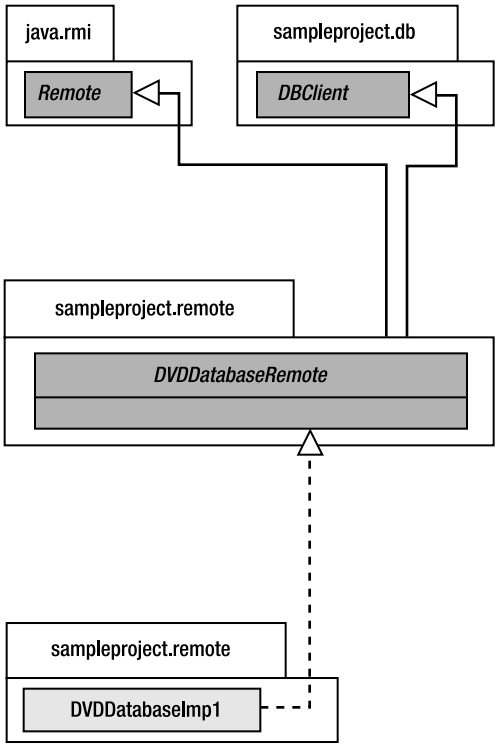


Figure 6-11. *DvdDatabaseRemote class diagram*

■ **Note** Since the advantages that `Activatable` bestows are beyond the requirements of our project, we have opted to extend the simpler, more straightforward `UnicastRemoteObject` class.

The code for the remote interface is rather concise, as shown in Listing 6-7.

Listing 6-7. *The DvdDatabaseRemote.java*

```
package sampleproject.remote;

import java.rmi.Remote;
import sampleproject.db.*;

/**
 * The remote interface for the GUI-Client.
 * Exactly matches the DBClient interface in the db package.
 *
 * @author Denny DVD
 * @version 2.0
 */
public interface DvdDatabaseRemote extends Remote, DBClient {
}
```

To understand what behavior this interface is actually defining, it's useful to examine the `DBClient` class. That class diagram is shown in Figure 6-12, and the code file is shown in Listing 6-8.

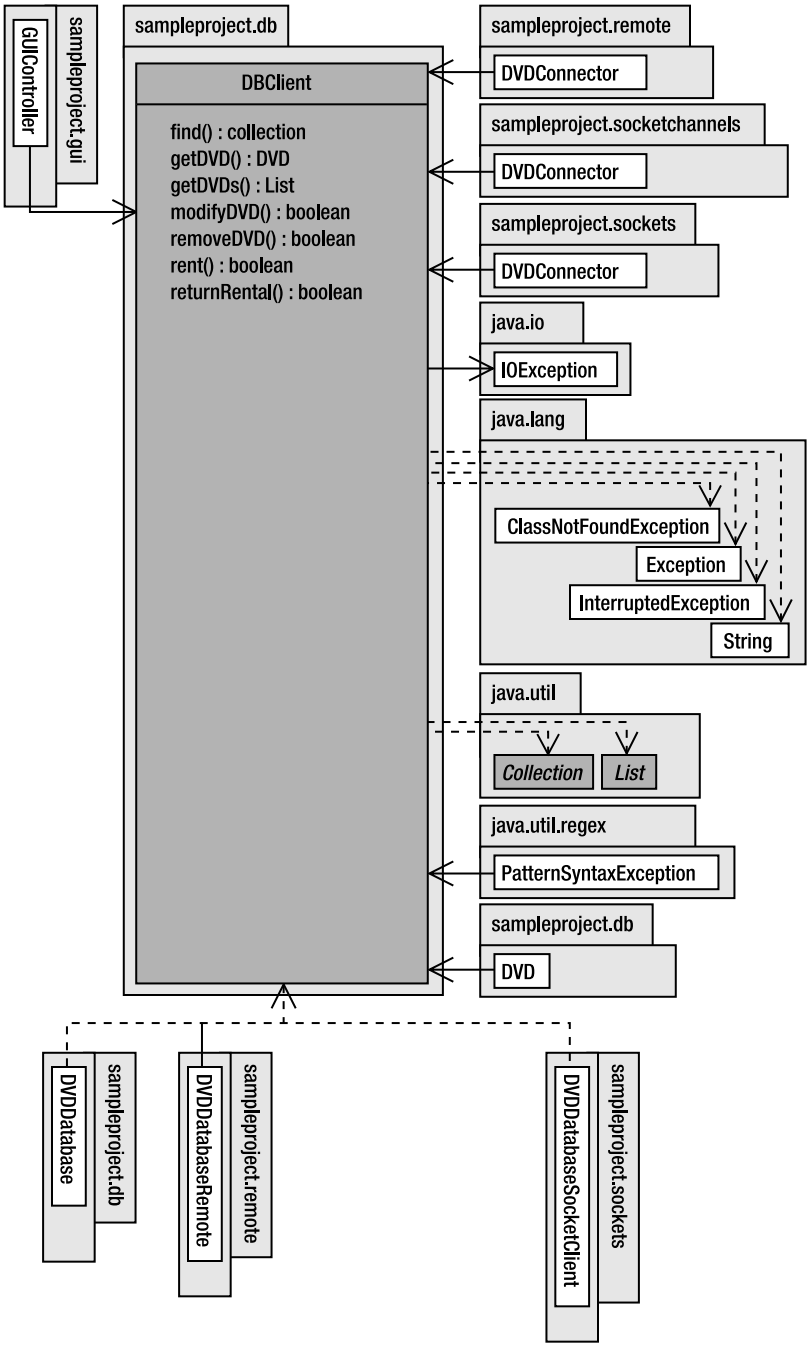


Figure 6-12. *The DBClient class diagram*

Listing 6-8. *DBClient.java*

```
public interface DBClient {
    /**
     * Adds a DVD to the database or inventory.
     *
     * @param dvd The DVD item to add to inventory.
     * @return Indicates the success/failure of the add operation.
     * @throws IOException Indicates there is a problem accessing the database.
     */
    public boolean addDVD(DVD dvd) throws IOException;

    /**
     * Locates a DVD using the UPC identification number.
     *
     * @param UPC The UPC of the DVD to locate.
     * @return The DVD object which matches the UPC.
     * @throws IOException if there is a problem accessing the data.
     */
    public DVD getDVD(String UPC)throws IOException;

    /**
     * Changes existing information of a DVD item.
     * Modifications can occur on any of the attributes of DVD except UPC.
     * The UPC is used to identify the DVD to be modified.
     *
     * @param dvd The Dvd to modify.
     * @return Returns true if the DVD was found and modified.
     * @throws IOException Indicates there is a problem accessing the data.
     */
    public  boolean modifyDVD(DVD dvd) throws IOException;

    /**
     * Removes DVDs from inventory using the unique UPC.
     *
     * @param UPC The UPC or key of the DVD to be removed.
     * @return Returns true if the UPC was found and the DVD was removed.
     * @throws IOException Indicates there is a problem accessing the data.
     */
    public  boolean removeDVD(String UPC) throws IOException;

    /**
     * Gets the store's inventory.
     * All of the DVDs in the system.
     *
     * @return A List containing all found DVDs.
```

```
     * @throws IOException Indicates there is a problem accessing the data.
     */
    public List<DVD> getDVDs() throws IOException;

     /**
      * A properly formatted <code>String</code> expressions returns all
      * matching DVD items. The <code>String</code> must be formatted as a
      * regular expression.
      *
      * @param query The formatted regular expression used as the search
      * criteria.
      * @return The list of DVDs that match the query. Can be an empty
      * Collection.
      * @throws IOException Indicates there is a problem accessing the data.
      * @throws PatternSyntaxException Indicates there is a syntax problem in
      * the regular expression.
     */
    public Collection<DVD> findDVD(String query)
            throws IOException, PatternSyntaxException;

    /**
     * Lock the requested DVD. This method blocks until the lock succeeds,
     * or for a maximum of 5 seconds, whichever comes first.
     *
     * @param UPC The UPC of the DVD to reserve
     * @throws InterruptedException Indicates the thread is interrupted.
     * @throws IOException on any network problem
     */
    boolean reserveDVD(String UPC) throws IOException, InterruptedException;

    /**
     * Unlock the requested record. Ignored if the caller does not have
     * a current lock on the requested record.
     *
     * @param UPC The UPC of the DVD to release
     * @throws IOException on any network problem
     */
    void releaseDVD(String UPC) throws IOException;
}
```

To complete an RMI implementation we need one more class. The class that actually implements the remote interface is displayed in Listing 6-9. Most of the code has been omitted for brevity, but the unexpurgated version can be viewed in the project download from the Source Code section of the Apress website. One of the important things to note is that we extend `UnicastRemoteObject` rather than `Activatable`. We also implement `DVDDatabaseRemote`. This will ensure that the proper methods are implemented.

Listing 6-9. *DvdDatabaseImpl.java*

```
public class DvdDatabaseImpl extends UnicastRemoteObject
        implements DvdDatabaseRemote {
    /**
     * A version number for this class so that serialization can occur
     * without worrying about the underlying class changing between
     * serialization and deserialization.
     */
    private static final long serialVersionUID = 5165L;

    /**
     * The Logger instance. All log messages from this class are routed through
     * this member. The Logger namespace is <code>sampleproject.remote</code>.
     */
    private static Logger log = Logger.getLogger("sampleproject.remote");

    /**
     * The database handle.
     */
    private DBClient db = null;

    /**
     * DvdDatabaseImpl default constructor
     * @throws RemoteException Thrown if a <code>DvdDatabaseImpl</code>
     * instance cannot be created.
     */
    public DvdDatabaseImpl() throws RemoteException {
        try {
            db = new DvdDatabase();
        } catch (FileNotFoundException e) {
            throw new RemoteException(e.getMessage());
        } catch (IOException e) {
            throw new RemoteException(e.getMessage());
        }
    }

    /**
     * Returns the sampleproject.db.Dvd object matching the UPC.
     *
     * @param upc The upc code of the DVD to retrieve.
     * @return The matching DVD object.
     * @throws RemoteException  Thrown if an exception occurs in the
     * <code>DvdDatabaseImpl</code> class.
     * @throws IOException Thrown if an <code>IOException</code> is
     * encountered in the <code>db</code> class.
     * <br>
     * For more information, see {@link DvdDatabase}.
```

```
 * @throws ClassNotFoundException Thrown if a
 * <code>ClassNotFoundException</code> is
 * encountered in the <code>db</code> class.
 * <br>
 * For more information, see {@link DvdDatabase}.
 */
public DVD getDVD(String upc) throws RemoteException, IOException {
    return db.getDVD(upc);
}

/**
 * Gets the store's inventory.
 * All of the DVDs in the system.
 *
 * @return A collection of all found DVDs.
 * @throws IOException Indicates there is a problem accessing the data.
 * @throws ClassNotFoundException Indicates the Dvd class definition cannot
 * be found.
 */
 public List<DVD> getDVDs() throws IOException {
    return db.getDVDs();
}

/**
 * A properly formatted <code>String</code> expressions returns all matching
 * DVD items. The <code>String</code> must be formatted as a regular
 * expression.
 *
 * @param query A regular expression search string.
 * @return A Collection of DVD objects that match
 * the search criteria.
 * @throws IOException Thrown if an IOException is
 * encountered in the db class.
 * @throws ClassNotFoundException Thrown if an
 * ClassNotFoundException is encountered in the
 * db class.
 * @throws PatternSyntaxException Thrown if a
 * PatternSyntaxException is encountered in the
 * db class.
 */
 public  Collection<DVD> findDVD(String query)
        throws IOException, PatternSyntaxException {
    return db.findDVD(query);
 }

/**
 * Modifies a DVD database entry specified by a DVD object.
```

```
     *
     * @param item The DVD to modify.
     * @return A boolean indicating the success or failure of the modify
     * operation.
     * @throws RemoteException  Thrown if an exception occurs in the
     * <code>DvdDatabaseImpl</code> class.
     * @throws IOException Thrown if an <code>IOException</code> is
     * encountered in the <code>db</code> class.
     * <br>
     * For more information, see {@link DvdDatabase}.
     */
    public boolean modifyDVD(DVD item) throws
                                       RemoteException,
                                       IOException {
        return db.modifyDVD(item);
    }

    /**
     * Removes a DVD database entry specified by a UPC.
     *
     * @param upc The UPC number of the DVD to remove.
     * @return A boolean indicating the success or failure of the removal
     * operation.
     * @throws RemoteException  Thrown if an exception occurs in the
     * DvdDatabaseImpl class.
     * @throws IOException Thrown if an IOException is
     * encountered in the db class.
     * <br>
     * For more information, see {@link DvdDatabase}.
     */
    public boolean removeDVD(String upc) throws
                                          RemoteException,
                                          IOException {
        return db.removeDVD(upc);
    }

    /**
     * Lock the requested DVD. This method blocks until the lock succeeds,
     * or for a maximum of 5 seconds, whichever comes first.
     *
     * @param upc The UPC of the DVD to reserve
     * @throws InterruptedException Indicates the thread is interrupted.
     */
    public boolean reserveDVD(String upc)
            throws InterruptedException, IOException {
        return db.reserveDVD(upc);
    }
```

```
    /**
     * Unlock the requested record. Ignored if the caller does not have
     * a current lock on the requested record.
     *
     * @param upc The UPC of the DVD to release
     */
    public void releaseDVD(String upc) throws IOException {
        db.releaseDVD(upc);
    }

    /**
     * Adds a DVD to the database or inventory.
     *
     * @param dvd The DVD item to add to inventory.
     * @return Indicates the success/failure of the add operation.
     * @throws IOException Indicates there is a problem accessing the database.
     */
    public boolean addDVD(DVD dvd) throws IOException, RemoteException {
        return db.addDVD(dvd);
    }
}
```

This `DVDDatabaseImpl` source code is interesting since it accesses the database via a wrapper, or an adapter, called the `DvdDatabase`. We could have easily placed the actual database method implementations, found in the class `DvdFileAccess`, directly in the remote object `DvdDatabaseImpl`. Or we could have extended `UnicastRemoteObject` on `DVDFileAccess` and made that object the remote object. We chose to do it this way for the following reasons:

- Adding a level of abstraction between the RMI implementation and the actual database-level code makes it a clearer object-oriented design and helps to separate the RMI details from the application logic details. We wanted to create a level of separation from the application logic and the two networking approaches: RMI and sockets. With the preceding design, we will be able to easily use either the socket or the RMI code depending on our preferences. We have eliminated any dependencies between the networking packages and the database packages. This will be clarified in the project wrap-up in Chapter 8.

- In order for our locking strategy to work properly, it is important that all database modification requests first achieve a lock prior to the modification identified with a unique `DvdDatabase` object. We require a unique instance of `DvdDatabase` in order to have a unique identifier for the lock method; otherwise we cannot ensure that the client who locked a record is the same one who is modifying, deleting, or even unlocking that same record. Once the lock is granted, the modification may safely occur. Finally we unlock the record. Because these operations must not occur in a synchronized method (see the "Locks" section in Chapter 4), and all of the modification methods in the database must be synchronized, we must access the database by way of our adapter.

All RMI implementations must be thread-safe because invocations on a particular object could occur concurrently if the RMI runtime spawns multiple threads. There is no guarantee regarding RMI thread management. Thus, it is your responsibility to make sure that remote object implementations are thread-safe. In our case, using an RMI factory pattern in collaboration with the `reserveDVD` and `releaseDVD` methods in the `DvdDatabase` class ensures that our implementation is thread-safe.

If you choose not to extend `UnicastRemoteObject` but rather call the class method `UnicastRemoteObject.export` in the constructor, then the object methods such as `hashcode`, `toString`, and `equals` must be implemented. Listing 6-10 shows how the class constructor and declaration would accommodate that approach (the rest of the code has been omitted for conciseness).

Listing 6-10. *Exporting UnicastRemoteObject*

```
public class DvdDatabaseImpl implements DvdDatabaseRemote
{
        public DvdDatabaseImpl() throws RemoteException {
                UnicastRemoteObject.exportObject(this);
             this.dvdDatabase = new DvdDatabase();
        }
}
```

Stubs and Skeletons

Only clients actually invoke methods on a *stub*, which is a local representation, or proxy, of the remote object. While it appears that the client is calling the remote object directly, unbeknownst to the client it is actually calling a proxy method in the stub that initiates communication with the destination VM. The stub is responsible for packaging the parameters of the remote method call prior to sending them across the network. The process of packaging the parameters prior to shipping them across a network is called *marshaling*.

On the other side of the wire, or server side of the network, the incoming invocation is eventually received by the remote reference layer, which demarshals the arguments and dispatches the call to the actual server object. The server then marshals the return value back to the caller on the client side.

J2SE 1.5 introduces a new feature that alleviates the need to separately generate stub classes for the client using the tool `rmic`. Now stub classes can be dynamically generated at runtime by the JVM. (However, we are still required to use `rmic` for stub generation in our submission.) To unconditionally generate stubs dynamically, set the JVM parameter `java.rmi.server.ignoreStubClasses` to true. This can be done using the `-D` option (i.e., `java -Djava.rmi.server.ignoreStubClasses=true`).

■ **Note** You must still generate stub classes with `rmic` for backward compatibility with pre-J2SE 1.5 clients.

The tool `rmic` can be used in a number of ways depending on context. `rmic` allows for compatibility with previous versions of Java such as Java 1.1 (stubs and skeletons) and Java 1.2 (stubs only). J2SE 1.5 does not require the use of skeletons, but as we mentioned earlier, you must still use `rmic` with your project submission.

■ **Note** The skeleton interface has been deprecated in the Java 2 platform (i.e., JDK version 1.2 and greater). To create stubs for version 1.2 only in our RMI implementation, we use the following syntax with `rmic`:

```
rmic -v1.2 sampleproject.remote.DVDDatabaseImpl
```

■ **Note** RMI makes use of the Proxy design pattern. The Proxy pattern designates a surrogate object that controls access to the real object. There are three key players in this design pattern: a `Subject`, a `Proxy`, and a `Real-Subject`. The Proxy object has a reference to the `Real-Subject` and is responsible for communication to and from the `Real-Subject`. The `Subject` displays the same interface for both the `Proxy` and the `Real-Subject`. The `Real-Subject` is the actual implementation of the object the `Proxy` represents and that the `Subject` has defined. In RMI, the `Subject` corresponds to the remote implementation. The stub is the `Proxy` and the remote object implementation is the `Real-Subject`. Figure 6-13 illustrates the various elements in the Proxy design pattern.

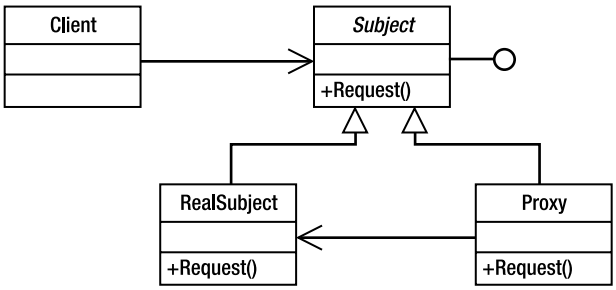


Figure 6-13. *The Proxy design pattern*

For backward stub//skeleton compatibility with previous versions of Java, you can read more about `rmic` at `http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/rmic.html` for Windows. For Unix, visit `http://java.sun.com/j2se/1.5.0/docs/tooldocs/solaris/rmic.html`.

Parameter Passing

RMI method invocations require communication between objects on different machines and in different JVMs, even though the call behaves as if it is a local call within the same JVM. This section describes how RMI transfers parameters and return values between different

JVMs. There are three classes of parameters and return values to consider: primitive types, `Serializable` objects, and remote objects.

■ **Note** Arrays of primitives and arrays of `Serializable` objects can also be safely passed as a remote method parameter.

Primitive Types

RMI makes a copy of the primitive type for both method arguments and return values. When a copy of the primitive is sent across the wire, we say that the parameter or return value is *pass-by-value*.

Serializable Objects

When an object is marshaled across the wire as a method parameter or a return value, RMI makes a copy of the object and all of the objects it references as one large object graph. Object parameters are sent across the wire as pass-by-value, just like primitive types.

The object being passed must be `Serializable` or a `NotSerializableException` will be thrown. In addition, all of the class members of the object being marshaled must also be `Serializable`, unless the member is declared as a transient member or the `Externalizable` interface is implemented instead of `Serializable`.

Remote Objects

This is the most complex scenario of the three. When the item being returned is a remote object (i.e., one that implements `java.rmi.Remote`), the proxy for that object, or the stub, is returned in place of the `Real-Object`. In this way, remote objects are pass-by-reference and the stub is marshaled back and forth.

Remote objects, by default, implement `Serializable`. Table 6-3 summarizes how RMI handles various parameters and return values. RMI uses one of two mechanisms to obtain remote object references. The RMI registry is one way, and another is simply passing the proxy or stub, which was discussed previously. The RMI registry uses `java.lang.Naming` to store references to remote objects. `java.rmi.Naming` provides URL-based methods to associate name-object pairs located on a particular host and port. RMI can also load bytecodes with a valid URL naming protocol such as FTP or HTTP. When a client interacts with a remote object, it also interacts with the remote object interface, not with the actual remote implementation.

Table 6-3. *RMI Parameter Passing*

Object/Type	Parameter/Return Value
Remote objects	Pass-by-reference: A stub is passed instead of a copy.
Nonremote/serialized objects	Pass-by-value: An actual copy of an object is passed as a serialized object.
Primitive types	Pass-by-value: A copy is made.

Security and Dynamic Loading

Running an RMI application over a network requires that certain files be accessible to the server and client class loaders. Table 6-4 summarizes which files are required.

Table 6-4. *Class Loader Requirements*

Client Class Loader Requires	Server Class Loader Requires
Remote interfaces	Remote interfaces, remote implementations
Stubs for remote objects	Stubs for remote objects
Server classes used as return values	Skeletons for remote objects (JDK version 1.1 only)
Miscellaneous client classes	Miscellaneous server classes

When a client has the remote object stubs and class files locally, the task is much simpler and there are no special runtime considerations. However, if a client only has the remote interface, Java provides the capability to load the classes and stubs dynamically. Dynamic loading is the feature in RMI that allows for an object not available locally to be retrieved. Two important classes make dynamic loading possible: `RMIClassLoader` and `SecurityManager`. Besides these two important classes, you need to know the location of the classes to be loaded.

This is referred to as a code base. Think of a code base as similar to a classpath, except on a different machine accessible with a URL. A classpath would be analogous to a local code base. The `codebase` property is `java.rmi.server.codebase`. It would be specified on the command line as follows:

```
java -Djava.rmi.server.codebase <URL>
```

The `java.rmi.server.RMIClasssLoader` class is required for dynamic class loading when using RMI.

The `RMIClassLoader` class makes use of the `codebase` property. The `RMIClassLoader` class can load classes from either an applet or an RMI application. For our purposes, and the purposes of the certification exam, our discussion on security will be brief. We do not need to worry about security or policy files in our implementation, since our application does not use a security manager. Java 2 makes use of a security manager option due to compatibility with pre–Java 2 applications. However, in a professional RMI application, a security manager is a given. In addition, if your network application uses dynamic class loading, a security manager is required.

Firewall Issues

When the client and server are separated by a firewall, the RMI transport layer is forbidden from creating socket connections. In situations where a socket connection is prohibited, RMI makes use of a technique called HTTP tunneling. Tunneling is the process of wrapping RMI calls in an HTTP POST request, which firewalls typically allow.

Tunneling is done automatically. When a socket connection is denied by the transport layer, a last-ditch effort is made to service the request via HTTP tunneling.

■ **Note** Java programs have permissions that are governed by policy files. The Java 2 security model requires that programs and the RMI registry have permission to create sockets. After all, RMI is based on sockets. The client also needs to specify a policy file on start-up. Here is how we would start the RMI registry to load a policy file other than the default: `rmiregistry -J-Djava.security.policy=ourSecurityFile.policy`.

Port 80 is used as the default port unless one is specified in the server property `http.proxyHost`. However, HTTP tunneling comes with a number of costs. Performance is sacrificed and security can be compromised. Tunneling can be disabled by setting the server property `java.rmi.server.disableHttp` to true.

Summary

This chapter covered a great deal of information. First, we reviewed serialization.

Any object that can potentially be sent across a network must be serialized prior to its trip across the wire. Both sockets (which will be discussed in the next chapter) and RMI require serialization. In RMI, both the return values and the method parameters must be serializable for the application to work.

Finally, we explored RMI in depth and implemented an RMI solution for Denny's DVDs. Considering the amount of effort that went into building the socket implementation, we were able to appreciate the amount of work RMI does under the hood. We did not have to worry about writing a multithreaded server, since RMI handles that aspect for us. We also did not have to worry about creating command and result objects in order to implement an application protocol.

However, RMI did require us to define a remote interface and make sure that all of our method parameters were serializable. Additionally, we had to concern ourselves with registering our remote object with the RMI naming service and starting the RMI registry so that our object is accessible to a remote client. We hope you have learned enough about serialization, sockets, and RMI so that you can design and implement a complete networking solution for the SCJD exam. Using this project networking package as a guide, along with the chapter, you should be adequately prepared.

FAQ

Q Is RMI thread-safe?

A No. The J2SE 1.4 RMI specification makes no guarantee regarding the number of threads that will have access to your remote object. For this reason, your design has to take into account issues of thread safety. See Chapter 4 for details on locking and thread safety. The approach we used in this chapter was to create an RMI factory ensuring a distinct instance of the `DvdDatabase` wrapper class and using that as our unique identifier for record locking.

Q Which method is better, RMI or sockets?

A Each approach has its strengths. Sockets are useful for sending large amounts of data (with or without a protocol) without a lot of overhead. With RMI it is much easier to implement a multithreaded server, since RMI handles the threading aspect for you. The remote object also behaves as if it is a local object complete with a protocol or set of public methods, whereas sockets require that you implement a protocol. It is really up to you, but both approaches should work fine as long as you understand the key technical issues pertaining to the protocol.

Q Is a security manager, or some sort of authentication, required?

A This depends largely on the details of your exam. Read the instructions very carefully! Each test is different. In this chapter, we briefly discussed a number of security issues such as policy files, dynamic class loading, HTTP tunneling, firewalls, and security managers. However, in our implementation, we do include a security manager and do not make use of policy files.

Q Do I need to include the skeletons in my J2SE 5.0 implementation of RMI?

A Yes. Skeletons are only required for those Java versions prior to 1.2 (i.e., version 1.1). By default, skeletons are generated when `rmic` is run on your remote object implementations, but you can control this feature by using the `-keep v1.2` option of `rmic`. However, at the time of this writing, the certification project still requires that stubs be included with your submission and that you do not rely on the dynamic stub generation technique in the Tiger version (i.e., J2SE 5.0).

Q Am I required to start the RMI registry manually?

A No. The registry must be started programmatically. Again, read the instructions on your exam. But most likely, you will be required to start the registry as we did in the `RegDVDDatabase` class. The class has a `register` method that binds the remote object implementation class in the RMI naming service and uses the `LocateRegistry.register` method. When running this across a network, as we will in Chapter 8, you should call the `getRemote` method in `Connector` from a `main` method.

CHAPTER 7

Networking with Sockets

This chapter describes how to build the Denny's DVDs networking layer by using sockets.

Sockets are a software abstraction that allows applications to communicate with each other across a network. Any connection between two machines over IP (Internet Protocol) requires the use of sockets. Java provides programmers with the ability to connect application sockets, but as we will see in this chapter, much work needs to be done to ensure that clients and servers can communicate effectively using standard sockets. To alleviate some of that effort, Java also provides the Remote Method Invocation (RMI) framework, which is built on top of sockets. As you learned in the previous chapter, however, using RMI introduces its own problems. Either networking solution has its pros and cons, and it is up to you as a developer to determine which one you feel is better for a given situation. In this chapter, we will detail a simple sockets solution for our sample project, providing you with what you need to make an informed decision regarding which solution to use in your Sun assignment.

Our major areas to cover include

- An overview of sockets
- Why you would want to use sockets
- The basic information required for connecting sockets
- How to build a TCP socket client
- How to build a TCP socket server
- Serialized objects with sockets
- Socket lifecycles

Socket Overview

A *socket* is one of the endpoints in a communication link between two programs and is always bound to a port. A socket connection is useful for connecting to remote machines, and sending and receiving data. To create a socket connection, you will need the host address to which you want to connect and the port number. Sockets are not specific to Java. However, all versions of Java since JDK 1.0 have provided facilities for creating sockets. As with RMI, the discussion is demarcated by client and server considerations.

The server is a listening service that receives connection requests from socket clients. When a request is granted, the server establishes a socket-to-socket connection. The client

will use the server's hostname (or IP address) and port to request a connection. On the server side of the network, `java.net.ServerSocket` is the class that listens for connection requests from socket clients. The class `java.net.Socket` is one of the endpoints of a socket connection. Both the server and the client will have an instance of `java.net.Socket` per socket connection.

Why Use Sockets

Choosing between sockets and RMI is one of the biggest decisions you will have to make for the certification project. But each choice comes with its own advantages and trade-offs. If the JavaRanch web forum on the certification exam is any indication, it seems that most developers opt for the RMI solution. In fact, in the first edition of this book we recommended using RMI instead of sockets just because we believed at the time that the approach seemed more intuitive from a Java programmer's perspective. However, since then we have backed off from this assertion. Either RMI or sockets will work fine and both are reasonably straightforward to implement. But for the adventurous at heart, let's discuss some of the reasons you might opt for a socket solution.

Socket servers are more scalable and faster than RMI servers. Acquiring a remote object handle requires a network call in the form of a Registry lookup. In addition, each remote object has a client-side proxy, or stubs, which effectively adds an object layer between the client and the actual remote object implementation class. This communication overhead is paid for in terms of performance.

Typically one of the reasons mentioned for *not* using sockets is the implicit contract, or application protocol, that needs to be in place between the socket clients and the socket server. We discuss this in more detail later in this chapter in the section "The Application Protocol." However, what if the protocol is very simple? In cases where the socket interface is simple, sockets are an excellent choice.

One final reason worth considering a socket implementation involves threading. Each socket client request in our socket solution spawns a new thread, which we can use to maintain a lock on the `DvdDatabase`. This means we do not have to worry about implementing a factory as we did with RMI. The very nature of our socket solution avoids this potentially complicating issue. Of course, you now have to deal with the problem of developing a multithreaded socket server, but help is on the way! Denny's DVDs is a multithreaded socket server, and we describe our server implementation later in this chapter.

Socket Basics

In this section, we will explore the types of sockets available to the Java programmer as well as some of the fundamental concepts related to socket development.

Addresses

An address is a unique number used to identify a device connected to a network, much like a postal address identifies the location of a building in a city. An IP address is a unique number, usually consisting of 12 digits or more, that is technically a 32-bit or 128-bit unsigned number used by the Internet protocol (i.e., IP) and for sending messages between socket addresses on TCP- or UDP-based networks. The `java.net.InetAddress` class is the Java class that encapsulates an IP address for use with Java sockets.

■ **Note** The 32-bit addressing scheme used by TCP/IP has been around since the 1970s. With more and more devices requiring Internet addresses, it is estimated that all currently available addresses will be used somewhere between 2016 and 2023 (however, previous estimates on Internet growth have been far below reality, so this figure is not to be relied on). To alleviate this problem, a new addressing scheme has been recommended and is slowly being implemented. The 32-bit addressing scheme is known as IPv4, and the new addressing scheme is known as IPv6. Java supports both schemes with the `Inet4Address` class and `Inet6Address` classes, respectively. In this chapter we will be using IPv4 as this is the most common addressing scheme currently in use.

TCP and UDP Sockets Overview

Over the years programming has evolved from manually setting binary instructions (1GL or first-generation languages), through assembly language (2GL), human-readable languages (Java, C++), and specification languages like SQL (4GL). The later languages do not eliminate the need for the lower-level languages—they just do the hard work of translating the programmer's code into lower-level code. For example, when programming in Java your code will be translated into bytecode by the Java compiler (corresponding roughly to the output from an assembly language program), which is later translated into the binary instructions for the computer by the JVM. While you can still find work as a machine language programmer, many programmers find that such work is slow and error prone—working in a higher-level language such as Java can greatly improve productivity and quality of work.

Similarly, network protocols have evolved over the years, so that we no longer need to worry ourselves with the lowest-level network details. When you are programming your Java program, you typically do not care how to physically send signals over a piece of wire (or other medium), nor do you care whether your computers are communicating over an Ethernet or a Token Ring network; all you care about is how to make the connection. Finally, in order to use sockets, we will be using IP rather than one of the other network layers (X.25, ICMP, IPX). But even having chosen to use sockets, leaving all the low-level communication to the IP layer and layers beneath it, we still have a choice between TCP and UDP sockets.

UDP Sockets

User Datagram Protocol (UDP) enables machines on a network to send datagram packets to each other. UDP sockets work in much the same way as the U.S. postal system. Messages, or letters, are sent to a particular address and an immediate response by the server, or addressee, endpoint is not necessary. UDP sockets do not require a two-way connection. Similarly, the postal system does not require that the recipient be present when a letter is delivered.

Let's consider the mail analogy a little more. A letter is dropped off at a mailbox and delivered to the stated address. The mail carrier then places the letter in the destination mailbox. The mail system does not require that someone be there to accept the letter (of course, I realize that sometimes packages require signatures, but for the sake of convenience let's ignore those kinds of packages for now). Once the individual to whom the letter is addressed returns home, he or she can then collect the mail.

UDP discards corrupted messages (datagrams) so there is no guarantee that messages sent via UDP will make it to their final destination, just as there is no guarantee that a letter

sent through the postal system will make it to the intended recipient. The main classes used are `DatagramSocket` and `DatagramPacket` in the Java API. A *packet* is information in the form of byte sequences that are sent across a network. Since messages are not guaranteed to be delivered and can be received out of order, UDP socket applications have to deal with reordering and data loss.

Efficiency and flexibility are the main reasons you would choose a UDP socket approach. We do not use UDP for a few reasons. First, efficiency is not something that is an overriding concern. Second, and most important, the exam does not permit a UDP approach. So that sort of makes our decision rather clear-cut. But if we were to use a UDP approach, then both the client and the server would use a `DataGramSocket` class for sending and receiving `DataGramPacket` objects. The remainder of this chapter and the sample project use TCP sockets instead of UDP sockets, so our discussion of UDP will end here.

■ **Caution** The exam requirements tend to be vague regarding this point. Most likely the instructions read "You must use either serialized objects over a simple socket connection, or RMI." It is not precisely clear as to what a simple socket is and whether this precludes a UDP solution. We take it to mean that you should use the `java.net.Socket` class and not the `java.net.DatagramSocket` class. While it would be technically feasible to create a UDP solution for this assignment, doing so would require a lot of work in developing packet-handling code, and would go against the standard expected usage of UDP (simple, short [often less than 100 bytes] messages). We therefore recommend that, if you're using sockets as the networking protocol, you use the simpler TCP approach. However, that being said, you should always refer to your specific exam for instructions.

UDP is designed for minimal messaging applications—the very small messages that you might use if you were trying to monitor network applications, such as SNMP (Simple Network Management Protocol)—or for very simple query/responses, such as DNS (Domain Name System) queries. It is better to use TCP for larger messages, such as the large messages that are possible in response to a database query in our sample application. The next section describes how to develop a socket solution using TCP.

TCP Sockets

TCP stands for Transmission Control Protocol—a protocol that controls the transmission of packets so that messages are guaranteed to be received and that packets are received in the same order as they were sent. To meet these guarantees, the TCP protocol uses slightly more network traffic than UDP (and since RMI provides more features and guarantees than a plain TCP socket, it uses even more network traffic than TCP). However, since the protocol provides these features for us, we do not have to verify the order in which packets are received in our application. This makes it a much better choice for Denny's DVDs.

A TCP socket is a socket connection that utilizes a TCP/IP connection as its underlying transfer protocol. Each end of the connection is identified with an IP address and a port number. TCP socket clients send requests, and TCP socket servers listen for requests.

The following are the basic steps involved in TCP socket communication:

1. Create an instance of a socket class.
2. Send serialized messages using the socket I/O streams.
3. Close the socket connection.

■ **Note** For the initial connection there are two requirements: a socket client that sends the initial connection request and a socket server that is listening for incoming connection requests. But once the connection has been set up, you have a connection between two sockets, and either one can send or receive data. It is up to the communication protocol to determine which socket is sending data and which socket is receiving data at any given time. We will cover application protocols in the section "The Application Protocol" later in this chapter. As you might imagine, if either client or server does not follow the application protocol, the point-to-point communication will break down.

TCP Socket Clients

The class `java.net.Socket` is responsible for implementing a socket-to-socket connection.

The constructor for this class, which actually attempts to connect to the destination machine, requires both the hostname as a URL (as a string) and the port of the destination machine. Table 7-1 lists the various public constructors for `java.net.Socket`.

Table 7-1. *Public Constructors for java.net.Socket*

Socket Class	Socket Class Constructors
`public Socket()`	Creates an unconnected socket.
`public Socket(InetAddress address, int port)`	The most commonly used. Creates a connected socket to the supplied address and port.
`public Socket(InetAddress address, int port, InetAddress localAddr, int localPort)`	Creates a socket that connects to the remote address and port and also binds to the local address and port.
`public Socket(String host, int port)`	Creates a stream socket and connects it to the specified host and port.
`public Socket(String host, int port, InetAddress localAddr, int localPort)`	Connects to a remote host and port and binds to the local address and port.

Once a socket connection is made, you can obtain information about the connection through a variety of getter methods. The client-side socket is not bound to the port number the server is residing on. Rather, the client-side socket is assigned a local port it uses to communicate with the server.

A number of constants are defined in the interface `SocketOptions`. These constants are used as settings to customize a client socket. We do not implement the `SocketOptions` interface directly, but we have access to its constants through getter and setter methods in the `Socket` class. In the sections that follow, we discuss a few of the more useful options.

Using SO_TIMEOUT

When reading data from a socket through the use of the input stream's `read` method, the call blocks other requests as determined by this setting. The `SO_TIMEOUT` option determines the amount of time in milliseconds that blocking operations can block until they time out. If the operation does not complete in the allotted time, then a `java.net.SocketTimeoutException` is thrown, as shown in Figure 7-1. If this option has not been set or is set to 0, then the call blocks requests until complete and will not time out.

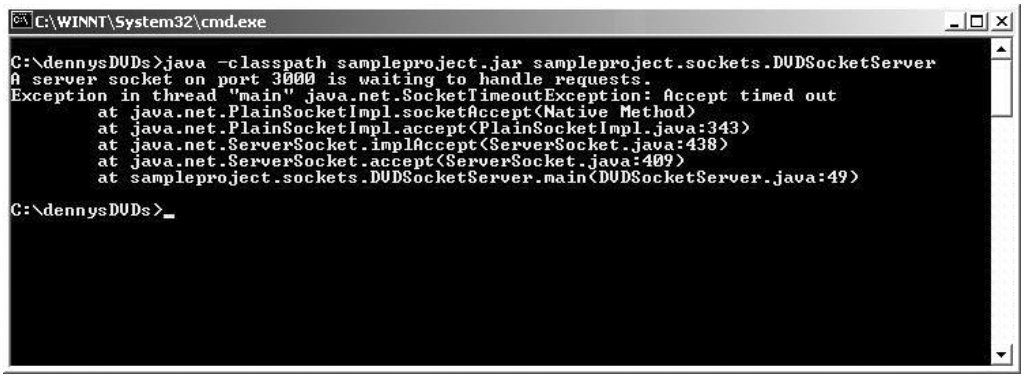


Figure 7-1. *A SocketTimeoutException*

The `Socket` methods used for setting and getting this option are as follows:

```
public void setSoTimeout(int timeout) throws SocketException
public int getSoTimeout() throws SocketException
```

We use this option in `DVDSocketServer.java`, although we could just as easily have used this option in our socket client, `DVDSocketClient.java`. The following line will shut down the socket server after 1 minute, or 60,0000 milliseconds, of inactivity:

```
serverSocket.setSoTimeout(60000);
```

After the specified time elapses, a `java.net.SocketTimeoutException` is thrown. Figure 7.1 illustrates the `DVDSocketServer` console when the exception occurs.

Using SO_SNDBUF

This option sets the buffer size for data transmissions from this socket. When setting this option, it is merely a suggestion, or a hint, to the platform about the size of buffer the application requires for output operations over the socket. The getter indicates the actual size of the output buffer. These two methods can be used together to determine whether the platform

was able to make use of the hint in the set method by a call to the get method after the output operation. The Socket methods used for setting and getting this option are as follows:

```
public int getSendBufferSize() throws SocketException
public void setSendBufferSize(int size) throws SocketException
```

The send buffer size value determines how many packets will be sent before waiting for an acknowledgment that the packets have been received at the other application. On a reliable network, you can set this to a high value to get the best throughput of data. In a worst-case scenario, you could set the send buffer to the same size as the packet size. In this case, for every packet sent, an acknowledgment would have to be received—if you needed to send 100 packets, you would need to receive 100 packets. Contrast this with a send buffer that is 100 times larger than the packet size—when sending the 100 packets you would have to receive only one packet; it would complete in nearly half the time.

Using SO_RCVBUF

This option is similar to SO_SNDBUF, which is used for setting the send buffers. The difference here is that this option deals with the receive buffers. As with the send buffer option, setting the receive buffer size value is actually a hint to the platform. To verify the size the buffer was set to, use the getReceiveBufferSize method. To improve the performance of a high-volume socket connection, increase the size of the receive buffer. Decreasing this value can reduce incoming backlog.

```
public void setReceiveBufferSize(int size) throws SocketException
public int getReceiveBufferSize() throws SocketException
```

Using SO_BINDADDR

This option indicates the local address that a socket is bound to. This option is read-only. A socket address cannot be changed after it is created and must be specified in the constructor. The method signature is as follows:

```
public InetAddress getLocalAddress()
```

One of the possible uses for this method is determining what IP address you are bound to when running on a computer with multiple IP addresses (e.g., a server with multiple network cards, a PC with both a local IP address and a networked address, or a computer with both a network address and a virtual private network address).

The DvdSocketClient

There are a few important points to note about the DVDSocketClient. The socket implementation in our sample project uses port 3000 by default. Also by default, we use localhost as the hostname. This is useful when we want to run our application in networking mode on one machine (i.e., when we do not have access to a network). If we run our application on different machines, we will have to specify the IP address of the machine that is running the socket server.

Another key point regarding the socket client is that it implements `DBClient`. This ensures that our `rent` and `return` methods are available to any client using `DVDSocketServer`. The `DvdSocketClient.java` is displayed in Listing 7-1. The entire source code for the sockets package can be downloaded from the Apress web site (`http://www.apress.com`) in the Source Code section—more details on what can be downloaded and how the code can be compiled and packaged are listed in Chapter 9.

■ **Tip** On Linux and most Unix computers, you can also specify `localhost.localdomain` as the hostname. You can also use the `loopback` addresses (127.0.0.0–127.0.0.254) instead of the hostname. Accepted practice is to use 127.0.0.1 as the standard loopback address, leaving the other potential `loopback` addresses for specialist purposes (e.g., for testing multiple identical applications on different addresses simultaneously).

Listing 7-1. *DVDSocketClient.java*

```
public class DvdSocketClient implements DBClient {
    /**
     * The socket client that gets instaniated for the socket connection.
     */
    private Socket socket = null;
    /**
     * The outputstream used to write a serialized object to a socket server.
     */
    private ObjectOutputStream oos = null;
    /**
     * The inputstream used to read a serialized object (a response)
     * from the socket server.
     */
    private ObjectInputStream ois = null;
    /**
     * The IP address of the machine the client is going to attempt a
     * connection.
     */
    private String ip = null;
    /**
     * The port number we will be connecting on.
     */
    private int port = 3000;
    /**
     * Default constructor.
     */
    public DvdSocketClient () throws UnknownHostException, IOException {
        this("localhost", "3000");
    }
```

```
/**
 * Constructor takes in a hostname of the server to connect.
 *
 * @param hostname The hostname to connect to.
 * @throws NumberFormatException if portNumber is not valid.
 */
public DvdSocketClient(String hostname, String portNumber)
        throws UnknownHostException, IOException {
    ip = hostname;
    this.port = Integer.parseInt(portNumber);
    this.initialize();
}

/**
 * Adds a DVD to the database or inventory.
 *
 * @param dvd The DVD item to add to inventory.
 * @return A boolean value that indicates the success/failure of the
 * add operation.
 * @throws IOException Indicates there is a problem accessing the data.
 */
public boolean addDVD(DVD dvd) throws IOException {
    DvdCommand cmdObj = new DvdCommand(SocketCommand.ADD, dvd);
    return getResultFor(cmdObj).getBoolean();
}

/**
 * Gets a <code>DVD</code> from the system using a UPC.
 *
 * @param upc The UPC of the DVD you want to view.
 * @return A DVD that matches the supplied UPC.
 *
 * @throws IOException Indicates there is a problem accessing the data.
 */
public DVD getDVD(String upc) throws IOException {
    DVD dvd = new DVD();
    dvd.setUPC(upc);

    DvdCommand cmdObj = new DvdCommand(SocketCommand.GET_DVD, dvd);
    return getResultFor(cmdObj).getDVD();
}

/**
 * Attempts to rent the DVD matching the provided UPC.
 *
 * @param upc is the UPC of the DVD you want to rent.
 * @return true if the DVD was rented. false if it cannot be rented.
```

```
     *
     * @throws IOException Thrown if an <code>IOException</code> is
     * encountered in the <code>db</code> class.
     * <br>
     * For more information, see {@link DVDDatabase}.
     * @throws InterruptedIOException Thrown if an
     * <code>interrupted Threading exception</code> is
     * encountered in the <code>db</code> class.
     * <br>
     * For more information, see {@link DVDDatabase}.
     */
    public boolean rent(String upc) throws IOException {
        DVD dvd = new DVD();
        dvd.setUPC(upc);

        DvdCommand cmdObj = new DvdCommand(SocketCommand.RENT, dvd);
        return getResultFor(cmdObj).getBoolean();
    }

    /**
     * Attempts to return the DVD matching the provided UPC.
     *
     * @param upc The UPC of the DVD you want to rent.
     * @return true if the DVD was rented. false if it cannot be rented.
     *
     * @throws IOException Thrown if an <code>IOException</code> is
     * encountered in the <code>db</code> class.
     * <br>
     * For more information, see {@link DVDDatabase}.
     * <br>
     * For more information, see {@link DVDDatabase}.
     */
    public boolean returnRental(String upc) throws IOException {
        DVD     dvd    = new DVD();
        dvd.setUPC(upc);

        DvdCommand cmdObj = new DvdCommand(SocketCommand.RETURN, dvd);
        return getResultFor(cmdObj).getBoolean();
    }

    /**
     * Gets the store's inventory.
     * All of the DVDs in the system.
     *
     * @return A collection of all found DVD's.
     * @throws IOException Indicates there is a problem accessing the data.
     */
```

```
public List<DVD> getDVDs() throws IOException {
    DvdCommand cmdObj = new DvdCommand(SocketCommand.GET_DVDS);
    return getResultFor(cmdObj).getList();
}

/**
 * A properly formatted <code>String</code> expressions returns all matching
 * DVD items. The <code>String</code> must be formatted as a regular
 * expression.
 *
 * @param query A regular expression search string.
 * @return A <code>Collection</code> of <code>DVD</code> objects that match
 * the search criteria.
 * @throws IOException Thrown if an <code>IOException</code> is
 * encountered in the <code>db</code> class.
 */
public Collection<DVD> findDVD(String query)
        throws IOException, PatternSyntaxException {
    DvdCommand cmdObj = new DvdCommand(SocketCommand.FIND);
    cmdObj.setRegex(query);

    DvdResult serialReturn = getResultFor(cmdObj);

    if (serialReturn.isException()
            && serialReturn.getException() instanceof PatternSyntaxException) {
        throw (PatternSyntaxException) serialReturn.getException();
    } else {
        return serialReturn.getCollection();
    }
}

/**
 * Removes a <code>DVD</code> from the system using a UPC.
 *
 * @param upc The UPC of the DVD you want to remove from the database.
 * @return true if the item was removed, false if it was not removed.
 * @throws IOException Indicates there is a problem accessing the data.
 */
public boolean removeDVD(String upc) throws IOException {
    DVD     dvd    = new DVD();
    dvd.setUPC(upc);

    DvdCommand cmdObj = new DvdCommand(SocketCommand.REMOVE, dvd);
    return getResultFor(cmdObj).getBoolean();
}
```

```
    /**
     * Modifies a DVD database entry specified by a DVD object.
     *
     * @param dvd The DVD to modify.
     * @return A boolean indicating the success or failure of the modify
     * operation.
     * @throws IOException Thrown if an <code>IOException</code> is
     * encountered in the <code>db</code> class.
     * <br>
     * For more information, see {@link DVDDatabase}.
     */
    public boolean modifyDVD(DVD dvd) throws IOException {
        DvdCommand cmdObj = new DvdCommand(SocketCommand.MODIFY, dvd);
        return getResultFor(cmdObj).getBoolean();
    }

    /**
     * Lock the requested DVD. This method blocks until the lock succeeds,
     * or for a maximum of 5 seconds, whichever comes first.
     *
     * @param UPC The UPC of the DVD to reserve
     * @throws InterruptedException Indicates the thread is interrupted.
     * @throws IOException on any network problem
     */
    public boolean reserveDVD(String upc) throws IOException,
InterruptedException {
        DVD dvd = new DVD();
        dvd.setUPC(upc);
        DvdCommand cmdObj = new DvdCommand(SocketCommand.RESERVE, dvd);
        return getResultFor(cmdObj).getBoolean();
    }

    /**
     * Unlock the requested record. Ignored if the caller does not have
     * a current lock on the requested record.
     *
     * @param UPC The UPC of the DVD to release
     * @throws IOException on any network problem
     */
    public void releaseDVD(String upc) throws IOException {
        DVD dvd = new DVD();
        dvd.setUPC(upc);
        DvdCommand cmdObj = new DvdCommand(SocketCommand.RELEASE, dvd);
        getResultFor(cmdObj).getBoolean();
    }
```

```
    private DvdResult getResultFor(DvdCommand command) throws IOException {
//        this.initialize();
        try {
            oos.writeObject(command);
            DvdResult result = (DvdResult) ois.readObject();
            Exception e = result.getException();

            if (! result.isException()) {
                return result;
            } else if (e instanceof ClassNotFoundException ) {
                IOException ioe = new IOException(
                     "problem with demarshaling DvdCommand)");
                ioe.setStackTrace(e.getStackTrace());
                throw ioe;
            } else if (e instanceof IOException) {
                throw (IOException) e;
            } else {
                // well, we still have an exception, but it is up to the
                // calling method to handle it
                return result;
            }
        } catch (ClassNotFoundException cnfe) {
            IOException ioe = new IOException(
                 "problem with demarshaling DvdResult)");
            ioe.setStackTrace(cnfe.getStackTrace());
            throw ioe;
//        } finally {
//            closeConnections();
        }
    }

    public void finalize() throws java.io.IOException {
        closeConnections();
    }

    /**
     * A helper method which initializes a socket connection on specified port
     *
     * @throws UnknownHostException if the IP address of the host could not be
     *         determined.
     * @throws IOException Thrown if the socket channel cannot be opened.
     */
    private void initialize() throws UnknownHostException, IOException {
        socket = new Socket(ip, port);
```

```
        oos = new ObjectOutputStream(socket.getOutputStream());
        ois = new ObjectInputStream(socket.getInputStream());
    }

    /**
     * A helper method which closes the socket connection.
     * Needs to be called from within a try-catch
     *
     * @throws IOException Thrown if the close operation fails.
     */
    private void closeConnections() throws IOException {
        oos.close();
        ois.close();
        socket.close();
    }
}
```

Socket Servers

In this section, we discuss the various types of socket servers and the techniques often used to build them. We also introduce the concept of an application protocol and how we implemented the protocol in the Denny's DVDs socket server.

Multicast and Unicast Servers

A unicast server involves one-to-one communication between a client and a server. The sockets developed in Denny's DVDs are unicast. However, in addition to unicast sockets, there are multicast sockets.

Multicast sockets are connections between groups of machines. Multicasting, or broadcasting, is used when data is sent from one host to multiple clients. The class `java.net.MulticastSocket` is used when writing applications that broadcast messages to many clients. We will not discuss multicast sockets any further, but you should at least be familiar with the concept. Figure 7-2 highlights the differences between multicast and unicast sockets.

Multitasking

An iterative, or single-threaded, server is one that handles client requests sequentially. As one request comes in, the server responds and begins listening again once the request is completed. Client requests that come in while the server is processing a prior request are placed on a queue and serviced as prior requests are completed. In addition, as long as one client maintains the port connection to a single-threaded server, no other client can connect—if the connection is held for too long, some of the waiting clients may time out. (Refer to the `SO_TIMEOUT` setting earlier.) Figure 7-2 shows the lifecycle of an iterative, or sequential, socket server.

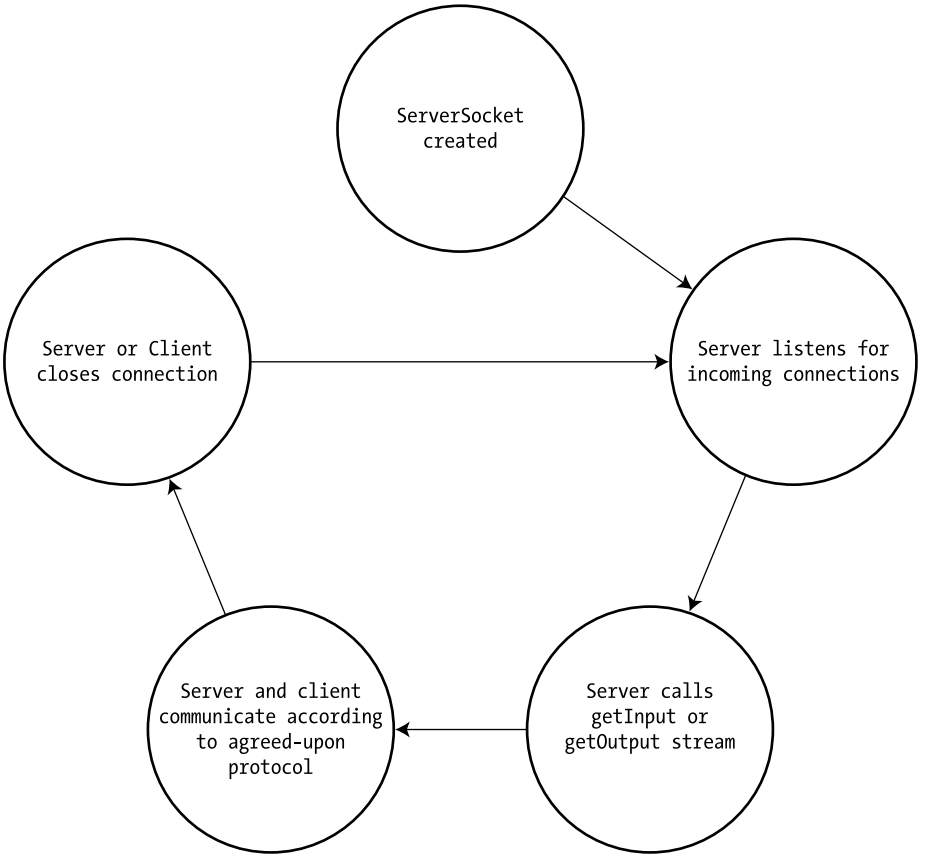


Figure 7-2. *Iterative socket server lifecycle*

Socket servers that can multitask can process multiple client requests simultaneously. This is accomplished with a multithreaded socket server. If you are anticipating a lot of requests, or requests take a long time to process, then multitasking is a very useful feature for a socket server to have. The Denny's DVDs socket version is a multithreaded socket server.

Each client gets serviced with its own thread, which makes the socket solution operate much like our RMI factory solution. We can thus be assured that each client has its own `DvdDatabase` object. (See our discussion on the RMI factory in Chapter 6.) The next section describes the details of our multitasking socket solution.

The Server Socket Class

The class `java.net.ServerSocket` allows you to create servers that listen to incoming connections on a specified port. A `ServerSocket` can send and receive data as well as act upon that data. Figure 7-3 illustrates the lifecycle of a multithreaded server. The public constructors for `ServerSocket` let you specify the port, queue length, and binding address. The server is responsible for implementing the protocol, or the rules that both the client and server must

conform to in order to communicate. If the port you try to open is being used when you try to instantiate your `ServerSocket`, then an `IOException` is thrown. If the socket is successfully created, the server listens on that port for incoming connections by using the `accept` method. The `ServerSocket.accept` method returns the client socket that is connected to the client.

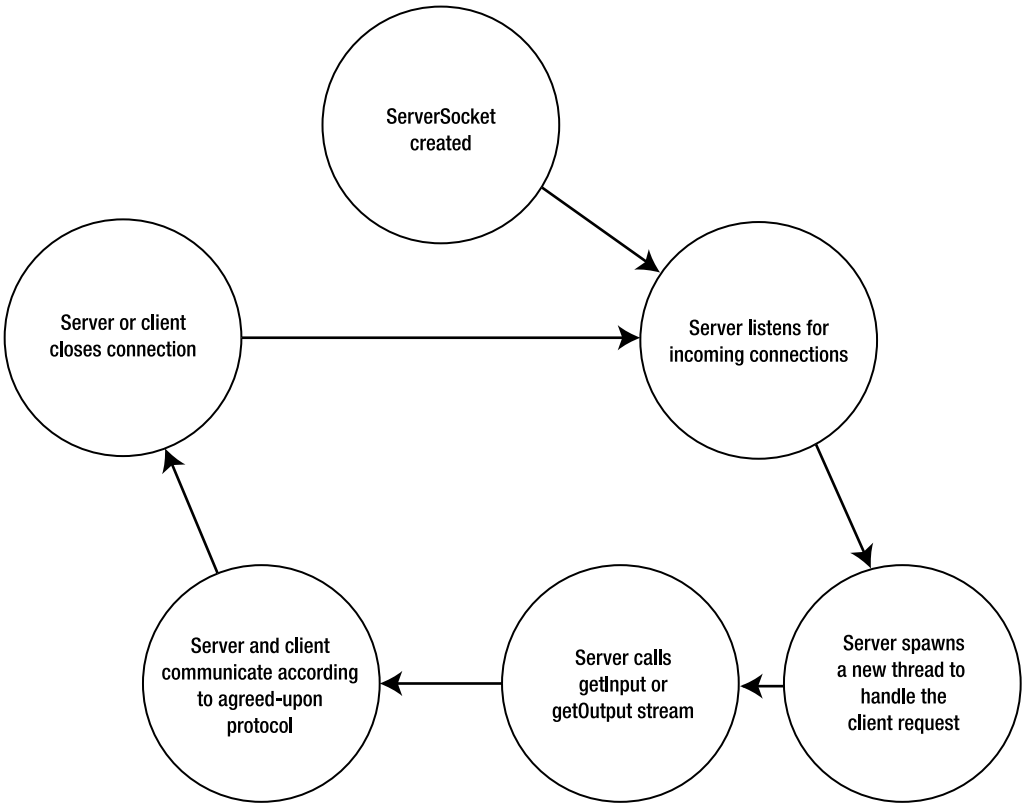


Figure 7-3. *Lifecycle of a multithreaded socket server*

■ **Note** More accurately, if the port you try to open is already being used when you try to instantiate your `ServerSocket` connection, a `BindException` will be thrown. However, the constructor for `ServerSocket` only specifies the superclass of `BindException` (`IOException`) will be thrown, as other exceptions may also be thrown during construction.

Next, the server opens readers and writers on the socket and communicates with the client by writing and reading to the socket. The `ServerSocket.accept` method blocks the port until a client connects, and then it returns a `Socket` object. The socket connection is then used to execute the client request, and the connection can be closed for that client.

SocketServer has many of the same socket options available as the Socket class. One important option that they share is the SO_TIMEOUT option. With the ServerSocket class, the SO_TIMEOUT option specifies how long the server should wait for an incoming socket connection with the accept method.

When the accept method times out, a java.net.SocketTimeoutException is raised. Let's look at a sample socket implementation in Listing 7-2 for our networking version of Denny's DVDs. Figure 7-4 shows a high-level view of how we must implement our socket solution.

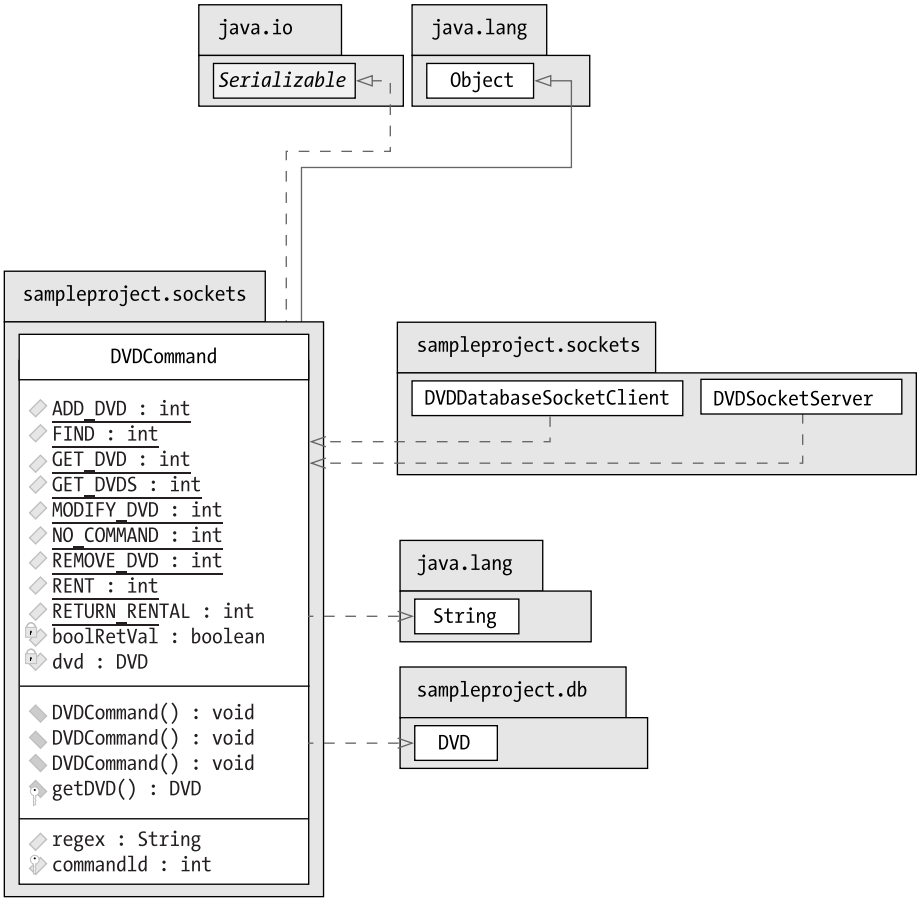


Figure 7-4. *Socket classes*

Listing 7-2. *DVDSocketServer.java*

```
package sampleproject.sockets;
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.logging.*;
import sampleproject.db.*;
```

```
/**
* DVDSocketServer is the class that handles socket client requests and
* passes the request to the database. The class recieves parameters in
* <code>DVDCommand</code> objects and returns results in
* <code>DVDResult</code> objects.
* @version 2.0
*/
public class DvdSocketServer extends Thread {
    private String dbLocation = null;
    private int port = 3000;

    /**
     * Starts the socket server
     *
     * @param argv Command line arguments.
     * @throws IOException Thrown if the socket server fails to start.
     */
    public static void main(String argv[]) {
        register(".", 3000);
    }

    public static void register(String dbLocation, int port) {
        new DvdSocketServer(dbLocation, port).start();
    }

    public DvdSocketServer(String dbLocation, int port) {
        this.dbLocation = dbLocation;
        this.port = port;
    }

    public void run() {
        try {
            listenForConnections();
        } catch (IOException ioe) {
            ioe.printStackTrace();
            System.exit(-1);
        }
    }

    public void listenForConnections() throws IOException {
        ServerSocket aServerSocket = new ServerSocket(port);
        //block for 60,000 msecs or 1 minute
        aServerSocket.setSoTimeout(60000);

        (Logger.getLogger("sampleproject.sockets")).log(Level.INFO,
                "a server socket created on port " +
                aServerSocket.getLocalPort());
```

```
        while (true) {
            Socket aSocket = aServerSocket.accept();
            DbSocketRequest request = new DbSocketRequest(dbLocation, aSocket);
            request.start();
        }
    }
}
```

The main method creates a `ServerSocket` object on port 3000. There is a loop that listens for requests from any Java or non-Java clients on port 3000. Once a request is received, the connection is accepted, meaning that the `accept` method stops blocking and the `Socket` object is returned. The resulting socket is passed in as a parameter to the `DBSocketRequest`. The server socket then spawns a new thread for the client request. This design enables multiple clients to connect to the socket server since each request is serviced in a separate thread.

The `DBSocketRequest` class extends `Thread`. You will recall that one of the requirements is that multiple clients need to be able to use the `DVDDatabase` services.

Now our socket server can create multiple threads as needed. Once an object is accepted on the port (port 3000 in our case), which happens in the `run` method, the `execCmdObject` method is called. This method is a big `switch` statement. It inspects the `command` object for the action to be performed and then calls the matching method in `DVDDatabase`. Refer to the project download for the entire `DBSocketRequest` class. For brevity, Listing 7-3 only shows the `run` method.

Listing 7-3. *DBSocketRequest.java*

```
/**
* Required for a class that extends thread, this is the main path
* of execution for this thread.
public void run() {
    try {
        ObjectOutputStream out =
        new ObjectOutputStream(client.getOutputStream());
        ObjectInputStream in =
            new ObjectInputStream(client.getInputStream());
        DVDCommand cmdObj = (DVDCommand) in.readObject();
        out.writeObject(execCmdObject(cmdObj));
        if (client != null) {
            client.close();
        }
        out.flush();
    }
    catch (SocketException e) {
        logger.log(Level.SEVERE,
         "SocketException in Socket Server: " + e.getMessage());
    }
    catch (Exception e) {
        logger.log(Level.SEVERE,
           "General Exception in Socket Server: "
           + e.getMessage()
    );
}
```

A final point about DBSocketRequest is that DBSocketRequest is where we implemented our application protocol. We have separated the protocol from the actual socket, DVDSocketServer.

The Application Protocol

The socket client transmits a serialized object to the socket server. But how does the server know how to respond to the object? A serialized object is technically just data in the form of a byte stream; on the surface it does not communicate action. This is where we need a *protocol*, or set of rules, that defines how our client is to interact with our server.

At a high level, here is how our protocol will work:

1. The client will make a request, such as rent or return rental, of the server.
2. The server will execute the request.
3. The result status or return value will be sent to the client.

Deliberating on the preceding list should lead to the following two questions: "How will the server interpret the request?" and "How will the result be sent back to the client?"

The Denny's DVDs application adopts an approach of encapsulation. The request is encapsulated in a command object and the result is encapsulated in a result object. Let's take a closer look at the command and result objects.

The Command Enum

The DvdCommand class encapsulates the client request by storing it as a SocketCommand member, commandId. When a GUI client calls one of the DBClient methods on DVDSocketClient, the socket client sets the commandId property and sends it off to the socket server, DVDSocketServer. Since DVDCommand is sent, or marshaled, across the wire, it must be serializable. When the server receives the DVDCommand object, it uses the commandId to call the corresponding method on the DVDDatabase, which is a local call to our server. Any parameters (for instance, the UPC value) that are required for the request are passed in the DVD class member. The regex attribute is used exclusively for the find method.

Listing 7-4 shows the constructors that take a SocketCommand enum ID (see the sidebar "Using Enum Constants") and a dvd object as a parameter. The dvd object is useful for storing the UPC for rent and return. For the modify method, a dvd parameter can be used to set the other dvd attributes for a particular DVD. We do not actually use the modify method publicly in our implementation, but the class has been designed with this enhancement in mind.

■ **Note** The DVDCommand object is an example of the Command pattern. A command object encapsulates a request as an object. You can find more information about the Command pattern in the book *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides (Addison-Wesley, 1995). The Proxy and Adapter patterns are also described in this literary software masterpiece.

Listing 7-4. *The Public Methods of DVDCommand.java*

```
/**
     * Default constructor
     */
    public DvdCommand() {
        this(SocketCommand.UNSPECIFIED);
    }

    /**
     * Constructor that requires the type of command to execute as a parameter.
     *
     * @param command The id of the command the server is to perform.
     */
    public DvdCommand(SocketCommand command) {
        this (command, new DVD());
    }

    /**
     * Constructor that requires the type of command and the DVD object.
     *
     * @param command The id of the command the server is to perform.
     * @param aDvd
     */
    public DvdCommand(SocketCommand command, DVD dvd) {
        setCommandId(command);
        this.dvd = dvd;
    }

    /**
     * Gets the query that was used for searching.
     *
     * @return The string representing the regualr expression to use in find().
     */
    public String getRegex() {
        return regex;
    }

    /**
     * Sets the regular expression
     *
     * @param re The regular expression to use in find().
     */
    public void setRegex(String re) {
        regex = re;
    }
```

DVDSocketClient makes use of the DVDCommand object in each of its DBClient methods. For example, in the find method, the command object is constructed with the FIND enum value of SocketCommand. Next, the regex attribute is set using the query parameter of the find method. These two steps are shown again here:

```
DvdCommand cmdObj = new DvdCommand(SocketCommand.FIND);
cmdObj.setRegex(query);
```

The Result Object

The DVDResult object is used to encapsulate the result of the request. When the DVDDatabase has completed a request, the result is sent back to DVDSocketServer, which in turn wraps the result in a DVDResult object and sends it back to the socket client. As in the case of the command object, DVDResult must be serializable since it is sent across the network. Even exceptions are wrapped in DVDResult and sent back to the client.

The result object is similar to the DVDCommand object in its operation. There are a number of constructors that take the varying types of return values from the DBClient methods. You can find the code in the project download, but the following is an example of its usage in the find method of DVDSocketClient:

```
DVDResult serialReturn = (DVDResult)ois.readObject();
if (!serialReturn.isException()) {
retVal = serialReturn.getCollection();
}
```

The DVDResult is received from the socket server. The result object is checked for exceptions, and if no exceptions are detected, the Collection is extracted from DVDResult. We know it must be a Collection since our protocol ensures that for DBClient's find method, a Collection is always returned.

■ **Note** Our application protocol was separated from our socket server, based on good design principles. Thus, changes in our protocol, or even additional protocols, can now be handled without affecting the socket server directly.

USING ENUM CONSTANTS

The Application protocol is implemented by specifying the command type. This was performed with the following code taken from the constructor of the DvdCommand class:

```
public DvdCommand(SocketCommand command, DVD dvd) {
    setCommandId(command);
    this.dvd = dvd;
}
```

Prior to JDK 5, the commandId variable in the DvdCommand class would probably have been constants, of the form:

```
public final static int FIND = 0;
public final static int RENT = 1;
public final static int RETURN = 2;
```

There are many problems with doing this, though, including the possibility that someone might directly set the commandId variable to an integer value that is not supported, or they might do something illogical with the constants (such as try to add them). Furthermore, we cannot enumerate over the number of modes, and if we tried to print the value of the commandId variable, a number would be returned—we would have to look up this number in the documentation or in the source code to determine what the number really means.

JDK 5 provides us with a simpler way of defining these constants: using an enum. The SocketCommand enum is defined as

```
package sampleproject.gui;

public enum SocketCommand {
    UNSPECIFIED, /* indicate that the command object has not been set */
    FIND,          /* request will be performing a Find action */
    RENT,         /* renting a DVD */
    RETURN,         /* returning a DVD */
    MODIFY,         /* updating status of a DVD */
    ADD,          /* creating a new DVD record */
    REMOVE,         /* delete a DVD record */
    GET_DVD,     /* retrieve a single DVD from database */
    GET_DVDS,     /* retrieve multiple DVDs from database */
    RESERVE,
    RELEASE

}
```

Now any variable of type SocketCommand can *only* contain one of these listed options—no other options are possible.

We also have the benefit that anytime we print (or log) the contents of the commandSocket variable, the string UNSPECIFIED, FIND, RENT, RETURN, MODIFY, ADD, REMOVE, GET_DVD, GET_DVDS, RESERVE, or RELEASE will be printed (or logged)—it will be instantly clear from looking at the output what the variable was set to.

There are many more benefits of using enums. We recommend you read the release notes related to enums available at http://java.sun.com/j2se/1.5.0/docs/guide/language/enums.html for more information.

Summary

In this chapter, we discussed the Denny's DVDs socket implementation. We also provided a brief overview of the different types of sockets, and the various types of TCP socket development strategies. We laid out the application protocol for our socket implementation and demonstrated the Command pattern with the help of Java enums. The choice of sockets as the network protocol in your exam solution should not be perceived as something esoteric and frightening that is to be avoided at all costs. Even though most students opt to not develop a socket solution, we believe that the choice isn't any more challenging than an RMI solution and should be fairly straightforward, given the sample code base that accompanies this text. Sockets are a technology that underlies most networking protocols. Most of the cool new technologies, such as web services and EJBs, ultimately rely on sockets. As we have discussed, even RMI is built on top of sockets. So you would be well served to become acquainted with sockets, if only to help deepen your understanding of these other networking technologies. The next chapter covers the graphical user interface (GUI) for the Denny's DVDs application.

FAQs

Q Do I need to implement a multithreaded socket server?

A Yes. The certification exam requires that the server allow multiuser access. If you use a single-threaded solution, then there will never be a way to demonstrate concurrent access. Your application will block serially until each request is resolved one at a time. This may be an interesting solution and will circumvent the need to figure out a locking strategy, but one that Sun will not permit.

Q Should I utilize a thread pool for the socket server implementation?

A This is completely up to you. You could do so and it would be a good practice. Every time a new connection is accepted, our server spawns a new thread to handle the request. However, creating and destroying threads is not a free lunch. On a really busy, highly trafficked server, it might be better to control the creation of threads. A thread pool will instantiate a set number of threads, a number that can be calibrated based on your application performance requirements, upon startup. When the server receives a new request, it is serviced with one of the threads from the pool. When the request is completed, the thread is returned to the pool for later use. A thread pool will eliminate the overhead associated with creating and destroying new threads. We do not make use of a thread pool in Denny's DVDs and you will not have to implement a thread pool for your certification project since the exam does not require that your server perform well under a heavy load.

Q What is meant by TCP sockets being a connection-oriented protocol?

A Sometimes in the literature, the term "connection-oriented" will be used when referring to TCP sockets. What this means is that before communication can occur between the endpoints, a connection must exist. Contrast this with UDP sockets, a connectionless protocol.

Q Was it necessary to use the `SocketCommand` object to indicate which command to perform on the database?

A No. We chose this implementation for clarity: it was a nice way to enumerate over the possible commands and demonstrate the command pattern. The drawback is that if new commands were added to the database, the client would have to receive an updated `SocketCommand` class in order to submit future requests since we would no longer be able to deserialize older command objects on the server. (Of course, we could check the `serialversionID` so that we could be backward compatible, but you get the idea.) A different approach would be to embed the command as a string in the `DvdCommand`. This way, as new commands are added, there would be no need to add them to the `SocketCommand` class, but the server would be able to recognize the new command.

Q Should I allow the user to change the port to be used in this application?

A While not strictly necessary, doing so is generally considered a good idea. Otherwise, if another server application is using your desired port, your server application will be unable to start.

Q When choosing a port number to use in my application (whether hard-coded or a default value), are there any numbers I should avoid?

A The Internet Assigned Numbers Authority (IANA) specifies a list of well-known ports (from port number 0 through 1023), registered ports (from 1024 through 49151), and dynamic and/or private ports (from 49152 through 65535). It is recommended that you choose a port number from the private port range to avoid conflict with any other service. You should avoid using a well-known port since, depending on your operating system, you may also find that you cannot run your server using a well-known port without administrator privileges.

Q How can I perform system cleanup when a client disconnects?

A If the thread that is dedicated to that client is listening for a new command from the client, then it will receive an exception when the client disconnects. Alternatively, if the client disconnects before your server has an opportunity to respond to a previous request, then the thread dedicated to that client will receive an exception when you attempt to send the response to the client. In either of these cases, you can add cleanup code to your catch block.

Alternatively, if you are only interested in cleaning up any outstanding locks, you can use the thread dedicated to the client as a key within a `WeakHashMap` containing the locks. When the client disconnects (and the thread dies), then eventually the lock will be automatically removed from the `WeakHashMap`. Refer to Chapter 5 for more information.

Q How can I automatically update all clients whenever a booking is made by any other client on the server?

A This is not required for the Sun assignment; however, you would have to open an additional socket between the server and the client so that the server can send messages to the client. You could combine this with the Observer design pattern (described in Chapter 8) so that clients can register their desire to be notified of bookings.

CHAPTER 8

The Graphical User Interfaces

In the preceding chapters, the implementation of the data and network tiers for the Denny's DVD application were discussed in detail. Now it is time to deal with the final development tier: the graphical user interface (GUI).

In this chapter we will cover the following topics:

- Designing simple yet usable GUIs
- Reviewing Swing components and the event model
- Implementing a `JTable`
- Implementing the Model-View-Controller (MVC) design pattern
- Implementing the Observer design pattern

It is not necessary to have read the networking chapters prior to reading this chapter. The majority of this chapter details how to design your GUI, and as such, the networking sections are not required. The sections of this chapter that deal with connecting to the database use the Factory design pattern, which will provide an instance of a class that implements our connection interface. Naturally, though, you will not be able to connect the GUI to the database using the networking options described in this chapter until after you have completed the networking chapters.

Tip The way we are using the connection interfaces shows one of the benefits of using an interface: the interface provides a contract that we can assume will be implemented—we don't need the actual implementation to develop our factory and use it.

While developing the Sun assignment, you might choose to start development of the GUI prior to developing the networking classes. This is also a reasonable approach, and might even be the preferred approach in real life—developing the GUI before the networking code means that the customer could start working with the stand-alone application before the final code is written.

For the purposes of this book, it made sense for us to develop the networking code first, as we will be connecting to the database via direct connection and via the various networking options from within our GUI.

■ **Caution** A common misconception made by end users is that when they have seen the user interface, then the entire project is close to completion. It can save a lot of confusion if you spend extra time with the client to make sure that they understand how much additional work is required at the time you show them the user interface. One attempt to get around this misconception is the Napkin Look & Feel for Swing applications. See `http://napkinlaf.sourceforge.net` for further details.

Of all components in an application, none affects the user quite as much as the GUI. This is true by an interface's nature: It is the method by which an end user interacts with a system. Unfortunately, the GUI is often the most de-emphasized part of the application development process. This is truly a fallacy, since a user interface can sometimes make or break an entire system. If a GUI is convoluted and difficult to navigate, users will quickly become frustrated and the result will be a poor overall user experience. In the end, a system is only successful if people can successfully use it.

This chapter aims to introduce concepts of GUI layout, design, and implementation to those who may not have given it much thought in the past. An interface is a required portion of the SCJD exam, and this chapter provides novices with all the information they need to get one up and running.

GUI Concepts

The SCJD exam requires that one individual complete all development work. This includes development of all three tiers of the application: the server tier, the middle tier, and the presentation tier. This is different from the more common working environment, where developers often specialize in a certain tier. It is common practice in the workforce to have a group of back-end developers and a separate group of front-end developers working on a three-tiered project. In the case of the SCJD exam, one developer must create all three tiers.

■ **Tip** The fact that you are doing the work that might normally be split into three teams can cause confusion when reading Sun's instructions. Often candidates feel that requirements in the GUI section of the instructions contradict instructions in the Data section. But when considered from the perspective of individual teams, you should find that the instructions do not contradict each other; the team building the `Data` class has clear instructions to carry out. The team building the GUI has been told what the `Data` class will provide, and can then add their own restrictions to meet the requirements of the GUI.

Creating the front-end can be a daunting task, especially for those who are primarily not front-end developers. There are many Java developers who never write a line of interface code but who are still extremely fluent programmers in general. This is further exacerbated by the fact that only a limited textual description of the interface is provided; we have not been given any examples of what the end user would like to see. The crux of the next two sections is to ease the burden of developing the GUI.

Layout concepts are discussed at a high level, in addition to some basic human interface concepts that can be incorporated into the interface for the SCJD exam.

■ **Note** Layout concepts and human interface concepts are courses of study unto themselves. The next two sections only introduce ideas of relevance to the SCJD exam.

Layout Concepts

It is a common misconception that GUI layout is more an art than a science. This is a slight misrepresentation of the process of layout design and information architecture. In the early days of computing, computer programming was considered more of an art than a science. This fact seems almost ludicrous by today's standards. It was only after an engineering process was applied to the art of programming that it quickly became more of a science or an engineering practice. The same can be said for interface design concepts.

Layout design revolves around the presentation of information in a clear and concise manner. The user should be able to process the information a user interface presents with little or no effort. This last statement is vague, but for a reason. No matter how standards oriented a process may become, there is always a level of intuition in decision making. Not all roads can be anticipated and laid out in a standard process. Intuition is where the art of layout design comes into play, in deciding how to deal with those gray areas of development where no one has treaded before.

Fortunately for us, many have treaded into the design considerations presented in the SCJD exam. In fact, the layout of data has become so standardized that Sun actually includes a Swing component, the `JTable`, to deal specifically with this "database"-style data. Sun even goes so far as to require the use of the `JTable` component for the SCJD exam. The `JTable` component, which is discussed in detail later in this chapter, is the main method of data presentation in this book's example application.

On a high level, let's take a look at how the `JTable` efficiently displays data. The `JTable` uses the spreadsheet paradigm of presenting items divided into rows and columns. This spreadsheet-style layout is illustrated in Figure 8-1. Notice that when users need to locate data under this paradigm, they only need to cross-reference a column value with a row value. Where the two values intersect is the location of the data. The paradigm also is the ultimate reconciliation of data versus display space. If another row of data is added to the table, the size will increase by approximately the height of the font used. Adding an additional column impacts the size of more than a row, but the layout is extremely flexible, so updates to the underlying system data are easily represented in the presentation. In addition to all of these features, the table schema presents a great quantity of data in a very small area.

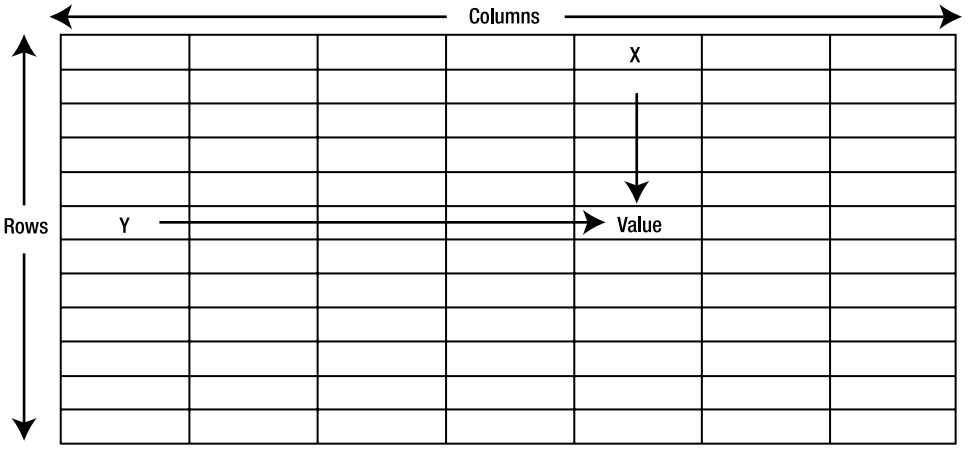


Figure 8-1. *A high-level table schema*

While the preceding analysis may appear to be stating the obvious, it does emphasize the layout principles the table schema does very well. The analysis performed on the `JTable` can and should be done for the overall layout of any GUI interface. You should look for these criteria in a user interface:

- All data is presented in the minimum amount of space, without being cluttered or disorganized.
- The user should be able to locate necessary information quickly.
- In most cases, the interface must scale to fit more or less data.

Human Interface Concepts

Human interface concepts go beyond simple data layout organizational techniques. Human interface design schemas organize the user's flow of events when attempting to complete a task. For instance, deciding how to lay out data in order to maximize its readability is a *layout* decision. Deciding how to lay out the entire act of selecting a data item and modifying it is a *human interface* decision. In short, human interface decisions determine how users interact with the system.

The point of a GUI is to create an easy method of interaction between a user and a system. The GUI, in a certain sense, acts as a layer of shielding, abstracting the actual functions of the system away from the user. When the user clicks a button to save a document, the system may have to go through a number of steps to save the file:

1. The application validates the file integrity.
2. The system calculates the physical size of the file.
3. The system checks the hard disk to see if there is enough available space to save the file.
4. The application writes the file to the file system.

These are steps almost every application takes in order to save a file. But most applications require only one item of input from the user to initiate these actions: selecting the Save option from the File menu.

Imagine if an application required the user to select a Validate File menu item, followed by a Calculate File Size option, followed by a Check Hard Drive Space option; to then enter the calculated file size obtained from the previous step; and finally, after confirming that there is enough hard drive space, to select a Write File to File System option. This application would make saving a document more trouble than it is worth.

The previous example is extreme, but it still demonstrates the notion of abstracting the operations of the system away from the user as much as possible. The system should be able to complete many of the tasks listed previously without any input from the user. The only time the user needs to be notified of the system's activities is if an error should occur. For instance, if there is not enough space on the file system to save the file, the user should be prompted and notified that the system can no longer complete this step on its own. Beyond these circumstances, the user should be allowed to simply select Save and have the system abstract away all necessary substeps.

This is one of the primary focuses of interface design: reconciling system instructions with the actions a user must take to actually interface with the system. This may seem like an obvious and simple task, but quite frankly, it is not simple. It requires an individual to view a system from the standpoint of the average person who has never used the application. Being objective is challenging for many application designers and developers. Their intimate knowledge of the entire system may make it difficult to judge what actions are necessary for the user versus what actions are necessary for the system.

This is where the time taken to get a fresh reading of the requirements can be extremely valuable. To detach yourself from the system design and approach the application from the point of view of the user, step back and reread the requirements for the application and look for certain interaction information:

- Who are the primary users of the system?
- What do users employ the system for? What tasks must it allow them to complete?
- What actions are required by the system in order to complete these tasks?

Write down the answers to these questions. To give an example, if you were considering creating software that allows the user to add footnotes to a document, you should get a list that looks something like Table 8-1.

Table 8-1. *Sample Use Case Skeleton*

User	Interactions with Footnote Facilities
Writer	Add a Footnote
Writer	View a footnote
Writer	Edit a footnote
Writer	Delete a footnote

Those who are experienced in project requirements by now have noticed that what we are doing is creating a set of simple use cases. *Use cases* spell out items such as how all of the

actors interact with the system and what tasks the system will allow them to complete. A set of use cases will usually go into further detail and spell out the additional steps required for the completion of a task. A simple use-case diagram for Table 8-1 would look similar to that shown in Figure 8-2.

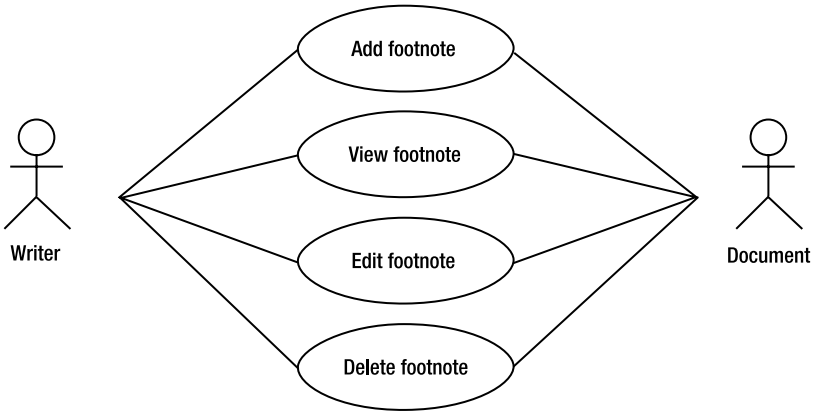


Figure 8-2. *Use case diagram for user interactions with footnote facilities*

■ **Note** User interfaces should attempt to bridge the gap between the use cases and the functionality of the system. The system should, by design, allow for all the actions detailed in the use cases. The interface's job is to make these tasks as easy as possible for the user. If the system architecture does not allow for all of the actions detailed in the use cases, it indicates a serious flaw in the application's design.

Next, fill in the details of each action, like this: "The user may add a footnote to text by first selecting the text to attach a footnote to. Next, the user may optionally specify a name for the footnote, and then add some body text for the footnote. The user then saves the footnote."

Now there should be enough detail to begin the interface design. First and foremost, consider how to reconcile the actions the system requires versus the actions required by the user. Plan an interface that requires the minimum number of steps from the user in order to complete a task. This should be the guiding principle in your interface design.

■ **Tip** Usually, user interfaces allow for multiple ways to complete a task. For example, an application may allow a user to save a document via a menu item, or a designated keystroke, or an onscreen button. This is considered good interface design, and users will memorize and use the method that works best for them. While not explicitly listed as a requirement for the SCJD assignment, developers are expected to follow standard practices, and you would do well to consider adding common features such as keystroke shortcuts to button operations and menu operations.

Another simple principle to consider is how users actually traverse and interpret the user interface. In most Western societies, users generally begin analyzing an interface at the upper-left corner. The tendency from that point is to continue down and to the right. This is due to the nature of the written word in Western nations: Words on a page flow from left to right, so most Western users tend to perceive the left portion of the screen as a beginning point and the right portion as the end. Anything in between these two points represents the necessary steps to travel from the beginning to the end. This concept is illustrated in Figure 8-3.

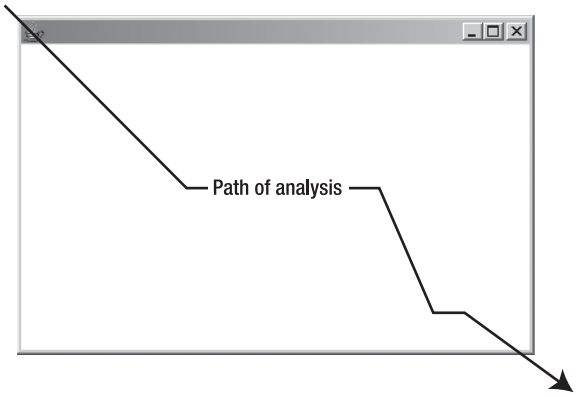


Figure 8-3. *Path of a Western user's eye when interpreting a GUI*

■ **Tip** Even if you're an Asian developer working in Asia, for the purpose of the Sun assignment you should probably assume that the assignment is being evaluated with Western expectations.

When you plan the workflow of an interface, it is advantageous to the user to arrange items in order of importance along this path. The elements we want the user to be aware of first should, of course, be closest to the upper-left corner. The order in which the interface items are placed descending from the starting corner also suggests an order to the actions. For example, consider Figure 8-4.

The path illustrated in Figure 8-4 suggests that users should perform operations on the table and then click the Enter button when they have completed the operations. If the Enter button was placed before the table, the user would likely still figure out and interpret the order of operations, but the interface would seem nonintuitive and clearly not as well organized.

Grouping items also suggests similar functionality. This is why document-storage buttons, such as Open, Save, and New Document, are located in close proximity to each other in most word processing programs.

Tip If you have a question about how to design an action that is intuitive for the end user, the answer is sometimes very easy to find: Look for another popular program with similar functionality. Designing an interface that is similar to other application interfaces to which the user may be accustomed is a helpful way to ensure your application is intuitive as well.

Figure 8-4. *An applied path*

Also, placing a button next to a text field suggests that the button performs some type of action associated with the text field. This principle dictates a significant design error you should be cautious of committing. Because the user interprets grouped items as grouped functionally, placing a Quit button directly next to a Save button is a very bad idea. The Quit item is considered a destructive action because it terminates program execution, whereas the Save button is a nondestructive and frequently used action. This particular interface design error could result in users quitting the program when they really intended to save their work.

It is important to always remember that the average user is accident-prone. Developers must assume that users will make mistakes and click a wrong button every now and then. A good interface design with proper button groupings and placements will minimize this possibility. In many real-world applications, the user interface might even go a step further and implement an "undo" mechanism using the facilities available in the `javax.swing.undo` package. This enhanced functionality allows the user to revert the application state to a preceding point in time and erase the ramification of an action that may have been a mistake. Although this functionality is useful, it is far beyond what is required for the SCJD exam.

Another consideration to keep in mind when designing a user interface is that you as the developer may have assumptions about an application's operating environment, but these assumptions can never be considered fact. For example, just because your development computer has a mouse attached to it does not mean you can assume that every computer your application runs on will have a mouse. This is especially true for Java applications because they aim to target almost every platform in existence. To ensure the functionality of your user interface, each functional widget on the screen should have a keystroke mnemonic mapping,

an equivalent function key, or even a hotkey. That way, if your application is run on a platform that does not have a mouse, the end user will still be able to navigate the application via the keyboard.

An excellent final interface principle is to place items in stationary, predictable locations. Following this principle helps the user develop what is known as muscle memory. The phenomenon occurs when the user becomes so accustomed to the location of an item on the screen that his or her mind subconsciously knows where it is located. Thus, working with the interface becomes less thought-intensive and more like second nature to the user. For example, think about the web browser you have used for the past three years. Even if you have switched browsers numerous times, the Back and Forward buttons are almost always located in the same place. Therefore, locating these buttons with the eye and targeting them with the mouse has become quicker for you, basically because you always know where they are. It is like riding a bicycle: You may stop riding for a few years, but you never forget how to keep your balance.

■ **Note** Users adapt to new UI standards, and therefore what is intuitive is always in flux. Over the years, popular new UI concepts have been introduced. For example, when the tabbed panel metaphor was introduced, it became an instant hit with developers since it offers a highly efficient method to display multiple panels of data in a single window. Due to its efficient nature, the tabbed panel was, and still is, used quite liberally by developers. Because the metaphor has been used so often, users have become accustomed to using it. Some other examples of new UI paradigms that have entered the average user's interface vocabulary include the icon button bar and the web browser-style "back and forward" navigational interface.

As you design and develop an interface for your application, it's always a good idea to test it. Testing an interface isn't as cut and dried as testing application code, however. The best way to test the GUI is to have someone who has never seen or used the application sit down and use it.

First, create an interface prototype. The prototype can be a simple, nonfunctional version of the proposed interface. At this stage, even an interface mock-up sketched out on paper is sufficient. The point is, don't put too much work into a prototype before you're certain of its usability value.

Next, present the test subject with a written list of tasks and ask the user to complete those tasks. Observe the user while they attempt to complete the list using your interface prototype. While you observe the tester, do not instruct them on how to get past areas they may misinterpret or get hung up on. During this process, expect the user to get hung up on your interface prototype and fail at some tasks. This is very common, so be careful not to become aggravated with the testing process. When hang-ups occur, simply ask the test user what they are trying to accomplish and the location where the user expects their next action. Take note of these comments—they are actually suggestions on how to adjust the interface for the next user. Continue this process as many times as you see fit or until the supreme moment when all your testers can use the interface with few or no problems. For more extensive information on usability testing and usability design issues in general, please refer to the FAQs section of this chapter for some valuable URLs.

Model-View-Controller Pattern

The previous section demonstrated the necessity of reconciling system functionality with a user's interaction with the application through the GUI layout. The Model-View-Controller (MVC) pattern limits the dependencies between a system's interface and implementation, and buffers the data and middle tiers from design decisions that may affect the presentation tier.

Why Use the MVC Pattern?

The main purpose of the MVC pattern is the separation of responsibility for the various tasks involved in the user interface (UI). A typical UI can be split into the following areas of responsibility:

1. The interface to the system
2. The displaying of the data to the end user
3. Accepting input from the user, then parsing and processing it

These three areas of responsibility correspond to the Model, the View, and the Controller, respectively.

It is possible, using the MVC design pattern, to keep these three areas of responsibility logically separate. On large teams it might be possible to assign these areas of responsibility to teams who have the best skills to perform the work—for example, those who have good layout skills might be assigned to work on the View, while those who are better at parsing inputs from various sources might work on the Controller.

Using the MVC design pattern allows us to change the UI rapidly. For this book's sample assignment and for the Sun SCJD assignment we will be creating a GUI that will run as a stand-alone application. However, by changing the View and the Controller, we can easily create a web-based application. Another View and Controller can be used to provide a web services interface to the application.

■ **Tip** It is often recommended that when learning any new subject you should try to put the concepts into practice. If you are planning to study for the Sun Certified Web Component Developer certification, you could try creating a web interface to your assignment. This can be easier than trying to implement a brand-new project as part of your studies, as you will have already created the business logic and will therefore only be concentrating on the information pertinent to your new certification studies.

MVC in Detail

As its name suggests, the MVC pattern consists of three primary players, as shown in Table 8-2.

The MVC pattern may seem complex, but it's actually an incredibly simple concept. The concept is very easy to follow when it's boiled down to a real-world example. Let's consider it from the perspective of ordering from your local fast-food restaurant.

Table 8-2. *Components of an MVC Architecture*

Component	Role
Model	The Model is the interface to the remainder of the application. It provides a consistent way for the Controller(s) to access and/or modify data, and returns data in a specified format. It is not uncommon for the Model to incorporate other design patterns—for example, the Façade pattern to hide low-level application details; the observable side of the Observer pattern to provide automatic updates to view(s); the Singleton pattern to ensure that only one instance of the model can exist; and so on.
View	The View is the actual visual representation of the data. The underlying system may store data in a number of ways, but it is the function of the View to interpret and transform the system data into an appropriate format. Some appropriate presentation formats may include an HTML page, a PostScript document, or the GUI of a Java application. No matter what type of data display is chosen, the fact remains that the View is solely responsible for transforming the system's internal data format into the presentation format. The View must also allow for human interaction. The user may click a button in the GUI interface, and the View must handle and dispatch that action. This translates into a call to the Controller. It is important to realize that a View does not necessarily have to correspond to a screen layout—it is also possible in a business-to-business (B2B) scenario that the View may be an XML document that is being sent to a remote application.
Controller	The Controller is the mediator between the data and presentation portions of an application. The View calls the Controller and instructs it to execute a certain action. The action may require data passed from the View, or it may pass data back to the View. In any case, one thing is certain: All interaction between the data tiers and the presentation tiers must occur through the Controller. To preserve the high level of abstraction the MVC allows, the View must never bypass the Controller and communicate directly with components below it. For instance, the View should never call the database directly and tell it to execute a query. Instead, the View should call the Controller and instruct it to call the database component.

When you get to the restaurant you look at the menu and decide the salad looks really nice. So you ask the counter person for a salad and a milk shake. That individual goes to the kitchen, collects your food, and returns it to you.

This is a pretty common scenario, but it is a perfect representation of the MVC pattern. Let's break down the main actors in this scenario and map them to their equivalents in the MVC pattern.

The View in this example is the menu being presented to you. The type of View may vary—you may be inside the restaurant looking at a menu in your hands, or you may be in the drive-through looking at a menu on the wall. The same information is presented in either View, but it might be presented in different formats—a paper menu for patrons inside the restaurant or a painted menu for those in the drive-through.

The Controller in this scenario is the counter person who takes your order. That person acts as the buffer between you and the actual operations of the fast-food restaurant. The counter person takes note of the meal you request, and arranges for it to be provided to you.

■ **Note** It is possible to consider the counter person as part of the View, since they are asking the customer questions and possibly requesting payment for the meal. However, for the purposes of simplicity, we are ignoring the multiple roles the counter person may play.

It is the Controller's job to handle the actual details of getting the meal for you, and you do not need to see or understand how they get your meal. The Controller may go to the kitchen and ask the chef to prepare your salad and milk shake, or the Controller may have to ask two separate people to do these two tasks. The fact is, when you order your food, you never go in and tell the individual employees to go off and make your meal, and you certainly do not prepare the meal yourself. Organizing the supply of food is the job of the Controller, and all you care about is getting the next View (the meal itself).

The next component of the MVC is the Model. In the fast-food example, the Model is the kitchen. The kitchen will provide the various Views—the standard menu, any "specials of the day" menus (which might be listed on a separate page on the menus used inside the restaurant, or on a blackboard for the drive-through customers)—then take the order and present the data (the food) for the next View.

Benefits of MVC

The main benefit of the MVC pattern is the abstraction of the data presentation from the system operations. The implications of this fact are many. For example, if a system must display data to many different output methods, this pattern is the ideal solution. Under the MVC pattern, the data model remains the same, but the method of display can very easily change. Thus, the same system could have a GUI front-end and a web-based HTML front-end, all without any changes to the rest of the system.

The MVC pattern also helps limit the scope of changes to a system. The majority of the time, a client will get preoccupied with the functionality of the presentation tier. This is quite understandable, considering that this is the portion of the application the end user will actually use. Many times, clients will request numerous changes to the front-end design. The MVC pattern limits the scope of these changes and often precludes them from impacting other portions of the system. This is not to say that the MVC pattern makes system updates a snap, but it certainly does make the front-end of the system much more flexible.

Drawbacks of MVC

Everything has a good side and a bad side, and the MVC pattern is no exception. MVC is not the best option for every system. For instance, a system that does not need the capability to display data through multiple sources may not be a good candidate for this pattern. MVC does employ a large amount of data abstraction, and sometimes the associated overhead is not worth the cost, both in development time and performance.

The MVC pattern is generally a good idea, and it is one of the more lightweight design patterns. It is also widely used, and is supported by most development platforms. Its usefulness almost directly correlates with the size of a project. If the development task requires

several programmers and substantial expandability, the MVC pattern is a sure win. If the project is very small in scope and will probably never be updated, the MVC pattern is probably unnecessary.

Alternatives to MVC

The most obvious alternative to the MVC pattern is not to use it at all. A system can be designed and implemented that tightly integrates the actual system functions with the user interface. This will, of course, limit the flexibly of the GUI. All changes will directly impact all areas of the system, since the system is basically one unit.

On some programming platforms, tight integration of the presentation and data tiers is possible. Due to the event-driven nature of the Java platform, some aspects of the MVC pattern are inherent in any user interface. All interface components produce actions, and there are always classes that are created to deal with these actions. In this sense, the MVC pattern is built in, but the programmer can choose to limit its effect and not use the full MVC architecture.

Swing and the Abstract Windows Toolkit

The previous sections emphasized some high-level methodologies of interface design. Most of the ideas, such as the MVC pattern, are platform agnostic and apply to the architecture of development projects, which can then be developed in almost any language. The next sections shift to a platform-specific approach using the Abstract Window Toolkit (AWT) and Swing to explain methods of interface implementation on the Java platform.

Entire books have been written on how to program with AWT and Swing. This section offers only an overview of some of the basic concepts. For more complete coverage of these topics, we recommend you read the tutorials available online at `http://java.sun.com/docs/books/tutorial/uiswing`.

Layout Manager Overview

`Components` (buttons, text areas, graphics, etc.) are added to `Containers` such as a `JFrame`, `JPanel`, or a `JWindow`. Containers use layout managers to specify how components should be laid out within the container. `Window` and `Frame` containers use `BorderLayout` as their default layout manager. This particular layout schema has some peculiar functionality you may want to review.

First, the `BorderLayout` divides the container into five areas, as shown in Figure 8-5.

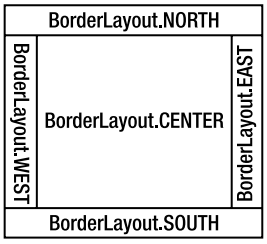


Figure 8-5. *The areas of BorderLayout*

A Component can be placed into a specified region of the BorderLayout by calling the Container's add method, with the first parameter being the object to add, and the second parameter being the constraint, which for BorderLayout will be one of the five regions of the BorderLayout, as represented by the following String constants:

- BorderLayout.NORTH
- BorderLayout.SOUTH
- BorderLayout.EAST
- BorderLayout.WEST
- BorderLayout.CENTER

When a component is placed into one of these five regions, it will immediately expand to fill that area, observing any constraints for that area. For instance, in Listing 8-1 a button is added to each of the regions. Figure 8-6 shows the result.

Listing 8-1. *BorderLayoutExample*

```
import java.awt.*;
import javax.swing.*;

public class BorderLayoutExample extends JFrame {
    public static void main(String[] args) {
        new BorderLayoutExample().setVisible(true);
    }

    public BorderLayoutExample() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        add(new JButton("North"), BorderLayout.NORTH);
        add(new JButton("South"), BorderLayout.SOUTH);
        add(new JButton("East"), BorderLayout.EAST);
        add(new JButton("West"), BorderLayout.WEST);
        add(new JButton("Center"), BorderLayout.CENTER);
        pack();
    }
}
```

■ **Note** Prior to JDK 1.5, it was necessary to get the content pane for a JFrame, and then add components to the content pane. JDK 1.5 provides overridden add methods for JFrame that allow the code shown earlier to make it appear that we are adding components directly to the JFrame—in fact we are not; the components are being added to the component pane in the overridden add method.

Figure 8-6. *An example of the BorderLayout in action*

As can be seen, the buttons that were added to the "North" and "South" regions have expanded to fill the region from the far left to the far right of the `JFrame`; however, they have not expanded to fill space above or below the button. The buttons added to the "East" and "West" have expanded to fill the space above and below the button, but they have not expanded to fill to the left or right. The button added to the "Center" can expand in all directions.

■ **Caution** If more than one component is added to a region, the newest component placed inside the region is drawn on top of all other previously added components. This often causes confusion when programmers are first adding components, as it appears that some of their components have been lost. There is probably no reason why you would ever want to intentionally add two components in the same region.

There are multiple strategies you can use to lay out components in a container and prevent the preceding anomalies. The best, and perhaps easiest, way to control component flow is to first place all components into a `JPanel` container and then place the `JPanel` container into one of the five `BorderLayout` areas. Unlike the `BorderLayout`, the `JPanel`'s default layout manager is the `FlowLayout`. The properties of this particular type of layout are much easier to deal with than that of the `GridBagLayout` or even the `GridLayout`. Every component placed in the `FlowLayout` will maintain its suggested size and will flow from left to right.

The `FlowLayout` also accepts some justification parameters. The constructor `FlowLayout (int index)` may be used with one of the following constants in the `FlowLayout` class:

- `FlowLayout.CENTER`
- `FlowLayout.RIGHT`
- `FlowLayout.LEFT`

Listing 8-2 uses a `JPanel` combined with a `JFrame` container to display three buttons. You should also try resizing the `JFrame` to see the effect this has on `FlowLayout`. Some examples are shown in Figures 8-7, 8-8, and 8-9.

Listing 8-2. *A FlowLayout with Three Centered Buttons*

```
import java.awt.*;
import javax.swing.*;

public class ExampleFlowLayout extends JFrame {
    public static void main(String[] args) {
        new ExampleFlowLayout().setVisible(true);
    }

    public ExampleFlowLayout () {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        JPanel thePanel = new JPanel();
        thePanel.add(new JButton("One"));
        thePanel.add(new JButton("Two"));
        thePanel.add(new JButton("Three"));
        add(thePanel, BorderLayout.CENTER);
        pack();
    }
}
```

Figure 8-7. *FlowLayout showing the default operation—all components laid out at their preferred size next to each other*

Figure 8-8. *An example of FlowLayout when the container has been expanded*

Figure 8-9. *An example of FlowLayout when the container has been reduced in size*

Notice that the button sizes are calculated by the `FlowLayout` and displayed at just the right size to show their text labels. Also note that the default justification for the `FlowLayout` is `CENTER`. Listing 8-3 demonstrates an alteration to the code segment in Listing 8-2 that changes the justification of the buttons. An example of how this might appear is shown in Figure 8-10.

Listing 8-3. *A FlowLayout with Three Buttons Aligned to the Right*

```
import java.awt.*;
import javax.swing.*;

public class MyFrame extends JFrame {
    public static void main(String[] args) {
        new MyFrame().setVisible(true);
    }

    public MyFrame() {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        JPanel thePanel = new JPanel(new FlowLayout(FlowLayout.RIGHT));
        thePanel.add(new JButton("One"));
        thePanel.add(new JButton("Two"));
        thePanel.add(new JButton("Three"));
        add(thePanel, BorderLayout.CENTER);
        pack();
        setSize(300, 70);
    }
}
```

Figure 8-10. *An example of FlowLayout with components aligned to the right*

■ **Tip** The combination of the `JFrame` and `JPanel` containers is a powerful yet simple way to lay out graphical components. There are certainly more complex ways to handle interface layout, but the combination of these two containers will probably be more than enough to lay out the client interface for the SCJD exam.

Look and Feel

Swing supports a pluggable look and feel. This feature is a welcome side effect of Swing's lightweight components that can be overwritten with programmer-defined design and functionality. A prime example of this feature is the Ocean look and feel. This interface style can be used on any platform that supports Java and graphical user interfaces, and it can be used as a standard interface across all platforms. If an interface absolutely must look and behave the same on every platform, then the Ocean look and feel is the right choice.

The look and feel of any Swing-based GUI can be changed on the fly programmatically within the application.

Here are some common LookAndFeel subclasses:

- javax.swing.plaf.metal.MetalLookAndFeel
- com.sun.java.swing.plaf.windows.WindowsLookAndFeel
- com.sun.java.swing.plaf.motif.MotifLookAndFeel
- com.sun.java.swing.plaf.gtk.GTKLookAndFeel
- com.apple.mrj.swing.MacLookAndFeel

■ **Caution** The only look and feel that is guaranteed to exist on all platforms is the one that is in the standard Java packages: javax.swing.plaf.metal.MetalLookAndFeel. All other look-and-feel packages are in nonstandard packages—the com.sun or com.apple (or other vendor) packages. The vendor packages probably won't exist in JVMs produced by other vendors, and may not even exist in all JVMs produced by a particular vendor. For example, the WindowsLookAndFeel is only available on Microsoft Windows platforms, and the GTKLookAndFeel is only available on platforms supporting GTK (typically Unix and Unix-like systems).

Since we have been using the Ocean theme from the Metal look and feel for the previous examples, Listing 8-4 will show an example of setting the look and feel to the Microsoft Windows look and feel, or to a look and feel specified on the command line. An example of how this will look is shown in Figure 8-11. Other look-and-feel examples are shown in Figures 8-12 through 8-15.

Listing 8-4. *Setting the Microsoft Windows Look and Feel*

```
import java.awt.*;
import javax.swing.*;

public class MyFrame extends JFrame {
    public static void main(String[] args) throws Exception {
        new MyFrame(args).setVisible(true);
    }

    public MyFrame(String[] args) throws Exception {
        String lookAndFeelName = (args.length > 0)
                ? args[0]
                : "com.sun.java.swing.plaf.windows.WindowsLookAndFeel";

        UIManager.setLookAndFeel(lookAndFeelName);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        Panel topPanel = new Panel(new FlowLayout(FlowLayout.LEFT));
        topPanel.add(new JTextField(15));
        add(topPanel, BorderLayout.NORTH);
```

```
        Panel centerPanel = new Panel(new FlowLayout(FlowLayout.RIGHT));
        centerPanel.add(new JButton("One"));
        centerPanel.add(new JButton("Two"));
        centerPanel.add(new JButton("Three"));
        add(centerPanel, BorderLayout.CENTER);
        pack();
        setSize(210, 100);
    }
}
```

Figure 8-11. *An example of using the Microsoft Windows look and feel*

Figure 8-12. *An example of using the Motif look and feel*

Figure 8-13. *An example of using the Metal look and feel with the Ocean theme*

Figure 8-14. *An example of using the Metal look and feel with the Steel theme*

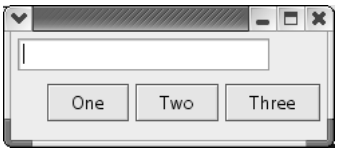


Figure 8-15. *An example of using the GTK look and feel (Unix platforms)*

You may have noticed that Figures 8-13 and 8-14 both use the Metal look and feel but different themes. Prior to JDK 1.5, the Metal look and feel looked like Figure 8-14; however, with JDK 1.5 Sun has improved the standard look and feel, making the Ocean theme shown in Figure 8-13. To revert to the old theme, the following command line was used:

```
java -Dswing.metalTheme=steel MyFrame javax.swing.plaf.metal.MetalLookAndFeel
```

■ **Caution** The layout managers have been designed to perform the hard work of determining where components should be placed relative to one another, as well as determining the size of containers. It is possible to explicitly place components with the `setLocation` method, and set their size with the `setSize` method (which we used in Listing 8-4); however, as shown in Figures 8-11 through 8-15, the size of components varies depending on the look and feel used, so deliberately setting the size or location of a component or a container can result in strange-looking GUIs. We strongly recommend that you leave the work of component placement and container sizing to the layout managers.

When a different look-and-feel package is used, the interface adopts not only the look of that particular platform but also the functionality of its interface widgets. For example, a drop-down menu in the Motif look and feel is vastly different from the menu functionality under the Windows or Metal look and feel. This is because Motif represents the interface look and also the functionality of an X Window system.

The JLabel Component

Standard user interfaces have labels near any component that accepts input, providing the user with a brief explanation of what the user input is for. An example of this might be having the label "Surname" next to the text field that accepts the surname data—the label serves to remind the user what sort of data is to be entered in the text field.

The Swing component that holds a label is the `JLabel`. A simple constructor for this could be

```
JLabel zipCodeLabel = new JLabel("Zip code");
```

While this on its own is not very exciting, you can also specify which character will be displayed as the mnemonic (which character will be underlined). Since it does not make sense for a label to have focus, you normally set the displayed mnemonic, and at the same time set which field will get focus if the mnemonic is pressed. This could look similar to

```
zipCodeLabel.setDisplayedMnemonic('Z');
zipCodeLabel.setLabelFor(zipCode);
```

In this case, if a user presses the mnemonic key (by pressing the Alt and Z keys simultaneously), then focus will be transferred to the `zipCode` field.

This technique will be demonstrated in Listing 8-5 in the next section, with a sample GUI displayed in Figure 8-16.

The JTextField Component

The `JTextField` is the basic data entry field. It allows the user to enter plain text up to a specified length. Here's a simple constructor for the `JTextField` with a size of 15 columns:

```
JTextField zipCode = new JTextField(15);
```

Other constructors for the `JTextField` allow you to specify a default value to be displayed in the text field, and a `Document` to validate data entry (data validation will be covered in the next subsection).

A very simple application to demonstrate the use of `JLabel`s and `JTextField`s is shown in Listing 8-5, and the GUI that this creates is shown in Figure 8-16. A `JButton` has been included in the GUI to allow you to experiment with transferring focus away from the `JTextField` and then use the mnemonic to transfer focus back again.

Listing 8-5. *Demonstration of JLabel and JtextField Components*

```
import java.awt.*;
import javax.swing.*;
import javax.swing.text.*;

public class MyFrame extends JFrame {
    public static void main(String[] args) throws Exception {
        new MyFrame().setVisible(true);
    }

    public MyFrame() throws Exception {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new FlowLayout());

        JLabel zipCodeLabel = new JLabel("Zip code");
        JTextField zipCode = new JTextField(15);

        zipCodeLabel.setDisplayedMnemonic('Z');
        zipCodeLabel.setLabelFor(zipCode);

        this.add(zipCodeLabel);
        this.add(zipCode);
        this.add(new JButton("A button for focus"));
        pack();
    }
}
```

Figure 8-16. *Demonstration of JLabel and JtextField components*

Validating the Contents of a JTextField

In a perfect world, users would only ever enter valid data into our applications. However, in the real world, people make mistakes, and unfortunately, if not caught in time, major problems can occur if invalid data is entered. So we should make a reasonable effort to prevent invalid data from being entered, and possibly validate data once it has been entered.

Listing 8-5 contained a field named `zipCode`. In the United States, zip codes are five digits long, and are used to identify the area where a letter or parcel is to be delivered. Unfortunately, our sample program will allow any data to be entered, regardless of length or content.

If we want to limit the zip code field to accept only five-digit zip codes, we can use a subclass of the `JTextField`: the `JFormattedTextField` in combination with a `MaskFormatter`. A `JFormattedTextField` will only allow characters to be entered that match a specified mask. The `MaskFormatter` provides a simple method of specifying a text formatter based on a mask—any characters that do not match the mask will be discarded. The mask for a number is the # character. So we could change our definition of the `zipCode` variable from Listing 8-5 as follows:

```
MaskFormatter fiveDigits = new MaskFormatter("#####");
JTextField zipCode = new JFormattedTextField(fiveDigits);
zipCode.setColumns(5);
```

If you make these changes, you will find that you can no longer enter any character other than a number, and you cannot add more than five digits.

Taking validation a step further, let's consider the case where users want to enter the zip+4 code, where the original five-digit code is followed by a dash and an additional four digits to further narrow down the delivery address—for example, the first five digits specify the delivery office (post office), the next two digits identify a set of blocks on a major street, and the final two digits identify the particular block on the street.

However, zip+4 is not used everywhere, nor is it mandatory to use this format even where it's available (for that matter, it does not appear to be mandatory to use zip codes for anything going through the US Postal Service, but it is probably advisable if you want your letter to arrive in a timely fashion).

One method of handling this is by creating our own text field, with its own document model, in which we override the `insertString` method. So once again, our definition of the `zipCode` variable is changing:

```
JTextField zipCode = new ZipTextField(9);
```

The `ZipTextField` extends `JTextField`, but all it overrides are the constructors and the `createDefaultModel` method:

```
private class ZipTextField extends JTextField {
    ZipTextField() {
        super();
    }

    ZipTextField(int columns) {
        super(columns);
    }
```

```
    protected Document createDefaultModel() {
        return new ZipDocument();
    }
}
```

You could override more constructors if you wish, or you could even leave out the constructors (just use default constructors, and call the inherited `setColumns` method).

The hard work is all done in the model for our `ZipTextField`. For `JTextField` (which we are subclassing), the default model is a `PlainDocument`. We will subclass that to form our `ZipDocument`, in which we will validate text entered.

■ **Note** `JTextField`, like many Swing components, uses the Model-View-Controller design pattern internally. This gives us two potential areas where we can control data entry—in the Model that contains the data entered, and in the Controller before data is passed to the Model.

```
private class ZipDocument extends PlainDocument {
    public void insertString(int offs, String str, AttributeSet a)
            throws BadLocationException {
        if (str == null) {
            return;
        }


        for (char c : str.toCharArray()) {
            if (! ((Character.isDigit(c) && offs < 10 && offs != 5)
                    || (b == '-' && offs == 5))) {
                return;
            }
        }

        super.insertString(offs, str, a);
    }
}
```

To keep this example simple, we have only overriden the `insertString` method, and only checked that the character being entered is valid in the location it is being entered. If it is valid, we call the `insertString` method of the super class to perform the work of actually inserting the string.

Because we have chosen to keep this simple, we have not shown code for validating the deletion of entered data for which we would have to override the `remove` method. A more complete example is provided in the downloadable source code for this book in the port number validation.

■ **Caution** Nothing shown so far prevents the user from entering something like 12345-67 and then moving to the next field or clicking elsewhere in the document. Therefore, some additional validation is in order somewhere, an example of which is shown in the full code.

The JButton Component

The `JButton` is a "click to do something" type of button. What it does when the user clicks it is up to you. An example of a constructor for this is

```
JButton exitButton = new JButton("Exit");
```

Other constructors allow for an icon to be used instead of or as well as the text, and setting an `Action` for the button.

The constructor on its own is not very useful—we have not specified what should happen when the button is clicked. For that, we need to add an `ActionListener`:

```
exitButton.addActionListener(anActionListener);
```

The `ActionListener` interface allows you to listen for actions: button clicked on, button clicked on while a modifier (the Ctrl, Alt, Shift, or Meta key) was pressed, and so on. When such an action occurs, your `actionPerformed` method will be called by the event dispatcher thread.

The `ActionListener` can be an anonymous inner class, a private class, or an external class, or (since `ActionListener` is an interface) your View can implement the `ActionListener`.

Here is an example of creating an anonymous inner class for your `ActionListener`:

```
JButton exitButton = new JButton("Exit");
exitButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        System.out.println("Somebody clicked the Exit button");
        System.exit(1);
    }
});
```

■ **Note** The last line of the code snippet above may look confusing, but if you count backwards, you will find that the closing brace (}) matches the opening brace for the `new ActionListener() {`, and the closing bracket ()) matches the opening bracket for the `addActionListener(`.

As with all choices, there are good points and bad points for using an anonymous inner class. Some of the good points are as follows:

- There is no need to create a separate class to handle user events.
- There is no need to check which component triggered the action event.
- The code to handle the event is right with the component that triggered the event.

The first point can be worth consideration, but if you are using the MVC pattern, you might prefer to have the Controller handle the user events. Alternatively, a simple anonymous class can call a method in the Controller class, making the Controller class less dependent on the GUI architecture.

The last point can be both good and bad as well—you might not want the code *handling* the event with the code that is *configuring* the View. It might make more sense to keep event-handling code separate.

If you do decide to have a separate class or method handling events, you might want to consider using the `setActionCommand` method to set a string by which your component can be easily identified. The string you set can then be retrieved from the `ActionEvent` handed to the `actionPerformed` method. This is demonstrated in Listing 8-6 in the next section.

The JRadioButton Component

Radio buttons are small buttons that are logically grouped together, but only one of the group can be active at any given time. This is similar to the way a radio with buttons for several pre-set stations should only have one button pressed at any given time—you can only listen to one station at a time.

A simple constructor for a `JRadioButton` could be

```
JRadioButton serverButton = new JRadioButton("Server");
```

Other constructors allow for an icon to be used instead of or as well as the text, setting the initial state of the radio button, and setting an `Action` for the radio button.

As with the `JButton`, an `ActionListener` can be added to each `JRadioButton`. You might use this if you needed to enable or disable fields dependent on which button a user clicked. However, it is not always necessary to have an `ActionListener`—if you don't care about which button is clicked until after the user performs some other action, then you can use the `isSelected` method to check the user's choice.

For `JRadioButtons` to be effective, several of them should be logically grouped together, so that only one of the logical group can be selected at any given time. You do this by creating a `ButtonGroup`, and adding the radio buttons to it:

```
ButtonGroup applicationMode = new ButtonGroup();
applicationMode.add(serverButton);
```

A complete example of `JButtons` and `JRadioButtons` is demonstrated in Listing 8-6, and the window it would create is shown in Figure 8-17.

Listing 8-6. *Demonstration of JButton and JRadioButton Components*

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

public class MyFrame extends JFrame {
    private static final String EXIT_COMMAND = "EXIT";
    private static final String CLIENT_COMMAND = "CLIENT";
    private static final String SERVER_COMMAND = "SERVER";
```

```
    public static void main(String[] args) throws Exception {
        new MyFrame().setVisible(true);
    }

    public MyFrame() throws Exception {
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        ActionListener buttonHandler = new MyFrameActionListener();

        JButton exitButton = new JButton("Exit");
        exitButton.setActionCommand(EXIT_COMMAND);
        exitButton.addActionListener(buttonHandler);

        JRadioButton serverButton = new JRadioButton("Server");
        serverButton.setActionCommand(SERVER_COMMAND);
        serverButton.addActionListener(buttonHandler);
        JRadioButton clientButton = new JRadioButton("Client");
        clientButton.setActionCommand(CLIENT_COMMAND);
        clientButton.addActionListener(buttonHandler);

        ButtonGroup clientServerGroup = new ButtonGroup();
        clientServerGroup.add(serverButton);
        clientServerGroup.add(clientButton);

        JPanel clientServerPanel = new JPanel();
        clientServerPanel.add(serverButton, BorderLayout.NORTH);
        clientServerPanel.add(clientButton, BorderLayout.SOUTH);

        this.add(clientServerPanel, BorderLayout.CENTER);
        this.add(exitButton, BorderLayout.SOUTH);

        pack();
    }

    private class MyFrameActionListener implements ActionListener {
        public void actionPerformed(ActionEvent ae) {
            if (EXIT_COMMAND.equals(ae.getActionCommand())) {
                System.exit(0);
            } else if (SERVER_COMMAND.equals(ae.getActionCommand())) {
                System.out.println("Server selected");
            } else if (CLIENT_COMMAND.equals(ae.getActionCommand())) {
                System.out.println("Client selected");
            }
        }
    }
}
```

Figure 8-17 shows the result from Listing 8-6.

Figure 8-17. *Demonstration of JButton and JRadioButton components*

The JComboBox Component

Sometimes it makes sense to give users a clear choice of options, rather than requiring them to type in a choice (which is an error-prone approach). The `JComboBox` provides us with a simple box that, when clicked, pops down a list of options from which the user can choose.

If you know the items that are to be used in the list, then there are constructors that allow you to specify the initial items. Alternatively, you can dynamically add or remove items from the list using the `addItems` and `removeItems` methods, respectively.

As with the `JButton` and `JRadioButton`, an `ActionListener` can be added to each `JComboBox`. You might use this if you needed to take action immediately after the user selects an option. However, it is not always necessary to have an `ActionListener`—if you don't care about which button is clicked until after the user performs some other action, then you can use the `getSelectedIndex` method to determine the index of the item chosen, or the `getSelectedItem` method to find out the object selected.

`JComboBox` usage is demonstrated in Listing 8-7 shown in the next section, with the window created displayed in Figure 8-18.

The BorderFactory

So far all the components demonstrated have appeared to the end user to be in the same-`JFrame`—there is nothing to logically separate one component (or set of components) from another.

However, there are times when it makes sense to logically group components together by drawing a border around them. One of the most common uses of borders is to create a perceived link around several radio buttons or check boxes. We did not put a border around our radio buttons in the previous section because we want to emphasize that the border only creates a **user perception** that the buttons are logically linked—it is possible to have a border drawn around buttons that are not logically linked, and just creating a border around buttons does not logically link them.

Unlike with most Swing components, we do not create a border directly. Instead we call one of the static methods of the `BorderFactory` class to create a border for us, which we can then use in a `JPanel` or `JFrame`.

■ **Note** The Factory design pattern, used by the `BorderFactory`, is often used when many similar objects need to be created, but the user of the objects does not need to know the implementation details—all the user of the object needs to know about is how to use it. Each of the `BorderFactory`'s create methods will create a `Border`, where `Border` is an interface. Since we have an interface for all potential borders, we can use the created border in any `JFrame` or `JPanel` without concerning ourselves with what type of border was created. Furthermore, we don't have to worry about how a particular border will be created on different operating systems—the factory handles all that for us.

Which particular border to create and use is up to you. Frequently status bars use a lowered bevel border (which you can create using the `createLoweredBevelBorder` method), groupings of radio buttons use a titled border (`createTitledBorder`), and groupings of buttons might be enclosed in a beveled border without a title (`createBevelBorder`).

An example of creating a titled border around a `JComboBox` is shown in Listing 8-7, with the window created shown in Figure 8-18. For this example we included some spacing labels to make the border more obvious.

Listing 8-7. *Demonstration of the JComboBox Component and the BorderFactory*

```
import java.awt.*;

import javax.swing.*;


public class MyFrame extends JFrame {
    private final static String TITLE = "Title goes here";

    public static void main(String[] args) throws Exception {
        new MyFrame().setVisible(true);
    }

    public MyFrame() throws Exception {
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        String[] items = {"One", "Two", "Three", "Four", "Five"};
        JComboBox choosableItems = new JComboBox(items);

        JPanel clientServerPanel = new JPanel();

        clientServerPanel.setBorder(BorderFactory.createTitledBorder(TITLE));
        clientServerPanel.add(new JLabel("Pick a number:"), BorderLayout.EAST);
        clientServerPanel.add(choosableItems, BorderLayout.CENTER);

        this.add(new JLabel("    "), BorderLayout.NORTH);
        this.add(new JLabel("    "), BorderLayout.SOUTH);
        this.add(new JLabel("    "), BorderLayout.EAST);
```

```
        this.add(new JLabel("    "), BorderLayout.WEST);
        this.add(clientServerPanel, BorderLayout.CENTER);

        pack();
    }
}
```

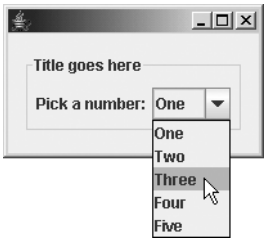


Figure 8-18. *Demonstration of the JComboBox component and the BorderFactory*

Listing 8-7 was designed to show how to make a simple border without introducing too many new features at once. To that end, we put empty labels around the border just to make it stand out—normally you would not do this in practice. Instead, you might consider using a compound border, as shown in Listing 8-8. This will produce an almost identical result to that shown in Figure 8-18.

Listing 8-8. *Demonstration of a Compound Border*

```
import java.awt.*;
import javax.swing.*;

public class MyFrame extends JFrame {
    private final static String TITLE = "Title goes here";

    public static void main(String[] args) throws Exception {
        new MyFrame().setVisible(true);
    }

    public MyFrame() throws Exception {
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        String[] items = {"One", "Two", "Three", "Four", "Five"};
        JComboBox choosableItems = new JComboBox(items);

        JPanel clientServerPanel = new JPanel();
        clientServerPanel.setBorder(
                BorderFactory.createCompoundBorder(
                        BorderFactory.createEmptyBorder(10, 10, 10, 10),
                        BorderFactory.createTitledBorder(TITLE)));
```

```
        clientServerPanel.add(new JLabel("Pick a number:"), BorderLayout.EAST);
        clientServerPanel.add(choosableItems, BorderLayout.CENTER);
        this.add(clientServerPanel, BorderLayout.CENTER);

        pack();
    }
}
```

The JTable Component

As discussed earlier in this chapter, the table paradigm has developed into a de facto data display mechanism. As you may recall, the table schema is a great way to display data because it offers ease of interpretation for the user and efficiently displays large amounts of data in a small area.

The `JTable` renders data in the familiar "data table" style. For the developer, it is the perfect combination of ease of use and extensibility. You can create a `JTable` for casual use by simply calling the `JTable` constructor method `JTable (Object rowData [][], Object columnNames[])` and defining two arrays containing header names and the data rows, as demonstrated in Listing 8-9.

Listing 8-9. *A Simple JTable Constructor*

```
Object [ ][ ] rows = {{"Data 1", "Data 2"}, {"Data 3", "Data 4"},
                      {"Data 6", "Data 6"} };
Object [ ] colNames =  {"Header 1", "Header 2"};

JTable table = new JTable (rows, colNames);
```

Listing 8-9 creates a table composed of three rows. This table is shown in Figure 8-19.

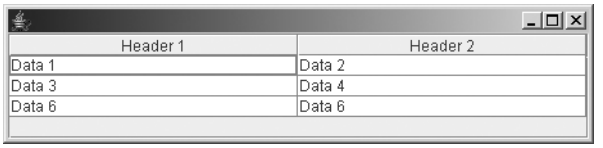


Figure 8-19. *A sample table*

Note that each row is labeled with a header. If an array of table headers is not specified, the table will only display the rows and columns without any headers. Also notice that the table columns are automatically sized, but they may be resized and even rearranged. The following methods can be called on a `JTable` instance to enable and disable these capabilities:

- `setSelectionMode(ListSelectionModel.SINGLE_SELECTION)`
- `setAutoResizeMode(JTable.AUTO_RESIZE_ALL_COLUMNS)`

The preceding table example is perfect if the data in the table only needs to be displayed once, but the example is not a very malleable method with which to display data. To become a

more flexible data display, a good table class would need additional methods and constructors to fit within the paradigm of the MVC pattern. It so happens that the `JTable` fits incredibly well into this pattern—so much so that the `JTable` actually defines its own model: the `TableModel` interface.

■ **Note** Many Swing components, such as the `JTable` and the `JTree`, implement their own MVC architecture.

The TableModel

The `TableModel` is one of the easiest and most flexible ways to display a data set in a `JTable`. Earlier in this chapter, the interaction between the Model and the View in the MVC pattern was discussed in detail. As you may recall, the View is solely responsible for taking the data contained in a Model object and converting it into a visual representation. The `JTable` actually incorporates this part of the MVC pattern into its standard API. An implemented `TableModel` interface acts as the Model in this case, and the `JTable` instance acts as the View. The `TableModel` is essentially handed off to a `JTable` instance, and the table widget renders the data model into the visual representation. The Model may be specified in the `JTable` constructor:

```
JTable table = new JTable (TableModel model);
```

or set via the void `setTableModel (TableModel model)` instance method. Once the `TableModel` is passed to the table instance, the `JTable` class literally takes care of the rest.

At this point, the `JTable` may seem like a snap to use. While the `JTable` is quite easy to use, the bulk of the work for the developer comes in the task of implementing the `TableModel` interface.

As with event listener interfaces, not all of the methods specified in the `TableModel` interface are always necessary for a given data set. For instance, if a developer wanted to implement a simple `TableModel`, it would be inefficient to have to provide implementations for all the interface methods, such as `removeTableModelListener(TableModelListener modListener)`. Thus, an adapter class is provided that functions in a manner similar to event adapter classes. The `AbstractTableModel` class implements the `TableModel` interface such that each method already has a default implementation. Therefore, extending the `AbstractTableModel` class is usually the best starting point when creating a custom `TableModel`.

In version 2.0 of the Denny's DVDs application, the `DVDTableModel` class is an extension of the `AbstractTableModel` class. The `DVDTableModel` does not need to implement the `getColumnClass`, `removeTableModelListener`, and `addTableModelListener` methods, so their default implementations provided by the abstract implementation will suffice. All other methods in the `TableModel` interface will be implemented.

First, notice the addition of the two member variables in the `DVDTableModel` class in Listing 8-10.

Listing 8-10. *The DVDTableModel's Internal Members That Hold the Table Rows*

```
private String [] headerNames = {"UPC", "Movie Title", "Director",
                                 "Lead Actor", "Supporting Actor", "Composer",
                                 "Copies in Stock"};

private List<String[]> dvdRecords = new ArrayList<String[]>(5);
```

Remember that Model objects act as data containers. The View object that transforms a Model into a display format does not require knowledge of how they internally represent data. On a very abstract level, tables encapsulate two things:

- The names of the columns
- The data contained in each row

These two items require a `TableModel` to internally represent two types of collections: one that contains a list of header names and one that contains a list of row values. The only decision that remains is what type of collection is appropriate to represent these two types of data.

In the case of the `JTable` contained in our client application, the header names will remain the same throughout the life of the application. Therefore, an array of `String` objects will be the perfect data structure to hold the column names. Next, the row column values for each row are immutable in the sense that once a row is created, the number of values in a row does not need to change. Thus, an array of `String` values can also represent a row of `DVD` data.

The actual collection that holds the rows is entirely different from the table headers and column data in that the number of rows changes—and changes quite often. If a user searches for a list of `DVD` records, there is no way to ensure how many rows will result, and therefore it will be impossible to anticipate the size of the data structure required to hold the values. The best option to hold rows of data is a dynamic collection that can have values easily appended, removed, and iterated through. The order of the results also matters, so a set collection is obviously a poor choice. The best choice is some type of dynamic list. The list can now be narrowed to a `Vector`, a `LinkedList`, or an `ArrayList`. As mentioned in the "Internally Synchronized Classes" section of Chapter 4, `Vector`s provide little benefit in most cases, but can provide a false sense of thread safety—we therefore generally recommend against their use. In this particular case, the values in our `TableModel` do not have to be thread-safe because only one thread is manipulating their values in a single instance. Thus, a `Vector` provides unneeded overhead in this application without providing any benefits. A `LinkedList` provides us with some additional methods over an `ArrayList`—specifically the ability to add and remove the first and last objects in the collection; however, we do not need this additional functionality. The best choice for the representation of `DVD` data rows in our `TableModel` is clearly an `ArrayList`.

Next, our `TableModel` must implement some of the required methods that the `JTable` uses to render the Model into a View. For instance, the `TableModel` must have a way to inform the `JTable` of how many columns it encapsulates. The method `getColumnCount` shown in Listing 8-11 provides this functionality.

Listing 8-11. *Our TableModel's getColumnCount Method*

```
public int getColumnCount() {
    return this.headerNames.length;
}
```

In this case, the number of columns is always equal to the number of header titles, so this method can simply return the length of the `headerNames` member. Our `TableModel` must also return the name of each column. Listing 8-12 returns the `String` value from the `headerNames` member at a specified index.

Listing 8-12. *The TableModel's getColumnName Method*

```
public String getColumnName (int column) {
    return headerNames[column];
}
```

Next, the `TableModel` returns and sets a cell value at a given row and column index. The methods shown in Listing 8-13 allow the assignment and retrieval of values at specific row and column indexes.

Listing 8-13. *The TableModel's set and get valueAt Methods*

```
public Object getValueAt(int row, int column) {
      String[] rowValues = this.dvdRecords.get(row);
      return rowValues[column];
}

public void setValueAt(Object obj, int row, int column) {
     Object[] rowValues = this.dvdRecords.get(row);
     rowValues[column] = obj;
}
```

Note, it would have been possible for the `getValueAt` method to have been written as

```
public Object getValueAt(int row, int column) {
      return this.dvdRecords.get(row)[column];
}
```

Whenever you are tempted to take a shortcut like this, though, you should consider what benefit it gives you. Is the creation or destruction of the `rowValues` **reference** to the array so expensive that this will gain much efficiency? Alternatively, will this code be harder to read and therefore harder to maintain?

Next, the method `getRowCount` shown in Listing 8-14 returns the number of data rows encapsulated in the `TableModel`. In our `TableModel`, the number of rows is the size of the `DVD ArrayList`.

Listing 8-14. *The DVDTableModel's getRowCount Method*

```
public int getRowCount() {
      return this.dvdRecords.size();
}
```

The `isCellEditable` method in our `TableModel` class (see Listing 8-15) indicates whether or not a cell is editable. No cells in our particular `TableModel` implementation are editable, so the method returns `false` by default. If particular cells in the `TableModel` were editable, this

method would take the column and row indexes into account and evaluate whether or not a cell was editable.

Listing 8-15. *The TableModel's isCellEditable Method*

```
public boolean isCellEditable (int row, int column) {
      return false;
}
```

■ **Note** Unlike the other methods shown so far, `isCellEditable` is already implemented in `AbstractTableModel`. As it happens, the `AbstractTableModel` implementation provides the same functionality as our overridden method. We therefore did not need to override this method; however, doing so makes it easier for you to experiment with this method.

Finally, our `DVDTableModel` class contains two methods beyond those required by the `TableModel` interface. These extra methods are included as a matter of convenience. As mentioned earlier in this chapter, the `DVD` object, which is used by the rest of the system, is not the same Model that is used by the View. In this case, the View object is a `JTable` and the Model object is the `DVDTableModel`. The `GUIController` class must go through the process of converting a `DVD` object (or a collection of `DVD` objects) to a `DVDTableModel` object. To make this task easier, the `DVDTableModel` implements the two methods in Listing 8-16.

Listing 8-16. *Convenience Methods Within the DVDTableModel Class*

```
public void addDVDRecord (String upc, String name, String director,
                          String leadActor, String supportingActor,
                          String composer, int numberOfCopies) {
    String [] temp = {upc, name, director, leadActor, supportingActor,
                      composer, Integer.toString(numberOfCopies)};
    this.dvdRecords.add(temp);
}

public void addDVDRecord (DVD dvd) {
        addDVDRecord(dvd.getUPC(), dvd.getName(), dvd.getDirector(),
                     dvd.getLeadActor(), dvd.getSupportingActor(),
                     dvd.getComposer(), dvd.getCopy());
}
```

The methods in Listing 8-16 take in a `DVD` object (or the equivalent data) and append the new row to the data set encapsulated within the `DVDTableModel`. The first method receives the data contained in a `DVD` object as parts, whereas the second method simply requires a `DVD` object. In order to add a `DVD` to the `DVDTableModel`, the `GUIController` will simply call one of these two methods, thus avoiding a conversion process from within the `GUIController` each time an alteration to the data view occurs.

■ **Tip** The preceding `DVDTableModel` has the capability to convert DVD model objects one at a time, but sometimes it is more convenient to provide a method that will convert entire collections of model objects. That way, a search method could use a single call to convert its entire set of search results.

Using the TableModel with a JTable

Once a `TableModel` has been implemented, using it with a `JTable` is a simple task. A `TableModel` may be specified in the constructor of a `JTable` object. For example, the following snippet will create a new `JTable` using the `DVDTableModel` that was described in the previous section:

```
JTable table = new JTable (new DVDTableModel ());
```

The preceding statement will create a `JTable` using a `DVDTableModel`, but the table display will be empty, since no data is contained in the specified `DVDTableModel` instance.

■ **Note** A `JTable` instance can have its internal table model modified after it is instantiated. The `setModel` method updates a `JTable`'s internal `TableModel` member and therefore updates the data the `JTable` displays. Data can also be added to a `JTable`'s `TableModel` reference by calling the method `getModel` method on a `JTable` instance. In our case, this method can be called to get a reference to the `JTable`'s internal `DVDTableModel` reference. Then the method `addRecord` can be called to add a row to a `JTable`.

Because alterations to a `JTable`'s `TableModel` translates into an updated View, the client must be set up to take advantage of this schema. The `MainWindow` class contains the private member

```
private DVDTableModel tableData;
```

This data member will always hold the main `JTable`'s `TableModel`. Because all data transfer between the View and the Controller is done via a `DVDTableModel` object, this member is always updated to reflect changes to the database's state.

Once the database has been updated or queried, the resulting table model is placed into the `tableData` member. After calling the Controller, the `MainWindow` class calls its internal private method `setupTable`. This method contains the statements in Listing 8-17.

Listing 8-17. *The setupTable Method*

```
private void setupTable() {
        // Preserve the previous selection
        int index = mainTable.getSelectedRow();
        String prevSelected = (index >= 0)
                            ? (String) mainTable.getValueAt(index, 0)
                            : "";
```

```
    // Reset the table data
    this.mainTable.setModel(this.tableData);

    // Reselect the previous item if it still exists
    for (int i = 0; i < this.mainTable.getRowCount(); i++) {
        String selectedUpc = (String) mainTable.getValueAt(i, 0);
        if (selectedUpc.equals(prevSelected) ) {
            this.mainTable.setRowSelectionInterval(i,i);
           break;
        }
    }
}
```

Notice that the `setupTable` method refreshes the `MainWindow`'s `JTable` instance by calling the `setTableModel` method. As soon as the table's internal `TableModel` is replaced, the View will refresh and display the updated data set. If the `JTable` does not update its contents, the `updateUI` method can be called, effectively forcing the table to redisplay its contents.

Also notice that the `setupTable` method stores the last selected row before refreshing the table model. After the table model is reset, the method loops through the new data set and locates the previously selected row by the UPC number. After that, the table's `setRowSelectionInterval` method is called to reselect the previous row.

The JScrollPane

Occasionally we get a situation where we need to display more on the screen than will fit. For the Denny's DVDs application, it is possible that there could be so many DVDs in the database that displaying them all on screen simultaneously is impossible.

In these cases, we can add the component that is too large for the screen to a `JScrollPane`, which will put scrollbars around the component, *where necessary*, to allow scrolling to a different region and viewing the contents there. Note that, by default, the scrollbars are only shown when needed—if they are not needed, they will not appear.

Listing 8-18 shows an example of an application where there is too much data to appear in the desired text area, and the way this would appear on screen is shown in Figure 8-20.

Listing 8-18. *An Application That Has Too Much Data to Appear in the Window*

```
import javax.swing.*;


public class MyFrame {
    public static void main(String[] args) throws Exception {

        JFrame theFrame = new JFrame();
        theFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
        String outputString = "";
        for (int i = 0; i < 100; i++) {
            outputString += "The quick brown fox jumped over the lazy dog. ";
        }

        JTextArea textDisplay = new JTextArea(20, 60);
        textDisplay.setLineWrap(true);
        theFrame.add(textDisplay);
        theFrame.pack();
        textDisplay.setText(outputString);
        theFrame.setVisible(true);
    }
}
```

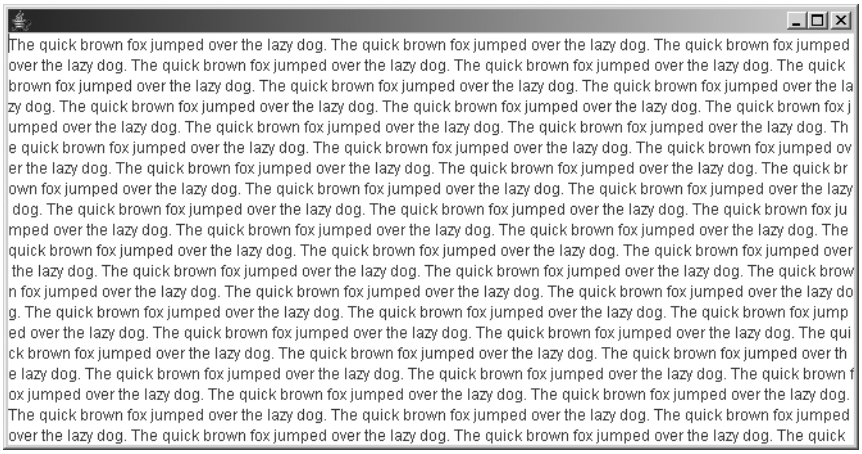


Figure 8-20. *Display when attempting to show too much text*

In most cases, the simplest solution to this problem is changing the line

```
theFrame.add(textDisplay);
```

to read

```
theFrame.add(new JScrollPane(textDisplay));
```

The results of this change are shown in Figure 8-21.

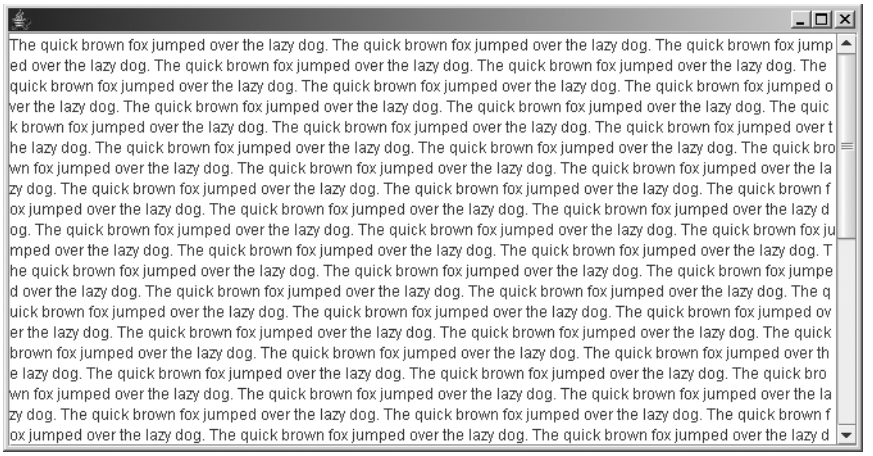


Figure 8-21. *Making a GUI object scrollable using a JScrollPane*

Bringing Denny's DVDs Together

Now it is time to bring the Denny's DVDs application together. The best way to launch the Denny's DVDs client is through the use of a shell class. The entire responsibility of this class is to initiate the launch of the application. Usually, an application shell will initialize some global settings, such as the application's look and feel, and resolve any preconditions that must exist for the application.

For the DVD client application, the main precondition is setting up the look and feel. In order to provide users with a look and feel they are familiar with, we have chosen to set the look and feel based on the system look and feel.

We also check the command-line options provided, as the command-line options set the mode for the application.

Application Startup Class

The `ApplicationRunner` class is essentially an application loader. The only thing the main method creates is an object of type `ApplicationRunner`. The `ApplicationRunner` class's constructor sets up the application's look and feel and instantiates either the `MainWindow` or the `ServerWindow` class, as shown in Listing 8-19.

Listing 8-19. *Denny's DVDs Main Application Loader*

```
public ApplicationRunner(String[] args) {

    if (args.length == 0 || "alone".equalsIgnoreCase(args[0])) {
        // Create an instance of the main application window
        new MainWindow(args);
    } else if ("server".equalsIgnoreCase(args[0])) {
        new ServerWindow();
    } else {
```

```
            log.info("Invalid parameter passed in startup: " + args[0]);
            // Logging may be turned off, or may be going to a file, so
            // send usage information to the error output (usually the screen).
            System.err.println("Command line options may be one of:");
            System.err.println("\"server\" - the server application will start");
            System.err.println("\"alone\"  - client start in non networked mode");
            System.err.println("\"\"       - (no command line option): " +
                               "networked client will start");
        }
    }
```

The `ApplicationRunner` class also contains the static method `handleException`. The method takes in a `String` argument and prompts the user with an error message box containing the string message. This method exists for the sole purpose of presenting application error feedback to the user. All exceptions that occur in the `MainWindow` class will be caught, and during the try/catch process a call to the `handleException` method will be made that will display error information for the user.

The Client GUI

The bulk of the GUI logic is contained within the `MainWindow` class, which we will present here.

GUI Design and Layout

As mentioned in Chapter 2 and at the start of this chapter, we recommend you start by hand-sketching your GUI. To give an example of what we mean, consider the sketch of the GUI we will be developing for our client application that is shown in Figure 8-22.

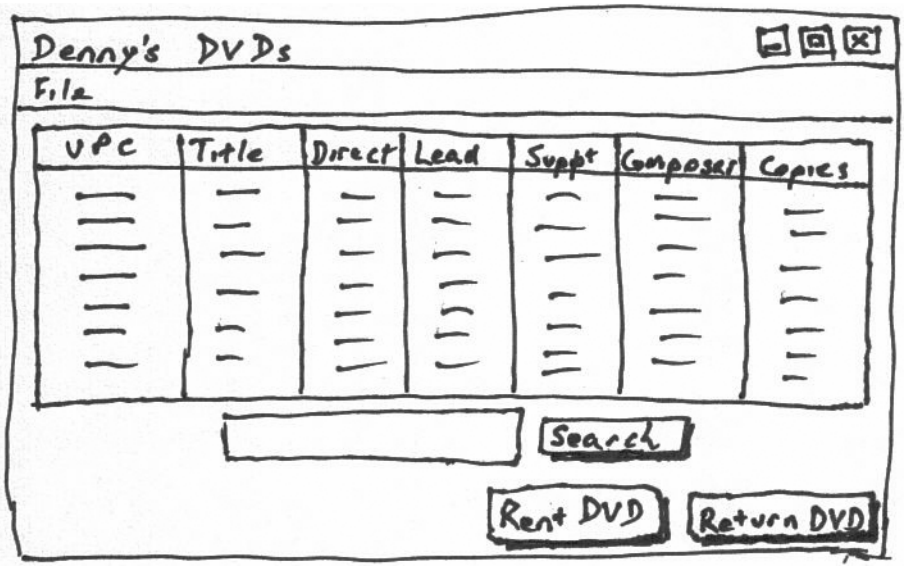


Figure 8-22. *A hand-drawn sketch of the client GUI interface*

Sketching a prototype in this way gives us the following advantages over coding the GUI directly:

- We can sketch a sample GUI quicker than we can write code to create a mock-up of the application.
- Following that point, if the end user would like a change made to the interface, you won't have to throw out code.
- The sketch can be shown to your sample audience/testers anywhere—you do not need a computer handy.
- When you show it to potential users, they will *know* that it is a sketch, and not a complete application, whereas if you show potential users a mocked-up GUI, there is the tendency for the users to believe that a large proportion of the work is complete.
- Having made an up-front decision about what looks good, we are more likely to stick to it. When coding without the up-front sketch, there is the temptation to change parts of the design whenever it appears to difficult to do.
- Having made an up-front decision about what will appear in the GUI, there is less temptation to add more as we go along.

Just like assembling a puzzle, there are many ways to solve a single problem. The preceding layout structure is, of course, merely the way we have chosen to lay out Denny's DVD application—you might choose a totally different layout for your application. The Denny's DVDs application interface takes a simple approach that incorporates very few additional bells and whistles. A basic Swing implementation is all that Sun requires for the SCJD exam, but you can choose to go above and beyond what is required and add more features than required. A `JToolBar`, for example, could be provided for additional ease of use, even though it is not specifically required to pass the exam. Whether to add such features is up to you. On the one hand, there are many features that you can add that will make your GUI more user friendly, and hence improve your score. On the other hand, your instructions may warn against going beyond specifications. Where to draw the line is up to you.

When we look at the sketch in Figure 8-22, we can see that there are three major areas (excluding the title bar and menu bar), as shown in Figure 8-23.

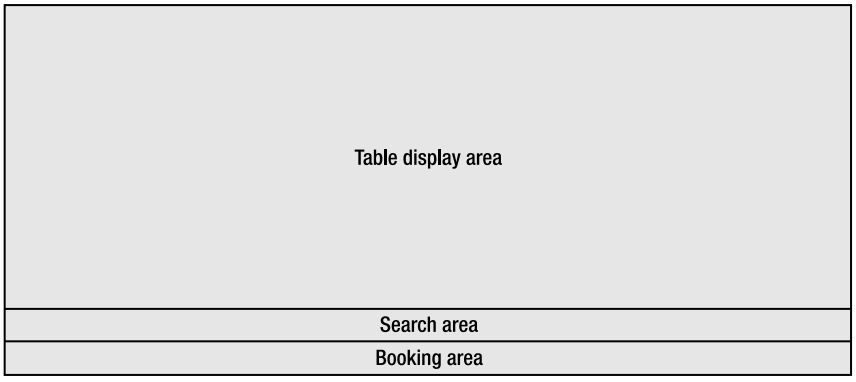


Figure 8-23. *The three main areas of the client GUI application*

We might be tempted to try to add our components directly into a `BorderLayout`; however, if the user should resize the screen, the object to the north and south will only stretch horizontally—it is the object in the center that will stretch vertically. This could lead to an awkward-looking screen, as demonstrated in Figure 8-24.

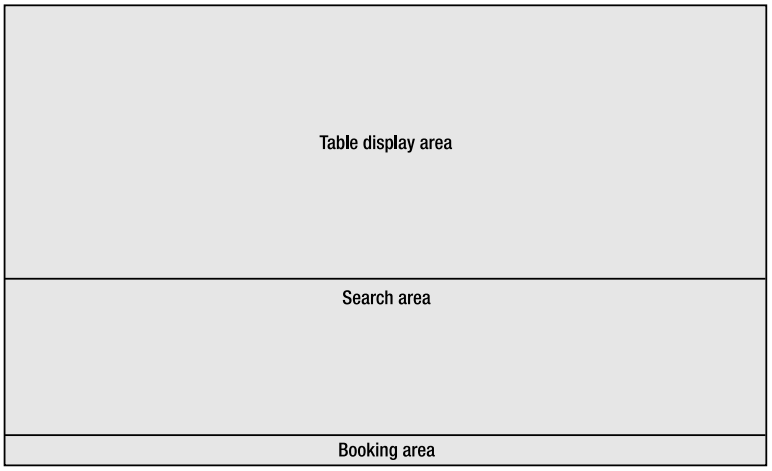


Figure 8-24. *Resizing the GUI application if the table is in the north, the search panel is in the center, and the booking panel is in the south*

■ **Note** This is an example of where creating a sketch of our application before coding helps ensure we end up with a GUI that the users want, not what is easiest for us. Without that sketch, we might be tempted to change the user-approved design just to make our life easier.

So what we need to do is to change how we break up the GUI. The major amount of information is going to be displayed in the table, so that is the component we want in the center. This means that we are going to need to create another panel that will contain both search options and booking options to go in the south of the main panel, as shown in Figure 8-25.

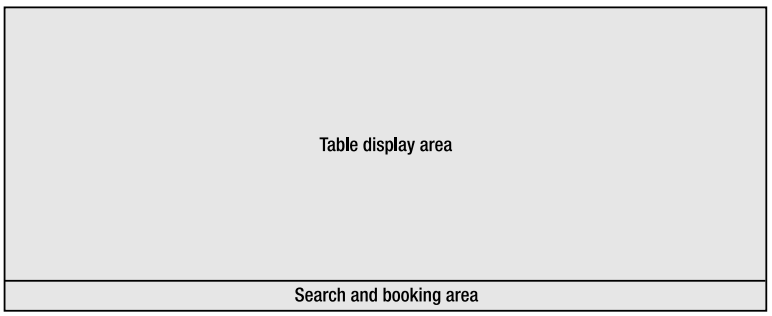


Figure 8-25. *The new panels for the main window*

We will create a `JPanel` that contains the search panel and the booking panel. Since these two panels do not need to change size when the user resizes the window, we will place the search panel in the north, and the booking panel in the south, as shown in Figure 8-26.

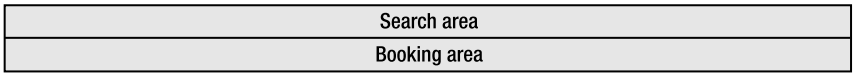


Figure 8-26. *The new panel for searching and for booking*

We can then take the panel shown in Figure 8-26 and add it to the south of the panel shown in Figure 8-25, which results in a GUI that will resize the table pane where needed, while leaving the search and booking panes the desired size. The code shown in Listing 8-20 demonstrates this techinique. Note, however, that labels are used in place of the real components—this is done so that the code may be easier to understand.

Listing 8-20. *Combining Several Panels to Form One Overall GUI*

```
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;

public class MyFrame {
    public static void main(String[] args) throws Exception {
        Border border = BorderFactory.createLineBorder(Color.BLACK);
        JFrame theFrame = new JFrame();
        theFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // This is the panel for the table - that is all it contains
        JPanel tablePanel = new JPanel();
        tablePanel.setBorder(border);
        JLabel table = new JLabel("Table display area",SwingConstants.CENTER);
        table.setPreferredSize(new Dimension(650,225));
        tablePanel.add(table);

        // The search options panel
        JPanel searchPanel = new JPanel();
        searchPanel.setBorder(border);
        JLabel search = new JLabel("Search area",SwingConstants.CENTER);
        search.setPreferredSize(new Dimension(650,15));
        searchPanel.add(search);

        // the booking options panel
        JPanel bookPanel = new JPanel();
        bookPanel.setBorder(border);
        JLabel book = new JLabel("Booking area",SwingConstants.CENTER);
        book.setPreferredSize(new Dimension(650,15));
        bookPanel.add(book);
```

```
        // The search & booking options panels are both added to an extra panel
        JPanel optionsPanel = new JPanel(new BorderLayout());
        optionsPanel.add(searchPanel, BorderLayout.NORTH);
        optionsPanel.add(bookPanel, BorderLayout.SOUTH);

        // The the tablePanel and optionsPanel are added to the JFrame
        theFrame.add(tablePanel, BorderLayout.CENTER);
        theFrame.add(optionsPanel, BorderLayout.SOUTH);

        theFrame.pack();
        theFrame.setVisible(true);
    }
}
```

Listing 8-20 shows roughly how we will be laying out the `MainWindow` for our client application. We will be providing the complete code for the `MainWindow` class next; however, it is worthwhile reading through the code presented in Listing 8-20 to ensure that you are comfortable with the layout concepts before continuing.

The contents of the screen generated by Listing 8-20 are shown in Figure 8-27.

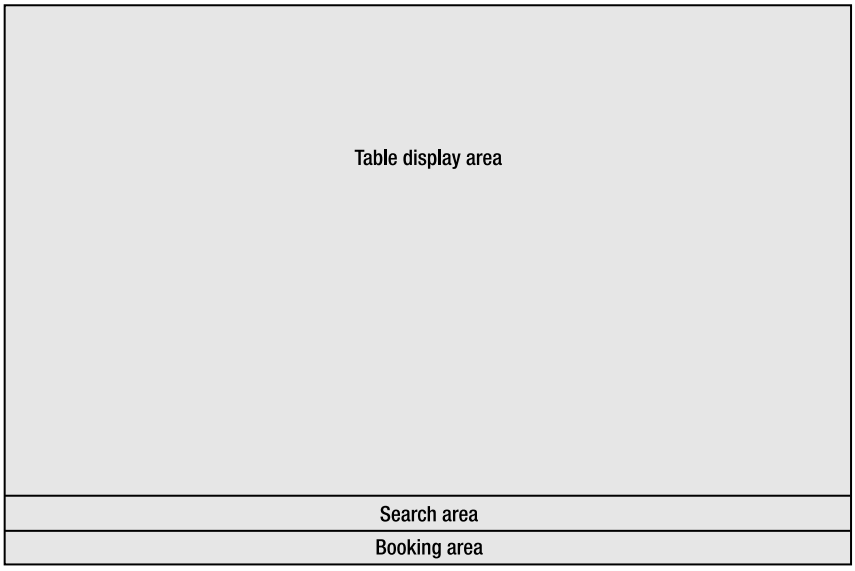


Figure 8-27. *The result of combining several frames*

The search area and the booking area each contain multiple items laid out in a row. We can use the `FlowLayout` for these two panes, as will be shown in Listing 8-21 in the next section.

■ **Tip** This technique of developing small sections of code to show a concept is very useful in developing and debugging code. It is often the case when attempting to incorporate some feature that does not appear to work correctly, or trying to debug some existing code, that a large portion of the code is not relevant to your problem. When you create a simple application just for testing the issue, you do not have irrelevant code to distract you. It also means that you have a small bit of code with which to ask a friend or colleague for help if necessary—asking a friend or colleague to help debug a thousand-line application is really pushing friendship.

The MainWindow

The `MainWindow` class extends `JFrame` and is the actual implementation of the Denny's DVDs main window. The constructor of the `MainWindow` goes through the process of setting up the application menu bar, the main data table, adding the `DVDScreen` (described after Listing 8-22), and setting the main window in the center of the operating system screen.

First, the `DVDMainWindow` constructor creates an instance of its super class, `JFrame`, setting the title of the application. Following this, a dialog box is created where the user can enter the location of the database (described in the upcoming section, "Specifying the Database Location"). The initialization procedure then continues to create the menu bar and all menu items.

As shown in Listing 8-21, there is one `JMenuBar` for the frame. The menu bar may have several `JMenus` attached (for example, one for the File menu, one for the Help menu, and so on). Each menu may have several `JMenuItems` attached—one for each action your user is likely to perform via a menu.

Each menu and menu item may have an optional mnemonic key and an optional icon attached. We have shown attaching the mnemonic key F to the File menu, and the mnemonic key Q to the Quit menu item. If the user presses the Alt and F keys, the File menu will pop down, and if they then press the Q key, the application will quit.

Normally each menu item has an `actionListener` attached to respond to events. We have shown attaching an instance of the `QuitApplication` class to the `quitMenuItem` (the `QuitApplication` class will be shown in Listing 8-22).

After the menus have been configured, and the data loaded from the database, an instance of the `DVDScreen` class is added to the `MainWindow` frame. `DVDScreen` is a `JPanel` that contains the elements described in the "GUI Design and Layout" section earlier. It will be shown in Listing 8-23.

Finally, an initial size for the application window is set, and the application window is centered on the screen.

Listing 8-21. *The MainWindow Constructor: Setting Up the Menu*

```
public MainWindow(String[] args) {
    super("Denny's DVDs");
    this.setDefaultCloseOperation(this.EXIT_ON_CLOSE);

    ApplicationMode connectionType = (args.length == 0)
                                   ? ApplicationMode.NETWORK_CLIENT
                                   : ApplicationMode.STANDALONE_CLIENT;
```

```
// find out where our database is
DatabaseLocationDialog dbLocation =
    new DatabaseLocationDialog(this, connectionType);

try {
    controller = new GuiController(dbLocation.getNetworkType(),
                                   dbLocation.getLocation(),
                                   dbLocation.getPort());
} catch (GUIControllerException gce) {
    ApplicationRunner.handleException(
            "Failed to connect to the database");
}


// Add the menu bar
JMenuBar menuBar = new JMenuBar ();
JMenu fileMenu = new JMenu ("File");
JMenuItem quitMenuItem = new JMenuItem ("Quit");
quitMenuItem.addActionListener(new QuitApplication ());
quitMenuItem.setMnemonic(KeyEvent.VK_Q);
fileMenu.add(quitMenuItem);
fileMenu.setMnemonic(KeyEvent.VK_F);
menuBar.add(fileMenu);

this.setJMenuBar(menuBar);

// A full data set is returned from an empty search
try {
    tableData = controller.getDVDs();
    setupTable();
} catch (GUIControllerException gce) {
    ApplicationRunner.handleException(
            "Failed to acquire an initial DVD list." +
            "\nPlease check the DB connection.");
}

this.add(new DvdScreen());

this.pack();
this.setSize(650, 300);

// Center on screen
Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
int x = (int) ((d.getWidth() - this.getWidth())/ 2);
int y = (int) ((d.getHeight() - this.getHeight())/ 2);
this.setLocation(x, y);
this.setVisible(true);
```

If the user chooses to quit the application using the Quit menu item, the event handler will call the `QuitApplication` class. This is one of the simplest event handlers we could write—all it does is call `System.exit(0)`. It is shown in Listing 8-22.

Listing 8-22. *The QuitApplication ActionListener*

```
private class QuitApplication implements ActionListener {
    public void actionPerformed (ActionEvent ae) {
        System.exit(0);
    }
}
```

Listing 8-23 contains the `DVDScreen` constructor. The major components of this have all been introduced in the earlier sections.

`DVDScreen` starts by adding a scroll pane for the table holding the DVDs—this is added to the center of the `JPanel`.

Following this, the panel for the search options is created, along the way creating the text field for entering the search parameters, and the button to begin a search. An action listener is added to the search button, and will be described after Listing 8-23 since it is a little more involved than our application listener for the Quit menu item.

Then the buttons and panel for the Rent and Return options are created, along with their action listeners.

A bottom panel is created to hold both the search options and the hiring options, with the search options panel added to the north, and the hiring options panel added to the south. The bottom panel is then added to the south of the master `JPanel`.

Finally the table is configured and some tooltips are added.

Listing 8-23. *The DVDScreen*

```
public DvdScreen() {
    this.setLayout(new BorderLayout());
    JScrollPane tableScroll = new JScrollPane (mainTable);
    tableScroll.setSize(500, 250);

    this.add(tableScroll, BorderLayout.CENTER);

    // Set up the search pane
    JButton searchButton = new JButton ("Search");
    searchButton.addActionListener(new SearchDVD ());
    searchButton.setMnemonic(KeyEvent.VK_S);
    // Search panel
    JPanel searchPanel = new JPanel(new FlowLayout(FlowLayout.CENTER));
    searchPanel.add(searchField);
    searchPanel.add(searchButton);
```

```
        // Setup rent and return buttons
        JButton rentButton = new JButton ("Rent DVD");
        JButton returnButton = new JButton ("Return DVD");

        // Add the action listenters to rent and return buttons
        rentButton.addActionListener(new RentDVD ());
        returnButton.addActionListener(new ReturnDVD ());
        // Set the rent and return buttons to refuse focus
        rentButton.setRequestFocusEnabled(false);
        returnButton.setRequestFocusEnabled(false);
        // Add the keystroke mnemonics
        rentButton.setMnemonic(KeyEvent.VK_R);
        returnButton.setMnemonic(KeyEvent.VK_U);
        // Create a panel to add the rental a remove buttons
        JPanel hiringPanel = new JPanel(new FlowLayout(FlowLayout.RIGHT));
        hiringPanel.add(rentButton);
        hiringPanel.add(returnButton);

        // bottom panel
        JPanel bottomPanel = new JPanel (new BorderLayout ());
        bottomPanel.add(searchPanel, BorderLayout.NORTH);
        bottomPanel.add(hiringPanel, BorderLayout.SOUTH);

        // Add the bottom panel to the main window
        this.add(bottomPanel, BorderLayout.SOUTH);

        // Set table properties
        mainTable.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        mainTable.setAutoResizeMode(JTable.AUTO_RESIZE_ALL_COLUMNS);
        mainTable.setToolTipText("Select a DVD record to rent or return.");

        // Add Tool Tips
        returnButton.setToolTipText(
                    "Return the DVD item selected in the above table.");
        rentButton.setToolTipText(
                    "Rent the DVD item selected in the above table.");
        searchField.setToolTipText(
                    "Enter infromation about a DVD you want to locate.");
        searchButton.setToolTipText("Submit the DVD search.");

    }
}
```

The GUI created by the `MainWindow` class and the `DVDScreen` class can be seen in Figure 8-28.

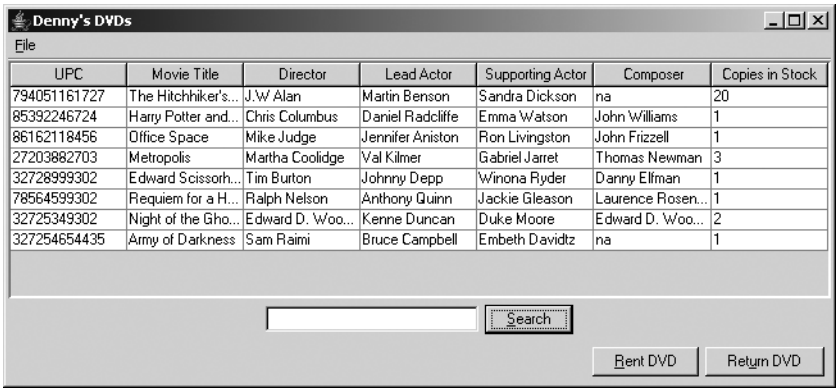


Figure 8-28. *The GUI created by the MainWindow class and the DVDScreen class*

Just like all other action handlers in the Denny's DVDs application that communicate with the `GUIController`, the search action shown in Listing 8-24 must be ready to receive a `GUIControllerException` from the `GUIController`. This particular method, however, handles the exception a little differently. The `GUIControllerException` that is thrown from the `find(String query)` method may contain a `PatternSyntaxException` as its cause. This particular exception is thrown if the user enters a search string that is not a properly formatted regular expression. In this case, the exception message may contain important information for the user pertaining to the incorrect syntax of the search. In order to present this information to the user, J2SE's exception chaining facility is put to use. When the `actionPerformed()` method in Listing 8-24 catches an exception, it checks the exception's cause to see if it is of type `PatternSyntaxException`. If it is this type, the exception message is added to the message that is passed to the `handleException` method. The end result is a very useful error dialog box that looks like the one shown in Figure 8-29.

Listing 8-24. *The SearchDVD Event Handler*

```
private class SearchDVD implements ActionListener {
    public void actionPerformed (ActionEvent ae) {
        previousSearchString = searchField.getText();
        try {
            tableData = controller.find(previousSearchString);
            setupTable();
        } catch (GUIControllerException gce) {
            // Inspect the exception chain
            Throwable rootException = gce.getCause();
            String msg = "Search operation failed.";
            // If a syntax error occurred, get the message
            if (rootException instanceof PatternSyntaxException) {
                msg += ("\n" + rootException.getMessage());
            }
            ApplicationRunner.handleException(msg);
```

```
                previousSearchString = "";
            }
            searchField.setText("");
        }
    }
```

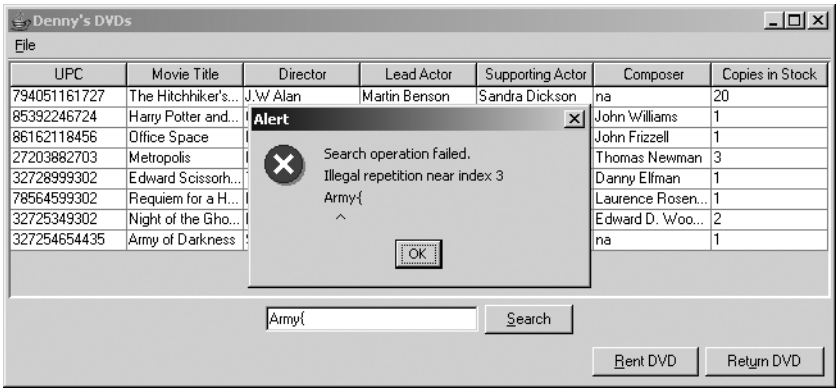


Figure 8-29. *The error message from a regular expression compilation failure*

Specifying the Database Location

When users start the client or server applications, they need to be able to specify where the database is located.

For the Denny's DVDs application, we have determined that this entails the following:

- In stand-alone client mode, the user must specify where the physical file is located.
- In networked client mode, the user must be able to specify the URL or IP address of the server computer, the type of server running (RMI or sockets), and the port number the server is using.
- In server mode, the user must be able to specify where the physical file is located, the type of server (RMI or sockets), and the port number the server will use.

■ **Note** You should not need anything so complex for your solution. Some of the options we are allowing for are specifically to allow for all the options of this book.

To allow these options to be entered in client mode, we are going to develop a dialog box, as shown in Figure 8-30. The major part of this dialog box is the reusable panel shown in Figure 8-31.

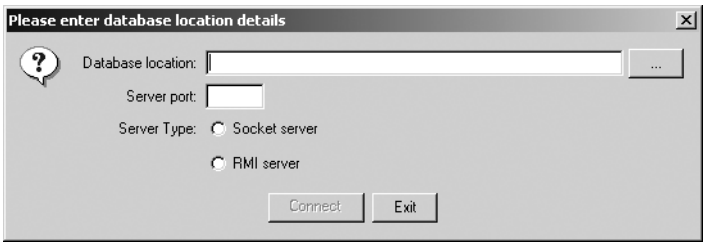


Figure 8-30. *The client application's dialog box for entering the database location*

Figure 8-31. *The JPanel for getting common parameters in both the client and the server application*

In Listing 8-28 we will see that the constructor for the common frame will change which options are displayed, and what tooltip text is displayed, depending on a parameter in the constructor. This allows us to use the same frame for stand-alone client, networked client, and server applications.

For the main client GUI, we used the simple graphical layout managers: `BorderLayout` and `FlowLayout`. These layout managers will serve in most cases. However, in some circumstances you may find that you want to lay out a panel in some form that cannot be readily achieved using these simple layout managers. The common frame is one such case—we want to have up to four rows of components, but each component has a label that we want to align vertically on its right edge.

In this section we introduce one of the most powerful layout managers—the `GridBagLayout`. We deliberately delayed introducing this layout manager until now, as it is more complicated to use than the other layout managers.

■ **Tip** Although this layout manager is very powerful, and works well in the Denny's DVD example application, you may find that you do not need to use this layout manager in your application. We recommend that you use the simplest solution that works for you—you will not get extra marks for proving that you can use all the layout managers.

The GridBagLayout

Java has two grid-based layout managers: the `GridLayout` and the `GridBagLayout` manager.

The `GridLayout` is the simpler of the two; however, it forces all cells in the grid to be the same size, as shown in Figure 8-32. The program that produced this is shown in Listing 8-25.

Listing 8-25. *A Simple GridLayout Application*

```
import java.awt.*;
import javax.swing.*;

public class MyFrame {
    public static void main(String[] args) throws Exception {
        JFrame theFrame = new JFrame();
        theFrame.setLayout(new GridLayout(2,2));
        theFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        theFrame.add(new JButton("One"));
        theFrame.add(new JLabel("A very long component"));
        theFrame.add(new JTextField("Three"));
        theFrame.add(new JTextArea("Four\nFive\nSix"));

        theFrame.pack();
        theFrame.setVisible(true);
    }
}
```

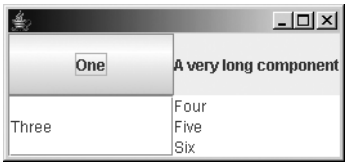


Figure 8-32. *Example of GridLayout demonstrating consistent cell sizes*

As you can see in Figure 8-32, the width of all components is the same as the width of the longest component—the `JLabel` with the contents "A very long component"—and the height of all components is the same as the height of the tallest component—the `JTextArea` that contains multiple lines of text.

While this layout can be very useful if most of the components are the same size, it is not very useful when you have considerable differences in component sizes, as we do in our common frame. If we were to use the `GridLayout`, then the space allocated to the labels would be the same as the space allocated to the largest component—the Database location field.

The `GridBagLayout` also works on a grid of cells; however, components are allowed to occupy more than one cell, the width of a column is calculated based on the width of the widest column that does not span multiple columns, and the height of a row is calculated based on the height of the tallest component in that row that does not span multiple rows.

We will start off with a simple example that contains the same components as used in Figure 8-32, but this time we will use the `GridBagLayout`. Listing 8-26 shows the program that creates the GUI shown in Figure 8-33.

Listing 8-26. *A Simple GridBagLayout Application Similar to Listing 8-25*

```
import java.awt.*;
import javax.swing.*;

public class MyFrame {
    static GridBagLayout grid = new GridBagLayout();
    static GridBagConstraints constraints = new GridBagConstraints();

    public static void main(String[] args) throws Exception {
        JFrame theFrame = new JFrame();
        theFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        theFrame.setLayout(grid);

        constraints.anchor = GridBagConstraints.WEST;

        JButton one = new JButton("One");
        grid.setConstraints(one, constraints);
        theFrame.add(one);

        constraints.gridwidth = GridBagConstraints.REMAINDER;
        theFrame.add(constrain(new JLabel("A very long component")));

        constraints.weightx = 0.0;
        constraints.gridwidth = 1;
        theFrame.add(constrain(new JTextField("Three")));
        theFrame.add(constrain(new JTextArea("Four\nFive\nSix")));

        theFrame.pack();
        theFrame.setVisible(true);
    }

    private static Component constrain(Component c) {
        grid.setConstraints(c, constraints);
        return c;
    }
}
```

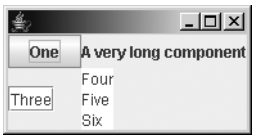


Figure 8-33. *Example of GridBagLayout demonstrating inconsistent cell sizes*

As can be seen in Figure 8-33, the components are no longer forced to the same height and width as the highest and widest components. The first row's height has no relationship to the second row's height, and the first column's width has no relationship to the second column's width.

The basic concept when using `GridBagLayout` is to configure the constraints to be used for the component, notify the `GridBagLayout` manager of these constraints, and then add the component to the pane. This is best exemplified in the following code snippet from Listing 8-26:

```
constraints.anchor = GridBagConstraints.WEST;

JButton one = new JButton("One");
grid.setConstraints(one, constraints);
theFrame.add(one);
```

Prior to this code snippet, the constraints were set to default values (starting in row 1, column 1, centered in the grid, occupying one grid position, etc.). The first line changes the location of each component so that they will now be left-justified. We then created our component and notified the `GridBagLayout` that the specified constraints were to be used with that particular component. Finally, we added the component to the panel.

■ **Note** Only one set of constraints is used throughout the application—when the `setConstraints` method is called, the layout manager clones the constraints provided. This makes it easier for developing, as you can set common constraint configuration at the start of your layout code and use those constraints throughout the program.

Since components can span multiple rows and multiple columns, the `GridBagLayout` cannot determine whether or not a component is the last component. Therefore, before adding the last item to row 1, we must specify via the constraints that this will be the last component in the row. The following line does this:

```
constraints.gridwidth = GridBagConstraints.REMAINDER;
```

As noted, we use the same set of constraints for all components. We set the constraints to specify that the component will be the last component in the row, and then used that constraint when adding the component to the container. However, if we do not reset the constraint, all future components will also appear as the last component in the row. The following line does this:

```
constraints.gridwidth = 1;
```

To close this section, we will demonstrate how to specify that components span multiple rows or columns. The program to do this is shown in Listing 8-27, and the output is shown in Figure 8-34.

Listing 8-27. *A GridBagLayout Application Demonstrating Components Spanning Multiple Cells*

```
import java.awt.*;
import javax.swing.*;

public class MyFrame {
    static GridBagLayout grid = new GridBagLayout();
    static GridBagConstraints constraints = new GridBagConstraints();

    public static void main(String[] args) throws Exception {
        JFrame theFrame = new JFrame();
        theFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        theFrame.setLayout(grid);

        constraints.fill = GridBagConstraints.BOTH;

        constraints.gridheight = 2;
        theFrame.add(constrain(new JButton("Deep")));

        constraints.gridheight = 1;
        constraints.gridwidth = GridBagConstraints.REMAINDER;
        theFrame.add(constrain(new JButton("Wide")));

        constraints.gridwidth = 1;
        theFrame.add(constrain(new JButton("One")));
        theFrame.add(constrain(new JButton("Two")));

        theFrame.pack();
        theFrame.setVisible(true);
    }

    private static Component constrain(Component c) {
        grid.setConstraints(c, constraints);
        return c;
    }
}
```

Figure 8-34. *Example of GridBagLayout demonstrating components spanning multiple cells*

■ **Note** `GridBagLayout` is one of the most useful layout managers available, and it is capable of far more than we can show in one section of one chapter. Unfortunately, going into all its capabilities is beyond the scope of this book.

The Common Database Location Frame

Now that the basics of the `GridBagLayout` have been explained, we can show the code that creates the common frame for displaying the database location.

The `ConfigOptions` constructor stores a local variable to show what mode it was configured with, and then constructs a `GridBagLayout` with a gap between components of 2 pixels in every direction.

Regardless of which mode we are in, we will be displaying the label for the Database location field, so we add that next.

The remaining options are configured depending on the mode we are in. If a component or option does not make sense in the mode we are in, then it is not added to the panel. If it does make sense to add it, then it is added, and the tooltip text is set relevant to the mode. Listing 8-28 demonstrates how the components used within a particular mode are laid out using the `GridBagLayout`.

Listing 8-28. *The ConfigOptions Constructor*

```
public ConfigOptions(ApplicationMode applicationMode) {
    super();
    this.applicationMode = applicationMode;

    GridBagLayout gridbag = new GridBagLayout();
    GridBagConstraints constraints = new GridBagConstraints();
    this.setLayout(gridbag);

    // Standard options
    // ensure there is always a gap between components
    constraints.insets = new Insets(2,2,2,2);

    // Build the Data file location row
    JLabel dbLocationLabel = new JLabel(DB_LOCATION_LABEL);
    gridbag.setConstraints(dbLocationLabel, constraints);
    this.add(dbLocationLabel);

    if (applicationMode == ApplicationMode.NETWORK_CLIENT) {
        locationField.setToolTipText(DB_IP_LOCATION_TOOL_TIP);
        constraints.gridwidth = GridBagConstraints.REMAINDER; //end row
    } else {
        locationField.setToolTipText(DB_HD_LOCATION_TOOL_TIP);
        // next-to-last location in row
        constraints.gridwidth = GridBagConstraints.RELATIVE;
```

```
locationField.addFocusListener(new ActionHandler());
locationField.setName(DB_LOCATION_LABEL);
gridbag.setConstraints(locationField, constraints);
this.add(locationField);

if ((applicationMode == ApplicationMode.SERVER)
        || (applicationMode == ApplicationMode.STANDALONE_CLIENT)) {
    browseButton.addActionListener(new BrowseForDatabase());
    constraints.gridwidth = GridBagConstraints.REMAINDER; //end row
    gridbag.setConstraints(browseButton, constraints);
    this.add(browseButton);
}

if ((applicationMode == ApplicationMode.SERVER)
        || (applicationMode == ApplicationMode.STANDALONE_CLIENT)) {
    // Build the Server port row if applicable
    constraints.weightx = 0.0;

    JLabel serverPortLabel = new JLabel(SERVER_PORT_LABEL);
    constraints.gridwidth = 1;
    constraints.anchor = GridBagConstraints.EAST;
    gridbag.setConstraints(serverPortLabel, constraints);
    this.add(serverPortLabel);

    portNumber.addFocusListener(new ActionHandler());
    portNumber.setToolTipText(SERVER_PORT_TOOL_TIP);
    portNumber.setName(SERVER_PORT_LABEL);
    constraints.gridwidth = GridBagConstraints.REMAINDER; //end row
    constraints.anchor = GridBagConstraints.WEST;
    gridbag.setConstraints(portNumber, constraints);
    this.add(portNumber);

    // Build the Server type option row 1 if applicable
    constraints.weightx = 0.0;

    JLabel serverTypeLabel = new JLabel("Server Type: ");
    constraints.gridwidth = 1;
    constraints.anchor = GridBagConstraints.EAST;
    gridbag.setConstraints(serverTypeLabel, constraints);
    this.add(serverTypeLabel);

    constraints.gridwidth = GridBagConstraints.REMAINDER; //end row
    constraints.anchor = GridBagConstraints.WEST;
    gridbag.setConstraints(socketOption, constraints);
    socketOption.setActionCommand(SOCKET_SERVER_TEXT);
    socketOption.addActionListener(new ActionHandler());
    this.add(socketOption);
```

```
        // Build the Server type option row 2 if applicable
        constraints.weightx = 0.0;

        constraints.gridwidth = GridBagConstraints.REMAINDER; //end row
        constraints.anchor = GridBagConstraints.WEST;
        constraints.gridx = 1;
        gridbag.setConstraints(rmiOption, constraints);
        rmiOption.addActionListener(new ActionHandler());
        rmiOption.setActionCommand(RMI_SERVER_TEXT);
        this.add(rmiOption);

        ButtonGroup serverTypesGroup = new ButtonGroup();
        serverTypesGroup.add(socketOption);
        serverTypesGroup.add(rmiOption);
    }
}
```

If the choice between RMI and sockets is displayed, we want to display two radio buttons under each other, with only one label on the left.

There are several ways we could have approached this. We could have set the label to be two columns deep, or we could have created another frame to hold the radio buttons. However, both these techniques have been shown before, so we opted instead to set the absolute grid position for the `rmiOption` radio button, with the instruction:

```
        constraints.gridx = 1;
```

This tells the `GridBagLayout` to add the specified component in position 1, where cells start at column 0.

If you would like to see how this common frame would look in the server application, you can skip forward to Figure 8-35. We recommend that you return to this section to find out how we have handled passing information from the common frame to the enclosing frame or dialog box.

The Observer Design Pattern

There are many options to be configured, but the client and server applications cannot start unless all the correct options are configured. However, in the client application, our common frame will be used in a dialog box, while in the server application it will form part of the server GUI. So we need some common way for either of these applications to create an instance of the common frame and receive notifications whenever a field changes. In effect, we want the dialog box and the server application to observe any changes in the common frame.

This is one occasion where we can use the Observer design pattern. In this pattern, you set up one class as the `Observable` class, and any classes that want to be notified of changes implement the `Observer` interface. The `Observer` classes then register themselves to the `Observable` class, and the `Observable` class notifies registered `Observer` classes of any changes.

■ **Note** The Observer design pattern is not normally used for one part of a GUI to receive notifications of changes to another part of the GUI. It is more common to have an `Observable` model in an MVC pattern with one or more `Observer` views. Or you might have an `Observable` server that notifies all the `Observer` clients whenever something changes on the server.

Normally you would have the class you wish to observe extend the `Observable` class, but our `ConfigOptions` panel already extends `JPanel`, so we cannot do this. Instead, we have created an inner class that extends `Observable`, and provided a convenient `getObservable` method that the client and server applications can use in order to register themselves as observers.

When using Java's inbuilt implementations of the observer pattern, you can specify an object that should be passed to the observers. We have created a value object class that can be used to pass the field that was changed along with the field contents. This class is presented in Listing 8-29. For more information on the Value Object design pattern, refer to Chapter 5.

Listing 8-29. *The OptionUpdate Value Object*

```
package sampleproject.gui;

public class OptionUpdate {
    public enum Updates {
        NETWORK_CHOICE_MADE,
        DB_LOCATION_CHANGED,
        PORT_CHANGED;
    }

    private Updates updateType = null;
    private Object payload = null;

    public OptionUpdate(Updates updateType, Object payload) {
        this.updateType = updateType;
        this.payload = payload;
    }

    public Updates getUpdateType() {
        return this.updateType;
    }

    public Object getPayload() {
        return payload;
    }
}
```

When the user changes something in the common dialog box (which is the class that might have some `Observers`), one of the event handlers will be called. If this is an event we

want to notify our Observer classes of, an OptionUpdate value update is constructed and sent to all the observers.

For example, when the user leaves the locationField or the portNumber field, then the following code is executed:

```
    public void focusLost(FocusEvent e) {
        if (DB_LOCATION_LABEL.equals(e.getComponent().getName())
                && ( ! locationField.getText().equals(location))) {
            location = locationField.getText();
            updateObservers(OptionUpdate.Updates.DB_LOCATION_CHANGED,
                            location.trim());
        }

        if (SERVER_PORT_LABEL.equals(e.getComponent().getName())
                && ( ! portNumber.getText().equals(port))) {
            port = portNumber.getText();
            updateObservers(OptionUpdate.Updates.PORT_CHANGED, port.trim());
        }
    }
```

Assuming that the user has changed one of these fields, the following method is called:

```
private void updateObservers(OptionUpdate.Updates updateType, Object payLoad) {
    OptionUpdate update = new OptionUpdate(updateType, payLoad);
    observerConfigOptions.setChanged();
    observerConfigOptions.notifyObservers(update);
}
```

■ **Note** We called the setChanged method on the Observable object before calling the notifyObservers method. If you call notifyObservers without first calling setChanged, no Observers will be notified.

That is all we need to do to notify however many observers we may have. While it may appear that we have gone to a lot of work to pass some information back to either the Client Database Location dialog box or to the server GUI, the advantage is that we have decoupled the panel from the users of the panel—any class can use this panel, and simply by registering themselves as an Observer of the panel they can get notification whenever anything changes on the panel.

The next two sections—"The Client Database Location Dialog Box" and "The Server GUI"—show the other half of the Observer-Observable pair. Both will be Observers of the panel described here.

The Client Database Location Dialog Box

Having created a common frame for the various modes, we can add it to a dialog box, as shown in Listing 8-30.

Our dialog box will not allow users to start the client application until they have entered the required details; however, the required details change depending on the client application mode—in stand-alone mode, we only need to know the URI of the data file. But in networked mode we need to know the URL of the server, its port number, and the type of server we are connecting to. The constructor starts by assuming that if we are building a dialog box for stand-alone mode, then we don't need a port number or connection type. It then goes on to create an instance of the `ConfigOptions` pane and add itself as an observer of the pane.

A simple OK/Cancel question `JOptionPane` is then created, with the common pane as its main object. We then override the buttons that will be displayed so that we can enable the connect button when the user has entered the required data.

If the user closes the dialog box rather than clicking the Connect or Exit button, we want to treat it as though they had clicked the Exit button. So we set the dialog box to do nothing on close, and then we add a window listener to the dialog box.

Listing 8-30. *The DatabaseLocationDialog*

```
public DatabaseLocationDialog (Frame parent, int connectionMode) {
    // the port and connection type are irrelevant in standalone mode
    if (connectionMode == ApplicationMode.STANDALONE_CLIENT)
        validPort = true;
        validCnx = true;
    }

    configOptions = (new ConfigOptions(connectionMode));
    configOptions.getObservable().addObserver(this);

    options = new JOptionPane(configOptions,
                              JOptionPane.QUESTION_MESSAGE,
                              JOptionPane.OK_CANCEL_OPTION);

    connectButton.setActionCommand(CONNECT);
    connectButton.addActionListener(this);
    connectButton.setEnabled(false);

    exitButton.setActionCommand(EXIT);
    exitButton.addActionListener(this);

    options.setOptions(new Object[] {connectButton, exitButton});

    dialog = options.createDialog(parent, TITLE);
    dialog.setDefaultCloseOperation(JDialog.DO_NOTHING_ON_CLOSE);
    dialog.addWindowListener(this);
    dialog.setVisible(true);
}
```

When the user makes a change in the common frame, all the observers are notified. The notification is sent to the method with the signature public void `update(Observable o, Object arg)`. In that method, we first confirm that we did receive an instance of the `OptionUpdate` value object, and if not we ignore the entire message. If it is an `OptionUpdate`, then we recast it so that we can easily get access to information about what has changed, as shown in the following code snippet:

```
if (! (arg instanceof OptionUpdate)) {
    log.log(Level.WARNING,
            "DatabaseLocationDialog received update type: " + arg,
            new IllegalArgumentException());
    return;
}

OptionUpdate optionUpdate = (OptionUpdate) arg;
```

■ **Caution** You should always check for nulls or for the type of an object before recasting it. However, if you check the type of the object, you may not need to explicitly check for null since `instanceof` will return false if passed a null reference. You should also check for nulls being passed into any API you have made public. Even though we know that the class we are currently observing should only send us instances of the `OptionUpdate` class, we cannot guarantee that this will never change in the future. Should this change, we will get a warning log message giving as much information on what has been received and where it came from as possible.

■ **Tip** Even though we created an `IllegalArgumentException` for the purposes of creating a stack trace in the log message, we never threw it, so the application will continue to run. Creating an exception simply for the information available from the exception can be a useful tool if you ever need to debug your code.

■ **Note** It may not be desirable to log all updates received that your particular code is not interested in. When AWT was first released, **all** events would be sent to any class that was interested in any event. This could mean that a class that was only interested in learning when the mouse moved over a particular field might get millions of updates as the mouse moved over other areas of the screen—in such a case you would not want to log all the unwanted events. However, in this particular case where we know all the events that can be generated at this time, it makes sense to log a warning if we receive an event we were not expecting.

We then perform validation on the field the user has entered, setting the Boolean flag if the user has entered valid data, as shown in the following code snippet for validating a field name:

```
location = (String) optionUpdate.getPayload();
if (configOptions.getApplicationMode()
        == ApplicationMode.STANDALONE_CLIENT) {
    File f = new File(location);
    if (f.exists() && f.canRead() && f.canWrite()) {
        validDb = true;
        log.info("File chosen " + location);
    } else {
        log.warning("Invalid file " + location);
    }
}
```

Finally we check whether we have all the required fields, and if so, we enable the Connect button as shown in the following code snippet:

```
boolean allValid = validDb && validPort && validCnx;
connectButton.setEnabled(allValid);
```

The Server GUI

The server GUI is shown in Figure 8-35. As can be seen, the majority of this GUI is the common panel developed earlier.

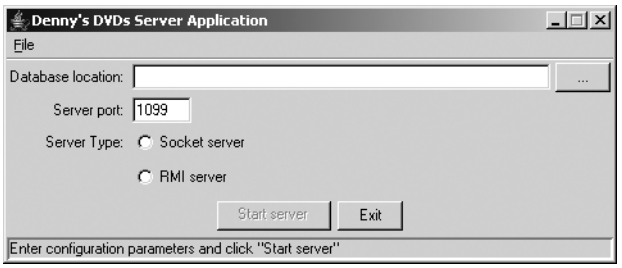


Figure 8-35. *The server GUI*

The constructor used to create the server GUI is shown in Listing 8-31. We start by setting the title of the GUI, configuring the application to exit if the close button is clicked, and ensuring the GUI cannot be resized.

We then create our menu bar the same way we did for the client application, and add our `ConfigOptions` panel to the main window. We then add the buttons to start the server (disabled until configuration options are set), and load any stored configuration options. Finally, we center the server window on the screen, and set it to be visible.

Listing 8-31. *The ServerWindow Constructor*

```
public ServerWindow() {
    super("Denny's DVDs Server Application");
    this.setDefaultCloseOperation(this.EXIT_ON_CLOSE);
    this.setResizable(false);
    Runtime.getRuntime().addShutdownHook(new CleanExit());

    // Add the menu bar
    JMenuBar menuBar = new JMenuBar ();
    JMenu fileMenu = new JMenu ("File");
    JMenuItem quitMenuItem = new JMenuItem ("Quit");
    quitMenuItem.addActionListener(new ActionListener() {
        public void actionPerformed (ActionEvent ae) {
            System.exit(0);
        }
    });
    quitMenuItem.setMnemonic(KeyEvent.VK_Q);
    fileMenu.add(quitMenuItem);

    fileMenu.setMnemonic(KeyEvent.VK_F);
    menuBar.add(fileMenu);

    this.setJMenuBar(menuBar);

    configOptionsPanel.getObservable().addObserver(this);
    this.add(configOptionsPanel, BorderLayout.NORTH);
    this.add(commandOptionsPanel(), BorderLayout.CENTER);

    status.setBorder(BorderFactory.createBevelBorder(BevelBorder.LOWERED));
    JPanel statusPanel = new JPanel(new BorderLayout());
    statusPanel.add(status, BorderLayout.CENTER);
    this.add(statusPanel, BorderLayout.SOUTH);

    // load saved configuration
    SavedConfiguration config = SavedConfiguration.getSavedConfiguration();

    // there may not be a default database location, so we had better
    // validate before using the returned value.
    String databaseLocation =
            config.getParameter(SavedConfiguration.DATABASE_LOCATION);
    configOptionsPanel.setLocationFieldText(
            (databaseLocation == null) ? "" : databaseLocation);

    // there is always at least a default port number, so we don't have to
    // validate this.
    configOptionsPanel.setPortNumberText(
            config.getParameter(SavedConfiguration.SERVER_PORT));
```

```
        status.setText(INITIAL_STATUS);

        this.pack();
        // Center on screen
        Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
        int x = (int) ((d.getWidth() - this.getWidth())/ 2);
        int y = (int) ((d.getHeight() - this.getHeight())/ 2);
        this.setLocation(x, y);        this.setVisible(true);
    }
```

When the user has entered all the valid information and clicked the Start Server button, all the user fields and buttons with the exception of the Exit button are disabled, as shown in Figure 8-36.

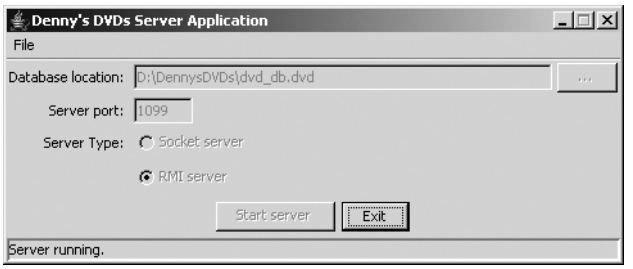


Figure 8-36. *The running server GUI*

This is achieved with the following code in the action listener for the Start Server button:

```
configOptionsPanel.setLocationFieldEnabled(false);
configOptionsPanel.setPortNumberEnabled(false);
configOptionsPanel.setBrowseButtonEnabled(false);
configOptionsPanel.setSocketOptionEnabled(false);
configOptionsPanel.setRmiOptionEnabled(false);

startServerButton.setEnabled(false);
```

We then add a shutdown hook to handle any exit events:

```
Runtime.getRuntime().addShutdownHook(new CleanExit());
```

Finally we start the appropriate server depending on what type of server the user has chosen.

The shutdown hook code is very simple—it is simply an initialized thread that gets called when the application is shutting down. It locks the database so that no other thread can attempt to write to the file while we are shutting down, and then exits. The complete code for the shutdown hook is shown in Listing 8-32.

■ **Tip** Adding a shutdown hook is a good way of handling shutdowns within an application that will be running on a server. No matter whether your code calls `System.exit`, or the user clicks the close button, or the application is told by the operating system that it must shut down, the same hook will be run. Handling the cases where the operating system tells the application to shut down is especially valuable—consider the case where the server is running on an uninterruptible power supply (UPS) and the power goes out. Normally the UPS runs for a while on battery power, but that can only last so long. So before the battery is likely to run out, the UPS sends a message to the operating system telling it to shut down. The operating system then sends a message to all running applications telling them to shut down. The shutdown hook allows your application to receive this message and perform a clean shutdown.

Listing 8-32. *The Shutdown Hook Code*

```
package sampleproject.gui;

import java.io.IOException;
import java.util.logging.*;
import sampleproject.db.*;

public class CleanExit extends Thread {
    private Logger log = Logger.getLogger("sampleproject.gui");

    public void run() {
        log.info("Ensuring a clean shutdown");
        try {
            DVDDatabase database = new DVDDatabase();
            database.setDatabaseLocked(true);
        } catch (IOException fne) {
            log.log(Level.SEVERE, "Failed to lock database before exiting", fne);
        }
    }
}
```

Swing Changes in J2SE 5

Sun has introduced several changes to Swing for the latest release of J2SE. Several of these improvements can be used to provide a more polished submission. We will briefly discuss them in this section.

Improve Default Look and Feel of Swing

Prior to JDK 5, if you wanted to provide a common look and feel for your application on multiple platforms, you had to use the Metal look and feel, which by default used to look like Figure 8-37.

Figure 8-37. *An example of using the Metal look and feel with the Steel theme*

With JDK 5, Sun has modified this look and feel, as shown in Figure 8-38.

Figure 8-38. *An example of using the Metal look and feel with the Ocean theme*

To help reduce confusion, Sun has named these two versions of the same look and feel "Steel" and "Ocean," respectively.

If you wish to use the old theme, you can set the following system property:

```
-Dswing.metalTheme=steel
```

Skins Look and Feel

A "skin" (also called a "theme") is a way of changing the look and feel of an application or website, without changing any code. This is usually achieved by modifying a configuration file.

Sun has created `javax.swing.plaf.synth.SynthLookAndFeel` as a skinnable look and feel, allowing the look and feel to be specified in a file. This means that your users could modify this file to make your application meet their preferred look and feel without needing any coding changes.

Adding Components to Swing Containers Has Been Simplified

Prior to JDK 5, it was not possible to directly add components to any class that implemented `RootPaneContainer`, namely `JApplet`, `JDialog`, `JFrame`, `JInternalFrame`, and `JWindow`. Instead, you had to get the content pane, then add the components to it. This resulted in code that looks like this:

```
JFrame theFrame = new JFrame();
theFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

Container thePane = theFrame.getContentPane();

thePane.add(new JButton("Exit"));
theFrame.pack();
theFrame.setVisible(true);
```

A large number of users of classes that implement `JRootPane` do not need to use the various panes that exist—they only need to add content to the content pane. So in JDK 5 Sun has rewritten the `add` methods of these classes so that they perform in the way most users would expect. This allows us to rewrite the previous code as shown here:

```
JFrame theFrame = new JFrame();
theFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

theFrame.add(new JButton("Exit"));
theFrame.pack();
theFrame.setVisible(true);
```

This reduces confusion as to when we should be dealing with the frame itself or the panel, and makes the code a little more readable.

Summary

A sound interface design will bridge the gap between the user and the system. By using solid design patterns, such as the Model-View-Controller (MVC) architectural paradigm, you can ensure that changes to the data display have minimal impact on the rest of the system. On a superficial level, the end user will most likely judge the quality of an application based on the functionally of its interface, so it is important to plan and design a quality front-end to a system.

In this chapter we introduced GUI design, and provided some examples of how you can combine various layout managers and components to achieve your desired design. It is important to realize that there is no "one right way" to develop a GUI, so you can use the techniques introduced here to develop GUIs that you believe will be usable for your instructions.

FAQs

Q The instructions state that I may only use Swing components, but none of the layout managers are part of the `javax.swing` package—will this cause me to fail?

A You will be fine using any of the layout managers. However, you must not use an AWT component where there is a Swing replacement. For example you should not use a `Button` since there is a Swing replacement: the `JButton`.

Q Does Swing replace AWT?

A Swing is not a replacement for AWT. Swing is built on the patterns and groundwork set forth in AWT. Both Swing and AWT components can be mixed and matched in any interface, to an extent (however, only Swing components may be used in the Sun assignment).

Swing does offer some large improvements over AWT both in performance and functionality. Therefore, Sun now emphasizes Swing over AWT. As a result, Swing is a required part of the SCJD exam.

Q How do I center a window on screen?

A To center a window on screen, get the window dimensions from the `java.awt.Toolkit.getDefaultToolkit().getScreenSize()` method. Use the returned `Dimension` to calculate the vertical and horizontal center of the screen while taking into account the dimensions of the window you want to center. Finally, place the window at that location, as in the following code snippet used for centering our `connectionDialog` object:

```
// Center on screen
Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
int x = (int) ((d.getWidth() - connectionDialog.getWidth())/ 2);
int y = (int) ((d.getHeight() - connectionDialog.getHeight())/ 2);
dialog.setLocation(x, y);
```

Q How do I create keystroke mnemonics for my GUI components?

A You can add keystroke mnemonics to virtually any component via the `setMnemonic(int keyValue)` method. Lists of predefined key values are located in the `KeyEvent` class.

Q Can I use the Macintosh look and feel on a Windows operating system, or vice versa?

A Sun provides several cross-platform interface look and feel packages. Metal and Motif were the only guaranteed look-and-feel libraries prior to JDK 1.5. With the release of JDK 1.5 Sun has included Ocean (a custom Metal theme), and Synth (a skinnable look and feel). Most other libraries are operating system–specific. Although it might be technically possible to use the Windows look and feel on a Mac, or vice versa, it is usually not legal to distribute an application that uses one operating system's look and feel on another platform.

Part of the power of Swing is its capability to be extended and make custom look-and-feel libraries. Some custom libraries are available that are cross-platform, but they are not part of the standard Java distributions and are difficult to rely on for interface design.

Q I found a look and feel on the Web that I prefer to any of Sun's look and feels. Can I use it instead?

A Check your instructions carefully—most instructions state that **all** code submitted must be your own. If you use a look and feel that you did not write and that is not part of the JDK, then you will be violating this rule.

Q Where can I find out more about Java look-and-feel guidelines?

A You can find extensive information pertaining to Java interface design and standard practices at `http://java.sun.com/products/jlf/`.

Q Where can I find out more information on interface design and usability testing?

A The following URLs are great places to begin research into usability engineering and testing:

`http://www.useit.com`

`http://www.asktog.com`

PART 3

Wrap-Up

CHAPTER 9

Project Wrap-Up

Congratulations! You've made it through a myriad of complex topics and intricate details. We've covered a lot of information that will help you pass the Sun Certified Java Developer (SCJD) exam, and we've exposed you to some of the new features of JDK 5. This chapter summarizes some of the architectural decisions we confronted during the completion of our sample project, and it wraps up some of the loose ends regarding packaging and running the application. In this chapter we will cover

- Understanding the design decisions
- Finding out where to get the code samples
- Compiling and packaging the application
- Creating a manifest file
- Running the application in local mode
- Running the application in network mode
- Running a multithreaded test client
- Packaging the submission

Figure 9-1 presents an overview of the Denny's DVDs 2.0 application.

Note Normally you would expect to see an overall diagram of an application drawn from left to right, or top-down—in other words, the client would normally be in the far left of the diagram or in the top location. There is a reason we have drawn the diagram bottom-up, though: it matches the way we have developed the project through this book. We started with the `DvdDatabase` class in Chapter 5, proceeded with the RMI and sockets classes in Chapters 6 and 7, and built the GUI classes in Chapter 8.

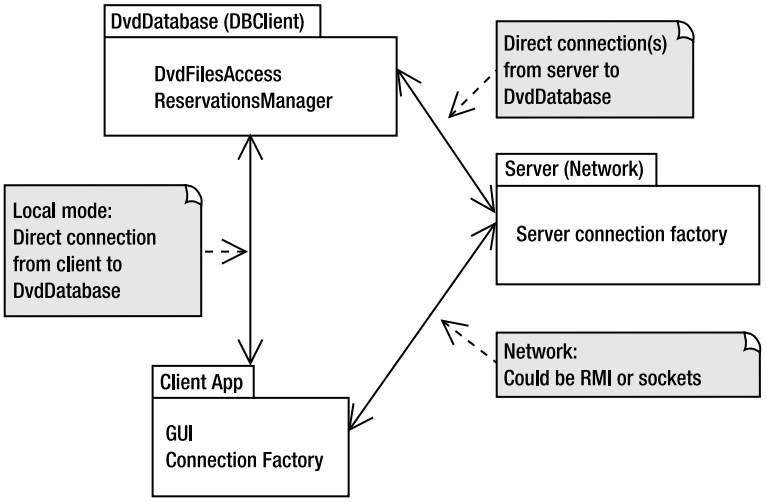


Figure 9-1. *Denny's DVDs 2.0 overview*

The design decisions we made centered around the three tiers of the application, which are central to the SCJD exam. The areas of interest are locking in the `db` package, choosing between RMI and sockets for the network layer, and using the MVC pattern in the GUI package.

We also need to cover how to package, install, run, and test the application.

You can find all of the sample code in the Source Code section of the Apress web site (`http://www.apress.com`). This chapter explains how to compile and package the final version, Denny's DVDs 2.0.

Let's get started!

Thread Safety and Locking

Thread safety was covered in depth in Chapter 4. The main thrust of Chapter 4 was to explain threading and related topics such as synchronization, locking, and concurrency and the new `concurrent` package of JDK 5. Waiting was also explained in detail, as were other issues relating to sharing a single resource across multiple client threads.

Chapter 5 also implemented a locking strategy that demonstrated these concepts. For example, the networking code base creates separate instances of the `DvdDatabase` class for each client so that we can identify the owner of the lock as described in the discussion points.

The Choice Between RMI and Sockets

It can be difficult to choose whether to develop an RMI solution or a serialized object over sockets solution to the Sun SCJD assignment. Sun will accept either solution, with no extra marks awarded or deducted for the choice alone. Many candidates simply choose the technology they are least familiar with as that gives them greatest scope for learning. But if you do not know either technology, or if you are equally comfortable with both technologies, it is beneficial to be able to look at the benefits of each.

■ **Note** When reading these benefits you should be aware that the benefit might be a disadvantage of the other choice. Also, many of the perceived benefits can be easily countered in an argument—where possible we provide counterarguments in our discussions in the same place we discuss the benefit.

Benefits of Using a Serialized Objects Over Sockets Solution

In real-world applications, the choice between RMI and sockets often comes down to the scalability and performance requirements of your application.

Sockets are ideal if performance is a must since you can limit the degree of overhead and sockets are well suited for sending data, often in compressed form that does not require a heavy protocol. As you may recall, the larger and more complex your protocol becomes, the more you may want to use RMI. A well-designed, simple socket interface can outperform an RMI-based server. If you must handle a large number of requests efficiently, sockets may be for you.

This does not mean that you cannot compress or encrypt your data using RMI—if you want to, you can use custom socket factories with RMI to provide specialist functionality (for more information, refer to `http://java.sun.com/j2se/1.5.0/docs/guide/rmi/socketfactory/`). However, when you implement a straightforward sockets solution, you can easily incorporate only the functionality you need, and leave out the functionality you do not need. For example, you might decide to drop the "heartbeat" functionality that is incorporated into every RMI solution. The heartbeat functionality simply ensures that at regular intervals the server sends a signal—a heartbeat—to the RMI registry to let the registry know that the server is still alive; similarly the RMI clients send a signal—a heartbeat—to the server to let it know that they are still alive.

Another advantage to sockets is that most system administrators are already familiar with what needs to be done to implement a server that listens on a particular socket, even if they need to go through a firewall. Few are familiar with setting up an RMI registry, and even fewer know how to configure RMI to work through a firewall. RMI can work through a firewall, but it is not quite as simple as setting up a simple sockets solution through the firewall.

■ **Note** Sockets have been a standard method of performing network connectivity for many years, whereas RMI is a relatively new standard. However, this alone does not justify using sockets over RMI; using serialized objects over the sockets connection almost ensures that only another Java class can connect to your server (only "almost" because the Java specifications make it explicitly clear how a serialized class will appear, which means it is *possible* for somebody to develop a client in a different language—just very difficult). Conversely, using RMI over IIOP (the Internet Inter-Orb Protocol used by CORBA) would allow any CORBA-compliant client to connect to your server (although using RMI over IIOP is not allowed in the current assignments). For more information on IIOP and CORBA, refer to the Wikipedia pages `http://en.wikipedia.org/wiki/IIOP` and `http://en.wikipedia.org/wiki/CORBA`, respectively.

It may be easier to propagate changes to remotely accessible methods over a sockets solution than an RMI solution. Unless you do something major like remove a method that is being used, or change the port it is running on, you can usually change a sockets-based server without the clients even being aware of it—they just won't get the new functionality. In comparison, an RMI solution that has pre-JDK 5 clients or that doesn't allow stubs to be dynamically generated will require recompilation of the client stubs, and redistribution to clients if you do not use dynamic downloading. If you do use dynamic downloading, you may also have associated issues with security and setting the `codebase` option.

A sockets solution should also require less sockets and network traffic between client and server. A sockets solution for this assignment might need only one socket per client, and the network traffic can be minimal depending on how minimal you make your solution. An RMI solution, on the other hand, will always open a listening port for your RMI registry, a listening port for your RMI server, a connected port between the server and the registry, and one socket per client between the clients and the server. The RMI solution will also require more network traffic while connecting (doing the lookup on the registry) and also while idling (performing distributed garbage collection and sending heartbeat messages).

This brings up another point: with RMI you always have an extra process running (the RMI server). While most computers are not going to be taxed by this extra process, if you were using most of the CPU and memory resources of your server, this could become an issue.

A *well-written* sockets solution *can* be easier for a junior programmer to understand and maintain than an RMI solution. This contradicts the commonly accepted view that RMI is simpler. But there are several items to consider here. If you write your sockets code well, the networking code will be hidden from the client and server applications—they will just be calling your sockets API, in the same way that in an RMI solution the clients and servers are just calling the RMI API. But in this case, the junior programmer does not need to learn about registries, or how to use `rmic`, or how to deal with the RMI stubs. So there is less of a learning curve for them. However, there is little doubt that once that learning curve has been passed, an RMI solution is easier to develop, understand, and maintain. And most likely, the only way a sockets solution can be clearer than an RMI solution is if the sockets solution is really well written (and possibly also if the RMI solution is not well written).

General client identification is easier in a sockets solution—you can use the connection's thread to identify the client. However, if your assignment requires you to use cookies for locking, unlocking, and modification calls, then you can use the cookie as the client identification. Likewise, if you decide to use thread pooling for scaling reasons, then you will no longer be able to use the thread identifier. Regardless, for either RMI or sockets, using a connection factory will provide a simple client identifier.

There is less chance that somebody will accidentally disable your server. With RMI, if another server happened to do use the `Registry` class's `rebind` method with the same remote reference name, they will replace your code. With sockets, once you have bound to a particular port, no other server can bind to that port. However, this can also work to your disadvantage—if somebody deployed another server that happened to use the port number, then restarted the computer, it may not be possible to predict which server will bind to the port first, so you may not know which server will be accepting connections. This is called a race condition; a race condition in relation to threads is discussed in Chapter 4.

Benefits of Using an RMI Solution

However, let's be blunt. **For the SCJD exam, scalability and performance are not design considerations**, and as shown in the "Benefits of Using a Serialized Objects Over Sockets Solution" section earlier, neither solution provides a method of allowing non-Java clients to connect.

We mentioned that a *well-written* sockets solution *might* be easier for a junior programmer to understand and maintain. However, as a developer you would be expected to know both of these technologies anyway so that you can make a valid choice between them (and to help you, we have devoted Chapter 6 to RMI and Chapter 7 to sockets). Once you know the two technologies, you may find that your job could actually be a lot less tedious using RMI. And the assessor is expected to understand RMI, so there is no problem with submitting an RMI solution. Let's quickly list a few of the advantages of RMI.

Implementing socket servers and socket clients involves creating a custom protocol. A sockets implementation must send serialized objects across the network that the receiving socket point must know how to handle at runtime. Thus, a sockets-based application can be a little more tedious and awkward to write than an RMI program. This can be negated slightly if you write well-factored code, such that each class has only one responsibility, and the methods are likewise well factored. However, using the Command pattern for the client-server communication protocol trivializes building the new protocol.

The details of object serialization and network communication are hidden by RMI, whereas using sockets you have to implement it all yourself. In other words you will be reinventing a basic technology that already exists.

No matter how well you write your code, there is no doubt that you will write more code for a socket-based solution than an RMI one. And the more code you write, the more chance of making mistakes—it can be safer to leverage off the code written by the RMI developers, which has been tried and tested for many years; much of the RMI code base has been around since JDK 1.1, which was released in September 1997.

RMI provides network transparency, which means that to the client a remote object seems to behave as if it is a local object. Thus there is no need to implement a handshake protocol or worry about low-level details such as opening and closing socket connections.

Due to the use of interfaces and remote methods behaving as though they were local, developing code to call a remote method is usually type safe. The same can apply to a sockets solution, but unlike RMI there is no requirement to use interfaces. Consequently, it is easy to end up with a solution that does not provide any form of compile-time (or even runtime) type safety.

RMI also relieves you of the responsibility of having to write multithreaded servers, which can be tricky. You are still required to write thread-safe code in either protocol, but the actual server does not have to spawn threads or manage thread pools.

Thread pooling can provide a significant performance improvement when systems are scaled to large numbers of simultaneous users, and RMI provides it for free. Yet this comes at a cost: client identification is slightly more complex if you cannot use cookies.

RMI is also extensively used in Enterprise JavaBean (EJB) technology, so learning and using RMI for the SCJD assignment will assist you in EJB projects.

The RMI registry helps you deploy your server side code dynamically—using the `Registry` class's `rebind` method allows you to load the new server functionality with a minimum of downtime. With a sockets solution you would either have to stop the old server and then start the new server (with a longer downtime than RMI's `rebind`), or use a different socket port (which would require reconfiguring or replacing clients).

Using RMI frees you from the requirements of specifying which port a particular service will be available on. With a sockets solution you must specify which port your server will be listening on, and the clients must make a connection to that port. If you do not make this configurable and some other application is developed in such a way that it uses the same port (and is also not configurable), then one or the other will have to be recompiled. Even if the port number is reconfigurable you would have to reconfigure all the clients to use the new port number, which may not be easy or practical. RMI solutions, by default, use random port numbers for the servers, where the port number is specified by the RMI registry. The clients need only know the port number of the registry, which they then query to determine how to contact the server they are interested in.

Although not permitted within the SCJD assignment, RMI allows you to download executable classes. You could use it, for instance, to download some security algorithm that will encrypt all the data between clients and server—since the code is downloaded dynamically, the programmer for the client side never gets to see how you implemented it (indeed, you could even hide the fact that it is being used at all), thus reducing the chances of someone trying to hack your encryption algorithms if they never get to see them.

Choosing Between the Two Solutions

Probably by now you have seen that there are good arguments for both solutions, and no absolute differentiator between either. We believe that Sun has deliberately done this to see your decision-making logic.

You must decide which of the arguments we've listed (or others that you determine for yourself) make the most sense to you, and which of them you are willing to defend if you are asked to explain your decision.

The MVC Pattern in the GUI

Chapter 8 covered GUI-related design decisions ranging from general information on interface layout principles all the way to the discussion of specific Swing components, such as the `JTable`. Unlike the choice between an RMI and a sockets-based network implementation (discussed in Chapters 6 and 7 as well as earlier in this chapter), Sun literally requires the use of the `JTable` in order to pass the SCJD exam, so the decision to use it is a no-brainer if you want to pass the exam.

The design decision therefore shifts from what GUI widget is appropriate for use in the Denny's DVDs application to how to properly incorporate the `JTable` into a Swing user interface. The solution we suggest in Chapter 8 is the application-level use of the MVC design pattern. As you learned in Chapter 8, Swing components, such as the `JTree` and the `JTable`, implement the MVC pattern internally, but the choice to use the MVC pattern beyond these components is entirely up to the developer. The main benefit gained through the use of an application-wide MVC architecture is increased abstraction between the data display and the actual data implementation. The `JTable` and its own MVC implementation in turn becomes a smaller player in the overall application display layer and works in cooperation with the application's MVC implementation.

Locating the Code Samples

You can find the code samples for this book in the Source Code section of the Apress web site (`http://www.apress.com`). There you will a zip file with the final code base (i.e., Denny's DVDs version 2.0). In this chapter you will be concerned with the version 2.0 zip file.

Unzip the final code base in the location of your choice on your machine. There should be one directory, `sampleproject`, which contains four directories: `db`, `remote`, `sockets`, and `gui`. These directories correspond to four packages:

- `sampleproject.db`
- `sampleproject.remote`
- `sampleproject.sockets`
- `sampleproject.gui`

Unless you've skipped around quite a bit and started out with Chapter 9, you should be very familiar with each of these packages. Once you've unzipped all of the Java source files into something resembling the preceding directory structure, you're ready to compile the application.

Compiling and Packaging the Application

Bring up a command prompt and type `java -version`. Make sure that you are using J2SE 5 or later (see Figure 9-2). If you aren't, please download it and install it from the Sun J2SE download site (`http://java.sun.com/j2se/1.5.0/download.jsp`).

■ **Note** The "b05" shown in the version number in Figure 9-2 indicates that this is build number 05; it does not imply a beta version of the software.

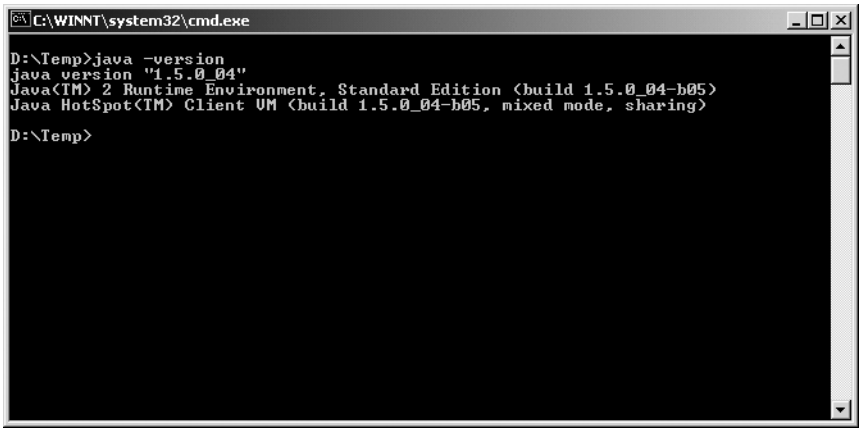


Figure 9-2. *Verify that J2SE 5 is properly installed.*

Navigate to the root directory of Denny's DVDs. This is the directory where you unzipped the project's `.java` files. For demonstration purposes, we created a directory called `dennysDVDs2.0` and chose that as the root directory.

■ **Tip** Zip files can usually be decompressed with the `jar` executable, as shown in Figure 9-3. Both zip files and JAR files use the same compression algorithms.

Next, you'll want to compile the `.java` files into a destination directory. By default, Java will place the `.class` files in the same directory as the source files. Using the Java compiler's `-d` option allows you to separate the source and compiled files. Separating the source and compiled files helps you organize your project.

You need to decide where to direct your `.class` files. For simplicity, we decided to place the compiled files in a directory below the root called `classes`. Using the command `mkdir classes` from the command prompt will create the necessary directory. Or you can add the directory using Windows Explorer.

Next, compile each package separately using `javac` with the `-d` option set to your destination directory. Figure 9-3 illustrates the battery of commands needed to successfully compile the sample project. Make sure that you compile the packages in this order: `db`, `remote`, `sockets`, and finally `gui`. This order highlights the project dependencies. Recall from Chapter 5 that we only had the `db` package. We added the other packages as topics were introduced in Chapters 6, 7, and 8, respectively.

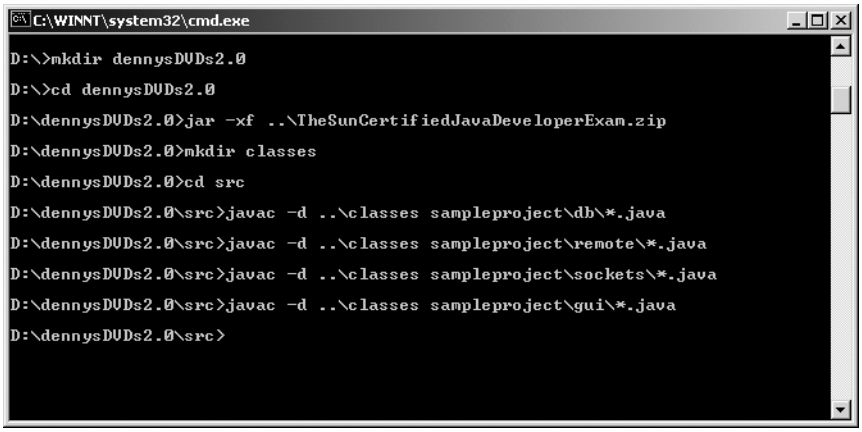


Figure 9-3. *Compiling the source into the classes directory*

Creating a Manifest File

Now you'll want to create a manifest file. This file will be packaged in your project's JAR file and used by the JVM to load the correct class and run the application. The key entry in the manifest file is the class name of the main method you want to execute when running the application. In our case, this is the `ApplicationRunner` class in the `gui` package.

The contents of the manifest file are minimal. Insert the following two lines of code and save them in a new file called `Manifest.mf`. We have placed the manifest file in the root directory, `dennysDVDs2.0`.

```
Manifest-Version: 1.0
Main-Class: sampleproject.gui.ApplicationRunner
```

■ **Note** The name of the manifest file can be anything you like if you are using Sun's `jar` tool to create the JAR file. We will go into this in more detail when we describe using the `jar` tool in the section "Packaging the Application" later in this chapter. If you are not using Sun's `jar` tool (there is really no reason not to use it, though), then the final manifest file **must** be named `MANIFEST.MF` and must be placed in the `META.INF` directory, which must be in the root directory of the JAR file.

The location of the manifest file comes into play when we actually create a JAR file containing our project's class files. All JAR files contain manifests, and the manifest can be used to specify many attributes for the JAR file. Some of the more common attributes are shown in Table 9-1.

Table 9-1. *Manifest Attributes*

Attribute	Use
`Manifest-Version`	Specifies which version of the `Manifest.mf` definition you are conforming to. At present only version 1.0 has been defined.
`Created-By`	Specifies the `jar` tool's creator and version number. This is automatically added by the `jar` tool itself, so you should not set it.
`Class-Path`	You may optionally use this to specify libraries needed at runtime. These must be specified relative to the current JAR file (so you can specify `lib/another.jar` but you cannot specify an absolute path like `d:\libs\another.jar`). If you have multiple JAR files, separate them by spaces.
`Main-Class`	Tells the JVM which class to execute if the JVM is started with the `-jar` parameter.

There are many other attributes that can be set as well, but delving into them is beyond the scope of this book. If you are interested in reviewing these options, we recommend you look at the Sun documentation for JAR files available online at `http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html#JAR Manifest`.

The `jar` tool needs to know where to load the manifest file so that it can be included in the JAR file. Otherwise, the `jar` tool will create one by default. The manifest file that is created by default when you do not specify one will not include a main class label.

Running rmic on the Remote Package

One of the benefits of using JDK 5 is that using `rmic` to create stubs is not strictly necessary; the stubs can be generated dynamically. However, as explained in Chapter 6, stubs are still required if you have pre-JDK 5 clients or if you are not allowed to dynamically generate stubs. This section has been provided for the benefit of those who may require stubs.

■ **Caution** At the time of this writing, all current assignments have a prohibition against *requiring* the dynamic downloading of stubs—you must provide all stubs precompiled in your executable JAR file. Since dynamically generating stubs would result in the stubs being dynamically downloaded, you cannot use the JDK 5 dynamic stub generation feature. However, you should still check the assignment instructions **you** downloaded from Sun; future assignments may remove this prohibition.

Using RMI involves creating stubs with the Java tool `rmic`. The stubs need to be packaged up with the JAR file in order to run the program using RMI via the remote package.

The `rmic` command only needs to be run on the remote object implementation class file. In our case, that class is `DvdDatabaseImpl` of the remote package. Make sure you run the `rmic` command from the destination directory because the class files are required by `rmic`. Figure 9-4 shows `rmic` being run against the `DvdDatabaseImpl` class. In addition, rmic also needs to be run against the `DvdDatabaseFactoryImpl` class.

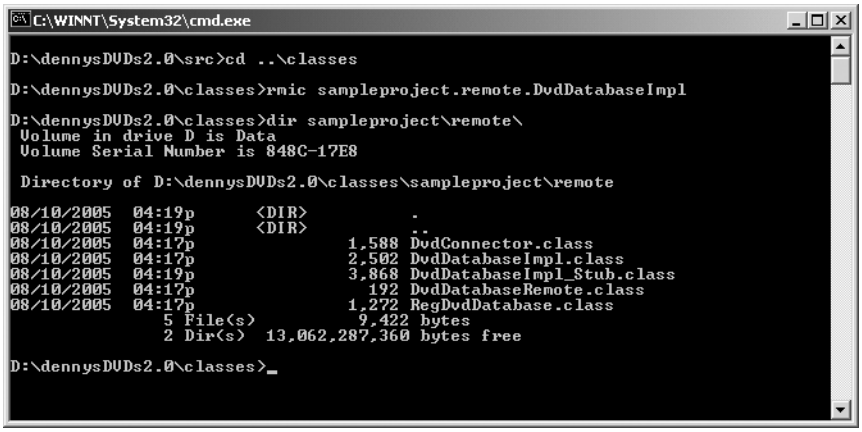


Figure 9-4. *Running rmic on the remote object DvdDatabaseImpl*

If you would like more detailed information than what is shown in Figure 9-4, run `rmic` with the `-verbose` flag. The `-help` option will display all of the other options that are available. The file displayed in the remote directory, `DvdDatabaseImpl_Stub.class`, is the result of the `rmic` command.

Packaging the Application

You are ready to create your JAR file. The `jar` command allows you to create archives of files of various types into a single compressed file based on the zip format. To run the `jar` command, make sure you comply with the following syntax:

```
jar [options] [manifest] destination input-file [input-files]
```

To create your JAR file, use the `c`, `v`, `f`, and `m` options, which are described in Table 9-2. You will also need to specify the location of your manifest file, which is in the `dennysDVDs2.0` root directory.

Table 9-2. *jar Tool Options Used to Create sampleproject.jar*

Option	Description
`c`	**C**reates a new archive; unless the `f` parameter is provided, the archive will be created on standard output.
`v`	Generates **v**erbose output.
`f`	Indicates that the name of the JAR **f**ile to be created (not on stdout) will be specified as the next argument in order.
`m`	Includes a **m**anifest file that will be specified as the next argument in order. The `jar` tool will read the contents of the file you specified and store them in the file named `MANIFEST.MF` in the `META.INF` directory.

■ **Note** When more than one `jar` command-line option requires a parameter, the parameters must appear in the same order as the command-line options. That is, if the command-line options are `-cvfm`, then the output filename must appear before the manifest file name (since the `f` appeared before the `m`). However, if the command-line options provided were `-cvmf`, then the manifest file name must appear before the output filename (since the `m` appeared before the `f`).

There are other options you can use with the `jar` tool. Running `jar -help` will give a brief synopsis of the various arguments and usage options. Figure 9-5 captures the `sampleproject.jar` file creation in the root directory using the command:

```
jar -cfm sampleproject.jar Manifest.mf -C classes .
```

■ **Caution** There is one period in the command line following the directory named classes. It is easy to miss that period in a book, particularly when it occurs at the end of a sentence. If you are unsure of the command line we used, compare it with Figure 9-5.

Using the `-v` option will display verbose output detailing which files are being added to the JAR and information related to file compression. To display verbose information, replace `-cfm` with `-cvfm` in the `jar` command.

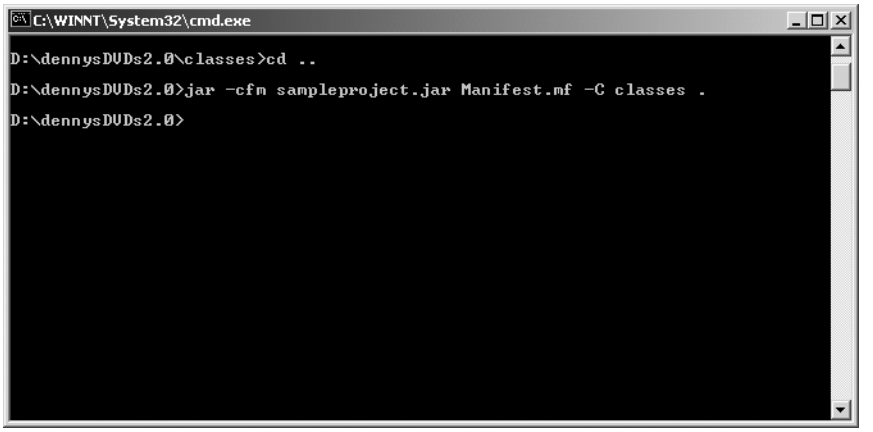


Figure 9-5. *Creating the sampleproject.jar file*

Running the Denny's DVDs Application

The Denny's DVDs application runs in multiple modes, depending on the command-line options provided. These modes are presented in the following sections.

Running the Client Application in Stand-alone Mode

Running in local mode is easy. Navigate to the directory where the `sampleproject.jar` file is located.

The `java` and `javaw` executables allow you to specify a JAR that you want to execute using the `-jar` option. At the command prompt, type the following:

```
javaw -jar sampleproject.jar alone
```

The `sampleproject.jar` file can be anywhere on your machine since the entire project classes are packaged in the JAR. This should bring up the database location dialog box shown in Figure 9-6. Select the location of the database (it should be in the root directory) and click `Connect`. The main window will then load as shown in Figure 9-7.

Figure 9-6. *Starting the application in local mode*

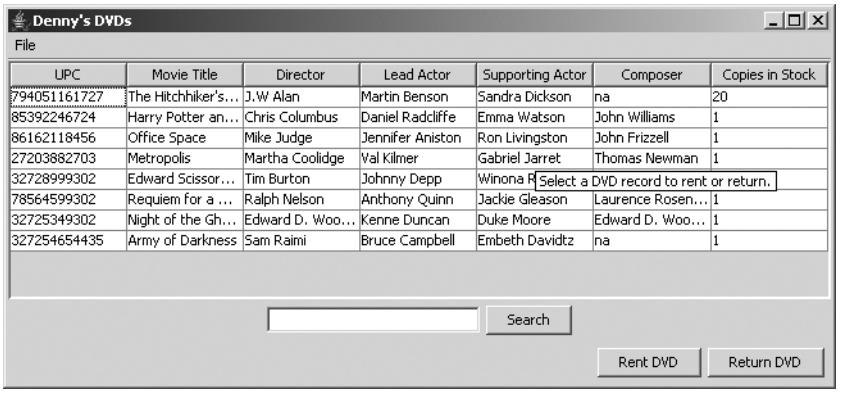


Figure 9-7.*The application running in local mode*

Running Denny's DVDs Server

Running Denny's DVD server requires you to specify the location of the database file, which port you want to use, and the type of server (RMI or sockets). To start the server, at the command prompt, type the following:

```
javaw -jar sampleproject.jar server
```

This should bring up the server configuration window shown in Figure 9-8. Select the location of the database (it should be in the root directory), change the port number if desired, select the network type, and click the Start Server button. The server window will then display information that it is running and will disable most controls, as shown in Figure 9-9.

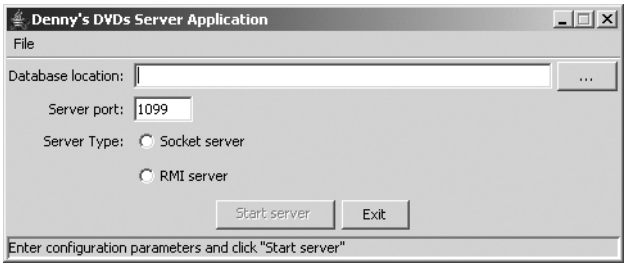


Figure 9-8. *Configuring the network server*

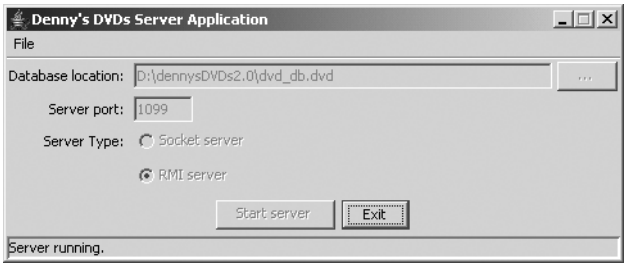


Figure 9-9. *Running the network server*

We specified that the server should use port 1099 in Figure 9-8. With our RMI solution, this means that the RMI registry will be listening on port 1099 (if we had chosen the socket server, it would have meant that our server itself would have been listening on port 1099). We can confirm this by running the `netstat -a` command to show all connected or listening ports, as shown in Figure 9-10.

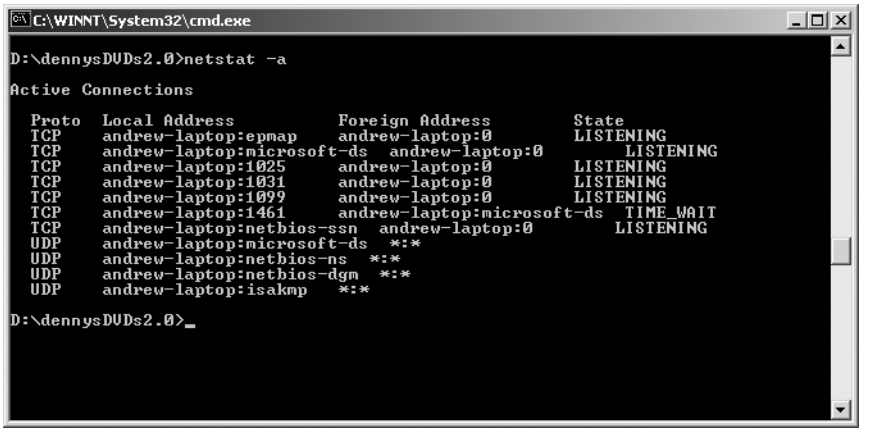


Figure 9-10. *Checking which ports are in use*

In Figure 9-10 we can see that the fifth `Local Address` is listening on port 1099. The port that the server itself is running on is unknown—it will be dynamically configured by the RMI registry.

Running the Client Application in Networked Mode

Running the application in network mode requires specifying the IP address or host name of the machine that the server and database files are located on, the port number the RMI registry or the server is listening on, and the server type. To start the networked client, at the command prompt type the following:

```
javaw -jar sampleproject.jar server
```

This should bring up the connection configuration dialog box shown in Figure 9-11.

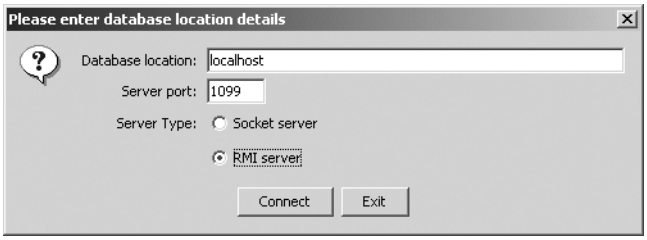


Figure 9-11. *Starting the networked client*

Enter the location of the database (either by the name of the computer hosting the database, e.g., `localhost`, or by the IP address of the computer host, e.g., `127.0.0.1`), change the port number if necessary to match the port number specified when starting the server, and select the same network type as specified when starting the server. Then click `Connect`, and the main window will load as shown in Figure 9-7 earlier.

Testing

In version 2.0 of Denny's DVDs is a package called `test`, which is in a different directory structure than the standard source. The standard source is in the directory named `src`, while the test package is in a subdirectory called `test`. The different directory structures will assist us to ensure that we do not submit any classes we do not want to submit—this will be explained in the "Packaging Your Submission" section later in this chapter.

This may be a little confusing, so we recommend you look at the directory tree structure displayed in Figure 9-12. If you have been following along with this chapter, you will have a root directory similar to `D:\dennysDVDs2.0` (the actual drive and directory can be different). In that root directory so far are the directories `classes` (where `javac` has been storing our compiled classes), `src` (which is the root directory for our sample assignment source), and `test` (which is the root directory for our test source). Our package structure appears within each of those directories, starting with the top-level package named `sampleproject`.

■ **Note** There should also be a bonus directory in the root directory named `src - examples`. This has not previously been described anywhere in the book, but it contains sample code for many of the topics discussed in the book that are not part of the project itself.

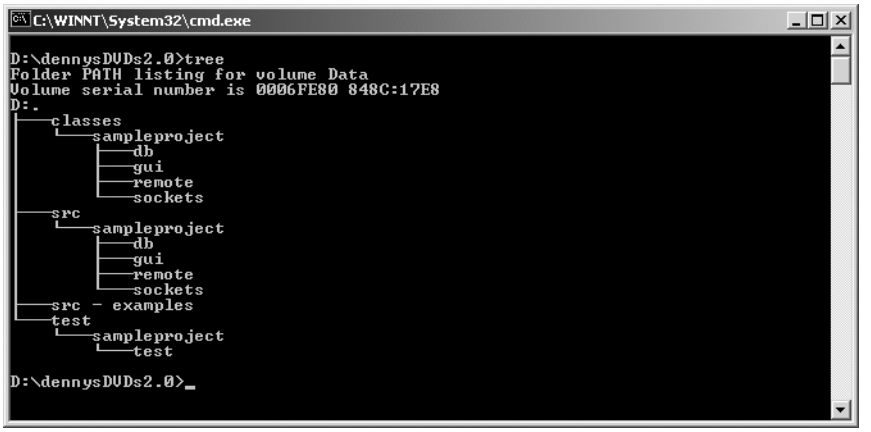


Figure 9-12. *Tree structure for the development project so far*

The test package contains our multithreaded test harness. Testing your application with multiple clients is a must. We will use our test harness to simulate a group of users all vying voraciously for database access.

In the test package are two classes. One called `DBTester`, which extends `Thread` and simulates the client that attempts to rent and return DVDs. The other, called `DBTestRunner`, it spawns multiple `DBTester` test threads, waits for them to finish running, and then reports on their success (or failure).

Since the test package is in a different directory structure from our standard source, we must compile it separately. To do this, we can change into the test directory and run the following command:

```
javac -cp ..\classes -d ..\classes sampleproject\test\*.java
```

Listing 9-1 shows the test client code. Unlike the real application, we have chosen to directly connect to the RMI `DvdConnector` or the sockets `DvdConnector` just to show how this could be done. To simplify the connection code, we have commented out one of the two network connector import statements.

Listing 9-1. *DBTester.java*

```
package sampleproject.test;

import sampleproject.db.*;
import java.util.Date;
```

```
// rather than going through a factory, we are directly calling the connector
// Uncomment the DVDConnector of the protocol you want to use.
import sampleproject.remote.DvdConnector;
//import sampleproject.sockets.DvdConnector;

/**
 * A DBTester is the test equivalent of a client who is trying to book one or
 * more DVDs. However we know exactly how the DBTester is going to behave,
 * therefore we can predict the results of this testing.
 */
public class DBTester extends Thread {
    // various status for what can happen when we try to book over the network
    public enum Status {
        SUCCESS, OUT_OF_STOCK, TIMEOUT
    }

    private String dvdUpc; // the DVD we are supposed to rent
    private int numberOfRentals; // number of times to rent it
    private DBClient db; // connection to the remote database

    private int successfulRentals = 0;  // number of times we rented the DVD
    private int outOfStock = 0; // number of times we failed due to no copies left
    private int timeouts = 0; // number of times timed out trying to reserve DVD

    // To make the screen output easier to read, we are using a pretend logger
    // If we chose to convert to the real JDK logger, we could just change this
    private PretendLogger log = new PretendLogger();

    public DBTester(String title, int numberOfRentals, String dvdUpc)
            throws Exception {
        super (title);
        this.numberOfRentals = numberOfRentals;
        this.dvdUpc = dvdUpc;

        db = DvdConnector.getRemote();
    }
```

Most of the work is performed in the `run` method—the client goes into a loop based on the number of times they are supposed to rent the DVD. Within that loop, they try to rent the DVD. If they are successful, they watch it for two seconds, then return it. If they are not successful because the store is out of stock, they complain to management for a second. If they are not successful because trying to obtain the lock took longer than five seconds, they just take note of the fact. No matter which of those events happened, they will then wait for another two seconds before trying to obtain another lock.

■ **Note** We have used specific times for each of these events, providing us with some degree of certainty that we can duplicate our tests. We cannot be absolutely guaranteed that we can get exactly the same result every time since minor changes in how long network traffic takes over multiple bookings could have an effect. But for a small number of bookings on a local area network with low traffic (or on a single machine), we should be able to predict with confidence what the outcome will be.

```
public void run() {
    int secondsForWatchingDvd = 2;
    int secondsForComplaining = 1;
    int secondsForBrowsingStore = 2;

    try {
        for (int i = 0; i < this.numberOfRentals; i++) {
            switch (rentDvd(dvdUpc)) {
                case RENTAL_SUCCESS:
                    successfulRentals++;
                    // watch the DVD
                    Thread.sleep(secondsForWatchingDvd * 1000);
                    // then return it so somebody else can rent it
                    returnDvd(dvdUpc);
                    break;
                case RENTAL_OUT_OF_STOCK:
                    outOfStock++;
                    // complain that it is not in stock
                    Thread.sleep(secondsForComplaining * 1000);
                    break;
                case RENTAL_TIMEOUT:
                    // just track that we had the problem, and continue
                    timeouts++;
                    break;
            }
            // wander around the DVD store looking at DVDs.
            Thread.sleep(secondsForBrowsingStore * 1000);
        }
    } catch (Exception e) {
        // This should never ever go into production code, but for testing
        // we are simply catching *every* exception and displaying it
        System.err.println("Exception thrown by " + getName());
        e.printStackTrace(System.err);
        System.err.println();
    }
}
```

The `rentDvd` method duplicates the business logic required by our application. It reserves the DVD so that no other client can modify it (assuming, of course, that the other client also

follows the protocol of reserving a DVD before modifying it), retrieves the DVD (to ensure we have the latest copy), checks that there are enough DVDs available (and if so it removes one), and saves the modified DVD back to the database.

```
private int rentDvd(String upc) throws Exception {
    if (db.reserveDVD(upc)) {
        try {
            DVD dvd = db.getDVD(upc);
            int copiesInStock = dvd.getCopy();
            if (copiesInStock > 0) {
                copiesInStock--;
                log.info(getName() +
                         "       -> (Rent)     " +
                         "Copies in stock = " + copiesInStock );
                dvd.setCopy(copiesInStock);
                db.modifyDVD(dvd);
                return RENTAL_SUCCESS;
            } else {
                log.info(getName() + "    OO    (No stock)");
                return RENTAL_OUT_OF_STOCK;
            }
        } finally {
            db.releaseDVD(upc);
        }
    } else {
        log.info(getName() + "    XX    (Timeout)");
        return RENTAL_TIMEOUT;
    }
}
```

Similarly the `returnDvd` method reserves the DVD so no other client can modify it, retrieves the DVD (to ensure we have the latest copy), increases the number of copies, and then saves the modified DVD.

```
private void returnDvd(String upc) throws Exception {
    if (db.reserveDVD(upc)) {
        try {
            DVD dvd = db.getDVD(upc);
            int copiesInStock = dvd.getCopy() + 1;
            dvd.setCopy(copiesInStock);
            log.info(getName() +
                     " <-       (Return)   Copies in stock = " +
                     copiesInStock );
            db.modifyDVD(dvd);
        } finally {
            db.releaseDVD(upc);
        }
    }
}
```

When the client has finished running, the test harness will want to know how many successful and unsuccessful bookings were made. So we have a number of getters to provide that information.

```
    public int getSuccessfulRentals() {
        return successfulRentals;
    }

    public int getOutOfStock() {
        return outOfStock;
    }

    public int getTimeouts() {
        return timeouts;
    }
```

Finally, we want to display information on what is happening while the test is running; however, we do not want to use the standard logger as it can make the resultant screen output hard to decipher. In a larger test environment we would probably consider creating our own log `Formatter`, but this is overkill for this chapter, so we have opted to create a pretend logger instead. This provides a very simple logging facility, and if we later decided to change to using the JDK's logger, we would only need to change the definition of our `log` variable.

```
    private class PretendLogger {
        void info(String logInformation) {
            System.out.format("%tT %s%n", new Date(), logInformation);
        }
    }
}
```

The test harness code is very simple—all it needs to do is to create multiple clients, run them, wait until they have finished, and then display some statistics. The code for this is displayed in Listing 9-2.

Listing 9-2. *DBTestRunner*

```
package sampleproject.test;

import java.util.Calendar;

public class DBTestRunner {
    private int numberOfClients = 4; // how many test clients we will start
    private int rentalsPerClient = 2; // number of rentals each client will make
    private String dvdUpc = "32725349302"; // the DVD they will rent

    private DBTester[] clients = null; // an array of the test clients
```

```
    public static void main(String[] args) throws Exception {
        new DBTestRunner();
    }

    DBTestRunner() throws Exception {
        clients = new DBTester[numberOfClients];

        startClients();
        waitForClientsToDie();
        displayStatistics();
    }

    private void startClients() throws Exception {
       for (int i = 0; i < numberOfClients; i++) {
           String clientName = "Client " + i;
           clients[i] =  new DBTester(clientName, rentalsPerClient, dvdUpc);
           clients[i].start();
       }
   }

    private void waitForClientsToDie() throws Exception {
       // wait for them all to finish
       for (DBTester client : clients) {
           client.join();
       }
   }
```

It is important to realize that even though the client threads are all in `TERMINATED` state by the time the statistics are being generated, the `DBTester` objects still exist, and we can still call the public methods on that object to get the information needed for our statistics.

```
    private void displayStatistics() {
       // display some statistics
       System.out.println();
       formatLine("========", "========", "========", "========", "========");
       formatLine("Client #", "Rented", "No stock", "Timeout", "Total");
       formatLine("--------", "--------", "--------", "--------", "--------");
       for (DBTester client : clients) {
           formatLine(client.getName(),
                      client.getSuccessfulRentals(),
                      client.getOutOfStock(),
                      client.getTimeouts());
       }
       formatLine("========", "========", "========", "========", "========");
    }
```

```
    private void formatLine(String name, int rentals, int noStock, int timeout) {
        formatLine(name,
                   "" + rentals,
                   "" + noStock,
                   "" + timeout,
                   "" + (rentals + noStock + timeout));
    }

    private void formatLine(String name, String rentals, String noStock,
                            String timeout, String total) {
        System.out.format("%tT %8s %8s %8s %8s %8s%n",
                          Calendar.getInstance(),
                          name,
                          rentals,
                          noStock,
                          timeout,
                          total);
    }
}
```

When printing statistics (and in our `PretendLogger`), we use the `PrintStream.format` method that was introduced in JDK 5. Remember that the `System` `out` static variable is an instance of `PrintStream`, so we get to use this new method in defining our output.

The `format` method takes a `String` argument that describes the formatting of the output, and then takes a variable number of arguments (VarArgs in action). This means that no matter how many arguments you provide, one definition of the `format` method can handle them all.

The `String` argument that describes the formatting of the output has many options—far too many to mention here. Table 9-3 describes the options we have used in our test harness. For more information, we recommend you look at the documentation for `PrintStream`'s `format` method, and the `Format` class available online at `http://java.sun.com/j2se/1.5.0/docs/api/index.html`.

Table 9-3. *String Format Options Used in the Test Harness*

Option	What It Does
`%tT`	Declares that the corresponding argument after the format string should be displayed as time (the lowercase `t`), and that the time format should be 24-hour format (the uppercase `T`)
`%8s`	Declares that the corresponding argument after the format string should be displayed right justified in a minimum of 8 characters, and it should be a `String` (the lowercase `s`)
`%n`	Outputs the correct characters to start a new line for your operating system

As shown in Figure 9-13, the DVD database as supplied contains three copies of the movie *Night of the Ghouls*. In our test harness, we have specified that there should be four clients trying to rent this movie—therefore we know that at least one will miss out. Furthermore, we

have declared that each client should try to rent this movie twice, and because of our explicit timings, we know that the client who missed out first time should get the movie the second time, and one of the other clients should miss out.

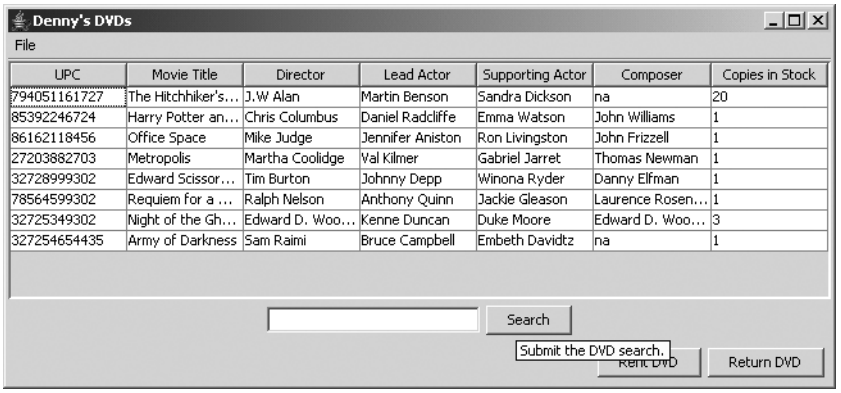


Figure 9-13. *The database contents prior to, and after, running our test*

Figure 9-14 shows a sample run of the test harness. As expected, one of the clients (client 2 in this particular run) missed out on their first attempt to rent the movie, and a different client (client 3) missed out on their second attempt to rent the movie.

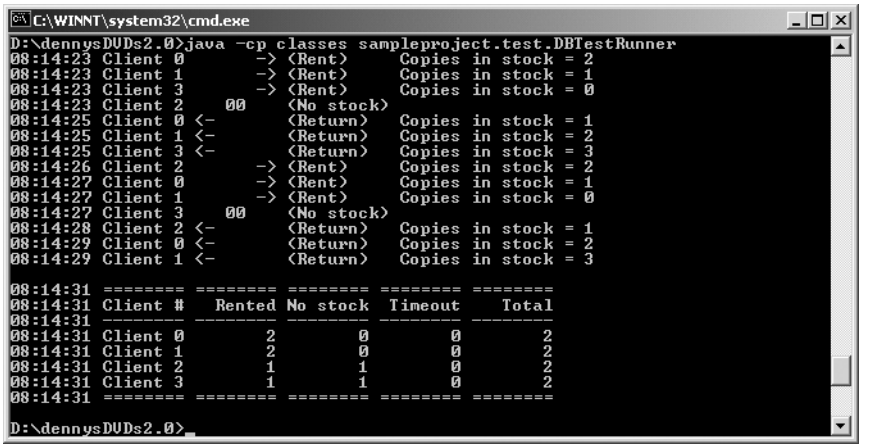


Figure 9-14. *Test harness output*

■ **Caution** Although we have made a reasonable attempt to make a multithreaded networked test harness that will produce repeatable results, you should be aware that the JVM thread scheduler and network latency may result in slightly different results.

Packaging Your Submission

We now have to check our instructions to determine what needs to be packaged up and sent to the assessor. Since we don't have any instructions in our assignment, we are going to follow something that will be *similar* to your instructions—namely we are going to create a JAR file that contains the executable runtime, the database, the source code in a directory called `src`, and the API documentation for our project in a subdirectory of the `docs` directory. What we will end up with is a JAR file *containing* a JAR file, several directories, and our database file, as shown in Figure 9-15.

■ **Caution** The Sun assignment instructions are likely to have a few other requirements as well (such as requiring a design decisions document), and may require slightly different directory structures. You *must* read the instructions you received from Sun carefully, and follow them to the letter. If you get one part of the packaging incorrect, the assessor could fail you on the spot.

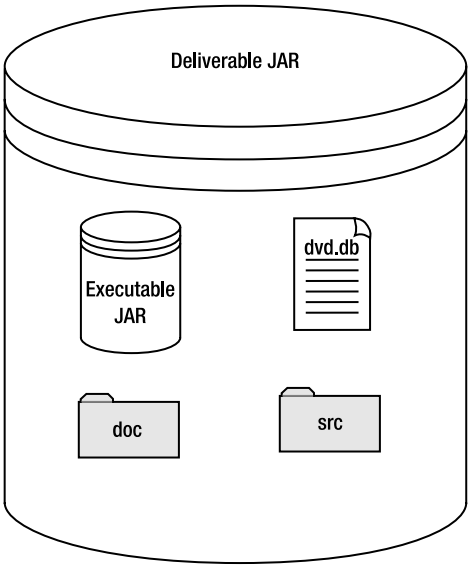


Figure 9-15. *Submission contents*

We have already created our executable JAR file, and since we created it prior to compiling the test classes, we know it does not contain anything we do not want to submit. If we were unsure, or if there's a chance we might go through this sequence more than once, we could just delete the contents of the `classes` directory, then recompile the source in the `src` directory, and we could be certain we have a clean code base. This is where having separate directories for the main project source and the test programs source comes in handy.

Naturally we have all the source code sitting in the src directory, so that can easily be incorporated into our submission. Likewise we already have the database file sitting in the root directory, so the only thing remaining is the Javadoc API documentation.

First up we need to create a directory to hold the documentation. We can do that with the following command:

```
mkdir doc\api
```

The command to build the Javadoc API would be

```
javadoc -quiet  -link http://java.sun.com/j2se/1.5.0/docs/api/
  -d doc\api -sourcepath src -public
  sampleproject.db sampleproject.gui sampleproject.remote sampleproject.sockets
```

■ **Note** Note That Javadoc command line is all one line—we have just spread it over three lines in this book to make it easier to read.

An example of running that command line is shown in Figure 9-16.

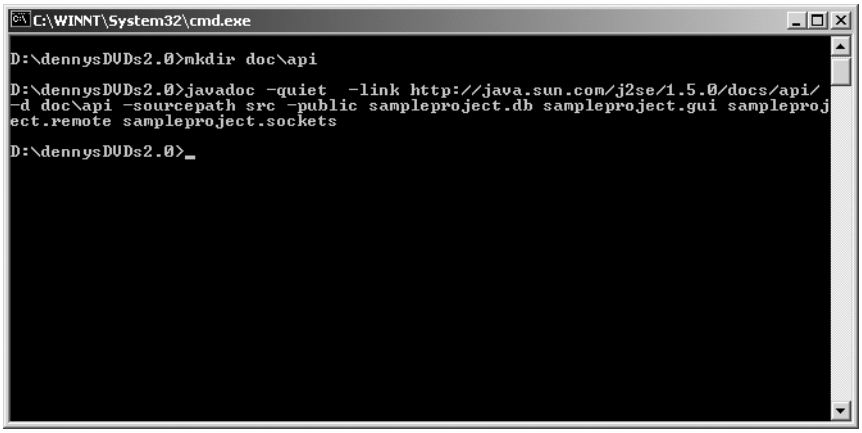


Figure 9-16. *Building the Javadoc API on the command line*

A full explanation of the options we used in our command line, plus many more options we have not used (but you may want to), are listed in the "Running Javadoc from the Command Line" section of Chapter 2.

A simpler option, especially if you are likely to be re-creating the API documentation on a regular basis, is to store all the options in one or more option files. These can be any plain-text file, and each option can be space delimited or newline delimited. An example of using such a file is shown in Figure 9-17.

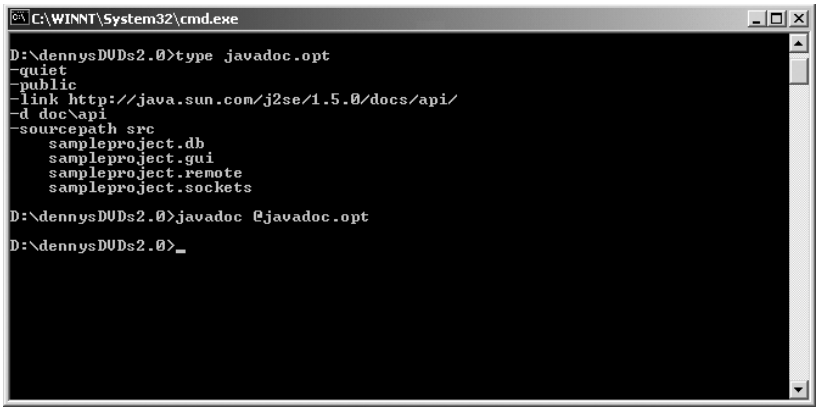


Figure 9-17. *Building the Javadoc API using a file to contain the Javadoc options*

Once we have done this, we can create a single JAR file, holding all the necessary parts. The following command line will perform this for us:

```
jar -cf submission.jar sampleproject.jar dvd_db.dvd src doc
```

■ **Note** We did not specify a manifest for this JAR file, as this JAR is only meant to be a container to get your submission to the assessors—it is not meant to be an executable JAR file. So the default manifest file that the `jar` tool creates for us will suit us perfectly.

An example of building the submission JAR file is shown in Figure 9-18.

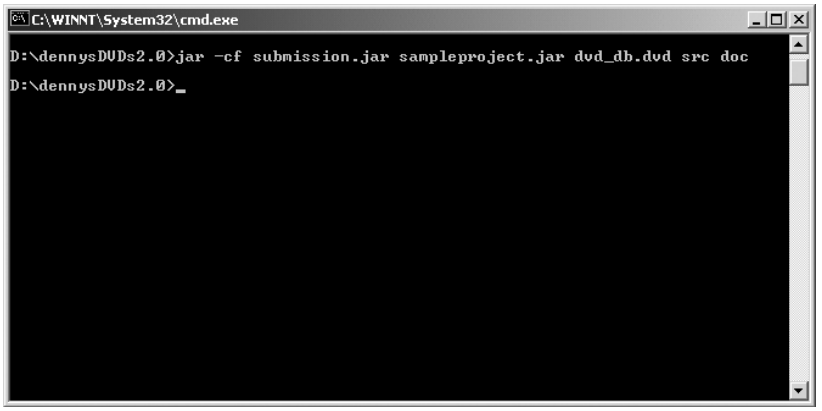


Figure 9-18. *Building the Javadoc API using a file to contain the Javadoc options*

We are now ready to submit. Remember to refer to the information you received from Sun for *your* specific submission instructions. If your instructions differ from the ones in this

At the time of this writing, you upload your Sun submission JAR file to the same site that you downloaded the initial JAR file that contained your Sun instructions and database. When you go to upload the JAR file, you will be given explicit instructions on the naming convention for the submission JAR file (the one that *contains* everything, including the executable JAR file).

■ **Tip** Currently you have to request upload permission before you can upload your submission. Unfortunately, we cannot guarantee that this will always hold true, though, and the only way to find out is to try to submit the assignment—which might not be a good idea if your assignment is not complete (although you could always close your browser window at the point where it asks you to select your submission JAR file). Since you are supposed to submit your assignment before you sit for the exam, you should allow a couple of working days between when you plan to submit the assignment and when you sit the exam, just in case you find you must ask permission first.

Summary

We hope that you have enjoyed this book. More important, we hope that you are now prepared to complete the SCJD exam. You have been exposed to a lot of material, and putting everything that you have learned into context is a large task. Undoubtedly, you have some lingering questions, such as "Have I tested my application properly?" and "How will my application behave in this scenario?"

There always seems to be some change or enhancement that can make your application better, and at some point you need to feel confident that your project will pass Sun's scrutiny. After all, your goal is to pass the exam, not develop a commercial project. Sun estimates that about 100 hours of development time is needed to develop a working solution.

To ensure that you pass your exam, here is a list of to-dos:

- Read the SCJD exam instructions very carefully. Even though many of the tests have the same name, their details differ.
- Read this book. At the very least, look at the sample project to see how we solved some of the basic problems such as locking, networking, and the GUI implementation, especially the `JTable`. We believe that if you understand the code, you should have all of the tools you need to pass your exam.
- Refer to this book throughout your development process.
- Join the JavaRanch SCJD forum (see the FAQs section).
- Test your application. Write a test harness similar to the one we discussed in this chapter. If possible, test it across an actual network. Test all of the use cases. Test your application in a multithreaded environment with a test class such as `DBTestRunner`.
- Package up all of your ReadMe and design documents along with your application and submit the project to Sun as your exam directions stipulate.

FAQs

Q Where do I get the database files to run the sample project?

A It is included in the zip file you can download from the Apress web site (`http://www.apress.com`).

Q Do I have to package my application as a JAR file?

A The answer to this question depends on the specifics of your exam. However, our experience has been that most exams require that the examinees use JAR files for their submission. It is recommended that separate JAR files be included in a master JAR file for submission.

Q What do I need to include with my exam submission?

A The basic elements are typically as follows:

- Source files
- An executable JAR file
- Database file(s)
- A design document explaining some of the key design decisions
- A file that explains the development environment you used
- Javadocs
- User documentation

However, you **must** carefully read the instructions provided to you by Sun to ensure that you meet the submission requirements. If there is a discrepancy between what we have described and what is in the Sun instructions, you must follow the Sun instructions.

Q What platform should I test my submission on?

A Since Java is platform independent, your submission should run on either Unix or Windows. We strongly recommend that you test your exam on Windows, Unix, Linux, and Mac OS X (i.e., Macintosh), since there can be subtle platform differences that could cause problems with your submission. For consistency and simplicity, the examples in this book have been demonstrated on Windows 2000. Even though there is no guarantee what platform your application will be tested on once it is submitted, make sure you document the platform you worked on in your ReadMe.

Q What should I do after I have passed the exam?

A Celebrate. Then send us an e-mail at `scjd@apress.com`.

Q If I have comments or questions regarding this book, whom should I contact?

A You can contact the authors at `scjd@apress.com`.

Q Are there any online resources useful for passing the SCJD exam?

A There are many excellent online resources. Here are just a few that we found particularly helpful:

- The JavaRanch web site at `http://www.javaranch.com`
- The SCJD discussion groups on Yahoo! (`http://groups.yahoo.com`)
- Java 2 Development Kit, Standard Edition Documentation page (`http://java.sun.com/j2se/1.5.0/docs/`)
- The Java Tutorial: RMI (`http://java.sun.com/docs/books/tutorial/rmi/`)
- The JFC Swing Tutorial (`http://java.sun.com/docs/books/tutorial/books/swing/`)
- JavaWorld article: "Sockets programming in Java: A tutorial" at (`http://www.javaworld.com/javaworld/jw-12-1996/jw-12-sockets.html`)
- Portland Pattern Repository (`http://c2.com/ppr/`)
- Java Coding Conventions (`http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html`)
- Sun's Javadoc Style Guidelines (`http://java.sun.com/j2se/javadoc/writingdoccomments/index.html`)
- Java Look and Feel Guidelines (`http://java.sun.com/products/jlf/`)
- User interface design and testing information (`http://www.useit.com` and `http://www.asktog.com`)

Index

A

about this book, 6
Abstract Window Toolkit
  see AWT
abstraction
  RMI implementation, 191
AbstractTableModel class
  TableModel interface, 255
accept method, ServerSocket class, 214, 215, 217
ActionListener interface
  JButton component, 248
  JComboBox component, 251
  JRadioButton component, 249
Activatable class, java.rmi, 177
activation
  lazy activation, 177
active object, 177
addDVD method
  calling persistDVD method, 144
  DBClient interface, 60, 137
  indicating success/failure, 145
addDVDRecord method
  DVDTableModel class, 258
addresses
  getting SO_BINDADDR, 205
  loopback addresses, 206
  socket addresses, 200
addTableModelListener method
  TableModel interface, 255
anonymous inner classes, 248
API documentation
  Javadoc, 35–44
application protocol
  socket servers, 218–221
  TCP sockets, 203
  using sockets, 200
application tiers, 226
ApplicationRunner class
  creating manifest file, 303
  Denny's DVDs, 262–263
  handleException method, 263
architecture
  Denny's DVDs project, 63
ArrayList objects
  object synchronization, 111
assertions
  best practice, 49
  caution: not performing actions for methods, 50
  caution: validating inputs on public methods, 49
  enabling at runtime, 50
assignments
  caution: grading after exam taken, 4
  caution: online instructions, 18
  caution: variation in instructions, 17
  downloading, 5
  SCJD exam, 4
assumptions
  clarifying requirements, 12
atomic operations, 104–106
  logical record locking, 149
  synchronized object definition, 111
authentication
  requirements for, 197
author (-author) option
  Javadoc command line options, 43
author (@author) Javadoc tag, 38
autoboxing
  accessing data, 142
  coding conventions for JDK 5, 32
automatic code reformatters
  caution: avoiding automatic code reformatters, 28
automatic type conversions
  coding conventions for JDK 5, 32
await method, Lock class
  locking in JDK 5, 97
AWT (Abstract Window Toolkit), 64
  Swing and, 237, 261, 291

B

beginning comments
  Java coding conventions, 23
binary operators
  spacing, 27
BindException, 214
bkp subdirectory
  organizing projects, 17
block comments, 28
BLOCKED state, threads, 92
  Thread objects, 107
blocking, 78
  caution: state of threads paused for I/O, 110
  example, 108
  Thread objects, 108–110
  threads refusing to release resources, 108

BorderFactory class
example using, 252
Factory design pattern, 252
Swing, 251–254
BorderLayout manager, 237
bottom (-bottom) option
Javadoc command line options, 43
build scripts
providing in submission, 160
buttons
JButton component, 248
JRadioButton component, 249

C

caching records
data access, 148
running out of memory, 161
using cache in Sun SCJD assignment, 161
CamelCase, 21
case sensitivity
operating system files, 45
casts
spacing, 27
catching exceptions
wrapping exceptions in RuntimeException, 131
wrapping within allowed exception, 130
centering a window, 292
certification process
SCJD exam, 4–5
Sun Certification project requirements, 57–66
character
displaying as mnemonic in label, 244
Checkstyle, 27
Child.java example
multithreading, 79–82
class comments, Javadoc, 37
class declarations
Java coding conventions, 24
class dependencies
working with packages, 47
class diagrams
Denny's DVDs overview, 64
class instance
identifying lock owner using, 151
class naming conventions
Java coding conventions, 21
Class objects
locking class instances, 89–90
class variable comments, Javadoc, 37
Class-Path attribute, 304
ClassCastException, 138
validating data, 143
classes
considering what a class does, 134
internally synchronized classes, 115
Javadoc coding conventions, 36
Javadoc tags, 38
one responsibility per class, 21
providing locking, 148
reading data file, 137
RMI, 175
working with packages, 44, 47
caution: classes with same name, 46
classes subdirectory
organizing projects, 17
ClassLockNotObjectLock class, 96
classpaths
code base and, 195
working with packages, 46
cleanup
system cleanup when client disconnects, 223
client identification
choosing between RMI and sockets, 298
client synchronization, 112–113
clients
class loader requirements, 195
definition in RMI scenarios, 172
handling client crashes, 156–157
locking records, 156
preventing deadlocks, 156
thick (fat) client, 149
thin client, 149
transferring data with server, 120
code
see also source code
caution: avoiding automatic code reformatters, 28
using code samples provided, 67
code base
dynamic loading, RMI, 195
code samples
locating, 301
code tags
Javadoc comments, 37, 38
JDK 1.5, 41
coding conventions
see also Java coding conventions
Java standards, 19–35
Javadoc, 36–44
JDK 5 new features, 29–35
Sun Code Conventions for Java, 20
combo boxes
JComboBox component, 251
Command enum
socket servers, 218–220
command line
running Javadoc from, 42–44
Command pattern
DvdCommand class, 218
commas
spacing, 27
comments
beginning comments, 23

implementation comments, 28, 85
caution: adding, 28
Javadoc, 28, 35–44
source code formatting, 28–29
compilation
Denny's DVDs, 301
components
see also Swing components
adding components to JFrame, 238
caution: lost components, 239
specifying last component, 277
Swing changes in J2SE 5, 290
concurrency
atomic operations, 149
logical record locking, 150
ReadWriteLocks, 142
Condition
LockInformation class, 159
config logging level, 51
ConfigOptions constructor, 279
connectivity
choosing between RMI and sockets, 297
constant naming conventions
Java coding conventions, 22
constants
Sun Coding Conventions, 24
constructors
caution: overloading with versions containing VarArgs, 33
Javadoc tags, 40
containers
JPanel container, 239
layout managers, 237
Controller component
MVC pattern, 235
anonymous inner classes, 249
cookies
identifying lock owner using tokens, 151
CORBA
choosing between RMI and sockets, 297
RMI and, 172
CPU usage
multithreading, 73
CreatedBy attribute, 304
createRegistry method, 180

D

d (-d) option
Javadoc command line options, 43
daemon threads
changing daemon status of threads, 79
example using, 79
IceCreamMan.java example, 83
join statements, 80
data
transferring between client and server, 120
validating, 142
data access
caching records, 148
Data class
building in stages, 160
handling unlisted exceptions, 126
using a value object, 161
using Façade pattern, 161
data entry
JTextField component, 245
validating, 245
validating contents of JTextField, 246
data files
class reading data file, 137
creating for Sun SCJD assignment, 160
showing record persisted to file, 147
synchronized blocks, 144
why include in submission, 66
data presentation
benefits of MVC pattern, 236
JTable component, 227
data tables
JTable component, 254
data types
generic data types, 29
databases
database files, 322
specifying location for Denny's DVDs, 273–286
DatagramPacket class, java.net, 202
DatagramSocket class, java.net, 202
DBClient class
RMI implementation, 185
DBClient interface
coding for, 61
DvdDatabase class, 136
methods, 60
methods indicating success/failure, 145
timeouts on locks, 155
DBSocketRequest class, 217
execCmdObject method, 217
DBTester package
testing Denny's DVDs application, 310–314
DBTestRunner package
testing Denny's DVDs application, 310, 314–316
deadlocks, 98–100
blocked threads, 156
deadlock handling, 156
preventing, 156
race conditions leading to, 100
debugging DVD class
constructor logging, 123
delivery stack, RMI, 173–174
demarshaling
serialization, 164
Denny's DVDs project, 262–289
application overview, 63–66
ApplicationRunner class, 262–263
architecture, 63

building network layer using sockets, 206–212, 215–217, 219
compiling and packaging, 301
creating GUI overview, 64
DBClient.java interface, 60
GridBagLayout manager, 275–279
GridLayout manager, 274–275
GUI design and layout, 263–267
introduction to, 59–63
JAR files, 58
launching the application, 262
MainWindow class, 263, 268–273
network or local database system, 65
networking tier, 163
overview diagram, 295
packaging application, 305
project overview, 57–66
RMI Factory pattern applied to, 178
running application, 306–309
client application in networked mode, 309
client application in stand-alone mode, 307
Denny's DVD server, 307
server GUI, 286
socket servers, 212, 213
specifying database location for, 273–286
testing application, 309–317
deprecated (@deprecated) Javadoc tag, 38, 39, 40
design
documenting, 15
prototyping GUI, 13
design decisions document, 15, 18–19, 296
best practice, 47
considering other options, 146
documentation required, 17
JTable component, 300
packaging submission, 318, 319
design patterns
implementing projects, 14
Observer design pattern, 281–283
Proxy pattern, 193
purchasing/downloading pattern resources, 6
destroy method, Thread class, 106
development methodologies, 12
directories
working with packages, 45
directory structure
organizing projects, 16
using custom directory structure, 55
distributed object system
definition in RMI scenarios, 172
illustration of, 173
doc subdirectory
organizing projects, 17
DocCheck, Javadoc plug in, 36
docRoot (@docRoot) Javadoc tag, 39, 40
docs directory
Sun Certification project requirements, 58
doctitle (-doctitle) option
Javadoc command line options, 43
documentation
best practice, 47
documenting design decisions, 15
high level project documentation required, 17–19
Javadoc, 35–44
local documentation, 44
SCJD exam, 5
DOSClient class, 64
downloads
assignment, 5
DVD class, 119–125
comparing instances for equality, 124
constructors, 122
getters and setters, 123
logging, 123
transferring data between client and server, 120
using serialver tool on, 165, 166
DvdCommand class, 218–220
Command pattern, 218
methods, 219
using enum constants, 220
DVDConnector class
RMI implementation, 180
DvdDatabase class, 134–137
after request completed, 220
caching records, 148
constructors, 135
creating classes required for, 119–133
DVD class, 119–125
DBClient interface, 136
DvdFileAccess class, 137–148
Façade pattern, 134
Factory pattern producing, 177, 178
handling exceptions, 136
referencing other classes, 135
ReservationsManager class, 148–160
RMI implementation, 179, 182
DvdDatabaseFactory class
RMI implementation, 179, 180
DvdDatabaseImpl class
RMI implementation, 181, 188, 191
running rmic on remote package, 304
DvdDatabaseRemote class
RMI implementation, 183
DvdFileAccess class, 137–148
getDVDs method, 141
RandomAccessFile, 138, 139
recordNumbers collection, 138
restricting classes calling, 138
RMI implementation, 180
singleton classes, 137
using serialver tool on, 165, 166
DVDMainWindow constructor, 268
DVDResult class, 220–221

DVDScreen class, 270
DVDSocketClient class, 205–212
  using DVDCommand object, 220
DVDSocketServer class, 215
DVDTableModel class
  AbstractTableModel class, 255
  addDVDRecord method, 258
  additional methods, 258
  using TableModel with JTable, 259
dynamic loading, RMI, 195

E

emptyRecordString
  DvdFileAccess class, 139
enum constants, 220
  SocketCommand enum, 221
environment variables, 8
EOFException
  wrapping within allowed exception, 131
equals method
  overriding, 125
event dispatcher thread
  multithreading with Swing, 114
event handling
  anonymous inner classes, 249
exams
  see SCJD exam
  see also submissions
exception (@exception) Javadoc tag, 40
exceptions
  DvdDatabase class handling, 136
  handling unlisted exceptions, 126–133
  including more exceptions in interface, 66
  logging an exception, 128
  NotSerializableException, 194
  RemoteException, 176
  RuntimeException, 131, 132
  swallowing an exception, 128
  wrapping within allowed exception, 130, 131, 132
execCmdObject method
  DBSocketRequest class, 217
expressions
  spacing, 27
Externalizable interface, 169
  comparing Serializable interface, 171
  customizing serialization with, 169–171
  use of transient in class implementing, 171

F

Façade pattern, 134
  Data class using, 161
factories
  RMI factory, 177
Factory design pattern, 178
  BorderFactory class, 252
  RMI factory, 177
  RMI implementation, 179
  uniquely identifying clients, 153
fat client, 149
fields
  Javadoc tags, 39
file layout
  Java coding conventions, 22–24
FileHandler class, 52
FileNotFoundException
  DvdDatabase class constructor, 136
files
  creating manifest file, 303
  database files, 322
  including extra files in submissions, 18
final keyword
  making methods final, 140
finally block
  releasing locks in, 142, 145
find method, 147
  DBClient interface, 137
findDVD method
  DBClient interface, 60
fine logging level, 51
finer/finest logging levels, 51
firewalls
  choosing between RMI and sockets, 195, 297
FlowLayout manager, 239
  buttons aligned right, 240
  illustrations of, 240
for loop, enhanced
  coding conventions for JDK 5, 31–32
for statements
  source code indentation, 26
  spacing, 27
format method
  testing Denny's DVDs application, 316
formatters
  MaskFormatter, 246
  outputting log messages, 54
Frame containers
  layout managers, 237

G

generic types
  coding conventions for JDK 5, 29–31
getColumnCount method
  TableModel interface, 256
getColumnName method
  TableModel interface, 257
getComposer getter
  DVD class constructor, 123
getDVD method, 143
  DBClient interface, 60, 137
getDvdList method, 141
getDVDs method
  DBClient interface, 60, 137
  DvdFileAccess class, 141

naming conventions, 141
overriding methods, 140
getLocalAddress method, Socket class, 205
getLock method
handling unlisted exceptions, 127
logging an exception, 129
getReceiveBufferSize method, Socket class, 205
getRowCount method
TableModel interface, 257
getSelectedIndex method, 251
getSelectedItem method, 251
getSendBufferSize method, Socket class, 205
getSoTimeout method, Socket class, 204
getStackTrace method, Thread class, 116
getState method, Thread class, 116
getters
automatic generation, 124
DVD class constructor, 123
naming getters, 123
getValueAt method
TableModel interface, 257
Graphical User Interface
see GUI
GridBagLayout manager
components spanning multiple cells, 278
ConfigOptions constructor, 279
Denny's DVDs, 275–279
specifying last component, 277
GridLayout manager
Denny's DVDs, 274–275
GTKLookAndFeel
illustration of, 243
LookAndFeel subclasses, 242
platforms for, 242
GUI (Graphical User Interface)
application tiers, 226
arranging workflow items in, 231
BorderLayout layout manager, 237
client GUI, 263
Denny's DVDs design and layout, 263–267
Denny's DVDs overview, 64
grouping functionality, 231
how users view information in, 231
human interface concepts, 233
keystroke mnemonics, 292
keystroke shortcuts, 230
layout concepts, 227–228
MVC pattern, 234–237
notifications of changes in, 282
online resources, 323
prototyping, 13, 264
server GUI, 286
testing usability, 233
user interface concepts, 228
GUIControllerException
Denny's DVDs, 272

H

handleException method
ApplicationRunner class, 263
Handler object
logging messages to a file, 52
hashCode methods
generating, 125
overriding equals method, 125
heartbeat functionality, RMI, 297
hibernation
threads, 78
holdsLock method, Thread class, 116
hooks
shutdown hook, 288
HTML markup tags
Javadoc comments, 37
HTTP (Hypertext Transfer Protocol)
Java RMI and, 174
HTTP tunneling, 195

I

IceCreamMan example
multithreading, 83–86
IDL (Interface Definition Language), 172
Java-IDL transfer protocol, 174
if statements
best practices for threading, 115
source code indentation, 26
IIOP (Internet Inter-Orb Protocol)
choosing between RMI and sockets, 174, 297
Java RMI-IIOP transfer protocol, 174
RMI-IIOP, 172
using CORBA-related clients, 174
implementation comments, 28, 85
import statements
Java coding conventions, 23
indentation
source code formatting, 25
if, for, and while statements, 26
third or fourth level of indentation, 26
Inet4Address class, java.net, 201
Inet6Address class, java.net, 201
InetAddress class, java.net, 200
info logging level, 51
instance methods
static methods compared, 89
instance variable comments
Javadoc comments, 37
instructions
caution: online instructions, 18
including in submission, 66
integers
synchronization, 115
interface declarations
Java coding conventions, 24

interface naming conventions
Java coding conventions, 21
interfaces
see also DBClient interface
adding another method to interface, 67
including more exceptions in interface, 66
Javadoc tags, 38
RMI, 175–177
internally synchronized classes, 111
best practices for threading, 115
InterruptedException
handling unlisted exceptions, 127
threads, 110
wrapping within allowed exception, 130
InterruptedExceptionExample, 126
InvalidClassException
deserialization of class instance, 120
invokeAndWait method, SwingUtilities class
multithreading with Swing, 114
invokeLater method, SwingUtilities class
multithreading with Swing, 114
IOException
DvdDatabase class constructor, 136
wrapping within allowed exception, 130
IP (Internet Protocol)
networking with sockets, 199
IP addresses, 200
IPv4/IPv6, 201
isCellEditable method
TableModel interface, 257
isSelected method, 249
iterative server
lifecycle, 213

J

J2SE 5
code name Tiger, 57
purpose of, 3
setting up J2SE 5 JDK, 8
Swing changes in, 289–291
JAR files
creating, 305
creating manifest file, 303
Denny's DVDs project, 58
jar tool options, 305
manifest files and, 303
original purpose of, 58
packaging application as, 322
packaging Denny's DVDs application, 305
packaging submission, 318
Sun Certification project requirements, 58
working with packages, 45, 46
Java
evolution of programming, 201
multithreading, 73
online resources, 323
Java coding conventions, 19–35
file layout, 22–24
beginning comments, 23
class declarations, 24
import statements, 23
interface declarations, 24
package statements, 23
JDK 5 new features, 29–35
autoboxing, 32
enhanced for loop, 31–32
generic types, 29–31
static imports, 34–35
VarArgs, 32–34
naming conventions, 20–22
source code formatting, 24–28
comments, 28–29
indentation, 25
line lengths/wrapping, 25–27
spacing, 27
statement formatting, 27
variable declaration formatting, 28
variable naming, 20
Java-IDL transfer protocol
RMI transfer protocols, 174
Java look and feel
see look and feel
Java objects
locks, 87
Java RMI-IIOP transfer protocol, 174
Java sockets
caution: using serialized objects, 164
Java standards
coding conventions, 19–35
Javadoc usage, 35–44
packaging concepts, 44–47
Java utilities
working with packages, 45
java.nio (NIO) packages, 161
java.util.concurrent package, 161
java.util.logging
leaving logging code in submission, 55
Javadoc, 35–44
best practice, 48
coding conventions, 36–44
classes and methods, 36
command line options, 43
converting Unix-style into platform-specific pathnames, 43
Javadoc comments, 28, 35
caution: implementation specific details, 37
class and interface tags, 38
constructor and method tags, 40
DVD class constructor, 122
field tags, 39
HTML markup tags, 37
see (@see) link tags, 40
where to place in source code, 37

Javadoc tags, 38
JDK 5, changes in, 42
local documentation, 44
online resources, 323
package documentation, 41–42
plug ins, 36
running from command line, 42–44
Javadoc API
building on command line, 319
building using file to contain Javadoc options, 320
JavaRanch, 5
online resources, 323
JButton component
ActionListener interface, 248
example using, 249
illustration of, 251
Swing components, 248
JComboBox component
ActionListener interface, 251
example using, 252
illustration of, 253
Swing components, 251
JDK
setting up J2SE 5 JDK, 8
using current for SCJD exam, 5
JDK 5
coding conventions for new features, 29–35
autoboxing, 32
automatic type conversions, 32
enhanced for loop, 31–32
generic types, 29–31
static imports, 34–35
VarArgs, 32–34
Javadoc changes in, 42
locking in JDK 5, 96–98
new tags, 41
validating data, 142
JFormattedTextField subclass
MaskFormatter, 246
JFrame component
adding components, 238
combined with JPanel, 239, 241
JLabel component
example using, 245
illustration of, 245
Swing components, 244
join statements
daemon threads, 80
WAITING state, threads, 92
JPanel container, 239
combined with JFrame, 239, 241
JRadioButton component
ActionListener interface, 249
example using, 249
illustration of, 251
Swing components, 249
JRMP (Java Remote Method Protocol)
Java RMI and, 174
RMI-JRMP transfer protocol, 174
JScrollPane component
illustration of, 261
Swing components, 260
JTable component
GUI layout concepts, 227
methods, 259
sizing columns, 254
Swing components, 254
TableModel interface, 255–260
using in exam, 300
JTextArea component
Swing components, 275
JTextField component
example using, 245
illustration of, 245
JFormattedTextField subclass, 246
MVC pattern, 247
Swing components, 245, 247
validating contents of, 246
JUnit
testing project implementations, 15
JUnit code
including unit tests in submission, 55
JVM thread scheduler
testing, 317
threads resuming execution, 86

K

keystroke mnemonics
creating for GUI components, 292
keystroke shortcuts
GUI design, 230
keywords
spacing, 27

L

labels
JLabel component, 244
layout concepts, 227
layout managers
BorderLayout, 237
caution: placing components explicitly, 244
FlowLayout, 239
GridBagLayout, 275
GridLayout, 274
overview, 237
setConstraints method, 277
using, 291
lazy activation, 177
lazy loading of records, 161
li tag
Javadoc comments, 37
line lengths
source code formatting, 25–27

link (-link) option
Javadoc command line options, 44
link (@link) Javadoc tag, 39, 40
linkoffline (-linkoffline) option, Javadoc, 43, 44
linkplain (@linkplain) Javadoc tag, 39, 40
listeners
ActionListener interface, 248
literal tag, JDK 1.5, 41
local mode
running client application in local mode, 307
LockAttemptFailedException
handling unlisted exceptions, 127
wrapping within allowed exception, 130
LockInformation class, 159
locking
coverage summarized, 296
RMI implementation, 191
LockObjectNotMemberVariables class, 95
locks, 87–98
best practices for threading, 114
class providing locking, 148
clients locking records, 156
deadlocks, 98–100
handling client crashes, 157
identifying owner, 150–154
using class instance/thread/token, 151
Java objects, 87
locking class instances, 89–90
locking in JDK 5, 96–98
locking objects, 88
locking objects directly, 90–92
logical record locking, 150
member variables of locked objects, 95, 116
multiple notifications on lock release, 157
multithreading, 73
mutex, 127
nonimplicit locking, 95–96
notify method, 92–94
notifyAll method, 92–94
ReadWriteLocks, 142
releasing in finally block, 142
releasing locks in finally clause, 145
synchronization, 87
timeouts on locks, 155
violating locking objects, 88
log subdirectory
organizing projects, 17
logger class
utility methods for, 52
logging
best practice, 50
DVD class constructor, 123
leaving logging code in submission, 55
logging an exception, 128
logging messages to a file, 52
outputting log messages, 54
predefined logging levels, 51
reading log messages, 53
temp.log file, 53
testing Denny's DVDs application, 314
logical record locking, 150
identifying lock owner using threads/tokens, 151
look and feel
Java look and feel guidelines, 292
Napkin Look & Feel for Swing applications, 226
online resources, 323
Swing, 241
Swing changes in J2SE 5, 289
using a different look and feel, 292
using Macintosh look and feel, 292
LookAndFeel subclasses, 242
caution: platform support, 242
illustrations of, 243
loopback addresses, 206

M

MacLookAndFeel subclass, 242
using Macintosh look and feel, 292
Main-Class attribute, 304
MainWindow class
Denny's DVDs, 263, 268–273
using TableModel with JTable, 259
manifest files, 303
Manifest-Version attribute, 304
marshaling
serialization, 164
MaskFormatter
JFormattedTextField subclass, 246
Matcher class, 147
member variables
locking objects directly, 90
MetalLookAndFeel
illustration of Ocean theme, 243
illustration of Steel theme, 243
LookAndFeel subclasses, 242
platforms for, 242
method comments, Javadoc, 37
method naming conventions, 21
methodologies
considering what a method does, 134
following methodologies and standards, 54
methods
adding another method to interface, 67
caution: overloading with versions containing VarArgs, 33
Javadoc coding conventions, 36
Javadoc tags, 40
respecting synchronization, 87
static methods and instance methods, 89
Model component
MVC pattern, 235
Model-View-Controller pattern
see MVC pattern

modifyDVD method
DBClient interface, 60, 137
indicating success/failure, 145
MotifLookAndFeel
drop down menu, 244
illustration of, 243
LookAndFeel subclasses, 242
multicast servers, 212
MulticastSocket class, java.net, 212
multitasking
socket servers, 212
multithreading, 72–87
best practices for threading, 114, 115
blocking, 78
Child.java example, 79–82
competing thread's behavior, 74
distinct types of pausing execution, 74
IceCreamMan.java example, 83–86
implementing multithreaded socket servers, 222
locking objects, 73, 88
multithreaded socket server lifecycle, 214
multithreading with Swing, 113–114
real life analogies, 73
sleeping, 78
synchronization, 73
testing Denny's DVDs application, 310
waiting, 74–77
yielding, 77–78
mutex, 127
requesting read lock on, 143
MVC (Model-View-Controller) pattern, 234–237
alternatives to, 237
anonymous inner classes, 249
benefits of, 236
Controller component, 235
design decisions, 301
drawbacks of, 236
JTextField component, 247
Model component, 235
reasons for using, 234
restaurant analogy, 235
TableModel interface, 255
View component, 235

N

Naming (java.rmi.Naming)
storing remote object references, 194
naming conventions
class naming, 21
constant naming, 22
getDVDs method, 141
interface naming, 21
Java coding conventions, 20–22
method naming, 21
package naming, 20
Sun Coding Conventions, 21
variable naming, 21
Napkin Look & Feel for Swing applications, 226
nesting locks
best practices for threading, 114
network latency
testing, 317
network protocols
evolution of, 201
network transparency
RMI (Remote Method Invocation), 172
networking
benefits of RMI solution, 299
choosing between RMI and sockets, 297
configuring network server, 307
Denny's DVDs system overview, 64, 65
RMI, 171–195
running client application in networked mode, 309
running network server, 308
sockets, 199–223
transport protocol, 173
working with packages, 46
NIO packages
using, 161
using in certification exam, 58
nonimplicit locking, 95–96
nonrepeatable reads
Denny's DVDs overview, 66
notify method
best practices for threading, 116
calling, 84
waiting threads, 76
notify method, Thread class, 92–94
notifyAll method compared, 92
wait method and, 77
notifyAll method, Thread class, 92–94
best practices for threading, 116
caution: order in which threads are notified, 94
notify method compared, 92
wait method and, 77
waiting threads, 76
notifyObservers method
notifications of changes in GUI, 283
NotifyVersusNotifyAll class, 92
NotSerializableException, 194
NullPointerException
comparing DVD class instances, 125
wrapping exceptions within RuntimeException, 132
nulls
checking for nulls before recasting objects, 285

O

object synchronization, 111–112
ObjectInputStream class, 166
ObjectOutputStream class, 166, 169

objects
multiple notifications on lock release, 157
Value Object design pattern, 119
Observable class, 281
notifyObservers method, 283
Observer classes, 281
Observer design pattern, 281–283
ol tag
Javadoc comments, 37
online instructions
caution: online instructions, 18
OptionUpdate value object, 285
Value Object design pattern, 282
overriding
making methods final, 140

P

p tag
Javadoc comments, 37
package documentation, Javadoc, 41–42
package naming conventions
Java coding conventions, 20
package statements
Java coding conventions, 23
packages
caution: classes with same name, 46
working with, 44–47
packaging
application as JAR file, 322
packaging submission, 318–321
packets, 202
param (@param) Javadoc tag, 40
parameter passing
RMI method invocations, 193
parenthesis
spacing, 27
pass by value, 194
passive object, 177
pathnames
converting Unix-style into platform-specific, 43
paths of execution
see threads
patterns
see design pattern
PatternSyntaxException
Denny's DVDs, 272
pausing, threads, 86
performance
choosing between RMI and sockets, 297
sockets, 297
using a profiler, 140
using cache in Sun SCJD assignment, 161
Value Object design pattern, 120
permissions
RMI, 196
submissions, 321
persistDVD method
addDVD method calling, 144
demonstrating FileOutputStream, 167
releasing locks in finally clause, 145
serialization process, 166
StringBuilder, 146
platforms
testing submission, 322
Portland Pattern Repository
online resources, 323
ports
allowing user to change, 223
port numbers to avoid, 223
running network server, 308
pre tag
Javadoc comments, 37
primitive types, RMI, 194
private methods
overriding methods, 140
profilers
using to improve performance, 140
programming
evolution of (1GL to, 4GL), 201
projects
see also Denny's DVDs project
documentation required, 17–19
gathering requirements, 12–13
getting started, 11, 12
implementing, 11–16
implementing projects
design patterns, 14
documenting design decisions, 15
testing, 15
organizing, 16–17
Sun Certification project requirements, 57–66
working with packages, 44–47
protocols
choosing between RMI and sockets, 174
client/server interaction, 218
transfer protocol, 174
transport protocol, 173
Proxy design pattern, 193

Q

QuitApplication class
Denny's DVDs, 270

R

race conditions
thread safety and, 100–102
radio buttons
JRadioButton component, 249
RandomAccessFile
DvdFileAccess class, 138, 139
readExternal method, 169, 170
readFully method, 144

readObject method
ObjectInputStream class, 166, 169
method signature for, 169
ReadWriteLocks, 142
performance using, 144
rebind method, Registry class
benefits of RMI solution, 300
RecordFieldReader class, 144
RecordFieldWriter class, 146
RecordNotFoundException
wrapping within allowed exception, 131
recordNumbers collection
DvdFileAccess class, 138
recordNumbers map, 142
threads, 143
reentrant locks
multiple notifications on lock release, 158
RegDvdDatabase class
register method, 180
RMI implementation, 180
register method
RegDvdDatabase class, 180
registration
SCJD assignment and examination, 5
registry
createRegistry method, 180
starting registry programmatically, 180
releaseDVD method, ReservationsManager class
creating logical release method, 155–160
DBClient interface, 60, 137
using Condition from LockInformation class, 159
Remote interface, RMI, 176
RemoteException, 176
RMI implementation, 179
Remote Method Invocation
see RMI
remote objects
access to, 177
choosing between RMI and sockets, 174
exporting, 177
RMI, 194
Remote Procedure Call (RPC), 172
RemoteException
Remote interface, RMI, 176
RemoteObject class, java.rmi, 176
RemoteServer class, java.rmi, 177
removeDVD method
DBClient interface, 60, 137
indicating success/failure, 145
removeTableModelListener method
TableModel interface, 255
rentDVD method
testing Denny's DVDs application, 312
replace method, StringBuilder, 146
requirements
gathering requirements, 12–13
prototyping GUI, 13
ReservationsManager class, 148–160
creating logical release method, 155–160
creating logical reserve methods, 154–155
reserveDVD method, ReservationsManager class
creating logical reserve methods, 154, 155
DBClient interface, 60, 137
deadlock handling, 156
less CPU intensive version, 158
Result object
socket servers, 220–221
resume method, Thread class, 106
retrieveDVD method, 143
demonstrating FileInputStream, 167
serialization process, 166
return (@return) Javadoc tag, 40
returnDVD method
testing Denny's DVDs application, 313
RMI (Remote Method Invocation), 171–196
Activatable class, 177
choosing between RMI and sockets, 174–175, 197, 200, 296–300
benefits of RMI solution, 299
class loader requirements, 195
classes, 175
client definition in RMI scenarios, 172
delivery stack, 173–174
distributed object system definition in, 172
exporting remote objects, 177
firewall issues, 195
heartbeat functionality, 297
interfaces, 175–177
introduction, 171
network transparency, 172
online resources, 323
passing parameters, 194
permissions, 196
primitive types, 194
Remote interface, 176
remote objects, 194
RemoteObject class, 176
RemoteServer class, 177
requirement for skeletons, 197
RMI layers, 176
RMI method invocations, 193
RMI transfer protocols, 174
RPC/CORBA/SOAP and, 172
running rmic on remote package, 304
security and dynamic loading, 195
Serializable objects, 194
server definition in RMI scenarios, 172
stubs and skeletons, 192–193
thread reuse in RMI, 151
thread safety, 196

transferring parameters, 193
UnicastRemoteObject class, 177
RMI factory, 177–196
Factory pattern, 178
test program examples, 183
RMI implementation, 179–192
abstraction, 191
DvdDatabaseFactory.java, 179
Factory pattern, 179
locking strategy, 191
RMI registry
benefits of RMI solution, 300
manual starting of, 197
obtaining remote object references, 194
RMI-IIOP, 172
Java RMI-IIOP transfer protocol, 174
RMI-JRMP
RMI transfer protocols, 174
rmic tool
running rmic on remote package, 304
RMIClassLoader class
dynamic loading, 195
RmiFactoryExample class
main method, 182
RMI implementation, 181, 183
running test program, 182
RmiNoFactoryExample class
main method, 181
RMI implementation, 181, 182
running test program, 182
RPC (Remote Procedure Call), 172
run method
best practices for threading, 115
RmiFactoryExample class, 182
RmiNoFactoryExample class, 181
testing Denny's DVDs application, 311
Runnable interface
creating threads, 71, 79
RUNNABLE state
Thread objects, 107
RuntimeException
caution: throwing but not catching, 134
wrapping an exception within, 131
wrapping an exception within subclass of, 132

S

scalability
choosing between RMI and sockets, 297
SCJD (Sun Certified Java Developer) exam, 4–5
see also submissions
assignment, 4
caution: grading of assignment, 4
cost of, 9
creating data file, 160
discussion groups, 323
documentation and questions, 5
downloading assignment, 5
loss of exam, 9
online resources, 323
packages allowed, 161
purpose, 5
registering for assignment and examination, 5
resources, 5
SCJD certification process, 4–5
submitting test classes and build scripts, 160
Sun's expectations, 5
using caching, 161
using current JDK 5
written test, 4
SCJP (Sun Certified Java Programmer) exam, 8
scrolling
JScrollPane component, 260
SearchDVD class
Denny's DVDs, 272
security
requirements for, 197
RMI, 195
SecurityManager class
dynamic loading, RMI, 195
see (@see) Javadoc tag, 38, 39, 40
examples of, 40
serial (@serial) Javadoc tag, 39
Serializable class/interface, java.io
declaring serialVersionUID, 121
Externalizable subinterface, 169
comparing interfaces, 171
inspecting classes for, 166
purpose of implementing, 165
Serializable objects, 194
using serialver tool, 165
serialization, 164–171
creating byte stream from object graph, 165
customizing with Externalizable interface, 169–171
deserialization of class instance, 120
interface required, 165
marshaling, 164
NotSerializableException, 194
object copy not reference, 165
reading and writing serializable objects, 166
serialization mechanism, 166–169
serializing object state, 168
transient keyword, 168
using serialver tool, 165–166
serialized objects
choosing between RMI and sockets, 297
benefits of RMI solution, 299
serialver tool, 165–166
serialVersionUID
deserialization of class instance, 120
mandatory modifiers, 121
not declared in Serializable class, 121
serialization process, 168

server factory
handling client crashes, 157
server GUI
Denny's DVDs, 286
servers
class loader requirements, 195
configuring network server, 307
definition in RMI scenarios, 172
iterative server, 212
running Denny's DVD server, 307
running network server, 308
single-threaded server, 212
socket servers, 212–221
utilizing thread pool, 222
transferring data with client, 120
ServerSocket class, java.net, 213–218
accept method, 214, 215, 217
description, 200
ServerWindow constructor
server GUI, 286
setActionCommand method, 249
setComposer setter
DVD class constructor, 123
setConstraints method
layout managers, 277
setLocation method
caution: placing components explicitly, 244
setMnemonic method
creating keystroke mnemonics, 292
setReceiveBufferSize method, Socket class, 205
setSendBufferSize method, Socket class, 205
setSize method
caution: placing components explicitly, 244
setSoTimeout method, Socket class, 204
setters, 123
automatic generation, 124
setupTable method
using TableModel with JTable, 259
setValueAt method
TableModel interface, 257
severe logging level, 51
shutdown hook
server GUI, 288
since (@since) Javadoc tag, 38, 39, 40
single line comments, 28
singleton classes, 137
skeletons
requirement for, 197
skeleton interface, 193
skins
Swing changes in J2SE 5, 290
sleep method, Thread class
going into wait state, 74
monitoring for objects queued for processing, 83
threads resuming execution, 87
sleeping, 78
yielding compared, 79
SOAP (Simple Object Access Protocol)
RMI and, 172
Socket class, java.net
constructors, 203
description, 200
getting SO_BINDADDR, 205
setting and getting SO_RCVBUF, 205
setting and getting SO_SNDBUF, 205
setting and getting SO_TIMEOUT, 204
SocketOptions interface, 204
socket network solution
handling client crashes, 157
SocketCommand class
indicating command to perform, 223
SocketCommand enum, 221
SocketOptions interface
accessing via Socket class, 204
SO_BINDADDR option, 205
SO_RCVBUF option, 205
SO_SNDBUF option, 204
SO_TIMEOUT option, 204
sockets
automatically updating clients, 223
benefits of RMI solution, 299
benefits of Serialized Objects, 297
choosing between RMI and sockets, 197, 200, 296–300
choosing for exam, 222
description, 199
DVDSocketClient class, 205–212
exam restrictions, 202
IP addresses, 200
networking with, 199–223
overview, 199–200
performance, 297
reasons for using, 200
socket addresses, 200
socket servers, 212–221
application protocol, 200, 218–221
Command enum, 218–220
description, 199
iterative socket server lifecycle, 213
listening to incoming connections, 213
multicast servers, 212
multitasking, 212
multithreaded required or not, 222
multithreaded socket server lifecycle, 214
Result object, 220–221
ServerSocket class, 213–218
unicast servers, 212
using enum constants, 220
TCP socket clients, 203–205
TCP sockets, 202–203
technologies requiring, 222
UDP sockets, 201–202
SocketTimeoutException, 204, 215
source (-source) option, Javadoc, 43, 50

source code
caution: making it clear, 25
placing Javadoc comments, 37
source code formatting
comments, 28–29
indentation, 25
if, for, and while statements, 26
third or fourth level of indentation, 26
Java coding conventions, 24–28
line lengths, 25–27
spacing, 27
statements, 27
variable declaration, 28
wrapping, 25–27
sourcepath (-sourcepath) option, Javadoc, 43
SO_BINDADDR option, 205
SO_RCVBUF option, 205
SO_SNDBUF option, 204
SO_TIMEOUT option, 204
spacing
source code formatting, 27
src directory
organizing projects, 17
Sun Certification project requirements, 58
standards
following methodologies and, 54
starvation
thread safety and, 102–104
state
Thread objects, 107
statements
source code formatting, 27
static imports
coding conventions for JDK 5, 34–35
static keyword, 90
static methods, 89
static objects, 79
static variables
locking class instances, 89
stop method, Thread class, 106
string format options
testing Denny's DVDs application, 316
StringBuffer, 146
StringBuilder
DvdFileAccess class, 139
persistDVD method, 146
replace method, 146
strings
synchronization, 115
strong typing
choosing between RMI and sockets, 174
stubs, 192
caution: dynamic downloading of stubs, 304
choosing between RMI and sockets, 174
dynamic stub generation, 175
dynamically generated stub classes, 192
running rmic on remote package, 304
submissions
including extra files in, 18
packaging, 318–321
permissions, 321
platform for testing, 322
what to include with, 322
Sun assignments
caution: variation in instructions, 17
Sun Certification project
requirements, 57–66
Sun Certified Java Developer exam
see SCJD exam
Sun Code Conventions for Java, 20
Sun Coding Conventions
class or interface declarations, 24
constants, 24
file layout, 22
import statements, 23
indentation, 25
inserting blank lines between methods, 81
JDK 5 updates to, 29
naming conventions, 21
packages, 23
tools confirming code conforms to, 27
variable declaration, 28
Sun Java utilities, 45
suspend method, Thread class, 106
swallowing an exception, 128
Swing
AWT and Swing, 237–261, 291
best practices for threading, 115
BorderFactory class, 251–254
changes in J2SE 5, 289–291
components, 290
look and feel, 289
skins, 290
described, 64
layout managers, 237
look and feel, 241
multithreading with Swing, 113–114
best practices for threading, 114
online resources, 323
online tutorials, 237
Swing components
JButton, 248
JComboBox, 251
JLabel, 244
JRadioButton, 249
JScrollPane, 260
JTable, 227, 254
JTextArea, 275
JTextField, 245–247
layout managers available, 291
SwingWorker class
caution: not using directly in assignment, 114

synchronization
best practices for threading, 115
client synchronization, 112–113
description, 87
IceCreamMan.java example, 84, 85
integers and strings, 115
internally synchronized classes, 111, 115
locking member variables of locked objects, 98
locking objects directly, 90, 91
meaning of synchronized object, 117
multithreading, 73
object synchronization, 111–112
performance, 117
synchronization on class itself, 127
synchronized and unsynchronized methods, 89
synchronized object definition, 111
synchronizing data, 117
synchronizing methods, 90
violating locking objects, 88
synchronized blocks
accessing data files, 144
best practices for threading, 115
calling unsynchronized block, 116
thread safety, 104
synchronized keyword
locking class instances, 89
SynthLookAndFeel
changes in J2SE 5, 290
system tests, 16

T

TableModel interface, 255–260
AbstractTableModel class, 255
getColumnName method, 257
getRowCount method, 257
getValueAt method, 257
isCellEditable method, 257
setValueAt method, 257
using with JTable component, 259–260
tables
JTable component, 254
TCP (Transmission Control Protocol), 202
TCP sockets, 202–203
connection-oriented protocol, 222
DVDSocketClient class, 205–212
TCP socket clients, 203–205
temp.log file
logging messages, 53
TERMINATED state
Thread objects, 107
ternary operators
readability of, 124
testing
Denny's DVDs application, 309–317
GUI usability, 233
implementing projects, 15
JVM thread scheduler, 317
network latency, 317
providing test classes in submission, 160
selecting testers, 14, 16
system tests, 16
unit tests, 16
text fields
JTextField component, 245
thick client, 149
thin client, 149
handling client crashes, 157
this object
synchronizing methods, 90
Thread class/objects
BLOCKED state, 107
blocking, 108–110
creating a thread of execution, 71
deprecated methods, 106
RMI implementation, 181
RUNNABLE state, 107
states, 107
TERMINATED state, 107
TIMED_WAITING state, 107
WAITING state, 107
thread safety, 98–106
atomic operations, 104–106
benefits of RMI solution, 299
coverage summarized, 296
deadlocks, 98–100
race conditions affecting, 100–102
RMI, 192, 196
starvation, 102–104
synchronized blocks, 104
Vector objects, 111
threads/threading, 71–117
best practices for threading, 114–116
BLOCKED and RUNNABLE states, 78
blocking, 78
caution: not relying on thread priorities, 78
caution: order in which threads are notified, 94
caution: state of threads paused for I/O, 110
changing daemon status of threads, 79
creating a thread of execution, 71
daemon threads, 79
deadlock handling, 156
Denny's DVDs system overview, 66
exiting before thread complete, 117
hibernation, 78
identifying lock owner using, 151
InterruptedException, 110
joining threads, 72, 79
locking member variables of locked objects, 95
locks, 87–98
multiple notifications on lock release, 157
multithreaded socket server lifecycle, 214
multithreading, 72–87
see also multithreading
pausing, 86

reading from a data file, 143
ReadWriteLocks, 142
recordNumbers map, 142
restarting, 117
simple example illustrating concept, 72
single-threaded server, 212
sleeping, 78
testing Denny's DVDs application, 310
thread priorities, 102, 104
best practices for threading, 115
thread reuse in RMI, 151
thread scheduler, 77
threads resuming execution, 86
using sockets, 200
utilizing thread pool, 222
violating locking objects, 88
waiting, 74
WAITING state, 92
yielding, 77
Tiger, J2SE 5, 57
TIMED_WAITING state
Thread objects, 107
tmp subdirectory
organizing projects, 17
tokens
identifying lock owner using, 151
Transfer Object pattern, 119
transfer protocols, 164, 174
RMI transfer protocols, 174
transient keyword
serialization, 168
transport protocol, 173
tryLock method, Lock class
locking in JDK 5, 97
tst subdirectory
organizing projects, 17
tunneling
HTTP tunneling, 195

U

UDP (User Datagram Protocol), 201
UDP sockets, 201–202
advantages/disadvantages, 202
exam restrictions, 202
ul tag
Javadoc comments, 37
UML use case diagram
Denny's DVDs, 59
unicast servers, 212
UnicastRemoteObject class, java.rmi.server, 177
export method, 192
RMI implementation, 179
unit tests, 16
including unit tests in submission, 55
Universal Product Code
see UPC
unmarshaling
serialization, 164
unreferenced method
handling client crashes, 157
UPC (Universal Product Code) number, 59, 121
comparing DVD class instances, 124, 125
reading or writing records, 140
recordNumbers map, 142
updating
automatically updating clients, 223
use (-use) option
Javadoc command line options, 43
use cases, 229
user documentation
best practice, 48
user instructions
caution: online instructions, 18
documentation required, 17, 18
User Interface
see GUI

V

validation
validating contents of JTextField, 246
zip codes, 246
value (@value) Javadoc field tag, 39
value object
Data class using, 161
Value Object design pattern, 119
OptionUpdate value object, 282
VarArgs (variable argument lists)
caution: overloading with versions containing, 33
coding conventions for JDK 5, 32–34
variable declaration
source code formatting, 28
variable naming conventions, Java, 20, 21
Vector objects
object synchronization, 111
thread safety, 111
vectors
synchronized data, 117
version (-version) option, Javadoc, 43
version (@version) Javadoc tag, 38
version.txt file
documentation required, 17
View component
MVC pattern, 235

W

wait method, Thread class
calling wait method on an object, 81
going into wait state, 75
notify and notifyAll methods and, 77
sleep or yield compared, 77
threads resuming execution, 86

WAITING state, threads, 92
waits occurring in loops, 82
waiting, 74–77
children buying ice cream analogy, 74
code example, 75
distinct types of pausing execution, 74
WAITING state, threads, 92, 107
warning logging level, 51
Waterfall Model, 12
WeakHashMap
handling client crashes, 157
while statements
best practices for threading, 115, 116
source code indentation, 26
Window containers
layout managers, 237
windows
centering a window, 292
WindowsLookAndFeel
code setting, 242
illustration of, 243
LookAndFeel subclasses, 242
platforms for, 242
windowtitle (-windowtitle) option, Javadoc, 43
wrapping
source code formatting, 25–27
writeExternal method
customizing serialization, 169, 170
writeObject method
ObjectOutputStream class, 166, 169
written test
SCJD exam, 4

X

XDoclet, Javadoc plug in, 36

Y

yield method, Thread class
going into wait state, 74
yielding, 77–78
best practices for threading, 114
reasons for, 77
sleeping compared, 79
when to yield, 84

Z

zip codes
validation, 246
zip files, 302
ZipTextField class
validating zip codes, 246